Quest Documentation

Release 3.1.1

Environmental Simulator Team

Apr 29, 2019

TABLE OF CONTENTS

1 Installation Instructions **3**
- 1.1 Install Released Conda Package . . . 3
- 1.2 Install from Source . . . 3

2 Quickstart **5**
- 2.1 Examples . . . 5

3 Core Concepts **13**
- 3.1 Local Data Organization . . . 13
- 3.2 Data Transformations . . . 14
- 3.3 Data Repositories . . . 14

4 Extending Quest **17**
- 4.1 Provider Plugins . . . 17
- 4.2 Tool Plugins . . . 17
- 4.3 I/O Plugins . . . 17

5 API Reference **19**

6 Developer Documentation **35**
- 6.1 Table of Contents . . . 35

7 Glossary **45**

8 Indices and tables **47**

Python Module Index **49**

Quest is a python library designed to automate the following data management tasks:

- Discovery
- Retrieval
- Organization
- Manipulation
- Archival

Quest can search for and download data from multiple web-based data providers. It can also be configured to search for data from local data repositories. Quest is designed to be extensible, so additional data providers can be added to Quest through plugins. To get started using Quest please see the installation instructions and quickstart reference below.

CHAPTER

ONE

INSTALLATION INSTRUCTIONS

These instructions will walk you through installing Quest either from the released package or from the source code.

1.1 Install Released Conda Package

1. Install Miniconda [or Anaconda although Miniconda is preferred] for your OS
2. Install Quest from the ERDC Environmental Simulator conda channel using one of the following methods:
    a. Install Quest into new environment:

    ```
    conda create -n quest -c conda-forge quest
    ```

    b. Install Quest into existing environment:

    ```
    conda install -c conda-forge quest
    ```

    Note: Because of incompatibilities with the dependencies between conda-forge and the defaults channel, the environment must have been created with conda-forge.

3. Refer to *Quickstart* for more help getting started with Quest.

1.2 Install from Source

1. Clone the repository:

```
git clone https://github.com/erdc/quest.git
```

2. Install the dependencies using conda:
    a. Install Miniconda [or Anaconda although Miniconda is preferred] for your OS.
    b. Create new environment with dependenciest:

    ```
    conda env create -n quest --file conda_environment.yml
    conda activate quest
    ```

    c. Install Quest in develop mode:

```
python setup.py develop
```

1.2.1 Optional

d. Run tests:

```
pytest
```

CHAPTER

TWO

QUICKSTART

Before using Quest you must first activate the quest environment. You can then start a IPython console and import quest:

```
conda activate quest
(quest) $ ipython
```

```
In [1]: import quest
```

The simplest way to download some data is to use the *quest.api.get_data()* call.

```
In [2]: data = quest.api.get_data(
   ...:     collection_name='quick-start',
   ...:     service_uri='svc://usgs-nwis:iv',
   ...:     search_filters={'bbox': [-91, 32.25, -90.8, 32.4]},
   ...:     download_options={'parameter': 'streamflow'},
   ...: )[0]

In [3]: data.head()
Out[3]:
                    qualifiers  streamflow
datetime
2018-04-03 16:00:00          P   1180000.0
2018-04-03 17:00:00          P   1180000.0
2018-04-03 18:00:00          P   1180000.0
2018-04-03 19:00:00          P   1180000.0
2018-04-03 20:00:00          P   1180000.0
```

Quest can download many different types of data from various data providers. In this example we've downloaded timeseries streamflow data from the USGS National Water Information System (NWIS). This type of data is returned as a `pandas.DataFrame` (see Pandas Documentation).

For other examples of how Quest can be used refer to our Jupyter Notebooks or review the examples listed below.

2.1 Examples

2.1.1 Slow Start: A step by step breakdown of the Quickstart example

The quickstart example demonstrated the fastest way to download and start working with data using Quest:

```
In [1]: import quest

In [2]: data = quest.api.get_data(
   ...:     collection_name='quick-start',
   ...:     service_uri='svc://usgs-nwis:iv',
   ...:     search_filters={'bbox': [-91, 32.25, -90.8, 32.4]},
   ...:     download_options={'parameter': 'streamflow'},
   ...: )[0]

In [3]: data.head()
Out[3]:
                     qualifiers  streamflow
datetime
2018-04-03 16:00:00          P   1180000.0
2018-04-03 17:00:00          P   1180000.0
2018-04-03 18:00:00          P   1180000.0
2018-04-03 19:00:00          P   1180000.0
2018-04-03 20:00:00          P   1180000.0
```

There is a lot going on in this seemingly simple example, so we're going to break it down and explain every step.

The first thing to note is that the function *quest.api.get_data()*, is a workflow function, or in other words a function that calls several other functions in succession. This provides a convenient way to get your data in one step when you already know all of inputs you need. You can also use Quest to do the same workflow in a more interactive way. The *quest.api.get_data()* call performs the following steps behind the scenes:

1. *Create a Collection*
2. *Select a Data Service*
3. *Search for Datasets*
4. *Add Datasets to Collection*
5. *Download Datasets*
6. *Open Datasets*

The following sections will explain each of these steps in detail.

Create a Collection

When Quest downloads data it needs to know where to put them. To keep data organized Quest provides a local organization hierarchy to manage data (see *Local Data Organization*). At the top of the hierarchy is a *project*, and all Quest calls will always apply to whatever project is active. For more details about managing projects see (*Project Management*). Within *projects* are *collections* . All data that are downloaded by Quest are put in a *collection* . In the *quest.api.get_data()* example above the `collection_name` argument specifies which *collection* to put the data in. If there isn't already a *collection* with the name specified by the `collection_name` argument then `get_data` function will create it.

This process can also be done manually. Using the Quest API we can get a list of the collections with the `quest.api.get_collection()` function:

```
In [4]: quest.api.get_collections()
Out[4]: ['quick-start']
```

As you can see there currently is only one collection called "quick-start" that was created as a result of the `get_data` call made previously. To create a new collection we manually we can use the *quest.api.new_collection()* function:

```
In [5]: quest.api.new_collection('slow-start')
Out[5]:
{'name': 'slow-start',
 'display_name': 'slow-start',
 'description': '',
 'created_at': datetime.datetime(2019, 4, 4, 13, 43, 14, 823227),
 'updated_at': None,
 'metadata': {}}
```

This function returns the metadata that is associated with this newly created collection. For more details about working with collection see *Collection Management*.

Select a Data Service

Once we have a place to store data locally we need to decide what data we want to download. Quest provides the ability to search for data among many different data sources, or *providers*. Each *provider* will offer one or more data *services* (see *Data Repositories*). We can list the available *services* by calling *`quest.api.get_services()`*:

```
In [6]: quest.api.get_services()
Out[6]:
['svc://cuahsi-hydroshare:hs_geo',
 'svc://cuahsi-hydroshare:hs_norm',
 'svc://noaa-coast:coops-meteorological',
 'svc://noaa-coast:coops-water',
 'svc://noaa-coast:ndbc',
 'svc://noaa-ncdc:ghcn-daily',
 'svc://noaa-ncdc:gsod',
 'svc://quest:quest',
 'svc://usgs-ned:1-arc-second',
 'svc://usgs-ned:13-arc-second',
 'svc://usgs-ned:19-arc-second',
 'svc://usgs-ned:alaska-2-arc-second',
 'svc://usgs-nlcd:2001',
 'svc://usgs-nlcd:2006',
 'svc://usgs-nlcd:2011',
 'svc://usgs-nwis:dv',
 'svc://usgs-nwis:iv',
 'svc://wmts:seamless_imagery']
```

Each *service* is represented by a service URI. In our quickstart example we used the penultimate service URI listed here: 'svc://usgs-nwis:iv'. This service URI is needed to tell Quest where to search for data.

Search for Datasets

Each *service* has a *catalog* or listing of the data it provides. To search for data we need to tell Quest which service's or services' catalog to search. To limit our search we can pass in a dictionary of key-value pairs that specify filter criteria to filter the catalog entries by. In the quickstart example we filtered the catalog using a bounding box.

```
...:     search_filters={'bbox': [-91, 32.25, -90.8, 32.4]},
```

To manually search the catalog we can call the Quest API function *`quest.api.search_catalog()`* and pass it the service URI and the filters dictionary:

```
In [7]: quest.api.search_catalog(uris='svc://usgs-nwis:iv', filters={'bbox': [-91, 32.
↪25, -90.8, 32.4]})
Out[7]: ['svc://usgs-nwis:iv/07289000']
```

The return value from *quest.api.search_catalog()* is a list of *catalog entry* URIs. The *catalog entry* URI looks just like the *service* URI that it came from with an appended catalog ID number. This *catalog entry* URI is used to download the data associated to that *catalog entry*.

