DataLad Catalog

Release 1.1.1+48.g821b12f

DataLad team

Apr 27, 2024

CONTENTS

Welcome to the user and technical documentation of DataLad Catalog, a DataLad extension that allows you to create a user-friendly data browser from structured metadata.

CHAPTER

ONE

ACKNOWLEDGEMENTS

This software was developed with support from the German Federal Ministry of Education and Research (BMBF 01GQ1905), the US National Science Foundation (NSF 1912266), and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grant SFB 1451 (431549029, INF project).

CHAPTER

TWO

DEMO

See our demo catalog, hosted via Netlify. This catalog was generated from the studyforrest dataset.

CHAPTER

THREE

INDEX

3.1 Overview

3.1.1 What is DataLad Catalog?

DataLad Catalog is a free and open source command line tool with a Python API that allows you to turn structured metadata into a user-friendly, browser-based data catalog.

It is an extension to, and dependent on, Datalad, and is interoperable with Datalad Metalad:

- **DataLad** is a distributed data management system that keeps track of your data, creates structure, ensures reproducibility, supports collaboration, and integrates with widely used data infrastructure.
- **DataLad MetaLad** extends and equips DataLad with a command suite for metadata handling. This includes traversing a full data tree (of arbitrarily large size) to conduct metadata extraction (using extractors for various data types), metadata aggregation, and also reporting.

By combining the functionality of these three tools, you can:

1. manage the full lifecycle of your data (applying version control and capturing provenance records along the way)
2. add and extract detailed metadata records about every single item in a multi-level dataset, and
3. convert the metadata into a user-friendly browser application that increases the findability and accessibility of your data.

As a bonus, these processes can be applied in a decentralized and collaborative way.

3.1.2 Why use DataLad Catalog?

Working collaboratively with large and distributed datasets poses particular challenges for FAIR data access, browsing, and usage.

- the **administrative burden of keeping track** of different versions of the data, who contributed what, where/how to gain access, and representing this information centrally and accessibly can be significant
- **data privacy regulations** might restrict data from being shared or accessed across multi-national sites
- **costs of centrally maintained infrastructure** for data hosting and web-portal type browsing could be prohibitive

These challenges impede the many possible gains obtainable from distributed data sharing and access. Decisions might even be made to forego FAIR principles in favour of saving time, effort and money, leading to the view that these efforts have seemingly contradicting outcomes.

DataLad Catalog helps counter this contradiction by focusing on interoperability with structured, linked, and machine-readable metadata.

Metadata about datasets, their file content, and their links to other datasets can be used to create abstract representations of datasets that are separate from the actual data content. This means that data content can be stored securely while metadata can be shared and operated on widely, thus improving decentralization and FAIRness.

By combining these features, DataLad Catalog can create a user-friendly catalog of your dataset and make it publicly available, complete with all additionally supplied metadata, while you maintain secured and permission-based access control over your actual file content. This catalog can itself be maintained and contributed to in a decentralized manner without compromising metadata integrity.

3.1.3 How does it work?

DataLad Catalog can receive commands to `create` a new catalog, `add` and `remove` metadata entries to/from an existing catalog, `serve` an existing catalog locally, and more. Metadata can be provided to DataLad Catalog from any number of arbitrary metadata sources, as an aggregated set or as individual items/objects. DataLad Catalog has a dedicated *Catalog Schema* (using the JSON Schema vocabulary) against which incoming metadata items are validated. This schema allows for standard metadata fields as one would expect for datasets of any kind (such as `name`, `doi`, `url`, `description`, `license`, `authors`, and more), as well as fields that support identification, versioning, dataset context and linkage, and file tree specification.

The process of generating a catalog, after metadata entry validation, involves:

1. aggregation of the provided metadata into the catalog filetree
2. generating the assets required to render the user interface in a browser

The output is a set of structured metadata files, as well as a Vue.js-based browser interface that understands how to render this metadata in the browser. What is left for the user is to host this content on their platform of choice and to serve it for the world to see.

For an example of the result, visit our demo catalog.

Note: A detailed description of these steps can be found in the *Pipeline Description*

3.2 Installation

You can install and run DataLad Catalog on all major operating systems by following the steps below in the command line.

3.2.1 Step 1 - Setup and activate a virtual environment

With your virtual environment manager of choice, create a virtual environment and ensure you have a recent version of Python installed. Then activate the environment.

With venv:

```
python -m venv my_catalog_env
source my_catalog_env/bin/activate
```

With miniconda:

```
conda create -n my_catalog_env python=3.11
conda activate my_catalog_env
```

3.2.2 Step 2 - Install via PyPI

```
pip install datalad-catalog
```

Congratulations! You have now installed DataLad Catalog!

3.2.3 Optional - Clone the repo and install the package

If you want to access the latest, unreleased version of the software or contribute to the code, access the repository via GitHub:

```
git clone https://github.com/datalad/datalad-catalog.git
cd datalad-catalog
pip install -e .
```

3.2.4 Dependencies

Because this is an extension to `datalad` and builds on metadata handling functionality, the installation process also installed datalad and datalad-metalad as dependencies, although these do not have to be used as the only sources of metadata for a catalog. In addition datalad-next is installed in order to use the latest improvements and patches to the `datalad` core package.

While the catalog generation process does not expect data to be structured as DataLad datasets, it can still be very useful to do so when building a full (meta)data management pipeline from raw data to catalog publishing. For complete instructions on how to install `datalad` and `git-annex`, please refer to the DataLad Handbook.

Similarly, the metadata input to `datalad-catalog` can come from any source as long as it conforms to the catalog schema. While the catalog does not expect metadata originating only from `datalad-metalad`'s extractors, this tool has advanced metadata handling capabilities that will integrate seamlessly with DataLad datasets and the catalog generation process.

In order to translate metadata extracted using `datalad-metalad` into the catalog schema, `datalad-catalog` provides translation modules that are dependent on jq.

3.3 Usage

DataLad Catalog can be used from the command line or with its Python API. You can access detailed usage information in the *Command Line Reference* or the *Python Module Reference* respectively.

The overall catalog generation process actually starts several steps before the involvement of `datalad-catalog`. Typical steps include:

1. curating data into datasets (a group of files in an hierarchical tree)
2. adding metadata to datasets and files (the process for this and the resulting metadata formats and content vary widely depending on domain, file types, data availability, and more)
3. extracting the metadata using an automated tool to output metadata items into a standardized and queryable set
4. translating the metadata into the catalog schema

These steps can follow any arbitrarily specified procedures and can use any arbitrarily specified tools to get the job done. Once they are completed, the `datalad-catalog` tool can be used for catalog generation and customization.

3.3.1 Create a catalog

To create a new catalog, start by running `datalad catalog-create`

```
datalad catalog-create --catalog /tmp/my-cat
```

This will create a catalog with the following structure:

- `artwork`: images that make the catalog pretty
- `assets`: mainly the JavaScript and CSS code that underlie the user interface of the catalog
- `metadata`: where metadata content for any datasets and files rendered by the catalog will be contained
- `schema`: which contains JSON documents laying out the schema that the specific catalog complies with
- `templates`: HTML template documents for rendering specific components
- `config.json`: the configuration file with rules for rendering and updating the catalog
- `index.html`: the main HTML content rendered in the browser

Note: The `--config-file` argument allows the catalog to be created with a custom configuration. If not specified, a default configuration is applied at the catalog level.

3.3.2 Add metadata

To add metadata to an existing catalog, run `datalad catalog-add`, specifying the (location of the) metadata to be added. DataLad Catalog accepts metadata in the form of:

- a path to a file containing JSON lines
- JSON lines from STDIN
- a JSON serialized string

where each line or record is a single, correctly formatted, JSON object. The correct format for the added metadata is specified by the *Catalog Schema*.

```
datalad catalog-add --catalog /tmp/my-cat --metadata path/to/metadata.jsonl
```

The `metadata` directory is now populated.

Note: The `--config-file` argument allows the specific dataset in the catalog to be created with a custom configuration. If not specified, the configuration at the dataset level will be inferred from the catalog level.

Metadata validation

To check if metadata is valid before adding it to a catalog, `datalad catalog-validate` can be run to check if the metadata conforms to the *Catalog Schema*.

```
datalad catalog-validate --catalog /tmp/my-cat --metadata path/to/metadata.jsonl
```

The metadata will then be validated against the schema version of the supplied catalog. If the `--catalog` argument is not provided, validation happens against the schema version contained in the installed `datalad-catalog` package.

Note: The validator runs internally whenever `datalad catalog-add` is called, so there is no need to run validation explicitly unless desired.

3.3.3 Set catalog properties

Properties of the catalog can be set via the `datalad catalog-set` command. For example, setting a "main" dataset is necessary in order to indicate which dataset will be shown on the catalog homepage. To set this homepage, run `datalad catalog-set home`, specifying the `dataset_id` and `dataset_version`:

```
datalad catalog-set --catalog /tmp/my-cat --dataset_id abcd --dataset_version 1234 home
```

Note: Tip

It could be a good idea to populate the catalog with datasets that are all linked as subdatasets from the main dataset displayed on the home page, since this would allow users to navigate to all other datasets from the main page. This linkage is done implicitly if the catalog home page is a DataLad superdataset with nested subdatasets.

3.3.4 View the catalog

To serve the content of a catalog via a local HTTP server for viewing or testing, run `datalad catalog-serve`.

```
datalad catalog-serve --catalog /tmp/my-cat
```

Once the content is served, the catalog can be viewed by visiting the localhost URL.

3.3.5 Update

Catalog content can be updated using the `add` or `remove` commands. To add content, simply re-run `datalad catalog-add`, providing the path to the new metadata.

```
datalad catalog-add --catalog /tmp/my-cat --metadata path/to/new/metadata.jsonl
```

If a newly added dataset or version of a dataset was added incorrectly, `datalad catalog-remove` can be used to get rid of the incorrect addition.

```
datalad catalog-remove --dataset_id abcd --dataset_version 1234 --reckless
```

Note: A standard `catalog-remove` call without the `--reckless` flag will provide a warning and do nothing else, for safety. Remember to add the flag in order to remove the metadata.

3.3.6 Configure

A useful feature of the catalog process is to be able to configure certain properties according to your preferences. This is done with help of a config file (in either `JSON` or `YAML` format) and the `-F/--config-file` flag. A config file can be passed during catalog creation in order to set the config on the catalog level:

```
datalad catalog-create --catalog /tmp/my-custom-cat --config-file path/to/custom_config.
↪json
```

A config file can also be passed when adding metadata in order to set the config on the dataset-level:

```
datalad catalog-add --catalog /tmp/my-custom-cat --metadata path/to/metadata.jsonl --
↪config-file path/to/custom_dataset_config.json
```

In the latter case, the config will be set for all new dataset entries corresponding to metadata source objects in the metadata provided to the `catalog-add` operation.

If no config file is specified on the catalog level, a default config file is used. The catalog-level config also serves as the default config on the dataset level, which is used if no config file is specified via the `catalog-add` command.

Note: For detailed information on how to structure and use config files, please refer to the dedicated documentation in *Catalog Configuration*.

