Overview

- Archive impl turned out to be more than a stub
  - This was due to the some complexities in the archive service API, and due to some of the expected integration use cases
  - Wanted to test the search index with archive data at non-negligible scale
  - “full dip” specification is still a question - to aid the index and access services.
  - Need to test batch load, and observe effects of populated data on searches, Uis, etc.

Overview

- Ended up creating a “framework” for relatively agile incorporation of stores and Dip production algorithms
- Pluggable stores. Initially created a filesystem-based impl. memory to follow. Could theoretically even create a Fedora impl
- Framework takes care of all entity parsing, dip assembly/disassembly based upon limited capability of underlying stores

ELM implementation (aka an archive in two days)

- Impl named “ELM” for “Entities and Link Metadata”
- Requires “two” underlying blob stores
  - EntityStore – simple get/put opaque blob for a given entity id. Contain dcp entity fragments
  - MetadataStore – get/put blobs containing key/value pair sets for a given entity id. Used in an “append-only” fashion.
    - MetadataStore is actually a bit more abstract than that. Can implement using a blob store, but can also implement read-only using some other underlying technolgy

Initial file implementation

- FsEntityStore: Each entity stored as a file containing a single dcp entity fragment
  - File named after a hash of the id. No lookup table necessary to retrieve a file
- FsMetadataStore: Each metadata blob stored as a single .csv file
  - File named after a hash of the id. Again, no lookup table.
- If using both fs store impls, there will be two files for each entity.

ELM features

- Stores can be arbitrarily dumb
- MetadataStore can be arbitrarily smart
- Pluggable DipLogic for building different kind of dips
- Can mix and match MetadataStore and EntityStore implementations when configuring the ElmArchiveStore.