d6tpipe Documentation

Release 0.1.1

nn

Sep 22, 2020

Contents

d6tpipe is a python library which makes it easier to manage and exchange data files for data science projects. It's like git for data! But better because you can include it in your data science code.

For more details see https://github.com/d6t/d6tpipe

CHAPTER 1

Installation

Follow github instructions https://github.com/d6t/d6tpipe#installation

CHAPTER 2

Quickstart

See *Quickstart*

CHAPTER 3

Command line help

The command line tool is the fastest way to pull/push files.

```
d6tpipe --help
```

CHAPTER 4

User Guide

d6tpipe is easy to get started but has many advanced features. The core concepts are covered first, followed by advanced features that you can come back to once you are familiar with d6tpipe.

4.1 Quickstart

4.1.1 Installation

Follow github instructions https://github.com/d6t/d6tpipe#installation

4.1.2 First-time setup and registration

1. Sign up at https://pipe.databolt.tech/gui/user-signup/
2. Get API token at http://pipe.databolt.tech/gui/api-access
3. Set up d6tpipe to use token

Using command line

```
d6tpipe profiles token set --token <your-token>
```

Or using python

```
import d6tpipe
# cloud repo API token
api = d6tpipe.api.APIClient()
api.setToken('<your-token>') # DONT SHARE YOUR TOKEN! Do not save in code, just run it once
```

Run this ONCE. You do NOT to have to run this every time you use d6tpipe

Advanced: create profile with custom filerepo path

```
import d6tpipe
d6tpipe.api.ConfigManager(profile='profile-name').init({'filerepo':'/path/to/data'})
api = d6tpipe.api.APIClient(profile='profile-name')
api.setToken('<your-token>') # DONT SHARE YOUR TOKEN! Do not save in code, just run␣
↪it once
```

See *Config* and *Connect* for details.

4.1.3 Where are files saved?

See *Config*

4.1.4 Pull files from remote to local

Using the command line tool is the fastest way to pull files.

```
# list data pipes you have access to
d6tpipe list remote

# pull files from a data pipe
d6tpipe pull --pipe intro-stat-learning --preview
d6tpipe pull --pipe intro-stat-learning
d6tpipe --help
```

To pull files using python, you connect to a data pipe. A data pipe lets you manage remote and local data files.

```
import d6tpipe
api = d6tpipe.api.APIClient()

# list data pipes you have access to
api.list_pipes()

# pull files from a data pipe
pipe = d6tpipe.Pipe(api, 'intro-stat-learning') # connect to a data pipe
pipe.pull_preview() # preview files and size
pipe.pull() # download all data with just one command

# advanced: show location and credentials
pipe.remote_prefix
pipe._get_credentials()
```

See *Pull Files* for details.

4.1.5 Access and read local files

Remote files that are pulled get stored in a central local file directory.

```
# show local files
pipe.dirpath # where files are stored
pipe.files()

# read a file into pandas
import pandas as pd
```

(continues on next page)

(continued from previous page)

```
df = pd.read_csv(pipe.dirpath/'Advertising.csv')
print(df.head())
```

See *Read Files* for details.

4.1.6 Process files

You now have a lot of powerful functions to easily manage all your files from a central location across multiple projects.

```
# use schema to quickly load data
df = pd.read_csv(pipe.dirpath / 'Advertising.csv', **pipe.schema['pandas'])
print(df.head())

# read multiple files into dask
import dask.dataframe as dd
files = pipe.filepaths(include='Advertising*.csv')
ddf = dd.read_csv(files, **pipe.schema['dask'])
print(ddf.head())

# open most recent CSV
df = pd.read_csv(pipe.files(sortby='mod')[-1])

# save data to local files
df.to_csv(pipe.dirpath/'new.csv')
```

See *Process Files* for details.

4.1.7 Advanced Topics

This covers pushing files and creating your own remote file storage and data pipes.

Write Local Files and Push to Remote

You can easily save new files to the pipe. You can also push files from local to remote if you have write access or manage your own pipes.

```
# create some new data
import sklearn.preprocessing
df_scaled = df.apply(lambda x: sklearn.preprocessing.scale(x))

# conveniently save files in a central repo
df_scaled.to_csv(pipe.dirpath/'Advertising-scaled.csv') # pipe.dirpath points to␣
↪local pipe folder

# alternatively, import another folder
pipe.import_dir('/some/folder/')

# list files in local directory
pipe.scan_local()

# upload files - just one command!
```

(continues on next page)

(continued from previous page)

```
pipe.push_preview() # preview files and size
pipe.push() # execute
```

See *Push* for details.

Alternatively you can use the command line tool.

```
d6tpipe push --pipe intro-stat-learning --preview
d6tpipe push --pipe intro-stat-learning
d6tpipe --help
```

Register and administer pipes

You can register your own pipes that point to your own remote data storage. d6tpipe has managed remotes which makes it very easy for you to set up and manage professional remote data file storage.

```
import d6tpipe
api = d6tpipe.api.APIClient()

# managed remote file stores can be created quickly with just one command
d6tpipe.upsert_pipe(api, {'name': 'your-pipe'})
```

See *Pipes* for details. For creating self-hosted remotes, see *Advanced Pipes*.

Share pipes

After you've registered a pipe, you can give others access to the remote data. By default only you have access so to share it with others you have to grant them access.

```
import d6tpipe
api = d6tpipe.api.APIClient()

# give another user access
settings = {"username":"another-user","role":"read"} # read, write, admin
d6tpipe.upsert_permissions(api, 'your-pipe', settings)

# make data repo public
settings = {"username":"public","role":"read"}
d6tpipe.upsert_permissions(api, 'your-pipe', settings)
```

See *Permissions* for details.

