BagIt Fixer-Upper

Scaling BagIt Tools to Manage the Ingest of Petabytes of Digitization Work

Nick Krabbenhoeft
New York Public Library
34-11 Thompson Street
Long Island City, New York 11101
nickkrabbenhoeft@nypl.org

ABSTRACT

The New York Public Library has created over 1.5 PB of files from digitizing over 50,000 audio and video items for the long-term preservation of their content. This paper details the Library's usage of the BagIt File Packaging Format during Quality Assurance and Audit Submissions functions as defined by OAIS. It also discusses extensions of the bagit-python library in order repair bags that do not pass those functions.

Working with thousands of terabytes stored in hundreds of thousands of bags requires that our approaches to ingest scale appropriately. Common changes to bags such as the accidental creation of system files in bags or purposeful edits of metadata files will invalidate the entire bag. Noting and responding to these errors is critical for improving workflows, but manual response is impossible. Using the bagit-python library, NYPL has created tools to selectively clean system files from bag directories and manifests, update or add checksums, and create event logs of repairs.

KEYWORDS

Quality Assurance, Audit Submissions, Scalability, Fixity, BagIt, Python, Digitization

ACM Reference format:
Nick Krabbenhoeft. 2017. BagIt Fixer-Upper. In *Proceedings of iPres 2017, Kyoto, Japan, September (iPres17)*, 5 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

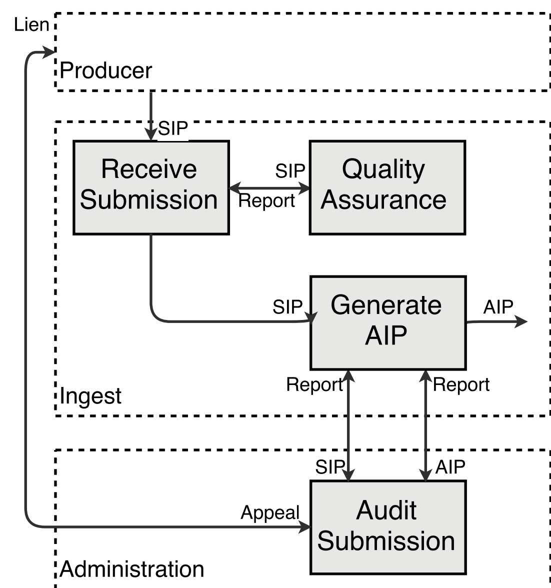

Figure 1: Diagram of the subset of the OAIS functions discussed in this paper. Dotted boxes denote functional entities; grey boxes denote functions, and text next to arrows denotes information provided by one function to another.

1 INTRODUCTION

Before an archive commits to the long-term preservation of a submission, it must verify that it has received expected information. The OAIS Reference Model defines two functions for checking content during ingest: Quality Assurance and Audit Submissions.[3]

Quality Assurance is performed after a producer has transferred a SIP to the archive's[1] Receive Submission function. It is used to "identify any file transfer or media read/write errors." After passing Quality Assurance, the Receive Submission function transfers the SIP to the Generate AIP function. From there, the SIP and/or the AIP can be submitted to the Audit Submissions function in order to "verify that submissions (SIP or AIP) meet the specifications of the Submission Agreement." If a SIP or AIP fails the audit, either the producer can appeal the audit or the archive can request a new SIP. Restated, Quality Assurance checks that the structure of the SIP was not damaged in transit, and Audit Submissions checks that the intellectual content of the SIP and/or AIP matches expectations. See Figure 1 for a visualization of these relationships.

It is the archive's responsibility to define how Quality Assurance and Audit Submission functions are performed. The text of the OAIS includes suggesions for methods such as using system logs and/or hash functions for quality assurance (QA) and review committees for audits, but these are non-exclusive. In implementing these functions at NYPL, two questions have shaped implementation decisions.

(1) How well do the QA and auditing methods scale?
(2) How can information generated by QA and auditing methods improve workflows?

