Data archive and deletion

A quick guide on the current code for monitoring data that can be either deleted or archived, in order to avoid paying storage costs that we needn't pay for.

Overview

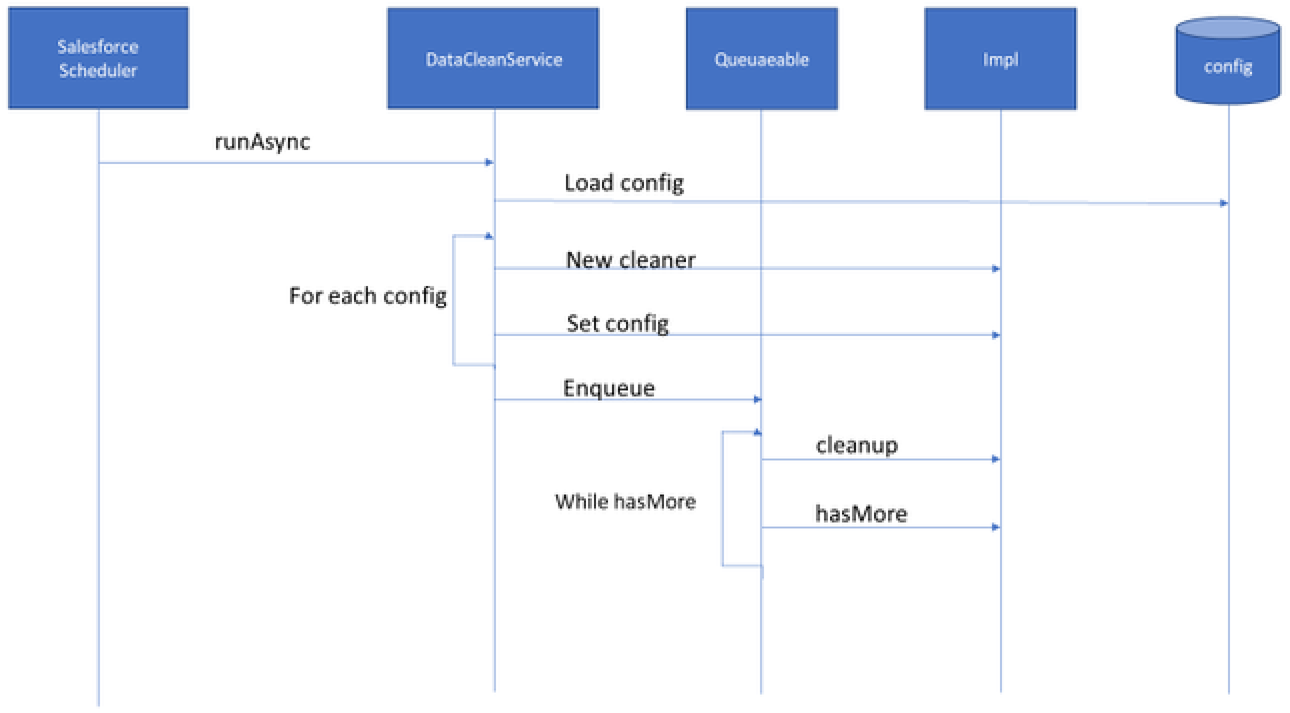
There are a variety of object types that have been identified as being candidates for some form of archival process, although at the time of writing these need more discussion with the business on what rules to apply etc.

The Job table (FFBC), however, is a major issue and has no requirement for actual archival as opposed to simple deletion - it is transitory data to control some async processes.

Therefore, a service to handle data deletion/archiving has been added, for the moment it just handles the ffbc Job deletion. Hopefully the current design will make supporting new object types fairly easy and painless - the hard part will be the requirements around archive for each object type, which is more a business decision.

High Level Design

The process looks somewhat like this.



Detail Design

Nearly all the code lives in DataCleanService, it implements the Schedulable interface and the Queueable is an inner class etc.