4.2 Command Line Interface

Key functions are implemented in the CLI. For full functionality you will need to use the python library.

```
d6tpipe --help
```

4.3 Configuring d6tpipe

4.3.1 Show Configuration

The config by default this is in `~/d6tpipe/cfg.json`. You can also load it to show your config. See advanced section on what the config options are.

```
d6tpipe.api.ConfigManager().load()
```

4.3.2 Where are local files stored?

d6tpipe stores all files in a central location to make it easy for you to reference data files across multiple projects. Subfolders will be created for each pipe.

```
# show local file repo AFTER init
api.filerepo
```

4.3.3 Manage Local File Storage

You can change where files are stored locally.

```
# option 1: set custom path BEFORE init
d6tpipe.api.ConfigManager().init({'filerepo':'/some/path/'})

# option 2: move file repo AFTER init
api = d6tpipe.api.APILocal()
api.move_repo('/new/path')
print(api.filerepo)

# option 3: manually move file repo and update config
d6tpipe.api.ConfigManager().update({'filerepo':'/some/path/'})
```

4.3.4 Advanced Topics

Customize Init Options

Optionally, you can pass in a config dict. Your options are:

- `filerepo`: path to where files are stored
- `server`: repo API server
- `token`: auth token for REST API server
- `key`: encryption key for encrypting credentials

```
# example: pass file repo path
d6tpipe.api.ConfigManager().init({'filerepo':'/some/path/'})
```

Update an Existing Config

You can change config options by passing the settings you want to update.

```
# example: update REST token and username
d6tpipe.api.ConfigManager().update({'token':token})
d6tpipe.api.ConfigManager().update({'username':username})
```

NB: Don't use config update to change settings for remotes and pipes.

Using Multiple Profiles

d6tpipe supports the use of profiles so you can use different settings. Local files in one profile are completely separate from files in another profile.

```
# show profiles
d6tpipe.api.list_profiles()

# make profiles
d6tpipe.api.ConfigManager(profile='user2').init()
d6tpipe.api.ConfigManager(profile='projectA').init({'filerepo':'/some/path/'})
d6tpipe.api.ConfigManager(profile='projectB').init({'filerepo':'/another/path/'})
d6tpipe.api.ConfigManager(profile='cloud').init({'server':'http://api.databolt.tech'})
d6tpipe.api.ConfigManager(profile='onprem').init({'server':'http://yourip'})

# connect using a profile name
api = d6tpipe.api.APIClient(profile='onprem')
```

4.4 Connect to repo API

4.4.1 What is the cloud repo API?

The repo API stores details about pipes needed to pull/push data. There are alternatives to the cloud API, covered in advanced sections, but the cloud repo API is the best way to get started.

4.4.2 Register user and login

To use the cloud repo API, you need to register which you can do from inside python. See *registration*

Now that you are registered you can *pull files*.

4.4.3 Setting Proxy

If you are behind a proxy, you may have to set your proxy to connect to the repo API.

```
import os
cfg_proxy = "http://yourip:port"
os.environ["http_proxy"] = cfg_proxy; os.environ["HTTP_PROXY"] = cfg_proxy;
os.environ["https_proxy"] = cfg_proxy; os.environ["HTTPS_PROXY"] = cfg_proxy;
```

4.4.4 Advanced Topics

Managing your cloud API token

Run the below if you forgot your token or need to reset.

```
# retrieve token
api.forgotToken('your-username','password')

# reset token
api = d6tpipe.api.APIClient(token=None)
api.register('your-username','your@email.com','password')
```

Local Mode

The best way to get started is to use the cloud API. As an alternative to the cloud API, you can use d6tpipe in local mode which stores all details locally. That is fast and secure, but limits functionality.

```
import d6tpipe
api = d6tpipe.api.APILocal() # local mode
```

Local mode requires you to manage more settings on your own (see below). Particularly you need to manage protocols and remote directories on your own. Additionally, there is no pipe setting inheritance.

```
# s3
settings['protocol']='s3'
settings['options']={'remotepath': 's3://bucketname/foldername'}
d6tpipe.upsert_pipe(api, settings)
settings['parent']='pipe-parent' # won't do anything
```

Local vs Server Mode

The local mode stores everything locally so nothing ends up in the cloud. While that is fast and secure, it limits functionality. Only the server can:

- automatically manage remote paths
- provide pipe inheritance
- manage permissions
- share data across teams and organizations
- remotely scan for file changes and centrally cache results
- regularly check for file changes on a schedule
- remotely mirror datasets, eg from ftp to S3

Overall the way you interface with d6tpipe very similar between the two modes so you can easily switch between them. So you can start in local mode and then switch to server mode to take advantage of advanced features.

Onprem repo API

You can deploy an onprem repo API to take advance of server functionality without using the cloud server, contact <support@databolt.tech> for details.

4.5 Pull Files from Remote to Local

4.5.1 Connecting to Pipes

To pull and push files, you first connect to a data pipe. A data pipe lets you manage remote and local data files.

```
import d6tpipe
api = d6tpipe.api.APIClient()
api.list_pipes() # show available pipes

pipe = d6tpipe.Pipe(api, 'pipe-name') # connect to a pipe
```

4.5.2 Show remote files

To show files in the remote storage, run `pipe.scan_remote()`.

```
pipe = d6tpipe.Pipe(api, 'pipe-name')
pipe.scan_remote() # show remote files
```

4.5.3 Pulling Files to Local

Pulling files will download files from the remote data repo to the local data repo. Typically you have to write a lot of code to download files and sync remote data sources. With d6tstack you can sync pull with just a few lines of python.

```
pipe = d6tpipe.pipe.Pipe(api, 'pipe-name')
pipe.pull_preview() # preview
pipe.pull() # execute
```

Your files are now stored locally in a central location and conveniently accessible. See *Accessing Pipe Files* to learn how to use files after you have pulled them.

4.5.4 Which files are pulled?

Only files that you don't have or that were modified are downloaded. You can manually control which files are downloaded or force download individual files, see advanced topics.

