

---

# Amazon Simple Queue Service

## Developer Guide

API Version 2012-11-05



## Amazon Simple Queue Service: Developer Guide

Copyright © 2015 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

The following are trademarks of Amazon Web Services, Inc.: Amazon, Amazon Web Services Design, AWS, Amazon CloudFront, AWS CloudTrail, AWS CodeDeploy, Amazon Cognito, Amazon DevPay, DynamoDB, ElastiCache, Amazon EC2, Amazon Elastic Compute Cloud, Amazon Glacier, Amazon Kinesis, Kindle, Kindle Fire, AWS Marketplace Design, Mechanical Turk, Amazon Redshift, Amazon Route 53, Amazon S3, Amazon VPC, and Amazon WorkDocs. In addition, Amazon.com graphics, logos, page headers, button icons, scripts, and service names are trademarks, or trade dress of Amazon in the U.S. and/or other countries. Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon.

All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

---

## Table of Contents

What is Amazon Simple Queue Service? .....	1
Architectural Overview .....	1
Amazon SQS Features .....	2
How Amazon SQS Queues Work .....	3
Properties of Distributed Queues .....	4
Message Order .....	4
At-Least-Once Delivery .....	4
Message Sample .....	4
Queue and Message Identifiers .....	5
Queue URLs .....	5
Message IDs .....	5
Receipt Handles .....	6
Resources Required to Process Messages .....	6
Visibility Timeout .....	6
General Recommendations for Visibility Timeout .....	7
Extending a Message's Visibility Timeout .....	7
Terminating a Message's Visibility Timeout .....	8
API Actions Related to Visibility Timeout .....	8
Message Lifecycle .....	8
Amazon SQS and JMS .....	10
Amazon SQS and JMS Prerequisites .....	10
Getting Started with the Amazon SQS Java Messaging Library .....	11
Creating a JMS Connection .....	11
Creating an Amazon SQS Queue .....	12
Sending Messages Synchronously .....	12
Receiving Messages Synchronously .....	13
Receiving Messages Asynchronously .....	13
Client Acknowledge Mode .....	14
Unordered Acknowledge Mode .....	15
Code Samples .....	16
ExampleConfiguration.java .....	16
TextMessageSender.java .....	18
AsyncMessageReceiver.java .....	19
SyncMessageReceiver.java .....	20
SyncMessageReceiverClientAcknowledge.java .....	21
SyncMessageReceiverUnorderedAcknowledge.java .....	24
SpringExampleConfig.xml .....	26
SpringExample.java .....	27
ExampleCommon.java .....	29
Reference/Appendix .....	30
Supported JMS 1.1 Common Interfaces: .....	30
Supported Message Types .....	30
Supported Message Acknowledgment Modes .....	30
JMS Defined Headers and Reserved Properties .....	30
Message Attributes .....	32
Message Attribute Items and Validation .....	32
Data Types .....	33
Using Message Attributes with the AWS Management Console .....	33
Using Message Attributes with the AWS SDKs .....	36
Using Message Attributes with the Amazon SQS Query API .....	38
MD5 Message-Digest Calculation .....	39
Long Polling .....	41
Enabling Long Polling with the AWS Management Console .....	42
Enabling Long Polling with the Query API .....	44
Delay Queues .....	46

Creating Delay Queues with the AWS Management Console .....	47
Creating Delay Queues with the Query API .....	49
Message Timers .....	50
Creating Message Timers Using the Console .....	50
Creating Message Timers Using the Query API .....	52
Batch API Actions .....	54
Maximum Message Size for SendMessageBatch .....	54
Making API Requests .....	55
Endpoints .....	56
Making Query Requests .....	57
Structure of a GET Request .....	57
Structure of a POST Request .....	58
Related Topics .....	59
Making SOAP Requests .....	59
Request Authentication .....	60
What Is Authentication? .....	60
Your AWS Account .....	62
Your Access Keys .....	62
HMAC-SHA Signatures .....	62
Query Request Authentication .....	67
Responses .....	68
Structure of a Successful Response .....	68
Structure of an Error Response .....	68
Related Topics .....	69
Shared Queues .....	70
Simple API for Shared Queues .....	70
Advanced API for Shared Queues .....	70
Understanding Permissions .....	70
Granting Anonymous Access to a Queue .....	71
Programming Languages .....	72
Dead Letter Queues .....	73
Setting up Dead Letter Queue with the AWS Management Console .....	73
Using Dead Letter Queue with the Amazon SQS API .....	75
Using The Access Policy Language .....	77
Overview .....	78
When to Use Access Control .....	78
Key Concepts .....	78
Architectural Overview .....	81
Using the Access Policy Language .....	83
Evaluation Logic .....	84
Basic Use Cases for Access Control .....	88
Amazon SQS Policy Examples .....	91
Special Information for Amazon SQS Policies .....	95
Access Control .....	96
IAM-Related Features of Amazon SQS Policies .....	96
IAM and Amazon SQS Policies Together .....	98
Amazon SQS ARNs .....	100
Amazon SQS Actions .....	101
Amazon SQS Keys .....	101
Example IAM Policies for Amazon SQS .....	102
Using Temporary Security Credentials .....	103
Monitoring Amazon SQS with CloudWatch .....	106
Access CloudWatch Metrics for Amazon SQS .....	106
Set CloudWatch Alarms for Amazon SQS Metrics .....	107
Amazon SQS Metrics .....	107
Logging Amazon SQS API Calls By Using CloudTrail .....	109
Amazon SQS Information in CloudTrail .....	109
Understanding Amazon SQS Log File Entries .....	110

AddPermission .....	110
CreateQueue .....	111
DeleteQueue .....	111
RemovePermission .....	112
SetQueueAttributes .....	112
Appendix A: Increasing Throughput with Horizontal Scaling and Batching .....	114
Horizontal Scaling .....	114
Batching .....	115
Example .....	116
Running the Example .....	117
Monitoring Volume Metrics from Example Run .....	119
Appendix B: Client-Side Buffering and Request Batching .....	120
Getting Started with AmazonSQSBufferedAsyncClient .....	120
Advanced Configuration .....	121
Appendix C: Subscribe Queue to Amazon SNS Topic .....	123
Subscribe Queue to Amazon SNS Topic with the AWS Management Console .....	123
Resources .....	125
Document History .....	126

# What is Amazon Simple Queue Service?

---

Amazon Simple Queue Service (Amazon SQS) offers reliable and scalable hosted queues for storing messages as they travel between computers. By using Amazon SQS, you can move data between distributed components of your applications that perform different tasks without losing messages or requiring each component to be always available.

Amazon SQS is a distributed queue system that enables web service applications to quickly and reliably queue messages that one component in the application generates to be consumed by another component. A queue is a temporary repository for messages that are awaiting processing.

Using Amazon SQS, you can decouple the components of an application so they run independently, with Amazon SQS easing message management between components. Any component of a distributed application can store messages in a fail-safe queue. Messages can contain up to 256 KB of text in any format. Any component can later retrieve the messages programmatically using the Amazon SQS API.

The queue acts as a buffer between the component producing and saving data, and the component receiving the data for processing. This means the queue resolves issues that arise if the producer is producing work faster than the consumer can process it, or if the producer or consumer are only intermittently connected to the network.

Amazon SQS ensures delivery of each message at least once, and supports multiple readers and writers interacting with the same queue. A single queue can be used simultaneously by many distributed application components, with no need for those components to coordinate with each other to share the queue.

Amazon SQS is engineered to always be available and deliver messages. One of the resulting tradeoffs is that SQS does not guarantee first in, first out delivery of messages. For many distributed applications, each message can stand on its own, and as long as all messages are delivered, the order is not important. If your system requires that order be preserved, you can place sequencing information in each message, so that you can reorder the messages when the queue returns them.

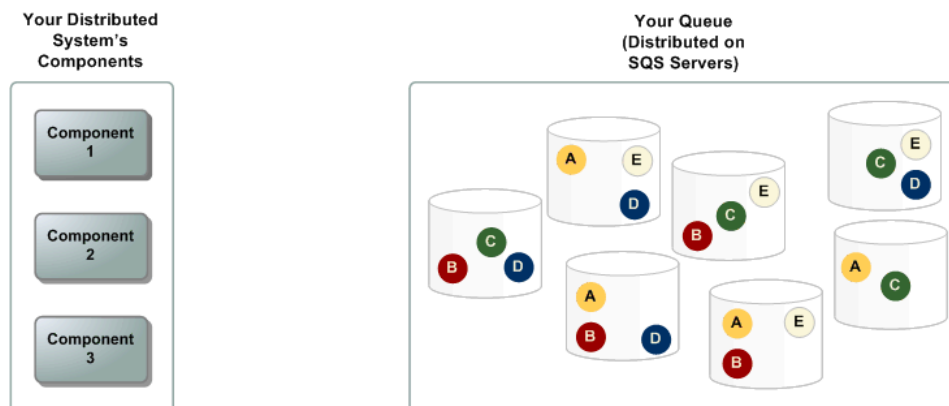
Be sure to read about distributed queues, which will help you understand how to design an application that works correctly with Amazon SQS. For more information, see [Properties of Distributed Queues \(p. 4\)](#).

## Architectural Overview

There are three main actors in the overall system:

- The components of your distributed system
- Queues
- Messages in the queues

In the following diagram, your system has several components that send messages to the queue and receive messages from the queue. The diagram shows that a single queue, which has its messages (labeled A-E), is redundantly saved across multiple Amazon SQS servers.



## Amazon SQS Features

Amazon SQS provides the following major features:

- **Redundant infrastructure** – Guarantees delivery of your messages at least once, highly concurrent access to messages, and high availability for sending and retrieving messages
- **Multiple writers and readers** – Multiple parts of your system can send or receive messages at the same time  
Amazon SQS locks the message during processing, keeping other parts of your system from processing the message simultaneously.
- **Configurable settings per queue** – All of your queues don't have to be exactly alike  
For example, one queue can be optimized for messages that require a longer processing time than others.
- **Variable message size** – Your messages can be up to 262,144 bytes (256 KB) in size  
For even larger messages, you can store the contents of the message using the Amazon Simple Storage Service (Amazon S3) or Amazon SimpleDB, and use Amazon SQS to hold a pointer to the Amazon S3 or Amazon SimpleDB object. Alternately, you can split the larger message into smaller ones.  
For more information about the services, go to the [Amazon S3 detail page](#) and the [Amazon SimpleDB detail page](#).
- **Access control** – You can control who can send messages to a queue, and who can receive messages from a queue
- **Delay Queues** – A delay queue is one which the user sets a default delay on a queue such that delivery of all messages enqueued will be postponed for that duration of time. You can set the delay value when you create a queue with `CreateQueue`, and you can update the value with `SetQueueAttributes`. If you update the value, the new value affects only messages enqueued after the update.

---

# How Amazon SQS Queues Work

---

## Topics

- [Properties of Distributed Queues \(p. 4\)](#)
- [Queue and Message Identifiers \(p. 5\)](#)
- [Resources Required to Process Messages \(p. 6\)](#)
- [Visibility Timeout \(p. 6\)](#)
- [Message Lifecycle \(p. 8\)](#)

This section describes the basic properties of Amazon SQS queues, identifiers for queues and messages, how you determine the general size of the queue, and how you manage the messages in a queue.

A queue can be empty if you haven't sent any messages to it or if you have deleted all the messages from it.

You must assign a name to each of your queues (for more information, see [Queue URLs \(p. 5\)](#)). You can get a list of all your queues or a subset of your queues that share the same initial characters in their names (for example, you could get a list of all your queues whose names start with "T3").

You can delete a queue at any time, whether it is empty or not. Be aware, however, that queues retain messages for a set period of time. By default, a queue retains messages for four days. However, you can configure a queue to retain messages for up to 14 days after the message has been sent.

Amazon SQS can delete your queue without notification if one of the following actions hasn't been performed on it for 30 consecutive days: `SendMessage`, `ReceiveMessage`, `DeleteMessage`, `GetQueueAttributes`, `SetQueueAttributes`, `AddPermission`, and `RemovePermission`.

### Important

It is a violation of the intended use of Amazon SQS if you repeatedly create queues and then leave them inactive, or if you store excessive amounts of data in your queue.

The following table lists the API actions to use.

To do this...	Use this action
Create a queue	<a href="#">CreateQueue</a>
Get the URL of an existing queue	<a href="#">GetQueueUrl</a>
List your queues	<a href="#">ListQueues</a>



To do this...	Use this action
Delete a queue	<a href="#">DeleteQueue</a>

## Properties of Distributed Queues

The following information can help you design your application to work with Amazon SQS correctly.

### Message Order

Amazon SQS makes a best effort to preserve order in messages, but due to the distributed nature of the queue, we cannot guarantee you will receive messages in the exact order you sent them. If your system requires that order be preserved, we recommend you place sequencing information in each message so you can reorder the messages upon receipt.

### At-Least-Once Delivery

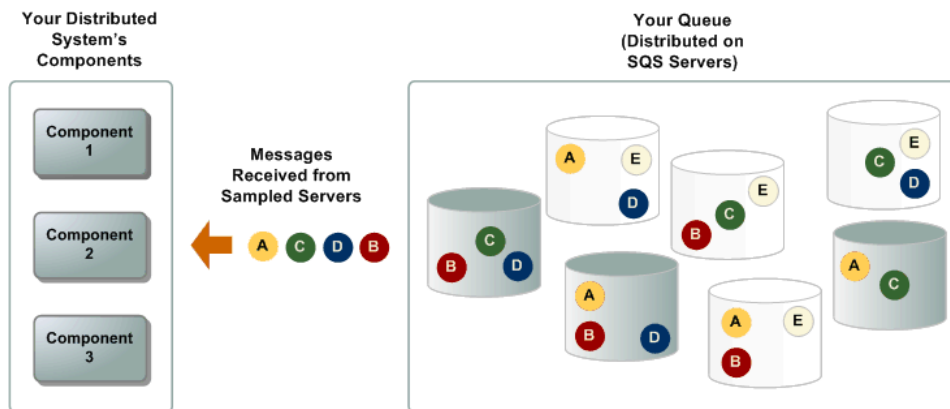
Amazon SQS stores copies of your messages on multiple servers for redundancy and high availability. On rare occasions, one of the servers storing a copy of a message might be unavailable when you receive or delete the message. If that occurs, the copy of the message will not be deleted on that unavailable server, and you might get that message copy again when you receive messages. Because of this, you must design your application to be idempotent (i.e., it must not be adversely affected if it processes the same message more than once).

### Message Sample

The behavior of retrieving messages from the queue depends whether you are using short (standard) polling, the default behavior, or long polling. For more information about long polling, see [Amazon SQS Long Polling \(p. 41\)](#).

With short polling, when you retrieve messages from the queue, Amazon SQS samples a subset of the servers (based on a weighted random distribution) and returns messages from just those servers. This means that a particular receive request might not return all your messages. Or, if you have a small number of messages in your queue (less than 1000), it means a particular request might not return any of your messages, whereas a subsequent request will. If you keep retrieving from your queues, Amazon SQS will sample all of the servers, and you will receive all of your messages.

The following figure shows short polling behavior of messages being returned after one of your system components makes a receive request. Amazon SQS samples several of the servers (in gray) and returns the messages from those servers (Message A, C, D, and B). Message E is not returned to this particular request, but it would be returned to a subsequent request.



## Queue and Message Identifiers

Amazon SQS uses the following three identifiers that you need to be familiar with:

- Queue URL
- Message ID
- Receipt handle

### Queue URLs

When creating a new queue, you must provide a queue name that is unique within the scope of all your queues. If you create queues using both the latest WSDL and a previous version, you still have a single namespace for all your queues. Amazon SQS assigns each queue you create an identifier called a *queue URL*, which includes the queue name and other components that Amazon SQS determines. Whenever you want to perform an action on a queue, you provide its queue URL.

The following is the queue URL for a queue named "queue2" owned by a person with the AWS account number "123456789012".

```
http://sqs.us-east-1.amazonaws.com/123456789012/queue2
```

#### Important

In your system, always store the entire queue URL as Amazon SQS returned it to you when you created the queue (for example,

`http://sqs.us-east-1.amazonaws.com/123456789012/queue2`). Don't build the queue URL from its separate components each time you need to specify the queue URL in a request because Amazon SQS could change the components that make up the queue URL.

You can also get the queue URL for a queue by listing your queues. Even though you have a single namespace for all your queues, the list of queues returned depends on the WSDL you use for the request. For more information, see [ListQueues](#).

### Message IDs

Each message receives a system-assigned *message ID* that Amazon SQS returns to you in the [SendMessage](#) response. This identifier is useful for identifying messages, but to delete a message, you

need the message's *receipt handle* instead of the message ID. The maximum length of a message ID is 100 characters.

## Receipt Handles

Each time you receive a message from a queue, you receive a *receipt handle* for that message. The handle is associated with the act of receiving the message, not with the message itself. To delete the message or to change the message visibility, you must provide the receipt handle and not the message ID. This means you must always receive a message before you can delete it (you can't put a message into the queue and then recall it). The maximum length of a receipt handle is 1024 characters.

### Important

If you receive a message more than once, each time you receive it, you get a different receipt handle. You must provide the most recently received receipt handle when you request to delete the message or the message might not be deleted.

Following is an example of a receipt handle.

```
MbZj6wDWli+JvwwJaBV+3dcjk2YW2vA3+STFF1jTM8tJJg6HRG6PYSasuWXPJB+Cw  
Lj1FjgXUv1uSj1gUPAWV66FU/WeR4mq2OKpEGYWbnLmpRCJVAYeMjeU5ZBdteCQ+QE  
auMZc8ZRv37sIW2iJKq3M9MFx1YvV11A2x/KSbkJ0=
```

## Resources Required to Process Messages

To help you estimate the resources needed to process your queued messages, Amazon SQS can provide you with an approximate number of messages in a queue. You can view the number of messages that are visible or you can view the number of messages that are not visible. For more information about visibility, see [Visibility Timeout \(p. 6\)](#).

### Important

Because of the distributed architecture of Amazon SQS, the result is not an exact count of the number of messages in a queue. In most cases it should be close to the actual number of messages in the queue, but you should not rely on the count being precise.

The following table lists the API action to use.

To do this...	Use this action	With <code>AttributeName</code> set to
Get the approximate number of messages in the queue	<a href="#">GetQueueAttributes</a>	<code>ApproximateNumberOfMessages</code>
Get the approximate number of messages in the queue that are not visible	<a href="#">GetQueueAttributes</a>	<code>ApproximateNumberOfMessages-NotVisible</code>

## Visibility Timeout

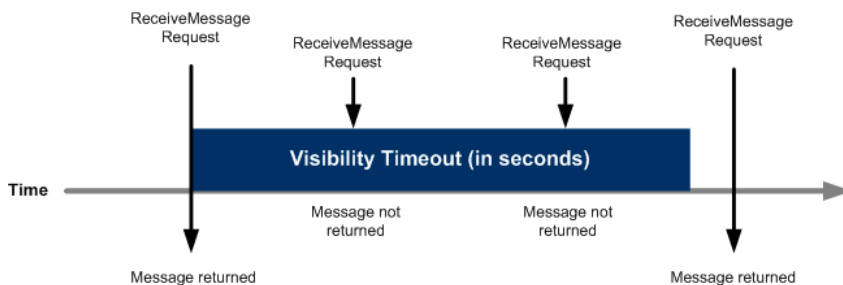
### Topics

- [General Recommendations for Visibility Timeout \(p. 7\)](#)
- [Extending a Message's Visibility Timeout \(p. 7\)](#)
- [Terminating a Message's Visibility Timeout \(p. 8\)](#)
- [API Actions Related to Visibility Timeout \(p. 8\)](#)

When a consuming component in your system receives and processes a message from the queue, the message remains in the queue. Why doesn't Amazon SQS automatically delete it?

Because your system is distributed, there's no guarantee that the component will actually receive the message (it's possible the connection could break or the component could fail before receiving the message). Therefore, Amazon SQS does not delete the message, and instead, your consuming component must delete the message from the queue after receiving and processing it.

Immediately after the component receives the message, the message is still in the queue. However, you don't want other components in the system receiving and processing the message again. Therefore, Amazon SQS blocks them with a *visibility timeout*, which is a period of time during which Amazon SQS prevents other consuming components from receiving and processing that message. The following figure and discussion illustrate the concept.



#### Note

There is a 120,000 limit for the number of inflight messages per queue. Messages are inflight after they have been received from the queue by a consuming component, but have not yet been deleted from the queue. If you reach the 120,000 limit, you will receive an `OverLimit` error message from Amazon SQS. To help avoid reaching the limit, you should delete the messages from the queue after they have been processed. You can also increase the number of queues you use to process the messages.

## General Recommendations for Visibility Timeout

The visibility timeout clock starts ticking once Amazon SQS returns the message. During that time, the component processes and deletes the message. But what happens if the component fails before deleting the message? If your system doesn't call `DeleteMessage` for that message before the visibility timeout expires, the message again becomes visible to the `ReceiveMessage` calls placed by the components in your system and it will be received again. If a message should only be received once, your system should delete it within the duration of the visibility timeout.

Each queue starts with a default setting of 30 seconds for the visibility timeout. You can change that setting for the entire queue. Typically, you'll set the visibility timeout to the average time it takes to process and delete a message from the queue. When receiving messages, you can also set a special visibility timeout for the returned messages without changing the overall queue timeout.

We recommend that if you have a system that produces messages that require varying amounts of time to process and delete, you create multiple queues, each with a different visibility timeout setting. Your system can then send all messages to a single queue that forwards each message to another queue with the appropriate visibility timeout based on the expected processing and deletion time for that message.

## Extending a Message's Visibility Timeout

When you receive a message from a queue and begin processing it, you may find the visibility timeout for the queue is insufficient to fully process and delete that message. To give yourself more time to process

the message, you can extend its visibility timeout by using the [ChangeMessageVisibility](#) action to specify a new timeout value. Amazon SQS restarts the timeout period using the new value.

For example, let's say the timeout for the queue is 30 seconds, and you receive a message from that queue. When you're 20 seconds into the timeout for that message (i.e., you have 10 seconds left), you want to give yourself 60 more seconds, so you immediately call `ChangeMessageVisibility` for the message with `VisibilityTimeout` set to 60 seconds. This means that you extended the message's visibility timeout from 30 seconds to 80 seconds: 20 seconds from the initial timeout setting plus 60 seconds from when you changed the timeout.

When you extend a message's visibility timeout, the new timeout applies only to that particular receipt of the message. `ChangeMessageVisibility` does not affect the timeout for the queue or later receipts of the message. If for some reason you don't delete the message and receive it again, its visibility timeout is the original value set for the queue.

## Terminating a Message's Visibility Timeout

When you receive a message from the queue, you might find that you actually don't want to process and delete that message. Amazon SQS allows you to terminate the visibility timeout for a specific message, which immediately makes the message visible to other components in the system to process. To do this, you call `ChangeMessageVisibility` with `VisibilityTimeout=0` seconds.

## API Actions Related to Visibility Timeout

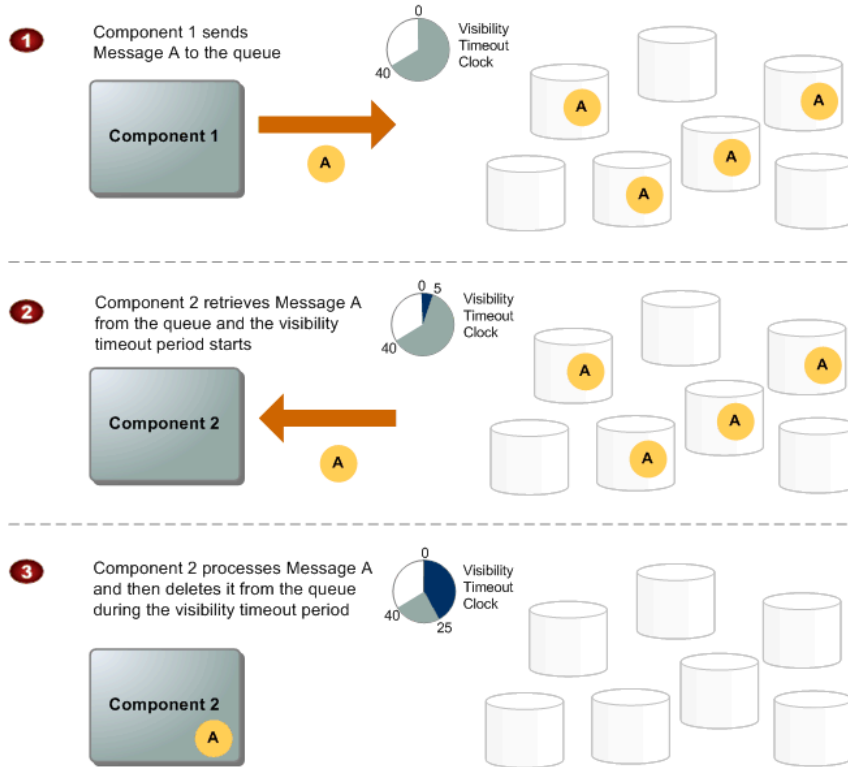
The following table lists the API actions to use to manipulate the visibility timeout. Use each action's `VisibilityTimeout` parameter to set or get the value.

To do this...	Use this action
Set the visibility timeout for a queue	<a href="#">SetQueueAttributes</a>
Get the visibility timeout for a queue	<a href="#">GetQueueAttributes</a>
Set the visibility timeout for the received messages without affecting the queue's visibility timeout	<a href="#">ReceiveMessage</a>
Extending or terminating a message's visibility timeout	<a href="#">ChangeMessageVisibility</a>
Extending or terminating the visibility timeout for up to ten messages.	<a href="#">ChangeMessageVisibilityBatch</a>

## Message Lifecycle

The following diagram and process describe the lifecycle of an Amazon SQS message, called *Message A*, from creation to deletion. Assume that a queue already exists.

## Amazon Simple Queue Service Developer Guide Message Lifecycle



### Message Lifecycle

1	Component 1 sends Message A to a queue, and the message is redundantly distributed across the SQS servers.
2	When Component 2 is ready to process a message, it retrieves messages from the queue, and Message A is returned. While Message A is being processed, it remains in the queue and is not returned to subsequent receive requests for the duration of the <i>visibility timeout</i> .
3	Component 2 deletes Message A from the queue to avoid the message being received and processed again once the visibility timeout expires.

#### Note

SQS automatically deletes messages that have been in a queue for more than maximum message retention period. The default message retention period is 4 days. However, you can set the message retention period to a value from 60 seconds to 1209600 seconds (14 days) with [SetQueueAttributes](#).

# Using JMS with Amazon SQS

---

The Amazon SQS Java Messaging Library is a JMS interface to Amazon SQS that enables you to leverage Amazon SQS in your applications that already use JMS. The interface enables you to use Amazon SQS as the JMS provider with minimal code changes. Along with the AWS SDK for Java, the Amazon SQS Java Messaging Library enables you to create JMS connections, sessions, producers, and consumers to send and receive messages to and from Amazon SQS queues.

The Amazon SQS Java Messaging Library supports sending and receiving messages to a queue (the JMS point-to-point model), as specified in the [JMS 1.1 specification](#). More specifically, the Amazon SQS Java Messaging Library supports synchronously sending text, byte, or object messages to Amazon SQS queues, as well as receiving them synchronously or asynchronously. For more information about the supported features of the JMS 1.1 specification in the Amazon SQS Java Messaging Library, see [Reference/Appendix \(p. 30\)](#) and the [Amazon SQS FAQs](#).

You must create an Amazon SQS queue to use with JMS. To create an Amazon SQS queue, you can use the AWS Management Console for Amazon SQS, the `CreateQueue` API, or the wrapped Amazon SQS client included in the Amazon SQS Java Messaging Library. For information about creating a queue with Amazon SQS using either the AWS Management Console or the `CreateQueue` API, go to [Creating a Queue](#). For information about using the Amazon SQS Java Messaging Library, go to [Getting Started with the Amazon SQS Java Messaging Library \(p. 11\)](#).

## Topics

- [Amazon SQS and JMS Prerequisites \(p. 10\)](#)
- [Getting Started with the Amazon SQS Java Messaging Library \(p. 11\)](#)
- [Receiving Messages Asynchronously \(p. 13\)](#)
- [Client Acknowledge Mode \(p. 14\)](#)
- [Unordered Acknowledge Mode \(p. 15\)](#)
- [Code Samples \(p. 16\)](#)
- [Reference/Appendix \(p. 30\)](#)

## Amazon SQS and JMS Prerequisites

To use Amazon SQS with JMS, you must have the following:

- **SDK for Java** – There are two different ways for including the SDK for Java in your project. You can either download and install the SDK for Java, or if you use Maven to obtain the Amazon SQS Java Messaging Library, then the SDK for Java is included as a dependency. The SDK for Java and Amazon

SQS Java Messaging Library require J2SE Development Kit 6.0 or later. For information about downloading the SDK for Java, go to [SDK for Java](#). For more information about using Maven see the note below.

- **Amazon SQS Java Messaging Library** – If you do not use Maven, then you must add the package file, `amazon-sqs-java-messaging-lib.jar`, to the Java build class path. For information about downloading, go to [Amazon SQS Java Messaging Library](#).

**Note**

The Amazon SQS Java Messaging Library includes support for [Maven](#) and the [Spring Framework](#). For code samples that use Maven, the Spring Framework, and the Amazon SQS Java Messaging Library, see [Code Samples \(p. 16\)](#).

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>amazon-sqs-java-messaging-lib</artifactId>
  <version>1.0.0</version>
  <type>jar</type>
</dependency>
```

## Getting Started with the Amazon SQS Java Messaging Library

After you have met the prerequisites, you can then use the following code examples to get started using Amazon SQS with JMS.

This example creates a JMS connection and session, and then sends and receives a message. The wrapped Amazon SQS client, which is included in the Amazon SQS Java Messaging Library, also checks if the Amazon SQS queue exists. It also uses the wrapped Amazon SQS client to check if the Amazon SQS queue exists. If it does not exist then the queue is created.

### Creating a JMS Connection

In order to create a JMS connection, you must first create a connection factory and call the `createConnection` method against the factory:

```
// Create the connection factory using the environment variable credential
// provider.
// Connections this factory creates can talk to the queues in us-east-1 region.