3.3.7 Get catalog properties

Properties of the catalog can be retrieved via the `datalad catalog-get` command. For example, the specifics of the catalog home page can be retrieved as follows:

```
datalad catalog-get --catalog /tmp/my-cat home
```

Or the metadata of a specific dataset contained in the catalog can be retrieved as follows:

```
datalad catalog-get --catalog /tmp/my-cat --dataset_id abcd --dataset_version 1234␣
↪metadata
```

3.3.8 Translate

`datalad-catalog` can translate a metadata item originating from a particular source structure, and extracted using `datalad-metalad`, into the catalog schema. Before translation from a specific source will work, an extractor-specific translator should be provided and exposed as an entry point (via a DataLad extension) as part of the `datalad.metadata.translators` group. Then, translate metadata as follows:

```
datalad catalog-translate --metadata path/to/extracted/metadata.jsonl
```

This command will output the translated objects as JSON lines to `stdout`, which can be saved to disk and later used, for example, for catalog entry generation.

3.3.9 Workflows

Several subprocesses need to be run in order to create a new catalog with multiple entries, or in order to update an existing catalog with new entries. These processes can include:

- tracking datasets that are intended to be entries in a catalog as subdatasets of a DataLad super-dataset
- extracting (and temporarily storing) metadata from the super- and subdatasets
- translating extracted metadata (and temporarily storing it)
- creating a catalog
- adding translated metadata to the catalog
- updating the catalog's superdataset (i.e. homepage) if the DataLad superdataset version changed

It is evident that these steps can become quite cumbersome and even resource intensive if run at scale. Therefore, in order to streamline these processes, to automate them as much as possible, and to shift the effort away from the user, `datalad-catalog` can run workflows for catalog generation and updates. It builds on top of the following functionality:

- *DataLad datasets* and nesting for maintaining a super-/subdataset hierarchy.
- `datalad-metalad`'s metadata extraction functionality
- `datalad-catalog`'s metadata translation functionality
- `datalad-catalog` for maintaining a catalog

workflow-new

To run a workflow from scratch on a dataset and all of its subdatasets:

```
datalad catalog-workflow --type new --catalog /tmp/my-cat --dataset path/to/superdataset␣
↪--extractor metalad_core
```

This workflow will:

1. Clone the super-dataset and all its first-level subdatasets
2. Create the catalog if it does not yet exists
3. Run dataset-level metadata extraction on the super- and subdatasets
4. Translate all extracted metadata to the catalog schema
5. Add the translated metadata as entries to the catalog

6. Set the catalog's home page to the *id* and *version* of the DataLad super-dataset.

workflow-update

To run a workflow for updating an existing catalog after registering a new subdataset to the superdataset which the catalog represents:

```
datalad catalog-workflow --type update --catalog /tmp/my-cat --dataset path/to/
↪superdataset --subdataset path/to/subdataset --extractor metalad_core
```

This workflow assumes:

- The subdataset has already been added as a submodule to the parent dataset
- The parent dataset already contains the subdataset commit

This workflow will:

1. Clone the super-dataset and new subdataset
2. Run dataset-level metadata extraction on the super-dataset and new subdataset
3. Translate all extracted metadata to the catalog schema
4. Add the translated metadata as entries to the catalog
5. Reset the catalog's home page to the latest *id* and *version* of the DataLad super-dataset.

3.4 Pipeline Description

The DataLad ecosystem provides a complete set of free and open source tools that, together, provide full control over dataset access and distribution, version control, provenance tracking, metadata addition, extraction, and aggregation, as well as catalog generation.

DataLad itself can be used for decentralised management of data as lightweight, portable and extensible representations. DataLad MetaLad can extract structured high- and low-level metadata and associate it with these datasets or with individual files. Then at the end of this workflow, DataLad Catalog can turn the structured metadata into a user-friendly data browser.

Importantly, DataLad Catalog can operate independently as well. Since it provides its own schema in a standard vocabulary, any metadata that conforms to this schema can be submitted to the tool in order to generate a catalog. Metadata items do not necessarily have to be derived from DataLad datasets, and the metadata extraction does not have to be conducted via DataLad MetaLad.

Even so, the provided set of tools can be particularly powerful when used together in a distributed (meta)data management pipeline. Below is an example for building a catalog using the full DataLad toolset; from data management, to metadata handling, to the end result of catalog generation.

3.4.1 An example end-to-end pipeline

Step 1 - Create/access a DataLad dataset

Our fundamental operational unit is a DataLad dataset. In order to generate a minimal catalog, we have to start with this unit. We do this either by cloning a DataLad dataset from a known location, or by creating a new dataset. See the DataLad Handbook for more information on working with DataLad datasets.

Clone:

```
datalad clone [dataset_location]
```

Create:

```
datalad create --force [dataset_location]
```

Step 2 - Add metadata

In order to extract arbitrary structured metadata from a DataLad dataset, this information first has to be added explicitly to the dataset. It can be added in your preferred location in the dataset tree. For example, here we add a `.studyminimeta.yaml` file to the root directory of the dataset:

```
cd mydataset
mv [path/to/studyminimeta.yaml] .
```

Once the dataset has been updated with metadata, it has to be saved:

```
datalad save -m "add metadata to mydataset"
```

Various metadata formats can be recognized by DataLad MetaLad's extraction process. See *Metadata and datalad-catalog* for an overview and DataLad Metalad's documentation for more detail.

Step 3 - Extract metadata

With Datalad MetaLad we can extract implicit and explicit metadata from our dataset. This can be done on the dataset as well as file level through the use of built-in or custom extractors. MetaLad provides several commands to streamline this process, especially for large datasets:

- `meta-add` adds metadata related to an element (dataset or file) to the metadata store
- `meta-dump` shows metadata stored in a local or remote dataset
- `meta-extract` runs an extractor (see below) on an existing dataset or file and emits the resulting metadata to stdout
- `meta-aggregate` combines metadata from a number of sub-datasets into the root dataset
- `meta-conduct` runs pipelines of extractors and adders on locally available datatasets/files, in order to automate metadata extraction and adding tasks

Below are example code snippets that can be run to extract metadata from the file and dataset level (respectively using the `metalad_core`, and both the `metalad_core` and `metalad_studyminimeta` extractors) and to subsequently write these metadata objects to disk in JSON format.

From dataset

Extract and add:

```
#!/bin/zsh
DATASET_PATH="path/to/mydataset"
PIPELINE_PATH="path/to/extract_dataset_pipeline.json"
datalad meta-conduct "$PIPELINE_PATH" \
    traverser.top_level_dir=$DATASET_PATH \
    traverser.item_type=dataset \
    traverser.traverse_sub_datasets=True \
    extractor1.extractor_type=dataset \
    extractor1.extractor_name=metalad_core \
    extractor2.extractor_type=dataset \
    extractor2.extractor_name=metalad_studyminimeta \
    adder.aggregate=True
```

where the pipeline in `path/to/extract_dataset_pipeline.json` looks like this:

```
{
  "provider": {
    "module": "datalad_metalad.pipeline.provider.datasettraverse",
    "class": "DatasetTraverser",
    "name": "traverser",
    "arguments": {}
  },
  "processors": [
    {
      "module": "datalad_metalad.pipeline.processor.extract",
      "class": "MetadataExtractor",
      "name": "extractor1",
      "arguments": {}
    },
    {
      "module": "datalad_metalad.pipeline.processor.extract",
      "class": "MetadataExtractor",
      "name": "extractor2",
      "arguments": {}
    },
    {
      "name": "adder",
      "module": "datalad_metalad.pipeline.processor.add",
      "class": "MetadataAdder",
      "arguments": {}
    }
  ]
}
```

Dump and write to disk:

```
#!/bin/zsh
DATASET_PATH="path/to/mydataset"
METADATA_OUT_PATH="path/to/dataset_metadata.json" # empty text file
datalad meta-dump -d "$DATASET_PATH" -r "*" > "$METADATA_OUT_PATH"
```

From files

Extract and write to disk:

```
#!/bin/zsh
DATASET_PATH="path/to/mydataset"
PIPELINE_PATH="path/to/extract_file_pipeline.json"
METADATA_OUT_PATH="path/to/file_metadata.json" # empty text file
# Add starting array bracket
echo "[" > "$METADATA_OUT_PATH"
# Extract file-level metadata, add comma
datalad -f json meta-conduct "$PIPELINE_PATH" \
    traverser.top_level_dir=$DATASET_PATH \
    traverser.item_type=file \
    traverser.traverse_sub_datasets=True \
    extractor.extractor_type=file \
    extractor.extractor_name=metalad_core \
    | jq '.["pipeline_element"]["result"]["metadata"][0]["metadata_record"]' \
    | jq -c . | sed 's/$/,/' >> "$METADATA_OUT_PATH"
# Remove last comma
sed -i '' '$ s/.$//' "$METADATA_OUT_PATH"
# Add closing array bracket
echo "]" >> "$METADATA_OUT_PATH"
```

where the pipeline in `path/to/extract_file_pipeline.json` looks like this:

```
{
  "provider": {
    "module": "datalad_metalad.pipeline.provider.datasettraverse",
    "class": "DatasetTraverser",
    "name": "traverser",
    "arguments": {}
  },
  "processors": [
    {
      "module": "datalad_metalad.pipeline.processor.extract",
      "class": "MetadataExtractor",
      "name": "extractor",
      "arguments": {}
    }
  ]
}
```

At the end of this process, you have two files with structured metadata that can eventually be provided to `datalad-catalog` in order to generate the catalog and its entries.

Step 4 - Translate the metadata

Before the extracted metadata can be provided to `datalad-catalog`, it needs to be in a format/structure that will validate successfully against the catalog schema. Extracted metadata will typically be structured according to whatever schema was specified by the extractor, and information in such a schema will have to be translated to the catalog schema. For this purpose, `datalad-catalog` provides a `catalog-translate` command together with dedicated translators for specific metadata extractors. See *Metadata and datalad-catalog* and the *Usage* instructions for more information.

To translate the extracted metadata, we do the following:

```
datalad catalog-translate -m [path/to/dataset_metadata.json] > [path/to/translated_
↪dataset_metadata.json]
datalad catalog-translate -m [path/to/file_metadata.json] > [path/to/translated_file_
↪metadata.json]
```

Step 5 - Run DataLad Catalog

Note: Detailed usage instructions for DataLad Catalog can be viewed in *Usage* and *Command Line Reference*.

The important subcommands for generating a catalog are:

- `catalog-create` creates a new catalog with the required assets, taking metadata as an optional input argument
- `catalog-add` adds dataset and/or file level metadata to an existing catalog

To create a catalog from the metadata we generated above, we can run the following:

```
#!/bin/zsh
TRANSLATED_DATASET_METADATA_OUT_PATH="path/to/translated_dataset_metadata.json"
TRANSLATED_FILE_METADATA_OUT_PATH="path/to/translated_file_metadata.json"
CATALOG_PATH="path/to/new/catalog"
datalad catalog-create -c "$CATALOG_PATH" -m "$TRANSLATED_DATASET_METADATA_OUT_PATH"
datalad catalog-add -c "$CATALOG_PATH" -m "$TRANSLATED_FILE_METADATA_OUT_PATH"
```

Step 6 - Next steps

Congratulations! You now have a catalog with multiple entries!