4.5.5 Advanced Topics

Pull Modes

You can control which files are pulled/pushed

- `default`: modified and new files
- `new`: new files only
- `mod`: modified files only
- `all`: all files, good for resetting a pipe

```
pipe = d6tpipe.pipe.Pipe(api, 'test', mode='all') # set mode
pipe.pull() # pull all files
pipe.setmode('all') # dynamically changing mode
```

Useful Pipe Operations

Below is a list of useful functions. See the reference modindex for details.

```
# advanced pull options
pipe.pull(['a.csv']) # force pull on selected files
pipe.pull(include='*.csv',exclude='private*.xlsx') # apply file filters

# other useful operations
api.list_local_pipes() # list pipes pulled
pipe.files() # show synced files
pipe.scan_remote() # show files in remote
pipe.scan_remote(sortby='modified_at') # sorted by modified date
pipe.is_synced() # any changes?
pipe.remove_orphans() # delete orphan files
pipe.delete_files() # reset local repo
```

Using Multipe Pipes

If you work with multiple data sources, you can connect to multiple pipes.

```
pipe2 = d6tpipe.Pipe(api, 'another-pipe-name') # connect to multiple

# todo: how to sync pipe1 files to pipe2?
```

4.6 Read and Write Local Files

4.6.1 Show Local Files

Files that are pulled from the remote get stored in a central local file directory.

```
pipe.dirpath # where local files are stored
pipe.files() # show synced local files
pipe.scan_local() # show all files in local storage
```

4.6.2 Read Local Files

You now have a lot of powerful functions to easily access all your files from a central location across multiple projects.

```
import pandas as pd

# open a file by name
df = pd.read_csv(pipe.dirpath/'test.csv')

# open most recent file
df = pd.read_csv(pipe.files(sortby='mod')[-1])
```

Applying File Filters

You can include file filters to find specific files.

```
files = pipe.files(include='data*.csv')
files = pipe.files(exclude='*.xls|*.xlsx')
```

Read multiple files

You can read multiple files at once.

```
# read multiple files into dask
import dask.dataframe as dd
files = pipe.filepaths(include='Advertising*.csv')
ddf = dd.read_csv(files, **pipe.schema['dask'])
print(ddf.head())
```

Quickly accessing local files without connecting to API

If you have pulled files and don't always want to or can't connect to the repo API, you can use *PipeLocal()* to access your local files.

```
pipe = d6tpipe.PipeLocal('your-pipe')
```

4.6.3 Write Local Files

Write Processed Data to Pipe Directory

To keep all data in once place, it's best to save your processes data back to the pipe directory. That also makes it easier to *push files* later on if you choose to do so.

```
# save data to pipe
df.to_csv(pipe.dirpath/'new.csv')
df.to_csv(pipe.dirpath/'subdir/new.csv')
```

Delete Files

```
pipe.delete_files_local() # delete all local files
pipe.delete_files_remote() # delete all remote files
pipe.delete_files(files=['a.csv']) # delete a file locally and remotely
pipe.reset() # reset local repo: delete all files and download

# remove orphan files
pipe.remove_orphans('local') # remove local orphans
pipe.remove_orphans('remote') # remove remote orphans
pipe.remove_orphans('both') # remove all orphans
```

Reset Local Files

```
pipe.reset() # will force pull all files
pipe.reset(delete=True) # delete all local files before pull
```

4.7 Push Files from Local to Remote

4.7.1 Adding Local Files

To add data to a pipe, you need to add files to the central pipe directory at `pipe.dir`. d6tpipe makes it easy to add new data with a set of convenience functions.

```
pipe = d6tpipe.pipe.Pipe(api, 'pipe-name')

# save new data directly to pipe
df.to_csv(pipe.dirpath/'new.csv')

# import files from another folder
pipe.import_dir('/some/folder'))

# import files a list of files
pipe.import_files(glob.iglob('folder/**/*.csv'), move=True)
```

You can also use any other (manual) methods to copy files to the pipe directory `pipe.dir`.

4.7.2 Pushing Data

After you've added data to a pipe (see above), pushing data is just as simple as pulling data.

```
pipe.push_preview() # preview
pipe.push() # execute
```

For this to work, you do need to have *write permissions* to the pipe or *registered your own pipe*.

Expired Token

If you might get this error: `ClientError: An error occurred (ExpiredToken) when calling the PutObject operation: The provided token has expired.`. For security reasons you receive short-term credentials. You can force renewal

```
pipe._reset_credentials()
```

4.7.3 Customize Push

You can manually control what gets pushed.

```
# advanced push options
pipe.push(['a.csv']) # force push/pull on selected files
pipe.push(include='*.csv',exclude='backup*.csv') # apply file filters
```

4.8 Register and Administer Pipes

4.8.1 Why register pipes?

To have a pipe point to a remote data storage that you control, you have to register a pipe. Normally creating remote data storage is a complex task. But with d6tpipe it is exceptionally quick by using managed pipes which automatically take care of storage, authentication, permissioning etc, see *security* for details. To register self-hosted pipes, see *Advanced Pipes*.

4.8.2 Register Managed Pipes

To register a new pipe you need pipe settings that specify how remote data is stored and returned. The minimum requirement is the name which creates a free remote data repo. [todo: make d6tpipe default protocol]

```
import d6tpipe
api = d6tpipe.api.APIClient()

response, data = d6tpipe.upsert_pipe(api, {'name': 'your-pipe'})
```

4.8.3 Customizing Pipe Settings

Pipes not only store data but also control how data is stored, returned and provide additional meta data.

Settings Parameters

- `name` (str): unique name
- `protocol` (str): storage protocol (`d6tfree`,‘‘s3‘‘,‘‘ftp‘‘,‘‘sftp‘‘)
- **`options` (json):**
  - `dir` (str): read/write from/to this subdirectory (auto created)
  - `include` (str): only include files with this pattern, eg `*.csv` or multiple with `*.csv|*.xls`
  - `exclude` (str): exclude files with this pattern
- `schema` (json): see *Schema*

4.8.4 Pipe Setting Inheritance

You can and should have multipe pipes that connect to the same underlying remote file storage but return different files. Why? Say you are a data vendor, you can create different pipes for different subscriptions without having to individually manage them. Say the vendors data files are:

```
dataA\\monthly*.csv
dataA\\daily*.csv
dataB\\reports*.xlsx
```

Those are 3 different datasets, so you should define 3 separate pipes: `vendor-monthly`, `vendor-daily`, `vendor-reports`. To avoid having to specify the same settings multiple times, you can create a parent pipe `vendor-parent` from which the child pipes can inherit settings.

```
settings_parent = {
    'name': 'vendor-parent',
    'protocol': 'd6tfree'
}
settings_monthly = {
    'name': 'vendor-monthly',
    'parent': 'vendor-parent',
    'options': {'dir':'dataA', 'include':'monthly*.csv'}
}
settings_daily = {
    'name': 'vendor-daily',
    'parent': 'vendor-parent',
    'options': {'dir':'dataA', 'include':'daily*.csv'}
}
settings_reports = {
    'name': 'vendor-reports',
    'parent': 'vendor-parent',
    'options': {'dir':'dataB', 'include':'reports*.xlsx'}
}
```

This way the vendor can push files to `vendor-parent` and then grant clients individual access to `vendor-monthly`, `vendor-daily`, `vendor-reports` based on which product they have subscribed to.