SQSConnectionFactory connectionFactory =
    SQSConnectionFactory.builder()
        .withRegion(Region.getRegion(Regions.US_EAST_1))
        .withAWSCredentialsProvider(new EnvironmentVariableCredentialsProvider())

        .build();

// Create the connection.
SQSConnection connection = connectionFactory.createConnection();
```



### Note

The [EnvironmentVariableCredentialsProvider](#) class in this example assumes that the `AWS_ACCESS_KEY_ID` (or `AWS_ACCESS_KEY`) and `AWS_SECRET_KEY` (or `AWS_SECRET_ACCESS_KEY`) environment variables are set. For more information, including other options for providing the required credentials to the factory, go to [AWSCredentialsProvider](#).

The `SQSConnection` class extends `javax.jms.Connection`. Along with the JMS standard connection methods, `SQSConnection` offers additional methods, such as `getAmazonSQSClient` and `getWrappedAmazonSQSClient`. The returned client objects from `getAmazonSQSClient` and `getWrappedAmazonSQSClient` can be used to perform administrative operations that are not included in the JMS specification – for example, creating an Amazon SQS queue.

If you use `getWrappedAmazonSQSClient`, then the returned client object will transform all exceptions into JMS exceptions. On the other hand, if you use `getAmazonSQSClient`, then the exceptions will be Amazon SQS exceptions. Therefore, if you have existing code that is expecting JMS exceptions, then you should use `getWrappedAmazonSQSClient`.

## Creating an Amazon SQS Queue

This example checks if an Amazon SQS queue exists, and if not then it creates one. Next, the wrapped client object is used to check if the Amazon SQS queue exists, and if not a new one is created:

```
// Get the wrapped client
AmazonSQSMessagingClientWrapper client = connection.getWrappedAmazonSQSClient();

// Create an SQS queue named 'TestQueue' - if it does not already exist.
if (!client.queueExists("TestQueue")) {
    client.createQueue("TestQueue");
}
```

## Sending Messages Synchronously

With the connection and underlying Amazon SQS queue ready, a non-transacted JMS session with `AUTO_ACKNOWLEDGE` mode is created:

```
// Create the non-transacted session with AUTO_ACKNOWLEDGE mode
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

Next, in order to send a text message to the queue, a JMS queue identity and message producer are created:

```
// Create a queue identity with name 'TestQueue' in the session
Queue queue = session.createQueue("TestQueue");

// Create a producer for the 'TestQueue'.
MessageProducer producer = session.createProducer(queue);
```

A text message is created and sent to the queue:

```
// Create the text message.
TextMessage message = session.createTextMessage("Hello World!");
```

```
// Send the message.  
producer.send(message);  
System.out.println("JMS Message " + message.getJMSMessageID());
```

## Receiving Messages Synchronously

To receive messages, a consumer is created on the same queue and the `start` method is invoked. The `start` method on the connection can be called anytime; however, the consumer will not start receiving messages until after it has been called.

```
// Create a consumer for the 'TestQueue'.  
MessageConsumer consumer = session.createConsumer(queue);  
  
// Start receiving incoming messages.  
connection.start();
```

The `receive` method is called on the consumer with a time-out set to 1 second and the content of the received message is printed:

```
// Receive a message from 'TestQueue' and wait up to 1 second  
Message receivedMessage = consumer.receive(1000);  
  
// Cast the received message as TextMessage and print the text to screen.  
if (receivedMessage != null) {  
    System.out.println("Received: " + ((TextMessage) receivedMessage).getText());  
}
```

Finally, the connection is closed, which also closes the session:

```
// Close the connection (and the session).  
connection.close();
```

The output will look similar to the following:

```
JMS Message ID:8example-588b-44e5-bbcf-d816example2  
Received: Hello World!
```

### Note

You can also use Spring to initialize these objects. For additional information, see `SpringExampleConfig.xml`, `SpringExample.java`, and other helper classes in `ExampleConfiguration.java` and `ExampleCommon.java`, located in [Code Samples \(p. 16\)](#).

For complete examples of send and receive, see `TextMessageSender.java` and `SyncMessageReceiver.java`.

## Receiving Messages Asynchronously

In the example provided in [Getting Started with the Amazon SQS Java Messaging Library \(p. 11\)](#), a message is sent to `TestQueue` and received synchronously. This example shows how to receive the messages asynchronously through a listener.

First, the `MessageListener` interface is implemented:

```
class MyListener implements MessageListener {  
  
    @Override  
    public void onMessage(Message message) {  
        try {  
            // Cast the received message as TextMessage and print the text to  
            // screen.  
            if (message != null) {  
                System.out.println("Received: " + ((TextMessage) message).get  
Text());  
            }  
        } catch (JMSEException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

The `onMessage` method of the `MessageListener` interface is called when a message is received. In this listener implementation, the text stored in the message is printed.

Next, instead of explicitly calling the `receive` method on the consumer, the message listener of the consumer to an instance of the `MyListener` implementation is set. And, the main thread sleeps for a second:

```
// Create a consumer for the 'TestQueue'.  
MessageConsumer consumer = session.createConsumer(queue);  
  
// Instantiate and set the message listener for the consumer.  
consumer.setMessageListener(new MyListener());  
  
// Start receiving incoming messages.  
connection.start();  
  
// Wait for 1 second. The listener onMessage() method will be invoked when a  
// message is received.  
Thread.sleep(1000);
```

The rest of the steps are identical to the ones provided in the example in [Getting Started with the Amazon SQS Java Messaging Library \(p. 11\)](#). For a complete example of an asynchronous receiver, see `AsyncMessageReceiver.java` in [Code Samples \(p. 16\)](#).

The output for this example will look similar to the following:

```
JMS Message ID:8example-588b-44e5-bbcf-d816example2  
Received: Hello World!
```

## Client Acknowledge Mode

In the example provided in [Getting Started with the Amazon SQS Java Messaging Library \(p. 11\)](#), `AUTO_ACKNOWLEDGE` mode is used, where every received message is automatically acknowledged – and therefore deleted from the underlying Amazon SQS queue. If you want to explicitly acknowledge the

messages after they are done being processed, then you must create the session with `CLIENT_ACKNOWLEDGE` mode:

```
// Create the non-transacted session with CLIENT_ACKNOWLEDGE mode.
Session session = connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);
```

Next, when the message is received, it is both displayed and explicitly acknowledged:

```
// Cast the received message as TextMessage and print the text to screen. Also
// acknowledge the message.
if (receivedMessage != null) {
    System.out.println("Received: " + ((TextMessage) receivedMessage).getText());

    receivedMessage.acknowledge();
    System.out.println("Acknowledged: " + message.getJMSMessageID());
}
```

### Note

In this mode, when a message is acknowledged, then all the messages received prior are implicitly acknowledged as well. For example, if 10 messages are received, and only the 10th message (in the order they are received) is acknowledged, then all the previous 9 messages are also acknowledged.

The rest of the steps are identical to the ones provided in the example in [Getting Started with the Amazon SQS Java Messaging Library \(p. 11\)](#). For a complete example of a synchronous receiver with client acknowledge mode, see `SyncMessageReceiverClientAcknowledge.java` in [Code Samples \(p. 16\)](#).

The output for this example will look similar to the following:

```
JMS Message ID: 4example-aa0e-403f-b6df-5e02example5
Received: Hello World!
Acknowledged: ID: 4example-aa0e-403f-b6df-5e02example5
```

## Unordered Acknowledge Mode

In the previous example shown in [Client Acknowledge Mode \(p. 14\)](#), (when using `CLIENT_ACKNOWLEDGE` mode) all the messages received prior to an explicitly acknowledged message are automatically acknowledged. In the JMS 1.1 specification this is referred to as the `ACK-RANGE`. The Amazon SQS Java Messaging Library provides another acknowledgement mode, `UNORDERED_ACKNOWLEDGE`, where all received messages must be explicitly acknowledged by the client – regardless of the order they are received. In other words, when using `UNORDERED_ACKNOWLEDGE` mode and a message is acknowledged, then only that specific message is acknowledged.

In this example, a session is created with `UNORDERED_ACKNOWLEDGE` mode:

```
// Create the non-transacted session with UNORDERED_ACKNOWLEDGE mode.
Session session = connection.createSession(false, SQSSession.UNORDERED_ACKNOWLEDGE);
```

The rest of the steps are identical to the ones provided in the example in [Client Acknowledge Mode \(p. 14\)](#). For a complete example of a synchronous receiver with unordered acknowledge mode, see `SyncMessageReceiverUnorderedAcknowledge.java` in [Code Samples \(p. 16\)](#).

The output for this example will look similar to the following:

```
JMS Message ID:dexample-73ad-4adb-bc6c-4357example7
Received: Hello World!
Acknowledged: ID:dexample-73ad-4adb-bc6c-4357example7
```

## Code Samples

The following code samples show how to use JMS with Amazon SQS.

### ExampleConfiguration.java

The following Java code is a configuration example that can be used for all subsequent Java examples. For example, this code sets the default queue name, region, and credentials to be used with the other java examples.

```
public class ExampleConfiguration {
    public static final String DEFAULT_QUEUE_NAME = "SQSJMSClientExampleQueue";

    public static final Region DEFAULT_REGION = Region.getRegion(Regions.US_EAST_1);

    private static String getParameter( String args[], int i ) {
        if( i + 1 >= args.length ) {
            throw new IllegalArgumentException( "Missing parameter for " +
args[i] );
        }
        return args[i+1];
    }

    /**
     * Parse the command line and return the resulting config. If the config
     parsing fails
     * print the error and the usage message and then call System.exit
     *
     * @param app the app to use when printing the usage string
     * @param args the command line arguments
     * @return the parsed config
     */
    public static ExampleConfiguration parseConfig(String app, String args[])
    {
        try {
            return new ExampleConfiguration(args);
        } catch (IllegalArgumentException e) {
            System.err.println( "ERROR: " + e.getMessage() );
            System.err.println();
            System.err.println( "Usage: " + app + " [--queue <queue>] [--region
<region>] [--credentials <credentials>] ");
            System.err.println( "    or " );
            System.err.println( "    " + app + " <spring.xml> " );
            System.exit(-1);
            return null;
        }
    }
}
```

```
    }

    private ExampleConfiguration(String args[]) {
        for( int i = 0; i < args.length; ++i ) {
            String arg = args[i];
            if( arg.equals( "--queue" ) ) {
                setQueueName(getParameter(args, i));
                i++;
            } else if( arg.equals( "--region" ) ) {
                String regionName = getParameter(args, i);
                try {
                    setRegion(Region.getRegion(Regions.fromName(regionName)));
                } catch( IllegalArgumentException e ) {
                    throw new IllegalArgumentException( "Unrecognized region "
+ regionName );
                }
                i++;
            } else if( arg.equals( "--credentials" ) ) {
                String credsFile = getParameter(args, i);
                try {
                    setCredentialsProvider( new PropertiesFileCredentialsPro
vider(credsFile) );
                } catch (AmazonClientException e) {
                    throw new IllegalArgumentException("Error reading credentials
from " + credsFile, e );
                }
                i++;
            } else {
                throw new IllegalArgumentException("Unrecognized option " +
arg);
            }
        }
    }

    private String queueName = DEFAULT_QUEUE_NAME;
    private Region region = DEFAULT_REGION;
    private AWSCredentialsProvider credentialsProvider = new DefaultAWS creden
tialsProviderChain();

    public String getQueueName() {
        return queueName;
    }

    public void setQueueName(String queueName) {
        this.queueName = queueName;
    }

    public Region getRegion() {
        return region;
    }

    public void setRegion(Region region) {
        this.region = region;
    }

    public AWSCredentialsProvider getCredentialsProvider() {
        return credentialsProvider;
    }
}
```

```
    }

    public void setCredentialsProvider(AWSCredentialsProvider credentialsProvider) {
        // Make sure they're usable first
        credentialsProvider.getCredentials();
        this.credentialsProvider = credentialsProvider;
    }
}
```

## TextMessageSender.java

The following Java code shows how to create a text message sender.

```
public class TextMessageSender {
    public static void main(String args[]) throws JMSEException {
        ExampleConfiguration config = ExampleConfiguration.parseConfig("TextMessageSender", args);

        ExampleCommon.setupLogging();

        // Create the connection factory based on the config
        SQSConnectionFactory connectionFactory =
            SQSConnectionFactory.builder()
                .withRegion(config.getRegion())
                .withAWSCredentialsProvider(config.getCredentialsProvider())

                .build();

        // Create the connection
        SQSConnection connection = connectionFactory.createConnection();

        // Create the queue if needed
        ExampleCommon.ensureQueueExists(connection, config.getQueueName());

        // Create the session
        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        MessageProducer producer = session.createProducer( session.createQueue(
            config.getQueueName() ) );

        sendMessages(session, producer);

        // Close the connection. This will close the session automatically
        connection.close();
        System.out.println( "Connection closed" );
    }

    private static void sendMessages( Session session, MessageProducer producer ) {
        BufferedReader inputReader = new BufferedReader(
            new InputStreamReader( System.in, Charset.defaultCharset() ) );

        try {
            String input;
```

```
        while( true ) {
            System.out.print( "Enter message to send (leave empty to exit):
" );

            input = inputReader.readLine();
            if( input == null || input.equals(" " ) ) break;

            TextMessage message = session.createTextMessage(input);
            producer.send(message);
            System.out.println( "Send message " + message.getJMSMessageID()
);
        }
    } catch (EOFException e) {
        // Just return on EOF
    } catch (IOException e) {
        System.err.println( "Failed reading input: " + e.getMessage() );
    } catch (JMSEException e) {
        System.err.println( "Failed sending message: " + e.getMessage() );

        e.printStackTrace();
    }
}
}
```

## AsyncMessageReceiver.java

The following Java code shows how to create an asynchronous message receiver.

```
public class AsyncMessageReceiver {
    public static void main(String args[]) throws JMSEException, InterruptedEx
ception {
        ExampleConfiguration config = ExampleConfiguration.parseConfig("Asyn
cMessageReceiver", args);

        ExampleCommon.setupLogging();

        // Create the session
        Session session = connection.createSession(false, Session.CLIENT_AC
KNOWLEDGE);
        MessageConsumer consumer = session.createConsumer( session.createQueue(
config.getQueueName() ) );

        ReceiverCallback callback = new ReceiverCallback();
        consumer.setMessageListener( callback );

        // No messages will be processed until this is called
        connection.start();

        callback.waitForOneMinuteOfSilence();
        System.out.println( "Returning after one minute of silence" );

        // Close the connection. This will close the session automatically
        connection.close();
        System.out.println( "Connection closed" );
    }
}
```



```
private static class ReceiverCallback implements MessageListener {
    // Used to listen for message silence
    private volatile long timeOfLastMessage = System.nanoTime();

    public void waitForOneMinuteOfSilence() throws InterruptedException
    {
        for(;;) {
            long timeSinceLastMessage = System.nanoTime() - timeOfLast
Message;
            long remainingTillOneMinuteOfSilence =
                TimeUnit.MINUTES.toNanos(1) - timeSinceLastMessage;

            if( remainingTillOneMinuteOfSilence < 0 ) {
                break;
            }
            TimeUnit.NANOSECONDS.sleep(remainingTillOneMinuteOfSilence);
        }
    }

    @Override
    public void onMessage(Message message) {
        try {
            ExampleCommon.handleMessage(message);
            message.acknowledge();
            System.out.println( "Acknowledged message " + mes
sage.getJMSMessageID() );
            timeOfLastMessage = System.nanoTime();
        } catch (JMSEException e) {
            System.err.println( "Error processing message: " + e.getMes
sage() );
            e.printStackTrace();
        }
    }
}
```

## SyncMessageReceiver.java

The following Java code shows how to create a synchronous message receiver.

```
public class SyncMessageReceiver {
    public static void main(String args[]) throws JMSEException {
        ExampleConfiguration config = ExampleConfiguration.parseConfig("SyncMes
sageReceiver", args);

        ExampleCommon.setupLogging();

        // Create the connection factory based on the config
        SQSConnectionFactory connectionFactory =
            SQSConnectionFactory.builder()
                .withRegion(config.getRegion())
                .withAWSCredentialsProvider(config.getCredentialsProvider())

                .build();
```

```
// Create the connection
SQSConnection connection = connectionFactory.createConnection();

// Create the queue if needed
ExampleCommon.ensureQueueExists(connection, config.getQueueName());

// Create the session
Session session = connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);
MessageConsumer consumer = session.createConsumer( session.createQueue(
config.getQueueName() ) );

connection.start();

receiveMessages(session, consumer);

// Close the connection. This will close the session automatically
connection.close();
System.out.println( "Connection closed" );
}

private static void receiveMessages( Session session, MessageConsumer consumer ) {
    try {
        while( true ) {
            System.out.println( "Waiting for messages");
            // Wait 1 minute for a message
            Message message = consumer.receive(TimeUnit.MINUTES.toMillis(1));

            if( message == null ) {
                System.out.println( "Shutting down after 1 minute of silence" );
                break;
            }
            ExampleCommon.handleMessage(message);
            message.acknowledge();
            System.out.println( "Acknowledged message " + message.getJMSPMessageID() );
        }
    } catch (JMSEException e) {
        System.err.println( "Error receiving from SQS: " + e.getMessage() );
        e.printStackTrace();
    }
}
}
```

## SyncMessageReceiverClientAcknowledge.java

The following Java code shows an example of a synchronous receiver with client acknowledge mode.

```
/**
 * An example class to demonstrate the behavior of CLIENT_ACKNOWLEDGE mode for
 * received messages. This example
 * complements the example given in {@link SyncMessageReceiverUnorderedAcknow
```

```
ledge} for UNORDERED_ACKNOWLEDGE mode.
*
* First, a session, a message producer, and a message consumer are created.
Then, two messages are sent. Next, two messages
* are received but only the second one is acknowledged. After waiting for the
visibility time out period, an attempt to
* receive another message is made. It is shown that no message is returned for
this attempt since in CLIENT_ACKNOWLEDGE mode,
* as expected, all the messages prior to the acknowledged messages are also
acknowledged.
*
* This is NOT the behavior for UNORDERED_ACKNOWLEDGE mode. Please see {@link
SyncMessageReceiverUnorderedAcknowledge}
* for an example.
*/
public class SyncMessageReceiverClientAcknowledge {

    // Visibility time-out for the queue. It must match to the one set for the
queue for this example to work.
    private static final long TIME_OUT_SECONDS = 1;

    public static void main(String args[]) throws JMSEException, InterruptedEx
ception {
        // Create the configuration for the example
        ExampleConfiguration config = ExampleConfiguration.parseConfig("SyncMes
sageReceiverClientAcknowledge", args);

        // Setup logging for the example
        ExampleCommon.setupLogging();

        // Create the connection factory based on the config
        SQSConnectionFactory connectionFactory =
            SQSConnectionFactory.builder()
                .withRegion(config.getRegion())
                .withAWSCredentialsProvider(config.getCredentialsPro
vider())
                .build();

        // Create the connection
        SQSConnection connection = connectionFactory.createConnection();

        // Create the queue if needed
        ExampleCommon.ensureQueueExists(connection, config.getQueueName());

        // Create the session with client acknowledge mode
        Session session = connection.createSession(false, Session.CLIENT_ACKNOW
LEDGE);

        // Create the producer and consume
        MessageProducer producer = session.createProducer(session.createQueue(con
fig.getQueueName()));
        MessageConsumer consumer = session.createConsumer(session.createQueue(con
fig.getQueueName()));

        // Open the connection
        connection.start();

        // Send two text messages
```

```
        sendMessage(producer, session, "Message 1");
        sendMessage(producer, session, "Message 2");

        // Receive a message and don't acknowledge it
        receiveMessage(consumer, false);

        // Receive another message and acknowledge it
        receiveMessage(consumer, true);

        // Wait for the visibility time out, so that unacknowledged messages
        reappear in the queue
        System.out.println("Waiting for visibility timeout...");
        Thread.sleep(TimeUnit.SECONDS.toMillis(TIME_OUT_SECONDS));

        // Attempt to receive another message and acknowledge it. This will
        result in receiving no messages since
        // we have acknowledged the second message. Although we did not expli-
        citly acknowledge the first message,
        // in the CLIENT_ACKNOWLEDGE mode, all the messages received prior to
        the explicitly acknowledged message
        // are also acknowledged. Therefore, we have implicitly acknowledged
        the first message.
        receiveMessage(consumer, true);

        // Close the connection. This will close the session automatically
        connection.close();
        System.out.println("Connection closed.");
    }

    /**
     * Sends a message through the producer.
     *
     * @param producer Message producer
     * @param session Session
     * @param messageText Text for the message to be sent
     * @throws JMSEException
     */
    private static void sendMessage(MessageProducer producer, Session session,
        String messageText) throws JMSEException {
        // Create a text message and send it
        producer.send(session.createTextMessage(messageText));
    }

    /**
     * Receives a message through the consumer synchronously with the default
     timeout (TIME_OUT_SECONDS).
     * If a message is received, the message is printed. If no message is re-
     ceived, "Queue is empty!" is
     * printed.
     *
     * @param consumer Message consumer
     * @param acknowledge If true and a message is received, the received message
     is acknowledged.
     * @throws JMSEException
     */
    private static void receiveMessage(MessageConsumer consumer, boolean acknow-
        ledge) throws JMSEException {
        // Receive a message
```

```
        Message message = consumer.receive(TimeUnit.SECONDS.to
Millis(TIME_OUT_SECONDS));

        if (message == null) {
            System.out.println("Queue is empty!");
        } else {
            // Since this queue has only text messages, cast the message object
            and print the text
            System.out.println("Received: " + ((TextMessage) message).getText());

            // Acknowledge the message if asked
            if (acknowledge) message.acknowledge();
        }
    }
}
```

## SyncMessageReceiverUnorderedAcknowledge.java

The following Java code shows an example of a synchronous receiver with unordered acknowledge mode.

```
/**
 * An example class to demonstrate the behavior of UNORDERED_ACKNOWLEDGE mode
 * for received messages. This example
 * complements the example given in {@link SyncMessageReceiverClientAcknowledge}
 * for CLIENT_ACKNOWLEDGE mode.
 *
 * First, a session, a message producer, and a message consumer are created.
 * Then, two messages are sent. Next, two messages
 * are received but only the second one is acknowledged. After waiting for the
 * visibility time out period, an attempt to
 * receive another message is made. It is shown that the first message received
 * in the prior attempt is returned again
 * for the second attempt. In UNORDERED_ACKNOWLEDGE mode, all the messages must
 * be explicitly acknowledged no matter what
 * the order they are received.
 *
 * This is NOT the behavior for CLIENT_ACKNOWLEDGE mode. Please see {@link
 * SyncMessageReceiverClientAcknowledge}
 * for an example.
 */
public class SyncMessageReceiverUnorderedAcknowledge {

    // Visibility time-out for the queue. It must match to the one set for the
    // queue for this example to work.
    private static final long TIME_OUT_SECONDS = 1;

    public static void main(String args[]) throws JMSEException, InterruptedEx
ception {
        // Create the configuration for the example
        ExampleConfiguration config = ExampleConfiguration.parseConfig("SyncMes
sageReceiverUnorderedAcknowledge", args);

        // Setup logging for the example
        ExampleCommon.setupLogging();
    }
}
```

```
        // Create the connection factory based on the config
        SQSConnectionFactory connectionFactory =
            SQSConnectionFactory.builder()
                .withRegion(config.getRegion())
                .withAWSCredentialsProvider(config.getCredentialsProvider())
                .build();

        // Create the connection
        SQSConnection connection = connectionFactory.createConnection();

        // Create the queue if needed
        ExampleCommon.ensureQueueExists(connection, config.getQueueName());

        // Create the session with unordered acknowledge mode
        Session session = connection.createSession(false, SQSConnection.UNORDERED_ACKNOWLEDGE);

        // Create the producer and consumer
        MessageProducer producer = session.createProducer(session.createQueue(config.getQueueName()));
        MessageConsumer consumer = session.createConsumer(session.createQueue(config.getQueueName()));

        // Open the connection
        connection.start();

        // Send two text messages
        sendMessage(producer, session, "Message 1");
        sendMessage(producer, session, "Message 2");

        // Receive a message and don't acknowledge it
        receiveMessage(consumer, false);

        // Receive another message and acknowledge it
        receiveMessage(consumer, true);

        // Wait for the visibility time out, so that unacknowledged messages
        // reappear in the queue
        System.out.println("Waiting for visibility timeout...");
        Thread.sleep(TimeUnit.SECONDS.toMillis(TIME_OUT_SECONDS));

        // Attempt to receive another message and acknowledge it. This will
        // result in receiving the first message since
        // we have acknowledged only the second message. In the UNORDERED_ACKNOWLEDGE
        // mode, all the messages must
        // be explicitly acknowledged.
        receiveMessage(consumer, true);

        // Close the connection. This will close the session automatically
        connection.close();
        System.out.println("Connection closed.");
    }

    /**
     * Sends a message through the producer.
     */
```

```
* @param producer Message producer
* @param session Session
* @param messageText Text for the message to be sent
* @throws JMSEException
*/
private static void sendMessage(MessageProducer producer, Session session,
String messageText) throws JMSEException {
    // Create a text message and send it
    producer.send(session.createTextMessage(messageText));
}

/**
 * Receives a message through the consumer synchronously with the default
 * timeout (TIME_OUT_SECONDS).
 * If a message is received, the message is printed. If no message is re
 * ceived, "Queue is empty!" is
 * printed.
 *
 * @param consumer Message consumer
 * @param acknowledge If true and a message is received, the received message
 * is acknowledged.
 * @throws JMSEException
 */
private static void receiveMessage(MessageConsumer consumer, boolean acknow
ledge) throws JMSEException {
    // Receive a message
    Message message = consumer.receive(TimeUnit.SECONDS.to
Millis(TIME_OUT_SECONDS));

    if (message == null) {
        System.out.println("Queue is empty!");
    } else {
        // Since this queue has only text messages, cast the message object
        and print the text
        System.out.println("Received: " + ((TextMessage) message).getText());

        // Acknowledge the message if asked
        if (acknowledge) message.acknowledge();
    }
}
}
```

## SpringExampleConfig.xml

The following shows an example Spring configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:util="http://www.springframework.org/schema/util"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframe
        work.org/schema/beans/spring-beans-3.0.xsd
```

```
        http://www.springframework.org/schema/util http://www.springframe
work.org/schema/util/spring-util-3.0.xsd
    ">

    <bean id="CredentialsProviderBean" class="com.amazonaws.auth.DefaultAWSCre
dentialsProviderChain"/>

    <bean id="ConnectionFactoryBuilder" class="com.amazon.sqs.javames
saging.SQSConnectionFactory$Builder">
        <property name="regionName" value="us-east-1"/>
        <property name="numberOfMessagesToPrefetch" value="5"/>

        <property name="awsCredentialsProvider" ref="CredentialsProviderBean"/>
    </bean>

    <bean id="ConnectionFactory" class="com.amazon.sqs.javamessaging.SQSConnection
Factory"
        factory-bean="ConnectionFactoryBuilder"
        factory-method="build" />

    <bean id="Connection" class="javax.jms.Connection"
        factory-bean="ConnectionFactory"
        factory-method="createConnection"
        init-method="start"
        destroy-method="close" />

    <bean id="QueueName" class="java.lang.String">
        <constructor-arg value="SQSJMSClientExampleQueue"/>
    </bean>

</beans>
```

## SpringExample.java

The following Java code shows an example that uses [SpringExampleConfig.xml](#) (p. 26).

```
public class SpringExample {
    public static void main(String args[]) throws JMSEException {
        if( args.length != 1 || !args[0].endsWith(".xml")) {
            System.err.println( "Usage: " + SpringExample.class.getName() + "
<spring config.xml>" );
            System.exit(1);
        }

        File springFile = new File( args[0] );
        if( !springFile.exists() || !springFile.canRead() ) {
            System.err.println( "File " + args[0] + " does not exist or is not
readable." );
            System.exit(2);
        }

        ExampleCommon.setupLogging();

        FileSystemXmlApplicationContext context =
            new FileSystemXmlApplicationContext( "file://" + springFile.get
AbsolutePath() );
    }
}
```



```
        Connection connection;
        try {
            connection = context.getBean(Connection.class);
        } catch( NoSuchBeanDefinitionException e ) {
            System.err.println( "Could not find the JMS connection to use: " +
e.getMessage() );
            System.exit(3);
            return;
        }

        String queueName;
        try {
            queueName = context.getBean("QueueName", String.class);
        } catch( NoSuchBeanDefinitionException e ) {
            System.err.println( "Could not find the name of the queue to use:
" + e.getMessage() );
            System.exit(3);
            return;
        }

        if( connection instanceof SQSConnection ) {
            ExampleCommon.ensureQueueExists( (SQSConnection) connection,
queueName );
        }

        // Create the session
        Session session = connection.createSession(false, Session.CLIENT_ACKNOW
LEDGE);
        MessageConsumer consumer = session.createConsumer( session.createQueue(
queueName) );

        receiveMessages(session, consumer);

        // The context can be setup to close the connection for us
        context.close();
        System.out.println( "Context closed" );
    }

    private static void receiveMessages( Session session, MessageConsumer con
sumer ) {
        try {
            while( true ) {
                System.out.println( "Waiting for messages");
                // Wait 1 minute for a message
                Message message = consumer.receive(TimeUnit.MINUTES.toMillis(1));

                if( message == null ) {
                    System.out.println( "Shutting down after 1 minute of silence"
);

                    break;
                }
                ExampleCommon.handleMessage(message);
                message.acknowledge();
                System.out.println( "Acknowledged message" );
            }
        } catch (JMSEException e) {
            System.err.println( "Error receiving from SQS: " + e.getMessage()
);
        }
    }
}
```

```
);  
        e.printStackTrace();  
    }  
}  
}
```

## ExampleCommon.java

The following Java code first checks to see if an Amazon SQS queue exists and then creates one if it does not already exist. In addition, example logging code is provided.

```
public class ExampleCommon {  
    /**  
     * A utility function to check the queue exists and create it if needed.  
     For most  
     * use cases this will usually be done by an administrator before the ap  
     plication  
     * is run.  
     */  
    public static void ensureQueueExists(SQSConnection connection, String  
queueName) throws JMSEException {  
        AmazonSQSMessagingClientWrapper client = connection.getWrappedAmazon  
SQSClient();  
  
        /**  
         * For most cases this could be done with just a createQueue call, but  
         GetQueueUrl  
         * (called by queueExists) is a faster operation for the common case  
         where the queue  
         * already exists. Also many users and roles have permission to call  
         GetQueueUrl  
         * but do not have permission to call CreateQueue.  
         */  
        if( !client.queueExists(queueName) ) {  
            client.createQueue( queueName );  
        }  
    }  
  
    public static void setupLogging() {  
        // Setup logging  
        BasicConfigurator.configure();  
        Logger.getRootLogger().setLevel(Level.WARN);  
    }  
  
    public static void handleMessage(Message message) throws JMSEException {  
        System.out.println( "Got message " + message.getJMSMessageID() );  
        System.out.println( "Content: " );  
        if( message instanceof TextMessage ) {  
            TextMessage txtMessage = ( TextMessage ) message;  
            System.out.println( "\t" + txtMessage.getText() );  
        } else if( message instanceof BytesMessage ) {  
            BytesMessage byteMessage = ( BytesMessage ) message;  
            // Assume the length fits in an int - SQS only supports sizes up  
to 256k so that  
            // should be true  
            byte[] bytes = new byte[(int)byteMessage.getBodyLength()];
```

```
byteMessage.readBytes(bytes);
System.out.println( "\t" + Base64.encodeAsString( bytes ) );
} else if( message instanceof ObjectMessage ) {
    ObjectMessage objMessage = (ObjectMessage) message;
    System.out.println( "\t" + objMessage.getObject() );
}
}
```

## Reference/Appendix

The following lists some of the supported [JMS 1.1 specification](#) implementations in the Amazon SQS Java Messaging Library. For more information about the supported features and capabilities of the Amazon SQS Java Messaging Library, see the [Amazon SQS FAQ](#).