This catalog can be served locally (`datalad catalog-serve`) to view/test it, deployed to an open or/restricted cloud server in order to make it available to the public or colleagues/collaborators (e.g. via Netlify in the case of publicly available catalogs), and updated with new entries in future (with a `datalad catalog-add`).

Happy cataloging!

3.5 Metadata and `datalad-catalog`

The catalog is rendered from structured metadata generated by `datalad-catalog`. In this section, more information is provided about the nature of metadata (in general and in relation to the catalog) and the states that metadata generally pass through in order to end up as part of a catalog.

3.5.1 What is metadata?

Metadata describe the files in your dataset, as well as its overall content. Implicit metadata include basic descriptions of the data itself (such as the names, types, sizes, and relative locations of all files in your dataset), while explicit metadata items (such as a description of your dataset, its contributors and project specifications) can be added to your dataset as you see fit. MetaLad provides functionality for adding metadata items of arbitrary size, format, and amount, and does not impose restrictions in this regard.

Many standards exist for specifying and structuring metadata. Some examples include:

- **DataCite**: The DataCite Metadata Schema is a list of core metadata properties chosen for an accurate and consistent identification of a resource for citation and retrieval purposes, along with recommended use instructions.
- **XMP**: The Extensible Metadata Platform is an ISO standard for the creation, processing and interchange of standardized and custom metadata for digital documents and data sets. It also provides guidelines for embedding XMP information into popular image, video and document file formats, such as JPEG and PDF.
- **Frictionless Data**: The Frictionless Data Package is a container format for describing a coherent collection of data in a single 'package', providing the basis for convenient delivery, installation and management of datasets.

Standardized file formats may also contain format-specific information (such as bit rate and duration for audio files, or resolution and color mode for image files), while domain- standard files (such as Digital Imaging and Communications in Medicine, i.e. DICOM) also supply embedded or sidecar metadata.

Note: In order to create a user-friendly catalog, DataLad Catalog should receive structured metadata adhering to a specified *Catalog Schema* as input. This means that structured metadata first has to be sourced and then translated into the schema.

3.5.2 Metadata handling with MetaLad

Since `datalad-catalog` provides its own schema in a standard vocabulary, any metadata that conforms to this schema can be submitted to the tool in order to generate a catalog and its entries. Metadata items do not necessarily have to be derived from DataLad datasets, and the metadata extraction does not have to be conducted via `datalad-metalad`. However, `datalad-metalad` provides highly applicable functionality that simplifies the process of metadata handling for the purpose of generating structured outputs that could be used for catalog generation.

`datalad-metalad` has functionality to:

- `add` metadata of an arbitrary format to a DataLad dataset
- `dump` metadata that was previously added to a DataLad dataset
- `extract` metadata from files or datasets using format-specific extractors

as well as to run batch jobs with these and other methods. Find out more about MetaLad and its capabilities in the dedicated DataLad Handbook Chapter.

The benefit of using `datalad-metalad` in a catalog-generation workflow comes with the use of its extraction interface and custom extractors. An extractor is nothing other than something that understands a specific schema (or data structure) and can *extract* information from a file or dataset that adheres to said schema. For example, the `metalad_core`

extractor that ships with `datalad-metalad` can extracting implicit metadata, such as author information, dataset identifier/version, bytesize (for files), and more, from a DataLad dataset and its files. And the `metalad_studyminimeta` extractor extracts information from DataLad dataset containing a `.studyminimeta.yaml` file in its root directory. MetaLad ships with a variety of dataset- and file-level extractors, and so does a number of DataLad extensions including `datalad-catalog`. If an extractor for a specific metadata format is not available a custom extractor can be created and provided via a DataLad extension. If this sounds like something you need, please refer to the documentation on writing your own extractor.

An extractor will output its metadata, which has a structure specified by the dedicated extractor, inside a wrapper object provided by `datalad-metalad`. This means the top-level structure of all metadata extracted by `datalad-metalad` will be the same, while that of the property containing the actual extracted metadata will differ based on the extractor.

3.5.3 Metadata translation

As mentioned above, `datalad-catalog` provides its own schema in a standard vocabulary, and incoming metadata need to validate successfully against this schema. Since extracted metadata will typically be structured according to whatever schema was specified by the extractor, and information in such a schema will first have to be translated to the catalog schema before catalog entry generation can continue.

`datalad-catalog` provides a `catalog-translate` command through which custom translators can be created and used to translate MetaLad-extracted metadata into the catalog schema. The catalog ships with several translators (including ones for `metalad_core` and `metalad_studyminimeta`) and provides a base class that makes it straightforward to implement custom translators. Before translation from a specific source will work, the extractor-specific translator should be provided and exposed as an entry point (via a DataLad extension) as part of the `datalad.metadata.translators` group.

Then `datalad-catalog` will be able to find the correct translator automatically based on unique properties in a MetaLad-extracted metadata object. This is done by applying matching criteria that is specified by the translator, and running a `translate()` method if the match was successful.

3.5.4 The Catalog Schema

Finally, the result of the metadata extraction and translation workflow will be metadata that conforms to the catalog's own schema, which uses the vocabulary defined by JSON Schema (specifically draft 2020-12). Find out more about the *Catalog Schema*.

3.6 Catalog Schema

Metadata submitted to DataLad Catalog has to conform to its own schema, which uses the vocabulary defined by JSON Schema (specifically draft 2020-12).

Source files defining the catalog's schema can be found here:

- catalog
- dataset
- file
- authors
- metadata_sources

A rendering of the schema can be accessed at: https://datalad.github.io/datalad-catalog/display_schema.html

3.7 Catalog Configuration

A useful feature of the catalog process is to be able to configure certain properties according to your preferences. This is done with help of a config file (in either `JSON` or `YAML` format) and the `-F/--config-file` flag. A config file can be passed during catalog creation in order to set the config on the catalog level, or when adding metadata in order to set the config on the dataset-level.

As an example, `datalad-catalog`'s default config file can be viewed here.

3.7.1 Catalog-level configuration

Via the catalog-level config (provided during `catalog-create`) you can specify the following properties:

- the catalog name
- a path to a logo file to be used in the rendered catalog header
- the HEX color code to be used for links in the rendered catalog
- the HEX color code to be used when a cursor hovers over links in the rendered catalog
- default rules for rendering metadata on a dataset level (see detailed specification below)

The catalog-level configuration file will be located at: `path-to-catalog/config.json`.

3.7.2 Dataset-level configuration

The dataset-level config (provided during `catalog-add`) can specify the exact same content, although then the catalog-level properties mentioned above will be ignored.

This configuration file will be located at: `path-to-catalog/metadata/<dataset_id>/<dataset_version>/config.json`.

This configuration file will be created for all dataset-level metadata items in the metadata provided to the `catalog-add` operation. For each dataset, this file will override the default config specified on the catalog level.

3.7.3 Inheritance rules

- If not specified by the user on the catalog-level, a default built-in config file is used.
- The catalog-level config serves as the default for dataset-level config.
- If not specified on the dataset-level by the user, the rendering rules will be inherited from the catalog level.

3.7.4 Prioritizing rendered metadata properties

`datalad-catalog` can generate metadata entries that originate from various sources. Through the particular mechanism of catalog entry generation, this information from multiple sources ends up in a single metadata entry in a catalog. It follows that one might want to prioritize information coming from a particular source over another. For example, if metadata from the `metalad_core` as well as the `metalad_studyminimeta` extractors both provide information that maps to the `authors` property of a dataset in a catalog, which one should end up being displayed in the catalog? Or should they be merged? How can I apply a rule to automate such prioritization? And can these rules be set per catalog property?

To cater to these challenges, the catalog's configuration file can specify specific *rules* and how they should be applied in relation to various *sources* of metadata. These rules and sources can be specified *per property* of a file and a dataset.

Here is an example config structure:

```
config = {
    ...
    "property_sources": {
        "dataset": {
            ...
            "description": {
                "rule": "single",
                "source": ["metalad_studyinimeta"]
            },
            "authors": {
                "rule": "priority",
                "source": ["metalad_studyinimeta", "bids_dataset", "datacite_gin"]
            },
            "keywords": {
                "rule": "merge",
                "source": "any"
            },
            "publications": {
                "rule": "merge",
                "source": ["metalad_studyinimeta", "bids_dataset"]
            },
            ...
        },
        "file": {}
    }
    ...
}
```

Rules

A rule can be:

- `single`: only save metadata from a single specified source
- `merge`: merge specified sources together
- `priority`: save only one source from a list of sources, where the sources are prioritised based on the order in which they appear in the list

If no rule is specified, the default rule is "first-come-first-served".

Sources

A source is generally a list of strings, with the list containing:

- a single element, when the `single` rule is specified
- multiple elements, when the `merge` or `priority` rules are specified

The source can also be `any`, meaning that any sources are allowed.

How it works

When metadata from a specific source is added to a catalog, the config is loaded (either from the file specified on the dataset level, or inherited from the catalog level) and this provides the specification (rules and sources) according to which all key-value pairs of the incoming metadata dictionary is evaluated and populated into the catalog metadata.

The catalog metadata for a dataset keeps track of which sources supplied the values for which keys in the metadata dictionary. This is done in order to allow metadata to be updated according to the config-specified rules and sources.

As an example, let's say a dataset in a catalog has the property `dataset_name` with a current value supplied by `source_B`. And let's say the config specifies that the `dataset_name` property can be populated by a number of sources in order of priority `["source_A", "source_B", "source_C"]`. Now, if a catalog update is made that supplies a new value for `dataset_name` from `source_A`, this should result in the new value for `dataset_name` being populated from `source_A`, and in this source information being tracked.

The tracking process is done in the `metadata_sources` of the metadata entry for the specific dataset in the catalog. For example (before the metadata update):

```
{
  "type": "dataset",
  "dataset_id": "....",
  "name": "value_from_source_B",
  ...
  "metadata_sources": {
    "key_source_map": {
      "type": ["metalad_core"],
      "dataset_id": ["metalad_core"],
      "name": ["source_B"],
      ...
    },
    "sources": [
      {
        "source_name": "metalad_core",
        "source_version": "0.0.1",
        "source_parameter": {},
        "source_time": 1643901350.65269,
        "agent_name": "John Doe",
        "agent_email": "email@example.com"
      },
      {
        "source_name": "source_B",
        "source_version": "2",
        "source_parameter": {},
        "source_time": 1643901350.65269,
        "agent_name": "John Doe",
        "agent_email": "email@example.com"
      },
    ]
  }
}
```

As can be seen in the above object, the structure of `metadata_sources`,

- `metadata_sources["sources"]` contains a list of metadata sources (with extra info such as version, agent, etc) that have provided content for this particular metadata record.

- `metadata_sources["key_source_map"]` provides a mapping of which metadata sources were used to provide content for which specific keys in the metadata record.

3.8 Resources

Here are some handy resources that can be useful on your metadata handling and catalog generation journey.

3.8.1 Tutorials

A set of `datalad-catalog`-themed primers and tutorials that can be run live in a Jupyter notebook on Binder.