4.8.5 Updating Pipe Settings

Here is how you update an existing pipe with more advanced settings. This will either add the setting if it didn't exist before or overwrite the setting if it existed.

```
settings = \
{
    'name': 'pipe-name',
    'remote': 'remote-name',
    'options': {
        'dir': 'some/folder',
        'include': '*.csv|*.xls',
        'exclude': 'backup*.csv|backup*.xls'
    },
    'schema': {
        'pandas': {
            'sep': ',',
            'encoding': 'utf8'
        }
    }
}

# update an existing pipe with new settings
response, data = d6tpipe.upsert_pipe(api, settings)
```

4.8.6 Data Schemas

When creating pipes you can add schema information see *Schema*

4.8.7 Administer Pipes with repo API

You can run any CRUD operations you can normally run on any REST API.

```
# listing pipes
api.list_pipes() # names_only=False shows all details

# CRUD
response, data = api.cnxn.pipes.post(request_body=settings)
response, data = api.cnxn.pipes._('pipe-name').get()
response, data = api.cnxn.pipes._('pipe-name').put(request_body=new_settings)
response, data = api.cnxn.pipes._('pipe-name').patch(request_body=new_settings)
response, data = api.cnxn.pipes._('pipe-name').delete()

# using pipe object
response, data = pipe.cnxnpipe.get()
response, data = pipe.cnxnpipe.put(request_body=all_settings)
response, data = pipe.cnxnpipe.patch(request_body=mod_settings)
response, data = pipe.cnxnpipe.delete()
```

4.9 Sharing Remote Data Storage

4.9.1 Managing Remote Access Permissions

Initially only you will be able to access the data. To share the data with others you will have to grant permissions. If you give access to a parent pipe, users will have access to all child pipes.

```
# give another user access
settings = {"username":"another-user","role":"read"} # read, write, admin
d6tpipe.upsert_permissions(api, 'pipe-name', settings)

# make data repo public
settings = {"username":"public","role":"read"}
d6tpipe.upsert_permissions(api, 'remote-name', settings)

# view permissions (owner only)
pipe.cnxnpipe.permissions.get()
```

4.9.2 Access Roles

You can specify what level of access a user has to your pipe.

- **`role` (str):**
  - `read` (str): can read data from repo but not write
  - `write` (str): can write data as well as read
  - `admin` (str): can change settings in addition to read and write

4.9.3 Invite User with email

If users are not registered with d6tpipe or you only know their email, you can grant access via email.

```
# give another user access
settings = {"email":"name@client.com","role":"read"} # read, write, admin
d6tpipe.upsert_permissions(api, 'pipe-name', settings)
```

4.10 Data Security

4.10.1 Is my data secure?

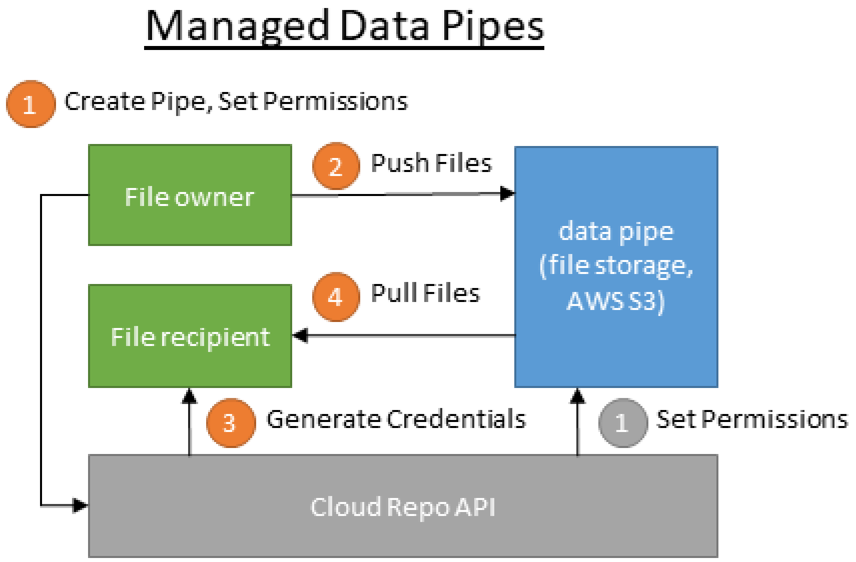
d6tpipe is designed such that only you have access to your pipes and data files. You explicity have to grant access for someone else to see your files.

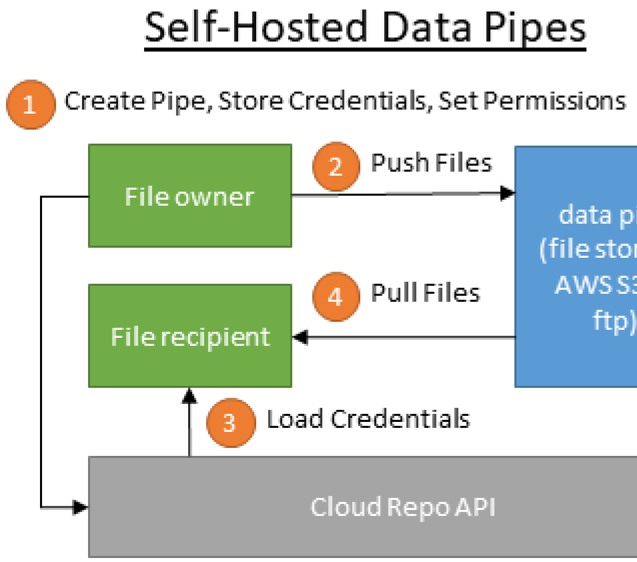
4.10.2 Where are my files stored and who has access to them?

