Oxford Common File Layout

Discussion Paper

Copied from prior version of this document. Existing comments have not been migrated.

2018-04-02: This document has been migrated to github. See https://ocfl.github.io/spec/ for the current version and create issues in https://github.com/OCFL/spec/issues, use cases in https://github.com/OCFL/Use-Cases

Andrew Hankinson
Bodleian Libraries, University of Oxford

The Oxford Common Filesystem Layout (OCFL)[1] is an attempt to define common practices for file-based storage systems in digital repositories. The goals of the OCFL initiative are to provide:

1. A common approach to file-and-folder hierarchies within file storage systems (i.e., technical specifications)
2. A community of practice around which to discuss issues of filesystem storage and develop common understandings (i.e., shared knowledge)
3. An ecosystem of software tools that encapsulate shared practices, clarifying and identifying best-practices for digital preservation (i.e., collaborative development)

This discussion paper is an effort to clarify the need for the OCFL initiative; to set out some technical principles (1) in order to promote and invite discussion and critique (2), leading to specifications (3) that may be developed into software to meet the real-world needs of the community. The specifications presented in this paper are meant as an invitation to critique, and may not represent the final, accepted form of the OCFL specification.

Need and Prior Art

Within the context of this paper, a digital repository software application is understood to be software that provides two primary functions: To manage files, and to manage metadata about those files that describe their contents, origins, and uses. This discussion will focus primarily on the file contents; secondary consideration will be given to the metadata as it relates to file management within an Archival Information Package[2].

A filesystem layout is, simply put, an expected hierarchy of files, directories and naming conventions. In this sense, the OCFL shares something in common with efforts like the Linux Filesystem Hierarchy Standard (FHS), which provides guidance for the directory

[1] The name "Oxford Common Filesystem Layout" should be understood in the same way as "Portland Common Data Model," and "Dublin Core" -- that is, it is not specific to the needs of the University of Oxford, but that the idea first arose at a conference in the city of Oxford.
[2] The term Archival Information Package is defined in the OAIS Reference Model https://public.ccsds.org/pubs/650x0m2.pdf

Deleted: system

Commented [2]: fixed on github

Commented [3]: Needs a succinct statement near the top on the problem(s) OCFL is trying to solve

Commented [4]: +1

hierarchy on POSIX systems and their expected contents, but not on the specific contents of these directories.

The files in a digital repository represent a wide range of digital outputs from the collection activities of an institution. They can be pre-print PDFs from institutional researchers, e-Theses from graduate students, datasets that were the product of research initiatives, or digital images from digitisation activities. Owing to the abundance of media types, and the applications used to create them, the digital repository system must treat these files as opaque to the system, managing their method of storage but not their contents.

The management of these files vary between applications, with many of them choosing to store their files in an implementation-specific way. Fedora 4, for example, delegates storage of binary file attachments to its Modeshape system by default[3]. This system uses a combination of relational database and on-disk files and folders to store and manage binary objects. DSpace uses its own internal bit stream storage methods which differ from Fedora.[4] E-prints has an object storage system that is managed through its own API.[5] In each of these cases, the needs of the application, and the content the application is being asked to manage, are mixed so as to become indistinguishable. A DSpace filesystem hierarchy cannot be managed by Fedora, and vice-versa.

Some institutions, however, choose to implement a standard for file system structures independent of specific digital repository systems. The two most significant efforts for this are the d-Flat, ReDD, and associated specifications from the California Digital Library,[6] and the Moab efforts at Stanford.[7] Both of these efforts attempted to describe software-agnostic approaches to storing digital contents on a filesystem. Other efforts include "RecordSilo" as part of the University of Oxford's DataBank system[8] and the University of Notre Dame's "Bendo" adaptation of the Moab format.[9] The BagIt specification[10] , adopted by Library of Congress and Research Data Alliance[11], is also notable in this context for providing inspiration for several filesystems in use, despite being designed primarily as a format for object transfer and not object structure.