### Supported JMS 1.1 Common Interfaces:

- ConnectionFactory
- Connection
- Destination
- Session
- MessageProducer
- MessageConsumer

### Supported Message Types

- TextMessage
- ByteMessage
- ObjectMessage

### Supported Message Acknowledgment Modes

- DUPS\_OK\_ACKNOWLEDGE
- AUTO\_ACKNOWLEDGE
- CLIENT\_ACKNOWLEDGE
- *UNORDERED\_ACKNOWLEDGE*

**Note**

The *UNORDERED\_ACKNOWLEDGE* acknowledgment mode is not part of the JMS 1.1 specification. It is added by Amazon SQS to allow for a JMS client to explicitly acknowledge the message.

### JMS Defined Headers and Reserved Properties

The Amazon SQS JMS client sets the following JMS defined headers:

- JMSTDestination

- JMSMessageID
- JMSRedelivered

The Amazon SQS JMS client sets the following JMS reserved property:

- JMSXDeliveryCount

# Using Amazon SQS Message Attributes

---

Amazon Simple Queue Service (Amazon SQS) provides support for *message attributes*. Message attributes allow you to provide structured metadata items (such as timestamps, geospatial data, signatures, and identifiers) about the message. Message attributes are optional and separate from, but sent along with, the message body. This information can be used by the receiver of the message to help decide how to handle the message without having to first process the message body. Each message can have up to 10 attributes. To specify message attributes, you can use the AWS Management Console, AWS software development kits (SDKs), or query API.

## Topics

- [Message Attribute Items and Validation \(p. 32\)](#)
- [Message Attribute Data Types and Validation \(p. 33\)](#)
- [Using Message Attributes with the AWS Management Console \(p. 33\)](#)
- [Using Message Attributes with the AWS SDKs \(p. 36\)](#)
- [Using Message Attributes with the Amazon SQS Query API \(p. 38\)](#)
- [MD5 Message-Digest Calculation \(p. 39\)](#)

## Message Attribute Items and Validation

Each message attribute consists of the following items:

- **Name** – The message attribute name can contain the following characters: A-Z, a-z, 0-9, underscore(\_), hyphen(-), and period (.). The name must not start or end with a period, and it should not have successive periods. The name is case sensitive and must be unique among all attribute names for the message. The name can be up to 256 characters long. The name cannot start with "AWS." or "Amazon." (or any variations in casing) because these prefixes are reserved for use by Amazon Web Services.
- **Type** – The supported message attribute data types are String, Number, and Binary. You can also provide custom information on the type. The data type has the same restrictions on the content as the message body. The data type is case sensitive, and it can be up to 256 bytes long. For more information, see the [Message Attribute Data Types and Validation \(p. 33\)](#) section.
- **Value** – The user-specified message attribute value. For string data types, the value attribute has the same restrictions on the content as the message body. For more information, see [SendMessage](#).

Name, type, and value must not be empty or null. In addition, the message body should not be empty or null. All parts of the message attribute, including name, type, and value, are included in the message size restriction, which is currently 256 KB (262,144 bytes).

## Message Attribute Data Types and Validation

Message attribute data types identify how the message attribute values are handled by Amazon SQS. For example, if the type is a number, Amazon SQS will validate that it's a number.

Amazon SQS supports the following logical data types (with optional custom type labels):

String	.<Custom Type> (Optional)
Number	.<Custom Type> (Optional)
Binary	.<Custom Type> (Optional)

- **String** – Strings are Unicode with UTF-8 binary encoding. For a list of code values, see [http://en.wikipedia.org/wiki/ASCII#ASCII\\_printable\\_characters](http://en.wikipedia.org/wiki/ASCII#ASCII_printable_characters).
- **Number** – Numbers are positive or negative integers or floating point numbers. Numbers have sufficient range and precision to encompass most of the possible values that integers, floats, and doubles typically support. A number can have up to 38 digits of precision, and it can be between  $10^{-128}$  to  $10^{+126}$ . Leading and trailing zeroes are trimmed.
- **Binary** – Binary type attributes can store any binary data, for example, compressed data, encrypted data, or images.
- *Custom Type* – You can append a custom type label to the supported data types (String, Number, and Binary) to create custom data types. This capability is similar to *type traits* in programming languages. For example, if you have an application that needs to know which type of number is being sent in the message, then you could create custom types similar to the following: *Number.byte*, *Number.short*, *Number.int*, and *Number.float*. Another example using the binary data type is to use *Binary.gif* and *Binary.png* to distinguish among different image file types in a message or batch of messages. The appended data is optional and opaque to Amazon SQS, which means that the appended data is not interpreted, validated, or used by Amazon SQS. The Custom Type extension has the same restrictions on allowed characters as the message body.

## Using Message Attributes with the AWS Management Console

You can use the AWS Management Console to configure message attributes. In the Amazon SQS console, select a queue, click the **Queue Actions** drop-down list, and then select **Send a Message**. The console expects the user to input a Base-64-encoded value for sending a Binary type.

**Amazon Simple Queue Service Developer Guide  
Using Message Attributes with the AWS Management  
Console**

**Send a Message to MyQueue** Cancel

**Message Body** **Message Attributes**

Name:

Type: String

Value:

Enter a string value.

[What is Message Attribute?](#)

Name	Type	Values
------	------	--------

On the **Message Attributes** tab, enter a name, select the type, and enter a value for the message attribute. Optionally, you can also append custom information to the type. For example, the following screen shows the *Number* type selected with *byte* added for customization. For more information about custom data for the supported data types, see the [Message Attribute Data Types and Validation](#) (p. 33) section.

**Send a Message to MyQueue** Cancel

**Message Body** **Message Attributes**

Name:

Type: Number

Value:

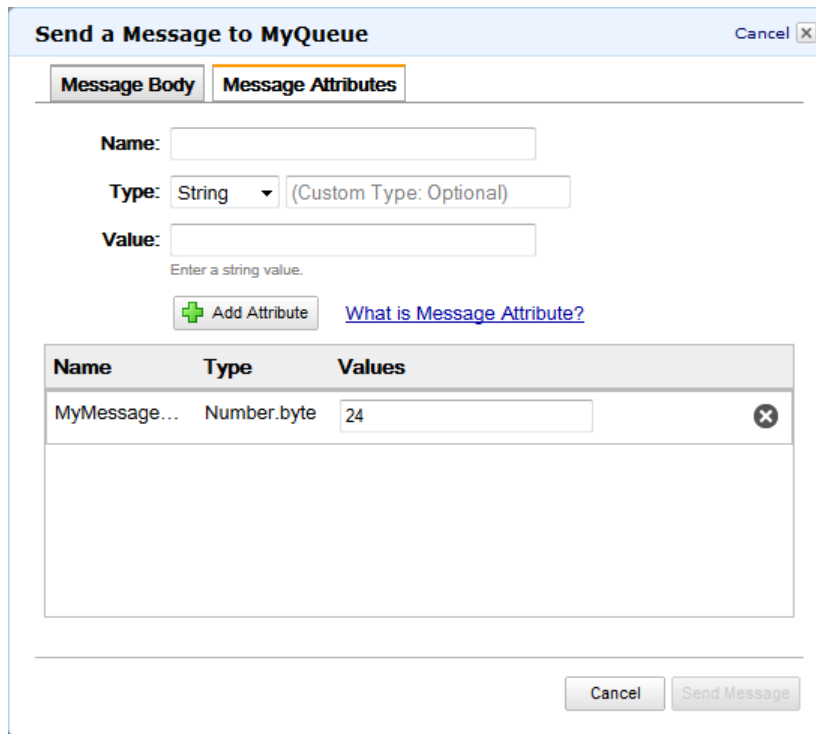
Enter a numerical value.

[What is Message Attribute?](#)

Name	Type	Values
------	------	--------

## Amazon Simple Queue Service Developer Guide Using Message Attributes with the AWS Management Console

To add an attribute, click **Add Attribute**. The attribute information will then appear in the **Name**, **Type**, and **Values** list.

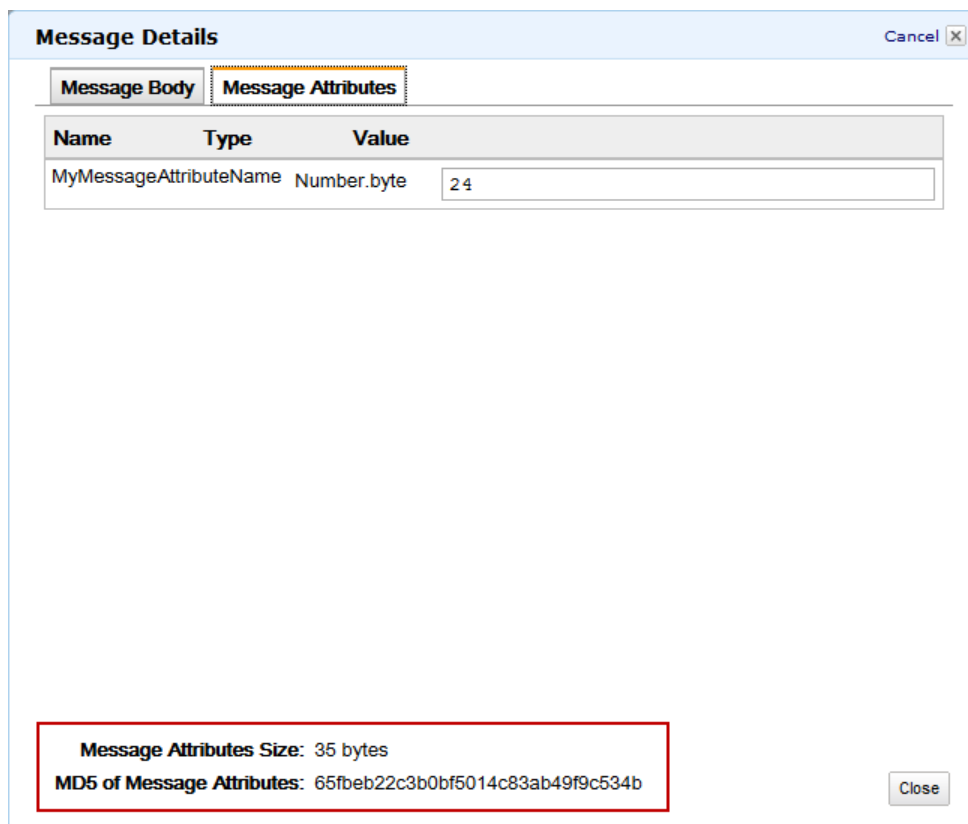


The screenshot shows the 'Send a Message to MyQueue' dialog box with the 'Message Attributes' tab selected. It features input fields for Name, Type (set to String), and Value. Below these is an 'Add Attribute' button and a link 'What is Message Attribute?'. A table below lists the attributes, with one entry: 'MyMessage...' with type 'Number.byte' and value '24'. At the bottom are 'Cancel' and 'Send Message' buttons.

Name	Type	Values
MyMessage...	Number.byte	24

You can also use the console to view information about the message attributes for received messages. In the console, select a queue, click the **Queue Actions** drop-down list, and then select **View/Delete Messages**. In the list of messages, click **Message Details** to view the information. For example, you can see the message attribute size and MD5 message digest.





## Using Message Attributes with the AWS SDKs

The [AWS SDKs](#) provide APIs in several languages for using message attributes with Amazon SQS. This section includes some Java examples that show how to work with message attributes. These examples can be integrated with the `SimpleQueueServiceSample.java` sample from the SDK for Java. `MessageBody` and `MessageAttributes` checksums are automatically calculated and compared with the data Amazon SQS returns by the latest SDK for Java. For more information about the SDK for Java, see [Getting Started with the AWS SDK for Java](#).

The following three Java examples show how to use the `MessageAttributeValue` method to set the *String*, *Number*, and *Binary* parameters for the message attributes:

### *String*

```
Map<String, MessageAttributeValue> messageAttributes = new HashMap<>();  
messageAttributes.put("attributeName", new MessageAttributeValue().withData  
Type("String").withStringValue("string-value-attribute-value"));
```

### *Number*

```
Map<String, MessageAttributeValue> messageAttributes = new HashMap<>();  
messageAttributes.put("attributeName", new MessageAttributeValue().withData
```

```
Type("Number").withStringValue("230.000000000000000001");
```

### *Binary*

```
Map<String, MessageAttributeValue> messageAttributes = new HashMap<>();  
messageAttributes.put("attributeName", new MessageAttributeValue().withData  
Type("Binary").withBinaryValue(ByteBuffer.wrap(new byte[10])));
```

The following three examples show how to use the optional custom type for the message attributes:

### *String—Custom*

```
Map<String, MessageAttributeValue> messageAttributes = new HashMap<>();  
messageAttributes.put("AccountId", new MessageAttributeValue().withData  
Type("String.AccountId").withStringValue("000123456"));
```

### *Number—Custom*

```
// NOTE Because the type is a number, the result in the receive message call  
will be 123456.  
Map<String, MessageAttributeValue> messageAttributes = new HashMap<>();  
messageAttributes.put("AccountId", new MessageAttributeValue().withDataType("Num  
ber.AccountId").withStringValue("000123456"));
```

### *Binary—Custom*

```
Map<String, MessageAttributeValue> messageAttributes = new HashMap<>();  
messageAttributes.put("PhoneIcon", new MessageAttributeValue().withDataType("Bin  
ary.JPEG").withBinaryValue(ByteBuffer.wrap(new byte[10])));
```

To send a message using one of the previous message attribute examples your code should look similar to the following:

```
SendMessageRequest request = new SendMessageRequest();  
request.withMessageBody("A test message body.");  
request.withQueueUrl("MyQueueUrlStringHere");  
request.withMessageAttributes(messageAttributes);  
sqs.sendMessage(request);
```

## Using Message Attributes with the Amazon SQS Query API

To specify message attributes with the query API, you call the [SendMessage](#), [SendMessageBatch](#), or [ReceiveMessage](#) actions.

A query API request for this example will look similar to the following:

How you structure the AUTHPARAMS depends on how you are signing your API request. For information on AUTHPARAMS in Signature Version 4, go to [Examples of Signed Signature Version 4 Requests](#).

```
POST http://sqs.us-east-1.amazonaws.com/123456789012/MyQueue
...
?Action=SendMessage
&MessageBody=This+is+a+test+message
&MessageAttribute.1.Name=test_attribute_name_1
&MessageAttribute.1.Value.StringValue=test_attribute_value_1
&MessageAttribute.1.Value.DataType=String
&MessageAttribute.2.Name=test_attribute_name_2
&MessageAttribute.2.Value.StringValue=test_attribute_value_2
&MessageAttribute.2.Value.DataType=String
&Version=2012-11-05
&Expires=2014-05-05T22%3A52%3A43PST
&AUTHPARAMS
```

The query API response should look similar to the following:

```
HTTP/1.1 200 OK
...
<SendMessageResponse>
  <SendMessageResult>
    <MD5OfMessageBody>
      fafb00f5732ab283681e124bf8747ed1
    </MD5OfMessageBody>
    <MD5OfMessageAttributes>
      3ae8f24a165a8cedc005670c81a27295
    </MD5OfMessageAttributes>
    <MessageId>
      5fea7756-0ea4-451a-a703-a558b933e274
    </MessageId>
  </SendMessageResult>
  <ResponseMetadata>
    <RequestId>
      27daac76-34dd-47df-bd01-1f6e873584a0
    </RequestId>
  </ResponseMetadata>
</SendMessageResponse>
```

When using [SendMessageBatch](#), the message attributes need to be specified on each individual message in the batch.

A query API request for this example will look similar to the following:

```
POST http://sqs.us-east-1.amazonaws.com/123456789012/MyQueue
...
?Action=SendMessageBatch
&SendMessageBatchRequestEntry.1.Id=test_msg_001
&SendMessageBatchRequestEntry.1.MessageBody=test%20message%20body%201
&SendMessageBatchRequestEntry.2.Id=test_msg_002
&SendMessageBatchRequestEntry.2.MessageBody=test%20message%20body%202
&SendMessageBatchRequestEntry.2.DelaySeconds=60
&SendMessageBatchRequestEntry.2.MessageAttribute.1.Name=test_attribute_name_1
&SendMessageBatchRequestEntry.2.MessageAttribute.1.Value.StringValue=test_at
tribute_value_1
&SendMessageBatchRequestEntry.2.MessageAttribute.1.Value.DataType=String
&Version=2012-11-05
&Expires=2014-05-05T22%3A52%3A43PST
&AUTHPARAMS
```

The query API response should look similar to the following:

```
HTTP/1.1 200 OK
...
<SendMessageBatchResponse>
<SendMessageBatchResult>
  <SendMessageBatchResultEntry>
    <Id>test_msg_001</Id>
    <MessageId>0a5231c7-8bff-4955-be2e-8dc7c50a25fa</MessageId>
    <MD5OfMessageBody>0e024d309850c78cba5eabbef77cae71</MD5OfMessageBody>
  </SendMessageBatchResultEntry>
  <SendMessageBatchResultEntry>
    <Id>test_msg_002</Id>
    <MessageId>15ee1ed3-87e7-40c1-bdaa-2e49968ea7e9</MessageId>
    <MD5OfMessageBody>7fb8146a82f95e0af155278f406862c2</MD5OfMessageBody>
    <MD5OfMessageAttributes>295c5fa15a51aae6884d1d7c1d99ca50</MD5OfMessageAt
tributes>
  </SendMessageBatchResultEntry>
</SendMessageBatchResult>
<ResponseMetadata>
  <RequestId>calad5d0-8271-408b-8d0f-1351bf547e74</RequestId>
</ResponseMetadata>
</SendMessageBatchResponse>
```

## MD5 Message-Digest Calculation

If you want to calculate the MD5 message digest for Amazon SQS message attributes and you are either using the query API or one of the AWS SDKs that does not support MD5 message digest for Amazon SQS message attributes, then you must use the following information about the algorithm to calculate the MD5 message digest of the message attributes.

### Note

Currently the AWS SDK for Java supports MD5 message digest for Amazon SQS message attributes. This is available in the `MessageMD5ChecksumHandler` class. If you are using the SDK for Java, then you do not need to use the following information.

The high-level steps of the algorithm to calculate the MD5 message digest for Amazon SQS message attributes are:

1. Sort all message attributes by name in ascending order.
2. Encode the individual parts of each attribute (name, type, and value) into a buffer.
3. Compute the message digest of the entire buffer.

**To encode a single Amazon SQS message attribute:**

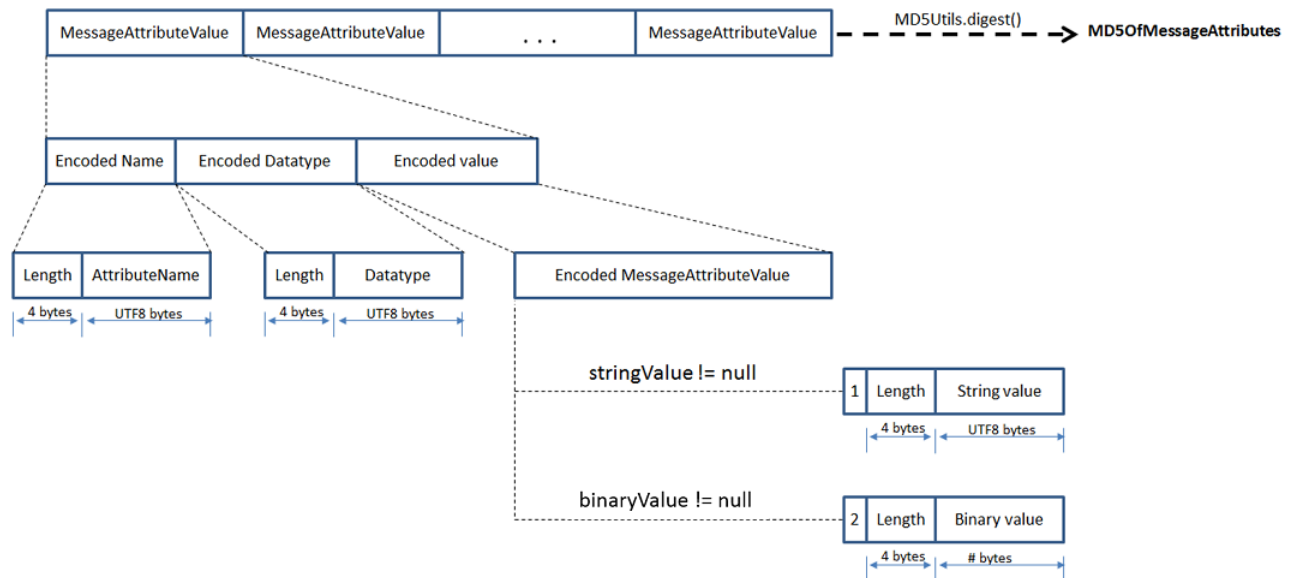
1. Encode the name (length of name [4 bytes] + UTF-8 bytes of the name).
2. Encode the type (length of type [4 bytes] + UTF-8 bytes of the type).
3. Encode the transport type (string or binary) of the value [1 byte].
  - a. For the *string* transport type, encode 1.
  - b. For the *binary* transport type, encode 2.

**Note**

The string and number logical data types use the *string* transport type. The binary logical data type uses the *binary* transport type.

4. Encode the attribute value.
  - a. For a *string* transport type, encode the attribute value (length [4 bytes] + the UTF-8 bytes of the value).
  - b. For a *binary* transport type, encode the attribute value (length [4 bytes] + use the raw bytes directly).

The following diagram shows the encoding of the MD5 message digest for a single message attribute:



# Amazon SQS Long Polling

---

## Topics

- [Enabling Long Polling with the AWS Management Console \(p. 42\)](#)
- [Enabling Long Polling with the Query API \(p. 44\)](#)

The API Version 2012-11-05 of Amazon SQS provides support for long polling.

One benefit of long polling with Amazon SQS is the reduction of the number of empty responses, when there are no messages available to return, in reply to a `ReceiveMessage` request sent to an Amazon SQS queue. Long polling allows the Amazon SQS service to wait until a message is available in the queue before sending a response. So unless the connection times out, the response to the `ReceiveMessage` request will contain at least one of the available messages (if any) and up to the maximum number requested in the `ReceiveMessage` call.

Another benefit is helping to eliminate false empty responses, where messages are available in the queue but are not included in the response. This happens when Amazon SQS uses short (standard) polling, the default behavior, where only a subset of the servers (based on a weighted random distribution) are queried to see if any messages are available to include in the response. On the other hand, when long polling is enabled, Amazon SQS queries all of the servers.

Reducing the number of empty responses and false empty responses also helps reduce your cost of using Amazon SQS.

Short polling occurs when the `WaitTimeSeconds` parameter of a `ReceiveMessage` call is set to 0. This happens in one of two ways – either the `ReceiveMessage` call sets `WaitTimeSeconds` to 0, or the `ReceiveMessage` call doesn't set `WaitTimeSeconds` and the queue attribute `ReceiveMessageWaitTimeSeconds` is 0.

### Note

A value set between 1 to 20 for the `WaitTimeSeconds` parameter for `ReceiveMessage` has priority over any value set for the queue attribute `ReceiveMessageWaitTimeSeconds`.

There are three different API action calls you can use to enable long polling in Amazon SQS, `ReceiveMessage`, `CreateQueue`, and `SetQueueAttributes`. For `ReceiveMessage`, you configure the `WaitTimeSeconds` parameter, and for `CreateQueue` and `SetQueueAttributes`, you configure the `ReceiveMessageWaitTimeSeconds` attribute.

### Important

If you decide to implement long polling with multiple queues then it is recommended that you use one thread for each queue, instead of trying to use one single thread for polling all of the

queues. When using one thread for each queue your application is able to process the messages in each of the queues as they become available, as opposed to one single thread for multiple queues where your application could be blocked from processing available messages in the other queues while waiting (up to 20 seconds) on a queue that does not have any available messages.

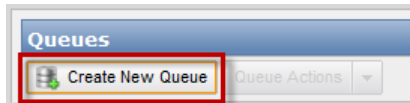
For most use cases when using long polling, you should set the timeout value to the maximum of 20 seconds. If the 20 second maximum does not work for your application, you can choose a shorter long poll timeout, down to as low as 1 second. If you are not using an AWS SDK to access Amazon SQS, or if you've specially configured your AWS SDK to have a shorter timeout, you may need to modify your Amazon SQS client to allow for longer requests or use a shorter long poll timeout.

## Enabling Long Polling with the AWS Management Console

You can enable long polling using the AWS Management Console by setting a **Receive Message Wait Time** value that is greater than 0.

### To enable long polling with the AWS Management Console for a new queue

1. Sign in to the AWS Management Console and open the Amazon SQS console at <https://console.aws.amazon.com/sqs/>.
2. Click **Create New Queue**.



3. In the **Create New Queue** dialog box, enter a name for your queue (e.g., `MyQueue`) in the **Queue Name** field.

## Amazon Simple Queue Service Developer Guide

### Enabling Long Polling with the AWS Management Console

**Create New Queue** Cancel

Please enter a name for your new queue. Queue names must be 1-80 characters in length and be composed of alphanumeric characters, hyphens (-), and underscores (\_). Your queue will be created in the US West (Oregon) region.

**Region:** US West (Oregon)

**Queue Name:** MyQueue

Configure your new queue by setting queue attributes (optional).

**Default Visibility Timeout:** 30 seconds Value must be between 0 seconds and 12 hours.

**Message Retention Period:** 4 days Value must be between 1 minute and 14 days.

**Maximum Message Size:** 256 KB Value must be between 1 and 256 KB.

**Delivery Delay:** 0 seconds Value must be between 0 seconds and 15 minutes.

**Receive Message Wait Time:** 0 seconds Value must be between 0 and 20 seconds.

Cancel Create Queue

4. Enter a positive integer value, from 1 to 20 seconds, for **Receive Message Wait Time**. You can leave the default value settings for the remaining fields or enter new values.

**Create New Queue** Cancel

Please enter a name for your new queue. Queue names must be 1-80 characters in length and be composed of alphanumeric characters, hyphens (-), and underscores (\_). Your queue will be created in the US East (Virginia) region.

**Region:** US West (Oregon)

**Queue Name:** MyQueue

Configure your new queue by setting queue attributes (optional).

**Default Visibility Timeout:** 30 seconds Value must be between 0 seconds and 12 hours.

**Message Retention Period:** 4 days Value must be between 1 minute and 14 days.

**Maximum Message Size:** 64 KB Value must be between 1 and 64 KB.

**Delivery Delay:** 0 seconds Value must be between 0 seconds and 15 minutes.

**Receive Message Wait Time:** 10 seconds Value must be between 0 and 20 seconds.

Cancel Create Queue

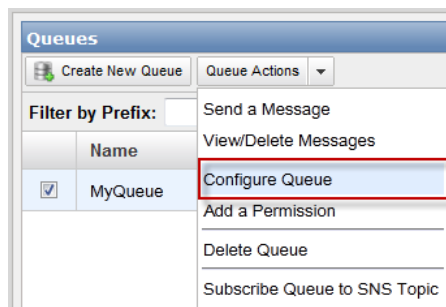
5. Click **Create Queue**.

You can use the AWS Management Console to change the **Receive Message Wait Time** setting for an existing queue by selecting the **Configure Queue** action with an existing queue highlighted.

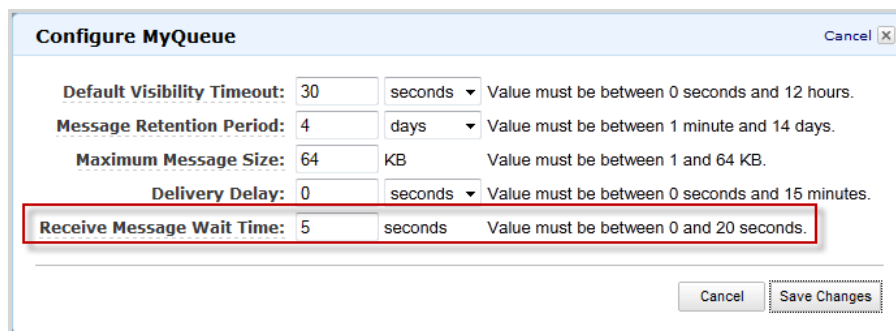
#### To set a new Receive Message Wait Time value for an existing queue

1. Select the **Configure Queue** action with an existing queue highlighted.





2. Change the value of **Receive Message Wait Time** to a positive integer value.



3. Click **Save Changes**.

## Enabling Long Polling with the Query API

The following Query API example enables long polling by calling the `ReceiveMessage` action with the `WaitTimeSeconds` parameter set to 10 seconds. For more information, go to [ReceiveMessage](#) in the *Amazon Simple Queue Service API Reference*.

