fdi

Release v0.18c

Maohai Huang

Nov 22, 2022

CONTENTS:

FDI, known as SPDC before, is written in Python for integrating different types of data, and letting the integrated product take care of inter-platform compatibility, serialisation, persistence, and data object referencing that enables lazy-loading.

CHAPTER

ONE

FEATURES

With FDI one can pack data of different format into **modular** Data Products, together with annotation (description and units) and meta data (data about data). One can make arrays or tables of Products using basic data structures such as sets, sequences (Python `list`), mappings (Python `dict`), or custom-made classes. FDI accomodates nested and highly complex structures.

Access APIs of the components of 'FDIs' are convenient, making it easier for **scripting and data mining** directly 'on FDIs'.

All levels of FDI Products and their component (datasets or metadata) are portable (**serializable**) in human-friendly standard format (JSON implemented), allowing machine data processors on different platforms to parse, access internal components, or re-construct "an FDI". Even a human with a web browser can understand the data.

The `toString()` method of major containers classes outputs nicely formated text representation of complex data to help converting FDI to ASCII.

Most FDI Products and components implement **event sender and listener interfaces**, allowing **scalable data-driven** processing pipelines and visualizers of live data to be constructed.

FDI storage 'pools' (file based and memory based) are provided as references for 1) queryable data **storage** and, 2) for all persistent data to be referenced to with **URNs** (Universal Resource Names).

FDI provides *Context* type of product so that references of other products can become components of a Context, enabling **encapsulation of rich, deep, sophisticated, and accessible contextual data**, yet remain light weight.

For data processors, an HTML **server** with **RESTful APIs** is implemented (named Processing Node Server, PNS) to interface data processing modules. PNS is especially suitable for **Docker containers** in pipelines mixing **legacy software** or software of incompatible environments to form an integral data processing pipeline.

This package attempts to meet scientific observation and data processing requirements, and is inspired by data models of, and designs APIs as compatible as possible with, European Space Agency's Interactive Analysis package of Herschel Common Science System (written in Java, and in Jython for scripting).

CHAPTER

TWO

FDI PYTHON PACKAGES

- The base data model is defined in package *dataset*.
- Persistent data access, referencing, querying, and Universal Resource Names are defined in package *pal*.
- A reference REST API server designed to communicate with a data processing docker using the data model is in package *pns*.

2.1 Install FDI

2.1.1 for developers

```
FDIINSTDIR=/tmp   # change this to your installation dir
cd $FDIINSTDIR
git clone ssh://git@mercury.bao.ac.cn:9005/mh/fdi.git
cd fdi
pip3 install -e .
```

2.1.2 for users

```
cd /tmp
git clone http://mercury.bao.ac.cn:9006/mh/fdi.git
cd fdi
pip3 install -e .
```

to install in /tmp.

2.2 dataset: Model for Data Container

2.2.1 Rationale

A data processing task produces data products that are meant to be shared by other people. When someone receives a data 'product' besides datasets s/he woud expect explanation informaion associated with the product.

Many people tend to store data with no note of meaning attached to them. Without attach meaning of the collection of numbers, it is difficult for other people to fully understand or use the data. It could be difficult for even the data producer to recall the exact meaning of the numbers after a while.

This package implements a data product modeled after Herschel Common Software System (v15) products, taking other requirements of scientific observation and data processing into account. The APIs are kept as compatible with HCSS (written in Java, and in Jython for scripting) as possible.

2.2.2 Definitions

Product

A product has

- zero or more datasets: defining well described data entities (say images, tables, spectra etc…).
- history of this product: how was this data created,
- accompanying meta data – required information such as who created this product, what does the data reflect (say instrument) and so on; possible additional meta data specific to that particular product type.

Dataset

Three types of datasets are implemented to store potentially any data as a dataset. Like a product, all datasets may have meta data, with the distinction that the meta data of a dataset is related to that particular dataset only.

array dataset
: a dataset containing array data (say a data vector, array, cube etc…) and may have a unit.

table dataset
: a dataset containing a collection of columns. Each column contains array data (say a data vector, array, cube etc…) and may have a unit. All columns have the same number of rows. Together they make up the table.

composite dataset
: a dataset containing a collection of datasets. This allows arbitrary complex structures, as a child dataset within a composite dataset may be a composite dataset itself and so on…

Metadata and Parameters

Meta data
: data about data. Defined as a collection of parameters.