If you are using *managed remotes* they will be stored in a DataBolt AWS S3 bucket where permissions and credentials are managed on your behalf. Again, permissions are managed such that only you have access, unless you grant access. Contact <support@databolt.tech> for details and any concerns.

If you are using *self-hosted remotes* you are in charge of where your data is stored and who has access to it. The repo API will store access credentials on behalf of authorized users. Those credentials can be encrypted, *see details*.

Here is a high-level schematic overview of the system architecture.





4.10.3 How do permissions work for managed remotes?

When users connect to a managed remote, they will receive short-term access credentials. Those temporary credentials are valid for a period of 12h. They are automatically renewed if the user still has access.

For **shared private remotes** the user has to be registered and authenticated with the repo API. **shared public remotes** the user can be anonymous.

4.10.4 Privacy Policy

See https://www.databolt.tech/index-terms.html#privacy

4.10.5 I don't trust any of that!

Use local mode or onprem repo API. See *Connect* for details.

4.11 Manage Data Schemas

4.11.1 Create Schema

`schema` settings takes any data so you can use that in a variety of ways to pass parameters to the reader. This makes it easier for the data recipient to process your data files.

```
settings = \
{
    'schema': {
        'pandas': {
            'sep': ',',
            'encoding': 'utf8'
        },
        'dask': {
            'sep': ',',
            'encoding': 'utf8'
        },
        'xls': {
            'pandas': {
                'sheet_name':'Sheet1'
            }
        }
    }
}

pipe.update_settings(settings) # update settings
```

The flexibility is good but you might want to consider adhering to metadata specifications such as https://frictionlessdata.io/specs/.

4.11.2 Using schema

You can pass `schema` information for downstream processing.

```
# show schema
print(pipe.schema)

# use schema
df = pd.read_csv(pipe.dirpath/'test.csv', **pipe.schema['pandas'])
df = dd.read_csv(pipe.dirpath/'test.csv', **pipe.schema['dask'])
df = pd.read_excel(pipe.dirpath/'others.xlsx', **pipe.schema['xls']['pandas'])
```

4.12 Manage Data Documentation

Missing data documentation is one of the most frustrating issues when exploring new dataset. With d6tpipe you can easily include data documentation and make the data consumers happy.

4.12.1 Use GUI

The databolt GUI allows you to create and publish dataset documentation including the documents listed below. Log in to GUI, select a pipe and then documentation. You can access the GUI at http://pipe.databolt.tech/gui/

4.12.2 Readme

Include a `README.md` file which will automatically be displayed when the dataset is pulled for the first time.

4.12.3 License

Include a `LICENSE.md` file which will automatically be displayed when the dataset is pulled for the first time.

4.12.4 Tearsheet

Data vendors can use tearsheet templates to succinctly describe their datasets, see https://github.com/siia-fisd/altdata-council/blob/master/documentation/tearsheet.md

Example tearsheet created in GUI http://pipe.databolt.tech/gui/docs/038531533749f46fb938db524da7cd96/

4.13 Accessing Premium Features

To request access premium features, visit https://pipe.databolt.tech/gui/request-premium/

4.14 Advanced: Self-hosted Remotes

4.14.1 Register Self-hosted Remotes

You can push/pull from your own S3 and (s)ftp resources. The repo API stores all the neccessary details so the data consumer does not have to make any changes if you make any changes to the remote storage.

```
settings = \
{
    'name': 'pipe-name',
    'protocol': 's3',
    'location': 'bucket-name',
    'options': {
        'remotepath': 's3://bucket-name/'}
    'credentials' : {
        'aws_access_key_id': 'AAA',
        'aws_secret_access_key': 'BBB'
    }
}

d6tpipe.upsert_pipe(api, settings)
```

Most resources in d6tpipe are managed in an REST API type interface so you will be using the API client to define resources like remotes, pipes, permission etc. That way you can easily switch between local and server deployment without having to change your code.

Parameters

- `name` (str): unique id
- `protocol` (str): [s3, ftp, sftp]
- `location` (str): s3 bucket, ftp server name/ip
- `credentials` (json): credentials for pulling. s3: aws_access_key_id, aws_secret_access_key. ftp: username, password
- **`options` (json): any options to be shared across pipes**
  - `remotepath` (str): path where data is located. If you connect to a Databolt Pipe server that is managed on your behalf
  - `dir` (str): read/write from/to this subdir (auto created)
- `schema` (json): any parameters you want to pass to the reader

Templates

```
# s3
settings = \
{
    'name': 'pipe-name',
    'protocol': 's3',
    'location': 'bucket-name',
    'options': {
        'remotepath': 's3://bucket-name/'}
    'credentials' : {
        'aws_access_key_id': 'AAA',
        'aws_secret_access_key': 'BBB'
    }
}

d6tpipe.upsert_pipe(api, settings)

# ftp
settings = \
{
    'name':'yourftp',
    'protocol':'ftp',
    'location':'ftp.domain.com',
    'options': {
        'remotepath': '/'}
    'credentials':{'username':'name', 'password':'secure'}
}

d6tpipe.upsert_pipe(api, settings)
```

4.14.2 Access Control

You can have separate read and write credentials

```
settings = \
{
    'name': 'remote-name',
    'protocol': 's3',
```

(continues on next page)

(continued from previous page)

```
    'location': 'bucket-name',
    'credentials': {
        'read' : {
            'aws_access_key_id': 'AAA',
            'aws_secret_access_key': 'BBB'
        },
        'write' : {
            'aws_access_key_id': 'AAA',
            'aws_secret_access_key': 'BBB'
        }
    }
}
```

4.14.3 Keeping Credentials Safe

Don't Commit Credentials To Source

In practice you wouldn't want to have the credentials in the source code like in the example above. It's better to load the settings from a json, yaml or ini file to a python dictionary that you can pass to the REST API. Alternatively for server-based setups you can work with REST tools like Postman.