3.8.2 Handbook chapters

There are detailed and user-friendly chapters in the DataLad Handbook that provide information about `datalad-catalog` and `datalad-metalad`:

- Metadata-Management with MetaLad
- DataCat - a shiny front-end for your dataset

3.9 Contributing

If you have any questions, comments, bug fixes or improvement suggestions, feel free to contact us via our GitHub page. Before contributing, be sure to read the contributing guidelines.

3.10 Command Line Reference

3.10.1 datalad catalog

Synopsis

```
datalad catalog [-h] [--version]
```

Description

Generate a user-friendly web-based data catalog from structured metadata.

`datalad catalog` can be used to `-create` a new catalog, `-add` and `-remove` metadata entries to/from an existing catalog, start a local http server to `-serve` an existing catalog locally. It can also `-validate` a metadata entry (validation is also performed implicitly when adding), `-set` dataset properties such as the `home` page to be shown by default, and `-get` dataset properties such as the `config`, specific `metadata`, or the `home` page.

Metadata can be provided to DataLad Catalog from any number of arbitrary metadata sources, as an aggregated set or as individual metadata items. DataLad Catalog has a dedicated schema (using the JSON Schema vocabulary) against which incoming metadata items are validated. This schema allows for standard metadata fields as one would expect for datasets of any kind (such as name, doi, url, description, license, authors, and more), as well as fields that support identification, versioning, dataset context and linkage, and file tree specification.

The output is a set of structured metadata files, as well as a Vue.js-based browser interface that understands how to render this metadata in the browser. These can be hosted on a platform of choice as a static webpage.

Note: in the catalog website, each dataset entry is displayed under <main page>/#/dataset/<dataset_id>/<dataset_version>. By default, the main page of the catalog will display a 404 error, unless the default dataset is configured with datalad catalog-set home.

Examples

CREATE a new catalog from scratch:

```
% datalad catalog-create -c /tmp/my-cat
```

ADD metadata to an existing catalog:

```
% datalad catalog-add -c /tmp/my-cat -m path/to/metadata.jsonl
```

SET a property of an existing catalog, such as the home page of an existing catalog - i.e. the first dataset displayed when navigating to the root URL of the catalog:

```
% datalad catalog-set -c /tmp/my-cat -i abcd -v 1234 home
```

SERVE the content of the catalog via a local HTTP server at http://localhost:8001:

```
% datalad catalog-serve -c /tmp/my-cat -p 8001
```

VALIDATE metadata against a catalog schema without adding it to the catalog:

```
% datalad catalog-validate -c /tmp/my-cat/-m path/to/metadata.jsonl'
```

GET a property of an existing catalog, such as the catalog configuration:

```
% datalad catalog-get -c /tmp/my-cat/ config
```

REMOVE a specific metadata record from an existing catalog:

```
% datalad catalog-remove -c /tmp/my-cat -i efgh -v 5678
```

TRANSLATE a metalad-extracted metadata item from a particular source structure into the catalog schema. A dedicated translator should be provided and exposed as an entry point (e.g. via a DataLad extension) as part of the 'datalad.metadata.translators' group.:

```
% datalad catalog-translate -c /tmp/my-cat -m path/to/metadata.jsonl
```

RUN A WORKFLOW for recursive metadata extraction (using datalad- metalad), translating metadata to the catalog schema, and adding the translated metadata to a new catalog:

```
% datalad catalog-workflow -t new -c /tmp/my-cat -d path/to/superdataset -e metalad_core
```

RUN A WORKFLOW for updating a catalog after registering a subdataset to the superdataset which the catalog represents. This workflow includes extraction (using datalad-metalad), translating metadata to the catalog schema, and adding the translated metadata to the existing catalog.:

```
% datalad catalog-workflow -t new -c /tmp/my-cat -d path/to/superdataset -s path/to/
↪subdataset -e metalad_core
```

Options

-h, --help, --help-np

show this help message. --help-np forcefully disables the use of a pager for displaying the help message

--version

show the module and its version which provides the command

Authors

datalad is developed by DataLad Developers <team@datalad.org>.

3.10.2 datalad catalog-create

Synopsis

```
datalad catalog-create [-h] [-c CATALOG] [-m METADATA] [-F CONFIG_FILE] [-f] [--version]
```

Description

Create a user-friendly web-based data catalog, with or without metadata.

If the catalog does not exist at the specified location, it will be created. If the catalog exists and the force flag is True, this will overwrite assets of the existing catalog, while catalog metadata remain unchanged.

Parameters

catalog
: [path-like object | WebCatalog instance] an instance of the catalog to be created

metadata
: [path-like object, optional] metadata to be added to the catalog after creation

force
: [bool, optional] if True, will overwrite assets of an existing catalog

Yields

status_dict
: [dict] DataLad result record

Examples

Create a new catalog from scratch:

```
% datalad catalog-create -c /tmp/my-cat
```

Create a new catalog at a location where a directory already exists. This will overwrite all catalog content except for metadata.:

```
% datalad catalog-create -c /tmp/my-cat --force
```

Create a new catalog and add metadata:

```
% datalad catalog-create -c /tmp/my-cat -m path/to/metadata.jsonl
```

Create a new catalog with a custom configuration:

```
% datalad catalog-create -c /tmp/my-cat -F path/to/custom_config_file.json
```

Options

-h, --help, --help-np

show this help message. --help-np forcefully disables the use of a pager for displaying the help message

-c *CATALOG*, --catalog *CATALOG*

Directory where the catalog is located or will be created.

-m *METADATA*, --metadata *METADATA*

The metadata records to be added to the catalog. Multiple input types are possible: - a path to a file containing JSON lines - JSON lines from STDIN - a JSON serialized string.

-F *CONFIG_FILE*, --config-file *CONFIG_FILE*

Path to config file in YAML or JSON format. Default config is read from datalad_catalog/config/config.json.

-f, --force

If content for the user interface already exists in the catalog directory, force this content to be overwritten. Content overwritten with this flag include the 'artwork' and 'assets' directories and the 'index.html' and 'config.json' files. Content in the 'metadata' directory remain untouched.

--version

show the module and its version which provides the command

Authors

datalad is developed by DataLad Developers <team@datalad.org>.

3.10.3 datalad catalog-validate

Synopsis

```
datalad catalog-validate [-h] [-m METADATA] [-c CATALOG] [--version]
```

Description

Validate metadata against the catalog schema

The schema version is determined from the catalog, if provided. Otherwise from the latest supported version of the package installation.

Examples

Validate metadata against a catalog schema without adding it to the catalog:

```
% datalad catalog-validate -c /tmp/my-cat/-m path/to/metadata.jsonl'
```

Options

-h, --help, --help-np

show this help message. --help-np forcefully disables the use of a pager for displaying the help message

-m *METADATA*, --metadata *METADATA*

Path to input metadata. Multiple input types are possible: - A '.json' file containing an array of JSON objects related to a single datalad dataset. - A stream of JSON objects/lines.

-c *CATALOG*, --catalog *CATALOG*

Location of the existing catalog.

--version

show the module and its version which provides the command

Authors

datalad is developed by DataLad Developers <team@datalad.org>.

3.10.4 datalad catalog-add

Synopsis

```
datalad catalog-add [-h] [-c CATALOG] [-m METADATA] [-F CONFIG_FILE] [--version]
```

Description

Add metadata to an existing catalog

Optionally, a dataset-level configuration file can be provided (defaults to the catalog-level config if not provided)

Examples

Add metadata from file to an existing catalog:

```
% datalad catalog-add -c /tmp/my-cat -m path/to/metadata.jsonl
```

Add metadata as JSON string to an existing catalog:

```
% datalad catalog-add -c /tmp/my-cat -m '{"my":"metadata"}'
```

Add metadata as subject to a dataset-level configuration:

```
% datalad catalog-add -c /tmp/my-cat -F path/to/dataset_config_file.json
```

Options

-h, --help, --help-np

show this help message. --help-np forcefully disables the use of a pager for displaying the help message

-c *CATALOG*, --catalog *CATALOG*

Location of the existing catalog.

-m *METADATA*, --metadata *METADATA*

The metadata records to be added to the catalog. Multiple input types are possible: - a path to a file containing JSON lines - JSON lines from STDIN - a JSON serialized string.

-F *CONFIG_FILE*, --config-file *CONFIG_FILE*

Path to config file in YAML or JSON format. Default config is read from: 'datalad_catalog/config/config.json'.

--version

show the module and its version which provides the command

Authors

datalad is developed by DataLad Developers <team@datalad.org>.

3.10.5 datalad catalog-remove

Synopsis

```
datalad catalog-remove [-h] [-c CATALOG] [-i DATASET_ID] [-v DATASET_VERSION] [--
↪reckless] [--version]
```

Description

Remove metadata from an existing catalog

This will remove metadata corresponding to a specified dataset_id and dataset_version from an existing catalog.

This command has to be called with the reckless flag to ignore a warning message.

Examples

REMOVE a specific metadata record from an existing catalog:

```
% datalad catalog-remove -c /tmp/my-cat -i efgh -v 5678 --reckless
```

Options

-h, --help, --help-np

show this help message. --help-np forcefully disables the use of a pager for displaying the help message

-c *CATALOG*, --catalog *CATALOG*

Location of the existing catalog.

-i *DATASET_ID*, --dataset-id *DATASET_ID*

The unique identifier of the dataset for which metadata or config has been requested.

-v *DATASET_VERSION*, --dataset-version *DATASET_VERSION*

The unique version of the dataset for which metadata or config has been requested.

--reckless

Remove the dataset in a potentially unsafe way. A standard catalog-remove call without this flag will provide a warning and do nothing else.

--version

show the module and its version which provides the command

Authors

datalad is developed by DataLad Developers <team@datalad.org>.

3.10.6 datalad catalog-serve

Synopsis

```
datalad catalog-serve [-h] [-c CATALOG] [--port PORT] [--version]
```

Description

Start a local http server to render and test a local catalog.

Optional arguments include a custom port.

Examples

SERVE the content of the catalog via a local HTTP server:

```
% datalad catalog-serve -c /tmp/my-cat
```

SERVE the content of the catalog via a local HTTP server at a custom port, e.g. http://localhost:8001:

```
% datalad catalog-serve -c /tmp/my-cat -p 8001
```

Options

-h, --help, --help-np

show this help message. --help-np forcefully disables the use of a pager for displaying the help message

-c *CATALOG*, --catalog *CATALOG*

Location of the existing catalog to be served.

--port *PORT*

The port at which the content is served at 'localhost' (default 8000). [Default: 8000]

--version

show the module and its version which provides the command

Authors

datalad is developed by DataLad Developers <team@datalad.org>.

3.10.7 datalad catalog-get

Synopsis

```
datalad catalog-get [-h] [-c CATALOG] [-i DATASET_ID] [-v DATASET_VERSION] [--record-
↪type RECORD_TYPE] [--record-path RECORD_PATH] [--version] property
```

Description

Utility for getting various properties of a catalog, based on the specified property ('home', 'config', 'metadata', 'report')

Used to get the catalog home page, get config at catalog- or dataset-level, get the metadata for a specific dataset/version, or get a summary report of the catalog.

