Data Butler - Current Design

What is the Data Butler

- Manages repositories of datasets
- Finds datasets by scientifically-meaningful key/value pairs
- Can automatically "rendezvous" one dataset with another based on key values
    - Example: calibrations linked by time
- Retrieves datasets as in-memory objects
- Persists in-memory objects to datasets
- Implemented in Python; no access from C++

Definitions

Repository

- Collection of datasets
- Configuration for accessing datasets
- Metadata databases for finding datasets
- Version (e.g. as of particular time)

Dataset

- The persisted form of an in-memory object
- Can be a single item, a composite, or a collection
- Examples: `int`/`long`, `PropertySet`, `ExposureF`, `WCS`, `PSF`, `set`/`list`/`dict`

Persistable class

- A Python class (often SWIGged from C++) that can be persisted and retrieved
- Must provide methods for doing persistence and retrieval

Dataset type

- A label given to a group of datasets reflecting their meaning or usage
- Used by convention by Tasks for their inputs and outputs
- Examples: `calexp`, `src`, `icSrc`

Dataset class

- A labeled set of basic access characteristics serving as the basis for a group of dataset types
- Used to define new dataset types

Storage

- A mechanism for reading/writing a dataset to/from an in-memory object
- Examples: `FitsStorage`, `SqlStorage`

Transport

- A mechanism for providing access to data
- Examples: `file:`, `http:`, `sqlite:`, `mysql:`

`DataId`

- A dictionary of key/value pairs

`DataRef`

- A `DataId` packaged with a `Butler` for access to datasets
- Can be used with multiple dataset types (if the keys are appropriate)

`DataRefSet`

- Logically, a set of `DataRefs`
- May be implemented as an iterator/generator
- Based on an input dataset type, but `DataRefs` can be used with other dataset types

- All `DataRef`s point to an existing input dataset at time of generation

`Mapper`

- Not used by application code; only used via `Butler`
- Driven by per-repository configuration
- Camera-specific subclasses recorded in repository configuration
- Obtains a location template based on dataset type
    - Inherits from dataset class
    - Includes URL path with transport, storage method, optionally Python type
    - Includes filesystem locations and database tables/queries
    - Read-only and write-only types
- Expands an input `DataId` with additional key/value pairs (fixed and/or as-needed) needed to expand location template
    - Queries registry databases in input repositories as needed
    - Globs in filesystem if needed
- Expands location template into a `ButlerLocation`
- Optionally can provide methods for standardizing (post-processing) retrieved data
- Can be used to bidirectionally map `DataId`s to numeric identifiers
    - By treating numeric identifier as a dataset or as a single `DataId` key's value
    - Uses special `IdStorage`
- Provides utilities for subclasses
    - Maintain templates for dataset types in repository configuration
    - Look up key/value pairs using equality or range joins in registry databases
    - Glob for key/value pairs in filesystem
    - Record metadata of new datasets in registries
    - Maintain registry of registries

`ButlerLocation`

- All location information needed for a Storage
- May include:
    - Expanded path template(s)
    - Python object class name
    - Storage class name
    - `DataId`
    - Additional key/value pairs

`Butler`

- Obtains mapper class name from repository
- Calls `Mapper` to translate `DataId` into location
- Calls appropriate Storage to retrieve or persist data
- Repository identified by root (URL) path
- Zero or more read-only input repositories
    - Input repositories identified by role
    - Role can be used in output repository configuration
- One output repository
    - Input repositories recorded in output repository with roles
    - Initial output repository configuration derived from camera-specific defaults and input repository overrides
    - User can provide overrides for output repository configuration
    - Tasks can add to output repository configuration
- Calibration (and other?) repositories permitted
- Input and output repositories
    - Provides utility for searching read-only parent repositories

Butler Interface

- `__init__(outputRepo, inputRepos=None)`
    - `outputRepo` is a repository URL (string)
    - `inputRepos` is a map from role name (string) to repository URL
- `get(self, datasetType, dataId={}, **kwArgs)`
    - returns object retrieved using `dataId` with keyword argument overrides
- `put(self, obj, datasetType, dataId={}, **kwArgs)`
    - persists `obj` using `dataId` with keyword argument overrides
    - The `Butler` (or actually either its `Mapper` or a `Storage`) is allowed to notice that the identical `obj` has been persisted before and not persist it again
        - This also applies to components of composite `obj`s, which can be persisted as a reference to the original
- `getKeys(self, datasetType=None)`
    - returns list of `DataId` keys appropriate for `datasetType` or all keys known for output repository
- `getDatasetTypes(self)`
    - returns list of known dataset types
- `createDatasetType(self, datasetType, datasetClass, pathTemplate, **kwArgs)`
    - creates a new `datasetType` based on the `datasetClass` using the provided path template and keyword arguments
- `getRefSet(self, datasetType, partialDataId={}, **kwArgs)`

    - returns `DataRefSet` enumerating all existing datasets of `datasetType` using `partialDataId` with keyword argument overrides
- `defineAlias(alias, datasetType)`
    - Henceforth, any use of "`@alias`" with this `Butler` becomes equivalent to `datasetType`

What's new?

- `ButlerFactory` is gone.
- `getDatasetTypes` and `createDatasetType` are passed through from the `Mapper`.
- `subset` is renamed to `getRefSet` to be more descriptive; its functionality subsumes the old `queryMetadata`.
- `datasetExists` is gone, since `getRefSet` only returns datasets that exist.
- `level` arguments have been removed, as the concept turned out to be useless in practice.
- `put` can handle duplicates (for configurations and provenance or for sharing objects).
- Much-requested dataset type aliasing facility enables `Tasks` to handle, e.g., any "`src`"-like dataset.

Mapper Interface

Interface used by `Butler`:

- `__init__(self, repo)`
    - `repo` is an output repository
- `map(self, datasetType, dataId)`
    - returns the `ButlerLocation` corresponding to the `dataId`for the given `datasetType`
- `getKeys(self, datasetType)`
    - returns list of `DataId` keys appropriate for `datasetType` or all keys known for output repository
- `getDatasetTypes(self)`
    - returns list of known dataset types
- `createDatasetType(self, datasetType, datasetClass, **kwArgs)`
    - creates a new `datasetType` based on the `datasetClass` using the keyword arguments
- `listDatasets(self, datasetType, partialDataId={}, **kwArgs)`
    - returns or generates set of `DataId`s enumerating all existing datasets of `datasetType` using `partialDataId` with keyword argument overrides
- `canStandardize(self, datasetType)`
    - returns `True` if the `datasetType` can be standardized
- `standardize(self, obj, datasetType, dataId)`
    - returns the standardized version of `obj`, given its `datasetType` and `dataId`

Interface for subclasses:

- (TBWritten)

What's new?

- `createDatasetType` has been added.
- `listDatasets` replaces the old `queryMetadata`.
- `validate` was never used and is gone. (It was originally supposed to do something like `CameraMapper`'s `Mapping`'s `need()`.)
- Much lookup functionality is now intended to be performed by custom Storage classes, like `IdStorage`.