Add Datasets to Collection

Before we can download the data associated with a *catalog entry* we need to create a *dataset* derived from that *catalog entry*. A Quest *dataset* represents a piece of data and stores all of the metadata associated with those data. Every Quest *dataset* has an associated *catalog entry* that links it back to the *service* where the data came from, and an associated *collection* that acts as a container for the data. We can create new *datasets* by calling *quest.api.add_datasets()* and passing it both the *collection* and the *catalog entry* or entries from which to create the *datasets*.

```
In [8]: quest.api.add_datasets('slow-start', 'svc://usgs-nwis:iv/07289000')
Out[8: ['d0b2baa58434445fb2d1fee0330d5acf']
```

The return value is a list of dataset IDs from the newly created datasets (in this case it's just a list of one ID. We can now use this dataset ID to download the data associated with it.

Download Datasets

To download data using Quest we use the *quest.api.download_datasets()* function. We need to pass it the *dataset* IDs for the data that we want to download. We also need to pass it a dictionary of download options. Each service specifies it's own set of download options. To figure out what the download options are for a particular dataset we can either refer to the documentation for that dataset's service or we can call *quest.api.get_download_options()* and pass it can pass in either the *service* URI the *catalog entry* URI, or the *dataset* ID.

```
In [9]: quest.api.get_download_options('d0b2baa58434445fb2d1fee0330d5acf')
Out[9]:
{'svc://usgs-nwis:iv/07289000': {'title': 'NWIS Instantaneous Values Service Download␣
↪Options',
  'properties': [{'name': 'parameter',
    'type': 'ObjectSelector',
    'description': 'parameter',
    'default': None,
    'range': [['gage_height', 'gage_height'],
     ['streamflow', 'streamflow'],
     ['water_temperature', 'water_temperature']]},
   {'name': 'start',
    'type': 'Date',
    'description': 'start date',
    'default': None,
    'bounds': None},
   {'name': 'end',
    'type': 'Date',
    'description': 'end date',
    'default': None,
    'bounds': None},
   {'name': 'period',
```

(continues on next page)

(continued from previous page)

```
    'type': 'String',
    'description': 'time period (e.g. P365D = 365 days or P4W = 4 weeks)',
    'default': 'P365D'}]}}
```

This returns a dictionary keyed by the URIs that were passed to the fucntion. For each URI key the value is a dictionary specifying the download options or *properties* for that URI. In this case the download options we can specify are:

- *parameter*: one of 'gage_height', 'streamflow', or 'water_temperature'
- *start*: the start date for the period of data want
- *end*: the end date for the period of data you want
- *period*: a string representing a period of data you want

Here either the start and end date can be specified or a period string can be specified. If neither are specified then the default period 'P365D' (meaning a period of 365 days ending with today) will be used by default. In the quickstart example we specified that we were interested in 'streamflow' data and we didn't specify a period so by default we got the past year of data. We can do the same here by calling *quest.api.download_datasets()*:

```
In [10]: quest.api.download_datasets(
    ...:     datasets='d0b2baa58434445fb2d1fee0330d5acf',
    ...:     options={'parameter': 'streamflow'},
    ...: )
Out[10]: {'d0b2baa58434445fb2d1fee0330d5acf': 'downloaded'}
```

The return value is a dictionary keyed by the dataset IDs that were passed in where the value is the status. In this case 'downloaded' means that the data associated with the dataset were successfully downloaded.

Open Datasets

When the data associated with a *dataset* are downloaded they are by default stored on disk. Quest can be used to transform, visualize, or publish the data and will only require the *dataset* ID as an argument. If you'd like to use other Python tools to work with your data you can use Quest to open your data and read it into a Python data structure. The data that we downloaded are a timeseries of streamflow values. The default data structure that Quest uses for this type of data is a `pandas.DataFrame`. Therefore, when we call *quest.api.open_dataset()* we will getback our data in a DataFrame.

```
In [6]: data = quest.api.open_dataset('d0b2baa58434445fb2d1fee0330d5acf')

In [7]: data.head()
Out[7]:
                    qualifiers  streamflow
  datetime
  2018-04-03 16:00:00          P   1180000.0
  2018-04-03 17:00:00          P   1180000.0
  2018-04-03 18:00:00          P   1180000.0
  2018-04-03 19:00:00          P   1180000.0
  2018-04-03 20:00:00          P   1180000.0
```

Where to Go from Here

2.1.2 Quest Examples

Project Management

```
In [1]: from quest import api

In [2]: api.get_active_project()
Out[2]: 'default'

In [3]: api.get_projects()
Out[3]: ['default']

In [4]: api.new_project('my_proj')
Out[4]:
{'created_at': datetime.datetime(2017, 10, 13, 15, 1, 42, 322881),
 'description': '',
 'display_name': 'my_proj',
 'metadata': {},
 'updated_at': None}

In [5]: api.get_projects()
Out[5]: ['my_proj', 'default']

In [6]: api.set_active_project('my_proj')
Out[6]: 'my_proj'

In [7]: api.get_active_project()
Out[7]: 'my_proj'

In [8]: api.delete_project('my_proj')
Out[8]: {'default': {'folder': 'default'}}

In [9]: api.get_active_project()
Out[9]: 'default'
```

Collection Management

```
In [1]: from quest import api

In [2]: api.get_collections()
Out[2]: []

In [3]: api.new_collection('demo')
Out[3]:
{'created_at': datetime.datetime(2017, 10, 13, 15, 5, 33, 385739),
 'description': '',
 'display_name': 'demo',
 'metadata': {},
 'name': 'demo',
 'updated_at': None}

In [4]: api.get_collections()
Out[4]: ['demo']

In [5]: api.delete('demo')
Out[5]: True
```

(continues on next page)

(continued from previous page)

```
In [6]: api.get_collections()
Out[6]: []
```

Applying Tools

Continuing from previous example.

```
In [16]: dataset = datasets[0]

In [17]: api.get_tools(filters={'dataset': dataset})
Out[17]: ['ts-flow-duration', 'ts-resample', 'ts-unit-conversion', 'ts-remove-outliers
↪']

In [18]: filter_name = 'ts-resample'

In [19]: api.apply_filter_options(filter_name)
Out[19]:
{'properties': {'method': {'description': 'resample method',
   'type': {'default': 'mean',
    'enum': ['sum', 'mean', 'std', 'max', 'min', 'median']}},
  'period': {'description': 'resample frequency',
   'type': {'default': 'daily',
    'enum': ['daily', 'weekly', 'monthly', 'annual']}}},
 'required': ['period', 'method'],
 'title': 'Resample Timeseries Filter',
 'type': 'object'}

In [20]: options = {'method': 'max', 'period': 'daily'}

In [21]: api.run_filter(filter_name, datasets=dataset, options=options)
Out[21]: {'datasets': ['db98e371e7a64a02a773004c6ddc90ff'], 'features': []}

In [22]: api.get_metadata('db98e371e7a64a02a773004c6ddc90ff')
Out[22]:
{'db98e371e7a64a02a773004c6ddc90ff': {'collection': 'demo',
  'created_at': Timestamp('2017-10-13 16:01:20.665627'),
  'datatype': 'timeseries',
  'description': 'TS Filter Applied',
  'display_name': 'db98e371e7a64a02a773004c6ddc90ff',
  'feature': 'fa2e58257ec04d4cb0f18feec51df736',
  'file_format': 'timeseries-hdf5',
  'file_path': '/path/to/quest/projects/default/demo/usgs-nwis/iv/
↪d70123cb1ad944a988f64f449a7d8e8e/db98e371e7a64a02a773004c6ddc90ff',
  'message': 'TS Filter Applied',
  'metadata': {},
  'name': 'db98e371e7a64a02a773004c6ddc90ff',
  'options': {'dataset': ['d70123cb1ad944a988f64f449a7d8e8e'],
   'features': None,
   'filter_applied': 'ts-resample',
   'filter_options': {'method': 'max', 'period': 'daily'}},
  'parameter': 'streamflow:daily:max',
  'source': 'derived',
  'status': 'filter applied',
  'unit': 'ft3/s',
  'updated_at': None,
```

(continues on next page)

(continued from previous page)

```
  'visualization_path': ''}}
```

CHAPTER

THREE

CORE CONCEPTS

Quest is a python library designed to automate the following data management tasks:

- Discovery
- Retrieval
- Organization
- Transformation
- Archival

At the heart of all of these tasks are *datasets*. Each of the tasks listed above involves finding, getting, storing, changing, or sharing a `dataset`. The underlying concepts for how Quest accomplishes these five tasks will be described below and are grouped into the following three sections:

- *Local Data Organization*
- *Data Transformations*
- *Data Repositories*
  - Discovery
  - Retrieval
  - Archival

3.1 Local Data Organization

Quest uses a hierarchical structure to organize and manage datasets, and data sources. The dataset hierarchy begins with *projects* which contains *collections* which have *datasets*. A more detailed description of each level is given below.

