Overview

This is a living document to convey some of the approaches that are being taken during the conversion process from direct ActiveFedora calls in Hyrax to calls through the Valkyrie based Wings Adapter for ActiveFedora.

The patterns described here will shift over time and there is potential for this document to not track all changes. If you see one of these patterns in use but not quite following the pattern as described here, contact devs on the hyrax-valkyrie channel to confirm whether the pattern is accurate or has changed.

Transition Goals

- Replace all persistence/retrieval with Valkyrie calls through a Valkyrie Resource
- Wrapping AF behavior is a means to move toward this goal to allow us to incrementally move toward all calls through Valkyrie

Work in models

- mapping parts of the AF API so it can be used via a Valkyrie::Resource

Large Area

Next steps

- AF queries: Use Valkyrie-native for this. In short term, provide a wrapper service?
- In hyrax code, pick a call on an AF object, cast to Valkyrie there and use the new model methods, then cast back to AF.
    - possibly start close to user input to reduce swapping back/forth

Model Conversion Pattern

Understanding this under ActiveFedora

Behavior classes are included into the ActiveFedora::Base classes that create the models for Hyrax.

For example,

- The Collection model includes Hyrax::CollectionBehavior
- Which includes Hydra::Works::CollectionBehavior
- Which includes PCDM::CollectionBehavior
- Which includes PCDM::PcdmBehavior

There are similar include chains for Works and Filesets.

There are relationships and methods defined in these include modules. They become the methods of the ActiveFedora::Base model classes.

Making this Work with Valkyrie

We want the Valkyrie Resource classes for each model to respond to the same methods. We also want these relationships and methods to 'play nicely' in the ActiveFedora environment and in the Valkyrie environment throughout the transition process.

This will be a multi-step process:

Step 1: Create equivalent relationships and methods in parallel modules that can be included in Valkyrie Resource classes

For example,

- The Valkyrie Resource model generated for Collection includes Wings::CollectionBehavior
- Which includes Wings::Hydra::Works::CollectionBehavior
- Which includes Wings::Pcdm::CollectionBehavior
- Which includes Wings::Pcdm::PcdmBehavior

All methods defined in those modules receive a parameter `valkyrie:` which is false by default.

When `valkyrie: false`

- passed in model objects will be ActiveFedora::Base objects
- passed in ids will be ActiveFedora ids
- returned model objects will be ActiveFedora::Base objects
- returned ids will be ActiveFedora ids

When `valkyrie: true`

- passed in model objects will be Valkyrie::Resource objects
- passed in ids will be Valkyrie ids
- returned model objects will be Valkyrie::Resource objects
- returned ids will be Valkyrie ids

The method may operate on the Valkyrie resource if it can OR it may use the original ActiveFedora version of the method. The choice is made on how difficult the full conversion to Valkyrie is for that method. For example, the replacement for Collection #add_member_object uses the ActiveFedora method since it has additional processing to check for multi-membership that is not ready for support in Valkyrie yet.

Step 2 and beyond: Refactor the converted methods getting closer and closer to do all the processing using Valkyrie resources and ids and processing.

Final step: Move the parallel file to the primary Hyrax code.

By using the same names and structure, the module files can be easily moved to the original file locations in hyrax/app/models. The only required change will be to change the first part of the module path from Wings to Hyrax.

NOTE: There is an outstanding question about whether PCDM and Works gems will be converted to Valkyrie. Those modules may have a longer life in the wings directory of Hyrax until a decision is made about how/whether to transform this into Valkyrie versions.

File Conversion Pattern

Understanding this under ActiveFedora

This is an oversimplification to get at the
• upload a file via AF
• file is available from af_object via files method which returns an association container (similar to an Array) of File objects that each hold a URI to the file content

Valkyrie File Handling

• create a file and save using storage adapter
• id for new file is saved in `file_ids` on resource

Fedora File Handling

(Probably won't get this quite right. Most of it happens down in AF)

Implications for Transformer

In the short term, can the files live where AF puts them. We could create a `File` resource with an alt-id that is the URI. That would give us a place to store the technical metadata that is stored with each AF file. And we can use existing transformation process with perhaps some minor adjustments to maintain the relationship between the FileSet and Files.