Here is a recipe for loading settings from json and yaml files.

```
# create file
(api.repopath/'.creds.json').touch()

# edit file in `api.repo` folder. NB: you don't have to use double quotes in the json␣
↪but you have to use spaces for tabs
print(api.repo)

# load settings and create
settings = d6tpipe.utils.loadjson(api.repopath/'.creds.json')['pipe-name']
d6tpipe.upsert_pipe(api, settings)

# or if you prefer yaml
(api.repopath/'.creds.yaml').touch()
settings_remote = d6tpipe.utils.loadyaml(api.repopath/'.creds.json')['pipe-name']
d6tpipe.upsert_pipe(api, settings)
```

See example templates in https://github.com/d6t/d6tpipe/tree/master/docs

4.14.4 Premium Features

See *Premium Features* to gain access to premium features.

Encrypting Credentials

By default, credentials are stored in clear text.

To keep your credentials safe, especially on the cloud API, you can encrypt them which is very easy easy to do with `api.encode()`.

```
d6tpipe.upsert_pipe(api, api.encode(settings))
```

This uses an encryption key which is auto generated for you, you can update that key if you like, see config section. If you change the encryption key, you will have to recreate all encrypted pipes.

Any form of security has downsides, here is how encrypting credentials impacts functionality:

- If you lose the encryption key, you will have to recreate all encrypted pipes
- All operations have to take place locally, that is you can't schedule any sync or mirroring tasks on the sever because it won't be able to access the source
- You won't be able to share any credentials with other users unless they have your encryption key.

Managing Your Encryption Key

A key is used to encrypt the data. A random key is generated automatically but you can change it, eg if you want to share it across your team.

```
# get key
api.key

# set key
d6tpipe.api.ConfigManager().update({'key':'yourkey'})
```

Creating d6tpipe Compatible S3 buckets

d6tpipe comes batteries included with convenience functions to set up s3 buckets with appropriate users and permissions. It creates a read and write user with API credentials that can be directly passed into the REST API.

```
session = boto3.session.Session(
    aws_access_key_id=settings['AAA'],
    aws_secret_access_key=settings['BBB'],
)
settings = d6tpipe.utils.s3.create_bucket_with_users(session, 'remote-name')
d6tpipe.upsert_pipe(api, settings)
```

See module refernce for details including how to customize. In case you have trust issues, you can inspect the source code to see what it does.

The AWS session need to refer to a user with the following permissions. If you customize `d6tpipe.utils.s3` parameters you might have to amend this. The lazy way of doing is this to create the AWS session with your AWS root keys.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "iam",
            "Effect": "Allow",
            "Action": [
                "iam:DeleteAccessKey",
                "iam:DeleteUser",
                "iam:GetUser",
                "iam:CreateUser",
                "iam:CreateAccessKey",
```

(continues on next page)

(continued from previous page)

```
                "iam:ListAccessKeys"
            ],
            "Resource": "d6tpipe-*"
        },
        {
            "Sid": "VisualEditor0",
            "Effect": "s3-detail",
            "Action": [
                "s3:DeleteObject",
                "s3:GetObject",
                "s3:PutObject",
                "s3:HeadBucket"
            ],
            "Resource": "arn:aws:s3:::d6tpipe-*/*"
        },
        {
            "Sid": "s3-bucket",
            "Effect": "Allow",
            "Action": [
                "s3:CreateBucket",
                "s3:GetBucketPolicy",
                "s3:PutBucketPolicy",
                "s3:ListBucket",
                "s3:DeleteBucket"
            ],
            "Resource": "arn:aws:s3:::d6tpipe-*"
        }
    ]
}
```

Removing d6tpipe S3 buckets

```
# to remove bucket and user
d6tpipe.utils.s3.delete_bucket(session, 'd6tpipe-[remote-name]')
d6tpipe.utils.s3.delete_user(session, 'd6tpipe-[remote-name]-read')
d6tpipe.utils.s3.delete_user(session, 'd6tpipe-[remote-name]-write')
```

4.15 Advanced: Non-python Access

4.15.1 Access REST API

The *repo API* is a standard REST API that you can access with any tool that you normally use to interface with APIs like postman or curl.

To authenticate pass `Authorization Token {token value}`. You can get your API token http://pipe.databolt.tech/gui/api-access.

Example with curl using the demo account

```
curl -H "Authorization: Token 69337d229e1e35677623341973a927518a6abc7c" https://pipe.
↪databolt.tech/v1/api/pipes/
```

4.15.2 REST API Documentation

Swagger documentation is provided at https://pipe.databolt.tech/swagger/

4.15.3 AWS CLI

d6tpipe uses short-term credentials to secure data pipes. That means you have to request new credentials from the REST API. To use the AWS CLI, add the latest credentials to environmental variables and then you can use the AWS CLI as usual. For more details see https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_temp_use-resources.html#using-temp-creds-sdk-cli

```
c = pipe._get_credentials() # pass write=True for write credentials
# or GET https://pipe.databolt.tech/v1/api/pipes/{pipe_name}/credentials/?role=read

# linux - set env vars
print(f'export AWS_ACCESS_KEY_ID={c["aws_access_key_id"]}')
print(f'export AWS_SECRET_ACCESS_KEY={c["aws_secret_access_key"]}')
print(f'export AWS_SESSION_TOKEN={c["aws_session_token"]}')

# windows - set env vars
print(f'SET AWS_ACCESS_KEY_ID={c["aws_access_key_id"]}')
print(f'SET AWS_SECRET_ACCESS_KEY={c["aws_secret_access_key"]}')
print(f'SET AWS_SESSION_TOKEN={c["aws_session_token"]}')

