OXFORD COMMON FILESYSTEM LAYOUT

OXFORD COMMON FILESYSTEM LAYOUT

Version 0.1 Alpha

Storing and describing files are central to the functionality of institutional repositories. Unlike catalogues, where an electronic record exists to point to a physical object, an institutional repository itself contains and manages the electronic objects as well as the cataloguing data.

There are currently no agreed upon practices, however, for the low-level filesystem structures that institutional repository systems adopt to store these objects on disk. Some systems delegate this responsibility to third-party libraries, treating the storage layer as a 'black box' [e.g., Modeshape]. Others implement their own software-specific filesystem hierarchy.[EPrints, Dspace?] Yet others take no particular view on this, leaving the filesystem hierarchy up to the individual institutions to implement according to local practices.

In each of these implementations, there is no common approach to storing both file data and metadata on the disk. This can have significant implications on the long-term viability of the data, especially in systems that are built as "fire and forget" -- that is, static collections that 'just work' until they do not.

This document will propose a common approach to filesystem layout for institutional repositories, providing recommendations for how IR systems should structure and store files on disk. It is developed under the name "Oxford Common Filesystem Layout" (OCFL) because the impetus for this effort grew out of discussions held at the Fedora / Samvera Camp held at the University of Oxford, September 2017. It generally follows the model of naming an effort after the place where it originated (see: Dublin Core, Portland Common Data Model).

The goals for this effort include:

1. **Better support of decoupled microservices.** A common filesystem layout will provide an expected platform on which many services can act on the IR filesystem, independent of any single 'managerial' system. Services involved in delivery (e.g., a IIIF-compatible image server) can use the underlying filesystem directly, without needing to go through an intermediate retrieval system. Preservation and auditing systems can operate on the underlying data directly.
2. **Data migration and "rebuildability."** A common approach to filesystems, and a mandate to provide the ability to 'rebuild' an institutional repository from the

filesystem, can help obviate challenges in migrating from one system to another. (It could be argued that a large part of the time, and effort, involved in migrating systems is designing and building the process of translating from one filesystem structure to another.) Put another way, there should be no absolute requirement on a particular piece of software to use or make sense of the objects on the filesystem. A user (or software implementer) should be able to understand the repository with just the files (and possibly the OCFL spec for convenience.)

3. **Common object versioning model.** Digital object versioning has been implemented in several different ways, each with different impacts on storage capacity and performance. A common approach to this will provide an expected filesystem layout that follows best practices, but perhaps even more importantly it can document the decision process and discussions around these tradeoffs.
4. **Storage systems best-practices and recommendations.** Discussion around filesystem layout should not dig too deeply into specific implementations; however, there is a need for high-level discussions at the intersection of implementation and layout. For example, if object versioning relies on symbolic links, how does this translate to cloud-based systems like Amazon S3? What are the recommendations for storage systems that can span multiple devices or protocols? Are there any filesystem design practices that can have a negative impact when implemented "at scale"?
5. **Backup.** By storing the institutional repository data as 'plain' filesystem objects, and making a requirement on 'rebuildability' from the filesystem, the system presents a single interface to repository duplication. Backing up the entire repository is simply backing up the filesystem, without the need to rely on external processes for database exports. (Of course, database exports can still form part of a backup strategy, but an "apocalyptic scenario" disaster recovery strategy does not rely on having these files).
6. **Validation.** Part of the efforts around OCFL should be the creation of a filesystem layout validation tool which can flag both errors and warnings against a given filesystem and its conformance to the OCFL specification.

Some preliminary challenges and considerations might include:

1. **The 'rug' problem.** If many systems are expected to operate on a unified set of files, how do we prevent any one system from "having the rug pulled out from under it" -- that is, operations that take place that change the underlying data in ways that it was not expecting.
2. **The 'common data model' problem.** Institutions implement metadata in many different ways that are not immediately amenable to a standardised storage

approach. At a minimum, what is required to create a standardised filesystem layout while recognising that the exact contents of the objects being stored can vary in structure.

3. **The border between layout and specific technologies.** While many systems implement filesystem layout in the standard 'file-and-folder' paradigm, others might use specific technologies, like Apache Cassandra or HDFS, that require different approaches in their implementation.
4. **The 'here or there' problem.** Some systems use both local and remote storage to store the underlying data. For example, small files might be stored locally, while large files might be stored in a cheaper cloud storage location and simply 'pointed' to locally. How do we resolve the problem of having files that are there, but not?
5. **The consistency problem.** Depending on the nature of the files, the systems being used to provide them, and the location to which they are being written, it may not be possible to provide a synchronous guarantee that data that is provided has been written. Should a guarantee of 'eventual consistency' be enough, or should there be an absolute requirement on atomic and synchronous write operations within the IR?

Previous art

The Unix Filesystem Hierarchy Standard[1] describes a common filesystem hierarchy expected of compatible systems. It allows both users and software to try and predict likely locations of operating system components, such as '/lib' for libraries, '/include' for headers, or '/etc' for configuration files. The OCFL aims to describe a filesystem hierarchy for the same reasons: To promote a common approach for both user and systems to understand and work with the file systems underlying their institutional repositories.

There are a family of specifications oriented around filesystem layout developed as part of the California Digital Library efforts[2]. This includes PairTree, ReDD (Reverse Directory Deltas), and D-Flat, and several other associated specifications. These specifications may be a useful starting point for our discussions.

The MOAB system at Stanford University Libraries[3] builds on the work at California Digital Libraries for object versioning. They considered implementation details such as choosing forward or reverse versioning and their relative complexities.

Notes

While the OCFL is intended to serve as the underlying storage layout, it makes no assumptions on the delivery mechanisms that sit above it. Implementers should be free to implement services that promote faster access to the underlying objects, such as relational databases, triple-stores, or key-value stores. For write operations, consistency with the OCFL store can follow the 'eventual consistency' model, while caching layers might provide a faster synchronous storage layer.

Document versions

0.1 Alpha Initial proposal; Authors: Andrew Hankinson

---

1. https://wiki.linuxfoundation.org/lsb/fhs
2. https://confluence.ucop.edu/display/Curation/Microservices
3. http://journal.code4lib.org/articles/8482