APTrust Refactor

August 19, 2015

This document describes refactoring work for APTrust's bag processing and related services, including what needs to be done, and why.

Terms

- Fluctus (or "the Rails app") is APTrust's partner-facing web application. It allows partners to see everything they've deposited into APTrust, as well as the state of any pending requests for ingest, bag restoration and bag deletion.
- Ansible is our configuration management tool. We use it to automate server provisioning and configuration, and to do some automated deployment.
- The "Go services" refer to all of APTrust's bag processing services. These include ingest, bag restoration, bag deletion, and periodic fixity checks. These services are written in the Go programming language, hence "Go services."

What

Substantially refactor the existing APTrust code base. This includes code for ingest, restoration and deletion, as well as partner tools and the S3 and Fluctus clients. This current round of refactoring will affect our DPN processing code as well.

Why

The current code satisfies all functional requirements, but in many places the code unclear and hard to follow. In addition, almost all services rely on the NSQ queue service, which can be removed entirely, simplifying not only our APTrust services but also our server setup, configuration and maintenance. The refactor will also allow us to remove the APTrust volume manager, which manages requests for disk space on AWS's expensive and limited EBS volumes, and switch to the simpler, cheaper and more flexible Elastic File System. (AWS's Elastic File System was not available when APTrust code was first written.) The overall goal of the refactor is to make the code more maintainable, reliable and extensible.

Specific Goals

- Simplify and clarify existing code
- Move untestable code into testable units, and write tests for those units. (Currently, about 10-15% of APTrust code is not covered by tests.)

- Get rid of NSQ and use the Fluctus application to track the status of ingest, delete, restore and ongoing fixity jobs. Currently, we're tracking job status in both NSQ and Fluctus, and spending a lot of time making sure the two are in sync. See the section below on Using Fluctus as a Queue.
- Replace expensive and limited EBS volumes with cheaper, more flexible Elastic File System. (Currently, EBS volumes are a major expense, and we have to pay for the entire volume even when it is idle or minimally used. EBS volumes have limited space, which often causes us to throttle the amount of data we ingest. Elastic File System does not have this limitation.)
- Rewrite the Fluctus client, splitting it into a general REST client and a higher-level client that includes some logic. The current client includes too much custom logic to be extensible or useful for future features.
- Create a scheme for configuration management. The current configuration system supports loading configuration settings for a development environment, test environment, demo and production. But it evolved over many months and is currently a messy patchwork of code and configuration files. Configuration code should be limited to a single Go package, and each environment should require a single, easy-to-edit config file.

External Work

The refactor will require the following work outside of the Go code.

- We will use **Ansible** vault for sensitive configuration settings. We are already using Ansible for system configuration, updating and deployment. We currently store a number of sensitive configuration settings outside of all of our Go, Ruby and Ansible source code. These sensitive settings include database passwords, secret keys, etc. We don't put them in source control because our source repositories are public. Currently, a system administrator must manually copy these sensitive settings to all applicable servers. Manual copying can lead to errors and omissions. Storing essential configuration information outside of source control risks the loss of importation information. Ansible vault solves both of these problems letting us store sensitive configuration information in encrypted format inside our source control repository. Ansible decrypts the settings when necessary, such as when we deploy code changes and provision new servers.
- We will configure **Ansible** (or something similar, if it makes more sense) to handle automated deployments to our demo and production environments. Currently, deployment of the Go services is partially automated. Deployment of Fluctus is entirely manual, and should be automated.
- We'll need to update the Processed Item model in our Fluctus **Rails** code to store some additional information about the state of ingest, restore, delete, and fixity check operations that our Go services are performing.

- We'll need to update some of the Fluctus/**Rails** API endpoints that the Go services use. A few of these endpoints include unnecessary logic, and some Processed Item endpoints will need to be updated to work with changes to the Processed Item model.
- The Fluctus/**Rails** app will have some new features that allow the APTrust technical staff to view the state of items currently or recently in the process of ingest, restore, delete or fixity check. See Using Fluctus as a Queue below.

Using Fluctus as a Queue

Currently, APTrust tracks the status of all ingest, delete, restore and fixity check tasks in Fluctus, our partner-facing Rails application; and in NSQ, which our ingest/delete/restore/fixity services use to track outstanding work.

Fluctus contains a subset of the processing data in NSQ, allowing APTrust depositors to see the status of their ingest, delete and restore requests. NSQ contains that information plus some very detailed information about the state of each request, such as all the names, checksums, sizes and types of all the files in a bag, whether those files have been sent to long-term storage yet, etc.

The Go services continually read from and write to both Fluctus and NSQ, and there's a good deal of code to ensure that the two are in sync, and to resolve differences when they are in conflict. This creates quite a bit of complexity.

If we move the Go services state information from NSQ to Fluctus, we can get rid of NSQ entirely. That relieves us of the complexities of having to sync NSQ and Fluctus, and it relieves the system administrator of the burden of having to install, configure and maintain NSQ on a number of servers.

With a little Rails work, this move will also allow us to gain deeper insight into the state of individual items in our ingest, restore, delete and fixity check queues.

Currently, NSQ provides aggregate data about how many items are in a queue, how many have been successfully processed, and how many have failed. But it provides no insight into the state of individual items. When an item fails ingest or restoration, we go through the following process to uncover what happened.

1. Note that the number of failed items in NSQ has increased.
2. Search for recently failed items in Fluctus to see which bag failed.
3. Go to the server logs to find log messages about the failed item, and to find the dump of all the processing state information for that item.

If we remove NSQ and store all processing state information in Fluctus, we'll have a single, searchable browser-based source that shows the state of *all* tasks. This vastly simplifies the

process of debugging previously unencountered issues and allows us to respond to depositor inquiries in minutes instead of hours.

Typically, best practices say not to use your Rails application (or your database) as a work queue. And typically, that's because the work queue is managing a high volume of requests, and you don't want to overload your Rails app (or your database) with those requests. For example, NSQ was made to handle over 100,000 requests per minute.
The types of requests APTrust is handling are inherently long-running. A typical ingest request can take anywhere from 10 seconds to 12 hours to complete, depending on the size of the bag and the number of files it contains. The average ingest may take several minutes, and in this time, the Go services may make a dozen requests to the queue.

This may average out to two requests per minute per ingested bag. At any given time, we may be ingesting 2-20 bags. That would result in 4-40 requests per minute to the queue. This is nowhere near the 100,000 requests per minute that NSQ was built to handle. Our Rails app and the underlying database can easily handle this volume.

In addition, the state data we store in our queue must be persistent. That is, it must survive a server reboot. NSQ has so far shown that its persistence feature works for relatively small amounts of data (a few megabytes), but we have not yet tested its system for rotating files when the size of the data grows. Fluctus' database backend was specifically designed for data persistence, and is well proven.

For all of these reasons, eliminating NSQ and using Fluctus to track both the status and state of bags as they move through the ingest/restore/delete/fixity check processes.