3.1.1 Projects

A Quest Project is the base organizing factor. The first time Quest is started a default project is created. Only one project can be active at a time and currently the api does not allow copying data from one project to another.

Physically, a project maps to a folder on the computer. All data and metadata associated with a project is stored under the project folder. The metadata is stored in a sqlite database.

3.1.2 Collections

Collections are a way of organizing data within a project. Collection names are unique and the collection name maps directly to a folder name in the project folder.

3.1.3 Datasets

These are the actual individual data files or in some cases a folder of data. Datasets have associated metadata that is stored in the project directory.

3.2 Data Transformations

Quest facilitates transforming data through the use of *tools*. Some examples of the kinds of transformations that Quest can do include merging datasets, aggregating data within a dataset, or changing the format that the data is stored in.

3.2.1 Tools

Quest `tools` are a way to perform some kind of operation on data. It is important to note that a `tool` will never perform "in-place" changes the datasets that it operates on. This means that datasets that are passed to a `tool` will remain unchanged, and the `tool` will create new datasets that have the transformed data. New `tools` can be added to Quest through *Tool Plugins*.

Tools define a set of options that a user must specify when using the tool.

3.3 Data Repositories

When Quest is used to search for data it searches among all of the data repositories or data *providers* that are registered with Quest. Similar to *Tools Providers* are added to Quest as plugins (see *Provider Plugins*). *Providers* contain one or more *services*. *Services* provide an interface for a single data product. Each service has a *Catalogs*, which stores metadata about the datasets that are available from that service and is what enables Quest to search for data.

3.3.1 Providers

Data *providers* are the top level source of data. Providers are composed of one or more *Services*, and typically represent an organization or specific part of an organization that provides data. In Quest, *providers* are a way of grouping related services.

3.3.2 Services

A data service is a specific type or channel of data that is offered from a *Providers*, and are the primary means of ingesting data into Quest.

3.3.3 Catalogs

3.3.4 Catalog Entries

Catalog Entries are a unique identifiers that indicate a group of datasets. Typically, these are geospatial locations, i.e., monitoring stations, counties, lakes, roads at which data exists. Features can also just be a tag or name to group data that does not have a geospatial component (i.e. geotypical datasets). Features are always either part of a collection or part of a web service.

CHAPTER

FOUR

EXTENDING QUEST

4.1 Provider Plugins

4.2 Tool Plugins

4.3 I/O Plugins

CHAPTER

FIVE

API REFERENCE

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

Python API for Environmental Simulator Quest (QUEST).

This module defines the Python API for the Environmental Simulator Data Services Library.

`quest.api.`**`add_datasets`**(*collection*, *catalog_entries*)

Adds new datasets (created from `catalog_entries`) to `collection`

Parameters

- **`collection`** (*`string, Required`*) – name of collection
- **`catalog_entries`** (string, comma separated strings, list of strings, or `pandas.DataFrame`, Required) – list of catalog entry uris from which to create new datasets to add to the collection.

Returns uris of newly created datasets

Return type uris (list)

`quest.api.`**`add_project`**(*name*, *path*, *activate=True*)

Add a existing QUEST project to the list of available projects

Parameters

- **`name`** (*`string, Required`*) – name of project; existing name can be used or project can be renamed
- **`path`** (*`string, Required`*) – path to existing project
- **`activate`** (*`bool, Optional, Default=True`*) – if True, the added project is set as the currently active project

`quest.api.`**`add_user_provider`**(*uri*)

Add a custom web service created from a file or http folder.

Converts a local/network or http folder that contains a quest.yml and associated data into a service that can be accessed through quest

Parameters **`uri`** (*`string, Required`*) – uri of new 'user' service

Returns status of adding service (i.e. failed/success)

Return type message (string)

`quest.api.`**`authenticate_provider`**(*uri*, ***kwargs*)

Authenticate the user.

Parameters **`uri`** – uri of 'user service'

quest.api.**cancel_tasks**(*task_ids*)
Cancel tasks.

Parameters **task_ids** (*string or list of strings, Required*) – id of tasks to be cancelled

quest.api.**copy**(*uris*, *destination_collection*, *as_dataframe=None*, *expand=None*)

quest.api.**delete**(*uris*)
Delete metadata for resource(s)

Warning: deleting a collection deletes all associated datasets

Parameters **uris** (*string, comma separated string or list of strings, Required*) – uri(s) of collection, and/or dataset to delete

Returns True on success

Return type status (bool)

quest.api.**delete_project**(*name*)
Delete a project.

Deletes a project and all data in the project folder.

Parameters **name** (*string, Required*) – name of a project

Returns all remaining projects and their respective folders

Return type projects (dict)

quest.api.**delete_user_provider**(*uri*)
Remove 'user' service.

Parameters **uri** – uri of 'user service'

quest.api.**download**(*catalog_entry*, *file_path*, *dataset=None*, ***kwargs*)
Download dataset and save it locally.

Parameters

- **catalog_entry** (*string, Required*) – uri of catalog_entry within a service or collection
- **file_path** (*string, Required*) – path location to save downloaded data
- **dataset** (*string, Optional, Default=None*) – maybe only be used by some providers
- **async** – (bool, Optional, Default=False) if True, download in background
- **kwargs** – optional download kwargs

Returns details of downloaded data

Return type data (dict)

quest.api.**download_datasets**(*datasets*, *options=None*, *raise_on_error=False*)
Download datasets and save them in the Quest project.

Parameters

- **datasets** (*string or list, Required*) – datasets to download

- **options** (*dict, Optional, Default=None*) – Dictionary of download options to stage the datasets with before downloading (see *quest.api.stage_for_download()*)
- **raise_on_error** (*bool, Optional, Default=False*) – if True, if an error occurs raise an exception
- **async** (*bool, Optional, Default=False*) – if True, download in background

---

Note: If *options* are not provided then the *datasets* should already download options set by calling *quest.api.stage_for_download()*.

Returns download status of datasets

Return type status (dict)

quest.api.**get_download_options**(*uris*, *fmt='json'*)
List optional kwargs that can be specified when downloading a dataset.

Parameters

- **uris** (*string or list, Required*) – uris of catalog_entries or datasets
- **fmt** – format in which to return download_options. One of ['json', 'param']

quest.api.**get_active_project**()
Get active project name.

Returns name of currently active project

Return type name (string)

quest.api.**get_api_version**()
Get QUEST API version.

Returns version of QUEST API being used

Return type QUEST version (string)

quest.api.**get_auth_status**(*uri*)
Check to see if a provider has been authenticated

Parameters **uri** – uri of 'user service'

quest.api.**get_collections**(*expand=False*, *as_dataframe=False*)
Get available collections.

Collections are folders on the local disk that contain downloaded or created data along with associated metadata.

Parameters

- **expand** (*bool, Optional, Default=False*) – include collection details and format as dict
- **as_dataframe** (*bool, Optional, Default=False*) – include collection details and format as pandas dataframe

Returns all available collections

Return type collections (list, dict, or pandas dataframe, Default=list)

quest.api.**get_data**(*service_uri*, *search_filters=None*, *search_queries=None*, *download_options=None*, *collection_name='default'*, *expand=False*, *use_cache=True*, *max_catalog_entries=10*, *as_open_datasets=True*, *raise_on_error=False*)

Downloads data from source uri and adds to a quest collection.

