TECH PAPER

SPINNAKER SUPPORT

Queueable Transaction Finalizers

Check it out on GitHub: https://github.com/AaronCPacheco/DataEncryptionQueueableDemo

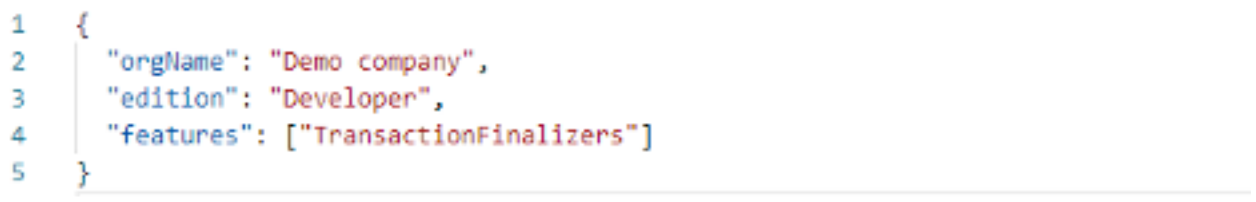
WHY TRANSACTION FINALIZERS ARE USEFUL

The new Transaction Finalizers feature makes it possible to reliably execute some additional code after the asynchronous execution of any class that uses the Queueable interface regardless of whether it succeeds or fails. This could be useful in a lot of ways, the most immediately obvious being for capturing and reporting errors that are otherwise difficult or even impossible to programmatically react to. Beyond just logging errors, for example, you could schedule further processing for records that were successfully processed already or attempt to retry processing for failed records. You could also use it to send an email to help keep track of the job status. Being able to guarantee the execution of code even despite throwing an uncatchable exception is useful in a lot of ways.

In this example, we'll be using it to make sure that processing of records continues even if there is an error as well as using the debug log to give the developer more useful information to address the root cause of the failure.

SETUP

Since this feature is still in pilot stage it's only supported in Scratch Orgs. You'll need to enable Developer Hub in your production org or sign up for a free Developer Edition org where you can enable it. In your project-scratch-def.json file, you'll want to include the "TransactionFinalizers" feature, like so:



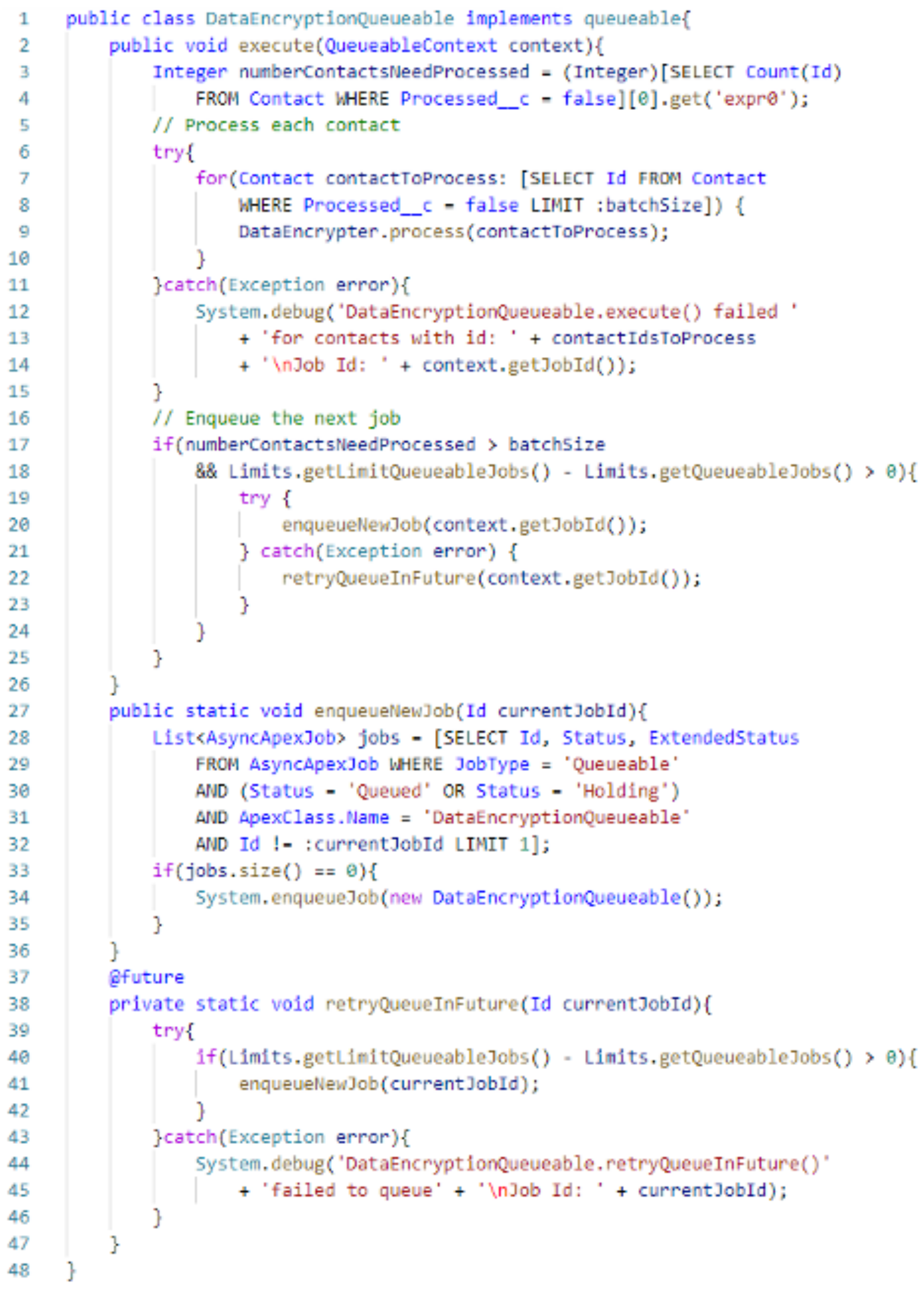
```
{
  "orgName": "Demo company",
  "edition": "Developer",
  "features": ["TransactionFinalizers"]
}
```