How you structure the AUTHPARAMS depends on how you are signing your API request. For information on AUTHPARAMS in Signature Version 4, go to [Examples of Signed Signature Version 4 Requests](#).

```
http://sqs.us-east-1.amazonaws.com/123456789012/testQueue/  
?Action=ReceiveMessage  
&WaitTimeSeconds=10  
&MaxNumberOfMessages=5  
&VisibilityTimeout=15  
&AttributeName=All;  
&Version=2012-11-05  
&Expires=2013-10-25T22%3A52%3A43PST  
&AUTHPARAMS
```

The following example shows another option for enabling long polling. Here, the `ReceiveMessageWaitTimeSeconds` attribute for the `SetQueueAttributes` action is set to 20 seconds. For more information, go to [SetQueueAttributes](#) in the *Amazon Simple Queue Service API Reference*.

```
http://sqs.us-east-1.amazonaws.com/123456789012/testQueue/  
?Action=SetQueueAttributes  
&Attribute.Name=ReceiveMessageWaitTimeSeconds
```

```
&Attribute.Value=20  
&Version=2012-11-05  
&Expires=2013-10-25T22%3A52%3A43PST  
&AUTHPARAMS
```

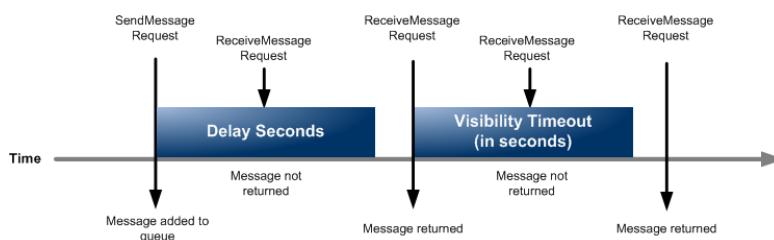
# Amazon SQS Delay Queues

## Topics

- [Creating Delay Queues with the AWS Management Console \(p. 47\)](#)
- [Creating Delay Queues with the Query API \(p. 49\)](#)

Delay queues allow you to postpone the delivery of new messages in a queue for a specific number of seconds. If you create a delay queue, any message that you send to that queue will be invisible to consumers for the duration of the delay period. You can use `CreateQueue` to create a delay queue by setting the `DelaySeconds` attribute to any value between 0 and 900 (15 minutes). You can also turn an existing queue into a delay queue by using `SetQueueAttributes` to set the queue's `DelaySeconds` attribute.

Delay queues are similar to visibility timeouts in that both features make messages unavailable to consumers for a specific period of time. The difference between delay queues and visibility timeouts is that for delay queues, a message is hidden when it is first added to the queue, whereas for visibility timeouts, a message is hidden only after a message is retrieved from the queue. The following figure illustrates the relationship between delay queues and visibility timeouts.



## Note

There is a 120,000 limit for the number of inflight messages per queue. Messages are inflight after they have been received from the queue by a consuming component, but have not yet been deleted from the queue. If you reach the 120,000 limit, you will receive an `OverLimit` error message from Amazon SQS. To help avoid reaching the limit, you should delete the messages from the queue after they have been processed. You can also increase the number of queues you use to process the messages.

To set delay seconds on individual messages, rather than for an entire queue, use message timers. If you send a message with a message timer, Amazon SQS uses the message timer's delay seconds value instead of the delay queue's delay seconds value. For more information, see [Amazon SQS Message Timers \(p. 50\)](#).

### Important

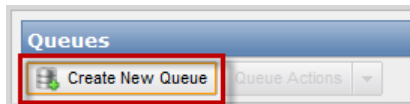
All Amazon SQS queues have delay functionality enabled. You can turn any queue that you create using the 2008-01-01 or 2009-02-01 API versions into a delay queue. Message timers are available in the 2011-10-01 API version and later API versions. If you want to apply specific delay values to individual messages, you must use the 2011-10-01 API version or later API versions.

## Creating Delay Queues with the AWS Management Console

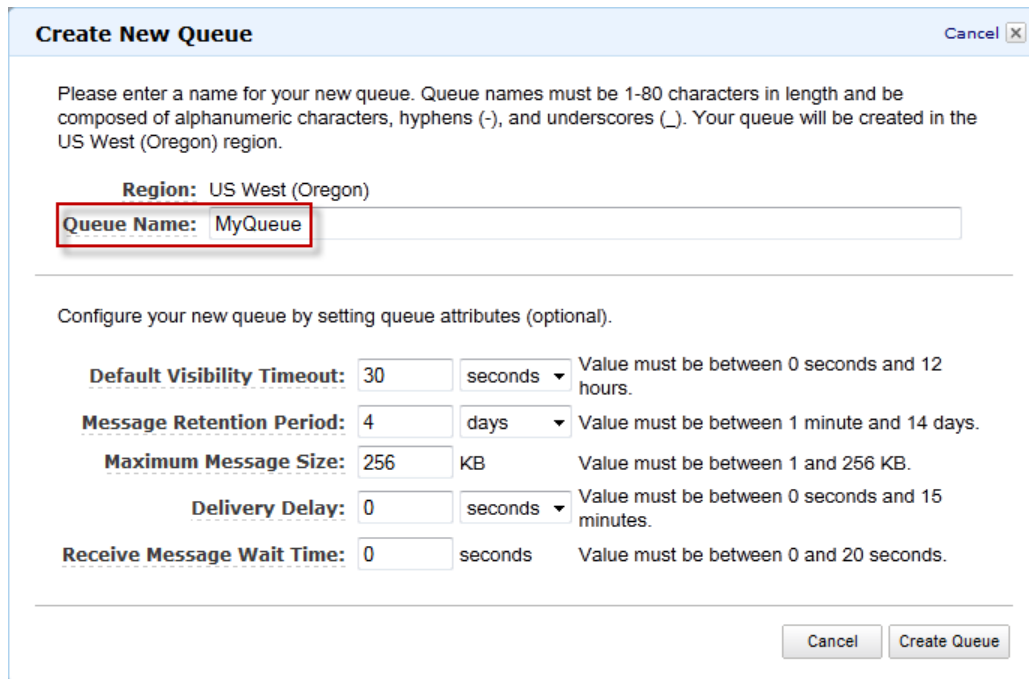
You can create a delay queue using the AWS Management Console by setting a **Delivery Delay** value that is greater than 0.

### To create a delay queue with the AWS Management Console

1. Sign in to the AWS Management Console and open the Amazon SQS console at <https://console.aws.amazon.com/sqs/>.
2. Click **Create New Queue**.



3. In the **Create New Queue** dialog box, enter a name for your queue (e.g., `MyQueue`) in the **Queue Name** field.

A screenshot of the 'Create New Queue' dialog box. The dialog has a title bar with 'Create New Queue' and a 'Cancel' button. Below the title bar is a text area with instructions: 'Please enter a name for your new queue. Queue names must be 1-80 characters in length and be composed of alphanumeric characters, hyphens (-), and underscores (\_). Your queue will be created in the US West (Oregon) region.' Below this is a 'Region' label with the value 'US West (Oregon)'. The 'Queue Name' field is highlighted with a red box and contains the text 'MyQueue'. Below the name field is a section titled 'Configure your new queue by setting queue attributes (optional)'. This section contains five rows of attributes, each with a label, a text input field, a unit dropdown, and a description: 'Default Visibility Timeout: 30 seconds Value must be between 0 seconds and 12 hours.', 'Message Retention Period: 4 days Value must be between 1 minute and 14 days.', 'Maximum Message Size: 256 KB Value must be between 1 and 256 KB.', 'Delivery Delay: 0 seconds Value must be between 0 seconds and 15 minutes.', and 'Receive Message Wait Time: 0 seconds Value must be between 0 and 20 seconds.' At the bottom right of the dialog are 'Cancel' and 'Create Queue' buttons.

4. Enter a positive integer value (e.g., 30) for the Delivery Delay attribute. You can leave the default value settings for the remaining fields or enter new values.

## Amazon Simple Queue Service Developer Guide Creating Delay Queues with the AWS Management Console

**Create New Queue** Cancel

Please enter a name for your new queue. Queue names must be 1-80 characters in length and be composed of alphanumeric characters, hyphens (-), and underscores (\_). Your queue will be created in the US East (Virginia) region.

**Region:** US West (Oregon)

**Queue Name:** MyQueue

Configure your new queue by setting queue attributes (optional).

**Default Visibility Timeout:** 30 seconds Value must be between 0 seconds and 12 hours.

**Message Retention Period:** 4 days Value must be between 1 hour and 14 days.

**Maximum Message Size:** 64 KB Value must be between 1 and 64 KB.

**Delivery Delay:** 30 seconds Value must be between 0 seconds and 15 minutes.

Cancel Create Queue

5. Click **Create Queue**.

You can use the AWS Management Console to change the **Delivery Delay** setting for an existing queue by selecting the **Configure Queue** action with an existing queue highlighted.

### To set a new delivery delay value for an existing queue

1. Select the **Configure Queue** action with an existing queue highlighted.

**Queues**

Create New Queue Queue Actions

Filter by Prefix: Send a Message

Name	Actions
<input checked="" type="checkbox"/> MyQueue	View/Delete Messages <b>Configure Queue</b> Add a Permission Delete Queue Subscribe Queue to SNS Topic

2. Change the value of **Delivery Delay** to a positive integer value.

**Configure MyQueue** Cancel

**Default Visibility Timeout:** 30 seconds Value must be between 0 seconds and 12 hours.

**Message Retention Period:** 4 days Value must be between 1 hour and 14 days.

**Maximum Message Size:** 64 KB Value must be between 1 and 64 KB.

**Delivery Delay:** 0 seconds Value must be between 0 seconds and 15 minutes.

Cancel Save Changes

3. Click **Save Changes**.

## Creating Delay Queues with the Query API

The following Query API example calls the `CreateQueue` action to create a delay queue that hides each message from consumers for the first 45 seconds that the message is in the queue.

How you structure the AUTHPARAMS depends on how you are signing your API request. For information on AUTHPARAMS in Signature Version 4, go to [Examples of Signed Signature Version 4 Requests](#).

```
http://sqs.us-east-1.amazonaws.com/  
?Action=CreateQueue  
&QueueName=testQueue  
&Attribute.1.Name=DelaySeconds  
&Attribute.1.Value=45  
&Version=2011-10-01  
&Expires=2011-10-20T22%3A52%3A43PST  
&AUTHPARAMS
```

You can also change an existing queue into a delay queue by changing the `DelaySeconds` attribute from its default value of 0 to a positive integer value that is less than or equal to 900. The following example calls `SetQueueAttributes` to set the `DelaySeconds` attribute of a queue named `testQueue` to 45 seconds.

```
http://sqs.us-east-1.amazonaws.com/123456789012/testQueue/  
?Action=SetQueueAttributes  
&Attribute.Name=DelaySeconds  
&Attribute.Value=45  
&Version=2011-10-01  
&Expires=2011-10-20T22%3A52%3A43PST  
&AUTHPARAMS
```

# Amazon SQS Message Timers

---

## Topics

- [Creating Message Timers Using the Console \(p. 50\)](#)
- [Creating Message Timers Using the Query API \(p. 52\)](#)

Amazon SQS message timers allow you to specify an initial invisibility period for a message that you are adding to a queue. For example, if you send a message with the *DelaySeconds* parameter set to 45, the message will not be visible to consumers for the first 45 seconds that the message resides in the queue. The default value for *DelaySeconds* is 0.

### Note

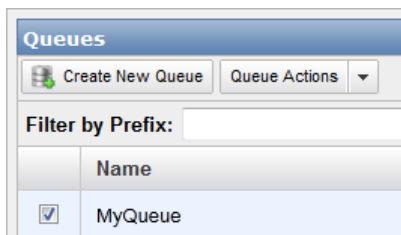
There is a 120,000 limit for the number of inflight messages per queue. Messages are inflight after they have been received from the queue by a consuming component, but have not yet been deleted from the queue. If you reach the 120,000 limit, you will receive an *OverLimit* error message from Amazon SQS. To help avoid reaching the limit, you should delete the messages from the queue after they have been processed. You can also increase the number of queues you use to process the messages.

To set a delay period that applies to all messages in a queue, use delay queues. For more information, see [Amazon SQS Delay Queues \(p. 46\)](#). A message timer setting for an individual message overrides any *DelaySeconds* value that applies to the entire delay queue.

## Creating Message Timers Using the Console

### To send a message with a message timer using the AWS Management Console

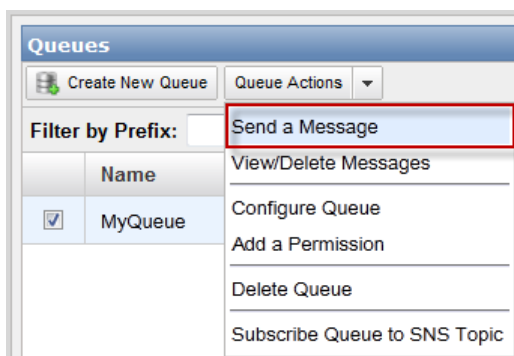
1. Sign in to the AWS Management Console and open the Amazon SQS console at <https://console.aws.amazon.com/sqs/>.
2. Select a queue.



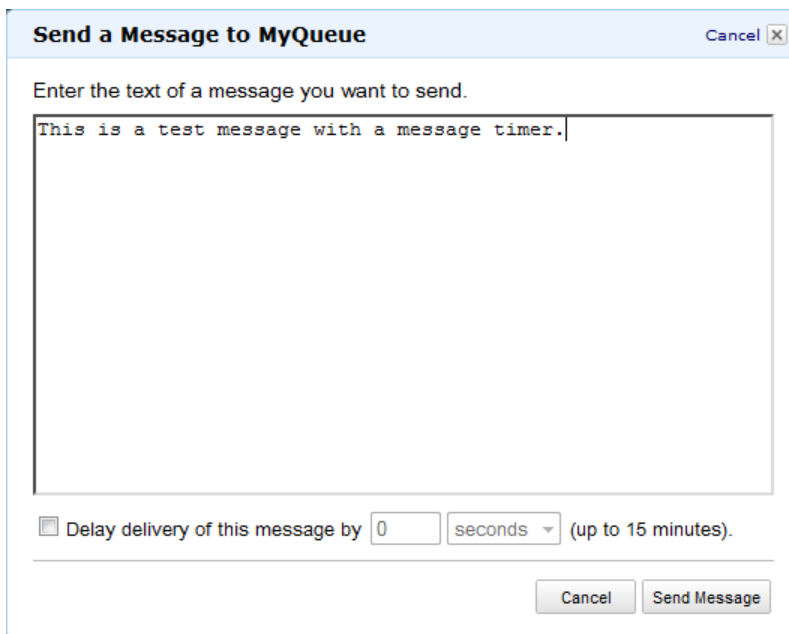
3. Select **Send a Message** from the **Queue Actions** drop-down list.

**Note**

The **Queue Actions** drop-down list is available only if a queue is selected.



4. In the **Send a Message to MyQueue** dialog box, enter a message (e.g., This is a test message with a message timer.).



5. Enter a delay value (e.g., 30) in the **Delay delivery of this message by** text box.



**Send a Message to MyQueue** Cancel

Enter the text of a message you want to send.

This is a test message with a message timer.

Delay delivery of this message by  seconds (up to 15 minutes).

Cancel Send Message

6. Click **Send Message**.
7. In the **Send a Message to MyQueue** confirmation box click **Close**.

**Send a Message to MyQueue** Cancel

Your message has been sent and is ready to be received.

Note: It may take up to 60 seconds for the *Messages Available* column to update.

**Sent Message Attributes:**

**Message Identifier:** d546048e-4afd-41ea-8435-2d33f907ee94

**MD5 of Body:** 6a1559560f67c5e7a7d5d838bf0272ee

Send Another Message Close

## Creating Message Timers Using the Query API

The following Query API example applies a 45 second initial visibility delay for a single message sent with `SendMessage`.

How you structure the `AUTHPARAMS` depends on how you are signing your API request. For information on `AUTHPARAMS` in Signature Version 4, go to [Examples of Signed Signature Version 4 Requests](#).

```
http://sqs.us-east-1.amazonaws.com/123456789012/testQueue/  
?Action=SendMessage  
&MessageBody=This+is+a+test+message  
&Attribute.Name=DelaySeconds  
&Attribute.Value=45  
&Version=2011-10-01  
&Expires=2011-10-18T22%3A52%3A43PST  
&AUTHPARAMS
```

You can also use the Query API `SendMessageBatch` action to send up to ten messages with message timers. You can assign a different `DelaySeconds` value to each message or assign no value at all. If you do not set a value for `DelaySeconds`, the message might still be subject to a delay if you are adding the message to a delay queue. For more information about delay queues, see [Amazon SQS Delay Queues \(p. 46\)](#). The following example uses `SendMessageBatch` to send three messages: one message without a message timer and two messages with different values for `DelaySeconds`.

```
http://sqs.us-east-1.amazonaws.com/123456789012/testQueue/  
?Action=SendMessageBatch  
&SendMessageBatchRequestEntry.1.Id=test_msg_no_message_timer  
&SendMessageBatchRequestEntry.1.MessageBody=test%20message%20body%201  
&SendMessageBatchRequestEntry.2.Id=test_msg_delay_45_seconds  
&SendMessageBatchRequestEntry.2.MessageBody=test%20message%20body%202  
&SendMessageBatchRequestEntry.2.DelaySeconds=45  
&SendMessageBatchRequestEntry.3.Id=test_msg_delay_2_minutes  
&SendMessageBatchRequestEntry.3.MessageBody=test%20message%20body%203  
&SendMessageBatchRequestEntry.3.DelaySeconds=120  
&Version=2011-10-01  
&Expires=2011-10-18T22%3A52%3A43PST  
&AUTHPARAMS
```

# Amazon SQS Batch API Actions

---

## Topics

- [Maximum Message Size for SendMessageBatch \(p. 54\)](#)

In the 2009-02-01 API version of Amazon SQS, only one action—`ReceiveMessage`—supports batch processing, i.e., processing more than one message with a single call. With the 2011-10-01 API version, Amazon SQS adds batch functionality for sending messages, deleting messages, and changing message visibility timeout values. To send up to ten messages at once, use the `SendMessageBatch` action. To delete up to ten messages with one API call, use the `DeleteMessageBatch` action. To change the visibility timeout value for up to ten messages, use the `ChangeMessageVisibilityBatch` action.

To use the new batch actions, you must use either the Query API or a Software Development Kit (SDK) that supports the new batch actions. Check your specific SDK's documentation to see whether it supports the new Amazon SQS batch actions. The Amazon SQS console does not currently support the batch API actions.

For details and examples of the three batch API actions, go to the *Amazon Simple Queue Service API Reference*:

- [ChangeMessageVisibilityBatch](#)
- [DeleteMessageBatch](#)
- [SendMessageBatch](#)

## Maximum Message Size for SendMessageBatch

You can send a message as large as 262,144 bytes (256 KB) with `SendMessageBatch`. However, the total size of all the messages that you send in a single call to `SendMessageBatch` cannot exceed 262,144 bytes (256 KB).

# Making API Requests

---

## Topics

- [Endpoints \(p. 56\)](#)
- [Making Query Requests \(p. 57\)](#)
- [Making SOAP Requests \(p. 59\)](#)
- [Request Authentication \(p. 60\)](#)
- [Responses \(p. 68\)](#)
- [Shared Queues \(p. 70\)](#)
- [Programming Languages \(p. 72\)](#)

This section describes how to make requests to Amazon SQS. The topics acquaint you with the basic differences between the interfaces, the components of a request, how to authenticate a request, and the content of responses.

We also provide SDKs that enable you to access Amazon SQS from your preferred programming language. The SDKs contain functionality that automatically takes care of tasks such as:

- Cryptographically signing your service requests
- Retrying requests
- Handling error responses

For a list of available SDKs, go to [Tools for Amazon Web Services](#)

## Endpoints

For information about this product's regions and endpoints, go to [Regions and Endpoints](#) in the Amazon Web Services General Reference.

For example, to create a queue in Europe, you would generate a Query request similar to the following:

How you structure the AUTHPARAMS depends on how you are signing your API request. For information on AUTHPARAMS in Signature Version 4, go to [Examples of Signed Signature Version 4 Requests](#).

```
http://sqs.eu-west-1.amazonaws.com/  
?Action=CreateQueue  
&DefaultVisibilityTimeout=40  
&QueueName=testQueue  
&Version=2009-02-01  
&AUTHPARAMS
```

Each Amazon SQS endpoint is entirely independent. For example, if you have two queues called "MyQueue," one in sqs.us-east-1.amazonaws.com and one in sqs.eu-west-1.amazonaws.com, they are completely independent and do not share any data.

# Making Query Requests

## Topics

- [Structure of a GET Request \(p. 57\)](#)
- [Structure of a POST Request \(p. 58\)](#)
- [Related Topics \(p. 59\)](#)

Amazon SQS supports Query requests for calling service actions. Query requests are simple HTTP or HTTPS requests, using the GET or POST method. Query requests must contain an *Action* parameter to indicate the action to be performed. The response is an XML document that conforms to a schema.

## Structure of a GET Request

This guide presents the Amazon SQS GET requests as URLs, which can be used directly in a browser. The URL consists of:

- **Endpoint**—The resource the request is acting on (in the case of Amazon SQS, the endpoint is a queue)
- **Action**—The action you want to perform on the endpoint; for example: sending a message
- **Parameters**—Any request parameters

The following is an example GET request to send a message to an Amazon SQS queue.

How you structure the AUTHPARAMS depends on how you are signing your API request. For information on AUTHPARAMS in Signature Version 4, go to [Examples of Signed Signature Version 4 Requests](#).

```
http://sqs.us-east-1.amazonaws.com/123456789012/queue1?Action=SendMessage&MessageBody=Your%20Message%20Text&Version=2012-11-05&AUTHPARAMS
```

### Important

Because the GET requests are URLs, you must URL encode the parameter values. For example, in the preceding example request, the value for the *MessageBody* parameter is actually *Your Message Text*. However, spaces are not allowed in URLs, so each space is URL encoded as "%20". The rest of the example has not been URL encoded to make it easier for you to read.

To make the GET examples even easier to read, this guide presents them in the following parsed format.

```
http://sqs.us-east-1.amazonaws.com/123456789012/queue1
?Action=SendMessage
&MessageBody=Your%20Message%20Text
&Version=2012-11-05
&Expires=2011-10-15T12:00:00Z
&AUTHPARAMS
```

### Note

In the example Query requests we present in this guide, we use a false AWS Access Key ID and false signature, each with `EXAMPLE` appended. We do this to indicate that you shouldn't expect the signature in the example to be accurate based on the request parameters presented in the example. The one exception to this is in the instructions for creating Query request signatures. The example there shows a real signature based on a particular AWS Access Key ID we specify and the request parameters in the example (for more information, see [Query Request Authentication \(p. 67\)](#)).

In Amazon SQS, all parameters except *MessageBody* always have values that have no spaces. The value you provide for *MessageBody* in [SendMessage](#) requests can have spaces. In this guide, any example *SendMessage* Query requests with a *MessageBody* that includes spaces is displayed with the spaces URL encoded (as %20). For clarity, the rest of the URL is not displayed in a URL encoded format.

The first line represents the *endpoint* of the request. This is the resource the request acts on. The preceding example acts on a queue, so the request's endpoint is the queue's identifier, known as the *queue URL*. For more details about the queue URL, see [Queue URLs](#) (p. 5).

After the endpoint is a question mark (?), which separates the endpoint from the parameters. Each parameter is separated by an ampersand (&).

The *Action* parameter indicates the action to perform (for a list of the actions, see [API Actions](#) in the Amazon SQS API Reference). For a list of the other parameters that are common to all Query requests, see [Common Parameters](#) in the Amazon SQS API Reference.

## Structure of a POST Request

Amazon SQS also accepts POST requests. With a POST request, you send the query parameters as a form in the HTTP request body as described in the following procedure.

How you structure the AUTHPARAMS depends on how you are signing your API request. For information on AUTHPARAMS in Signature Version 4, go to [Examples of Signed Signature Version 4 Requests](#).

### To create a POST request

1. Assemble the query parameter names and values into a form.

This means you put the parameters and values together like you would for a GET request (with an ampersand separating each name-value pair). The following example shows a *SendMessage* request with the line breaks we use in this guide to make the information easier to read.

```
Action=SendMessage
&MessageBody=Your Message Text
&Version=2012-11-05
&Expires=2011-10-15T12:00:00Z
&AUTHPARAMS
```

2. Form-URL-encode the form according to the *Form Submission* section of the HTML specification (for more information, go to [http://www.w3.org/MarkUp/html-spec/html-spec\\_toc.html#SEC8.2.1](http://www.w3.org/MarkUp/html-spec/html-spec_toc.html#SEC8.2.1)).

```
Action=SendMessage
&MessageBody=Your+Message+Text
&Version=2012-11-05
&Expires=2011-10-15T12%3A00%3A00Z
&AUTHPARAMS
```

3. Add the request signature to the form (for more information, see [Query Request Authentication](#) (p. 67)).

```
Action=SendMessage
&MessageBody=Your+Message+Text
&Version=2012-11-05
&Expires=2011-10-15T12%3A00%3A00Z
&AUTHPARAMS
```

4. Provide the resulting form as the body of the POST request.
5. Include the `Content-Type` HTTP header with the value set to `application/x-www-form-urlencoded`.

The following example shows the final POST request.

```
POST /queue1 HTTP/1.1
Host: sqs.us-east-1.amazonaws.com
Content-Type: application/x-www-form-urlencoded

Action=SendMessage
&MessageBody=Your+Message+Text
&Version=2012-11-05
&Expires=2011-10-15T12%3A00%3A00Z
&AUTHPARAMS
```

Amazon SQS requires no other HTTP headers in the request besides `Content-Type`. The authentication signature you provide is the same signature you would provide if you sent a GET request (for information about the signature, see [Query Request Authentication \(p. 67\)](#)).

**Note**

Your HTTP client typically adds other items to the HTTP request as required by the version of HTTP the client uses. We don't include those additional items in the examples in this guide.

## Related Topics

- [Query Request Authentication \(p. 67\)](#)
- [Responses \(p. 68\)](#)

## Making SOAP Requests

**Important**

As of August 8, 2011, Amazon SQS no longer supports SOAP requests.



# Request Authentication

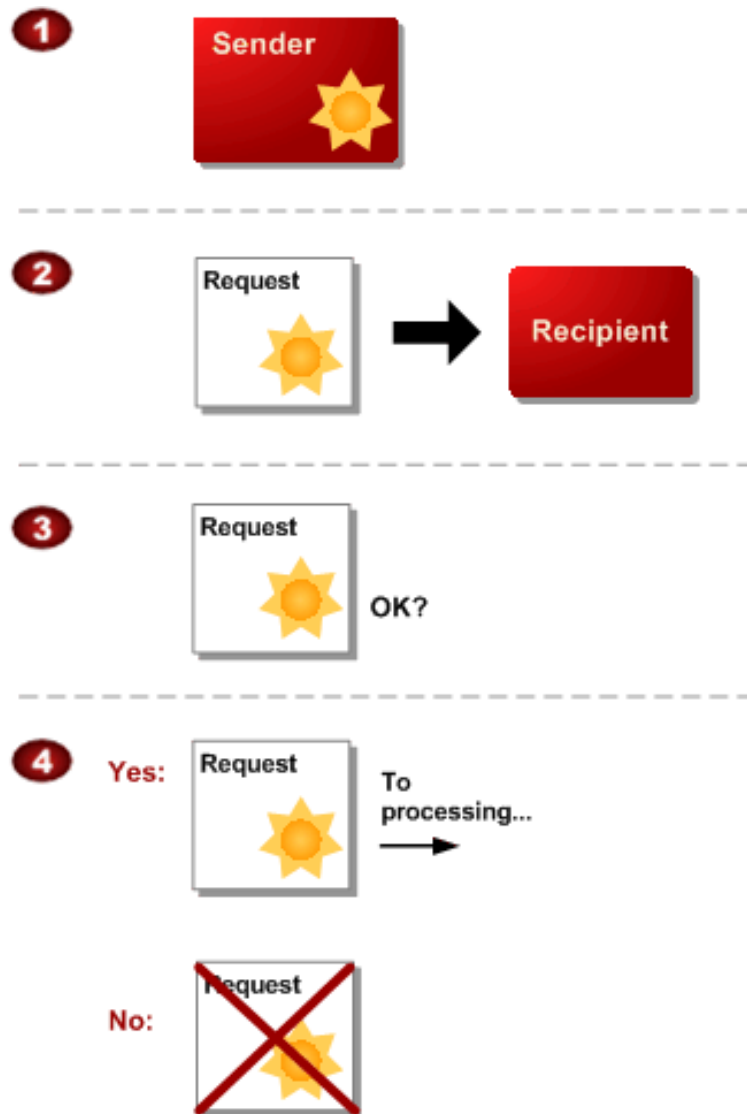
## Topics

- [What Is Authentication? \(p. 60\)](#)
- [Your AWS Account \(p. 62\)](#)
- [Your Access Keys \(p. 62\)](#)
- [HMAC-SHA Signatures \(p. 62\)](#)
- [Query Request Authentication \(p. 67\)](#)

The topics in this section describe how Amazon SQS authenticates your requests. In this section you can learn about the basics of authentication, how your AWS account and access keys are used to support authentication, and how to create an HMAC-SHA1 signature. This section also covers the request authentication requirements for Query requests.

## What Is Authentication?

Authentication is a process for identifying and verifying who is sending a request. The following diagram shows a simplified version of an authentication process.