In the inaugural OCFL community call on 2017-12-01, the observation was made by Mark Phillips (UNT) that the software that manages their digital repository has been rebuilt several times, but that the structure of the filesystem has changed very little. This observation highlights several key principles:

[3] https://wiki.duraspace.org/display/FF/ModeShape
[4] https://wiki.duraspace.org/display/DSDOC6x/Storage+Layer
[5] https://wiki.eprints.org/w/StorageController
[6] https://confluence.ucop.edu/display/Curation/D-flat
[7] http://journal.code4lib.org/articles/8482
[8] https://github.com/anusharanganathan/RecordSilo
[9] https://github.com/ndlib/bendo/blob/master/architecture/bundle.md
[10] https://tools.ietf.org/html/draft-kunze-bagit-14
[11] https://rd-alliance.org/approaches-research-data-packaging-rda-11th-plenary-bof-meeting

Commented [5]: is this it

Commented [6]: Software is still needed to understand the layout of these file system conventions (otherwise all access is manually done by a human). Maybe its that these attempt to define (at least in the CDL case) file system conventions that will be adopted/usable by multiple institutional repositories?

Commented [7]: Agreed. I believe, however, that this "software-agnostic" claim is coming out of the context of the previous paragraph where the content storage is described as being intertwined with the repository software.

Commented [8]: Maybe that point could be made more clearly?

Commented [9]: Moab did have software that came with it in order to deposit content, but when actually reviewing the content, the "software" was just shell scripts that let you quickly traverse the file system (i.e. so you didn't have to type cd /ab/123/cd/4567/ab123cd4567/v1/....). plus using the shell scripts made it impossible to delete something from the file system.

Commented [10]: Yes; Moab's contents can be hand parsed pretty easily once you're familiar with the conventions - both metadata and preserved content. It was a deliberate design decision not to rename preserved files on disk, for example, to maintain readability.

Commented [11]: We were surprised that BagIt was being for repository AIPs, but that was because more full featured storage options were available.

Deleted:

1. Application-specific needs should be separable from the contents that the application is asked to manage. An institution will almost inevitably need to replace or rebuild the software on top of a filesystem hierarchy.
2. Filesystem hierarchy migrations place the underlying repository at risk. Mapping from one system to another may involve changing semantics ("what constitutes a file?") or be subject to built-in optimizations (e.g., de-duplication based on checksum) that may not be immediately obvious post-migration. While lengthy and comprehensive testing procedures can help identify and isolate these problems, it will not entirely eliminate them.
3. Digital objects within a filesystem hierarchy should be designed with the assumption that they will be managed by many different applications. Access and digital preservation is a process and not a fixed objective, and no single application will encapsulate all necessary actions.
4. File and directory hierarchies are pervasive organizational metaphors across most computing systems, and as such can persist across CPU architectures, disk formats, and operating systems. Object storage systems, such as Amazon S3, also persist this metaphor. As such, it is a fundamentally stable "technology" on which to build a digital repository.
5. Many repository systems store large amounts of data (hundreds or terabytes or petabytes) that are time consuming and/or expensive to migrate or reorganize. Thus stable filesystem organization has significant value

The OCFL initiative, in following the example of Moab and the CDL family of specifications, seeks to promote the filesystem hierarchy as separable from a management application, and a fundamentally stable and loosely coupled component of a digital repository. To this end, it will:

1. Provide a specification of the filesystem layout independent of repository management software implementations;
2. Enable the preservation of the object, and versions of the object;
3. Maintain a record of actions on the object;
4. Permit the object, and the collection of objects, to be validated against a versioned specification;
5. Not require special software to provide basic functions of viewing, moving, copying, renaming, and deleting, other than those shipped with a given operating system;
6. Work with a wide variety of storage systems, including disk (local and network), tape, and cloud storage platforms;
7. Permit the full restoration of digital objects within a digital repository using only data stored on the filesystem.

The following discussions, although they may be worthwhile in their own right, are deemed out-of-scope for the OCFL efforts:

1. Storage systems that do not maintain the file-and-directory hierarchy. This includes solutions that use databases, such as MongoDB or Cassandra;

Commented [12]: Starting about here we have two different concepts of object: the thing and the thing and all its versions (and admin info) -- sometimes the latter is "OCFL object" but there needs to be some convention to separate the concepts

Commented [13]: this is suspiciously vague. Cygwin? *nix with gnu utils? OSX? Chromebook?

Commented [14]: +armintor@gmail.com : Agreed. Wordsmithing would be appreciated.

Commented [15]: Above, a repository is described as software, and per Mark's comment a repository can come and go, it is the digital objects that we are trying to persist.

Commented [16]: Agreed. I believe the essence of this bullet is to say that a given software stack built over OCFL content can be completely deleted and reloaded (rebuilt) from the OCFL data on disk. Similar to Fedora 3... but not Fedora 4.

