beSharp.
we make IT run.

DECOUPLING SERVICES USING SQS AS A LAMBDA TRIGGER

Amazon Simple Queue Service (SQS) | AWS Lambda | DevOps | Microservices | Serverless

beSharp | 24 January 2020

Design a microservices-based application presents some challenges such as service discovery, standardize internal cross-service communication, services synchronization, and much more.

One of the most common scenarios is the decoupling of two services when the first relies on the other one to asynchronously process some data. We can solve these kinds of problems using a classic producer-consumer approach.

In the Cloud, we can leverage managed services to ease the realization of such architecture.

The reference infrastructure for those situations is to use one or more SQS queues to decouple the producer from the consumer.

Using managed services guarantees several benefits, including:

- No need to check for consumer availability
- Reliable queue metrics
- No need to implement intricate logic to adjust the throughput of the producer service
- Messages can be stored up to days until a consumer is ready to pick them

Before starting our deep dive into the solution, and to discover how to implement it on AWS in details, we will cover the basics of the services we will be using.

What is Amazon SQS?

Amazon Simple Queue Service (SQS) is a fully managed message queuing service that enables you to decouple and scale microservices, distributed systems, and serverless applications.

It’s a fully managed service, so it removes the complexity and overhead associated with managing and operating message-oriented middleware and empowers developers to focus on differentiating work. Using the API you can send, store, and receive messages between software components at any volume, without losing messages or requiring other services to be available.

SQS offers two types of message queues. Standard queues offer maximum throughput, best-effort ordering, and at-least-once delivery. SQS FIFO queues are designed to guarantee that messages are processed exactly once, in the exact order that they are sent.

What is AWS Lambda?

AWS Lambda is a compute service that lets you run code without provisioning or managing servers. AWS Lambda executes your code only when needed and scales automatically, from a few requests per day to thousands per second.

It also runs your code on a high-availability compute infrastructure by design and performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, code monitoring and logging.

This is in exchange for flexibility, which means you cannot log in to compute instances, or customize the operating system on provided runtimes.

Now that the services have been introduced, we can start to explore the solution.

The problem

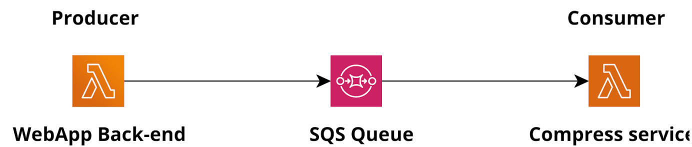
To make the discussion more pragmatic, let’s pretend to be in the following situation:

> *We have a serverless web application that allows users to select documents to download. To optimize transfer time, we need to build a service to create a compressed archive containing the files and save it on S3. The service will also create a signed link to allow the download of the archive.*

We won’t implement any of the features, indeed, we will just use some stub code to demonstrate the solution. Anyway, having a plausible situation will help us understand the mechanism of operation and the potential problems that may arise.

The solution

The following is the typical AWS infrastructure for a producer-consumer pattern using AWS Lambda and SQS.



For this to work, we need to configure an SQS trigger for the consumer, so that the consumer lambda will be triggered when messages are available in the queue.

Using our example:

The producer lambda is the lambda function in the web application backend that needs to send a request to the compression service, while the consumer is another Lambda based microservice that can read a list of files (maybe s3 URIs) and compress them before storing the result on s3. For the sake of the example, let's assume that this service will also generate and store a signed link in a database, for later use.

Here is the basic flow to implement the producer-consumer pattern:

- The Lambda powered back-end service uses the AWS SDK to perform a SendMessage API call to SQS, putting a new message (a compress job specification) in the queue.
- The consumer Lambda is automatically invoked by the Lambda service when a new job is present in the queue
- The consumer will then proceed to fulfill the job requested, consuming the message

How does the SQS trigger work?

Your function will be invoked by the Lambda service, and it will receive the messages as an input parameter.

For standard queues, Lambda uses standard SQS long polling to poll a queue until it becomes active. This will make a call every 20 seconds. When messages are available, Lambda reads up to 5 batches and sends them to your function.

The size of a "batch" can be changed in the trigger settings (1 – 10) and is the number of messages sent to each lambda. If messages are still available, Lambda increases the number of processes that are reading batches by up to 60 more instances per minute. The maximum number of batches that can be processed simultaneously by an event source mapping is 1000.

For FIFO queues, Lambda sends messages to your function in the order that it receives them. When you send a message to a FIFO queue, you specify a message group ID. Amazon SQS ensures that messages in the same group are delivered to Lambda in order. Lambda sorts the messages into groups and sends only one batch at a time for a group. If the function returns an error, all

retries are attempted on the affected messages before Lambda receives additional messages from the same group.

So, basically, if you configure the trigger with a batch size of 1, Lambda will poll the queue and invoke a function each time a new job is available in the queue. Each lambda will receive exactly 1 job.

If you configure a batch size of 5, Lambda will still invoke your function as soon as possible, but up to 5 messages are delivered to your function, at once.

When a lambda receives a message, it has to fulfill the job and return without an error.

If the function does not return an error, the messages sent to the function are considered correctly consumed and are automatically removed from the queue by the Lambda service.

If your function returns any kind of exception (it fails) then the messages are not removed from the queue and after a configurable timeout, they will become available again.

What about failed jobs?

Right now, there is a big problem with our solution. What if a job fails? For example, in case the lambda function is out of disk space? or invalid input URIs are given in the job message?