There is 1 other class (interface) IDataClean, and anything that wishes to perform some archive/deletion process on an object type should implement that interface. There is currently 1 impl for the FFBC__Job__c object, which again is currently an inner class inside DataCleanService. The implementation does not have to be an inner class of the service, that is just as I have done it for the moment.

The DataCleanService is constructed with a list of IDataClean implementations that need running, and service will be enqueued to run via a queueable. The service itself implements IDataClean so that it can act as a single instance that the queueable can call and handle the list state etc on behalf of the caller. I.e. the complexity of orchestrating many different implementations for different object types is handled by the Service itself implementing the IDataClean interface and forwarding calls to the child implementations.

The current mechanism of setting this up is done via a static method on the service, runAsync, that will load the IDataClean implementations via some custom metadata (Archive_Clean__mdt). The metadata records will each define an implementing class as well as some config for that implementation. For now the config includes:

- Active__c
  - a way to switch off the specific data cleaner from running if needed.

- RetentonPeriod__c
  - the time, in days, for which data should be retained in the primary form. After this date it can be processed. NB the reference date is for each implementation to choose, the existing Job processor uses last modified date, but it might be contract end dates or whatever makes sense for the object type.
- Chunk_Size__c
  - as ever on Salesforce, processing records in bulk needs to be mindful of governors whilst also handling as much as possible in a single transaction. This setting allows each implementation to have its own chunk size set, which determines how many records it will try and process each time it is called.

The handling of the config is done via an abstract base class (inner class of the service) that any implementation should extend (rather than implement the interface). Note, this is the intended way of running but another caller who wishes to exert more control can create IDataClean implementations without using the config or abstract base class etc. The config and Abstract class is purely for a dynamic way of setting the service up, not the actual processing it does once setup.

Once the Queueable is running it will simple call the cleanup method, and if there is more work after that requeue itself. Once there is no more work it stops.

A queueable was chosen as the nature of the work would mean a bit more effort to work out how many times to run batch, plus the work could take a while and batch slots are a valuable resource, so avoiding a batch is useful.

The nature of how this is ran means it is serial, and whilst it is likely that the job table can be processed with concurrent queueables it is highly unlikely other object types could be handled in anything other than a serial fashion. There is no real speed requirement here, so the effort of handling concurrency for what may be the exception is not worth the cost.

Extending to other object type

Assuming nothing overly complex is wanted then other object type can be implemented fairly easily.

Create a class e.g. MyDataCleaner that extends DataCleanService.AbstractCleaner, and implement the cleanUp and hasMore methods. This can call the methods on the base class to find any config values (getChunkSize() and getLiveRetentionPeriod()). The specifics of the cleanup and hasmore methods are for the implementer to determine, but in the simple case just query for records over the retention period and delete them, then query to see if there is at least 1 record still over the retention period.

Create a Archive_Clean__mdt record, that defines the class name and the other settings you want to use.

Deploy the class and metadata. The next time the Service runs it will automatically pick up the new implementation and run it.

Extending to Archiving

As of yet there is no explicit archive logic or config. It is not intended that there will be a need for specific archive logic in the service, nor on the Interface. The cleanup method should do what ever it need to, which may well be different for each object type.

I expect that there may be a need for a new config field - 'ArchivePeriod__c'. That should allow each implementation to work out which records should be ignored (retentionPeriod__c) and then following that which records should be archived, and then following that which records can be deleted (archivePeriod__c).

The only code that will need changing is the initial query for the config to retrieve the new field, and a new getter on the Abstract class. Apart from that the the service does not care what happens and the existing implementation does not bother with archive. So simply adding the new field and the tweaks to expose it are all that will be needed.

The complexity will be in what happens when something is archived, and as that will probably be type specific it is the job of each IDataClean implementation to handle - if needed they can call some other new Archiving code for any generic logic without touching the existing service code.