2. Discussion of the advantages and disadvantages of specific filesystem formats, such as ZFS, NTFS, or XFS. The OCFL should work on top of any given filesystem as long as it maintains the file-and-directory hierarchy;
3. The contents of each object beyond administrative metadata. OCFL will be implemented in a variety of contexts, and as such will not mandate any set of binary or object metadata formats;
4. Software dependencies. The only 'software' needed to create an OCFL system will be available built-in to an operating system, and no other dependencies will be permitted.[12] (Note that this does not preclude the creation of OCFL libraries that help manage these operations, but there is no dependency on these).

Design Principles

> For reference, a gist containing a prototype directory structure and version file can be found here: https://git.io/vNt22

An OCFL *digital object* is a collection of files and metadata that can have a notional but largely implicitly-understood boundary: "This is the thing, and this is not the thing." An e-Thesis, for example, could be composed of a single PDF file, or it could be composed of several Latex files, a number of datasets, and a PDF file. The object should also contain a record of the metadata that describes the origin, character, and purpose of the collection of files. This might be stored in Dublin Core, METS, MARC, or a collection of several standards as needed. OCFL will not mandate the use of any particular metadata format, as this is likely to be governed by local considerations.

An OCFL digital object sits within a hierarchy of files and folders. A common practice is to use a unique identifier scheme to compose this folder hierarchy, typically arranged according to some form of the PairTree specification[13]. Many different object identifier schemes are in use across institutions. Some use UUIDs, since they can be cheaply 'minted' and have a high probability of global uniqueness. Examples of other object identifier schemes include ARK and DRUIDs.

OCFL will not recommend a particular identifier scheme over another, nor will it recommend a particular method of subdividing these identifiers into paths. These are likely governed by local considerations, such as the amount of identifier entropy required, and the identifier standards adopted by the institution. What will be assumed, however, is that a digital object's folder name will correspond to its full and complete identifier. To illustrate, for an object with the identifier "abcd123g":

Commented [17]: Most of this "Design principles" section is a draft specification... while I like specs at least as much as most people, I think it would be really helpful -- especially at this early stage -- to clearly separate out the discussion of design principles and options considered.

Commented [18]: Agreed. Being very clear on "Design Principles" would be helpful. A following section could go into "Implementation Considerations".

[12] As always, there is an exception to this rule. OCFL mandates the use of SHA256 checksums for implementing fixity checks and tracking file identity across versions. There are SHA256 checksum utilities available for every operating system, but they may not be installed by default. Checksum utilities are only needed when adding new files to an OCFL object, or validating its contents.
[13] https://confluence.ucop.edu/display/Curation/PairTree

Incorrect
/ab/cd/12/3g/... object files here

Correct
/ab/cd/12/3g/abcd123g/... object files here

This will allow the digital object to be easily migrated to other hierarchical structures should the need arise:

Alternative Hierarchies
/ark:/12345/abc/d123/g/abcd123g/
/my/repository/abcd123g/

Digital object identifiers should have enough "uniqueness" in them to be able to identify them throughout a given namespace. While more human-friendly object identifiers like "book-object-1" are *possible*, consideration should be given to how likely it is that two 'book-object-1' objects will ever need to be stored together or available in the same namespace.[14]

OCFL objects and file systems should be self-describing. The OCFL should adopt semantic versioning as a way of managing compatibility between the filesystem layout and client software. Some initial suggestions for making OCFL filesystems self-describing:

1. Require a file in the root of the repository giving the OCFL version number.
2. Require a copy of the plain-text OCFL documentation to exist in the root of the repository.
3. Require a file within the digital object giving the OCFL version number.

An OCFL digital object is composed of several required files and directories. The required files are:

1. An empty file whose name gives the Namaste-formatted[15] OCFL version.
2. A 'versions.jsonld' file containing the object's manifest and version history
3. An empty file whose name gives the Namaste-formatted SHA256 checksum of the 'versions.jsonld' file
4. A numbered sequence of versioning directories, starting at v0001.
5. A 'logs' directory

An example is shown in Figure 1:

```
.
├── 0=ocfl_object_1.0
├── 1=6881d9eea5cc6558fc48113307a967266dd29a845713615c9bb012566065e3fa
├── logs
```

[14] While the objects themselves may not reside in the same parent folder, they may end up with the same public URL, and may thus the collision would appear only when publishing.
[15] https://confluence.ucop.edu/display/Curation/Namaste

Commented [19]: If the id here has been migrated to ark:12345abcd123g then the path is now illegal as the folder name does not correspond to the complete id

Commented [20]: Later comments seem to assume that mixed versioning is not allowed but I'm not sure that is necessary/helpful -- the specs could allow a mix and make it a local choice as to whether to to enforce a uniformity check for a specific version

Commented [21]: What happens with 10,000th version? Need to define whether that is illegal (and debate number of digits) or there is extension to v10000 to deal with possible edge case (as in e.g. https://github.com/ndlib/bendo/blame/master/architecture/bundle.md#L16-L18)

Commented [22]: +1

Commented [23]: It's a concern, but in the many years that Stanford has been using this format, the average version # in our repo (out of 1.5MM objects) is 2.5, and the highest version number is 20.

