SQS

19 June 2023 15:27

Introduction

Amazon Simple Queue Service (Amazon SQS) offers a secure, durable, and available hosted queue that lets you integrate and decouple distributed software systems and components. Amazon SQS offers common constructs such as dead-letter queues and cost allocation tags. It provides a generic web services API that you can access using any programming language that the AWS SDK supports.

How does SQS work?

Amazon SQS consists of three major components: *producers* (components that send messages to the queue), *the queue* (which stores messages across multiple Amazon SQS servers), and *consumers* (other components that receive messages from the queue).

With SQS, a producer sends a message to a queue for redundant distribution across the Amazon SQS servers. A consumer that's ready to process messages receives the message from the queue. The message that's being processed remains in the queue and it isn't returned to subsequent receive requests until the message visibility timeout lapses. The consumer deletes the message from the queue after processing to prevent the message from being visible again in the queue after the visibility timeout.
With Amazon SQS, multiple consumers can't receive the same message simultaneously. A message can only be received from a queue by one consumer that's ready to process and then delete the message received. A message can be retained in queues for 14 days maximum and 1 minute minimum

- **Amazon SQS Visibility Timeout**: This is the period of time that a message received from a queue by one consumer won't be visible to the other consumer. Visibility timeout for a message ranges between 0 seconds to 12 hours, with the default set at 30 seconds.
- **Dead-Letter Queue**: This is a queue that other source queues can target for messages that fail to process correctly . Whether it's client or server errors, failed messages are moved to a dead-letter queue for further checks, analysis, and reprocessing.
- **Amazon SQS Short and Long Polling**: These are the two ways of message processing in Amazon SQS. Short polling is used by default and it occurs when the waiting time equals 0 (zero). With the short polling, you get an immediate response (even with no message found) because the Receive Message request queries only a subset of the servers (where the queue stores and processes messages) to find available messages to include in the response. However, it changes to long polling when the waiting time for the Receive Message API action is greater than 0 (zero). With long polling, it takes more time to return a response because the ReceiveMessage request queries all of the servers for messages and waits until a message arrives in the queue or the message long poll times out. A long polling frequency

can be set between 1 to 20 seconds.

Important points to remember:

- SQS is pull-based, not push-based.
- Messages are 256 KB in size.
- Messages are kept in a queue from 1 minute to 14 days.
- The default retention period is 4 days.
- It guarantees that your messages will be processed at least once.

Benefits of using Amazon SQS

- **Security** — You control who can send messages to and receive messages from an Amazon SQS queue. Server-side encryption (SSE) lets you transmit sensitive data by protecting the contents of messages in queues using keys managed in AWS Key Management Service (AWS KMS).
- **Durability** — For the safety of your messages, Amazon SQS stores them on multiple servers. Standard queues support at-least-once message delivery, and FIFO queues support exactly-once message processing.
- **Availability** — Amazon SQS uses redundant infrastructure to provide highly-concurrent access to messages and high availability for producing and consuming messages.
- **Scalability** — Amazon SQS can process each buffered request independently, scaling transparently to handle any load increases or spikes without any provisioning instructions.
- **Reliability** — Amazon SQS locks your messages during processing, so that multiple producers can send and multiple consumers can receive messages at the same time.
- **Customization** — Your queues don't have to be exactly alike — for example, you can set a default delay on a queue. You can store the contents of messages larger than 256 KB using Amazon Simple Storage Service (Amazon S3) or Amazon DynamoDB, with Amazon SQS holding a pointer to the Amazon S3 object, or you can split a large message into smaller messages.

Technical Overview of Amazon SQS

From a technical standpoint, Amazon SQS features five basic APIs for developers to get started with. These include: CreateQueue, SendMessage, ReceiveMessage, ChangeMessageVisibility, and DeleteMessage. There are also extra APIs for advanced functionality. With SQS APIs, serverless developers can directly implement an SQS queue with any other AWS service, including a third-party service, allowing them to work in most any programming language of their choice.

Amazon SQS supports message payloads comprising close to 256KB of text in any format. However, you can manage Amazon SQS messages content larger than 256KB (up to 2 GB) using Amazon Simple Storage Service (Amazon S3) or Amazon DynamoDB.

Amazon SQS supports server-side encryption (SSE) for queues to allow the exchange of sensitive data between applications. SQS SSE uses the AES-256 GCM algorithm which is the 256-bit Advanced Encryption Standard to keep sensitive data secure using a unique encryption key. The encryption key is managed within the AWS Key Management Service (KMS). In addition, AWS KMS logs all your encryption key usage to AWS CloudTrail for regulatory and compliance requirements.

Amazon SQS is subscribed to HIPAA Compliance Program to allow developers to build HIPAA-compliant applications. Developers can use SQS to store and transmit messages between healthcare systems (including PHI-messages).

Amazon SQS integrates well with other AWS infrastructure web services, including Redshift, DynamoDB, RDS,

EC2, ECS, Lambda, and S3 to increase the flexibility and scalability of your distributed application.