# use aws cli
print(f"aws s3 ls {pipe.settings['options']['remotepath']}")
```

4.16 Reference

4.16.1 d6tpipe package

Submodules

d6tpipe.api module

class `d6tpipe.api.`**`APIClient`**(*token='config'*, *config=None*, *profile=None*, *filecfg='~/d6tpipe/cfg.json'*)

Bases: `d6tpipe.api._APIBase`

Manager to interface with the remote API.

Parameters

- **`token`** (*`str`*) – your API token
- **`config`** (*`dict`*) – manually pass config object
- **`profile`** (*`str`*) – name of profile to use
- **`filecfg`** (*`str`*) – path to where config file is stored

`forgotToken`(*username*, *password*)

Retrieve your API token

Parameters

- **`username`** (*`str`*) –

- **password** (*str*) –

login (*username, password*)

Login if already registered

Parameters

- **username** (*str*) –
- **password** (*str*) –

register (*username, email, password*)

Register a new API user

Parameters

- **username** (*str*) –
- **email** (*str*) –
- **password** (*str*) –

setToken (*token*)

class d6tpipe.api.**APILocal** (*config=None, profile=None, filecfg='~/d6tpipe/cfg.json'*)

Bases: d6tpipe.api._APIBase

As an alternative to the remote API, you can store everything locally. It mirrors the basic funtionality of the remote API but is not as feature rich.

Parameters

- **config** (*dict*) – manually pass config object
- **profile** (*str*) – name of profile to use
- **filecfg** (*str*) – path to where config file is stored

class d6tpipe.api.**ConfigManager** (*profile=None, filecfg='~/d6tpipe/cfg.json'*)

Bases: object

Manage local config. The config is stored in JSON and can be edited directly *filecfg* location, by default '~/d6tpipe/cfg.json'

Parameters

- **profile** (*str*) – name of profile to use
- **filecfg** (*str*) – path to where config file is stored

init (*config=None, server='https://pipe.databolt.tech', reset=False*)

Initialize config with content

Parameters

- **config** (*dict*) – manually pass config object
- **server** (*str*) – location of REST API server
- **reset** (*bool*) – force reset of an existing config

load ()

Loads config

Returns config

Return type dict

update(*config*)

Update config. Only keys present in the new dict will be updated, other parts of the config will be kept as is. In other words you can pass in a partial dict to update just the parts you need to be updated.

Parameters **config** (*dict*) – updated config

d6tpipe.api.**list_profiles**(*filecfg='~/d6tpipe/cfg.json'*)

d6tpipe.api.**upsert_permissions**(*api, pipe_name, settings*)

Convenience function to create or update pipe permissions

Parameters

- **api** (*obj*) – api
- **settings** (*dict*) – permission settings

Returns server response data (dict): json returned by server

Return type response (obj)

d6tpipe.api.**upsert_pipe**(*api, settings*)

Convenience function to create or update a pipe

Parameters

- **api** (*obj*) – api
- **settings** (*dict*) – pipe settings

Returns server response data (dict): json returned by server

Return type response (obj)

d6tpipe.api.**upsert_pipe_json**(*api, path_json, name*)

Convenience function to create or update a resource. Loads settings from config file to secure credentials

Parameters

- **api** (*obj*) – api
- **path_json** (*str*) – path to config file in json format
- **name** (*str*) – name of json entry

Returns server response data (dict): json returned by server

Return type response (obj)

d6tpipe.api.**upsert_resource**(*apiroot, settings*)

Convenience function to create or update a resource

Parameters

- **apiroot** (*obj*) – API endpoint root eg *api.cnxn.pipes*
- **settings** (*dict*) – resource settings

Returns server response data (dict): json returned by server

Return type response (obj)

d6tpipe.pipe module

class d6tpipe.pipe.**Pipe**(*api, name, mode='default', sortby='filename', credentials=None*)

Bases: *d6tpipe.pipe.PipeBase*

Managed data pipe

Parameters

- **api** (*obj*) – API manager object
- **name** (*str*) – name of the data pipe. Has to be created first
- **mode** (*str*) – sync mode
- **sortby** (*str*) – sort files this key. *name*, *mod*, *size*
- **credentials** (*dict*) – override credentials

Note:

- **mode: controls which files are synced**
  - 'default': modified and new files
  - 'new': new files only
  - 'mod': modified files only
  - 'all': all files

is_synced (*israise=False*)

Check if local is in sync with remote

Parameters **israise** (*bool*) – raise an exception

Returns pipe is updated

Return type bool

pull (*files=None*, *dryrun=False*, *include=None*, *exclude=None*, *nrecent=0*, *merge_mode=None*, *cached=True*)

Pull remote files to local

Parameters

- **files** (*list*) – override list of filenames
- **dryrun** (*bool*) – preview only
- **include** (*str*) – pattern of files to include, eg **.csv* or *a-*.csv|b-*.csv*
- **exclude** (*str*) – pattern of files to exclude
- **nrecent** (*int*) – use n newest files by mod date. 0 uses all files. Negative number uses n old files
- **merge_mode** (*str*) – how to deal with pull conflicts ie files that changed both locally and remotely? 'keep' local files or 'overwrite' local files
- **cached** (*bool*) – if True, use cached remote information, default 5mins. If False forces remote scan

Returns filenames with attributes

Return type list

pull_preview (*files=None*, *include=None*, *exclude=None*, *nrecent=0*, *cached=True*)

Preview of files to be pulled

Parameters

- **files** (*list*) – override list of filenames
- **include** (*str*) – pattern of files to include, eg **.csv* or *a-*.csv|b-*.csv*
- **exclude** (*str*) – pattern of files to exclude
- **nrecent** (*int*) – use n newest files by mod date. 0 uses all files. Negative number uses n old files
- **cached** (*bool*) – if True, use cached remote information, default 5mins. If False forces remote scan

Returns filenames with attributes

Return type list

push (*files=None*, *dryrun=False*, *fromdb=False*, *include=None*, *exclude=None*, *nrecent=0*, *cached=True*)

Push local files to remote

Parameters

- **files** (*list*) – override list of filenames
- **dryrun** (*bool*) – preview only
- **fromdb** (*bool*) – use files from local db, if false scans all files in pipe folder
- **include** (*str*) – pattern of files to include, eg **.csv* or *a-*.csv|b-*.csv*