Commented [24]: And some of those high version numbers are because people were adding periods to the ends of titles.

Which brings up another point, we aren't going to say what constitutes a new version, correct?

Commented [25]: Why not just drop the leading zeros, the maximum version is thus limited by filesystem naming.

Commented [26]: I agree that dropping the leading zeros is probably the cleanest solution -- the combination of manually looking at the directory structure AND having > 9 versions very often seems like a real edge case

Commented [27]: The risk with dropping the leading zeroes is ASCIIbetical sorting -- v1, v10, v100, v2, v20, v200, etc.

I agree that dropping the zeroes has significant advantages, so I'm ok with it, but it would be good to recognise the disadvantages as well.

Commented [28]: +1 for some type of ASCIIbetical sorting. what would be the advantage of having all those 0s rather than just a single 0. i would imagine there is some, but i can not think of one (or remember [1]

Commented [29]: => https://github.com/OCFL/spec/issues/2

Commented [30]: I'm still uncertain about this. I'd like to capture provenance as part of version metadata (so inside the vNNNN directories), perhaps with the implicit [2]

Commented [31]: Julian, when you say provenance do you mean the audit trail or something else?

Commented [32]: The significant events of the object; we're taking that to *not* mean regular audits. The audit cycle is a property of the repo and we assert that the [3]

Commented [33]: To add to that; we're saying that the repo guarantees that each object will be fixity checked within a certain time period; if an object does not meet [4]

Commented [34]: I don't think fixity is necessary (see my DPC posting https://www.dpconline.org/blog/thoughts-on-fixity-checking-in-digital-preservation-systems). This [5]

```
│   └── .keep
├── v0001
│   └── .keep
└── versions.jsonld
```

Figure 1: A sample empty OCFL object. (.keep files are shown to illustrate directories and are not required)

The structure and contents of the versions.jsonld file will be discussed later.

Symbolic links and aliases within digital objects are invalid within an object for three reasons. The first is that they are not easily portable across file systems, especially Windows to Linux. The second is that they do not translate well to an object store system, such as S3. The third is that on some file systems there may be limits to the number of inodes, and this may be reached quite quickly if objects (and object versions) depend on symlinks to implement this behaviour.

One challenge that has been identified in discussions thus far is that modern storage systems are typically optimised to store files at an expected minimum size. This typically comes down to the inode size specified when the disk is formatted. The impact of this is that filesystems optimised to store larger files (datasets or high-resolution imagery) are inefficient for storing many smaller files. Likewise, a system optimised to store smaller files may significantly underperform when attempting to retrieve large files. For OCFL, which is implemented using a mix of many "small" files (the administrative metadata) as well as large files (the data portion of the digital object contained in the version directories), these considerations may have an impact on the recommendations for the construction of a digital object.

Commented [35]: I think this needs to be pulled apart to work out what the design principle is and then think about whether current ideas meet it

Commented [36]: I'm not sure this is true any more: inodes have negligible size but they map onto allocation units on storage which may have a minimum size efficiency limitation (unless the filesystem uses extents). However, zero length (and sometimes very small) files usually don't occupy allocation units - just inodes. Many filesystems allow you to configure inode numbers on creation.

It may be suitable to understand a digital object as either a collection of "plain" files and folders, or a single uncompressed TAR ("tape archive") file[16]. On modern systems, the overhead of seeking and extracting files from a TAR is low, especially if there is no overhead in decompressing the files. This may provide a workable solution for storing the object and its administrative metadata on a file system optimised for storage of larger files. Further study of this is required, especially as it applies to implementation on Windows systems.

Because OCFL does not mandate the use of a particular metadata format, it should be understood that the 'rebuildability' of an OCFL repository is bounded by the ability of a piece of software to understand the metadata formats within the object. That is, given an OCFL-compatible digital repository system, it should be able to import and understand its contents, version history and fixity. It is not, however, required to understand METS or Dublin Core, just as it would not be required to natively understand a given image or dataset format.

[16] Preliminary suggestion; open for discussion.