Parameters

- **service_uri** (*string, required*) – uri for service to get data from
- **search_filters** (*dict, optional, default=None*) – dictionary of search filters to filter the catalog search. At least one of *search_filters* or *search_queries* should be specified. (see docs for *quest.api.search_catalog()*)
- **search_queries** (*list, optional, default=None*) – list of string arguments to pass to pandas.DataFrame.query to filter the catalog search. At least one of *search_filters* or *search_queries* should be specified. (see docs for *quest.api.search_catalog()*)
- **download_options** (*dict or Parameterized, optional, default=None*) – dictionary or Parameterized object with download options for service (see docs for quest.api.download_datasets)
- **collection_name** (*string, optional, default='default'*) – name of collection to add downloaded data to. If collection doesn't exist it will be created.
- **expand** (*bool, optional, default=False*) – include dataset details and format as dict
- **use_cache** (*bool, optional, default=True*) – if True then previously downloaded datasets with the same download options will be returned rather than downloading new datasets
- **max_catalog_entries** (*int, optional, default=10*) – the maximum number of datasets to allow in the search. If exceeded a Runtime error is raised.
- **as_open_datasets** (*bool, optional, default=False*) – if True return datasets as Python data structures rather than as dataset ids (see docs for quest.api.open_dataset)
- **raise_on_error** (*bool, optional, default=False*) – if True then raise an exception if no datasets are returned in the search, or if there is an error while downloading any of the datasets.

Returns the quest dataset name, or an python data structure if open_dataset=True.

quest.api.**get_datasets**(*expand=None*, *filters=None*, *queries=None*, *as_dataframe=None*)

Return all available datasets in active project.

Parameters

- **expand** (*bool, Optional, Default=None*) – include dataset details and format as dict
- **filters** (*dict, Optional, Default=None*) – filter dataset by any metadata field
- **queries** (*list, Optional, Default=None*) – list of string arguments to pass to pandas.DataFrame.query to filter the datasets
- **as_dataframe** (*bool or None, Optional, Default=None*) – include dataset details and format as pandas dataframe

Returns staged dataset uids

Return type uris (list, dict, pandas Dataframe, Default=list)

quest.api.**get_mapped_parameters**()
Get list of common parameters.

Returns list of common parameters

Return type parameters (list)

quest.api.**get_metadata**(*uris*, *as_dataframe=False*)
Get metadata for uris.

Parameters

- **uris** (*string, comma separated string, or list of strings, Required*) – list of uris to retrieve metadata for
- **as_dataframe** (*bool, Optional, Default=False*) – include details of newly created dataset as a pandas Dataframe

Returns metadata at each uri keyed on uris

Return type metadata (dict or pd.DataFrame, Default=dict)

quest.api.**get_parameters**(*service_uri*, *update_cache=False*)
Get available parameters, even unmapped ones, for specified service.

Parameters

- **service_uri** (*string, Required*) – uri of service to get parameters for
- **update_cache** (*bool, Optional, Default=True*) – if True, update metadata cache

Returns all available parameters for specified service

Return type parameters (list)

quest.api.**get_pending_tasks**(***kwargs*)
Return list of pending tasks

calls get_tasks with filter -> status=pending, passes through other kwargs (filters={}, expand=None, as_dataframe=None, with_future=None)

quest.api.**get_projects**(*expand=False*, *as_dataframe=False*)
Get list of available projects.

Parameters **expand** – include collection details and format as dict

Returns all available projects

Return type projects (list, dict, or pandas Dataframe,Default=list)

quest.api.**get_providers**(*expand=None*, *update_cache=False*)
Return list of Providers.

Parameters

- **expand** (*bool, Optional, Default=None*) – include providers' details and format as dict
- **update_cache** (*bool, Optional, Default=False*) – reload the list of providers

Returns list of all available providers

Return type providers (list or dict, Default=list)

quest.api.**get_publishers**(*expand=None*, *publisher_type=None*)

This method returns a list of avaliable publishers.

The method first gets a dictionary filled with the available providers in Quest. Then we loop through grabbing the keys and the objects within the dictionary. Then we loop again, accessing each service getting another dictionary with the provider as the key and the metadata as the values. Then we create a publish uri, and get the publisher class name for the service. We return a list of publishers.

Parameters

- **expand** (*bool, Optional, Default=False*) – include providers' details and format as dict
- **publisher_type** (*string, Optional, Default=None'*) – filter to only include specific type

Returns list of all available providers

Return type providers (list or dict,Default=list)

quest.api.**get_quest_version**()

Get QUEST version.

Returns version of QUEST being used

Return type QUEST version (string)

quest.api.**get_seamless_data**(*service_uri*, *bbox*, *search_filters=None*, *search_queries=None*, *download_options=None*, *collection_name='default'*, *expand=False*, *use_cache=True*, *max_catalog_entries=10*, *as_open_dataset=True*, *raise_on_error=False*)

Downloads raster data from source uri and adds to a quest collection.

If multiple raster tiles are retrieved for the given bounds it calls a quest tool to merge the tiles into a single raster.

Parameters

- **service_uri** (*string, required*) – uri for service to get data from
- **bbox** (*list, required*) – list of lat/lon coordinates representing the bounds of the data in for form [lon_min, lat_min, lon_max, lat_max].
- **search_filters** (*dict, optional, default=None*) – dictionary of search filters to filter the catalog search (see docs for *quest.api.search_catalog()*)
- **search_queries** (*list, optional, default=None*) – list of string arguments to pass to `pandas.DataFrame.query` to filter the catalog search (see docs for *quest.api.search_catalog()*)
- **download_options** (*dict or Parameterized, optional, default=None*) – dictionary or Parameterized object with download options for service (see docs for quest.api.download_datasets)
- **collection_name** (*string, optional, default='default'*) – name of collection to add downloaded data to. If collection doesn't exist it will be created.
- **expand** (*bool, optional, default=False*) – include dataset details and format as dict
- **use_cache** (*bool, optional, default=True*) – if True then previously downloaded datasets with the same download options will be returned rather than downloading new datasets
- **max_catalog_entries** (*int, optional, default=10*) – the maximum number of datasets to allow in the search. If exceeded a Runtime error is raised.

- **as_open_dataset** (*bool, optional, default=False*) – if True return dataset as Python data structure rather than as a dataset id (see docs for quest.api.open_dataset)
- **raise_on_error** (*bool, optional, default=False*) – if True then raise an exception if no datasets are returned in the search, or if there is an error while downloading.

Returns the quest dataset name.

quest.api.**get_services**(*expand=None*, *parameter=None*, *service_type=None*)
Return list of Services.

Parameters

- **expand** (*bool, Optional, Default=False*) – include providers' details and format as dict
- **parameter** (*string, Optional, Default=None*) –
- **service_type** (*string, Optional, Default=None'*) – filter to only include specific type

Returns all available providers

Return type providers (list or dict, Default=dict)

quest.api.**get_settings**()
Get the settings currently being used by QUEST.

Returns A dictionary of the current settings.

Example

{'BASE_DIR': '/Users/dharhas/', 'CACHE_DIR': 'cache', 'PROJECTS_DIR': 'projects', 'USER_SERVICES': [], }

quest.api.**get_tags**(*service_uris*, *update_cache=False*, *filter=None*, *as_count=False*)
Get searchable tags for a given service.

Parameters

- **service_uris** (*string or list, Required*) – uris of providers
- **update_cache** (*bool, Optional*) – if True, update metadata cache
- **filter** (*list, Optional*) – list of tags to include in return value
- **as_count** (*bool, Optional*) – if True, return dictionary with the number of values rather than a list of possible values

Returns

dict keyed by tag name and list of possible values

Note: nested dicts are parsed out as a multi-index tag where keys for nested dicts are joined with ':'.

Return type tags (dict)

quest.api.**get_task**(*task_id*, *with_future=None*)
Get details for a task.

Parameters

- **task_id** (*string,Required*) – id of a task

- **with_future** (*bool, Optional, Default=None*) – If true include the task *future* objects in the returned dataframe/dictionary

quest.api.**get_tasks** (*filters=None*, *expand=None*, *as_dataframe=None*, *with_future=None*)

Get all available tasks.