[1] This paper follows the OAIS text's practice of referring to the organization preserving information as an archive.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
iPres17, Kyoto, Japan
© 2017 Copyright held by the owner/author(s). 978-x-xxxx-xxxx-x/YY/MM...$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

2 CASE STUDY: NEW YORK PUBLIC LIBRARY

The New York Public Library (NYPL) is currently digitizing over 200,000 audio, video, and film objects for long-term preservation. The items have a high risk of loss due to their deteriorating media, obsolescing playback equipment, and dwindling communities of playback engineers. NYPL's goal is to transfer all of the items to more stable media, primarily digital, within 10 years.

The current rate of digitization produces terabytes of material from multiple vendors every week. In order to confirm the successful transfer of SIPs, NYPL adopted the BagIt File Packaging Format created and maintained by John Kunze, Justin Littman, Liz Madden, Ed Summers, Andy Boyko and Brian Vargas. The BagIt format is a POSIX-style file and directory structure that allows for validating three types of fixity information.

Hashes comparing stored cryptographic hashes against ones generated from payload files

0xum comparing a stored count and total size of payload files against newly generated numbers

Completeness comparing a stored list of payload filepaths against those currently in the payload

The bag consists of data directory that contains the payload and several metadata files that record the above fixity information and other metadata. For NYPL, the payload comprises the preservation master files, metadata files, and service copies from a single digitized object.

2.1 Types of Fixity

The gold standard in fixity information of digital files is the cryptographic hash. Because generating a hash for an object after a transfer and comparing it to a previously known value can detect most bit-level losses, it makes sense as the mechanism for checking the technical accuracy of a file transfer. Recommendations for hash comparisons are common in digital preservation documents from the OAIS Standard to the US Government's Federal Agencies Digitization Guidelines Initiative[4].

However, hash calculation is I/O-bound. Every bit of a object must be read to calculate a hash.

There are other types of fixity information that are much faster to generate. For example, if a group of digital objects have not changed, then metadata describing the group like the total number of files, the total size of the files, and the list of filepaths, should remain fixed as well.

Unfortunately, these properties are not as sensitive to change as a cryptographic hash. A bit flip would not effect any of these values. On the other hand, these coarse measures of fixity are far more responsive to coarse modifications such as deleted files, accidentally renamed files, and renamed files.

While these other types of fixity information can not replace hash validation in a Quality Assurance function, using them prior to hashing can immensely speed up the discovery of common errors.

2.2 Benefits of BagIt for Quality Assurance

To make any QA method scalable, the required fixity information must be stored in a machine-readable format. For hashes, there are two widely used formats:

sidecar files storing an md5 hash for `example.mov` in `example.mov.md5`

manifest files listing the relative path and hash for a directory of files in a single CSV

These common formats, and the many tools that can read them, make hash validation relatively easy to adopt as a quality assurance method, but for other types of fixity information, there are very few common formats.

A single organization may specify a standard way to store other types of fixity, and even create software to generate and check this method. But, without community adoption, these methods are fragile at best and lost opportunity costs at worst.

The BagIt File Packaging Format defines a standard format for storing the total size and total number of files in a payload, known as the 0xum (total-file-size.total-file-count). It also places strict requirements on the hash manifest. Th path for every file in the payload must be represented in a hash manifest. As a result, it is possible to check that a bag is complete, that its payload contains only expected files, no more no less.

In terms of commonality, the BagIt format is both an IETF RFC and an open source specification on GitHub. Tools to create and validate bags have been implemented in multiple languages, including Java[10], Python[12], and Ruby[8], making these additional fixity checks widely available. Additionally, most implementations are open-source, allowing users to further customize and adapt the tools as needed. At NYPL, the Python, Ruby, and Java implementations have been adopted by different processes in the Library.

2.3 Disadvantages of BagIt for Quality Assurance

Before discussing BagIt's usage with the NYPL workflow, it is important to highlight a few difficulties that can result from adopting BagIt.

First, the strictness of completeness. The manifests of a valid bag must list every file in the payload directory. This requirement conflicts with the behavior of operating systems that zealously create hidden system files to store usability data such as preferred file sorting order, file thumbnails, and indexing information. Browsing a payload directory after bagging often causes these files to appear, which renders the manifests incomplete and the bag invalid. Even if the system files existed at the time of bagging, operating systems will silently modify them with new usability information, which renders the hashes inaccurate and the bag invalid.