Examples

Get the configuration of an existing catalog:

```
% datalad catalog-get -c /tmp/my-cat/ config
```

Get the home page details of an existing catalog:

```
% datalad catalog-get -c /tmp/my-cat/ home
```

Get metadata of a specific dataset from an existing catalog:

```
% datalad catalog-get -c /tmp/my-cat -i abcd -v 1234 metadata
```

Get metadata of a specific directory node in a dataset from an existing catalog:

```
% datalad catalog-get -c /tmp/my-cat -i abcd -v 1234 --record_type directory --record_
↪path relative/path/to/directory metadata
```

Get a report of the number of datasets, dataset-versions, and metadata files contained in existing catalog:

```
% datalad catalog-get -c /tmp/my-cat report
```

Options

property

The property to get in the catalog. Should be one of 'home', 'config', 'metadata' or 'report'.

-h, --help, --help-np

show this help message. --help-np forcefully disables the use of a pager for displaying the help message

-c *CATALOG*, --catalog *CATALOG*

Location of the existing catalog.

-i *DATASET_ID*, --dataset-id *DATASET_ID*

The unique identifier of the dataset for which metadata or config has been requested.

-v *DATASET_VERSION*, --dataset-version *DATASET_VERSION*

The unique version of the dataset for which metadata or config has been requested.

--record-type *RECORD_TYPE*

The type of record in a catalog for which metadata has been requested. Should be one of 'dataset' (default), 'directory', or 'file'.

--record-path *RECORD_PATH*

The relative path of record in a catalog for which metadata has been requested. Required if 'record_type' is 'directory' or 'file'.

--version

show the module and its version which provides the command

Authors

datalad is developed by DataLad Developers <team@datalad.org>.

3.10.8 datalad catalog-set

Synopsis

```
datalad catalog-set [-h] [-c CATALOG] [-i DATASET_ID] [-v DATASET_VERSION] [--reckless␣
↪RECKLESS] [--version] property
```

Description

Utility for setting various properties of a catalog, based on the specified property ('home' or 'config')

Used to set the catalog home page, or to reset config at catalog- or dataset-level. (Note: the latter is not fully supported yet and will yield an error result)

Examples

Set the home page of an existing catalog:

```
% datalad catalog-set -c /tmp/my-cat -i abcd -v 1234 home
```

Set a new home page of an existing catalog, where the home page has previously been set:

```
% datalad catalog-set -c /tmp/my-cat -i efgh -v 5678 --reckless overwrite home
```

Options

property

The property to set in the catalog. Should be one of 'home' or 'config'.

-h, --help, --help-np

show this help message. --help-np forcefully disables the use of a pager for displaying the help message

-c *CATALOG*, --catalog *CATALOG*

Location of the existing catalog.

-i *DATASET_ID*, --dataset-id *DATASET_ID*

The unique identifier of a dataset.

-v *DATASET_VERSION*, --dataset-version *DATASET_VERSION*

The unique version of a dataset.

--reckless *RECKLESS*

Set the property in a potentially unsafe way. Supported modes are: ["overwrite"]: if the property is already set, overwrite it.

--version

show the module and its version which provides the command

Authors

datalad is developed by DataLad Developers <team@datalad.org>.

3.10.9 datalad catalog-translate

Synopsis

```
datalad catalog-translate [-h] [-c CATALOG] [--version] metadata
```

Description

Translate datalad-metalad-extracted metadata items into the catalog schema

The to-be-translated-to schema version is determined from the catalog, if provided, otherwise from the latest supported version of the package installation.

Translators should be provided and exposed as a datalad entry point using the group: 'datalad.metadata.translators'.

Available translators will be filtered based on own matching criteria (such as extractor name, version, etc) to find the appropriate translator, after which the translator's translation code will be executed on the metadata item.

Examples

Translate a metalad-extracted metadata item from a particular source structure into the catalog schema, assuming a dedicated translator is locally available via the entry point mechanism:

```
% datalad catalog-translate -c /tmp/my-cat -m path/to/metadata.jsonl
```

Options

metadata

The metalad-extracted metadata that is to be translated. Multiple input types are possible: - a path to a file containing JSON lines - JSON lines from STDIN - a JSON serialized string.

-h, --help, --help-np

show this help message. --help-np forcefully disables the use of a pager for displaying the help message

-c *CATALOG*, --catalog *CATALOG*

Location of an existing catalog. If this argument is provided it will determine the to-be-translated-to schema version. If the version cannot be found in the catalog, it is determined from the latest supported version of the package installation. The latter is also the default when the 'catalog' argument is not supplied.

--version

show the module and its version which provides the command

Authors

datalad is developed by DataLad Developers <team@datalad.org>.

3.10.10 datalad catalog-workflow

Synopsis

```
datalad catalog-workflow [-h] [-c CATALOG] [-t TYPE] [-d DATASET] [-s SUBDATASET] [-r] [-
↪R RECURSION_LIMIT] [-e EXTRACTOR [EXTRACTOR ...]] [-F CONFIG_FILE] [-f] [--version]
```

Description

Run a workflow to create or update a catalog

This functionality requires the installation of datalad-metalad as well as any datalad extensions providing relevant translators for the extracted metadata items.

It will run a workflow of metadata extraction, translation, and catalog (entry) generation, given a DataLad dataset hierarchy and a specified workflow type: new/update.

Examples

Run a workflow for recursive metadata extraction (using the 'metalad_core' extractor), translating metadata to the latestcatalog schema, and adding the translated metadata to a new catalog:

```
% datalad catalog-workflow -t new -c /tmp/my-cat -d path/to/superdataset -e metalad_core
```

Run a workflow for updating a catalog after registering a subdataset to the superdataset which the catalog represents.:

```
% datalad catalog-workflow -t new -c /tmp/my-cat -d path/to/superdataset -s path/to/
↪subdataset -e metalad_core
```

Options

-h, --help, --help-np

show this help message. --help-np forcefully disables the use of a pager for displaying the help message

-c *CATALOG*, --catalog *CATALOG*

Location of the existing catalog.

-t *TYPE*, --type *TYPE*

Which type of workflow to run: one of ['new', 'update'].

-d *DATASET*, --dataset *DATASET*

The datalad dataset on which to run the workflow.

-s *SUBDATASET*, --subdataset *SUBDATASET*

The datalad subdataset on which to run the update workflow.

-r, --recursive

Specifies whether to recurse into subdatasets or not during workflow execution.

-R *RECURSION_LIMIT*, --recursion-limit *RECURSION_LIMIT*

Specifies how many levels to recurse down into the hierarchy when recursing into subdatasets.

-e *EXTRACTOR* [*EXTRACTOR* ...], --extractor *EXTRACTOR* [*EXTRACTOR* ...]

Which extractors to use during metadata extraction of a workflow. Multiple can be provided. If none are provided, the default extractor 'metalad_core' is used. Any extractor name passed as an argument should first be known to the current installation via datalad's entrypoint mechanism. [Default: ['metalad_core']]

-F *CONFIG_FILE*, --config-file *CONFIG_FILE*

Path to config file in YAML or JSON format. Default config is read from datalad_catalog/config/config.json.

-f, --force

If content for the user interface already exists in the catalog directory, force this content to be overwritten. Content overwritten with this flag include the 'artwork' and 'assets' directories and the 'index.html' and 'config.json' files. Content in the 'metadata' directory remain untouched.

--version

show the module and its version which provides the command

Authors

datalad is developed by DataLad Developers <team@datalad.org>.

3.11 Python Module Reference

catalog()	Generate a user-friendly web-based data catalog from structured metadata.
catalog_create(catalog[, metadata, ...])	Create a user-friendly web-based data catalog, with or without metadata.
catalog_validate(metadata[, catalog])	Validate metadata against the catalog schema
catalog_add(catalog, metadata[, config_file])	Add metadata to an existing catalog
catalog_remove(catalog, dataset_id, ...[, ...])	Remove metadata from an existing catalog
catalog_serve(catalog[, port])	Start a local http server to render and test a local catalog.
catalog_get(catalog, property[, dataset_id, ...])	Utility for getting various properties of a catalog, based on the specified property ('home', 'config', 'metadata', 'report')
catalog_set(catalog, property[, dataset_id, ...])	Utility for setting various properties of a catalog, based on the specified property ('home' or 'config')
catalog_translate(metadata[, catalog])	Translate datalad-metalad-extracted metadata items into the catalog schema
catalog_workflow(catalog, mode, dataset[, ...])	Run a workflow to create or update a catalog

3.11.1 datalad.api.catalog

datalad.api.**catalog**()

Generate a user-friendly web-based data catalog from structured metadata.

`datalad catalog` can be used to `-create` a new catalog, `-add` and `-remove` metadata entries to/from an existing catalog, start a local http server to `-serve` an existing catalog locally. It can also `-validate` a metadata entry (validation is also performed implicitly when adding), `-set` dataset properties such as the `home` page to be shown by default, and `-get` dataset properties such as the `config`, specific `metadata`, or the `home` page.

Metadata can be provided to DataLad Catalog from any number of arbitrary metadata sources, as an aggregated set or as individual metadata items. DataLad Catalog has a dedicated schema (using the JSON Schema vocabulary) against which incoming metadata items are validated. This schema allows for standard metadata fields as one would expect for datasets of any kind (such as name, doi, url, description, license, authors, and more), as well as fields that support identification, versioning, dataset context and linkage, and file tree specification.

The output is a set of structured metadata files, as well as a Vue.js-based browser interface that understands how to render this metadata in the browser. These can be hosted on a platform of choice as a static webpage.

Note: in the catalog website, each dataset entry is displayed under `<main page>/#/dataset/<dataset_id>/<dataset_version>`. By default, the main page of the catalog will display a 404 error, unless the default dataset is configured with `datalad catalog-set home`.

Examples

CREATE a new catalog from scratch:

```
> catalog_create(catalog='/tmp/my-cat')
```

ADD metadata to an existing catalog:

```
> catalog_add(catalog='/tmp/my-cat', metadata='path/to/metadata.jsonl')
```

SET a property of an existing catalog, such as the home page of an existing catalog - i.e. the first dataset displayed when navigating to the root URL of the catalog:

```
> catalog_set(property='home', catalog='/tmp/my-cat', dataset_id='abcd', dataset_
↪version='1234')
```

SERVE the content of the catalog via a local HTTP server at http://localhost:8001:

```
> catalog_serve(catalog='/tmp/my-cat/', port=8001)
```

VALIDATE metadata against a catalog schema without adding it to the catalog:

```
> catalog_validate(catalog='/tmp/my-cat/',metadata='path/to/metadata.jsonl')
```

GET a property of an existing catalog, such as the catalog configuration:

```
> catalog_get(property='config', catalog='/tmp/my-cat/')
```

REMOVE a specific metadata record from an existing catalog:

```
> catalog_remove(catalog='/tmp/my-cat', dataset_id='efgh', dataset_version='5678')
```

TRANSLATE a metalad-extracted metadata item from a particular source structure into the catalog schema. A dedicated translator should be provided and exposed as an entry point (e.g. via a DataLad extension) as part of the 'datalad.metadata.translators' group.:

```
> catalog_translate(catalog='/tmp/my-cat', metadata='path/to/metadata.jsonl')
```

RUN A WORKFLOW for recursive metadata extraction (using datalad- metalad), translating metadata to the catalog schema, and adding the translated metadata to a new catalog:

```
> catalog_workflow(mode='new', catalog='/tmp/my-cat/', dataset='path/to/superdataset
↪', extractor='metalad_core')
```

RUN A WORKFLOW for updating a catalog after registering a subdataset to the superdataset which the catalog represents. This workflow includes extraction (using datalad-metalad), translating metadata to the catalog schema, and adding the translated metadata to the existing catalog.:

```
> catalog_workflow(mode='update', catalog='/tmp/my-cat/', dataset='path/to/
↪superdataset', subdataset='path/to/subdataset', extractor='metalad_core')
```

3.11.2 datalad.api.catalog_create

datalad.api.**catalog_create**(*catalog: Path | WebCatalog*, *metadata=None*, *config_file=None*, *force: bool = False*)

Create a user-friendly web-based data catalog, with or without metadata.