### General Process of Authentication

1	The sender obtains the necessary credential.
2	The sender sends a request with the credential to the recipient.
3	The recipient uses the credential to verify the sender truly sent the request.
4	If yes, the recipient processes the request. If no, the recipient rejects the request and responds accordingly.

During authentication, AWS verifies both the identity of the sender and whether the sender is registered to use services offered by AWS. If either test fails, the request is not processed further.

For further discussion of authentication, go to the [techencyclopedia.com](http://techencyclopedia.com) entry for [authentication](#). For definitions of common industry terms related to authentication, go to the [RSA Laboratories Glossary](#).

The subsequent sections describe how Amazon SQS implements authentication to protect your data.

## Your AWS Account

To access any web services offered by AWS, you must first create an AWS account at <http://aws.amazon.com>. An AWS account is simply an Amazon.com account that is enabled to use AWS products; you can use an existing Amazon.com account login and password when creating the AWS account.

Alternately, you could create a new AWS-enabled Amazon.com account by using a new login and password. The e-mail address you provide as the account login must be valid. You'll be asked to provide a credit card or other payment method to cover the charges for any AWS products you use.

From your AWS account you can view your AWS account activity and view usage reports.

For more information, see [Creating an AWS Account](#) in the Amazon Simple Queue Service Getting Started Guide.

### Related Topics

- [Your Access Keys \(p. 62\)](#)

## Your Access Keys

For API access, you need an access key ID and secret access key. Use IAM user access keys instead of AWS root account access keys. IAM lets you securely control access to AWS services and resources in your AWS account. For more information about creating access keys, see [How Do I Get Security Credentials?](#) in the *AWS General Reference*.

### Related Topics

- [HMAC-SHA Signatures \(p. 62\)](#)
- [Query Request Authentication \(p. 67\)](#)

## HMAC-SHA Signatures

### Topics

- [Required Authentication Information \(p. 62\)](#)
- [Basic Authentication Process \(p. 64\)](#)
- [About the String to Sign \(p. 65\)](#)
- [About the Time Stamp \(p. 65\)](#)
- [Java Sample Code for Base64 Encoding \(p. 66\)](#)
- [Java Sample Code for Calculating HMAC-SHA1 Signatures \(p. 66\)](#)

The topics in this section describe how Amazon SQS uses HMAC-SHA signatures to authenticate Query requests.

### Required Authentication Information

When accessing Amazon SQS using the Query API, you must provide the following items so the request can be authenticated:

- **AWS Access Key ID**—Your AWS account is identified by your Access Key ID, which AWS uses to look up your Secret Access Key.
- **Signature**—Each request must contain a valid HMAC-SHA request signature, or the request is rejected. You calculate the request signature by using your Secret Access Key, which is a shared secret known only to you and AWS.
- **Date**—Each request must contain the time stamp of the request. You can provide an expiration date and time for the request instead of or in addition to the time stamp.

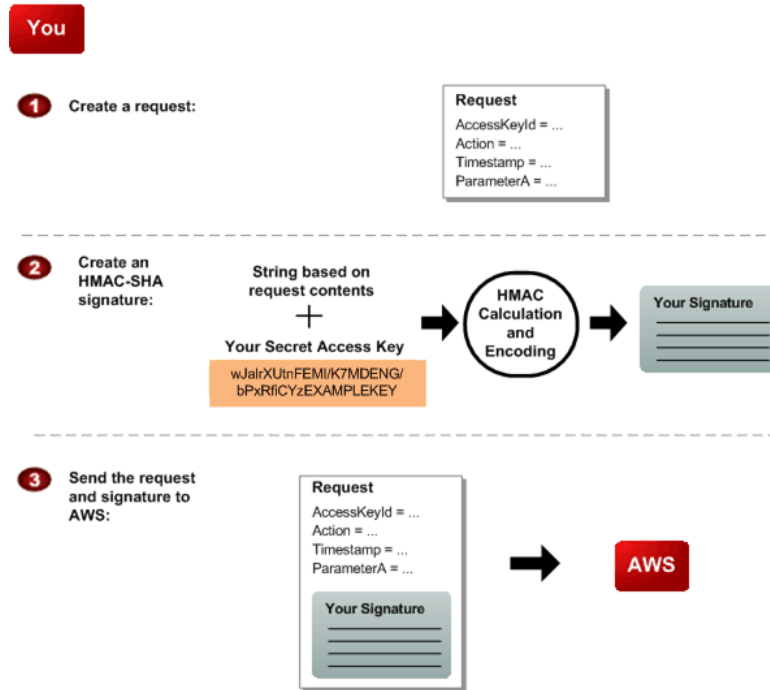
## **Related Topics**

- [Your Access Keys \(p. 62\)](#)

## Basic Authentication Process

Following is the series of tasks required to authenticate requests to AWS using an HMAC-SHA request signature. It is assumed you have already created an AWS account and created an Access Key ID and Secret Access Key. For more information about those, see [Your AWS Account \(p. 62\)](#) and [Your Access Keys \(p. 62\)](#).

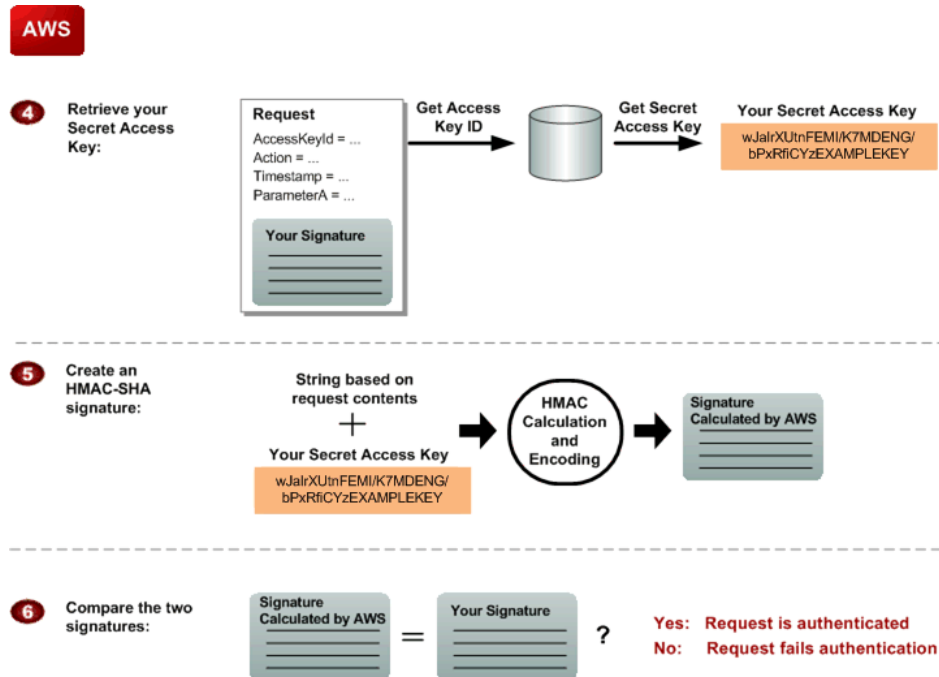
You perform the first three tasks.



### Process for Authentication: Tasks You Perform

1	You construct a request to AWS.
2	You calculate a keyed-hash message authentication code (HMAC-SHA) signature using your Secret Access Key (for information about HMAC, go to <a href="http://www.faqs.org/rfcs/rfc2104.html">http://www.faqs.org/rfcs/rfc2104.html</a> )
3	You include the signature and your Access Key ID in the request, and then send the request to AWS.

AWS performs the next three tasks.



**Process for Authentication: Tasks AWS Performs**

4	AWS uses the Access Key ID to look up your Secret Access Key.
5	AWS generates a signature from the request data and the Secret Access Key using the same algorithm you used to calculate the signature you sent in the request.
6	If the signature generated by AWS matches the one you sent in the request, the request is considered authentic. If the comparison fails, the request is discarded, and AWS returns an error response.

## About the String to Sign

Each AWS request you send must include an HMAC-SHA request signature calculated with your Secret Access Key. The details are covered in [Query Request Authentication \(p. 67\)](#).

## About the Time Stamp

The time stamp (or expiration time) you use in the request must be a `dateTime` object, with the complete date plus hours, minutes, and seconds (for more information, go to <http://www.w3.org/TR/xmlschema-2/#dateTime>). For example: 2007-01-31T23:59:59Z. Although it is not required, we recommend you provide the time stamp in the Coordinated Universal Time (Greenwich Mean Time) time zone.

If you specify a time stamp (instead of an expiration time), the request automatically expires 15 minutes after the time stamp (in other words, AWS does not process a request if the request time stamp is more than 15 minutes earlier than the current time on AWS servers). Make sure your server's time is set correctly.

### Important

If you are using .NET you must not send overly specific time stamps, due to different interpretations of how extra time precision should be dropped. To avoid overly specific time stamps, manually construct `dateTime` objects with no more than millisecond precision.

## Java Sample Code for Base64 Encoding

Request signatures must be base64 encoded. The following Java sample code shows how to perform base64 encoding.

```
package amazon.webservices.common;
/**
 * This class defines common routines for encoding data in AWS requests.
 */
public class Encoding {
/**
 * Performs base64-encoding of input bytes.
 *
 * @param rawData * Array of bytes to be encoded.
 * @return * The base64 encoded string representation of rawData.
 */
public static String EncodeBase64(byte[] rawData) {
return Base64.encodeBytes(rawData);
}
}
```

## Java Sample Code for Calculating HMAC-SHA1 Signatures

The following Java code sample shows how to calculate an HMAC request signature.

```
package amazon.webservices.common;

import java.security.SignatureException;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;

/**
 * This class defines common routines for generating
 * authentication signatures for AWS requests.
 */
public class Signature {
private static final String HMAC_SHA1_ALGORITHM = "HmacSHA1";

/**
 * Computes RFC 2104-compliant HMAC signature.
 * * @param data
 * The data to be signed.
 * @param key
 * The signing key.
 * @return
 * The Base64-encoded RFC 2104-compliant HMAC signature.
 * @throws
 * java.security.SignatureException when signature generation fails
 */
public static String calculateRFC2104HMAC(String data, String key)
```

```
throws java.security.SignatureException
{
String result;
try {

// get an hmac_sha1 key from the raw key bytes
SecretKeySpec signingKey = new SecretKeySpec(key.getBytes(), HMAC_SHA1_AL
GORITHM);

// get an hmac_sha1 Mac instance and initialize with the signing key
Mac mac = Mac.getInstance(HMAC_SHA1_ALGORITHM);
mac.init(signingKey);

// compute the hmac on input data bytes
byte[] rawHmac = mac.doFinal(data.getBytes());

// base64-encode the hmac
result = Encoding.EncodeBase64(rawHmac);

} catch (Exception e) {
throw new SignatureException("Failed to generate HMAC : " + e.getMessage());
}
return result;
}
}
```

## Query Request Authentication

When you programmatically call the functionality exposed by the Amazon SQS API, all calls sent to Amazon SQS must be signed. If you use an [AWS SDK](#), the SDK handles the signing process for you so that you do not have to manually complete the tasks. On the other hand, if you submit a Query request over HTTP/HTTPS, then you must include a signature in every Query request.

Amazon SQS supports signature version 4. Signature version 4 provides improved security and performance over previous versions. If you are creating new applications that use Amazon SQS, then you should use signature version 4.

For information on how to create the signature using signature version 4, see [Signature Version 4 Signing Process](#) in the AWS General Reference.



## Responses

### Topics

- [Structure of a Successful Response](#) (p. 68)
- [Structure of an Error Response](#) (p. 68)
- [Related Topics](#) (p. 69)

In response to an action request, Amazon SQS returns an XML data structure that contains the results of the request. This data conforms to the Amazon SQS schema. For more information, see [WSDL Location and API Version](#) in the Amazon SQS API Reference.

## Structure of a Successful Response

If the request succeeded, the main response element is named after the action, but with "Response" appended. For example, `CreateQueueResponse` is the response element returned for a successful `CreateQueue` request. This element contains the following child elements:

- `ResponseMetadata`, which contains the `RequestId` child element
- An optional element containing action-specific results; for example, the `CreateQueueResponse` element includes an element called `CreateQueueResult`

The XML schema describes the XML response message for each Amazon SQS action.

The following is an example of a successful response.

```
<CreateQueueResponse
  xmlns=http://sqs.us-east-1.amazonaws.com/doc/2012-11-05/
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xsi:type=CreateQueueResponse>
  <CreateQueueResult>
    <QueueUrl>
      http://sqs.us-east-1.amazonaws.com/770098461991/queue2
    </QueueUrl>
  </CreateQueueResult>
  <ResponseMetadata>
    <RequestId>cb919c0a-9bce-4afe-9b48-9bdf2412bb67</RequestId>
  </ResponseMetadata>
</CreateQueueResponse>
```

## Structure of an Error Response

If a request is unsuccessful, the main response element is called `ErrorResponse` regardless of the action that was called. This element contains an `Error` element and a `RequestId` element. Each `Error` includes:

- A `Type` element that identifies whether the error was a receiver or sender error
- A `Code` element that identifies the type of error that occurred
- A `Message` element that describes the error condition in a human-readable form
- A `Detail` element that might give additional details about the error or might be empty

The following is an example of an error response.

```
<ErrorResponse>
  <Error>
    <Type>
      Sender
    </Type>
    <Code>
      InvalidParameterValue
    </Code>
    <Message>
      Value (quename_nonalpha) for parameter QueueName is invalid.
      Must be an alphanumeric String of 1 to 80 in length
    </Message>
  </Error>
  <RequestId>
    42d59b56-7407-4c4a-be0f-4c88daeeea257
  </RequestId>
</ErrorResponse>
```

## Related Topics

- [Making Query Requests \(p. 57\)](#)

# Shared Queues

## Topics

- [Simple API for Shared Queues \(p. 70\)](#)
- [Advanced API for Shared Queues \(p. 70\)](#)
- [Understanding Permissions \(p. 70\)](#)
- [Granting Anonymous Access to a Queue \(p. 71\)](#)

Amazon SQS includes methods to share your queues so others can use them, using permissions set in an access control policy. A *permission* gives access to another person to use your queue in some particular way. A *policy* is the actual document that contains the permissions you've granted.

Amazon SQS offers two methods for setting a policy: a simple API and an advanced API. In the simple API, Amazon SQS generates an access control policy for you. In the advanced API, you create the access control policy.

## Simple API for Shared Queues

The simple API for sharing a queue has two operations:

- [AddPermission](#)
- [RemovePermission](#)

With the Simple API, Amazon SQS writes the policy in the required language for you based on the information you include in the `AddPermission` operation. However, the policy that Amazon SQS generates is limited in scope. You can grant permissions to principals, but you can't specify restrictions.

## Advanced API for Shared Queues

With the advanced API, you write the policy yourself directly in the access policy language and upload the policy with the `SetQueueAttributes` operation. The advanced API allows you to deny access or to apply finer access restrictions (for example, based on time or based on IP address).

If you choose to write your own policies, you need to understand how policies are structured. For complete reference information about policies, see [Using The Access Policy Language \(p. 77\)](#). For examples of policies, see [Amazon SQS Policy Examples \(p. 91\)](#).

## Understanding Permissions

A permission is the type of access you give to a *principal* (the user receiving the permission). You give each permission a label that identifies that permission. If you want to delete that permission in the future, you use that label to identify the permission. If you want to see what permissions are on a queue, use the `GetQueueAttributes` operation. Amazon SQS returns the entire policy (containing all the permissions).

Amazon SQS supports the permission types shown in the following table.

Permission	Description
*	This permission type grants the following actions to a principal on a shared queue: receive messages, send messages, delete messages, change a message's visibility, get a queue's attributes.
ReceiveMessage	This grants permission to receive messages in the queue.

Permission	Description
SendMessage	This grants permission to send messages to the queue. <code>SendMessageBatch</code> inherits permissions associated with <code>SendMessage</code> .
DeleteMessage	This grants permission to delete messages from the queue. <code>DeleteMessageBatch</code> inherits permissions associated with <code>DeleteMessage</code> .
ChangeMessageVisibility	This grants permission to extend or terminate the read lock timeout of a specified message. <code>ChangeMessageVisibilityBatch</code> inherits permissions associated with <code>ChangeMessageVisibility</code> . For more information about visibility timeout, see <a href="#">Visibility Timeout (p. 6)</a> . For more information about this permission type, see the <a href="#">ChangeMessageVisibility</a> operation.
GetQueueAttributes	This grants permission to receive all of the queue attributes except the policy, which can only be accessed by the queue's owner. For more information, see the <a href="#">GetQueueAttributes</a> operation..

#### Note

Setting permissions for `SendMessage`, `DeleteMessage`, or `ChangeMessageVisibility` also sets permissions for the corresponding batch versions of those actions: `SendMessageBatch`, `DeleteMessageBatch`, and `ChangeMessageVisibilityBatch`. Setting permissions explicitly on `SendMessageBatch`, `DeleteMessageBatch`, and `ChangeMessageVisibilityBatch` is not allowed.

Permissions for each of the different permission types are considered separate permissions by Amazon SQS, even though `*` includes the access provided by the other permission types. For example, it is possible to grant both `*` and `SendMessage` permissions to a user, even though a `*` includes the access provided by `SendMessage`.

This concept applies when you remove a permission. If a principal has only a `*` permission, requesting to remove a `SendMessage` permission does not leave the principal with an "everything but" permission. Instead, the request does nothing, because the principal did not previously possess an explicit `SendMessage` permission.

If you want to remove `*` and leave the principal with just the `ReceiveMessage` permission, first add the `ReceiveMessage` permission, then remove the `*` permission.

#### Tip

You give each permission a label that identifies that permission. If you want to delete that permission in the future, you use that label to identify the permission.

#### Note

If you want to see what permissions are on a queue, use the [GetQueueAttributes](#) operation. The entire policy (containing all the permissions) is returned.

## Granting Anonymous Access to a Queue

You can allow shared queue access to anonymous users. Such access requires no signature or Access Key ID.

To allow anonymous access you must write your own policy, setting the `Principal` to `*`. For information about writing your own policies, see [Using The Access Policy Language \(p. 77\)](#).

#### Caution

Keep in mind that the queue owner is responsible for all costs related to the queue. Therefore you probably want to limit anonymous access in some other way (by time or IP address, for example).

## Programming Languages

AWS provides libraries, sample code, tutorials, and other resources for software developers who prefer to build applications using language-specific APIs instead of Amazon SQS's Query API. These libraries provide basic functions (not included in Amazon SQS's Query API), such as request authentication, request retries, and error handling so you can get started more easily. Libraries and resources are available for the following languages:

- [Java](#)
- [PHP](#)
- [Python](#)
- [Ruby](#)
- [Windows and .NET](#)

For libraries and sample code in all languages, see [Sample Code & Libraries](#).

For mobile application development, see:

- [AWS SDK for Android](#)
- [AWS SDK for iOS](#)

# Using Amazon SQS Dead Letter Queues

---

Amazon SQS now provides support for *dead letter queues*. A dead letter queue is a queue that other (source) queues can target to send messages that for some reason could not be successfully processed. A primary benefit of using a dead letter queue is the ability to sideline and isolate the unsuccessfully processed messages. You can then analyze any messages sent to the dead letter queue to try and determine why they were not successfully processed.

To specify a dead letter queue, you can use the AWS Management Console or the query API. This must be done for each (source) queue that will send messages to a dead letter queue. You can have multiple (source) queues target a single dead letter queue.

## Important

You must use the same AWS account to create the queue which will be used as a dead letter queue and the other (source) queues that will send messages to the dead letter queue. Dead letter queues also must reside in the same region as the other queues that use the dead letter queue. For example, if you created a queue in the US East (N. Virginia) region and you want to use a dead letter queue with that queue, then both queues must be in the US East (N. Virginia) region.

## Topics

- [Setting up Dead Letter Queue with the AWS Management Console \(p. 73\)](#)
- [Using Dead Letter Queue with the Amazon SQS API \(p. 75\)](#)

## Setting up Dead Letter Queue with the AWS Management Console

You can use the AWS Management Console to configure dead letter queue functionality so that messages, which for some reason could not be successfully processed, are sent to a specified dead letter queue. This is accomplished by first selecting the source queue's **Use Redrive Policy** check box for existing and newly created queues.

## Amazon Simple Queue Service Developer Guide Setting up Dead Letter Queue with the AWS Management Console

### Dead Letter Queue Settings

**Use Redrive Policy:**

**Dead Letter Queue:**  Value must be an existing queue name.

**Maximum Receives:**  Value must be between 1 and 1000.

Next, enter the name of the queue to which messages will be sent from the source queues.

### Dead Letter Queue Settings

**Use Redrive Policy:**

**Dead Letter Queue:**  Value must be an existing queue name.

**Maximum Receives:**  Value must be between 1 and 1000.

You then set the **Maximum Receives** to a value between 1 and 1000.

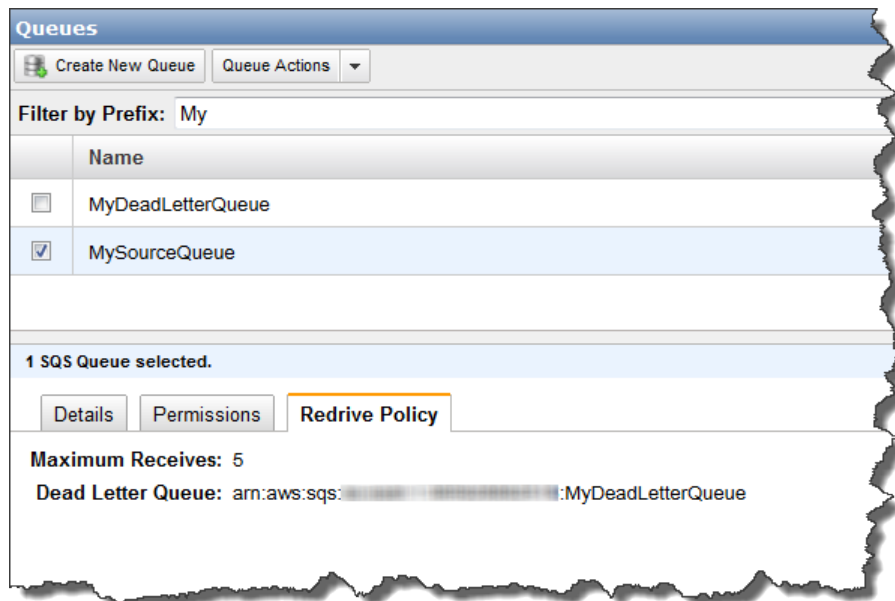
### Dead Letter Queue Settings

**Use Redrive Policy:**

**Dead Letter Queue:**  Value must be an existing queue name.

**Maximum Receives:**  Value must be between 1 and 1000.

The following figure shows the *MySourceQueue* queue, which has the **Redrive Policy** configured to have messages sent to the *MyDeadLetterQueue* queue.



## Using Dead Letter Queue with the Amazon SQS API

To specify a dead letter queue with the query API, you call either the `CreateQueue` or `SetQueueAttributes` actions and set the `maxReceiveCount` and `deadLetterTargetArn` parameters for the `RedrivePolicy` queue attribute.

You can set `maxReceiveCount` to a value between 1 and 1000. The `deadLetterTargetArn` value is the Amazon Resource Name (ARN) of the queue that will receive the dead letter messages.

The following Java example shows how to use `SetQueueAttributes` to set the `maxReceiveCount` and `deadLetterTargetArn` parameters for the `RedrivePolicy` queue attribute. This example is based on the `SimpleQueueServiceSample.java` sample from the AWS SDK for Java.

First, a string is set that contains JSON formatted parameters and values for the `RedrivePolicy` queue attribute:

```
String redrivePolicy = "{\"maxReceiveCount\": \"5\", \"deadLetterTargetArn\": \"arn:aws:sqs:us-east-1:123456789012:MyDeadLetterQueue\"}";
```

Next, `SetQueueAttributesRequest` is used to set the `RedrivePolicy` queue attribute:

```
SetQueueAttributesRequest queueAttributes = new SetQueueAttributesRequest();
Map<String,String> attributes = new HashMap<String,String>();
attributes.put("RedrivePolicy", redrivePolicy);
queueAttributes.setAttributes(attributes);
queueAttributes.setQueueUrl(myQueueUrl);
sqs.setQueueAttributes(queueAttributes);
```

An API query request for this example should look similar to the following:

```
http://sqs.us-east-1.amazonaws.com/123456789012/MySourceQueue
?Action=SetQueueAttributes
&Attribute.1.Value=%7B%22maxReceiveCount%22%3A%225%22%2C%22deadLetterTarget
Arn%22%3A%22arn%3Aaws%3Asqs%3Aus-east-1%3A123456789012%3AMyDeadLetterQueue%22%7D
&Version=2012-11-05
&Attribute.1.Name=RedrivePolicy
```

The API query response should look similar to the following:

```
<SetQueueAttributesResponse xmlns="http://queue.amazonaws.com/doc/2012-11-05/">
  <ResponseMetadata>
    <RequestId>40945605-b328-53b5-aed4-1cc24a7240e8</RequestId>
  </ResponseMetadata>
```



```
</SetQueueAttributesResponse>
```

# Using The Access Policy Language

---

## Topics

- [Overview \(p. 78\)](#)
- [Amazon SQS Policy Examples \(p. 91\)](#)
- [Special Information for Amazon SQS Policies \(p. 95\)](#)

This section is for Amazon SQS users who want to write their own access control policies. You don't need to write your own policies if you want to allow access based only on AWS account ID and basic permissions (e.g., `SendMessage`, `ReceiveMessage`). In that case, you can just use the Amazon SQS `AddPermission` action. If you want to explicitly deny access or allow it based on finer conditions (such as the time the request comes in or the IP address of the requester), you need to write your own policies and upload them to the AWS system using the Amazon SQS `SetQueueAttributes` action.

### Note

To write your own policies, you must be familiar with JSON. For more information, go to <http://json.org>.

The main portion of this section includes basic concepts you need to understand, how to write a policy, and the logic AWS uses to evaluate policies and decide whether to give the requester access to the resource. Although most of the information in this section is service-agnostic, there are some SQS-specific details you need to know. For more information, see [Special Information for Amazon SQS Policies \(p. 95\)](#).

## Overview

### Topics

- [When to Use Access Control](#) (p. 78)
- [Key Concepts](#) (p. 78)
- [Architectural Overview](#) (p. 81)
- [Using the Access Policy Language](#) (p. 83)
- [Evaluation Logic](#) (p. 84)
- [Basic Use Cases for Access Control](#) (p. 88)

This section describes basic concepts you need to understand to use the access policy language to write policies. It also describes the general process for how access control works with the access policy language, and how policies are evaluated.

## When to Use Access Control

You have a great deal of flexibility in how you grant or deny access to a resource. However, the typical use cases are fairly simple:

- You want to grant another AWS account a particular type of access to your queue (e.g., `SendMessage`). For more information, see [Use Case 1](#) (p. 88).
- You want to grant another AWS account access to your queue for a specific period of time. For more information, see [Use Case 2](#) (p. 88).
- You want to grant another AWS account access to your queue only if the requests come from your Amazon EC2 instances. For more information, see [Use Case 3](#) (p. 89).
- You want to *deny* another AWS account access to your queue. For more information, see [Use Case 4](#) (p. 90).

## Key Concepts

The following sections describe the concepts you need to understand to use the access policy language. They're presented in a logical order, with the first terms you need to know at the top of the list.

### Permission

A *permission* is the concept of allowing or disallowing some kind of access to a particular resource. Permissions essentially follow this form: "A is/isn't allowed to do B to C where D applies." For example, *Jane* (A) has permission to *receive messages* (B) from *John's Amazon SQS queue* (C), as long as *she asks to receive them before midnight on May 30, 2009* (D). Whenever Jane sends a request to Amazon SQS to use John's queue, the service checks to see if she has permission and if the request satisfies the conditions John set forth in the permission.

### Statement

A *statement* is the formal description of a single permission, written in the access policy language. You always write a statement as part of a broader container document known as a *policy* (see the next concept).

### Policy

A *policy* is a document (written in the access policy language) that acts as a container for one or more statements. For example, a policy could have two statements in it: one that states that Jane can use

John's queue, and another that states that Bob cannot use John's queue. As shown in the following figure, an equivalent scenario would be to have two policies, one containing the statement that Jane can use John's queue, and another containing the statement that Bob cannot use John's queue.



The AWS service implementing access control (e.g., Amazon SQS) uses the information in the statements (whether they're contained in a single policy or multiple) to determine if someone requesting access to a resource should be granted that access. We often use the term *policy* interchangeably with *statement*, as they generally represent the same concept (an entity that represents a permission).

## Issuer

The *issuer* is the person who writes a policy to grant permissions for a resource. The issuer (by definition) is always the resource owner. AWS does not permit AWS service users to create policies for resources they don't own. If John is the resource owner, AWS authenticates John's identity when he submits the policy he's written to grant permissions for that resource.

## Principal

The *principal* is the person or persons who receive the permission in the policy. The principal is A in the statement "A has permission to do B to C where D applies." In a policy, you can set the principal to "anyone" (i.e., you can specify a wildcard to represent all people). You might do this, for example, if you don't want to restrict access based on the actual identity of the requester, but instead on some other identifying characteristic such as the requester's IP address.

## Action

The *action* is the activity the principal has permission to perform. The action is B in the statement "A has permission to do B to C where D applies." Typically, the action is just the operation in the request to AWS. For example, Jane sends a request to Amazon SQS with `Action=ReceiveMessage`. You can specify one or multiple actions in a policy.

## Resource

The *resource* is the object the principal is requesting access to. The resource is C in the statement "A has permission to do B to C where D applies."

## Conditions and Keys

The *conditions* are any restrictions or details about the permission. The condition is D in the statement "A has permission to do B to C where D applies." The part of the policy that specifies the conditions can be the most detailed and complex of all the parts. Typical conditions are related to:

- Date and time (e.g., the request must arrive before a specific day)
- IP address (e.g., the requester's IP address must be part of a particular CIDR range)

A *key* is the specific characteristic that is the basis for access restriction. For example, the date and time of request.