Then create a scratch org using that project-scratch-def.json. From the VS Code command pallete run >SFDX: Create Default Scratch Org and use your scratch def file when prompted, or from a command line run sfdx force:org:create -f project-scratch-def.json. Now you're up and running with a Scratch Org with the feature enabled.

Make sure you are using APIv48.0 for your Apex classes when using this feature. On APIv47.0 and below you will get a "type not visible" error when trying to implement the Finalizer interface.

SCENARIO

I have a Queueable that currently uses a try/catch block and a @future method to attempt to recover gracefully from unexpected errors that can happen when daisy chaining another queueable process. It also does a callout, which means it can fail due to an uncatchable timeout exception.



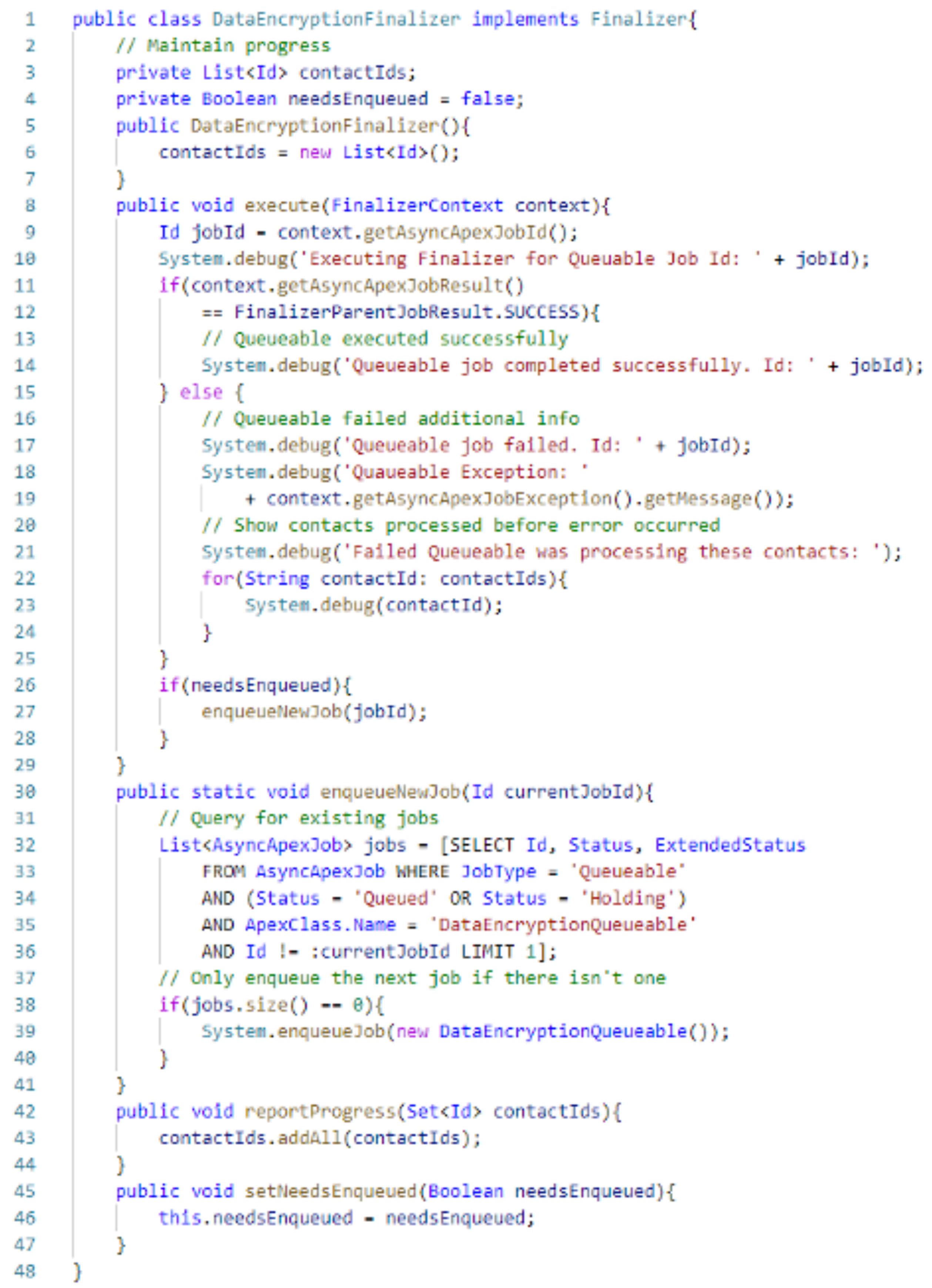
```
public class DataEncryptionQueueable implements queueable{
    public void execute(QueueableContext context){
        Integer numberContactsNeedProcessed = (Integer)[SELECT Count(Id)
            FROM Contact WHERE Processed__c = false][0].get('expr0');
        // Process each contact
        try{
            for(Contact contactToProcess: [SELECT Id FROM Contact
                WHERE Processed__c = false LIMIT :batchSize]) {
                DataEncrypter.process(contactToProcess);
            }
        }catch(Exception error){
            System.debug('DataEncryptionQueueable.execute() failed '
                + 'for contacts with id: ' + contactIdsToProcess