Filesystem Roots

A root directory is the base folder of a filesystem layout. It is not required that this sit as the top-level directory of a filesystem, as any OCFL root will contain a Namaste file identifying it as such.

Institutions may require multiple 'roots'; that is, several logical or physical volumes, or multiple 'buckets' in S3. Each OCFL root should be treated independently, and signal its version compatibility to client software. For example, an institution may have a 'newspapers' project that standardises on OCFL 1.0, but then may have a later letter digitisation project that wishes to use OCFL 1.2. Clients should be able to understand these versions, and must refuse to operate on OCFL systems for which they have not been tested.

A root directory should contain a Namaste-formatted file giving the OCFL version for this root. This is necessary to support validation of the filesystem layout, ensuring that objects conforming to a later version do not get created within an older-version root (e.g., an OCFL version 1.2 object in a version 1.0 root).

The root directory should also contain the OCFL specification in human-readable plain-text format in the root.

The root directory should contain its own ‘logs’ directory. The purpose of this is to track any problems with the structure of the layout on a whole. Problems with individual objects should be logged within that object’s own logs.

It is a validation error to store objects in an OCFL root that do not conform to the OCFL specification.

Commented [37]: I think there is a design principle to do with validation here -- given an OCFL filesystem it is possible to validate its conformance, and such validation should give x, y, z guarantees

Versioning

Object versioning is implemented using a folder hierarchy within the digital object’s root directory, and a "plain text" description of its history. An OCFL object's versions are immutable; once created, they should not be changed. Changes to the object must result in the creation of a new version. Versions are implemented as folders within the object root directory. 'v0001' would be the first version. From there, new version folders are created to hold subsequent changes.

While the objects representing the versions are 'plain' filesystem objects, it is anticipated that OCFL-specific software will help implementers create and manage object versioning. It is entirely possible to create versions 'manually,' but this would be a tedious task.

The 'versions.jsonld' file provides both the object's complete manifest, and its version history. An example can be seen at https://git.io/vNt22. The structure is explained below. (m = mandatory, o = optional).

- "`@context`" (m): Provides the versioned OCFL LD context for this JSON-LD structure.

Commented [38]: are extension contexts allowed? are extensions allowed?

Commented [39]: JSON-LD is great, but is in the middle of upheaval from v1.0 to v1.1. and v1.2 could easily come along in a couple more years. I have some concerns that it is too much of a moving target for an internal repository data format in systems that will have archival application

Commented [40]: +1.

Commented [41]: Yes, but ... :)

I would probably prefer JSON-LD in its unstable forms, over arbitrary non-typed JSON.

I would also prefer JSON-LD to XML, since the costs for that outweigh the benefits. Parsing XML, and all of the cruft that comes with it (namespaces, xpath queries, etc.) seems overkill for our uses.

Commented [42]: => https://github.com/OCFL/spec/issues/1

- "`@id`" (m): Provides the unique ID for this document. Most OCFL objects will not be directly web-addressable, so this may take both URL and URN values such as "urn:uuid:..." or "urn:ark:...".
- "`@type`" (m): Must be "ocfl:Object"
- "`head`" (m): Provides a hash-fragment pointer to the version that is the most recent, as identified by its "@id" value. So "head: #v0003" would point to the version object identified by "@id: v0003".
- "`manifest`" (m): Provides a complete manifest of every file in the object. This is implemented as a map, providing a SHA256 key and the file's path within the data folder as value.
- "`versions`" (m): Provides an array of all versions. Each version is an object containing:
    - "`@type`" (m): Must be "ocfl:Version".
    - "`@id`" (m): The ID of that version within the object. This is used as described in the 'head' tag.
    - "`created`" (m): Timestamp of the version's creation
    - "`message`" (o): An optional "commit message" documenting what changed in human-readable form
    - "`client`" (o): An optional string identifying the client software that created the version
    - "`user`" (o): An optional object containing "name" and "email" identifying the person who triggered the version. Automated processes may also use these tags to identify themselves.
    - "`members`" (m): An array of SHA256 keys for the files that make up the version.

Files within the OCFL digital object are identified by their checksum. While Moab allows MD5, SHA1, and SHA256 checksums, OCFL will support just SHA256 (based on "lessons learned" from the Moab implementation). The identification of these files by their checksum permits maintaining the original filename within the versioned directories of the OCFL object.

It is an error to attempt to write the same file to the manifest twice, as identified by the SHA256 checksum. In the case of a reversion (that is, a file is deleted in a version and then reinstated in a new version) the new version may still point to the older file, since no files are deleted within an object's history.