You use both *conditions* and *keys* together to express the restriction. The easiest way to understand how you actually implement a restriction is with an example: If you want to restrict access to before May 30, 2010, you use the condition called `DateLessThan`. You use the key called `AWS:CurrentTime` and set it to the value `2010-05-30T00:00:00Z`. AWS defines the conditions and keys you can use. The AWS service itself (e.g., Amazon SQS) might also define service-specific keys. For more information about the available keys, see [Amazon SQS Keys \(p. 101\)](#).

## Requester

The *requester* is the person who sends a request to an AWS service and asks for access to a particular resource. The requester sends a request to AWS that essentially says: "Will you allow me to do B to C where D applies?"

## Evaluation

*Evaluation* is the process the AWS service uses to determine if an incoming request should be denied or allowed based on the applicable policies. For information about the evaluation logic, see [Evaluation Logic \(p. 84\)](#).

## Effect

The *effect* is the result that you want a policy statement to return at evaluation time. You specify this value when you write the statements in a policy, and the possible values are *deny* and *allow*.

For example, you could write a policy that has a statement that *denies* all requests that come from Antarctica (effect=deny given that the request uses an IP address allocated to Antarctica). Alternately, you could write a policy that has a statement that *allows* all requests that *don't* come from Antarctica (effect=allow, given that the request doesn't come from Antarctica). Although the two statements sound like they do the same thing, in the access policy language logic, they are different. For more information, see [Evaluation Logic \(p. 84\)](#).

Although there are only two possible values you can specify for the effect (allow or deny), there can be three different results at policy evaluation time: *default deny*, *allow*, or *explicit deny*. For more information, see the following concepts and [Evaluation Logic \(p. 84\)](#).

## Default Deny

A *default deny* is the default result from a policy in the absence of an allow or explicit deny.

## Allow

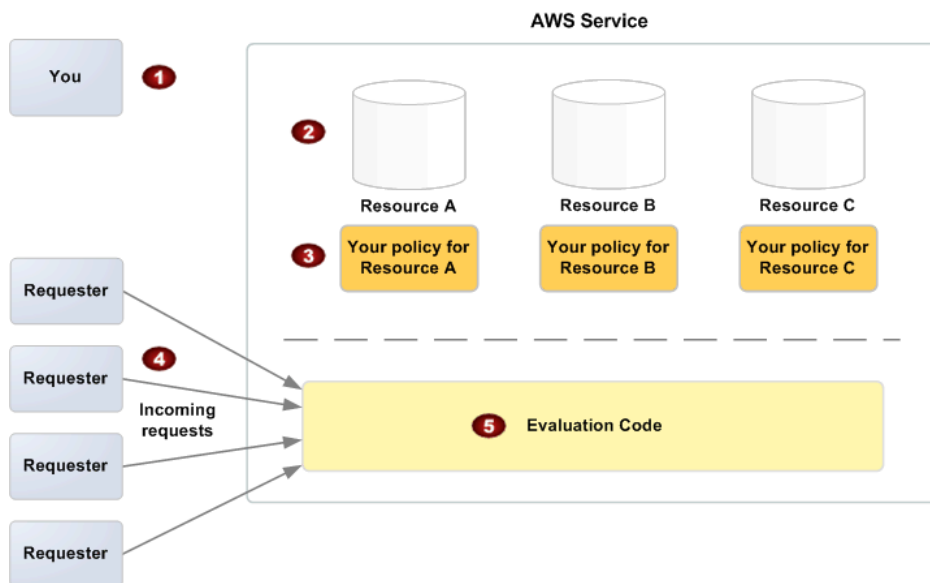
An *allow* results from a statement that has `effect=allow`, assuming any stated conditions are met. Example: Allow requests if they are received before 1:00 p.m. on April 30, 2010. An allow overrides all default denies, but never an explicit deny.

## Explicit Deny

An *explicit deny* results from a statement that has `effect=deny`, assuming any stated conditions are met. Example: Deny all requests if they are from Antarctica. Any request that comes from Antarctica will always be denied no matter what any other policies might allow.

## Architectural Overview

The following figure and table describe the main components that interact to provide access control for your resources.

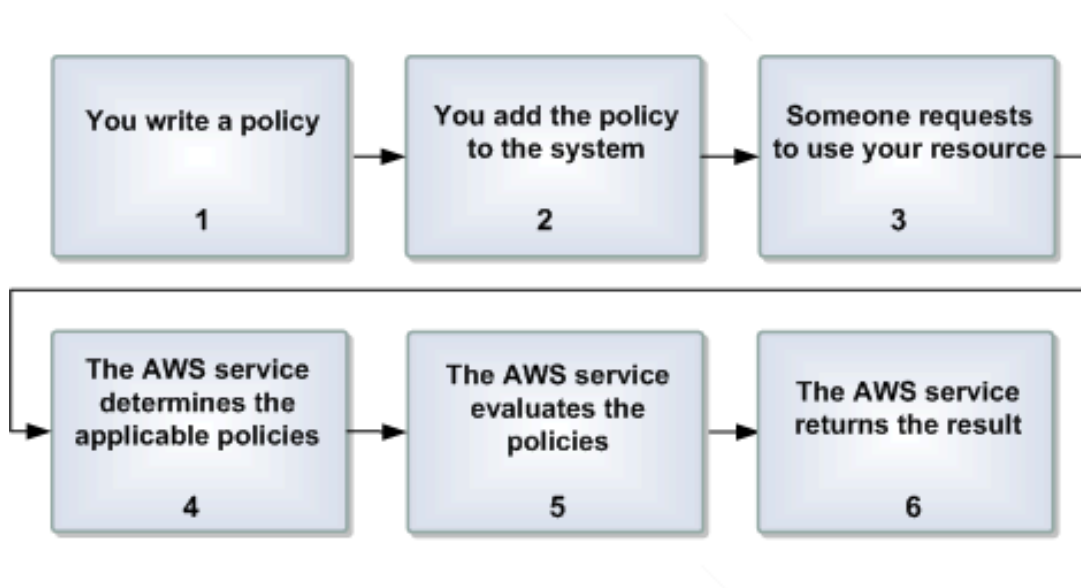


1	You, the resource owner.
2	Your resources (contained within the AWS service; e.g., Amazon SQS queues).
3	Your policies. Typically you have one policy per resource, although you could have multiple. The AWS service itself provides an API you use to upload and manage your policies.
4	Requesters and their incoming requests to the AWS service.
5	The access policy language evaluation code. This is the set of code within the AWS service that evaluates incoming requests against the applicable policies and determines whether the requester is allowed access to the resource. For information about how the service makes the decision, see <a href="#">Evaluation Logic (p. 84)</a> .

For the typical process of how the components work together, see [Using the Access Policy Language \(p. 83\)](#).

## Using the Access Policy Language

The following figure and table describe the general process of how access control works with the access policy language.



### Process for Using Access Control with the Access Policy Language

1	You write a policy for your resource. For example, you write a policy to specify permissions for your Amazon SQS queues.
2	You upload your policy to AWS. The AWS service itself provides an API you use to upload your policies. For example, you use the Amazon SQS <code>SetQueueAttributes</code> action to upload a policy for a particular Amazon SQS queue.
3	Someone sends a request to use your resource. For example, a user sends a request to Amazon SQS to use one of your queues.
4	The AWS service determines which policies are applicable to the request. For example, Amazon SQS looks at all the available Amazon SQS policies and determines which ones are applicable (based on what the resource is, who the requester is, etc.).
5	The AWS service evaluates the policies. For example, Amazon SQS evaluates the policies and determines if the requester is allowed to use your queue or not. For information about the decision logic, see <a href="#">Evaluation Logic (p. 84)</a> .
6	The AWS service either denies the request or continues to process it. For example, based on the policy evaluation result, the service either returns an "Access denied" error to the requester or continues to process the request.

### Related Topics

- [Architectural Overview \(p. 81\)](#)

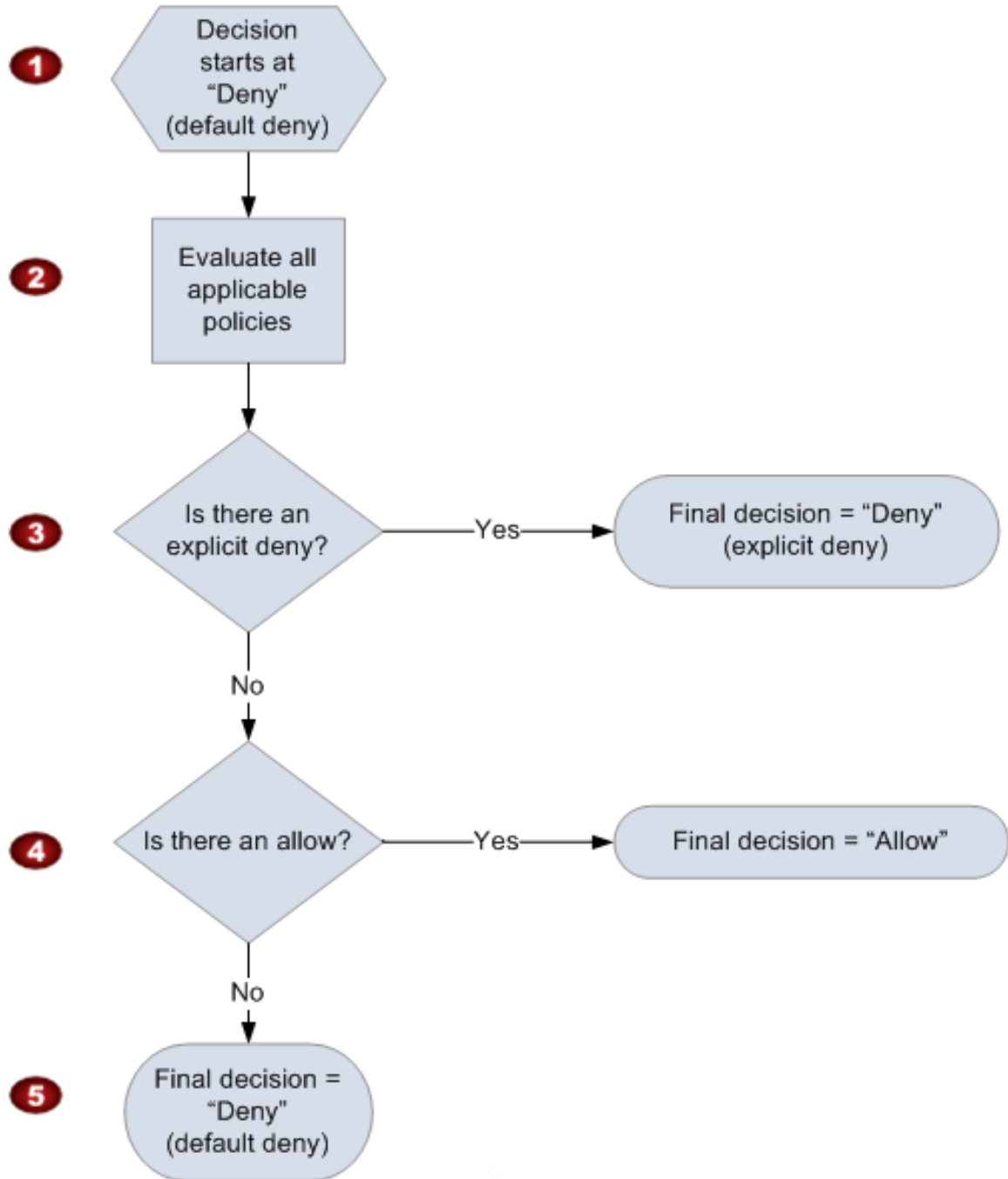


## Evaluation Logic

The goal at evaluation time is to decide whether a given request from someone other than you (the resource owner) should be allowed or denied. The evaluation logic follows several basic rules:

- By default, all requests to use your resource coming from anyone but you are denied
- An allow overrides any default denies
- An explicit deny overrides any allows
- The order in which the policies are evaluated is not important

The following flow chart and discussion describe in more detail how the decision is made.



1	The decision starts with a default deny.
2	The enforcement code then evaluates all the policies that are applicable to the request (based on the resource, principal, action, and conditions). The order in which the enforcement code evaluates the policies is not important.

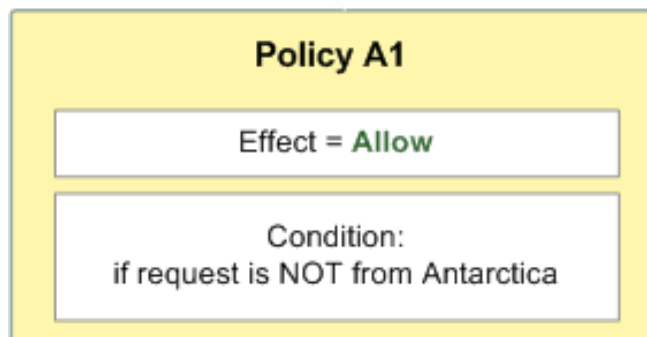
3	In all those policies, the enforcement code looks for an explicit deny instruction that would apply to the request. If it finds even one, the enforcement code returns a decision of "deny" and the process is finished (this is an explicit deny; for more information, see <a href="#">Explicit Deny (p. 81)</a> ).
4	If no explicit deny is found, the enforcement code looks for any "allow" instructions that would apply to the request. If it finds even one, the enforcement code returns a decision of "allow" and the process is done (the service continues to process the request).
5	If no allow is found, then the final decision is "deny" (because there was no explicit deny or allow, this is considered a <i>default deny</i> (for more information, see <a href="#">Default Deny (p. 80)</a> )).

## The Interplay of Explicit and Default Denials

A policy results in a default deny if it doesn't directly apply to the request. For example, if a user requests to use Amazon SQS, but the only policy that applies to the user states that the user can use Amazon SimpleDB, then that policy results in a default deny.

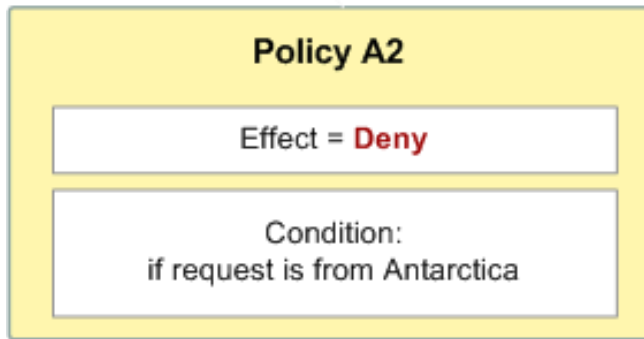
A policy also results in a default deny if a condition in a statement isn't met. If all conditions in the statement are met, then the policy results in either an allow or an explicit deny, based on the value of the Effect element in the policy. Policies don't specify what to do if a condition isn't met, and so the default result in that case is a default deny.

For example, let's say you want to prevent requests coming in from Antarctica. You write a policy (called Policy A1) that allows a request only if it doesn't come from Antarctica. The following diagram illustrates the policy.



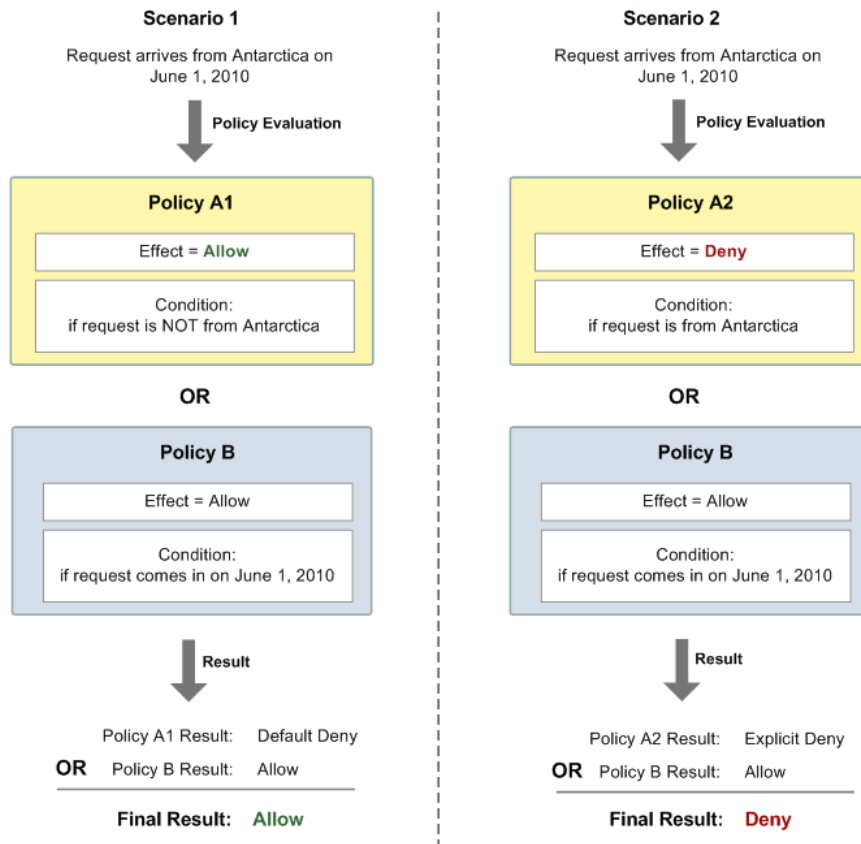
If someone sends a request from the U.S., the condition is met (the request is not from Antarctica). Therefore, the request is allowed. But, if someone sends a request from Antarctica, the condition isn't met, and the policy's result is therefore a default deny.

You could turn the result into an explicit deny by rewriting the policy (named Policy A2) as in the following diagram. Here, the policy explicitly denies a request if it comes from Antarctica.



If someone sends a request from Antarctica, the condition is met, and the policy's result is therefore an explicit deny.

The distinction between a default deny and an explicit deny is important because a default deny can be overridden by an allow, but an explicit deny can't. For example, let's say there's another policy that allows requests if they arrive on June 1, 2010. How does this policy affect the overall outcome when coupled with the policy restricting access from Antarctica? We'll compare the overall outcome when coupling the date-based policy (we'll call Policy B) with the preceding policies A1 and A2. Scenario 1 couples Policy A1 with Policy B, and Scenario 2 couples Policy A2 with Policy B. The following figure and discussion show the results when a request comes in from Antarctica on June 1, 2010.



In Scenario 1, Policy A1 returns a default deny, as described earlier in this section. Policy B returns an allow because the policy (by definition) allows requests that come in on June 1, 2010. The allow from Policy B overrides the default deny from Policy A1, and the request is therefore allowed.

In Scenario 2, Policy B2 returns an explicit deny, as described earlier in this section. Again, Policy B returns an allow. The explicit deny from Policy A2 overrides the allow from Policy B, and the request is therefore denied.

## Basic Use Cases for Access Control

This section gives a few examples of typical use cases for access control.

### Use Case 1

Let's say you have a set of queues in the Amazon SQS system. In the simplest case, you want to allow one or more AWS accounts a particular type of access to a queue (e.g., `SendMessage`, `ReceiveMessage`).

You can do this by simply using the Amazon SQS API action `AddPermission`. It takes a few input parameters and automatically creates a policy in the Amazon SQS system for that queue. For this use case, you don't need to read this appendix or learn how to write a policy yourself, because Amazon SQS can automatically create the policy for you.

The following example shows a policy that gives AWS account ID 1111-2222-3333 permission to send and receive from a queue you own named `queue2`. In this example, your AWS account ID is 4444-5555-6666.

```
{
  "Version": "2012-10-17",
  "Id": "UseCase1",
  "Statement" : [
    {
      "Sid": "1",
      "Effect": "Allow",
      "Principal" : {
        "AWS": "111122223333"
      },
      "Action": [ "sqs:SendMessage", "sqs:ReceiveMessage" ],
      "Resource": "arn:aws:sqs:us-east-1:444455556666:queue2",
    }
  ]
}
```

### Use Case 2

In this use case, you want to allow one or more AWS accounts access to your queues *only for a specific time period*.

You need to know how to write your own policy for the queue because the Amazon SQS `AddPermission` action doesn't let you specify a time restriction when granting someone access to your queue. In this case, you would write your own policy and then upload it to the AWS system with the `SetQueueAttributes` action. Effectively the action sets your policy as an attribute of the queue.

The following example is the same as in use case 1, except it also includes a condition that restricts access to before June 30, 2009, at noon (UTC).

```
{
  "Version": "2012-10-17",
  "Id": "UseCase2",
  "Statement" : [
    {
      "Sid": "1",
      "Effect": "Allow",
      "Principal" : {
        "AWS": "111122223333"
      },
      "Action": ["sqs:SendMessage", "sqs:ReceiveMessage"],
      "Resource": "arn:aws:sqs:us-east-1:444455556666:queue2",
      "Condition" : {
        "DateLessThan" : {
          "AWS:CurrentTime": "2009-06-30T12:00Z"
        }
      }
    }
  ]
}
```

### Use Case 3

In this use case, you want to allow access to your queues *only if the requests come from your Amazon EC2 instances*.

Again, you need to know how to write your own policy because the Amazon SQS `AddPermission` action doesn't let you specify an IP address restriction when granting access to your queue.

The following example builds on the example in use case 2, and also includes a condition that restricts access to the IP address range 10.52.176.0/24. So in this example, a request from AWS account 1234-5678-9012 to send or receive messages from queue2 would be allowed only if it came in before noon on June 30, 2009, *and* it came from the 10.52.176.0/24 address range.

```
{
  "Version": "2012-10-17",
  "Id": "UseCase3",
  "Statement" : [
    {
      "Sid": "1",
      "Effect": "Allow",
      "Principal" : {
        "AWS": "111122223333"
      },
      "Action": ["sqs:SendMessage", "sqs:ReceiveMessage"],
      "Resource": "arn:aws:sqs:us-east-1:444455556666:queue2",
      "Condition" : {
        "DateLessThan" : {
          "AWS:CurrentTime": "2009-06-30T12:00Z"
        },
        "IpAddress" : {
          "AWS:SourceIp": "10.52.176.0/24"
        }
      }
    }
  ]
}
```

```
]
}
```

## Use Case 4

In this use case, you want to specifically *deny* a certain AWS account access to your queues.

Again, you need to know how to write your own policy because the Amazon SQS `AddPermission` action doesn't let you *deny* access to a queue; it only lets you *grant* access.

The following example is the same as in the original use case (#1), except it *denies* access to the specified AWS account.

```
{
  "Version": "2012-10-17",
  "Id": "UseCase4",
  "Statement" : [
    {
      "Sid": "1",
      "Effect": "Deny",
      "Principal" : {
        "AWS": "111122223333"
      },
      "Action": [ "sqs:SendMessage", "sqs:ReceiveMessage" ],
      "Resource": "arn:aws:sqs:us-east-1:444455556666:queue2",
    }
  ]
}
```

From these use cases, you can see that if you want to restrict access based on special conditions or deny someone access entirely, you need to read this appendix and learn how to write your own policies. You can also see that the policies themselves are not that complex and the access policy language is straightforward.

## Amazon SQS Policy Examples

This section shows example policies for common Amazon SQS use cases.

The following example policy gives the developer with AWS account number 111122223333 the `SendMessage` permission for the queue named `444455556666/queue1` in the US East (N. Virginia) region.

```
{
  "Version": "2012-10-17",
  "Id": "Queue1_Policy_UUID",
  "Statement":
  {
    "Sid": "Queue1_SendMessage",
    "Effect": "Allow",
    "Principal": {
      "AWS": "111122223333"
    },
    "Action": "sqs:SendMessage",
    "Resource": "arn:aws:sqs:us-east-1:444455556666:queue1"
  }
}
```

The following example policy gives the developer with AWS account number 111122223333 both the `SendMessage` and `ReceiveMessage` permission for the queue named `444455556666/queue1`.

```
{
  "Version": "2012-10-17",
  "Id": "Queue1_Policy_UUID",
  "Statement":
  {
    "Sid": "Queue1_Send_Receive",
    "Effect": "Allow",
    "Principal": {
      "AWS": "111122223333"
    },
    "Action": ["sqs:SendMessage", "sqs:ReceiveMessage"],
    "Resource": "arn:aws:sqs:*:444455556666:queue1"
  }
}
```

The following example policy gives two different developers (with AWS account numbers 111122223333 and 444455556666) permission to use all actions that Amazon SQS allows shared access for the queue named `123456789012/queue1` in the US East (N. Virginia) region.

```
{
  "Version": "2012-10-17",
  "Id": "Queue1_Policy_UUID",
  "Statement":
  {
    "Sid": "Queue1_AllActions",
    "Effect": "Allow",
    "Principal": {
      "AWS": ["111122223333", "444455556666"]
    },
    "Action": "sqs:*",
  }
}
```



```
    "Resource": "arn:aws:sqs:us-east-1:123456789012:queue1"
  }
}
```

The following example policy gives all users `ReceiveMessage` permission for the queue named `111122223333/queue1`.

```
{
  "Version": "2012-10-17",
  "Id": "Queue1_Policy_UUID",
  "Statement":
  {
    "Sid": "Queue1_AnonymousAccess_ReceiveMessage",
    "Effect": "Allow",
    "Principal": "*",
    "Action": "sqs:ReceiveMessage",
    "Resource": "arn:aws:sqs:*:111122223333:queue1"
  }
}
```

The following example policy gives all users `ReceiveMessage` permission for the queue named `111122223333/queue1`, but only between noon and 3:00 p.m. on January 31, 2009.

```
{
  "Version": "2012-10-17",
  "Id": "Queue1_Policy_UUID",
  "Statement":
  {
    "Sid": "Queue1_AnonymousAccess_ReceiveMessage_TimeLimit",
    "Effect": "Allow",
    "Principal": "*",
    "Action": "sqs:ReceiveMessage",
    "Resource": "arn:aws:sqs:*:111122223333:queue1",
    "Condition": {
      "DateGreaterThan": {
        "aws:CurrentTime": "2009-01-31T12:00Z"
      },
      "DateLessThan": {
        "aws:CurrentTime": "2009-01-31T15:00Z"
      }
    }
  }
}
```

The following example policy gives all users permission to use all possible Amazon SQS actions that can be shared for the queue named `111122223333/queue1`, but only if the request comes from the `192.168.143.0/24` range.

```
{
  "Version": "2012-10-17",
  "Id": "Queue1_Policy_UUID",
  "Statement":
  {
    "Sid": "Queue1_AnonymousAccess_AllActions_WhitelistIP",
    "Effect": "Allow",
```

```
    "Principal": "*",
    "Action": "sqs:*",
    "Resource": "arn:aws:sqs:*:111122223333:queue1",
    "Condition" : {
      "IpAddress" : {
        "aws:SourceIp": "192.168.143.0/24"
      }
    }
  }
}
```

The following example policy has two statements:

- One that gives all users in the 192.168.143.0/24 range (except for 192.168.143.188) permission to use the SendMessage action for the queue named 111122223333/queue1.
- One that blacklists all users in the 10.1.2.0/24 range from using the queue.

```
{
  "Version": "2012-10-17",
  "Id": "Queue1_Policy_UUID",
  "Statement": [
    {
      "Sid": "Queue1_AnonymousAccess_SendMessage_IPLimit",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "sqs:SendMessage",
      "Resource": "arn:aws:sqs:*:111122223333:queue1",
      "Condition" : {
        "IpAddress" : {
          "aws:SourceIp": "192.168.143.0/24"
        },
        "NotIpAddress" : {
          "aws:SourceIp": "192.168.143.188/32"
        }
      }
    },
    {
      "Sid": "Queue1_AnonymousAccess_AllActions_IPLimit_Deny",
      "Effect": "Deny",
      "Principal": "*",
      "Action": "sqs:*",
      "Resource": "arn:aws:sqs:*:111122223333:queue1",
      "Condition" : {
        "IpAddress" : {
          "aws:SourceIp": "10.1.2.0/24"
        }
      }
    }
  ]
}
```

The following example policy enables a connection between the Amazon Simple Notification Service topic specified by the Amazon Resource Name (ARN)

arn:aws:sns:us-east-1:111122223333:test-topic and the queue named  
arn:aws:sqs:us-east-1:111122223333:test-topic-queue.

```
{
  "Version": "2012-10-17",
  "Id": "SNStoSQS",
  "Statement":
  {
    "Sid": "rule1",
    "Effect": "Allow",
    "Principal": "*",
    "Action": "sqs:*",
    "Resource": "arn:aws:sqs:us-east-1:111122223333:test-topic-queue",
    "Condition" : {
      "StringEquals" : {
        "aws:SourceArn": "arn:aws:sns:us-east-1:111122223333:test-topic"
      }
    }
  }
}
```

## Special Information for Amazon SQS Policies

The following list gives information specific to the Amazon SQS implementation of access control.

- Amazon SQS allows you to share only certain types of permissions (for more information, see [Understanding Permissions \(p. 70\)](#))
- Each policy must cover only a single queue (when writing a policy, don't include statements that cover different queues)
- Each policy must have a unique policy ID (`Id`)
- Each statement in a policy must have a unique statement ID (`sid`)
- Amazon SQS does not implement any special keys to use when you write conditions; the only keys available are the general AWS-wide keys.

The following table lists the maximum limits for policy information.

Name	Maximum Limit
Bytes	8192
Statements	20
Principals	50
Conditions	10

# Access Control Using AWS Identity and Access Management (IAM)

---

## Topics

- [IAM-Related Features of Amazon SQS Policies \(p. 96\)](#)
- [IAM and Amazon SQS Policies Together \(p. 98\)](#)
- [Amazon SQS ARNs \(p. 100\)](#)
- [Amazon SQS Actions \(p. 101\)](#)
- [Amazon SQS Keys \(p. 101\)](#)
- [Example IAM Policies for Amazon SQS \(p. 102\)](#)
- [Using Temporary Security Credentials \(p. 103\)](#)

Amazon SQS has its own resource-based permissions system that uses policies written in the same language used for AWS Identity and Access Management (IAM) policies. This means that you can achieve the same things with Amazon SQS policies that you can with IAM policies, such as using variables in IAM policies. For more information, see [Policy Variables](#) in the *Using IAM* guide.

The main difference between using Amazon SQS policies versus IAM policies is that you can grant another AWS Account permission to your queues with an Amazon SQS policy, and you can't do that with an IAM policy.