If the catalog does not exist at the specified location, it will be created. If the catalog exists and the force flag is True, this will overwrite assets of the existing catalog, while catalog metadata remain unchanged.

Parameters

- **catalog** (*path-like object | WebCatalog instance*) -- an instance of the catalog to be created

- **metadata** (*path-like object, optional*) -- metadata to be added to the catalog after creation
- **force** (*bool, optional*) -- if True, will overwrite assets of an existing catalog

Yields

status_dict (*dict*) -- DataLad result record

Examples

Create a new catalog from scratch:

```
> catalog_create(catalog='/tmp/my-cat')
```

Create a new catalog at a location where a directory already exists. This will overwrite all catalog content except for metadata.:

```
> catalog_create(catalog='/tmp/my-cat', force=True)
```

Create a new catalog and add metadata:

```
> catalog_create(catalog='/tmp/my-cat', metadata='path/to/metadata.jsonl')
```

Create a new catalog with a custom configuration:

```
> catalog_create(catalog='/tmp/my-cat', config_file='path/to/custom_config_file.json
↪')
```

Parameters

- **catalog** -- Directory where the catalog is located or will be created.
- **metadata** -- The metadata records to be added to the catalog. Multiple input types are possible: - a path to a file containing JSON lines - JSON lines from STDIN - a JSON serialized string. [Default: None]
- **config_file** -- Path to config file in YAML or JSON format. Default config is read from datalad_catalog/config/config.json. [Default: None]
- **force** (*bool, optional*) -- If content for the user interface already exists in the catalog directory, force this content to be overwritten. Content overwritten with this flag include the 'artwork' and 'assets' directories and the 'index.html' and 'config.json' files. Content in the 'metadata' directory remain untouched. [Default: False]
- **on_failure** (*{'ignore', 'continue', 'stop'}, optional*) -- behavior to perform on failure: 'ignore' any failure is reported, but does not cause an exception; 'continue' if any failure occurs an exception will be raised at the end, but processing other actions will continue for as long as possible; 'stop': processing will stop on first failure and an exception is raised. A failure is any result with status 'impossible' or 'error'. Raised exception is an IncompleteResultsError that carries the result dictionaries of the failures in its *failed* attribute. [Default: 'continue']
- **result_filter** (*callable or None, optional*) -- if given, each to-be-returned status dictionary is passed to this callable, and is only returned if the callable's return value does not evaluate to False or a ValueError exception is raised. If the given callable supports ***kwargs* it will additionally be passed the keyword arguments of the original API call. [Default: None]

- **result_renderer** -- select rendering mode command results. 'tailored' enables a command- specific rendering style that is typically tailored to human consumption, if there is one for a specific command, or otherwise falls back on the the 'generic' result renderer; 'generic' renders each result in one line with key info like action, status, path, and an optional message); 'json' a complete JSON line serialization of the full result record; 'json_pp' like 'json', but pretty-printed spanning multiple lines; 'disabled' turns off result rendering entirely; '<template>' reports any value(s) of any result properties in any format indicated by the template (e.g. '{path}', compare with JSON output for all key-value choices). The template syntax follows the Python "format() language". It is possible to report individual dictionary values, e.g. '{metadata[name]}'. If a 2nd-level key contains a colon, e.g. 'music:Genre', ':' must be substituted by '#' in the template, like so: '{metadata[music#Genre]}'. [Default: 'tailored']
- **result_xfm** (*{'datasets', 'successdatasets-or-none', 'paths', 'relpaths', 'metadata'} or callable or None, optional*) -- if given, each to-be-returned result status dictionary is passed to this callable, and its return value becomes the result instead. This is different from *result_filter*, as it can perform arbitrary transformation of the result value. This is mostly useful for top- level command invocations that need to provide the results in a particular format. Instead of a callable, a label for a pre-crafted result transformation can be given. [Default: None]
- **return_type** (*{'generator', 'list', 'item-or-list'}, optional*) -- return value behavior switch. If 'item-or-list' a single value is returned instead of a one-item return value list, or a list in case of multiple return values. *None* is return in case of an empty list. [Default: 'list']

3.11.3 datalad.api.catalog_validate

datalad.api.**catalog_validate**(*metadata*, *catalog=None*)

Validate metadata against the catalog schema

The schema version is determined from the catalog, if provided. Otherwise from the latest supported version of the package installation.

Examples

Validate metadata against a catalog schema without adding it to the catalog:

```
> catalog_validate(catalog='/tmp/my-cat/',metadata='path/to/metadata.jsonl')
```

Parameters

- **metadata** -- Path to input metadata. Multiple input types are possible: - A '.json' file containing an array of JSON objects related to a single datalad dataset. - A stream of JSON objects/lines.
- **catalog** -- Location of the existing catalog. [Default: None]
- **on_failure** (*{'ignore', 'continue', 'stop'}, optional*) -- behavior to perform on failure: 'ignore' any failure is reported, but does not cause an exception; 'continue' if any failure occurs an exception will be raised at the end, but processing other actions will continue for as long as possible; 'stop': processing will stop on first failure and an exception is raised. A failure is any result with status 'impossible' or 'error'. Raised exception is an IncompleteResultsError that carries the result dictionaries of the failures in its *failed* attribute. [Default: 'continue']

- **result_filter** (*callable or None, optional*) -- if given, each to-be-returned status dictionary is passed to this callable, and is only returned if the callable's return value does not evaluate to False or a ValueError exception is raised. If the given callable supports ***kwargs* it will additionally be passed the keyword arguments of the original API call. [Default: None]
- **result_renderer** -- select rendering mode command results. 'tailored' enables a command- specific rendering style that is typically tailored to human consumption, if there is one for a specific command, or otherwise falls back on the the 'generic' result renderer; 'generic' renders each result in one line with key info like action, status, path, and an optional message); 'json' a complete JSON line serialization of the full result record; 'json_pp' like 'json', but pretty-printed spanning multiple lines; 'disabled' turns off result rendering entirely; '<template>' reports any value(s) of any result properties in any format indicated by the template (e.g. '{path}', compare with JSON output for all key-value choices). The template syntax follows the Python "format() language". It is possible to report individual dictionary values, e.g. '{metadata[name]}'. If a 2nd-level key contains a colon, e.g. 'music:Genre', ':' must be substituted by '#' in the template, like so: '{metadata[music#Genre]}'. [Default: 'tailored']
- **result_xfm** (*{'datasets', 'successdatasets-or-none', 'paths', 'relpaths', 'metadata'} or callable or None, optional*) -- if given, each to-be-returned result status dictionary is passed to this callable, and its return value becomes the result instead. This is different from *result_filter*, as it can perform arbitrary transformation of the result value. This is mostly useful for top- level command invocations that need to provide the results in a particular format. Instead of a callable, a label for a pre-crafted result transformation can be given. [Default: None]
- **return_type** (*{'generator', 'list', 'item-or-list'}, optional*) -- return value behavior switch. If 'item-or-list' a single value is returned instead of a one-item return value list, or a list in case of multiple return values. *None* is return in case of an empty list. [Default: 'list']

3.11.4 datalad.api.catalog_add

datalad.api.**catalog_add**(*catalog: Path | WebCatalog*, *metadata*, *config_file=None*)

Add metadata to an existing catalog

Optionally, a dataset-level configuration file can be provided (defaults to the catalog-level config if not provided)

Examples

Add metadata from file to an existing catalog:

```
> catalog_add(catalog='/tmp/my-cat', metadata='path/to/metadata.jsonl')
```

Add metadata as JSON string to an existing catalog:

```
> catalog_add(catalog='/tmp/my-cat', metadata=json.dumps({'my':'metadata'}))
```

Add metadata as subject to a dataset-level configuration:

```
> catalog_add(catalog='/tmp/my-cat', config_file='path/to/dataset_config_file.json')
```

Parameters

- **catalog** -- Location of the existing catalog.

- **metadata** -- The metadata records to be added to the catalog. Multiple input types are possible: - a path to a file containing JSON lines - JSON lines from STDIN - a JSON serialized string.
- **config_file** -- Path to config file in YAML or JSON format. Default config is read from: 'datalad_catalog/config/config.json'. [Default: None]
- **on_failure** (*{'ignore', 'continue', 'stop'}, optional*) -- behavior to perform on failure: 'ignore' any failure is reported, but does not cause an exception; 'continue' if any failure occurs an exception will be raised at the end, but processing other actions will continue for as long as possible; 'stop': processing will stop on first failure and an exception is raised. A failure is any result with status 'impossible' or 'error'. Raised exception is an IncompleteResultsError that carries the result dictionaries of the failures in its *failed* attribute. [Default: 'continue']
- **result_filter** (*callable or None, optional*) -- if given, each to-be-returned status dictionary is passed to this callable, and is only returned if the callable's return value does not evaluate to False or a ValueError exception is raised. If the given callable supports ***kwargs* it will additionally be passed the keyword arguments of the original API call. [Default: None]
- **result_renderer** -- select rendering mode command results. 'tailored' enables a command- specific rendering style that is typically tailored to human consumption, if there is one for a specific command, or otherwise falls back on the the 'generic' result renderer; 'generic' renders each result in one line with key info like action, status, path, and an optional message); 'json' a complete JSON line serialization of the full result record; 'json_pp' like 'json', but pretty-printed spanning multiple lines; 'disabled' turns off result rendering entirely; '<template>' reports any value(s) of any result properties in any format indicated by the template (e.g. '{path}', compare with JSON output for all key-value choices). The template syntax follows the Python "format() language". It is possible to report individual dictionary values, e.g. '{metadata[name]}'. If a 2nd-level key contains a colon, e.g. 'music:Genre', ':' must be substituted by '#' in the template, like so: '{metadata[music#Genre]}'. [Default: 'tailored']
- **result_xfm** (*{'datasets', 'successdatasets-or-none', 'paths', 'relpaths', 'metadata'} or callable or None, optional*) -- if given, each to-be-returned result status dictionary is passed to this callable, and its return value becomes the result instead. This is different from *result_filter*, as it can perform arbitrary transformation of the result value. This is mostly useful for top- level command invocations that need to provide the results in a particular format. Instead of a callable, a label for a pre-crafted result transformation can be given. [Default: None]
- **return_type** (*{'generator', 'list', 'item-or-list'}, optional*) -- return value behavior switch. If 'item-or-list' a single value is returned instead of a one-item return value list, or a list in case of multiple return values. *None* is return in case of an empty list. [Default: 'list']

3.11.5 datalad.api.catalog_remove

datalad.api.**catalog_remove**(*catalog: Path | WebCatalog*, *dataset_id: str*, *dataset_version: str*, *reckless: bool = False*)

Remove metadata from an existing catalog

This will remove metadata corresponding to a specified dataset_id and dataset_version from an existing catalog.