All files present in the version directories must have an entry in the 'manifest' map.

The 'versions.jsonld' structure offers several key features for clients:

1. A client wanting the files in the latest version of the object just needs three pieces of information: The latest version (provided by the 'head' key, or the last version in the 'versions' array), the 'members' array for that object, and the "manifest" map which maps the SHA256 keys to the paths within the object.

Commented [43]: technically, it would be an error to write two files with the same SHA256 digest ;-) Negligible possibility of this happening by chance, but in a few years when asked to archive data from a cryptography researcher... But anyway, it should be written precisely, perhaps with a comment/link about why it is OK to have this restriction

Commented [44]: How is the renaming of files in subsequent versions accommodated?

Commented [45]: +1 -- TBD

2. A client requesting any previous version would likewise be able to read this list directly from the ‘members’ map for that version.
3. Ordering of the files within the object may be implemented within the ‘members’ array.
4. Version differences may be calculated by performing set operations between two ‘members’ arrays, showing the additions, deletions, or overlap between those versions.
5. Reverting an object to a previous state is a no-cost operation, since it entails no copying or moving of files, just a creation of a new object in the ‘versions’ array with the appropriate checksums listed.

Only under certain circumstances should it be necessary to expunge a file from the object’s history (e.g., sensitive data or copyright infringement).

It is recommended that for every version a serialized copy of the object’s metadata is included within the object.

The paths within the manifest block should be relative to the object’s root. They must not be absolute paths, as this would prevent migration of digital objects to other locations.

Should the “versions.jsonld” file become corrupted or out-of-sync, there is a risk that this will have a detrimental impact to the structure of the object. No data would be lost (the files would still exist) but fixity values may be out of sync. Possible failure modes might include:

1. The ‘versions.jsonld’ is missing or becomes completely unreadable. The probability of this is quite low. In this case the manifest could be reconstructed with new fixity values (which might be suspect). If files maintain the same paths relative to their version roots, it may be possible to reconstruct the object’s version history. (e.g., “v0001/page1.jp2” would be replaced by “v0002/page1.jp2”).
2. An operation on the object did not update the “versions.jsonld” file correctly. This might include creating a new version entry, storing the file checksums in the manifest, updating the value of “head”, or performing an operation on an object that does not match the expectations of the OCFL version.

Logging

Every OCFL Object has a ‘logs’ directory. This is meant to store records of all the actions taken on an object. The exact nature of these logs will vary from institution to institution, so the contents of this directory will be governed by local considerations. These logs are not versionable within the object; creating a new log does not mandate creating a new version of the object.

It is recommended that the following logs be maintained for every object:

1. A record of any periodic audits. If your system performs annual or monthly fixity checks or format checks, there should be a record of these in your logs.
2. A record of any versioning actions.

3. A record of any reported problems with the object. This could include failed fixity checks, failed format checks, or failed OCFL validation checks.

As non-versionable objects, maintaining checksums of the log files is not required. Institutions may wish to implement log format validation, but this is not required within the OCFL specification.

Validation

OCFL directory roots and objects should be validated periodically. An OCFL validator will check these objects against the specifications given in the stated version of the OCFL specification.

Directory validation

- A Namaste file exists with the name "0=ocfl_X.Y.Z", containing the version specification for that root.
    - If a client does not understand "X.Y.Z" -- it is too old or too new -- it should refuse to proceed.
- A plain text file with the name "ocfl_X.Y.Z.txt" exists. This will contain the specifications.
- Any directory within the OCFL root that contains files and does not identify itself as an OCFL object, or a TAR file, is an error.
    - In the case of a tar file, the "0=ocfl_object_X.Y.Z" file must still exist.
- The presence of any aliases or symlinks within the directory structure is an error.
- A "logs" directory must be present in the directory root.