Parameters

- **filters** (*dict, Optional, Default=None*) –

  filter tasks by one or more of the available filters

  available filters:

  - *task_ids* (str or list): task id or list of task ids
  - ***status* (str or list): single status or list of statuses. Must be subset of** ['pending', 'cancelled', 'finished', 'lost', 'error']
  - *fn* (str): name of the function a task was assigned
  - *args* (list): list of arguments that were passed to the task function
  - *kwargs* (dict): dictionary of keyword arguments that were passed to the task function
  - *result* (object): result of the task function

- **expand** (*bool, Optional, Default=None*) – include details of tasks and format as a dict
- **as_dataframe** (*bool, Optional, Default=None*) – include details of tasks and format as a pandas dataframe
- **with_future** (*bool, Optional, Default=None*) – If true include the task *future* objects in the returned dataframe/dictionary

Returns all available tasks

Return type tasks (list, dict, or pandas dataframe, Default=list)

quest.api.**get_tools** (*filters=None*, *expand=False*, ***kwargs*)

List available tool plugins

Parameters

- **filters** (*dict, Optional, Default=None*) – filter the list of tools by one or more of the available filters

  Available Filters:

  - *dataset*
  - *group*
  - *geotype*
  - *datatype*
  - *parameter*

  Note: If a dataset filter is used, all other filters are overridden and set from the dataset's metadata.

- **expand** (*bool, Optional, Default=None*) – if True, return details of the filters as a dict
- **kwargs** – optional filter kwargs

Returns all available tools

Return type tools (list or dict, Default=list)

quest.api.**get_tool_options**(*name*, *fmt='json'*, ***kwargs*)

Retrieve kwarg options for run_tool.

Parameters

- **name** (*string, Required*) – name of filter
- **fmt** (*string, Required, Default='json'*) – format in which to return options. One of ['json', 'param']
- **kwargs** – keyword arguments of options to set and exclude from return value.

Returns tool options that can be applied when calling quest.api.run_filter

Return type tool options (json scheme)

quest.api.**move**(*uris*, *destination_collection*, *as_dataframe=None*, *expand=None*)

quest.api.**new_catalog_entry**(*geometry=None*, *geom_type=None*, *geom_coords=None*, *metadata=None*)

Add a new entry to a catalog either a quest local catalog (table) or file.

Parameters

- **geometry** (*string or Shapely.geometry.shape, optional, Default=None*) – well-known-text or Shapely shape representing the geometry of the catalog_entry. Alternatively *geom_type* and *geom_coords* can be passed.
- **geom_type** (*string, Optional, Default=None*) – geometry type of catalog_entry (i.e. point/line/polygon)
- **geom_coords** (*string or list, Optional, Default=None*) – geometric coordinates specified as valid geojson coordinates (i.e. a list of lists i.e. '[[-94.0, 23.2], [-94.2, 23.4] . . .]' ——— OR ——— [[-94.0, 23.2], [-94.2, 23.4] . . .] etc)
- **metadata** (*dict, Optional, Default=None*) – optional metadata at the new catalog_entry

Returns uri of newly created entry

Return type uri (string)

quest.api.**new_collection**(*name*, *display_name=None*, *description=None*, *metadata=None*, *exists_ok=False*)

Create a new collection.

Create a new collection by creating a new folder in project directory and adding collection metadata in project database.

Parameters

- **name** (*string, Required*) – Name of the collection used in all quest function calls,must be unique. Will also be the folder name of the collection
- **display_name** (*string, Optional, Default=None*) – display name for collection
- **description** (*string, Optional, Default=None*) – description of collection
- **metadata** (*dict, Optional, Default=None*) – user defined metadata
- **exists_ok** (*bool, Optional, Default=False*) – If True then `ValueError` is not raised if the collection already exits. Rather the metadata of the existing colleciton is returned.

Returns details of the newly created collection

Return type dict

Raises **ValueError** – If collection with name already exists.

quest.api.**new_dataset**(*catalog_entry*, *collection*, *source=None*, *display_name=None*, *description=None*, *file_path=None*, *metadata=None*, *name=None*)

Create a new dataset in a collection.

Parameters

- **catalog_entry** (*string, Required*) – catalog_entry uri
- **collection** (*string, Required*) – name of collection to create dataset in
- **source** (*string, Optional, Default=None*) – type of the dataset such as timeseries or raster
- **display_name** (*string, Optional, Default=None*) – display name for dataset
- **description** (*string, Optional, Default=None*) – description of dataset
- **file_path** (*string, Optional, Default=None*) – path location to save new dataset's data
- **metadata** (*dict, Optional, Default=None*) – user defined metadata
- **name** (*dict, Optional, Default=None*) – optionally pass in a UUID starting with d as name, otherwise it will be generated

Returns uid of dataset

Return type uri (string)

quest.api.**new_parameter**(*uri*, *parameter_name*)

Add new parameter to collection.

quest.api.**new_project**(*name*, *display_name=None*, *description=None*, *metadata=None*, *folder=None*, *activate=True*)

Create a new QUEST project and add it to list of available projects.

Parameters

- **name** (*string, Required*) – name of newly created project
- **display_name** (*string, Optional, Default=None*) – display name for project
- **description** (*string, Optional, Default=None*) – description of project
- **metadata** (*dict, Optional, Default=None*) – user defined metadata
- **folder** (*string, Optional, Default=None*) – folder where all project data will be saved
- **activate** (*bool, Optional, Default=True*) – if True, set newly created project as currently active project

quest.api.**open_dataset**(*dataset*, *fmt=None*, ***kwargs*)

Open the dataset and return in format specified by fmt

Parameters

- **dataset** (*string, Required*) – uid of dataset to be opened

- **fmt** (*string, Optional, Default=None*) – format in which dataset should be returned will raise NotImplementedError if format requested is not possible

Returns contents of dataset

Return type data (pandas dataframe, json, or dict, Default=dataframe)

quest.api.**publish**(*publisher_uri*, *options=None*, ***kwargs*)

quest.api.**get_publish_options**(*publish_uri*, *fmt='json'*)

quest.api.**remove_project**(*name*)

Remove a project from the list of available projects.

This does not delete the project folder or data, just removes it from the index of available projects.

Parameters **name** – name of project

quest.api.**remove_tasks**(*task_ids=None*, *status=None*)

Remove tasks.

Parameters

- **task_ids** (*string or list, Optional, Default=None*) – tasks with specified id(s) will be removed
- **status** (*string or list, Optional, Default=None*) –

  tasks with specified status(es) will be removed. Valid statuses are:

  - *cancelled*
  - *finished*
  - *lost*
  - *error*

  NOTE: *pending* is not a valid option and will be ignored, since pending tasks must be canceled before they can be removed.
- **no status is specified, remove tasks with** (*If*) –
- **= ['cancelled', 'finished', 'lost', 'error'] from task list** (*status*) –

quest.api.**run_tool**(*name*, *options=None*, *as_dataframe=None*, *expand=None*, *as_open_datasets=None*, ***kwargs*)

Apply Tool to dataset.

Parameters

- **name** (*string,Required*) – name of filter
- **options** (*dict, Required*) – a dictionary of arguments to pass to the filter formatted as specified by *get_tool_options*
- **expand** (*bool, Optional, Default=False*) – include details of newly created dataset and format as a dict
- **as_dataframe** (*bool, Optional, Default=False*) – include details of newly created dataset and format as a pandas dataframe
- **as_open_datasets** (*bool, Optional, Default=False*) – returns datasets as Python data structures rather than Quest IDs
- **async** (*bool,Optional*) – if True, run filter in the background
- **kwargs** – keyword arguments that will be added to *options*

Returns resulting datasets and/or catalog_entries

Return type dataset/catalog_entry uris (dict or pandas dataframe, Default=dict)

quest.api.**save_settings**(*filename=None*)

Save settings currently being used by QUEST to a yaml file.

Parameters **filename** (*string*) – Path to the yaml file to save the settings.

Returns A true boolean if settings were saved successfully.

quest.api.**set_active_project**(*name*)

Set active QUEST project.

Parameters **name** (*string, Required*) – name of a project

Returns name of project currently set as active

Return type project (string)

quest.api.**search_catalog**(*uris=None*, *expand=False*, *as_dataframe=False*, *as_geojson=False*, *update_cache=False*, *filters=None*, *queries=None*)

Retrieve list of catalog entries from resources.