Second, multiple implementations of a standard can result in multiple interpretations. Ideally, a BagIt tool should create a bag that can be validated by other BagIt tools, but edge cases can make this difficult. For example, Mac systems can create a file named `Icon\r`. When written into a manifest, the `\r` acts as a carriage return, making it impossible for other BagIt tools to parse the manifest correctly. Following the requirement in the BagIt RFC to write filepaths with percent-encoding avoids this problem, but users should be aware that malformed bags can cause validation problems.[2]

[2] The development of a conformance suite by Library of Congress staff has made it much easier to test an implementation for common errors. [2]

Using BagIt successfully requires adapting workflows to these challenges.

3 NYPL WORKFLOW

NYPL's ingest workflow can be divided into roughly 4 stages.

(1) Bags are received on hard drives from digitization labs and validated.

If a bag is not valid, repairs are made when possible or a new transfer is requested.

(2) Valid bags are transferred to quality control and audited against the library's published specifications for signal quality, file format, metadata values, and file organization. [5] [7].

If a bag does not pass quality control, repairs are made when possible or redigitization is requested.

(3) QC'd Bags are transferred to a staging area on network storage and validated after transfer.

(4) During ingest to the repository, bags are validated again.

Like most real-world preservation workflows, the NYPL ingest process does not mirror the simplicity of the OAIS model (Figure 1). For example, there is no single Quality Assurance function. Instead, quality assurance is performed each time the bag moves to a different storage medium.[3]

This workflow also makes allowances for alterations to the SIP. While the Generate AIP function can include file format conversions and metadata gathering or conversion, it does not explicitly include provisions for altering the contents of SIP. However, despite efforts to produce comprehensive specifications, bags can fail the quality control processes.

In the standard, failure during the Audit Submissions function can result in either the producer negotiating for a pass or the archive requesting a new submission. No response to failure is discussed for the Quality Assurance function.

In practice, repair is the most preferable action, when appropriate. Whether the specifications were not exact enough or a the cause of failure was not severe enough, the cost of repair can be much lower than resubmission.

4 COMMON QA AND AUDIT FAILURES

In addition to signaling problems with a specific bag, QA and audit failures often highlight systemic workflow problems that require a combination of communicating with producers and refining specifications. The following lists the most common causes of QA and audit failures, why they are flagged as failures, and how they are remediated:

[3]The Inner OAIS-Outer OAIS model presents an interesting approach to the challenge of mapping real-world work to OAIS ideals. [13] It is possible to model the above workflow as a three linked OAIS archives, a receiving archive, which accepts and manages hard drives on behalf of its designated community, the quality control archive, which accepts and manages bags on behalf of its designated community, the repository archive, which accepts and manages digitized objects on behalf of its designated community, the access unit. Each "archive" maintains their own functional entities for Ingest (including a Quality Assurance function), Data Management, Storage, Dissemination, Administration, and Preservation Planning, and their work is coordinated by an Outer Administration and Preservation Planning entities. While full documentation of organizational units at this level of detail may be impractical, the scalability of the OAIS Reference Model to describe large and small "archives" is an interesting theoretical question.

4.1 Invalid Metadata

Submitted metadata is validated against a schema to ensure correct usage of controlled vocabulary and completion of required fields. A manual spot check of a bag's metadata revealed that a vendor had accidentally omitted a set of required fields while updating their workflow. Further investigation showed that this omission was not caught because a typo in an early version of the schema rendered those fields optional, and that several hundred bags shared the same problem.

The missing metadata included fields required for repository ingest and used to create descriptive records. In order to prevent future metadata omissions, the metadata schemas received a thorough review during which a similar mistake was found and fixed. To alleviate the hold on repository ingest and description, tools were developed to repair metadata for all affected bags.

