DAM API Design Notes

The following hopefully highlights some of the **design choices and intentions** that went into the `info.magnolia.dam.api` packages (at the time of writing, for version 2.0 - this is currently in the `magnolia-dam-2.0-split-work-in-progress` branch, which will replace master for 2.0.), explains why /how the interfaces were designed the way they were, and note some points that might be tempting but should ideally not be conceded to. Below are also some points which are still **under consideration** and/or could change:

General

- Main goal: clean, clear, easy-to-use and non-redundant API. It should both be clear how to use it (naming and location are important) as well as how to implement it.
- Everything returns Iterators: efficiency and laziness prevail. With Guava's `FluentIterable` or `Iterables`, one can still do powerful filtering and transformations.
- We are exposing `MediaType` in several places, for instance when we need to retrieve objects such as `AssetProviders` or `Renderers`, based on which mime type they support. This allows to do wildcard selections based on Internet Media Type queries (RFC-2045, RFC-2046), such as "give me providers that support images" (i.e `AssetProviderRegistry.getProvidersFor(MediaType.parse("image/*"))`
- The `com.google.common.net.MediaType` class is used and exposed in several places. We decided to use it as-is because
    - It's easy to convert to/from String, so it should integrate well with Node2Bean (MAGNOLIA-5686 - Getting issue details... STATUS)
    - It has no external dependency
    - Guava can be seen, essentially, as an "extension" to the JDK libraries, much like the commons-* libraries.
    - Alternatives were:
        - use Tika's `org.apache.tika.mime.MediaType`: that has a lot of dependencies, and we also depend on Tika through other dependencies (Jackrabbit, Lucene, ...). It's not as well designed and clean; there seems to be leaks between Tika's MediaType, MimeType and other classes
        - roll out our own (by getting inspired by Guava's) - while tempting, that'd mean also copying a whole lot of tests, and... well, there's really little point in doing that.
    - We will write our own "LibTest" class, which will ensure `MediaType` behaves as expected.
- Conversely, Asset exposes `String getMimeType();` because that is the **exact** mime type of the given asset.

AssetProvider

- `AssetProvider` has "capabilities" (`AssetProviderCapability` enum). Some implementations will not support all features. This can be used to drive a UI to enable/disable certain actions or UI elements. Implementations should still throw exceptions when those unsupported operations are called. (although this might not be consistently the case, for example when aggregating search results from different providers.)
- `ItemKey` is a specific type (composed of provider id and asset id) rather than a "magic string"; it is passed to most methods, rather than the `assetID` string; this allows for aggregation/delegation in specialized providers.
- Write operations are currently out of scope. They could become part of the Dam API. Suggestions:
    - Add capabilities in `AssetProviderCapability`
    - Add another sub interface for these - thus client code could safely query the provider:
        - do you support write operation ?
        - yes ok cool, let me cast you.
- The above also goes for say, `PathAwareAssetProvider`, or `JcrAssetProvider`, even. If the client code knows for sure what the provider is, it can cast and call additional operations that are not meant to be generalized into the `AssetProvider` API (for instance, I was even reluctant to have any path-based methods at all)
- To extend on the above, some providers won't be path-aware at all. Most potential providers I can think of have the notion of "folders" or bags of assets of sort, but they're not necessarily a path. Some assets might show up in several such "folders" or bags. Think of Flickr's albums/streams /groups, or Imgur albums.
- Likewise, configuration should not leak into the `AssetProvider` interface. Client code will and should never need to get/set stuff on the service. It needs to query it. For example, while (some) AssetProvider might have a configured list of supported media types, client code will just ask "do you support image/png?". The configurability, which is essentially a node2bean implementation detail, should thus be in the implementation. As a convenience, we provide an `AbstractAssetProvider` which allows some of these things to be configured. Should a provider *not* require such configuration, it could override those methods, or more elegantly, just implement the interface. Should this become a common pattern, we could obviously split `AbstractAssetProvider` in `AbstractAssetProvider` and `AbstractConfigurableAssetProvider`. Typically, a `ro`

otPath property will only be exposed on the `JcrAssetProvider` (one might think it's a good idea to introduced a `PathAwareConfigurableAssetProvider`... which is true to a certain extent, but will just eventually lead to leaks, spaghetti and other maintenance issue, for the sole benefit of avoiding 2 or 3 getter/setter pairs)
- `AssetProviderRegistry` follows the same pattern. Its configurability is left to the implementation. (although it currently derogates from this rule by exposing a getAllProviders method - because this could be useful in a UI too)
- Queries & search: `new AssetQuery.Builder().withXXX()` - is a builder API. Pass the results of `.build()` to `AssetProvider.list()`. If further refinements to the results are needed, we recommend filtering the results with Guava's `FluentIterable` or `Iterables`.