Object validation

- A Namaste file exists with the name "0=ocfl_object_X.Y.Z", containing the version specification for the root.
    - If a client does not understand "X.Y.Z" -- it is too old or too new -- it should refuse to proceed
    - If "X.Y.Z" in the object differs from "X.Y.Z" in the object's directory root, it is an error. An object must not have a different version than its root.
- A Namaste file exists with the name "1=[SHA256]". This checksum should correspond to the checksum of the "versions.jsonld" file. If it does not, it is an error.
- A "logs" directory must be present in the object.
- A "v0001" directory, at least, must be present.
- Any other directories must begin with "v" and contain a sequence of four digits
    - There must be no gaps in the sequence; it is an error to have "v0001" and then "v0003".
    - For each version directory there must be a corresponding entry in the 'versions.jsonld' "versions" block with an "@id" value corresponding to the folder name. It is an error to have a mismatch between these two.
- It is an error if a file within an object's manifest does not match its checksum

- It is an error if a file exists within a version directory that is not also listed in the manifest.
- It is a warning if a file exists in the manifest that does not exist in one of the versions' "members" array (effectively a hidden file).
- It is an error if the a file with the same checksum value appears in two or more version folders.
- If an object is implemented as a TAR file, it is an error if this tar file uses compression (e.g., ".gz" or ".bz")
- Versions.jsonld structure:
    - It is an error if the value in the "head" of the versions.jsonld does not correspond to a Version object's "@id" value.
    - It is an error if the outer object in the "versions.jsonld" file is not "@type: ocfl:Object"
    - It is an error if any of the mandatory values are missing from the "versions.jsonld" file.

Commented [46]: Is this a warning to allow for things like evil `.DS_Store` files? If I were implementing a validator I would make it take a list (system specific) of allowed extra file names and throw an error if anything else were found

Commented [47]: I think I made it a warning because, as an 'untracked' file, it does not affect the completeness (and therefore "validity") of an object as an OCFL concept. But it should be flagged and, if desired, ignored by a logging and monitoring system.

This way it might flag incomplete manual intervention, a bad OCFL client implementation, or malicious writes.

Implementation Patterns

While the purpose of the OCFL is to provide a 'bare-bones' approach to storing files using well-tested technologies, it is also assumed that these files can be more easily managed through client software, particularly with respect to the actions surrounding versioning of objects. This client software would need to conform to the OCFL specification, providing easier access to functionality such as "give me the latest version of object x", "show me all versions of object y", or "create a new version of object z with files A, B, and C." It is hoped that multiple, independent client implementations against the OCFL specification will be available to permit adoption in a wide variety of environments.

Some examples of possible client implementation patterns might include:

Direct read/write access

An OCFL client library is used within a digital repository system. The library coordinates the storage of files on the disk and reading those files back on request. The system hosting the repository software accesses the OCFL repository disk directly (e.g., physically attached or mounted with NFS).

Remote read/write access (OCFL-Agnostic server)

Suitable for object storage systems like S3 or minio. The digital repository system interacts with the file storage layer by translating OCFL-specific actions into the appropriate HTTP or RDP requests. The server system is OCFL-agnostic, knowing only how to store and access files, so any translation to the appropriate HTTP requests, for example, must be pre-calculated by the client prior to be sent.

Remote read/write access (OCFL-Aware server)

Similar to the server-agnostic approach, but where the server understands OCFL. Calls to the server might be done over HTTP (i.e., REST) using the client's OCFL-specific API, and OCFL-specific actions may be offloaded to the server. Suitable for institutions implementing microservices that require coordinated access to the same data. (The nature of the client API is not part of the core OCFL specification, but it may be a suitable topic for a complementary specification.)

Export-only

Institutions may allow their repository software to manage their own filestore, but implement OCFL as an export format suitable for long-term archiving. This may be on-demand or as part of a periodic backup scheme.

Import

Particularly in disaster recovery scenarios, repository software should be able to import an OCFL filesystem. This may be required in the case of disk failure or accidental file deletion, where the only option is to restore the repository from backup. This may not mean that all the object metadata within the OCFL object itself is imported and understood: A system built to understand only Dublin Core files would not understand how to import the metadata from METS files. However, it should mean that the repository software should understand and import object versioning and file fixity values held in the OCFL object.

Use Cases

Following use cases may not have been transcribed to GitHub issues:

Cached Read Access

An institution may wish to implement access to the filesystem objects by storing the direct, absolute paths to the files within a Solr index. An OCFL client is used to retrieve and resolve these object paths to their absolute values in the indexing process.

Eventual consistency

An institution may implement object writes through a Queue system, where write actions are decoupled from the calling application and are handled asynchronously. Several worker processes may use OCFL clients to manage these writes.

Microservices

An institution may implement their repository as a collection of smaller microservices. These microservices may operate independently on a given OCFL directory root, or set of roots. Some examples might be a IIIF-compatible image delivery service, a directory

Commented [48]: Should these now be removed from this document in order to focus discussion on the github issues?