                + '\nJob Id: ' + context.getJobId());
        }
        // Enqueue the next job
        if(numberContactsNeedProcessed > batchSize
            && Limits.getLimitQueueableJobs() - Limits.getQueueableJobs() > 0){
                try {
                    enqueueNewJob(context.getJobId());
                } catch(Exception error) {
                    retryQueueInFuture(context.getJobId());
                }
            }
        }
    }
    public static void enqueueNewJob(Id currentJobId){
        List<AsyncApexJob> jobs = [SELECT Id, Status, ExtendedStatus
            FROM AsyncApexJob WHERE JobType = 'Queueable'
            AND (Status = 'Queued' OR Status = 'Holding')
            AND ApexClass.Name = 'DataEncryptionQueueable'
            AND Id != :currentJobId LIMIT 1];
        if(jobs.size() == 0){
            System.enqueueJob(new DataEncryptionQueueable());
        }
    }
    @future
    private static void retryQueueInFuture(Id currentJobId){
        try{
            if(Limits.getLimitQueueableJobs() - Limits.getQueueableJobs() > 0){
                enqueueNewJob(currentJobId);
            }
        }catch(Exception error){
            System.debug('DataEncryptionQueueable.retryQueueInFuture()'
                + 'failed to queue' + '\nJob Id: ' + currentJobId);
        }
    }
}
```

HOW A FINALIZER IMPROVES THIS DESIGN

Using the Finalizer, we can get rid of the try/catch blocks and the @future method completely and instead handle that all with the finalizer.