4.2 System Files

As discussed previously, system files can render a bag invalid. The system files themselves do not pose a preservation risk, but their existence indicates that when the bag was QC'd, it was likely mounted with read-write privileges. This exposes the bag to the risk of corruption or deletion through human error. To prevent this possibility, the QC workstation is checked and configured to mount all drives as read-only.

4.3 File Name Changes

Part of the SIP packaging specification includes storing all preservation master files in a directory named `PreservationMasters`, even if the folder only includes a single file. Digitization engineers have bagged a project and then removed the s from the directory name, `PreservationMaster`, when they realized it only contained one file. This change causes a bag completeness failure, since the file path listed in the bag manifest no longer exists.

Since the folder name is part of the packaging specifications, it is also built into the repository ingest process. Bags without this directory cannot be ingested to the repository. Addressing this problem required working with engineers to make sure the specifications were written clearly.

4.4 Missing Checksums

The SIP specifications require that metadata files are packaged alongside the preservation master. An early workflow would bag the digitized media and then add the appropriate metadata files to the bag. This change causes a bag completeness error, since the metadata files are not listed in the manifest.

While hashes are most critical to ensure the fixity of preservation master files, we are also interested in ensuring the fixity of metadata files because they contain an audit trail of how the file was created. Again, the solution was to work with the engineers to make sure the specifications were written clearly.

4.5 Missing Files

The most worrying error is a bag completeness validation failing because of missing files. This is the realization of the accidental deletion specter discussed in regards to system files. An audit of our staging environment once revealed a bag that was missing most

of its files. Fortunately, the original item could re-digitized, but the discovery triggered an account audit of our staging server and review of our tools and procedures in order to reduce the chance of complete loss again.

5 AUTOMATING BAG REPAIR

Except for the last example, each of the above failures can be repaired. Bags remain eligible for ingest as long as it can be shown that a failure did not impact the fixity of the preservation master or mezzanine. However, the interlocking nature of 0xum, completeness, and hash fixity in bags means that any change to a bag's payload likely requires edits to multiple parts of the bag.

For example:

- adding metadata files to a bag requires adding the path and hash for each file to the bag manifest and updating the 0xum with the new number of files and total size
- removing system files from a bag requires searching for all common system file names in a bag, comparing that list to the paths in the manifest, and deleting only those files not listed in the manifest
- updating all metadata files with missing technical fields requires regenerating hashes for only the metadata files, recording them to the manifests, and then updating the 0xum with the new payload size.

These are relatively simple but very tedious procedures and prone to human error.

In response, NYPL extended the bagit-python library. The tool `update_bag.py` automates the repair of bags. [6] Using the methods of the Bag class within bagit-python, `update_bag.py` performs set operations to identify missing or unwanted files, add or update fixity information in the bag manifest and 0xum, and validate the bag.

Performing these repairs at scale also requires automating the creation of an event log to serve as an audit trails. At this time, every update performed by `update_bag.py` creates a PREMIS event record that is saved as a JSON file within the bag. Future repairs with `update_bag.py` are appended to the JSON file. For simpler events like an 0xum update, both the old and new 0xum are recorded in the PREMIS event. For more complex events like rewriting the hash manifests, copies of the original manifests are kept. The tool is available as part of the ami-tools package developed by NYPL.

6 DISCUSSION

Returning to the issue of repair and OAIS, this exploration of Quality Assurance and Audit Submissions functions at NYPL has reiterated how the density of OAIS functions and their relationships remains a hurdle to working with the Reference Model. The descriptions of the functions in the Model are carefully constructed to be neither overly proscriptive nor prescriptive. This leaves the responsibility of creating a shared understanding of allowed actions to the OAIS user community.

But the shared understanding remains elusive. What is described here as a metadata repair as part of the Generate AIP function in the Ingest functional entity, might be described as a preconditioning action that took place prior to ingest. [9] Or according to a proposal to create a Pre-Ingest functional entity it might be described as a pre-submission action on a Pre-Submission Information Package. [11] It remains unclear if the Reference Model is incapable of filling community needs for a common language to describe workflows or if the community is unwilling to engage with the Reference Model as a common language.