Pattern for Jobs Receiving Parameter that may be an AF Object or a Valkyrie Resource

Existing Job Example: VisibilityCopyJob

- #perform method receives a parameter that may be an ActiveFedora object (e.g. Hydra::Works::Collection | Work | FileSet or Hydra::PCDM::File) OR the Valkyrie::Resource equivalent
- VisibilityCopyJob calls a service (i.e., Hyrax::VisibilityPropagator) that determines which type of object it is working with and calls the appropriate service for the type of object
    - For AF object, calls the service (i.e., FileSetVisibilityPropagator) processing visibility through ActiveFedora
    - For valkyrie resource, calls the service (i.e., ResourceVisibilityPropagator) processing visibility for a Valkyrie resource

Splitting into 3 services is the approach to use if the processes for AF and Valkyrie objects are very different. If the process is mostly the same, then it is not necessary to implement 3 services for a single job. One service may be adequate. If the process is simple, it may be appropriate to do all the work in the job's #perform method.

See https://github.com/samvera/hyrax/pull/4123/files as an example that does not split into separate services. This PR updates many related and very similar jobs.

Pattern for use_valkyrie parameter/instance variable

To use the **`use_valkyrie`** pattern...

```
attr_reader :use_valkyrie
def initializer( _existing_args_, use_valkyrie: Hyrax.config.use_valkyrie?)
  @use_valkyrie = use_valkyrie
  # rest is the same as it was before applying the pattern
end

# no other method receives the use_valkyrie parameter, but instead calls the
use_valkyrie attr_reader method to get the value
```

This pattern cannot generally be used for class methods. For example, the query service has several class methods that accept `**use_valkyrie:**` as a parameter to determine if the result should return a valkyrie resource or an AF object. The preferred value is on a case by case usage of the caller.

The exception to the class methods pattern is solr_service where class methods are delegated to instance methods. Whether or not to use_valkyrie is determined by `**`Hyrax.config.query_index_from_valkyrie`**` and is set as the default on the initializer instead of defaulting to false.

Pattern for overriding custom queries

Hyrax version of custom query

The default hyrax implementation of the custom query should not have any adapter specific calls. It should only use standard valkyrie query service calls.

- Write standard valkyrie version of custom queries at `app/services/hyrax/custom_queries/your_custom_query_name.rb`.

NOTE: There are several `find_*` custom queries that provide a pattern for how to write a custom query.
Reference: [Valkyrie documentation for custom queries]

- Register the custom query in `lib/wings.rb` by updating the `custom_queries` array
- Write a test for the standard valkyrie custom query at `spec/services/hyrax/custom_queries/your_custom_query_name_spec.rb`

Wings override of custom query

You only need to write this version of the custom query if the custom query requires wings adapter specific calls. If wings needs to override the custom query...

- Write the override of the custom query in `lib/wings/services/custom_queries/your_custom_query.rb`.
- Register the override custom query in `lib/wings.rb` by replacing the Hyrax version with the Wings version. For example…

```
custom_queries =
  [...
```

```
      Wings::CustomQueries::FindAccessControl, # override
Hyrax::CustomQueries::FindAccessControl
     ...]
```

- Write a test for your custom query override at `spec/wings/services/custom_queries/your_custom_query_name_spec.rb`

Actor Stack Conversion Pattern

A brief description of “The Actor Stack”

The actor stack starts with `OptimisticLockValidator` and ends with `Terminator` and has a configurable number of actors in between. These actors were built, originally, to operate in the context of an ActiveFedora based work (and associated models, like FileSets).

Very near the end of the actor stack is a somewhat special actor called the `ModelActor`. The model actor determines what type of work it is operating on, for instance a `GenericWork`, and will initialize an instance of an actor expected to be in the system like `Hyrax::Actors::GenericWorkActor`. This “work actor” is a place for a developer to do something special/unique prior to and during the persistence of the works metadata.

A “work actor” inherits from a class `BaseActor` which defines the basic behavior of a works persistence strategies in the system. Methods like `create`, `update`, and `destroy` intend to make final preparations of metadata and relationships prior to persisting, updating, and deleting the work in the system. The `BaseActor#save` method is what eventually causes the work to be saved during creation and updating, while `BaseActor#destroy` deletes the work.

How does Valkyrie fit in?

The `BaseActor#save` method is called by way of the “work actor” being processed, which causes a work to be created or updated. Swapping the original ActiveFedora `work.save` with a Valkyrie equivalent will persist the work in the Fedora backend that Wings is configured to use. Part of the implementation performing this function is shown below:

```
def save
   adapter = Hyrax.config.valkyrie_metadata_adapter
   resource = adapter.persister.save(resource: env.curation_concern.valkyrie_resource)
   ...
```

What effect does this have on Hyrax?

Currently, the goal would be to use Wings to transform a Valkyrie Resource into an ActiveFedora model to be processed by the ActorStack. Throughout the actor stack, the actors do the work intended for each step, and eventually use the “work actor” to transform the ActiveFedora model back into a ValkyrieResource to be persisted into the backend.