Dependency: dcs-deposit-core

- Primary deposit interface (slightly abridged):

```
public interface DepositManager {

	public DepositInfo deposit(InputStream content,
				String contentType,
				String packaging,
				Map<String, String> metadata);

	public DepositInfo getDepositInfo(String id);
}

public interface DepositInfo {
	public DepositDocument getDepositContent();
	public DepositDocument getDepositStatus();
	public boolean hasCompleted();
	public boolean isSuccessful();
}
```

Ingest

- Contains two implementations of DepositManager
  - StagedFileUploadManager
  - DcpXmlSipStager
- Also contains core ingest utilities:
  - Bootstrap – starts an ingest
  - EventManager – manages events associated with a deposited SIP
  - SipStager – Stores SIPs during the ingest process
  - FileContentStager – Stores file content during ingest process.
- As well as ingest service implementations.

FileSystemContentStager

- Implementation of FileContentStager that stores files in a filesystem using a given algorithm
- For Y1P, using a sha-1 content digest to determine file path. This implies that the stager is content-addressable (only one file location for unique content). Hash calculated on the fly as content is streamed to disk.
- Intended to be the “final location” of uploaded files.
- For files that don't make it “in”, there is a delete() method. In this impl, it appends the path to a text file listing all delete requests. Actually deleting files needs to be done with care, separately.

InlineEventManager

- Impl of EventManager
- Simply writes all events to the SIP for storage in the archive
- If we do not want to archive all events, we would need another EventManager implementation

ElmSipStager

- Implementation of SipStager that uses an ElmEntityStore and ElmMetadataStore to persist sips in the filesystem
- Currently, does not remove sips after ingest has been completed. This allows persistent access to sip status and content through the sword/app interface
- Can be cleaned out manually with no consequence by removing staged files directory.

Bootstrap

- Adds a sip to the stager, and kicks of an ingest
- Current implementation is ExecutorBootstrap
- Configurable/injectable Executor for control over execution behaviour (synchronous, thread pool, etc)
- List of services to execute is injected as well
- Later on, will implement an orchestration-based bootstrap that kicks off a process orchestration.

IngestService

```
public interface IngestService {

    public void execute(String sipRef);

}
```

Very simple interface. Ingest service impl is responsible for using ingest framework components (SipStager, EventManager, FileContentStager, etc) to achieve its task. The intent is for these to be called individually in an orchestrated ingest

Part 3: Archive

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.