Handling Asynchronous Errors with Dead Letters

In this lesson, you will learn how to handle asynchronous errors with dead letters by modifying the template file.

WE'LL COVER THE FOLLOWING

- Asynchronous request IDs and retries
- Building from a different template

With synchronous calls, errors can be reported directly back to the caller, and the caller can then decide whether it's worth retrying or not. With asynchronous calls, the caller can't do that, because it is not waiting on results.

The good news is that to protect you against intermittent network problems or transient infrastructure issues, Lambda will automatically retry twice. AWS does not publish any official documentation on the duration between retries, but experience shows that the first retry will happen almost immediately after an error. If the function fails to process the event again, the second retry will happen about a minute later. Keep watching the logs while Lambda refuses to copy a file that's not an image, and you'll see the retries.

Asynchronous request IDs and retries

If Lambda retries processing an event, it will send the same request ID as in the original attempt. You can use the request ID to check for any partial results if your Lambda works in multiple stages, and then just partially re-process the event. This is another reason why it's good to use request IDs to generate resource references.

If the second retry fails as well, Lambda gives up on processing the event, concluding that the error is in the code rather than the infrastructure, or that

the infrastructure has not recovered quickly enough. If events don't work after two retries, it often means that you made some bad assumptions about the potential contents, for example, that people will always upload images. It helps to know about such events, especially while you're still developing something, so you can handle those cases better.

You can configure the allowed number of retries and the maximum waiting time for a request to be processed using the `MaximumRetryAttempts` and `MaximumRetryAttempts` parameters of a function. If you use the low-level CloudFormation resource (`AWS::Lambda::Function`), apply these properties directly to the function. For the SAM wrapper (`AWS::Serverless::Function`), these properties are sub-parameters of `EventInvokeConfig`. For more information configuring retry policies with SAM and an example, check out the `EventInvokeConfiguration` documentation page in the SAM Developer guide.

In Chapter 5, I mentioned that CloudWatch automatically tracks errors for Lambda functions. Although you can see statistics about failed retry errors in the CloudWatch console, you won't be able to see the actual contents of the events that caused problems there. However, failed events aren't necessarily lost forever; you can configure Lambda to send them to a *dead letter queue*. You can then set up additional processing or a custom retrial. For example, send the caller an error report, notify support staff about the problem, or even delay the event for a few minutes and then send it back to the same function.

Lambda can use two types of dead-letter queues:

- Amazon Simple Queue Service (SQS) or,
- Amazon Simple Notification Service (SNS).

The two types of queues are importantly different in how they deal with message delivery. SNS is a better choice for instant processing. SQS is a better choice for offline or batch processing.

SQS will store the message for a listener to retrieve it. If nobody is listening when a message arrives, SQS will keep it in the queue. SQS consumers compete for messages, so if there are many listeners, only one will get the chance to process a single message.

SNS will just ignore a message if it has no subscriptions when the message arrives. It will send each message to all active subscriptions. If there are many listeners, everyone gets a copy.

Here, you can set up a dead letter queue that sends you an email if there are problems. For that, you don't really need batch processing, so SNS is a better choice.

If neither SNS or SQS fit your usage needs for dead-letter queues, you can configure an alternative *Lambda destination* for errors. Lambda Destinations were introduced in November 2019, and allow users to forward results or error messages from a Lambda function somewhere else. Configuring a destination for errors is similar to setting up a dead letter queue, but supports more AWS services, including directly invoking another Lambda function or sending an event to AWS EventBridge. For more information on AWS destinations, check out the *Configuring Destinations* section of the *Asynchronous Invocation* page in the AWS Lambda Developer Guide.

> ## Building from a different template
>
> The main template file is without the dead letter queue in the directory for this chapter. The one with the dead letter queue configuration is `template-with-dlq.yaml`. As you are using the source code package from https://runningserverless.com, you will use `-t` to tell SAM to build it from a particular file, i.e., `sam build -t template-with-dlq.yaml`.

You can add a resource of type `AWS::SNS::Topic` to the template resources section (at the same indentation level as `ConvertFileFunction`).

```
NotifyAdmins:
  Type: AWS::SNS::Topic
```

Line 126 to Line 127 of code/ch9/template-with-dlq.yaml

To send dead letters from the `ConvertFileFunction` to the SNS topic, add the following three lines to the function template properties (at the same indentation level as `Policies`, for example).

```
DeadLetterQueue:
  Type: SNS
  TargetArn: !Ref NotifyAdmins
```

Line 123 to Line 125 of code/ch9/template-with-dlq.yaml

The nice thing about SAM and dead letter queues is that SAM will automatically set the permissions for a Lambda function to publish messages to the target queue. If you're using CloudFormation Lambda functions directly without SAM, you'll need to add the permissions yourself.

SNS can send messages to Lambda functions, so you could now implement another function that acts upon being notified about potential problems. SNS can also send messages by email, which is great for troubleshooting failed events during development. You now have a dead letter queue set up, so anyone can manually subscribe to the topic from the SNS Web Console and receive email notifications about any unexpected problems.

With many developers working in a team and people trying lots of different things out, the notifications can get quite frequent. It would be much better if people got notifications only for their own experiments. You can do that by setting up the SNS subscription directly in the template, based on a deployment parameter. Each developer can then set up the subscription automatically with their own email when deploying the stack.

Let's add another parameter for the email. The following lines are added to the `Parameters` section of the template.

```
ContactEmailAddress:
  Type: String
  Description: Email address for operational notifications
  Default: ''
```

Line 25 to Line 28 of code/ch9/template-with-dlq.yaml

In the next lesson, you will learn how to handle condition resources by making email subscription optional.