Parameters

- **uris** (*string or list, Required*) – uris of service_uris
- **expand** (*bool, Optional, Default=False*) – if true then return metadata along with catalog entries
- **as_dataframe** (*bool, Optional, Default=False*) – include catalog_entry details and format as a pandas DataFrame indexed by catalog_entry uris
- **as_geojson** (*bool, Optional, Default=False*) – include catalog_entry details and format as a geojson scheme indexed by catalog_entry uris
- **update_cache** (*bool, Optional,Default=False*) – if True, update metadata cache
- **filters** (*dict, Optional, Default=None*) – filter catalog_entries by one or more of the available filters

  ---

  Note:

  available filters:

  - *bbox* (string, optional): filter catalog_entries by bounding box
  - *geom_type* (string, optional): filter catalog_entries by geom_type, i.e. point/line/polygon
  - *parameter* (string, optional): filter catalog_entries by parameter
  - *display_name* (string, optional): filter catalog_entries by display_name
  - *description* (string, optional): filter catalog_entries by description
  - *search_terms* (list, optional): filter catalog_entries by search_terms

  catalog_entries can also be filtered by any other metadata fields

  ---

- **queries** (*list, Optional, Default=None*) – list of string arguments to pass to pandas.DataFrame.query to filter the catalog_entries

Returns datasets of specified service(s), collection(s) or catalog_entry(s)

Return type datasets (list, geo-json dict or pandas.DataFrame, Default=list)

quest.api.**stage_for_download**(*uris*, *options=None*)

Apply download options before downloading

Parameters

- **uris** (*string or list, Required*) – uris of datasets to stage for download
- **options** (*dict or list of dicts, Optional, Default=None*) – options to be passed to quest.api.download function specified for each dataset

  If options is a dict, then apply same options to all datasets, else each dict in list is used for each respective dataset

Returns staged dataset uids

Return type uris (list)

quest.api.**unauthenticate_provider**(*uri*)

Un-Authenticate the user.

Parameters **uri** – uri of 'user service'

quest.api.**update_metadata**(*uris*, *display_name=None*, *description=None*, *metadata=None*, *quest_metadata=None*)

Update metadata for resource(s)

Parameters

- **uris** (*string, comma separated string, or list of strings, Required*) – list of uris to update metadata for.
- **display_name** (*string or list, Optional,Default=None*) – display name for each uri
- **description** (*string or list, Optional,Default=None*) – description for each uri
- **metadata** (*dict or list of dicts, Optional, Default=None*) – user defiend metadata
- **quest_metadata** (*dict or list of dicts, Optional, Default=None*) – metadata used by QUEST

Returns metadata of each uri keyed on uris

Return type metadata (dict)

quest.api.**update_project_metadata**(*name*, *display_name=None*, *description=None*, *metadata=None*)

Updates a project's metadata

Parameters

- **name** (*string, required*) – name of project to update
- **display_name** (*string, optional, default=None*) – new display name for the project. If None then the display name will not be modified.
- **(string, optional, default=None** (*description*) – new description for the project. If None then the description will not be modified.
- **metadata** (*dict, optional, default=None*) – new metadata dict with which to update the project's current metadata. If None then the metadata will not be modified.

Returns dictionary of updated metadata for project

Return type dict

quest.api.**update_settings**(*config={}*)

Update the settings file that is being stored in the Quest settings directory.

Notes

Only key/value pairs that are provided are updated, any other existing pairs are left unchanged or defaults are used.

Parameters **config** (*dict*) – Key/value pairs of settings that are to be updated.

Returns Updated Settings

Example

{'BASE_DIR': '/Users/dharhas/', 'CACHE_DIR': 'cache', 'PROJECTS_DIR': 'projects', 'USER_SERVICES': [], }

quest.api.**update_settings_from_file**(*filename*)

Update the settings from a new yaml file.

Notes

Only key/value pairs that are provided are updated, any other existing pairs are left unchanged or defaults are used.

Parameters **filename** (*string*) – Path to the yaml file containing the new settings.

Returns Updated settings

Example

{'BASE_DIR': '/Users/dharhas/', 'CACHE_DIR': 'cache', 'PROJECTS_DIR': 'projects', 'USER_SERVICES': [], }

quest.api.**visualize_dataset**(*dataset*, *update_cache=False*, ***kwargs*)

Visualize the dataset as a matplotlib/bokeh plot.

Check for existence of dataset on disk and call appropriate file format driver.

Parameters

- **dataset** (*string, Required*) – uri of dataset to be visualized
- **update_cache** (*bool, Optional, Default=False*) – currently unused
- **kwargs** – optional download kwargs

Returns path to the newly visualized dataset

Return type path (string)

quest.api.**get_visualization_options**(*dataset*, *fmt='json'*)

Return visualization available options for dataset.

Parameters

- **dataset** (*string, Required*) – uid of dataset
- **fmt** (*string, Required, Default='json'*) – format in which to return options

Returns options that can be specified when calling quest.api.visualize_dataset

Return type get_visualization_options (dict)

CHAPTER

SIX

DEVELOPER DOCUMENTATION

This documentation is geared toward those wanting to contribute to the Quest source code.

6.1 Table of Contents

6.1.1 Quest Design

Quest has several (sometimes conflicting) design goals. The current design aims for a practical balance between these goals.

Architectural Goals:

- Cross platform: OS X, Windows, Linux
- Needs to be easily extendable.

API Goals:

- The api should be optimized to allow ease of scripting and interactive in python
- The api should allow for use as a backend library to drive web and gui interfaces

Data Goals:

- Downloaded data should be reasonably structured, portable and usable even if you don't use Quest later
- Should allow reasonable tracking of provenance and transformations of data
- Provide mechanisms to publish/share data that has been downloaded/transformed
- Easily publish structured data as user defined services

Core Concepts

Refer to *Core Concepts*.

Settings

Quest can be configured in three ways:

1. Setting Environmental Variables
2. Passing in a python dictionary to quest.api.update_settings()
3. Reading a yaml file with quest.api.update_settings_from_file()

Any settings that are not set explicitly are given default values

Description of Settings:

Variable Name	Description	Default
QUEST_BASE_DIR	Base directory to save quest data/metadata	determined by appdirs python package
QUEST_CACHE_DIR	Location to save cached data/metadata	QUEST_BASE_DIR/cache/
QUEST_PROJECT_FILE	Name of project metadata file	quest_project.yml
QUEST_PROJECTS_INDEX_FILE	Name of projects index file listing available projects and their paths	quest_projects_index.yml
QUEST_CONFIG_FILE	Name of quest_config file that these settings are saved in	quest_config.yml
QUEST_USER_SERVICES	list of web/file uris to user defined Quest services	None

You can add any extra settings needed by a plugin here as well using the keyword:arg structure.

Projects and Collections:

- A project is a folder that has some metadata and a set of collections
- All collections in a project are saved in subdirectories of the main project folder for portability
- Only one project can be active at a time, if none is specified a project called 'default' will be created and used
- Other projects can be opened as 'local' web services and features/data 'downloaded' in to the current project
- Only one dataset (with linear progression of versions) can exist in a (collection,parameter,feature) tuple. i.e. You cannot have two temperature datasets like 2015 Temperature and 2013 Temperature in the same collection+feature. You will either need to copy the feature with a new feature_id or copy to a new collection.
- Any 'project' can be added as a user defined Quest service (either from a local/network drive or http folder). In that case, the 'project' is equivalent to a 'provider' and each 'collection' is equivalent to a 'service'
- There will be a way to convert folders of non Quest data into a user defined service by adding a quest_project.yml to the folder with appropriate metadata. These will be read-only projects.

Services:

- **There will be three types of services available (use the service_type filter in quest.api.get_service() to return a specifi**
  - geo-discrete: These are what we currently use, feature based, features have location info
  - geo-seamless: This is for seamless datasets. There is no get_features function. Instead you pass a geometric feature (bbox, line etc) to the service and the data is extracted and returned (eg. GEBCO Global Bathymetry data)
  - geo-typical: This had features, by the features do not have geometry defined. Will function the same as geo-discrete. Will need to add a tag based search option.