Asset, Folder, Item

- Some of `Item/Folder/Asset` is mimicking the JCR API, but there are some subtle differences which are hopefully highlighted in the Javadoc.
- `Item/Folder/Asset` are meant to be implemented "lazily". Implementations will typically keep a reference to their provider, and the JCR implementation for example, should simply keep a reference to the corresponding `Node` instance, and delegating to it "on demand". Contrary to the 1.x version of the DAM module, a `getTitle()` method on `JcrAsset` should just delegate to `node.getProperty("title")` rather than eagerly load all the properties and act like a POJO later on.
  - This makes the implementation of these interfaces about 183% simpler and leaner.
- Removed `Object Asset.getCustomProperty(String propertyName)` method. There is no evidence this was used or needed. Will consider re-adding if needed, but it feels like this belongs to either MetaData classes, or specific implementations of `Asset`. Nothing prevents client code from casting to the specific implementation if needed. If this needs to be re-added somehow, think about the implications for `DisconnectedAsset` (see below).

Still under consideration

The following points are still **under consideration** and/or could change:

- `AssetProvider` should be queried - "can you handle this ItemKey?", rather than “forcing” the AssetProvider to handle the key, by comparing it with its providerID
- Queries: client code could do `assetProvider.newQuery()` - this would return Query"Builder" - and then call `execute()` on this (which would probably internally callback the provider)
- Queries: adding arbitrary `Predicate` (Guava and Commons) to a query might be added as a feature to QueryBuilder so that client code doesn't have to do it. OTOH, all features on the AssetQuery currently allow for (potential) optimizations by the AssetProvider, whereas arbitrary predicates would most likely mean those have to be done after the "remote" query returns. Perhaps it's wiser to keep those outside AssetProvider to avoid misunderstandings as to what criteria is used where.
- `DelegatingAssetProvider`, `AggregatingAssetProvider` - could probably be implemented thanks to the fact we pass `AssetKey` (almost) everywhere. Currently not implemented to avoid the questions of what to do with unsupported features on a single provider (ignore or reject), etc. Also, while an `AggregatingAssetProvider` sounds tempting to do federated search, current UI sketches show that we'd anyway group the results by provider (or some notion that's close to that). Letting the UI (or some intermediary) call each provider on independently would allow for asynchronous display of results (show up the JCR results while the slower CMIS is still loading)
- If the "laziness" of `Item/Folder/Asset` mentioned was to cause issues (which is perhaps going to be the case with templating etc), I would recommend the following:
  - implement a `DisconnectedAsset` class (or `AssetDTO`, or...) which
  - eagerly copies the properties of a given Asset into its fields,
  - refuses to provide an `InputStream` (in 1.x, the stream was **always** opened, and 99% never closed)
  - think a little longer about what to do if we also need this for folders 😑
    - perhaps a more restricted interface will be needed on Folder, and/or some methods moved out of Folder back to the AssetProvider.
  - ditto for MetaData. Perhaps the "copier" will need to take a list of MetaData that needs to be "copied" with the asset.

STILL TODOcument

Link api/integration

What exactly is a Link (a bunch of methods return String link and aren't exactly specifying **what** they return)

Exceptions: if Folder.getItem() throws AssetNotFoundException, we need to be consistent and add this to AssetProvider.get* as well. Currently leaning toward RuntimeExceptions anyway.

Metadata: how/why. Custom metadata in dialogs is currently "supported" via getCustomProperty()

Metadata: always lowercase

Metadata: expose getName() (see below)

AssetRenderer : to is a MediaType instance and no a string because ... (use-case that comes to mind is image/*, cause we don't care what the output format is)

dam-api is NOT a magnolia module. Dam-core is, and is called "dam" (helps transition?)