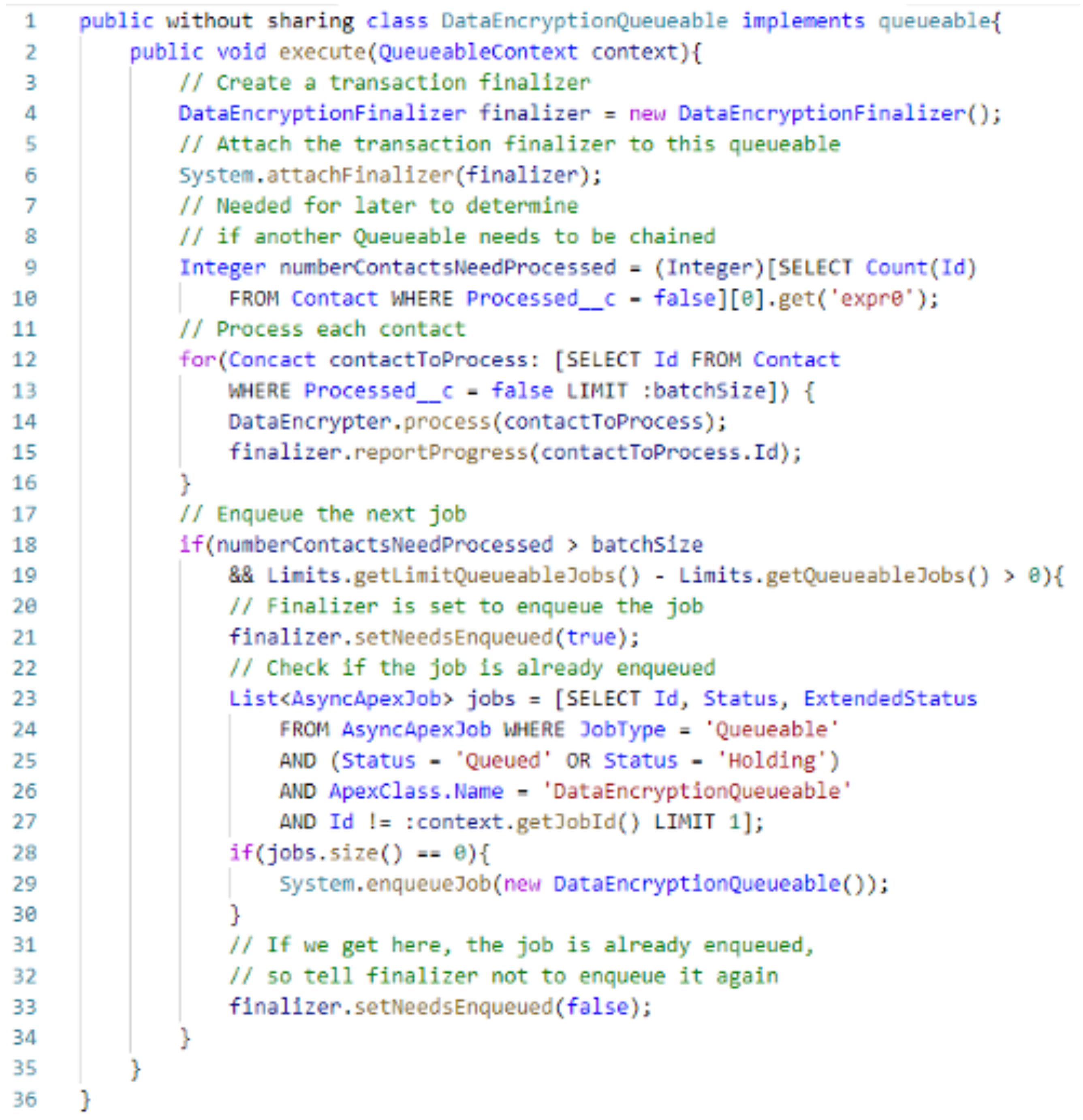
CREATING YOUR FINALIZER

You will want to create a new class for your finalizer. For this demo, we'll name it DataEncryptionFinalizer.



```
public class DataEncryptionFinalizer implements Finalizer{
    // Maintain progress
    private List<Id> contactIds;
    private Boolean needsEnqueued = false;
    public DataEncryptionFinalizer(){
        contactIds = new List<Id>();
    }
    public void execute(FinalizerContext context){
        Id jobId = context.getAsyncApexJobId();
        System.debug('Executing Finalizer for Queuable Job Id: ' + jobId);
        if(context.getAsyncApexJobResult()
            == FinalizerParentJobResult.SUCCESS){
            // Queueable executed successfully
            System.debug('Queueable job completed successfully. Id: ' + jobId);
        } else {
            // Queueable failed additional info
            System.debug('Queueable job failed. Id: ' + jobId);
            System.debug('Quaueable Exception: '
                + context.getAsyncApexJobException().getMessage());
            // Show contacts processed before error occurred
            System.debug('Failed Queueable was processing these contacts: ');
            for(String contactId: contactIds){
                System.debug(contactId);
            }
        }
        if(needsEnqueued){
            enqueueNewJob(jobId);
        }
    }
    public static void enqueueNewJob(Id currentJobId){
        // Query for existing jobs
        List<AsyncApexJob> jobs = [SELECT Id, Status, ExtendedStatus
            FROM AsyncApexJob WHERE JobType = 'Queueable'
            AND (Status = 'Queued' OR Status = 'Holding')
            AND ApexClass.Name = 'DataEncryptionQueueable'
            AND Id != :currentJobId LIMIT 1];
        // Only enqueue the next job if there isn't one
        if(jobs.size() == 0){
            System.enqueueJob(new DataEncryptionQueueable());
        }
    }
    public void reportProgress(Set<Id> contactIds){
        contactIds.addAll(contactIds);
    }
    public void setNeedsEnqueued(Boolean needsEnqueued){
        this.needsEnqueued = needsEnqueued;
    }
}
```

Now you can update the original Queueable class to get rid of all the try/catch statements and the kludgy logic for retrying a failed queue and have it all centralized in the finalizer instead.



```
public without sharing class DataEncryptionQueueable implements queueable{
    public void execute(QueueableContext context){
        // Create a transaction finalizer
        DataEncryptionFinalizer finalizer = new DataEncryptionFinalizer();
        // Attach the transaction finalizer to this queueable
        System.attachFinalizer(finalizer);