This command has to be called with the reckless flag to ignore a warning message.

Examples

REMOVE a specific metadata record from an existing catalog:

```
> catalog_remove(catalog='/tmp/my-cat', dataset_id='efgh', dataset_version='5678',␣
↪reckless=True)
```

Parameters

- **catalog** -- Location of the existing catalog.
- **dataset_id** -- The unique identifier of the dataset for which metadata or config has been requested.
- **dataset_version** -- The unique version of the dataset for which metadata or config has been requested.
- **reckless** (*bool, optional*) -- Remove the dataset in a potentially unsafe way. A standard catalog- remove call without this flag will provide a warning and do nothing else. [Default: False]
- **on_failure** (*{'ignore', 'continue', 'stop'}, optional*) -- behavior to perform on failure: 'ignore' any failure is reported, but does not cause an exception; 'continue' if any failure occurs an exception will be raised at the end, but processing other actions will continue for as long as possible; 'stop': processing will stop on first failure and an exception is raised. A failure is any result with status 'impossible' or 'error'. Raised exception is an IncompleteResultsError that carries the result dictionaries of the failures in its *failed* attribute. [Default: 'continue']
- **result_filter** (*callable or None, optional*) -- if given, each to-be-returned status dictionary is passed to this callable, and is only returned if the callable's return value does not evaluate to False or a ValueError exception is raised. If the given callable supports ***kwargs* it will additionally be passed the keyword arguments of the original API call. [Default: None]
- **result_renderer** -- select rendering mode command results. 'tailored' enables a command- specific rendering style that is typically tailored to human consumption, if there is one for a specific command, or otherwise falls back on the the 'generic' result renderer; 'generic' renders each result in one line with key info like action, status, path, and an optional message); 'json' a complete JSON line serialization of the full result record; 'json_pp' like 'json', but pretty-printed spanning multiple lines; 'disabled' turns off result rendering entirely; '<template>' reports any value(s) of any result properties in any format indicated by the template (e.g. '{path}', compare with JSON output for all key-value choices). The template syntax follows the Python "format() language". It is possible to report individual dictionary values, e.g. '{metadata[name]}'. If a 2nd-level key contains a colon, e.g. 'music:Genre', ':' must be substituted by '#' in the template, like so: '{metadata[music#Genre]}'. [Default: 'tailored']
- **result_xfm** (*{'datasets', 'successdatasets-or-none', 'paths', 'relpaths', 'metadata'} or callable or None, optional*) -- if given, each to-be-returned result status dictionary is passed to this callable, and its return value becomes the result instead. This is different from *result_filter*, as it can perform arbitrary transformation of the result value. This is mostly useful for top- level command invocations that need to provide the results in a particular format. Instead of a callable, a label for a pre-crafted result transformation can be given. [Default: None]
- **return_type** (*{'generator', 'list', 'item-or-list'}, optional*) -- return value behavior switch. If 'item-or-list' a single value is returned instead of a one-item return value

list, or a list in case of multiple return values. *None* is return in case of an empty list. [Default: 'list']

3.11.6 datalad.api.catalog_serve

datalad.api.**catalog_serve**(*catalog: Path | WebCatalog*, *port: int = 8000*)

Start a local http server to render and test a local catalog.

Optional arguments include a custom port.

Examples

SERVE the content of the catalog via a local HTTP server:

```
> catalog_serve(catalog='/tmp/my-cat/')
```

SERVE the content of the catalog via a local HTTP server at a custom port, e.g. http://localhost:8001:

```
> catalog_serve(catalog='/tmp/my-cat/', port=8001)
```

Parameters

- **catalog** -- Location of the existing catalog to be served.
- **port** -- The port at which the content is served at 'localhost' (default 8000). [Default: 8000]
- **on_failure** (*{'ignore', 'continue', 'stop'}, optional*) -- behavior to perform on failure: 'ignore' any failure is reported, but does not cause an exception; 'continue' if any failure occurs an exception will be raised at the end, but processing other actions will continue for as long as possible; 'stop': processing will stop on first failure and an exception is raised. A failure is any result with status 'impossible' or 'error'. Raised exception is an IncompleteResultsError that carries the result dictionaries of the failures in its *failed* attribute. [Default: 'continue']
- **result_filter** (*callable or None, optional*) -- if given, each to-be-returned status dictionary is passed to this callable, and is only returned if the callable's return value does not evaluate to False or a ValueError exception is raised. If the given callable supports ***kwargs* it will additionally be passed the keyword arguments of the original API call. [Default: None]
- **result_renderer** -- select rendering mode command results. 'tailored' enables a command- specific rendering style that is typically tailored to human consumption, if there is one for a specific command, or otherwise falls back on the the 'generic' result renderer; 'generic' renders each result in one line with key info like action, status, path, and an optional message); 'json' a complete JSON line serialization of the full result record; 'json_pp' like 'json', but pretty-printed spanning multiple lines; 'disabled' turns off result rendering entirely; '<template>' reports any value(s) of any result properties in any format indicated by the template (e.g. '{path}', compare with JSON output for all key-value choices). The template syntax follows the Python "format() language". It is possible to report individual dictionary values, e.g. '{metadata[name]}'. If a 2nd-level key contains a colon, e.g. 'music:Genre', ':' must be substituted by '#' in the template, like so: '{metadata[music#Genre]}'. [Default: 'tailored']
- **result_xfm** (*{'datasets', 'successdatasets-or-none', 'paths', 'relpaths', 'metadata'} or callable or None, optional*) -- if given, each to-be-returned result status dictionary is passed to this callable, and its return value becomes the result instead.

This is different from *result_filter*, as it can perform arbitrary transformation of the result value. This is mostly useful for top- level command invocations that need to provide the results in a particular format. Instead of a callable, a label for a pre-crafted result transformation can be given. [Default: None]

- **return_type** (*{'generator', 'list', 'item-or-list'}, optional*) -- return value behavior switch. If 'item-or-list' a single value is returned instead of a one-item return value list, or a list in case of multiple return values. *None* is return in case of an empty list. [Default: 'list']

3.11.7 datalad.api.catalog_get

datalad.api.**catalog_get**(*catalog: Path | WebCatalog*, *property: str*, *dataset_id: str = None*, *dataset_version: str = None*, *record_type: str = None*, *record_path: str = None*)

Utility for getting various properties of a catalog, based on the specified property ('home', 'config', 'metadata', 'report')

Used to get the catalog home page, get config at catalog- or dataset-level, get the metadata for a specific dataset/version, or get a summary report of the catalog.

Examples

Get the configuration of an existing catalog:

```
> catalog_get(property='config', catalog='/tmp/my-cat/')
```

Get the home page details of an existing catalog:

```
> catalog_get(property='home', catalog='/tmp/my-cat/')
```

Get metadata of a specific dataset from an existing catalog:

```
> catalog_get(property='metadata', catalog='/tmp/my-cat', dataset_id='abcd',␣
↪dataset_version='1234')
```

Get metadata of a specific directory node in a dataset from an existing catalog:

```
> catalog_get(property='metadata', catalog='/tmp/my-cat', dataset_id='abcd',␣
↪dataset_version='1234', record_type='directory', record_path='relative/path/to/
↪directory')
```

Get a report of the number of datasets, dataset-versions, and metadata files contained in existing catalog:

```
> catalog_get(property='report', catalog='/tmp/my-cat')
```

Parameters

- **catalog** -- Location of the existing catalog.
- **property** -- The property to get in the catalog. Should be one of 'home', 'config', 'metadata' or 'report'.
- **dataset_id** -- The unique identifier of the dataset for which metadata or config has been requested. [Default: None]

- **dataset_version** -- The unique version of the dataset for which metadata or config has been requested. [Default: None]
- **record_type** -- The type of record in a catalog for which metadata has been requested. Should be one of 'dataset' (default), 'directory', or 'file'. [Default: None]
- **record_path** -- The relative path of record in a catalog for which metadata has been requested. Required if 'record_type' is 'directory' or 'file'. [Default: None]
- **on_failure** (*{'ignore', 'continue', 'stop'}, optional*) -- behavior to perform on failure: 'ignore' any failure is reported, but does not cause an exception; 'continue' if any failure occurs an exception will be raised at the end, but processing other actions will continue for as long as possible; 'stop': processing will stop on first failure and an exception is raised. A failure is any result with status 'impossible' or 'error'. Raised exception is an IncompleteResultsError that carries the result dictionaries of the failures in its *failed* attribute. [Default: 'continue']
- **result_filter** (*callable or None, optional*) -- if given, each to-be-returned status dictionary is passed to this callable, and is only returned if the callable's return value does not evaluate to False or a ValueError exception is raised. If the given callable supports ***kwargs* it will additionally be passed the keyword arguments of the original API call. [Default: None]
- **result_renderer** -- select rendering mode command results. 'tailored' enables a command- specific rendering style that is typically tailored to human consumption, if there is one for a specific command, or otherwise falls back on the the 'generic' result renderer; 'generic' renders each result in one line with key info like action, status, path, and an optional message); 'json' a complete JSON line serialization of the full result record; 'json_pp' like 'json', but pretty-printed spanning multiple lines; 'disabled' turns off result rendering entirely; '<template>' reports any value(s) of any result properties in any format indicated by the template (e.g. '{path}', compare with JSON output for all key-value choices). The template syntax follows the Python "format() language". It is possible to report individual dictionary values, e.g. '{metadata[name]}'. If a 2nd-level key contains a colon, e.g. 'music:Genre', ':' must be substituted by '#' in the template, like so: '{metadata[music#Genre]}'. [Default: 'tailored']
- **result_xfm** (*{'datasets', 'successdatasets-or-none', 'paths', 'relpaths', 'metadata'} or callable or None, optional*) -- if given, each to-be-returned result status dictionary is passed to this callable, and its return value becomes the result instead. This is different from *result_filter*, as it can perform arbitrary transformation of the result value. This is mostly useful for top- level command invocations that need to provide the results in a particular format. Instead of a callable, a label for a pre-crafted result transformation can be given. [Default: None]
- **return_type** (*{'generator', 'list', 'item-or-list'}, optional*) -- return value behavior switch. If 'item-or-list' a single value is returned instead of a one-item return value list, or a list in case of multiple return values. *None* is return in case of an empty list. [Default: 'list']

3.11.8 datalad.api.catalog_set

datalad.api.**catalog_set**(*catalog: Path | WebCatalog*, *property: str*, *dataset_id: str = None*, *dataset_version: str = None*, *reckless: str = None*)

Utility for setting various properties of a catalog, based on the specified property ('home' or 'config')

Used to set the catalog home page, or to reset config at catalog- or dataset-level. (Note: the latter is not fully supported yet and will yield an error result)

Examples

Set the home page of an existing catalog:

```
> catalog_set(property='home', catalog='/tmp/my-cat', dataset_id='abcd', dataset_
↪version='1234')
```