### Note

When you grant other AWS accounts access to your AWS resources, be aware that all AWS accounts can delegate their permissions to users under their accounts. This is known as cross-account access. Cross-account access enables you to share access to your AWS resources without having to manage additional users. For information about using cross-account access, go to [Enabling Cross-Account Access](#) in *IAM User Guide*.

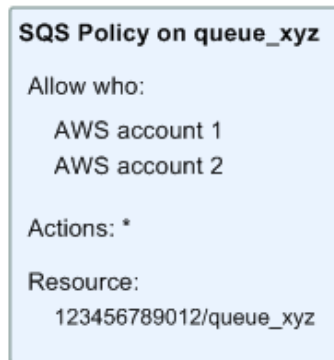
This section describes how the Amazon SQS policy system works with IAM.

## IAM-Related Features of Amazon SQS Policies

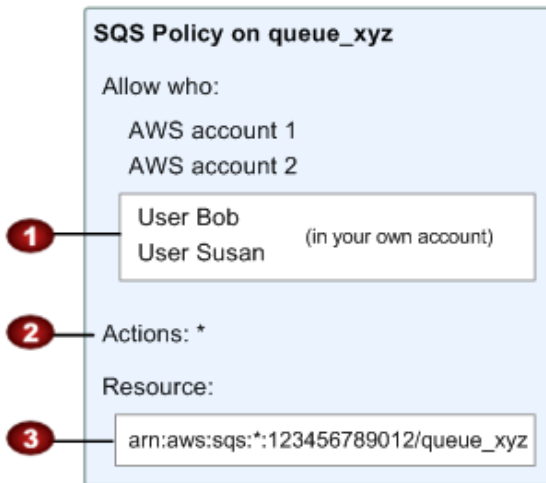
You can use an Amazon SQS policy with a queue to specify which AWS Accounts have access to the queue. You can specify the type of access and conditions (e.g., permission to use `SendMessage`, `ReceiveMessage`, if the request is before December 31, 2010). The specific actions you can grant

permission for are a subset of the overall list of Amazon SQS actions. When you write an Amazon SQS policy and specify \* to mean "all the Amazon SQS actions", that means all actions in that subset.

The following diagram illustrates the concept of one of these basic Amazon SQS policies that covers the subset of actions. The policy is for queue\_xyz, and it gives AWS Account 1 and AWS Account 2 permission to use any of the allowed actions with the queue. Notice that the resource in the policy is specified as 123456789012/queue\_xyz (where 123456789012 is the AWS Account ID of the account that owns the queue).



With the introduction of IAM and the concepts of *Users* and *Amazon Resource Names (ARNs)*, a few things have changed about SQS policies. The following diagram and table describe the changes.



1	In addition to specifying which AWS Accounts have access to the queue, you can specify which Users <i>in your own AWS Account</i> have access to the queue. The Users can't be in another AWS Account.
2	The subset of actions included in "*" has expanded (for a list of allowed actions, see <a href="#">Amazon SQS Actions (p. 101)</a> ).

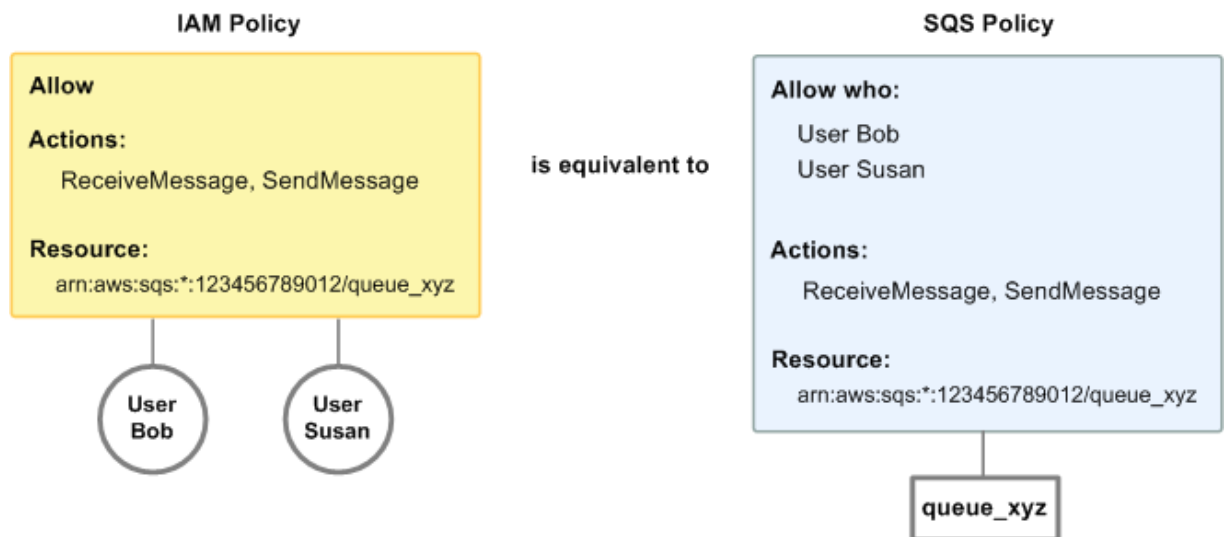
- 3 You can specify the resource using the *Amazon Resource Name (ARN)*, which is how you must specify resources in IAM policies. For information about the ARN format for Amazon SQS queues, see [Amazon SQS ARNs \(p. 100\)](#).  
You can still use the original format instead (<account\_ID>/<queue\_name>).

So for example, according to the Amazon SQS policy shown in the preceding figure, anyone possessing the security credentials for AWS Account 1 or AWS Account 2 could access queue\_xyz. Also, Users Bob and Susan in your own AWS Account (with ID 123456789012) can access the queue.

Before the introduction of IAM, Amazon SQS automatically gave the creator of a queue full control over the queue (e.g., access to all possible Amazon SQS actions with that queue). This is no longer true, unless the creator is using the AWS security credentials. Any User who has permission to create a queue must also have permission to use other Amazon SQS actions in order to do anything with the queues they create.

## IAM and Amazon SQS Policies Together

There are two ways you can give your Users permissions for your Amazon SQS resources: through the Amazon SQS policy system or the IAM policy system. You can use one or the other, or both. For the most part, you can achieve the same results with either. For example, the following diagram shows an IAM policy and an Amazon SQS policy that are equivalent. The IAM policy allows the Amazon SQS `ReceiveMessage` and `SendMessage` actions for the queue called queue\_xyz in your AWS Account, and it's attached to the Users Bob and Susan (which means Bob and Susan have the permissions stated in the policy). The Amazon SQS policy also gives Bob and Susan permission to access `ReceiveMessage` and `SendMessage` for the same queue.



### Note

The preceding example shows simple policies with no conditions. You could specify a particular condition in either policy and get the same result.

There is one difference between IAM and Amazon SQS policies: the Amazon SQS policy system lets you grant permission to other AWS Accounts, whereas IAM doesn't.

It's up to you how you use both of the systems together to manage your permissions, based on your needs. The following examples show how the two policy systems work together.

1

In this example, Bob has both an IAM policy and an Amazon SQS policy that apply to him. The IAM policy gives him permission to use `ReceiveMessage` on `queue_xyz`, whereas the Amazon SQS policy gives him permission to use `SendMessage` on the same queue. The following diagram illustrates the concept.

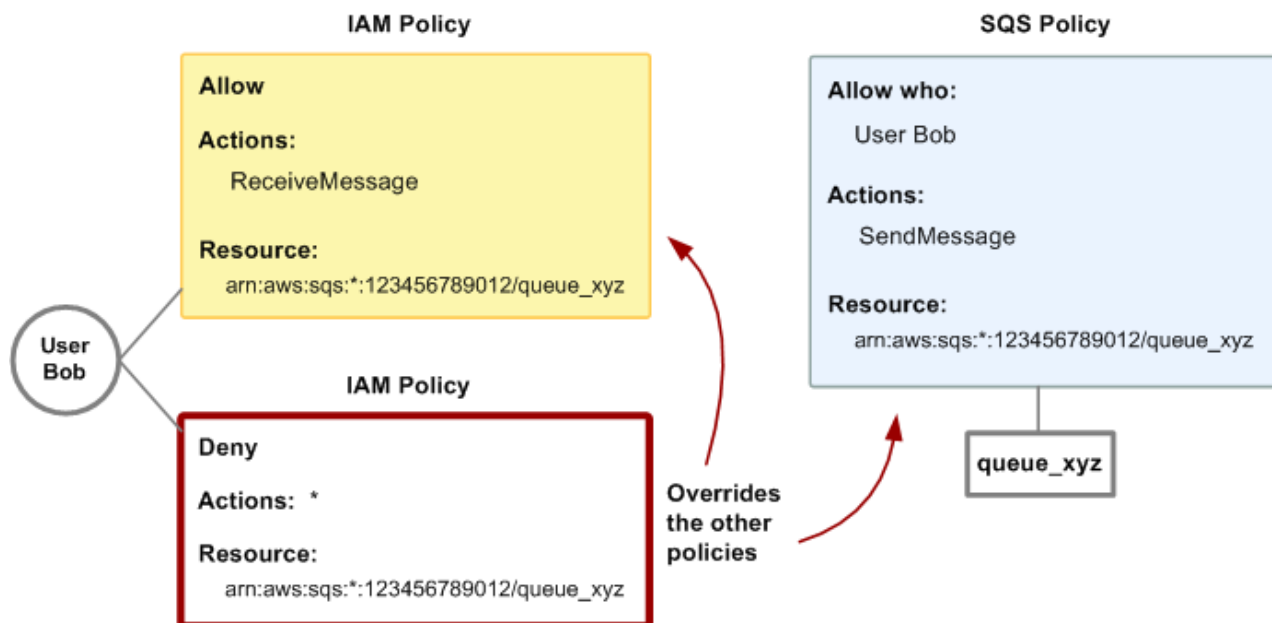


If Bob were to send a request to receive a message from `queue_xyz`, the IAM policy would allow the action. If Bob were to send a request to send a message to `queue_xyz`, the Amazon SQS policy would allow the action.

2

In this example, we build on example 1 (where Bob has two policies that apply to him). Let's say that Bob abuses his access to `queue_xyz`, so you want to remove his entire access to that queue. The easiest thing to do is add a policy that denies him access to all actions on the queue. This third policy overrides the other two, because an explicit deny always overrides an allow (for more information about policy evaluation logic, see [Evaluation Logic \(p. 84\)](#)). The following diagram illustrates the concept.





Alternatively, you could add an additional statement to the Amazon SQS policy that denies Bob any type of access to the queue. It would have the same effect as adding a IAM policy that denies him access to the queue.

For examples of policies that cover Amazon SQS actions and resources, see [Example IAM Policies for Amazon SQS \(p. 102\)](#). For more information about writing Amazon SQS policies, go to the [Amazon Simple Queue Service Developer Guide](#).

## Amazon SQS ARNs

For Amazon SQS, queues are the only resource type you can specify in a policy. Following is the Amazon Resource Name (ARN) format for queues:

```
arn:aws:sqs:region:account_ID:queue_name
```

For more information about ARNs, go to [IAM ARNs](#) in *Using IAM*.

Following is an ARN for a queue named my\_queue in the US East (N. Virginia) region, belonging to AWS Account 123456789012.

```
arn:aws:sqs:us-east-1:123456789012:my_queue
```

If you had a queue named my\_queue in each of the different Regions that Amazon SQS supports, you could specify the queues with the following ARN.

```
arn:aws:sqs:*:123456789012:my_queue
```

You can use \* and ? wildcards in the queue name. For example, the following could refer to all the queues Bob has created, which he has prefixed with bob\_.

```
arn:aws:sqs:*:123456789012:bob_*
```

As a convenience to you, Amazon SQS has a queue attribute called `Arn` whose value is the queue's ARN. You can get the value by calling the Amazon SQS `GetQueueAttributes` action.

## Amazon SQS Actions

All Amazon SQS actions that you specify in a policy must be prefixed with the lowercase string `sqs:`. For example, `sqs:CreateQueue`.

Before the introduction of IAM, you could use an Amazon SQS policy with a queue to specify which AWS Accounts have access to the queue. You could also specify the type of access (e.g., `sqs:SendMessage`, `sqs:ReceiveMessage`, etc.). The specific actions you could grant permission for were a subset of the overall set of Amazon SQS actions. When you wrote an Amazon SQS policy and specified `*` to mean "all the Amazon SQS actions", that meant all actions in that subset. That subset originally included:

- `sqs:SendMessage`
- `sqs:ReceiveMessage`
- `sqs:ChangeMessageVisibility`
- `sqs>DeleteMessage`
- `sqs:GetQueueAttributes` (for all attributes except `Policy`)
- `sqs:GetQueueUrl`

With the introduction of IAM, that list of actions expanded to include the following actions:

- `sqs:CreateQueue`
- `sqs>DeleteQueue`
- `sqs:ListQueues`

The actions related to granting and removing permissions from a queue (`sqs:AddPermission`, etc.) are reserved and so don't appear in the preceding two lists. This means that *Users* in the AWS Account can't use those actions. However, the *AWS Account* can use those actions.

## Amazon SQS Keys

Amazon SQS implements the following policy keys, but no others.

### AWS-Wide Policy Keys

- `aws:CurrentTime`—To check for date/time conditions.
- `aws:EpochTime`—To check for date/time conditions using a date in epoch or UNIX time.
- `aws:MultiFactorAuthAge`—To check how long ago (in seconds) the MFA-validated security credentials making the request were issued using Multi-Factor Authentication (MFA). Unlike other keys, if MFA is not used, this key is not present.
- `aws:principaltype`—To check the type of principal (user, account, federated user, etc.) for the current request.
- `aws:SecureTransport`—To check whether the request was sent using SSL. For services that use only SSL, such as Amazon RDS and Amazon Route 53, the `aws:SecureTransport` key has no meaning.

- `aws:SourceArn`—To check the source of the request, using the Amazon Resource Name (ARN) of the source. (This value is available for only some services. For more information, see [Amazon Resource Name \(ARN\)](#) under "Element Descriptions" in the *Amazon Simple Queue Service Developer Guide*.)
- `aws:SourceIp`—To check the IP address of the requester. Note that if you use `aws:SourceIp`, and the request comes from an Amazon EC2 instance, the public IP address of the instance is evaluated.
- `aws:UserAgent`—To check the client application that made the request.
- `aws:userId`—To check the user ID of the requester.
- `aws:username`—To check the user name of the requester, if available.

**Note**

Key names are case sensitive.

## Example IAM Policies for Amazon SQS

This section shows several simple IAM policies for controlling User access to Amazon SQS.

**Note**

In the future, Amazon SQS might add new actions that should logically be included in one of the following policies, based on the policy's stated goals.

### 1: Allow a User to create and use his or her own queues

In this example, we create a policy for Bob that lets him access all Amazon SQS actions, but only with queues whose names begin with the literal string `bob_queue`.

**Note**

Amazon SQS doesn't automatically grant the creator of a queue permission to subsequently use the queue. Therefore, in our IAM policy, we must explicitly grant Bob permission to use all the Amazon SQS actions in addition to `CreateQueue`.

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": "sqs:*",
    "Resource": "arn:aws:sqs:*:123456789012:bob_queue*"
  }
]
```

### 2: Allow developers to write messages to a shared test queue

In this example, we create a group for developers and attach a policy that lets the group use the Amazon SQS `SendMessage` action, but only with the AWS Account's queue named `CompanyTestQueue`.

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": "sqs:SendMessage",
    "Resource": "arn:aws:sqs:*:123456789012:CompanyTestQueue"
  }
]
```

### 3: Allow managers to get the general size of queues

In this example, we create a group for managers and attach a policy that lets the group use the Amazon SQS `GetQueueAttributes` action with all of the AWS Account's queues.

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": "sqs:GetQueueAttributes",
    "Resource": "*"
  }]
}
```

### 4: Allow a partner to send messages to a particular queue

You could do this with an Amazon SQS policy or an IAM policy. Using an Amazon SQS policy might be easier if the partner has an AWS Account. However, anyone in the partner's company who possesses the AWS security credentials could send messages to the queue (and not just a particular User). We'll assume you want to limit access to a particular person (or application), so you need to treat the partner like a User within your own company, and use a IAM policy instead of an Amazon SQS policy.

In this example, we create a group called `WidgetCo` that represents the partner company, then create a User for the specific person (or application) at the partner company who needs access, and then put the User in the group.

We then attach a policy that gives the group `SendMessage` access on the specific queue named `WidgetPartnerQueue`.

We also want to prevent the `WidgetCo` group from doing anything else with queues, so we add a statement that denies permission to any Amazon SQS actions besides `SendMessage` on any queue besides `WidgetPartnerQueue`. This is only necessary if there's a broad policy elsewhere in the system that gives Users wide access to Amazon SQS.

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": "sqs:SendMessage",
    "Resource": "arn:aws:sqs:*:123456789012:WidgetPartnerQueue"
  },
  {
    "Effect": "Deny",
    "NotAction": "sqs:SendMessage",
    "NotResource": "arn:aws:sqs:*:123456789012:WidgetPartnerQueue"
  }]
}
```

## Using Temporary Security Credentials

In addition to creating IAM users with their own security credentials, IAM also enables you to grant temporary security credentials to any user allowing this user to access your AWS services and resources. You can manage users who have AWS accounts; these users are IAM users. You can also manage users

for your system who do not have AWS accounts; these users are called federated users. Additionally, "users" can also be applications that you create to access your AWS resources.

You can use these temporary security credentials in making requests to Amazon SQS. The API libraries compute the necessary signature value using those credentials to authenticate your request. If you send requests using expired credentials Amazon SQS denies the request.

First, use IAM to create temporary security credentials, which include a security token, an Access Key ID, and a Secret Access Key. Second, prepare your string to sign with the temporary Access Key ID and the security token. Third, use the temporary Secret Access Key instead of your own Secret Access Key to sign your Query API request. Finally, when you submit the signed Query API request, don't forget to use the temporary Access Key ID instead of your own Access Key ID and include the security token. For more information about IAM support for temporary security credentials, go to [Granting Temporary Access to Your AWS Resources](#) in *Using IAM*.

### To call an Amazon SQS Query API action using Temporary Security Credentials

1. Request a temporary security token with AWS Identity and Access Management. For more information, go to [Creating Temporary Security Credentials to Enable Access for IAM Users](#) in IAM User Guide. IAM returns a security token, an Access Key ID, and a Secret Access Key.
2. Prepare your Query as you normally would, but use the temporary Access Key ID in place of your own Access Key ID and include the security token. Sign your request using the temporary Secret Access Key instead of your own.
3. Submit your signed query string with the temporary Access Key ID and the security token.

The following example demonstrates how to use temporary security credentials to authenticate an Amazon SQS request.

How you structure the AUTHPARAMS depends on how you are signing your API request. For information on AUTHPARAMS in Signature Version 4, go to [Examples of Signed Signature Version 4 Requests](#).

```
http://sqs.us-east-1.amazonaws.com/  
?Action=CreateQueue  
&DefaultVisibilityTimeout=40  
&QueueName=testQueue  
&Attribute.1.Name=VisibilityTimeout  
&Attribute.1.Value=40  
&Version=2011-10-01  
&Expires=2011-10-18T22%3A52%3A43PST  
&SecurityToken=SecurityTokenValue  
&AWSSecretAccessKey=Access Key ID provided by AWS Security Token Service  
&AUTHPARAMS
```

The following example uses Temporary Security Credentials to send two messages with `SendMessageBatch`.

```
http://sqs.us-east-1.amazonaws.com/  
?Action=SendMessageBatch  
&SendMessageBatchRequestEntry.1.Id=test_msg_001  
&SendMessageBatchRequestEntry.1.MessageBody=test%20message%20body%201  
&SendMessageBatchRequestEntry.2.Id=test_msg_002  
&SendMessageBatchRequestEntry.2.MessageBody=test%20message%20body%202  
&SendMessageBatchRequestEntry.2.DelaySeconds=60  
&Version=2011-10-01  
&Expires=2011-10-18T22%3A52%3A43PST
```

```
&SecurityToken=SecurityTokenValue  
&AWSAccessKeyId=Access Key ID provided by AWS Security Token Service  
&AUTHPARAMS
```

# Monitoring Amazon SQS with CloudWatch

---

Amazon SQS and CloudWatch are integrated so you can use CloudWatch to easily collect, view, and analyze metrics for your Amazon SQS queues. Once you have configured CloudWatch for Amazon SQS, you can gain better insight into the performance of your Amazon SQS queues and applications. For example, you can monitor the `NumberOfEmptyReceives` metric to make sure that your application isn't spending too much of its time polling for new messages. You can also set an alarm to send you an email notification if a specified threshold is met for an Amazon SQS metric, such as `NumberOfMessagesReceived`. For a list of all the metrics that Amazon SQS sends to CloudWatch, see [Amazon SQS Metrics \(p. 107\)](#).

The metrics you configure with CloudWatch for your Amazon SQS queues are automatically collected and pushed to CloudWatch every five minutes. These metrics are gathered on all queues that meet the CloudWatch guidelines for being active. A queue is considered active by CloudWatch for up to six hours from the last activity (i.e., any API call) on the queue.

## Note

There is no charge for the Amazon SQS metrics reported in CloudWatch; they are provided as part of the Amazon SQS service.

## Access CloudWatch Metrics for Amazon SQS

You can monitor metrics for Amazon SQS using the CloudWatch console, CloudWatch's own command line interface (CLI), or programmatically using the CloudWatch API. The following procedures show you how to access the metrics using these different options.

### To view metrics using the CloudWatch console

1. Sign in to the AWS Management Console and open the CloudWatch console at <https://console.aws.amazon.com/cloudwatch/>.
2. Click **View Metrics**.
3. From the **Viewing** drop-down menu select **SQS: Queue Metrics** to show the available metrics for each queue.
4. Click a specific metric in the **MetricName** column to see more detail, such as a graph of the data collected.

### To access metrics from the CloudWatch CLI

- Call `mon-get-stats`. You can learn more about this and other metrics-related functions in the [Amazon CloudWatch Developer Guide](#).

### To access metrics from the CloudWatch API

- Call `GetMetricStatistics`. You can learn more about this and other metrics-related functions in the [Amazon CloudWatch API Reference](#).

## Set CloudWatch Alarms for Amazon SQS Metrics

CloudWatch also allows you to set alarms when a threshold is met for a metric. For example, you could set an alarm for the metric, **NumberOfEmptyReceives**, so that when your specified threshold number is met within the sampling period, then an email notification would be sent to inform you of the event.

### To set alarms using the CloudWatch console

1. Sign in to the AWS Management Console and open the CloudWatch console at <https://console.aws.amazon.com/cloudwatch/>.
2. Click **Alarms**, and then click the **Create Alarm** button. This launches the **Create Alarm Wizard**.
3. Scroll through the Amazon SQS metrics to locate the metric you want to place an alarm on. Select the metric to create an alarm on and click **Continue**.
4. Fill in the **Name**, **Description**, **Threshold**, and **Time** values for the metric, and click **Continue**.
5. Choose **Alarm** as the alarm state. If you want CloudWatch to send you an email when the alarm state is reached, either select a preexisting Amazon SNS topic or click **Create New Email Topic**. If you click **Create New Email Topic**, you can set the name and email addresses for a new topic. This list will be saved and appear in the drop-down box for future alarms. Click **Continue**.

#### Note

If you use **Create New Email Topic** to create a new Amazon SNS topic, the email addresses must be verified before they will receive notifications. Emails are sent only when the alarm enters an alarm state. If this alarm state change happens before the email addresses are verified, they will not receive a notification.

6. At this point, the **Create Alarm Wizard** gives you a chance to review the alarm you're about to create. If you need to make any changes, you can use the **Edit** links on the right. Once you're satisfied, click **Create Alarm**.

For more information about using CloudWatch and alarms, see the [CloudWatch Documentation](#)

## Amazon SQS Metrics

Amazon SQS sends the following metrics to CloudWatch.

Metric	Description
NumberOfMessagesSent	The number of messages added to a queue.  Units: <i>Count</i>  Valid Statistics: Sum



<b>Metric</b>	<b>Description</b>
SentMessageSize	<p>The size of messages added to a queue.</p> <p>Units: <i>Bytes</i></p> <p>Valid Statistics: Minimum, Maximum, Average, and Count</p>
NumberOfMessagesReceived	<p>The number of messages returned by calls to the <code>ReceiveMessage</code> API action.</p> <p>Units: <i>Count</i></p> <p>Valid Statistics: Sum</p>
NumberOfEmptyReceives	<p>The number of <code>ReceiveMessage</code> API calls that did not return a message.</p> <p>Units: <i>Count</i></p> <p>Valid Statistics: Sum</p>
NumberOfMessagesDeleted	<p>The number of messages deleted from the queue.</p> <p>Units: <i>Count</i></p> <p>Valid Statistics: Sum</p>
ApproximateNumberOfMessagesDelayed	<p>The number of messages in the queue that are delayed and not available for reading immediately. This can happen when the queue is configured as a delay queue or when a message has been sent with a delay parameter.</p> <p>Units: <i>Count</i></p> <p>Valid Statistics: Average</p>
ApproximateNumberOfMessagesVisible	<p>The number of messages available for retrieval from the queue.</p> <p>Units: <i>Count</i></p> <p>Valid Statistics: Average</p>
ApproximateNumberOfMessagesNotVisible	<p>The number of messages that are "in flight." Messages are considered in flight if they have been sent to a client but have not yet been deleted or have not yet reached the end of their visibility window.</p> <p>Units: <i>Count</i></p> <p>Valid Statistics: Average</p>

# Logging Amazon SQS API Calls By Using AWS CloudTrail

---

Amazon SQS is integrated with CloudTrail, a service that captures API calls made by or on behalf of Amazon SQS in your AWS account and delivers the log files to an Amazon S3 bucket that you specify. CloudTrail captures API calls made from the Amazon SQS console or from the Amazon SQS API. Using the information collected by CloudTrail, you can determine what request was made to Amazon SQS, the source IP address from which the request was made, who made the request, when it was made, and so on. To learn more about CloudTrail, including how to configure and enable it, see the [AWS CloudTrail User Guide](#).

## Amazon SQS Information in CloudTrail

When CloudTrail logging is enabled in your AWS account, API calls made to Amazon SQS actions are tracked in log files. Amazon SQS records are written together with other AWS service records in a log file. CloudTrail determines when to create and write to a new file based on a time period and file size.

The following actions are supported:

- [AddPermission](#)
- [CreateQueue](#)
- [DeleteQueue](#)
- [RemovePermission](#)
- [SetQueueAttributes](#)

Every log entry contains information about who generated the request. The user identity information in the log helps you determine whether the request was made with root or IAM user credentials, with temporary security credentials for a role or federated user, or by another AWS service. For more information, see the **userIdentity** field in the [CloudTrail Event Reference](#).

You can store your log files in your bucket for as long as you want, but you can also define Amazon S3 lifecycle rules to archive or delete log files automatically. By default, your log files are encrypted by using Amazon S3 server-side encryption (SSE).

You can choose to have CloudTrail publish Amazon SNS notifications when new log files are delivered if you want to take quick action upon log file delivery. For more information, see [Configuring Amazon SNS Notifications](#).

You can also aggregate Amazon SQS log files from multiple AWS regions and multiple AWS accounts into a single Amazon S3 bucket. For more information, see [Aggregating CloudTrail Log Files to a Single Amazon S3 Bucket](#).

## Understanding Amazon SQS Log File Entries

CloudTrail log files contain one or more log entries where each entry is made up of multiple JSON-formatted events. A log entry represents a single request from any source and includes information about the requested action, any parameters, the date and time of the action, and so on. The log entries are not guaranteed to be in any particular order. That is, they are not an ordered stack trace of the public API calls.