More prosaicly, the BagIt format is an interesting match with the Reference Model. Its form is at once reminiscent of the archival box and the OAIS Information Package. Although `update_bag.py` contains logging features born of quality assurance needs, not long-term storage needs, others in the BagIt community have proposed more rigorous methods to update bags over time. For example, the Restful Bag Server specification proposes the use of version control methods like Git and Mercurial to store a fuller history of changes to the bag. [1] Enriching the container with its own PDI, the bag could mature into a common format of AIP. Already it containerizes material for ingest into the Digital Preservation Network (DPN) [?] and APTrust [?] and can be both input and output for Archivematica.

At the same time, it would be best if BagIt is one of multiple common AIP formats. For all of its benefits, BagIt is still based on a metaphor of managing the digital world like the physical world. When bags are ingested into DPN's and APTrust's cloud storage, they are turned into a tarball so that the entire bag can be treated by a single object by the object-based storage systems employed by S3, Azure, and other storage providers. Regaining access to one file in the bag requires downloading the entire bag and unzipping it.

7 CONCLUSION

BagIt is a mature technology that lends itself well to Quality Assurance functions in OAIS workflows. At NYPL, usage of the BagIt Format is a key component in the audio, video, and film digitization workflow. It allows for consistent, scalable fixity checking. It is also a finicky format that becomes invalid as a result of a litany of small changes or inconsistencies. In the field of digital preservation, this is feature not a bug. Even in the Generating AIP function where changes to the SIP are allowed.

The sensitivity to fixity problems forces the archive to consider how its infrastructure and workflows interact with the material that they manage. Lack of robustness magnifies irregularities in workflows that can effect fixity of submitted packages, and the accessibility of its open-source implementations allow for extension and customizability of these tools to fit very specific workflow needs.

8 REFERENCES

REFERENCES

[1] Chris Adams. 2014. Restful Bag Server. https://github.com/acdha/restful-bag-server. (2014).
[2] Chris Adams and John Scancella. 2016. BagIt Conformance Suite. https://github.com/LibraryOfCongress/bagit-conformance-suite. (2016).
[3] CCSDS. 2012. *Reference Model for an Open Archival Information System.* ISO ISO 14721:2012. CCSDS.
[4] FADGI Audio-Moving Image Working Group. 2016. *Digitizing Motion Picture Film.* Technical Report. Federal Agencies Digitization Guidelines Initiative. http://www.digitizationguidelines.gov/guidelines/FilmScan_PWS-SOW_20160418.pdf
[5] Rebecca Holte, Nick Krabbenhoeft, Jonah Volk, Genevieve Havemeyer-King, and Ben. Turkus. 2017. AMI Specifications. https://github.com/nypl/ami-specifications. (2017).
[6] Nick Krabbenhoeft. 2017. ami-tools. https://github.com/NYPL/ami-tools. (2017).

[7] Nick Krabbenhoeft, Jonah Volk, and Genevieve Havemeyer-King. 2016. AMI Metadata. https://github.com/nypl/ami-metadata. (2016).

[8] Jamie Little and others. 2017. bagit. https://github.com/tipr/bagit. (2017).

[9] Peter McKinney. 2012. *DIGITAL CONTENT PRECONDITIONING POLICY*. Policy. Joint Operations Group (National Library of New Zealand, Archives New Zealand, Internal Affairs). 19–24 pages.

[10] John Scancella and others. 2017. bagit-java. https://github.com/LibraryOfCongress/bagit-java. (2017).

[11] Barbara Sierman and Shira Peltzman. 2015. Pre-ingest. http://wiki.dpconline.org/index.php?title=Pre-ingest. (2015).

[12] Ed Summers, Chris Adams, and others. 2017. bagit-python. https://github.com/LibraryOfCongress/bagit-python. (2017).

[13] Eld Zeirau and Nancy McGovern. 2014. Supporting Analysis and Audit of Collaborative OAISâĂŹs by use of an Outer OAIS âĂŞ Inner OAIS (OO-IO) Model. In *Proceedings of the 11th International Conference on Digital Preservation (State Library of Victoria, Melbourne, Austrailia, October 2014)*. Melbourne, Austrailia, 209–218.