AssetFilter.includeDeleted == removed! not used. also very unlikely to be useful in a query.

Asset and Rendition 's mimeType return a String (because it's never going to be a "range"), but could return a MediaType instance for consistency/convenience. Either way, the string could also be "image/*", nothing prevents that so far, so ...

Using inner static classes for Exception (especially RuntimeExceptions) to avoid polluting the package (so every case can throw its own specific exception with its own specific constructor, without having to build String messages all over the place - ideally they shouldn't even be instantiable outside the implementations of the interfaces that declare them but i haven't figure that out yet)

Introduce a parent exception class ? (also runtime?)

```
+ // TODO I'm not keen on having this, as it would allow implementations to actually CHANGE their name.
+    // Force using class name as the identified ?
+    // TODO think about requirement of extensibility - plug in custom metadata suppliers
+    //
+   // String getName();
```

```
import com.google.common.base.CaseFormat;
public abstract class AbstractAssetMetadata implements AssetMetadata {
    //@Override
    public String getName() {
        return CaseFormat.UPPER_CAMEL.to(CaseFormat.LOWER_UNDERSCORE, this.getClass().getSimpleName());
    }
}


        assertEquals("my_meta_thing", new MyMetaThing().getName());
```

Write operations or CRUD thoughts

- See mention of write operations above
- We need to take a few more operations into account
    - BREAD for "Browse, Read, Edit, Add, Delete" or CRUDL (L for Listing)
    - IMO, for such a hierarchical system, Move is not Update.
    - Version (some providers might have a `versionOnUpdate` configuration item, but we maybe want a "manual", "create version now" type of operation
    - MADVERB 😛
- **BUT: do we really need this ?**
    - It may seem like a no-brainer at first sight ("be consistent"), but maybe we can reframe the need for an API. What do we really need the API for writing assets ?
        - *Browsing* jcr-based assets in the same way we browse external assets (the app)
        - *Use or reference* assets from any of source in a unified manner (the read api, the choose dialog, the templating functions)
        - I question the need for a unified API for *creating* assets because... if those assets are external, they presumably pre-exist Magnolia and/or are managed by different entities (software, people).
            - Would you upload a Youtube video through Magnolia, or through Youtube's very specialized UI ?
            - Would you upload a document through Magnolia's FileServer-AssetProvider, or by dragging and dropping it on your fileserver in Finder.app ?
            - I'm sure there are good cases / use-cases for this. But I only see two clients of such a writable-dam-api: the DAM App, and REST services.
                - I even see less of a point to upload-to-external-dam-through-Magnolia-REST. Presumably, your external DAM has a good reason to be in the system, and one of those is probably because it integrates well with whatever you'd be using the rest api instead ?
                - This leaves us with a few external DAMs that do need to have write-ops in Magnolia:
                    - Is the dam app really the best UI for those ? Maybe the external dam provides its own UI, or at least a paradigm where a tree or a flat list don't *really* match what we're looking for (some DAMs are better suited to have a search-only or tag-based browsing interface)
                    - What is the real benefit (time/cost) of generalizing write operations in DAM API *AND* DAM APP vs having a specialized (sub-)app for the cases where the external DAM does need write operations ?
- If the answer to the above is a resounding *yes*, then here are some additional thoughts/proposals for a WritableAssetProvider interface (all names just proposals as well)

- `AssetBuilder newAssset(Folder parent, String name).with(File/Stream/Path/URL).otherBuilderMethods (...).save();`.Rationale here is a) we can't just pass any Asset instance (not all properties can be set arbitrarily) b) Asset provides an InputStream for the binary, but we'll presumably need a different input to "upload" it.
- `AssetModifier update(Asset assetToModify).changeFoo(String s).changeBar(..).save();` The rationale here is that a) not all readable properties can be modified (some are auto-generated, or inferred from the binary, for example) b) we need to keep track of what is getting changed
- `void move(Asset asset, Folder newParent);`
- `void version(Asset, String label);`
- `void delete(Asset asset);`