### AddPermission

The following example shows a CloudTrail log entry for AddPermission:

```
{
  "Records": [
    {
      "eventVersion": "1.01",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "EX_PRINCIPAL_ID",
        "arn": "arn:aws:iam::123456789012:user/Alice",
        "accountId": "123456789012",
        "accessKeyId": "EXAMPLE_KEY_ID",
        "userName": "Alice"
      },
      "eventTime": "2014-07-16T00:44:19Z",
      "eventSource": "sqs.amazonaws.com",
      "eventName": "AddPermission",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "192.0.2.0",
      "userAgent": "Mozilla/5.0 (X11; Linux x86_64; rv:24.0) Gecko/20100101
Firefox/24.0",
      "requestParameters": {
        "actions": [
          "SendMessage"
        ],
        "awsAccountIds": [
          "123456789012"
        ],
        "label": "label",
        "queueUrl": "http://test-sqs.amazon.com/123456789012/hello1"
      },
      "responseElements": null,
      "requestID": "334cccd-b9bb-50fa-abdb-80f274981d60",
      "eventID": "0552b000-09a3-47d6-a810-c5f9fd2534fe"
    }
  ]
}
```

## CreateQueue

The following example shows a CloudTrail log entry for CreateQueue:

```
{
  "Records": [
    {
      "eventVersion": "1.01",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "EX_PRINCIPAL_ID",
        "arn": "arn:aws:iam::123456789012:user/Alice",
        "accountId": "123456789012",
        "accessKeyId": "EXAMPLE_KEY_ID",
        "userName": "Alice"
      },
      "eventTime": "2014-07-16T00:42:42Z",
      "eventSource": "sqs.amazonaws.com",
      "eventName": "CreateQueue",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "192.0.2.0",
      "userAgent": "Mozilla/5.0 (X11; Linux x86_64; rv:24.0) Gecko/20100101
Firefox/24.0",
      "requestParameters": {
        "queueName": "hello1"
      },
      "responseElements": {
        "queueUrl": "http://test-sqs.amazon.com/123456789012/hello1"
      },
      "requestID": "49ebdb7-5cd3-5323-8a00-f1889011fee9",
      "eventID": "68f4e71c-4f2f-4625-8378-130ac89660b1"
    }
  ]
}
```

## DeleteQueue

The following example shows a CloudTrail log entry for DeleteQueue:

```
{
  "Records": [
    {
      "eventVersion": "1.01",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "EX_PRINCIPAL_ID",
        "arn": "arn:aws:iam::123456789012:user/Alice",
        "accountId": "123456789012",
        "accessKeyId": "EXAMPLE_KEY_ID",
        "userName": "Alice"
      },
      "eventTime": "2014-07-16T00:44:47Z",
      "eventSource": "sqs.amazonaws.com",
      "eventName": "DeleteQueue",
      "awsRegion": "us-east-1",
```

```
    "sourceIPAddress": "192.0.2.0",
    "userAgent": "Mozilla/5.0 (X11; Linux x86_64; rv:24.0) Gecko/20100101
Firefox/24.0",
    "requestParameters": {
      "queueUrl": "http://test-sqs.amazon.com/123456789012/hello1"
    },
    "responseElements": null,
    "requestID": "e4c0cc05-4faa-51d5-aab2-803a8294388d",
    "eventID": "af1bb158-6443-4b4d-abfd-1b867280d964"
  }
]
}
```

## RemovePermission

The following example shows a CloudTrail log entry for RemovePermission:

```
{
  "Records": [
    {
      "eventVersion": "1.01",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "EX_PRINCIPAL_ID",
        "arn": "arn:aws:iam:123456789012:user/Alice",
        "accountId": "123456789012",
        "accessKeyId": "EXAMPLE_KEY_ID",
        "userName": "Alice"
      },
      "eventTime": "2014-07-16T00:44:36Z",
      "eventSource": "sqs.amazonaws.com",
      "eventName": "RemovePermission",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "192.0.2.0",
      "userAgent": "Mozilla/5.0 (X11; Linux x86_64; rv:24.0) Gecko/20100101
Firefox/24.0",
      "requestParameters": {
        "label": "label",
        "queueUrl": "http://test-sqs.amazon.com/123456789012/hello1"
      },
      "responseElements": null,
      "requestID": "48178821-9c2b-5be0-88bf-c41e5118162a",
      "eventID": "fed8a623-3fe9-4e64-9543-586d9e500159"
    }
  ]
}
```

## SetQueueAttributes

The following example shows a CloudTrail log entry for SetQueueAttributes:

```
{
  "Records": [
    {
```

```
"eventVersion": "1.01",
"userIdentity": {
  "type": "IAMUser",
  "principalId": "EX_PRINCIPAL_ID",
  "arn": "arn:aws:iam::123456789012:user/Alice",
  "accountId": "123456789012",
  "accessKeyId": "EXAMPLE_KEY_ID",
  "userName": "Alice"
},
"eventTime": "2014-07-16T00:43:15Z",
"eventSource": "sqs.amazonaws.com",
"eventName": "SetQueueAttributes",
"awsRegion": "us-east-1",
"sourceIPAddress": "192.0.2.0",
"userAgent": "Mozilla/5.0 (X11; Linux x86_64; rv:24.0) Gecko/20100101
Firefox/24.0",
"requestParameters": {
  "attributes": {
    "VisibilityTimeout": "100"
  },
  "queueUrl": "http://test-sqs.amazon.com/123456789012/hello1"
},
"responseElements": null,
"requestID": "7f15d706-f3d7-5221-b9ca-9b393f349b79",
"eventID": "8b6fb2dc-2661-49b1-b328-94317815088b"
}
]
}
```

# Appendix A: Increasing Throughput with Horizontal Scaling and Batching

---

*By Marc Levy, July 2012*

Amazon SQS queues can deliver very high throughput (many thousands of messages per second). The key to achieving this throughput is to horizontally scale message producers and consumers. In addition, you can use the batching actions in the Amazon SQS API to send, receive, or delete up to 10 messages at a time. In conjunction with horizontal scaling, batching achieves a given throughput with fewer threads, connections, and requests than would be required by individual message requests. Because Amazon SQS charges by the request instead of by the message, batching can also substantially reduce costs.

This appendix discusses horizontal scaling and batching in more detail. It then walks through a simple example that you can try out yourself. It also briefly discusses Amazon SQS throughput metrics that you can monitor by using CloudWatch.

## Horizontal Scaling

Because you access Amazon SQS through an HTTP request-response protocol, the request latency (the time interval between initiating a request and receiving a response) limits the throughput that you can achieve from a single thread over a single connection. For example, if the latency from an Amazon Elastic Compute Cloud (Amazon EC2) based client to Amazon SQS in the same region averages around 20 ms, the maximum throughput from a single thread over a single connection will average 50 operations per second.

Horizontal scaling means increasing the number of your message producers (making `SendMessage` requests) and consumers (making `ReceiveMessage` and `DeleteMessage` requests) in order to increase your overall queue throughput. You can scale horizontally by increasing the number of threads on a client, adding clients, or both. You should achieve essentially linear gains in queue throughput as you add more clients. For example, if you double the number of clients, you will get twice the throughput.

### Important

As you scale horizontally, you need to ensure that the Amazon SQS that you are using has enough connections or threads to support the number of concurrent message producers and consumers that will be sending requests and receiving responses. For example, by default, instances of the AWS SDK for Java [AmazonSQSClient class](#) maintain at most 50 connections to Amazon SQS. To create additional concurrent producers and consumers, you'll need to adjust that limit. For example, in the [AWS SDK for Java](#), you can adjust the maximum number of allowable producer and consumer threads on an `AmazonSQSClient` object with this line of code:

```
AmazonSQS sqsClient = new AmazonSQSClient(credentials,
                                           new ClientConfiguration().withMaxConnections(producerCount + consumerCount));
```

For the SDK for Java asynchronous client [AmazonSQSAsyncClient](#), you'll also need to make sure there are enough threads available. For more information, consult the documentation for the SDK library that you are using.

## Batching

The batching actions in the Amazon SQS API ([SendMessageBatch](#) and [DeleteMessageBatch](#)), which were introduced in October 2011 (WSDL 2011-10-01), can further optimize throughput by processing up to ten messages at a time. [ReceiveMessage](#) can process ten messages at a time, so there is no `ReceiveMessageBatch` action.

The basic idea of batching is to perform more work in each round trip to the service (e.g., sending multiple messages with a single `SendMessageBatch` request), and to distribute the latency of the batch operation over the multiple messages in the batch request, as opposed to accepting the entire latency for a single message (for example, a `SendMessage` request). Because each round-trip carries more work, batch requests make more efficient use of threads and connections and so improve throughput. Amazon SQS charges by the request, so the cost can be greatly reduced when fewer requests are processing the same number of messages. Moreover, fewer threads and connections reduce client-side resource utilization and can reduce client-side cost by doing the same work with smaller or fewer hosts.

Batching does introduce a bit of complication for the application. For example, the application has to accumulate the messages before sending them and it will sometimes have to wait longer for a response, but batching can be effective in the following circumstances:

- Your application is generating a lot of messages in a short time, so the delay is never very long.
- A message consumer fetches messages from a queue at its discretion, as opposed to typical message producers that need to send messages in response to events they do not control.

### Important

A batch request (`SendMessageBatch` or `DeleteMessageBatch`) may succeed even though individual messages in the batch have failed. After a batch request, you should always check for individual message failures and retry them if necessary.



## Example

The example presented in this section implements a simple producer-consumer pattern. The complete example is available as a free download at <https://s3.amazonaws.com/cloudformation-examples/sqs-producer-consumer-sample.tar>. The resources that are deployed by each template are described later in this section.

The code for the samples is available on the provisioned instances in `/tmp/sqs-producer-consumer-sample/src`. The command line for the configured run is in `/tmp/sqs-producer-consumer-sample/command.log`.

The main thread spawns a number of producer and consumer threads that process 1 KB messages for a specified time. The example includes producers and consumers that make single-operation requests and others that make batch requests.

In the program, each producer thread sends messages until the main thread stops the producer thread. The `producedCount` object tracks the number of messages produced by all producer threads. Error handling is simple: if there is an error, the program exits the `run()` method. Requests that fail on transient errors are, by default, retried three times by the `AmazonSQSClient`, so very few such errors are surfaced. The retry count can be configured as necessary to reduce the number of exceptions that are thrown. The `run()` method on the message producer is implemented as follows:

```
try {
    while (!stop.get()) {
        sqsClient.sendMessage(new SendMessageRequest(queueUrl, theMessage));
        producedCount.incrementAndGet();
    }
} catch (AmazonClientException e) {
    // By default AmazonSQSClient retries calls 3 times before failing,
    // so when this rare condition occurs, simply stop.
    log.error("Producer: " + e.getMessage());
    System.exit(1);
}
```

The batch producer is much the same. One noteworthy difference is the need to retry failed individual batch entries:

```
SendMessageBatchResult batchResult = sqsClient.sendMessageBatch(batchRequest);

if (!batchResult.getFailed().isEmpty()) {
    log.warn("Producer: retrying sending " + batchResult.getFailed().size() + "
messages");
    for (int i = 0, n = batchResult.getFailed().size(); i < n; i++)
        sqsClient.sendMessage(new SendMessageRequest(queueUrl, theMessage));
}
```

The consumer `run()` method is as follows:

```
while (!stop.get()) {
    result = sqsClient.receiveMessage(new ReceiveMessageRequest(queueUrl));

    if (!result.getMessages().isEmpty()) {
        m = result.getMessages().get(0);
        sqsClient.deleteMessage(new DeleteMessageRequest(queueUrl,
```

```
        m.getReceiptHandle());  
  
        consumedCount.incrementAndGet();  
    }  
}
```

Each consumer thread receives and deletes messages until it is stopped by the main thread. The `consumedCount` object tracks the number of messages that are consumed by all consumer threads, and the count is periodically logged. The batch consumer is similar, except that up to ten messages are received at a time, and it uses [DeleteMessageBatch](#) instead of [DeleteMessage](#).

## Running the Example

You can use the AWS CloudFormation templates provided to run the example code in three different configurations: single host with the single operation requests, two hosts with the single operation requests, one host with the batch requests.

### Important

The complete sample is available in a single .tar file. The resources that are deployed by each template are described later in this section.

The code for the samples is available on the provisioned instance(s) in `/tmp/sqs-producer-consumer-sample/src`. The command line for the configured run is in `/tmp/sqs-producer-consumer-sample/command.log`.

The default duration (20 minutes) is set to provide three or four 5-minute CloudWatch data points of volume metrics. The Amazon EC2 cost for each run will be the `m1.large` instance cost. The Amazon SQS cost varies based on the API call rate for each sample, and that should range between approximately 38,000 API calls / min for the batching sample and 380,000 API calls / min for the two host single API sample. For example, a run of the single API sample on a single host should cost approximately 1 instance hour of an `m1.large` (large standard on demand instance, \$0.32 as of July 2012) and 20 min x 190,000 API calls / min x \$1 / 1,000,000 API calls = \$3.80 for Amazon SQS operations with the default 20 min duration (as of July 2012, check current pricing).

If you want to deploy the AWS CloudFormation stack in a region other than the US East (N. Virginia) region, in the Region box of the AWS CloudFormation console, click the region that you want.

### To run the example

1. Click the link below that corresponds to the stack that you want to launch:

- **Single Operation API, One Host:** The `SQS_Sample_Base_Producer_Consumer.template` sample template uses the single operation form of Amazon SQS API requests: `SendMessage`, `ReceiveMessage`, and `DeleteMessage`. A single `m1.large` Amazon EC2 instance animates 16 producer threads and 32 consumer threads.

To view the template, go to [https://s3.amazonaws.com/cloudformation-templates-us-east-1/SQS\\_Sample\\_Base\\_Producer\\_Consumer.template](https://s3.amazonaws.com/cloudformation-templates-us-east-1/SQS_Sample_Base_Producer_Consumer.template)

- **Single Operation API, Two Hosts:** `SQS_Sample_Base_Producer_Consumer_x2.template` sample template uses the single operation form of Amazon SQS API requests, but instead of a single `m1.large` Amazon EC2 instance, it uses two, each with 16 producer threads and 32 consumer threads for a total of 32 producers and 64 consumers. It illustrates Amazon SQS' elasticity with throughput increasing proportionally to the greater number of producers and consumers.

To view the template, go to [https://s3.amazonaws.com/cloudformation-templates-us-east-1/SQS\\_Sample\\_Base\\_Producer\\_Consumer\\_x2.template](https://s3.amazonaws.com/cloudformation-templates-us-east-1/SQS_Sample_Base_Producer_Consumer_x2.template)

- [Batch API, One Host](#): The `SQS_Sample_Batch_Producer_Consumer.template` sample template uses the batch form of Amazon SQS API requests on a single m1.large Amazon EC2 instance with 12 producer threads and 20 consumer threads.

To view the template, go to [https://s3.amazonaws.com/cloudformation-templates-us-east-1/SQS\\_Sample\\_Batch\\_Producer\\_Consumer.template](https://s3.amazonaws.com/cloudformation-templates-us-east-1/SQS_Sample_Batch_Producer_Consumer.template)

2. If you are prompted, sign in to the AWS Management Console.
3. In the **Create Stack** wizard, on the **Select Template** page, click **Continue**.
4. On the **Specify Parameters** page, specify how long the program should run, whether or not you want to automatically terminate the Amazon EC2 instances when the run is complete, and provide an Amazon EC2 key pair so that you can access the instances that are running the sample. Here is an example:

**Create Stack** Cancel X

SELECT TEMPLATE    SPECIFY PARAMETERS    REVIEW

**Template Description:** Single EC2 m1.large producer-consumer processing (single operation API) 1K messages for the specified duration

**Specify Parameters**

Below are the parameters associated with your CloudFormation template. You may review and proceed with the default parameters or make customizations as needed below.

**DurationMinutes**    20  
Run duration in minutes (max 60)

**TerminateEC2Inst**    true  
Terminate the producer-consumer EC2 instance once the run is complete?

**KeyName**      
Name of an existing EC2 KeyPair to enable SSH access to the producer-consumer instance

**I acknowledge that this template may create IAM resources**

< Back    Continue >

5. Select the **I acknowledge that this template may create IAM resources** check box. All templates create an AWS Identity and Access Management (IAM) user so that the producer-consumer program can access the queue.
6. When all the settings are as you want them, click **Continue**.
7. On the **Review** page, review the settings. If they are as you want them, click **Continue**. If not, click **Back** and make the necessary changes.
8. On the final page of the wizard, click **Close**. Stack deployment may take several minutes.

To follow the progress of stack deployment, in the AWS CloudFormation console, click the sample stack. In the lower pane, click the **Events** tab. After the stack is created, it should take less than 5 minutes for the sample to start running. When it does, you can see the queue in the Amazon SQS console.

To monitor queue activity, you can do the following:

- Access the client instance, and open its output log file (`/tmp/sqs-producer-consumer-sample/output.log`) for a tally of messages produced and consumed so far. This tally is updated once per second.
- In the [Amazon SQS console](#), observe changes in the **Message Available** and **Messages in Flight** numbers.

In addition, after a delay of up to 15 minutes after the queue is started, you can monitor the queue in CloudWatch as described later in this topic.

Although the templates and samples have safeguards to prevent excessive use of resources, it is best to delete your AWS CloudFormation stacks when you are done running the samples. To do so, in the [Amazon SQS console](#), click the stack that you want to delete, and then click **Delete Stack**. When the resources are all deleted, CloudWatch metrics will all drop to zero.

## Monitoring Volume Metrics from Example Run

Amazon SQS automatically generates volume metrics for messages sent, received, and deleted. You can access those metrics and others through the [CloudWatch console](#). The metrics can take up to 15 minutes after the queue starts to become available. To manage the search result set, click **Search**, and then select the check boxes that correspond to the queues and metrics that you want to monitor.

Here is the `NumberOfMessagesSent` metric for consecutive runs of the three samples. Your results may vary somewhat, but the results should be qualitatively similar:



- The `NumberOfMessagesReceived` and `NumberOfMessagesDeleted` metrics show the same pattern, but we have omitted them from this graph to reduce clutter.
- The first sample (single operation API on a single `m1.large`) delivers approximately 210,000 messages over 5 minutes, or about 700 messages per second, with the same throughput for receive and delete operations.
- The second sample (single operation API on two `m1.large` instances) delivers roughly double that throughput: approximately 440,000 messages in 5 minutes, or about 1,450 messages per second, with the same throughput for receive and delete operations.
- The last sample (batch API on a single `m1.large`) delivers over 800,000 messages in 5 minutes, or about 2,500 messages per second, with the same throughput for received and deleted messages. With a batch size of 10, these messages are processed with far fewer requests and therefore at lower cost.

## Appendix B: Client-Side Buffering and Request Batching

---

The AWS SDK for Java (<http://aws.amazon.com/sdkforjava/>) includes a buffered asynchronous client, `AmazonSQSBufferedAsyncClient`, for accessing Amazon SQS. This new client allows for easier request batching by enabling client-side buffering, where calls made from the client are first buffered and then sent as a batch request to Amazon SQS.

Client-side buffering allows up to 10 requests to be buffered and sent as a batch request instead of sending each request separately. As a result, your cost of using Amazon SQS decreases as you reduce the number of requests sent to the service. `AmazonSQSBufferedAsyncClient` buffers both synchronous and asynchronous calls. Batched requests and support for long polling can also help increase throughput (the number of messages transmitted per second). For more information, see [Amazon SQS Long Polling](#) (p. 41) and [Appendix A: Increasing Throughput with Horizontal Scaling and Batching](#) (p. 114).

Migrating from the asynchronous client, `AmazonSQSAsyncClient`, to the buffered asynchronous client, `AmazonSQSBufferedAsyncClient`, should require only minimal changes to your existing code. This is because `AmazonSQSBufferedAsyncClient` implements the same interface as `AmazonSQSAsyncClient`.

### Getting Started with AmazonSQSBufferedAsyncClient

Before you begin using the example code in this section, you must first install the AWS SDK for Java and set up your AWS credentials. For instructions, see [Getting Started with the AWS SDK for Java](#).

The following code sample shows how to create a new `AmazonSQSBufferedAsyncClient` based on the `AmazonSQSAsyncClient`.

```
// Create the basic Amazon SQS async client
AmazonSQSAsync sqsAsync = new AmazonSQSAsyncClient();

// Create the buffered client
AmazonSQSAsync bufferedSqs = new AmazonSQSBufferedAsyncClient(sqsAsync);
```

After you have created the new `AmazonSQSBufferedAsyncClient`, you can make calls to it as you do with the `AmazonSQSAsyncClient`, as the following code sample demonstrates.

```
CreateQueueRequest createRequest = new CreateQueueRequest().withQueueName("MyTestQueue");

CreateQueueResult res = bufferedSqs.createQueue(createRequest);

SendMessageRequest request = new SendMessageRequest();
String body = "test message_" + System.currentTimeMillis();
request.setRequestBody(body);
request.setQueueUrl(res.getQueueUrl());

SendMessageResult sendResult = bufferedSqs.sendMessage(request);

ReceiveMessageRequest receiveRq = new ReceiveMessageRequest().withMaxNumberOfMessages(1).withQueueUrl(queueUrl);
ReceiveMessageResult rx = bufferedSqs.receiveMessage(receiveRq);
```

## Advanced Configuration

`AmazonSQSBufferedAsyncClient` is pre-configured with settings that will work for most use cases. If you would like to configure it yourself, you can use the `QueueBufferConfig` class to do so. Just create an instance of `QueueBufferConfig` with the settings you want and supply it to the `AmazonSQSBufferedAsyncClient` constructor, as the following sample code shows.

```
// Create the basic Amazon SQS async client
AmazonSQSAsync sqsAsync = new AmazonSQSAsyncClient();

QueueBufferConfig config = new QueueBufferConfig().withMaxInflightReceiveBatches(5).withMaxDoneReceiveBatches(15);

// Create the buffered client
AmazonSQSAsync bufferedSqs = new AmazonSQSBufferedAsyncClient(sqsAsync, config);
```

The parameters you can use for configuring `QueueBufferConfig` are as follows:

- `longPoll`—if this parameter is set to `true`, `AmazonBufferedAsyncClient` attempts to use long-polling when retrieving messages. The default value is `true`.
- `longPollWaitTimeoutSeconds`—the maximum amount of time, in seconds, that a receive message call blocks on the server waiting for messages to appear in the queue before returning with an empty receive result. This setting has no impact if long polling is disabled. The default value of this setting is 20 seconds.
- `maxBatchOpenMs`—the maximum amount of time, in milliseconds, that an outgoing call waits for other calls of the same type to batch with. The higher the setting, the fewer batches are required to perform the same amount of work. Of course, the higher the setting, the more the first call in a batch has to spend waiting. If this parameter is set to zero, submitted requests do not wait for other requests, effectively disabling batching. The default value of this setting is 200 milliseconds.

- *maxBatchSize*—the maximum number of messages that will be batched together in a single batch request. The higher the setting, the fewer batches will be required to carry out the same number of requests. The default value of this setting is 10 requests per batch, which is also the maximum batch size currently allowed by Amazon SQS.
- *maxBatchSizeBytes*—the maximum size of a message batch, in bytes, that the client attempts to send to Amazon SQS. The default value is 256 KB, which is also the maximum message and batch size currently allowed by Amazon SQS.
- *maxDoneReceiveBatches*—the maximum number of receive batches `AmazonBufferedAsyncClient` prefetches and stores on the client side. The higher the setting, the more receive requests can be satisfied without having to make a call to Amazon SQS server. However, the more messages are pre-fetched, the longer they will sit in the buffer, which means that their visibility timeout will be expiring. If this parameter is set to zero, all pre-fetching of messages is disabled and messages are retrieved only on demand. The default value is 10 batches.
- *maxInflightOutboundBatches*—the maximum number of active outbound batches that can be processed at the same time. The higher the setting, the faster outbound batches can be sent (subject to other limits, such as CPU or bandwidth). The higher the setting, the more threads are consumed by the `AmazonSQSBufferedAsyncClient`. The default value is 5 batches.
- *maxInflightReceiveBatches*—the maximum number of active receive batches that can be processed at the same time. The higher the setting, the more messages can be received (subject to other limits, such as CPU or bandwidth, are hit). Although, the higher the setting, the more threads will be consumed by the `AmazonSQSBufferedAsyncClient`. If this parameter is set to 0, all pre-fetching of messages is disabled and messages are only retrieved on demand. The default value is 10 batches.
- *visibilityTimeoutSeconds*—if this parameter is set to a positive nonzero value, this visibility timeout overrides the visibility timeout set on the queue from which messages are retrieved. A visibility timeout of zero seconds is not supported. The default value is -1, which means the default queue setting is used.

## Appendix C: Subscribe Queue to Amazon SNS Topic

---

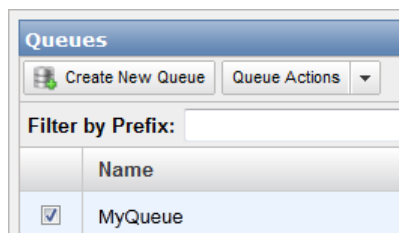
You can now subscribe an Amazon SQS queue to an Amazon SNS topic using the AWS Management Console for Amazon SQS, which simplifies the process. For example, you can choose from the list of available topics for the selected queue. Amazon SQS then manages the subscription of the queue to the topic and the addition of the necessary permissions. When a message is published to the topic, Amazon SNS sends an Amazon SQS message to the subscribed queue. For more information about Amazon SNS, see [Get Started with Amazon SNS](#). For more information about Amazon SQS, see [Get Started with Amazon SQS](#).

### Subscribe Queue to Amazon SNS Topic with the AWS Management Console

The following steps assume you have already created a queue and an Amazon SNS topic.

#### To subscribe a queue to an Amazon SNS topic with the AWS Management Console

1. Sign in to the AWS Management Console and open the Amazon SQS console at <https://console.aws.amazon.com/sqs/>.
2. Select the queue to which you want to subscribe an Amazon SNS topic.

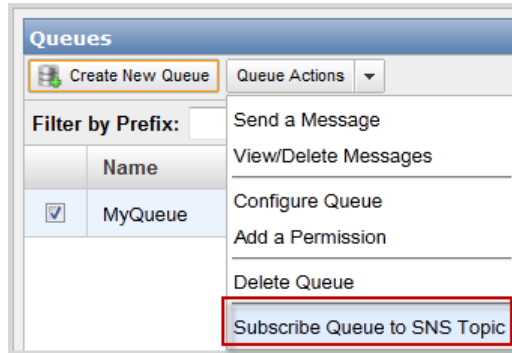


3. Select **Subscribe Queue to SNS Topic** from the **Queue Actions** drop-down list.



## Amazon Simple Queue Service Developer Guide

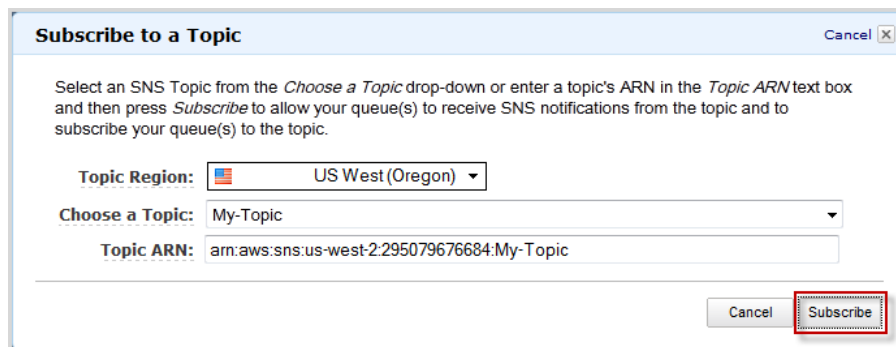
### Subscribe Queue to Amazon SNS Topic with the AWS Management Console



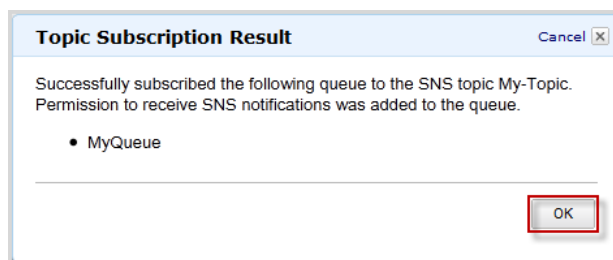
- From the **Choose a Topic** drop-down list, select an Amazon SNS topic to subscribe the queue to and then click **Subscribe**.

#### Note

You can also enter the ARN of the Amazon SNS topic in the **Topic ARN:** box. This is useful when you want to subscribe the queue to an Amazon SNS topic from an AWS account other than the one you used to create the queue. It's also useful if the Amazon SNS topic is not listed in the **Choose a Topic** drop-down list.



- In the **Topic Subscription Result** dialog box, click **OK**.



You can verify the results of the topic's queue subscription by publishing to the topic and viewing the message that the topic sends to the queue. For detailed steps, see [Test it out by publishing a message to the topic and reading the message from the queue](#).

# Amazon SQS Resources

---

The following table lists related resources that you'll find useful as you work with this service.

Resource	Description
<a href="#">Amazon Simple Queue Service Getting Started Guide</a>	The getting started guide provides a quick tutorial of the service based on a simple use case. Examples and instructions in multiple programming languages are included.
<a href="#">Amazon Simple Queue Service API Reference</a>	The API reference gives the WSDL location; complete descriptions of the API actions, parameters, and data types; and a list of errors that the service returns.
<a href="#">Amazon SQS Release Notes</a>	The release notes give a high-level overview of the current release. They specifically note any new features, corrections, and known issues.
<a href="#">Product information for Amazon SQS</a>	The primary web page for information about Amazon SQS.
<a href="#">Discussion Forums</a>	A community-based forum for developers to discuss technical questions related to Amazon SQS.
<a href="#">AWS Premium Support Information</a>	The primary web page for information about AWS Premium Support, a one-on-one, fast-response support channel to help you build and run applications on AWS infrastructure services.

# Document History

The following table describes the important changes to the documentation since the last release of the *Amazon Simple Queue Service Developer Guide*.

- **API version:** 2012-11-05
- **Latest documentation update:** December 29, 2014

Change	Description	Date Changed
New feature	Amazon SQS now enables you to use JMS (Java Message Service) with Amazon SQS queues. For more information, see <a href="#">Using JMS with Amazon SQS</a> in the <i>Amazon SQS Developer Guide</i> .	December 29, 2014
New feature	Amazon SQS now enables you to delete the messages in a queue by using the <code>PurgeQueue</code> API. For more information, see <a href="#">PurgeQueue</a> in the <i>Amazon SQS API Reference</i> .	December 8, 2014
Update	Updated information about access keys. For more information, see <a href="#">Your Access Keys (p. 62)</a> .	August 4, 2014
New feature	Amazon SQS now enables you to log API calls by using AWS CloudTrail. For more information, see <a href="#">Logging Amazon SQS API Calls By Using AWS CloudTrail (p. 109)</a> .	July 16, 2014
New feature	Amazon SQS now provides support for message attributes. For more information, see <a href="#">Using Amazon SQS Message Attributes (p. 32)</a> .	May 6, 2014
New feature	Amazon SQS now provides support for dead letter queues. For more information, see <a href="#">Using Amazon SQS Dead Letter Queues (p. 73)</a> .	January 29, 2014
New console feature	You can now subscribe an Amazon SQS queue to an Amazon SNS topic using the AWS Management Console for Amazon SQS, which simplifies the process. For more information, see <a href="#">Appendix C: Subscribe Queue to Amazon SNS Topic (p. 123)</a> .	November 21, 2012
New feature	The 2012-11-05 API version of Amazon SQS adds support for signature version 4, which provides improved security and performance. For more information about signature version 4, see <a href="#">Query Request Authentication (p. 67)</a> .	November 5, 2012

Change	Description	Date Changed
New feature	The AWS SDK for Java now includes a buffered asynchronous client, <code>AmazonSQSBufferedAsyncClient</code> , for accessing Amazon SQS. This new client allows for easier request batching by enabling client-side buffering, where calls made from the client are first buffered and then sent as a batch request to Amazon SQS. For more information about client-side buffering and request batching, see <a href="#">Appendix B: Client-Side Buffering and Request Batching (p. 120)</a> .	November 5, 2012
New feature	The 2012-11-05 API version of Amazon SQS adds long polling support. Long polling allows for Amazon SQS to wait for a specified amount time for a message to be available instead of returning an empty response if one is not available. For more information about long polling, see <a href="#">Amazon SQS Long Polling (p. 41)</a> .	November 5, 2012