Commented [49]: +1

Commented [50]: --> https://github.com/OCFL/Use-Cases

Commented [51]: What are the implications or requirements of async write? How might a client reading data understand or deal with possible inconsistency? (And does the model help/hinder)

Commented [52]: Some locking or state files may need to be specified. If these are present then we know there has been an unscheduled dismount and can proceed accordingly. Incrementing current version pointers last would be an example.

validation service, an object validation service, a metadata indexing service, and format validation (e.g., jpylyzer).

OCFL for tape backup

An institution has decided that they wish to export their digital repository contents to a disk that is used as a staging system for their tape backups. They build an exporter capable of writing incremental changes to this disk. The tape backup system provides incremental backups weekly, and then a full backup once a month.

OCFL for non-binary files

An institution uses a repository system to manage its collection of EAD files. A periodic export from this system writes the EAD metadata to individual OCFL objects. Their disaster recovery relies on this repository system to read in the files from the OCFL directory root, updating the database from the serialized EAD, including import of all previous versions.

Client Specifications

There may be four types of OCFL clients in use. Roughly in order of complexity:

1. OCFL Read/Write client. Able to understand and create OCFL objects. Can retrieve files from versions, create new versions, and write the 'versions.jsonld' file. In addition may be able to assist logging applications with writing logs to the object or directory logs.
2. OCFL Read-only client. Able to retrieve files from a given object and version, but cannot write to the object.
3. OCFL Object validation. Able to validate the contents and integrity of an OCFL object.
4. OCFL Directory validation. Able to validate the structure of a directory root and directory hierarchy.

The community may wish to develop a common API, including API methods, error codes, incremental exports, and file locking behaviours. However, these behaviours will not form part of the OCFL specification.

Request for Comments

As mentioned in the introduction, the purpose of this paper is to invite critique and to make explicit some of the ideas that have been floated. Comments on all aspects of this specification are welcome, but there are a few areas in particular where I feel there needs to be feedback:

1. Moab, on which this design is heavily based, contains significantly more administrative metadata. The OCFL specification simplifies this design to make resolving versions a more explicit process (see: http://journal.code4lib.org/articles/8482#5.1.5). However, it is important to flag any functionality that has been lost due to this simplification.

2. The choice of SHA256 was based feedback from the first community call, where it was stated that Richard Anderson would have preferred to have standardized on this to begin with.
3. Likewise, the choice of JSON-LD over XML or plaintext for the version information was to make the markup more lightweight, while still allowing some types of semantics to be embedded in the administrative metadata.
4. My JSON-LD knowledge is fairly rudimentary, so critique of the structure and values embedded in the JSON-LD example would be appreciated.
5. Some standards from the CDL family of specifications were adopted, particularly Namaste. I was inspired by John Kunze's remark that the purpose of Namaste was to "declare a directory's "type", somewhat like a file's "magic number." PairTree-ed directories are likely to appear in OCFL instances, but this is not a requirement. Are there other "micro-specifications" that can be useful? Perhaps the "LockIt" spec?[17]
6. More use-cases are always welcome.

[17] https://confluence.ucop.edu/display/Curation/LockIt

Page 5: [1] Commented [28] **Rosalyn Metz** **02/04/2018 13:36:00**

+1 for some type of ASCIIbetical sorting. what would be the advantage of having all those 0s rather than just a single 0. i would imagine there is some, but i can not think of one (or remember why it was done that way although i can remember being told why...)

Page 5: [2] Commented [30] **Julian Morley** **09/03/2018 16:17:00**

I'm still uncertain about this. I'd like to capture provenance as part of version metadata (so inside the vNNNN directories), perhaps with the implicit assumption that versioning includes a full fixity check. I'm also worried about the management overhead of having potentially millions of /logs directories that may be growing over time, containing data that we don't explicitly need to preserve.

Page 5: [3] Commented [32] **Julian Morley** **09/03/2018 16:29:00**

The significant events of the object; we're taking that to *not* mean regular audits. The audit cycle is a property of the repo and we assert that the object is good unless there's a provenance event that says, "scheduled audit found a problem! (and hopefully fixed it)."

Page 5: [4] Commented [33] **Julian Morley** **09/03/2018 16:31:00**

To add to that; we're saying that the repo guarantees that each object will be fixity checked within a certain time period; if an object does not meet that guarantee, that's a provenance event. But the regular audits are not - there'll be too many of them.

Page 5: [5] Commented [34] **Neil Jefferies** **16/03/2018 14:26:00**

I don't think fixity is necessary (see my DPC posting https://www.dpconline.org/blog/thoughts-on-fixity-checking-in-digital-preservation-systems). This is about file identification primarily.