Well, if the Lambda fails, as it should, when dealing with an impossible or incomplete job, the message will stay unavailable for the visibility timeout, and then, if not correctly consumed, it will become available for a new consumer to pick.

This is a giant infinite loop that will cost you a lot of money!

You may be tempted to just catch any kind of exception inside your lambda code, and just save somewhere that a job has failed, consuming the message and avoiding to keep trying unfulfillable jobs. While this approach will work just fine, there is an even better way to deal with the failing jobs.

Dead letter queues

In each SQS queue, there is a setting to specify an optional dead letter queue.

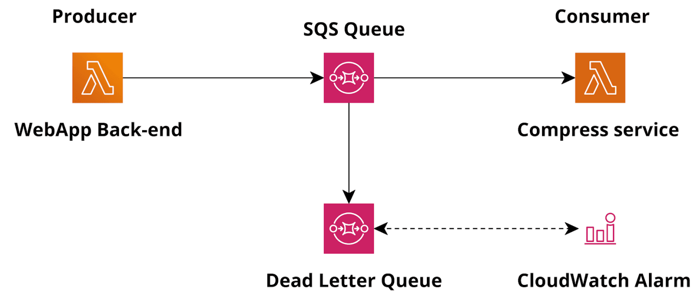
A dead letter queue is a special queue where SQS will automatically put messages that are rejected for a configurable number of times.

So, for example, you can create an additional queue and specify it's ARN as DLQ for the main queue. You can also specify the number of retries "maxReceiveCount" in the main queue so that our system will allow some retries. This is actually useful, because a message may be rejected for a lot of reasons, some of them are not related to the consumer code, and are likely temporary, such as hitting the Lambda concurrency limit threshold or other temporary situations.

Updated solution

So, adding a dead letter queue to our solution will protect us from infinite retries.

Also, we can configure a CloudWatch alarm to warn developers or the operations team when the number of failed jobs is unusually high and needs troubleshooting.



A working example

For your convenience, here a simple CloudFormation template that creates a queue, a dead letter queue, and a lambda function configured to be triggered on new messages.

```
AWSTemplateFormatVersion : 2010-09-09
Resources:

 LambdaExecutionRole:
   Type: AWS::IAM::Role
   Properties:
     AssumeRolePolicyDocument:
       Version: '2012-10-17'
       Statement:
       - Effect: Allow
         Principal:
           Service:
           - lambda.amazonaws.com
         Action:
         - sts:AssumeRole
     Policies:
       - PolicyName: allowLambdaLogs
         PolicyDocument:
           Version: '2012-10-17'
           Statement:
           - Effect: Allow
             Action:
             - logs:*
             Resource: arn:aws:logs:*:*:*
       - PolicyName: allowSqs
         PolicyDocument:
           Version: '2012-10-17'
           Statement:
           - Effect: Allow
             Action:
             - sqs:ReceiveMessage
             - sqs:DeleteMessage
             - sqs:GetQueueAttributes
             - sqs:ChangeMessageVisibility
             Resource: !GetAtt MyQueue.Arn
```

```
LambdaConsumer:
  Type: AWS::Lambda::Function
  Properties:
    Code:
      ZipFile: |
        def lambda_handler(event, context):
            for record in event['Records']:
              print(record['body'])
    Handler: index.lambda_handler
    Role: !GetAtt LambdaExecutionRole.Arn
    Runtime: python3.7
    Timeout: 10
    MemorySize: 128

LambdaFunctionEventSourceMapping:
  Type: AWS::Lambda::EventSourceMapping
  Properties:
    BatchSize: 1
    Enabled: true
    EventSourceArn: !GetAtt MyQueue.Arn
    FunctionName: !GetAtt LambdaConsumer.Arn

MyQueue:
  Type: AWS::SQS::Queue
  Properties:
    DelaySeconds: 0
    VisibilityTimeout: 30
    RedrivePolicy:
      deadLetterTargetArn : !GetAtt DLQ.Arn
      maxReceiveCount : 3
 DLQ:
  Type: AWS::SQS::Queue
  Properties:
    DelaySeconds: 0
    VisibilityTimeout: 180
```

You can deploy and test the template by putting any message into the queue and looking for the lambda execution logs.

```
START RequestId: 45dddad8-49d3-4378-ba7a-2e03217e9c40 Version: $LATEST
Hello from SQS!
END RequestId: 45dddad8-49d3-4378-ba7a-2e03217e9c40
REPORT RequestId: 45dddad8-49d3-4378-ba7a-2e03217e9c40  Duration: 1.56 ms       Billed
Duration: 100 ms        Memory Size: 128 MB     Max Memory Used: 56 MB  Init Duration:
119.17 ms
```

In this article we explored one of the most common patterns to decouple serverless microservices using high available and fully managed services on AWS.

The architecture is very flexible, and so it can be used to solve a plethora of similar problems.

The main advantage of this architecture is that it is really cost-effective, at the same time it’s resilient and allows you to make solid decoupled services.

Cool, isn’t it? 🙂

Feel free to contact us for more!

beSharp

Dal 2011 beSharp guida le aziende italiane sul Cloud. Dalla piccola impresa alla grande multinazionale, dal manifatturiero al terziario avanzato, aiutiamo le realtà più all'avanguardia a realizzare progetti innovativi in campo IT.

Get in touch

beSharp.it
proud2becloud@besharp.it

Copyright © 2011-2021 by beSharp srl - P.IVA IT02415160189