- **exclude** (*str*) – pattern of files to exclude
- **nrecent** (*int*) – use n newest files by mod date. 0 uses all files. Negative number uses n old files
- **cached** (*bool*) – if True, use cached remote information, default 5mins. If False forces remote scan

Returns filenames with attributes

Return type list

push_preview (*files=None*, *include=None*, *exclude=None*, *nrecent=0*, *cached=True*)

Preview of files to be pushed

Parameters

- **files** (*list*) – override list of filenames
- **include** (*str*) – pattern of files to include, eg **.csv* or *a-*.csv|b-*.csv*
- **exclude** (*str*) – pattern of files to exclude
- **nrecent** (*int*) – use n newest files by mod date. 0 uses all files. Negative number uses n old files
- **cached** (*bool*) – if True, use cached remote information, default 5mins. If False forces remote scan

Returns filenames with attributes

Return type list

remove_orphans (*files=None*, *direction='local'*, *dryrun=None*)

Remove file orphans locally and/or remotely. When you remove files, they don't get synced because pull/push only looks at new or modified files. Use this to clean up any removed files.

Parameters

- **direction** (*str*) – where to remove files
- **dryrun** (*bool*) – preview only

Note:

- **direction:**
  - 'local': remove files locally, ie files that exist on local but not in remote
  - 'remote': remove files remotely, ie files that exist on remote but not in local
  - 'both': combine local and remote

reset (*delete=False*)

Resets by deleting all files and pulling

scan_remote (*attributes=False*, *cached=True*)

Get file attributes from remote. To run before doing a pull/push

Parameters **cached** (*bool*) – use cached results

Returns filenames with attributes in remote

Return type list

setmode (*mode*)

Set sync mode

Parameters **mode** (*str*) – sync mode

Note:

- **mode: controls which files are synced**
  - 'default': modified and new files
  - 'new': new files only
  - 'mod': modified files only
  - 'all': all files

update_settings (*settings*)

Update settings. Only keys present in the new dict will be updated, other parts of the config will be kept as is. In other words you can pass in a partial dict to update just the parts you need to be updated.

Parameters **config** (*dict*) – updated config

class d6tpipe.pipe.**PipeBase** (*name*, *sortby='filename'*)

Bases: object

Abstract class, use the functions but dont instantiate the class

delete_files (*files=None*, *confirm=True*, *all_local=None*)

Delete files, local and remote

Parameters

- **files** (*list*) – filenames, if empty delete all
- **confirm** (*bool*) – ask user to confirm delete
- **all_local** (*bool*) – delete all files or just synced files? (local only)

delete_files_local (*files=None*, *confirm=True*, *delete_all=None*, *ignore_errors=False*)

Delete all local files and reset file database

Parameters

- **files** (`list`) – filenames, if empty delete all
- **confirm** (`bool`) – ask user to confirm delete
- **delete_all** (`bool`) – delete all files or just synced files?
- **ignore_errors** (`bool`) – ignore missing file errors

delete_files_remote (*files=None*, *confirm=True*)

Delete all remote files

Parameters

- **files** (`list`) – filenames, if empty delete all
- **confirm** (`bool`) – ask user to confirm delete

filepaths (*include=None*, *exclude=None*, *sortby=None*, *check_exists=True*, *aspathlib=True*, *aspath=True*, *fromdb=True*)

Full path to Files synced in local repo

Parameters

- **include** (`str`) – pattern of files to include, eg **.csv* or *a-*.csv|b-*.csv*
- **exclude** (`str`) – pattern of files to exclude
- **sortby** (`str`) – sort files this key. *name*, *mod*, *size*
- **check_exists** (`bool`) – check files exist locally
- **aspathlib** (`bool`) – return as pathlib object

Returns path to file, either *Pathlib* or *str*

Return type path

files (*include=None*, *exclude=None*, *sortby=None*, *check_exists=True*, *fromdb=True*)

Files synced in local repo

Parameters

- **include** (`str`) – pattern of files to include, eg **.csv* or *a-*.csv|b-*.csv*
- **exclude** (`str`) – pattern of files to exclude
- **sortby** (`str`) – sort files this key. *name*, *mod*, *size*
- **check_exists** (`bool`) – check files exist locally

Returns filenames

Return type list

import_dir (*dir*, *move=False*)

Import a directory including subdirs

Parameters

- **dir** (`str`) – directory
- **move** (`bool`) – move or copy

import_file (*file*, *subdir=None*, *move=False*)

Import a single file to repo

Parameters

- **files** (*str*) – path to file
- **subdir** (*str*) – sub directory to import into
- **move** (*bool*) – move or copy

import_files (*files*, *subdir=None*, *move=False*)
Import files to repo

Parameters

- **files** (*list*) – list of files, eg from *glob.iglob('folder/**/*.csv')*
- **subdir** (*str*) – sub directory to import into
- **move** (*bool*) – move or copy

scan_local (*include=None*, *exclude=None*, *attributes=False*, *sortby=None*, *files=None*, *fromdb=False*, *on_not_exist='warn'*)
Get file attributes from local. To run before doing a pull/push

Parameters

- **include** (*str*) – pattern of files to include, eg **.csv* or *a-*.csv|b-*.csv*
- **exclude** (*str*) – pattern of files to exclude
- **attributes** (*bool*) – return filenames with attributes
- **sortby** (*str*) – sort files this key. *name*, *mod*, *size*
- **files** (*list*) – override list of filenames
- **fromdb** (*bool*) – use files from local db, if false scans all files in pipe folder
- **on_not_exist** (*bool*) – how to handle missing files when creating file attributes

Returns filenames with attributes

Return type list

class d6tpipe.pipe.**PipeLocal** (*name*, *config=None*, *profile=None*, *filecfg='~/d6tpipe/cfg.json'*, *sortby='filename'*)
Bases: *d6tpipe.pipe.PipeBase*

Managed data pipe, LOCAL mode for accessing local files ONLY

Parameters

- **api** (*obj*) – API manager object
- **name** (*str*) – name of the data pipe
- **profile** (*str*) – name of profile to use
- **filecfg** (*str*) – path to where config file is stored
- **sortby** (*str*) – sort files this key. *name*, *mod*, *size*

Module contents

4.17 API Docs

- modindex

4.18 Search

- search

Python Module Index

Index