        // Needed for later to determine
        // if another Queueable needs to be chained
        Integer numberContactsNeedProcessed = (Integer)[SELECT Count(Id)
            FROM Contact WHERE Processed__c = false][0].get('expr0');
        // Process each contact
        for(Concact contactToProcess: [SELECT Id FROM Contact
            WHERE Processed__c = false LIMIT :batchSize]) {
            DataEncrypter.process(contactToProcess);
            finalizer.reportProgress(contactToProcess.Id);
        }
        // Enqueue the next job
        if(numberContactsNeedProcessed > batchSize
            && Limits.getLimitQueueableJobs() - Limits.getQueueableJobs() > 0){
            // Finalizer is set to enqueue the job
            finalizer.setNeedsEnqueued(true);
            // Check if the job is already enqueued
            List<AsyncApexJob> jobs = [SELECT Id, Status, ExtendedStatus
                FROM AsyncApexJob WHERE JobType = 'Queueable'
                AND (Status = 'Queued' OR Status = 'Holding')
                AND ApexClass.Name = 'DataEncryptionQueueable'
                AND Id != :context.getJobId() LIMIT 1];
            if(jobs.size() == 0){
                System.enqueueJob(new DataEncryptionQueueable());
            }
            // If we get here, the job is already enqueued,
            // so tell finalizer not to enqueue it again
            finalizer.setNeedsEnqueued(false);
        }
    }
}
```

FINAL NOTES

The Finalizer can run after an uncaught error because it runs in its own execution context. This means that that original queueable class instance that instantiated the finalizer no longer exists, so you should avoid trying to call back into the queueable class from the finalizer. Although you could in this example call back into the DataEncryptionQueueable from the DataEncryptionFinalizer.recordProgress() or .setNeedsEnqueued() methods, since they are called from the DataEncryptionFinalizer itself and therefore exist concurrently within its execution context, if you attempt to call back in from the DataEncryptionFinalizer.execute() method you will get a NullPointerException.

CONCLUSION

This feature has a lot of potential, especially if it can be expanded beyond just the Queueable context. Being able to react to uncatchable Apex errors without having to spend an additional 10% of your net Salesforce cost to become an Event Monitoring customer alone would make this worth it. It could be useful for doing post-processing after triggers have finished running or doing surgical database rollbacks when reacting to partial failures. I'm very much looking forward to seeing this feature be developed more by Salesforce.