Hyrax Implementation Recommendations

Potential Implementation Paths

1. Develop a “Hyrax” adapter which stores everything in Fedora/Solr the same way as AF does with Hyrax, then update Hyrax’s controllers to use Valkyrie’s API. (Swap out ActiveFedora - no data migration)
    a. Keep data - write an adapter to talk to data that exists. Don’t use AF.
    b. Pros
        i. Code work only - no data migrations.
        ii. Don’t have to maintain AF - just Valkyrie.
        iii. Highest chance for performance increase while using Fedora.
            1. Raw Fedora adapter has shown good performance so far.
            2. Can do solr updates in bulk at the end of a set of inserts, speeding that up.
    c. Cons
        i. Might be hard.
            1. Complex situations include indirect containers for membership, cleaning up relationships, direct containers for files, a sub-node for membership, etc.
        ii. May not be able to take into account hacks on top of AF some implementations have used for things like deeply nested data. These folks may have to write their own adapters.
        iii. Risk of adapter becoming bloated.
2. Change Hyrax controllers to use Valkyrie API, use the ActiveFedora adapter (and make it so the AF resource is configurable.)
    a. Keep data - use AF adapter to talk to data that exists. Valkyrie in front of AF.
    b. Pros
        i. Might not be much work to implement.
        ii. Code work only - no data migrations.
    c. Cons
        i. Things will probably be slower. It’s another abstraction.
        ii. If something breaks, it’s likely to be hard to track down.
        iii. Now we have to maintain Valkyrie & ActiveFedora as a community.
3. Switch controllers in Hyrax to use Valkyrie, then choose an adapter to use. Write a migration script to go from old data model to what the current Fedora adapter does.
    a. Migrate data
    b. Pros
        i. Don't have to maintain AF - just use Valkyrie
        ii. Good chance of performance increase - just choose a fast adapter to migrate to.
        iii. Simple. The adapter exists.
    c. Cons
        i. Requires data migration
            1. Migration may be different for each institution depending on how/what kind of data they store.
            2. Who writes the script?
    d. Questions
        i. Are we happy with what the data model that exists in Hyrax now?
        ii. Current Raw Fedora adapter doesn’t do things like (because they’re unnecessary for the application to work):
            1. Indirect Containers
            2. WebACLs
            3. hasMember predicates
4. Write some sort of backwards compatible layer - models that respond to #save but call a persister, form objects which fall back to ChangeSets, etc.
    a. Pros
        i. No code OR data migration
        ii. You could still switch metadata adapters.
    b. Cons
        i. Deep magic. This is at least 2 more layers of abstraction.
        ii. How do you unwind this after you migrate?
        iii. What happens when things break?
        iv. All or nothing.
        v. One of the goals of the data mapper group was to make the stack simpler - this is more complex.

Hyrax Features Which Might Be Changed

1. Actors
    a. The Valkyrie MVP app used a “ChangeSetPersister” pattern rather than Actors to solve the problem of callbacks and things like separating out ingesting of files. We’ve been happy with this, but Hyrax currently uses a set of actors.
    b. Jobs would likely have to change in order to accept a ChangeSetPersister or something in order to know where to store files /metadata.
    c. Implementation Options:
        i. Actors could take a ChangeSetPersister and use them.
            1. Code migration probably relatively easy.

        ii. Actors could be used instead of ChangeSetPersisters.
            1. Migration easiest (probably.) You'd have to register metadata adapters and storage adapters to use.
        iii. Actors could go away.
            1. Likely a significant code migration.
            2. (Figgy does this.)
2. Derivative Generation
    a. The Valkyrie MVP app stores the File ID of the derivative in the resource, so it can be stored wherever you like and the logic doesn't have to change.
    b. Hyrax just has code to go from ID -> spot on disk where derivative should be, and it goes and fetches it.
    c. The least amount of code change would be to stick with the Hyrax method.
3. Characterization
    a. Valkyrie provides a CharacterizationService for registering characterization methods. A FITS service will have to be written, or the CharacterizationService abstraction ignored.
4. LDP in PCDM
    a. The Fedora adapter doesn't do indirect containers for membership. If this is important, it will have to be implemented.
5. RDF Generation
    a. Valkyrie Resources have no concept of predicates or what their RDF serialization would be. They're an abstract model. The RDF serialization logic in Hyrax will have to map a context on top of it, rather than just dumping the graph from Fedora.
    b. Figgy implements this here: https://github.com/pulibrary/figgy/blob/master/app/models/concerns/linked_data.rb
6. Form Objects
    a. Valkyrie uses Reform to power its ChangeSets, which get fed into things like simple_form. Hyrax will either have to migrate to that, or find some sort of intermediary adapter to use.

Hyrax Features Which May Be Tough

1. Workflows.
    a. The Data Mapper Working Group didn't attempt to implement Sipity. However, the code for Sipity only requires GlobalID to be implemented for models. This is reasonably possible, although you'd have to store which metadata adapter to pull the object from as part of the GlobalID.
2. NOIDs.
    a. Ticket: https://github.com/samvera-labs/valkyrie/issues/207
    b. The Data Mapper Working Group didn't get this implemented during its sprints. However, we feel that a special "alternate_identifier" property could be added along with a query which would satisfy this use case.
    c. The NOID implementation that currently exists in Hyrax assumes the application can control the ID which gets saved, but some backends prefer to generate their own ID (like Postgres.)
3. Custom Queries
    a. Each query that is added to Valkyrie requires reserving a special property on models for a certain use case. We don't want to do this for every possible property, but there are times when an application will need a special query and will have to add one.
    b. We provided a mechanism for doing this here: https://github.com/samvera-labs/valkyrie/pull/245
    c. The mechanism hasn't been extensively tested to see how it feels to use.
4. WebACLs.
    a. Hyrax uses Hydra::AccessControls to store WebACLs for permissions, which are persisted in Fedora. The Valkyrie Fedora adapter doesn't do this, currently.
    b. Options:
        i. Bake it into the Fedora adapter. When it gets `read_groups`, `read_users`, `edit_users`, `edit_groups`, it persists them as WebACLs in Fedora. When someone asks for the resource, it performs all the queries to populate those fields on the model.
        ii. Treat permissions as special, and allow a way to provide a permissions strategy which decides how to serialize /unserialize the fields above for every persister. One would query, one would write to the object itself.
        iii. Treat ACLs & permissions as application logic. Just store the acl_id like other relationships, provide an indexer which can travel that relationship and index appropriately.
    c. Option #3 would result in the least amount of code in Valkyrie, and we'd recommend trying it to see if it will work. If not, either of the other options will work.

Potential New Features

1. Persist to Fedora async
    a. Since you're able to define the persister to use, and all persisters use the same interface, one could persist to a faster backend first and then have a job persist that resource to Fedora in the background - if it makes sense to do so.
2. Run specs using an in-memory adapter