Parameters:

- external_name: what it is called in the service
- external_vocabulary: what the external vocab is
- vdatum: vertical datum if relevant

- long_name: display name
- standard_name: quest name, i.e air_temperature:daily:max
- vocabulary: ERDC Environmental Simulator
- units: m
- concept: air_temperature
- frequency: hourly, daily, etc
- statistic: instantaneous, mean, min, max etc

Example Directory Structure:

```
/path_to_quest_base_dir/
    cache/                              # data caches go here
    quest_config.yml                      # quest configuration settings
    quest_projects_index.yml              # list of active projects & their paths. projects do not need to be in this directory
    myproject_1/                        # example project called myproject_1
        quest_project.yml                 # project metadata
        mycollection_1/                 # example collection inside myproject_1
            quest.yml                     # collection metadata
            features.h5                 # master list of features inside collection, can also be csv, geojson
            parameters.yml              # file to keep track of available parameters, download status, versions of downloaded data etc
            temperature/                # folder for all temperature data in mycollection_1
                feature_1/              #   folder for temperature data at feature_1 (feature_1 coords & metadata are in the master features.h5)
                    66a4e39d            #       temperature datasets at feature_1
                    f974a0c1            #       these are different versions of the same dataset, the last one is the final
                    203a91e3            #       the versioning and applied filters metadata is tracked in quest_collection.yml
                feature_2/
            precipitation/
                feature_1/
                feature_3/
                feature_4/
            adh/
                feature_5/              # directory containing adh model grid defined by a polygon called feature_5
                feature_6/              # directory containing adh model grid defined by a polygon called feature_6
            timeseries/
                66a4e39d
            vitd-terrain/
            raster/

/some_other_location/myproject_2/       # another project listed in quest_projects_index.yml but not in the QUEST_BASE_DIR
    quest_project.yml
    mycollection_1/
    mycollection_2/
```

6.1.2 Writing Quest Service Plugin

Quest is designed to be extensible. It has tree types of plugins: (1) Data Service plugins, (2) Filter plugins, and (3) I/O plugins. Each of them work in a similar way, but this documentation will focus the details of the first type (Data Service plugins). Since there is not a standard interface for accessing the many web services that provide various types of data, the Quest service plugins are used as adapters that translate the specific interface used by each service to a common Quest API for searching, accessing, and downloading data. If we find a new data source that we would like to make accessible through Quest then we need to create a new service plugin for that source.

6.1.3 Data Service Plugins

Services are channels for finding and importing data into Quest. Data services are organized by provider. A provider is composed of one or more services, and all services must be part of a provider. For example, the U.S. Geological Survey (USGS) provides various data products. A subset of these products, say, the National Elevation Datasets (NED) haved been grouped into a `usgs_ned` provider. That provider has four different services that provide the various NED data products (i.e. 1 arc second, 1/3 arc second, 1/9 arc second, and Alaska 2 arc second). The process of creating a new Data Service plugin involves subclassing both the `ProviderBase` class and the `ServiceBase` class. To illustrate this process, we will provide code examples that create an example web service provider (`ExampleProvider`) that contains two services (`ExampleService1` and `ExampleService2`).

1. Provider Base Class

The *ProviderBase* class acts as the gateway to all of the services that are part of a provider. Most of the code resides in the abstract base class, so subclassing it is very simple, and involves specifying a few attributes. For example:

```
from .base import ProviderBase

class ExampleProvider(ProviderBase):
    service_base_class = None  #TODO: This will be implemented in the next step
    display_name = 'Example Web Provider'
    description = 'Example ProviderBase subclass for Quest'
    organization_name = 'Example Data Provider Organization'
    organization_abbr = 'EDPO'
```

This is all that is required to subclass the `ProviderBase`. As you will notice the attribute `service_base_class` was left as `None`. This attribute refers to a base class that is the parent of all of the services that belong to this provider. The `ProviderBase` will find all of the subclasses of the class specified by `service_base_class` and register them as services of the provider. Therefore the next step is to create a *2. Service Base Class*.

2. Service Base Class

A data service plugin must subclass the `ServiceBase` class (or one of it's subclasses, see *Specialized Service Base Subclasses*) to act as the base class for all services in the plugin. This `ServiceBase` subclass is registered in the provider as the *service_base_class* attribute. As an example we will create an `ExampleServiceBase` class that subclasses the `ServiceBase` class:

```
from .base import ProviderBase, ServiceBase

class ExampleServiceBase(ServiceBase):
    service_name = None
    display_name = None
    description = None
```

(continues on next page)

(continued from previous page)

```
    service_type = None
    unmapped_parameters_available = None
    geom_type = None
    datatype = None
    geographical_areas = None
    bounding_boxes = None
    smtk_template = None
    _parameter_map = None

    def download(self, feature, file_path, dataset, **params):
        pass  #TODO: This will be implemented later

    def get_features(self, **kwargs):
        pass  #TODO: This will be implemented later

class ExampleProvider(ProviderBase):
   service_base_class = ExampleServiceBase
   ...
```

Note: The `ExampleServiceBase` class needed to be defined above the `ExampleProvider` class so we could reference it to assign the `service_base_class` attribute in the `ExampleProvider`.

The content of `ExampleServiceBase` has not yet been fully implemented. The above example simply illustrates the structure. All of the attributes and methods shown in the `ExampleServiceBase` will need to be implemented either in this class directly or in the services that subclass this base class. The specifics of how this are done will be different for each plugin, but the next step, *3. Service Classes* will demonstrate one way to do it.

Specialized Service Base Subclasses

There are a couple of special cases that apply to services from various providers. To allow all of these services to use the same codebase a couple of other base classes are available that can be used in place of the `ServiceBase`.

TimePeriodServiceBase

This base class simply adds two parameters, a *start* and *end* date to represent the time period for the data being requested (see *d. Specify the Download Options*).

SingleFileServiceBase

This base class implements the *download* method for services where there is simply a download url that links to a single zip file that contains the data.

3. Service Classes

After a `ServiceBase` subclass has been created (in our example this is the `ExampleServiceBase`) then the next step is to create classes for each specific service. While the specifics of this step can vary significantly between plugins, the overall structure and process are similar and will be broken down in to several sub-steps:

- *a. Required Service Class Attributes*

- *b. Implement the get_features Method*
- *c. Implement the download Method*
- *d. Specify the Download Options*

Continuing the example from above we will create two service classes that each subclass the `ExampleServiceBase`. We'll first focus on assigning all of the required class attributes.

a. Required Service Class Attributes

- `service_name` (String): A unique identifier for the service. It should contain only alpha-numeric characters or _ or −. There should be no spaces.
- `display_name` (String): A displayable version of the service name (may contain spaces) for use in GUIs.
- `description` (String): A brief description of the service that will be available in the service's metadata.
- `service_type` (String): A keyword that indicates the type of data that the service provides. Must be one of *geo-discrete*, *geo-seamless* or *geo-typical*. (# TODO: provide link to description of service types in the docs)
- `unmapped_parameters_available` (Bool): Whether or not additional parameters are available from the service other than those that are listed in the `_parameter_map`.
- `geom_type` (String): Describes what type of geometry represents the locations of the data (for *geo-discrete* services only). Must be *Point*, *Line*, *Polygon*. Leave as `None` for service of type other than *geo-discrete*.
- `datatype` (String): Represents the type of data that is accessible from the service. Must be *timeseries*, *raster*, or *other*.
- `geographical_areas` (List): A list of descriptive words that represent the areas where data is available (e.g. *['North America', 'Europe']*). Should be left as `None` for *geo-typical* service types.
- `bounding_boxes` (List): A list of bounding boxes represented as tuples in the form (x-min, y-min, x-max, y-max). For example *[(-180, -90, 180, 90)]*.
- `smtk_template` (String): The name of the SMTK template file that describes the download options for the service.
- `_parameter_map` (Dict): A mapping of parameters as they are called by the service, to the controlled vocabulary parameter names in Quest.