When to use Amazon SQS

Amazon offers three services to enable communication between apps: SQS, MQ (Message Broker), and SNS (Simple Notifications Service).

Amazon MQ is a managed message broker service for Apache ActiveMQ. Amazon recommends using it for migrating your messaging with existing applications to the cloud.

Amazon SNS is a push notifications service, and using it in combination with SQS is one of the best practices.

Amazon SQS will be a good fit for new applications built in the cloud. You can use it independently but it's always a good idea to compose it with other Amazon services like Amazon EC2, Amazon EC2 Container Service (Amazon ECS), and AWS Lambda. In addition, you can store SQS messaging data in Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB.

In general, message queueing is used to provide application scalability and to decouple the complex back-end operations from the front-end output.

Here is what else you can do with the queues in Amazon SQS:

- enable server-side encryption
- manage queue permissions
- use tags for a queue
- send messages with attributes ,with a timer, etc.

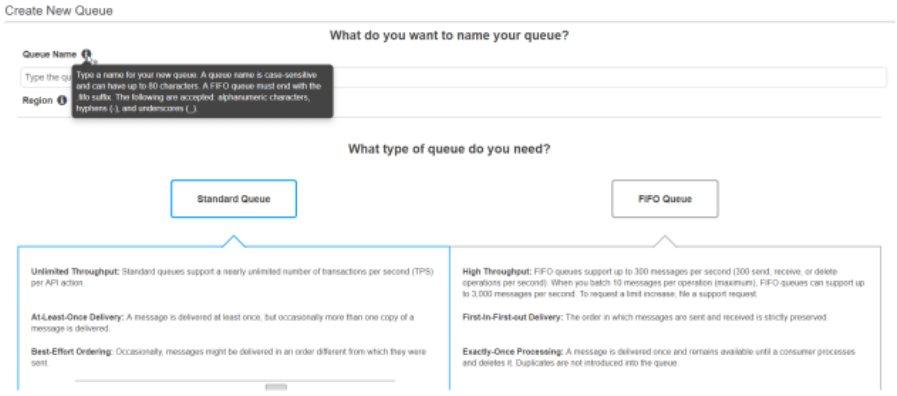
And a couple of words about important Amazon SQS limits:

- You can retain messages in queues for 14 days maximum.
- The maximum size of one message is 256 KB (you can send large messages via Amazon SQS Extended Client Library for Java, which stores larger payloads in Amazon S3).
- A message can contain XML, JSON, and the unformatted text. The limited number of Unicode characters is supported as well.
- The number of messages handled in standard queues is not limited! However, FIFO queues allow up to 3,000 messages per second.

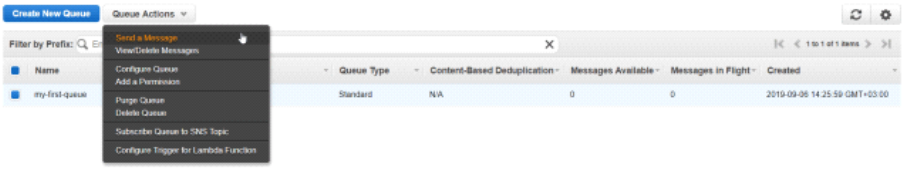
How to configure the Amazon SQS queue

You can access Amazon SQS via AWS Management Console or integrate it via API.

Getting started with the AWS Management Console is easy and straightforward. Just assess SQS service from the Services list on your AWS account homepage and follow the instructions. You will get explanations and hints from Amazon at every step.



Here you will be able to choose the type of the queue, make necessary configurations, and then start adding messages right away. Also, you can add metadata for each message, such as name, type, and value.



Once your message is sent to a queue, it is ready for retrieval and processing by another application. To connect your app with the queue in this Amazon service, you will need to pass the queue URL (which you will find in the queue **Details** tab).

From the Queue Actions menu, you can request a message to view (by clicking Start Polling for messages) and then delete it.

Also, you can subscribe queue to the SNS topic and configure trigger from Lambda function right from the **Queue Actions** menu.

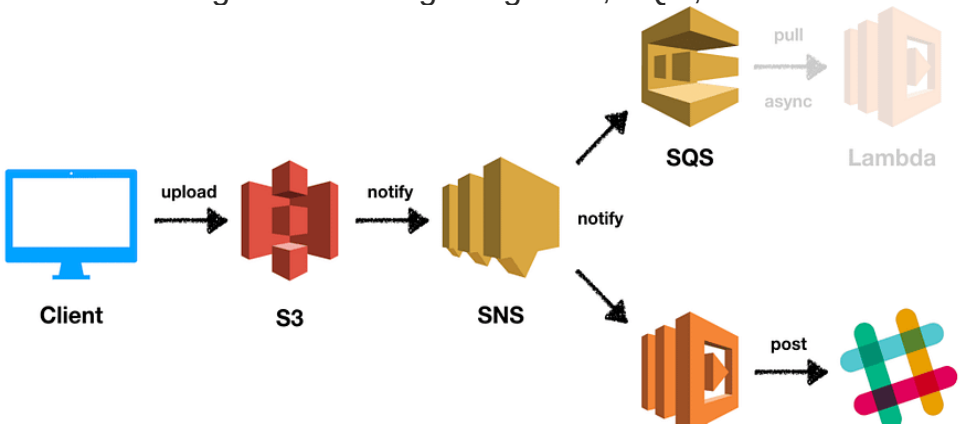
For more details and further steps, refer to this guide:

Getting started with Amazon SQS

This section helps you become more familiar with Amazon SQS by showing you how to manage queues and messages using the…

docs.aws.amazon.com

Event Handling in AWS integrating SNS, SQS, and Lambda



In reactive, message-driven applications it is crucial to decouple producers and consumers of messages. Combining publish/subscribe (pub/sub) and queueing components we are able to build resilient, scalable and fault-tolerant application architectures. AWS offers a variety of components which implement pub/sub or queueing. The goal is to develop an event pipeline which sends a message to a Slack channel whenever someone uploads a picture to an S3 bucket. For demonstration purposes we will also store the events in a queue for asynchronous processing. The architecture involves S3 event notifications, an SNS topic, an SQS queue, and a Lambda function sending a message to the Slack channel. Here is an animation of the final result.

Architecture

Let's look at the high level architecture. When a client uploads an image to the configured S3 bucket, an S3 event notification will fire towards SNS, publishing the event inside the respective topic. There will be two subscribers for that topic: An SQS queue and a Lambda function.

The SQS queue stores the event for asynchronous processing, e.g. thumbnail generation or image classification. The Lambda function parses the event and sends a notification message to a Slack channel. Within the scope of this blog post we are not going to discuss the asynchronous processing part. Due to the decoupling of publishing and subscribing with SNS we are free to add more consumers for the events later.

SNS is a simple pub/sub service which organizes around *topics*. A topic groups together messages of the same

type which might be of interest to a set of subscribers. In case of a new message being published to a topic, SNS will notify all subscribers. You can configure delivery policies including configuration of maximum receive rates and retry delays.
The goal is to send a Slack message on object creation within our S3 bucket. We achieve that by subscribing a Lambda function to the SNS topic. On invocation the Lambda function will parse and inspect the event notification, extract relevant information, and forward it to a preconfigured Slack webhook.
We will also subscribe an SQS queue to the topic, storing the events for asynchronous processing by, e.g., another Lambda function or a long running polling service. The next section explains how to implement the architecture.

Use Cases for message queue

Decoupling

At the start of a project**,** it's extremely difficult to predict what the future needs of the project will be. By introducing a layer between processes, message queues create an implicit, data-based interface that both processes implement. This allows you to extend and modify these processes independently, by simply ensuring they adhere to the same interface requirements.

Redundancy

Sometimes processes fail when processing data. Unless that data is persisted, it's lost forever. Queues mitigate this by persisting data until it has been fully processed. The put-get-delete paradigm, which many message queues use, requires a process to indicate explicitly that it has finished processing a message before removing it from the queue, ensuring your data is kept safe until you're done with it.

Scalability

Because message queues decouple your processes, it's easy to scale up the rate at which messages are added to the queue or processed; simply add another process. No need to change code**;** no need to tweak configurations. Scaling is as simple as adding more power

Elasticity & Spikability

When your application hits the front page of Hacker News, you're going to see unusual levels of traffic. Your application needs to keep functioning with this increased load, but the traffic is an anomaly, not the standard; it's wasteful to have enough resources on standby to handle these spikes. Message queues will allow beleaguered components to struggle through the increased load, instead of getting overloaded with requests and failing completely. Check out our **s**pikability blog post for more information about this.

Resiliency

When part of your architecture fails, it doesn't need to take the entire system down with it. Message queues decouple processes, so if a process that is processing messages from the queue fails, messages can still be added to the queue to be processed when the system recovers. This ability to accept requests that will be retried or processed at a later date is often the difference between an inconvenienced customer and a frustrated customer.

Delivery Guarantees

The redundancy provided by message queues guarantees that a message will be processed eventually, so long as a process is reading the queue. On top of that, IronMQ provides an only-delivered-once guarantee. No matter how many processes pull data from the queue, each message will only be processed a single time. This is possible because retrieving a message “reserves” that message, temporarily removing it from the queue. Unless the client specifically states that it’s finished with that message, the message will be placed back on the queue to be processed after a configurable amount of time.

Ordering Guarantees

In a lot of situations, the order with which data is processed is important. Message queues are inherently ordered, and capable of providing guarantees that data will be processed in a specific order. IronMQ guarantees that messages will be processed using FIFO (first in, first out), so the order in which messages are placed on a queue is the order in which they’ll be retrieved from it.

Asynchronous Communication

A lot of times, you don’t need to process a message immediately. Message queues enable asynchronous processing, which allows you to put a message on the queue without processing it immediately. For long-running API calls, SQL reporting queries, or any other operation that takes more than a second, consider using a queue. Queue up as many messages as you like, then process them at your leisure.