Set a new home page of an existing catalog, where the home page has previously been set:

```
> catalog_set(property='home', catalog='/tmp/my-cat', dataset_id='efgh', dataset_
↪version='5678', reckless='overwrite')
```

Parameters

- **catalog** -- Location of the existing catalog.
- **property** -- The property to set in the catalog. Should be one of 'home' or 'config'.
- **dataset_id** -- The unique identifier of a dataset. [Default: None]
- **dataset_version** -- The unique version of a dataset. [Default: None]
- **reckless** -- Set the property in a potentially unsafe way. Supported modes are: ["overwrite"]: if the property is already set, overwrite it. [Default: None]
- **on_failure** (*{'ignore', 'continue', 'stop'}, optional*) -- behavior to perform on failure: 'ignore' any failure is reported, but does not cause an exception; 'continue' if any failure occurs an exception will be raised at the end, but processing other actions will continue for as long as possible; 'stop': processing will stop on first failure and an exception is raised. A failure is any result with status 'impossible' or 'error'. Raised exception is an IncompleteResultsError that carries the result dictionaries of the failures in its *failed* attribute. [Default: 'continue']
- **result_filter** (*callable or None, optional*) -- if given, each to-be-returned status dictionary is passed to this callable, and is only returned if the callable's return value does not evaluate to False or a ValueError exception is raised. If the given callable supports ***kwargs* it will additionally be passed the keyword arguments of the original API call. [Default: None]
- **result_renderer** -- select rendering mode command results. 'tailored' enables a command- specific rendering style that is typically tailored to human consumption, if there is one for a specific command, or otherwise falls back on the the 'generic' result renderer; 'generic' renders each result in one line with key info like action, status, path, and an optional message); 'json' a complete JSON line serialization of the full result record; 'json_pp' like 'json', but pretty-printed spanning multiple lines; 'disabled' turns off result rendering entirely; '<template>' reports any value(s) of any result properties in any format indicated by the template (e.g. '{path}', compare with JSON output for all key-value choices). The template syntax follows the Python "format() language". It is possible to report individual dictionary values, e.g. '{metadata[name]}'. If a 2nd-level key contains a colon, e.g. 'music:Genre',

':' must be substituted by '#' in the template, like so: '{metadata[music#Genre]}'. [Default: 'tailored']

- **result_xfm** (*{'datasets', 'successdatasets-or-none', 'paths', 'relpaths', 'metadata'} or callable or None, optional*) -- if given, each to-be-returned result status dictionary is passed to this callable, and its return value becomes the result instead. This is different from *result_filter*, as it can perform arbitrary transformation of the result value. This is mostly useful for top- level command invocations that need to provide the results in a particular format. Instead of a callable, a label for a pre-crafted result transformation can be given. [Default: None]
- **return_type** (*{'generator', 'list', 'item-or-list'}, optional*) -- return value behavior switch. If 'item-or-list' a single value is returned instead of a one-item return value list, or a list in case of multiple return values. *None* is return in case of an empty list. [Default: 'list']

3.11.9 datalad.api.catalog_translate

datalad.api.**catalog_translate**(*metadata*, *catalog=None*)

Translate datalad-metalad-extracted metadata items into the catalog schema

The to-be-translated-to schema version is determined from the catalog, if provided, otherwise from the latest supported version of the package installation.

Translators should be provided and exposed as a datalad entry point using the group: 'datalad.metadata.translators'.

Available translators will be filtered based on own matching criteria (such as extractor name, version, etc) to find the appropriate translator, after which the translator's translation code will be executed on the metadata item.

Examples

Translate a metalad-extracted metadata item from a particular source structure into the catalog schema, assuming a dedicated translator is locally available via the entry point mechanism:

```
> catalog_translate(catalog='/tmp/my-cat', metadata='path/to/metadata.jsonl')
```

Parameters

- **metadata** -- The metalad-extracted metadata that is to be translated. Multiple input types are possible: - a path to a file containing JSON lines - JSON lines from STDIN - a JSON serialized string.
- **catalog** -- Location of an existing catalog. If this argument is provided it will determine the to-be-translated-to schema version. If the version cannot be found in the catalog, it is determined from the latest supported version of the package installation. The latter is also the default when the 'catalog' argument is not supplied. [Default: None]
- **on_failure** (*{'ignore', 'continue', 'stop'}, optional*) -- behavior to perform on failure: 'ignore' any failure is reported, but does not cause an exception; 'continue' if any failure occurs an exception will be raised at the end, but processing other actions will continue for as long as possible; 'stop': processing will stop on first failure and an exception is raised. A failure is any result with status 'impossible' or 'error'. Raised exception is an IncompleteResultsError that carries the result dictionaries of the failures in its *failed* attribute. [Default: 'continue']

- **result_filter** (*callable or None, optional*) -- if given, each to-be-returned status dictionary is passed to this callable, and is only returned if the callable's return value does not evaluate to False or a ValueError exception is raised. If the given callable supports ***kwargs* it will additionally be passed the keyword arguments of the original API call. [Default: None]
- **result_renderer** -- select rendering mode command results. 'tailored' enables a command- specific rendering style that is typically tailored to human consumption, if there is one for a specific command, or otherwise falls back on the the 'generic' result renderer; 'generic' renders each result in one line with key info like action, status, path, and an optional message); 'json' a complete JSON line serialization of the full result record; 'json_pp' like 'json', but pretty-printed spanning multiple lines; 'disabled' turns off result rendering entirely; '<template>' reports any value(s) of any result properties in any format indicated by the template (e.g. '{path}', compare with JSON output for all key-value choices). The template syntax follows the Python "format() language". It is possible to report individual dictionary values, e.g. '{metadata[name]}'. If a 2nd-level key contains a colon, e.g. 'music:Genre', ':' must be substituted by '#' in the template, like so: '{metadata[music#Genre]}'. [Default: 'tailored']
- **result_xfm** (*{'datasets', 'successdatasets-or-none', 'paths', 'relpaths', 'metadata'} or callable or None, optional*) -- if given, each to-be-returned result status dictionary is passed to this callable, and its return value becomes the result instead. This is different from *result_filter*, as it can perform arbitrary transformation of the result value. This is mostly useful for top- level command invocations that need to provide the results in a particular format. Instead of a callable, a label for a pre-crafted result transformation can be given. [Default: None]
- **return_type** (*{'generator', 'list', 'item-or-list'}, optional*) -- return value behavior switch. If 'item-or-list' a single value is returned instead of a one-item return value list, or a list in case of multiple return values. *None* is return in case of an empty list. [Default: 'list']

3.11.10 datalad.api.catalog_workflow

datalad.api.**catalog_workflow**(*catalog: Path | WebCatalog*, *mode: str*, *dataset: Dataset*, *subdataset: Dataset = None*, *recursive=False*, *recursion_limit=None*, *extractor=['metalad_core']*, *config_file=None*, *force: bool = False*)

Run a workflow to create or update a catalog

This functionality requires the installation of datalad-metalad as well as any datalad extensions providing relevant translators for the extracted metadata items.

It will run a workflow of metadata extraction, translation, and catalog (entry) generation, given a DataLad dataset hierarchy and a specified workflow type: new/update.

Examples

Run a workflow for recursive metadata extraction (using the 'metalad_core' extractor), translating metadata to the latestcatalog schema, and adding the translated metadata to a new catalog:

```
> catalog_workflow(mode='new', catalog='/tmp/my-cat/', dataset='path/to/superdataset
↪', extractor='metalad_core')
```

Run a workflow for updating a catalog after registering a subdataset to the superdataset which the catalog represents.:

```
> catalog_workflow(mode='update', catalog='/tmp/my-cat/', dataset='path/to/
↪superdataset', subdataset='path/to/subdataset', extractor='metalad_core')
```

Parameters

- **catalog** -- Location of the existing catalog.
- **mode** -- Which type of workflow to run: one of ['new', 'update'].
- **dataset** -- The datalad dataset on which to run the workflow.
- **subdataset** -- The datalad subdataset on which to run the update workflow. [Default: None]
- **recursive** (*bool, optional*) -- Specifies whether to recurse into subdatasets or not during workflow execution. [Default: False]
- **recursion_limit** -- Specifies how many levels to recurse down into the hierarchy when recursing into subdatasets. [Default: None]
- **extractor** -- Which extractors to use during metadata extraction of a workflow. Multiple can be provided. If none are provided, the default extractor 'metalad_core' is used. Any extractor name passed as an argument should first be known to the current installation via datalad's entrypoint mechanism. [Default: ['metalad_core']]
- **config_file** -- Path to config file in YAML or JSON format. Default config is read from datalad_catalog/config/config.json. [Default: None]
- **force** (*bool, optional*) -- If content for the user interface already exists in the catalog directory, force this content to be overwritten. Content overwritten with this flag include the 'artwork' and 'assets' directories and the 'index.html' and 'config.json' files. Content in the 'metadata' directory remain untouched. [Default: False]
- **on_failure** (*{'ignore', 'continue', 'stop'}, optional*) -- behavior to perform on failure: 'ignore' any failure is reported, but does not cause an exception; 'continue' if any failure occurs an exception will be raised at the end, but processing other actions will continue for as long as possible; 'stop': processing will stop on first failure and an exception is raised. A failure is any result with status 'impossible' or 'error'. Raised exception is an IncompleteResultsError that carries the result dictionaries of the failures in its *failed* attribute. [Default: 'continue']
- **result_filter** (*callable or None, optional*) -- if given, each to-be-returned status dictionary is passed to this callable, and is only returned if the callable's return value does not evaluate to False or a ValueError exception is raised. If the given callable supports ***kwargs* it will additionally be passed the keyword arguments of the original API call. [Default: None]
- **result_renderer** -- select rendering mode command results. 'tailored' enables a command- specific rendering style that is typically tailored to human consumption, if there is one for a specific command, or otherwise falls back on the the 'generic' result renderer; 'generic' renders each result in one line with key info like action, status, path, and an optional message); 'json' a complete JSON line serialization of the full result record; 'json_pp' like 'json', but pretty-printed spanning multiple lines; 'disabled' turns off result rendering entirely; '<template>' reports any value(s) of any result properties in any format indicated by the template (e.g. '{path}', compare with JSON output for all key-value choices). The template syntax follows the Python "format() language". It is possible to report individual dictionary values, e.g. '{metadata[name]}'. If a 2nd-level key contains a colon, e.g. 'music:Genre', ':' must be substituted by '#' in the template, like so: '{metadata[music#Genre]}'. [Default: 'tailored']

- **result_xfm** (*{'datasets', 'successdatasets-or-none', 'paths', 'relpaths', 'metadata'} or callable or None, optional*) -- if given, each to-be-returned result status dictionary is passed to this callable, and its return value becomes the result instead. This is different from *result_filter*, as it can perform arbitrary transformation of the result value. This is mostly useful for top- level command invocations that need to provide the results in a particular format. Instead of a callable, a label for a pre-crafted result transformation can be given. [Default: None]
- **return_type** (*{'generator', 'list', 'item-or-list'}, optional*) -- return value behavior switch. If 'item-or-list' a single value is returned instead of a one-item return value list, or a list in case of multiple return values. *None* is return in case of an empty list. [Default: 'list']

3.12 Change Log

Please refer to the code repository for a complete and up to date changelog.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

C