In some cases the attributes will be the same for both services, so they can be assigned in the `ExampleServiceBase` class. The rest of the attributes, that are different between the two services, will be assigned in the service classes themselves:

```
from .base import ProviderBase, ServiceBase

class ExampleServiceBase(ServiceBase):
    service_name = None
    display_name = None
    description = None
    service_type = 'geo-discrete'
    unmapped_parameters_available = False
    geom_type = 'Point'
    datatype = 'timeseries'
    geographical_areas = ['Worldwide']
    bounding_boxes = [
        [-180, -90, 180, 90],
    ]
    smtk_template = None
```

(continues on next page)

(continued from previous page)

```
    def get_features(self, **kwargs):
        pass  #TODO: This will be implemented later

    def download(self, feature, file_path, dataset, **params):
        pass  #TODO: This will be implemented later


class ExampleService1(ExampleServiceBase):
    service_name = 'example-1'
    display_name = 'Example Service 1'
    description = 'First example service'

    _parameter_map = {}


class ExampleService2(ExampleServiceBase):
    service_name = 'example-2'
    display_name = 'Example Service 2'
    description = 'Second example service'

    _parameter_map = {}

class ExampleProvider(ProviderBase):
   service_base_class = ExampleServiceBase
   ...
```

b. Implement the `get_features` Method

The purpose of the `get_features` method is to extract key metadata from the service that describes what data is available from that service. For *geo-discrete* services this would include a list of locations where the service has data in addition to other key metadata at each location. The return value for `get_features` should be a Pandas DataFrame indexed by a unique id (known as the *service_id*) with the following columns:

- *display_name*: (will be set to *service_id* if not provided)
- *description*: (will be set to '' if not provided)
- *service_id*: a unique id that is used by the web service to identify the data

For *geo-discrete* services the DataFrame must also include a representation of the features' geometry. Any of the following options are valid ways to specify the geometry:

1) *geometry*: a geojson string or Shapely object
2) *latitude* and *longitude*: two columns with the decimal degree coordinates of a point
3) *geometry_type*, *latitudes*, and *longitudes*: *Point*, *Line*, or *Polygon* with a list of coordinates
4) *bbox*: tuple with order (lon min, lat min, lon max, lat max)

All other fields that the DataFrame contains will be accumulated into a `dict` and placed in a column called *metadata*.

Similar to the attributes the `get_features` method may be implemented in the service classes (e.g. `ExampleService1` and `ExampleService2`), or in the base class (e.g. `ExampleServiceBase`), or some combination of both.

c. Implement the `download` Method

The `download` method is responsible for retrieving the data from the data source using the specified download options, save it to disk, and then return a dictionary of key metadata. The download method should accept several arguments:

- *feature*: the service_id for the feature that is associated with the data to be downloaded
- *file_path*: the path to the directory on disk where Quest expects the data to be written
- *dataset*: the Quest dataset id associated with the data to be downloaded
- ***params*: key-word arguments for the dataset options

After downloading the data and saving it to disk, this method should return a dictionary ith the following keys:

- *metadata*: any metadata that was returned by the data source when it was downloaded in the form of a `dict`
- *file_path*: the final file path (including the filename) where the data file was writen
- *file_format*: the format that the file was written in (to be used to determine which I/O plugin to use to read the file)
- *datatype*: a string representing the type of data. Must be *timeseries*, *raster*, or *other*.
- *parameter*: a string representing the parameter of the data
- *unit*: a string representing the units of the data

d. Specify the Download Options

Data sources's APIs often allow various options to be specified to determine what data to download, what format it should be in, etc.

The download options that are needed for each service are defined using the Python library Param. This library enables parameters to have features like type and range checking, documentation strings, default values, etc. Refer to the Param documentation for more information.

6.1.4 Testing

Quest has an expanding test suite the testing framework from pytest.

Running Tests

To run the tests you will need to install the pytest Python package after activating your environment:

```
(quest) $ conda install pytest
```

Once pytest is installed in your environment you can execute the tests from the command line with the following command (assuming your working directory is the quest source code directory):

```
(quest) $ pytest test
```

The first time the tests are run quest will build a complete cache of all of the feature metadata for each service. This can take 5 or 6 minutes. This is only done the first time the tests are run (or when a flag is passed to update the cache). The tests will run much faster after the first time.

The tests are configured to run through both a Python interpreter and through an RPC server. This lengthens the time it takes to run the tests, but provides more coverage. If you are interested in only running one set of test or the other, this can be achived by passing in *Custom Test Options*.

Custom Test Options

Several custom options have been configured to allow several subsets of the tests to be run.

- *–skip-slow*: Any tests that have been marked as slow (e.g. the *get_features* tests) will not be run.
- *–update-cache*: Triggers the feature metadata for each service to be re-downloaded (this process takes 5 or 6 minutes).

For example, to run most of the tests very quickly you can run:

```
(quest) $ pytest test --skip-slow
```

This will will give you the most bang for you buck, running the majority of the tests in just several seconds.

To get the most coverage you should run:

```
(quest) $ pytest test --update-cache
```

This will test all of the services by regenerating the cache and will run the complete set of tests. This process can take around 10 minutes.

Adding Tests

The Quest testing framework makes extensive use of pytest fixtures. Fixtures provide a very flexible and powerful way to provide the correct baseline configuration for each test, and for running the same test with multiple configurations. The heart of the testing configuration is determined by the fixtures defined in *conftest.py*.

CHAPTER

SEVEN

GLOSSARY

Catalog

Catalogs see *Catalogs*

Catalog Entry

Catalog Entries see *Catalog Entries*

Collection

Collections see *Collections*

Dataset

Datasets see *Datasets*

Project

Projects see *Projects*

Provider

Providers see *Providers*

Service

Services see *Services*

Tool

Tools see *Tools*

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

q

quest.api, 19

INDEX

A

add_datasets() (*in module quest.api*), 19
add_project() (*in module quest.api*), 19
add_user_provider() (*in module quest.api*), 19
authenticate_provider() (*in module quest.api*), 19

C

cancel_tasks() (*in module quest.api*), 19
Catalog, **45**
Catalog Entries, **45**
Catalog Entry, **45**
Catalogs, **45**
Collection, **45**
Collections, **45**
copy() (*in module quest.api*), 20

D

Dataset, **45**
Datasets, **45**
delete() (*in module quest.api*), 20
delete_project() (*in module quest.api*), 20
delete_user_provider() (*in module quest.api*), 20
download() (*in module quest.api*), 20
download_datasets() (*in module quest.api*), 20

G

get_active_project() (*in module quest.api*), 21
get_api_version() (*in module quest.api*), 21
get_auth_status() (*in module quest.api*), 21
get_collections() (*in module quest.api*), 21
get_data() (*in module quest.api*), 21
get_datasets() (*in module quest.api*), 22
get_download_options() (*in module quest.api*), 21
get_mapped_parameters() (*in module quest.api*), 22
get_metadata() (*in module quest.api*), 23
get_parameters() (*in module quest.api*), 23
get_pending_tasks() (*in module quest.api*), 23
get_projects() (*in module quest.api*), 23
get_providers() (*in module quest.api*), 23
get_publish_options() (*in module quest.api*), 29
get_publishers() (*in module quest.api*), 23
get_quest_version() (*in module quest.api*), 24
get_seamless_data() (*in module quest.api*), 24
get_services() (*in module quest.api*), 25
get_settings() (*in module quest.api*), 25
get_tags() (*in module quest.api*), 25
get_task() (*in module quest.api*), 25
get_tasks() (*in module quest.api*), 26
get_tool_options() (*in module quest.api*), 27
get_tools() (*in module quest.api*), 26
get_visualization_options() (*in module quest.api*), 32

M

move() (*in module quest.api*), 27

N

new_catalog_entry() (*in module quest.api*), 27
new_collection() (*in module quest.api*), 27
new_dataset() (*in module quest.api*), 28
new_parameter() (*in module quest.api*), 28
new_project() (*in module quest.api*), 28

O

open_dataset() (*in module quest.api*), 28

P

Project, **45**
Projects, **45**
Provider, **45**
Providers, **45**
publish() (*in module quest.api*), 29

Q

quest.api (*module*), 19

R

remove_project() (*in module quest.api*), 29
remove_tasks() (*in module quest.api*), 29

run_tool() (*in module quest.api*), 29

S

save_settings() (*in module quest.api*), 30
search_catalog() (*in module quest.api*), 30
Service, **45**
Services, **45**
set_active_project() (*in module quest.api*), 30
stage_for_download() (*in module quest.api*), 31

T

Tool, **45**
Tools, **45**

U

unauthenticate_provider() (*in module quest.api*), 31
update_metadata() (*in module quest.api*), 31
update_project_metadata() (*in module quest.api*), 31
update_settings() (*in module quest.api*), 32
update_settings_from_file() (*in module quest.api*), 32

V

visualize_dataset() (*in module quest.api*), 32