Parameter

named scalar variables.
: This package provides the following parameter types:

- _Parameter_: Contains value (classes whitelisted in ParameterTypes)
- _NumericParameter_: Contains a number with a unit.

Apart from the value of a parameter you can ask it for its description and -if it is a numeric parameter- for its unit as well.

History

The history is a lightweight mechanism to record the origin of this product or changes made to this product. Lightweight means, that the Product data itself does not records changes, but external parties can attach additional information to the Product which reflects the changes.

The sole purpose of the history interface of a Product is to allow notably pipeline tasks (as defined by the pipeline framework) to record what they have done to generate and/or modify a Product.

Serializability

In order to transfer data across the network between heterogeneous nodes data needs to be serializable. JSON format is used considering to transfer serialized data for its wide adoption, availability of tools, ease to use with Python, and simplicity.

2.2.3 run tests

In the install directory:

```
make test
```

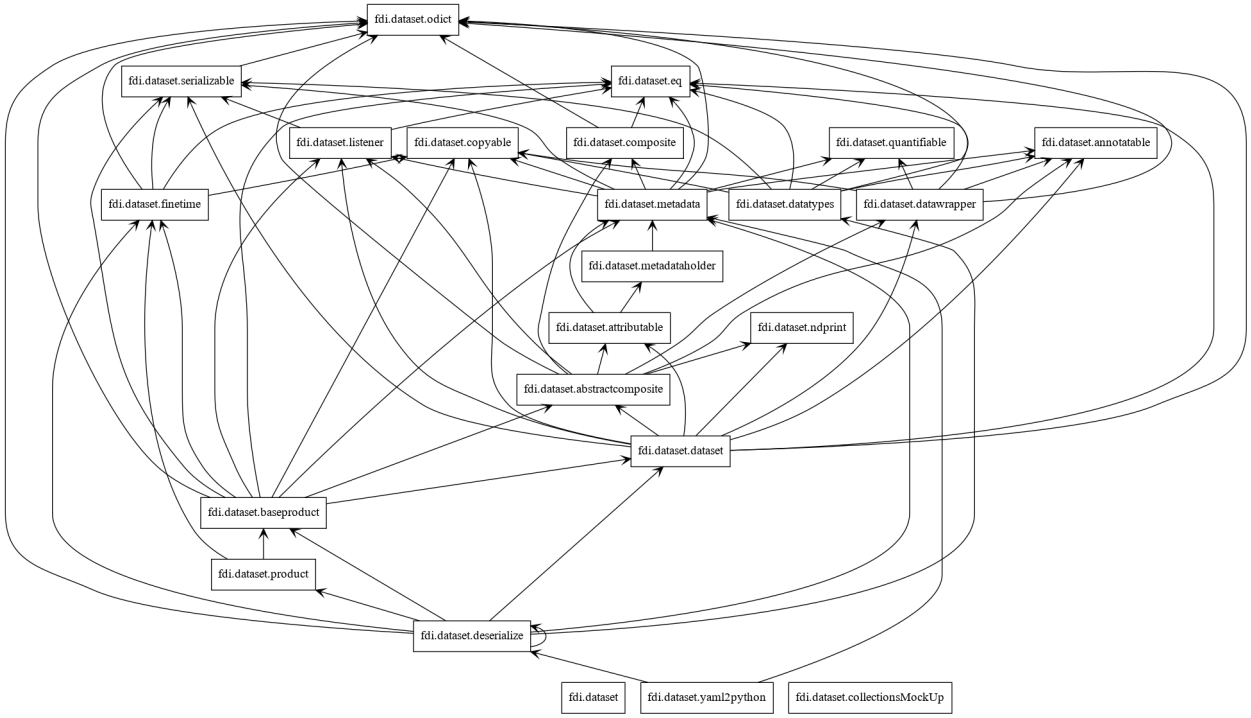
You can only test sub-package `dataset`, `pal`, `pns` or pns server self-test only, by changing `test` above to `test1`, `test2`, `test3`, `test4`, respectively. To pass command-line arguments to `pytest` do

```
make test T='-k Bas'
```

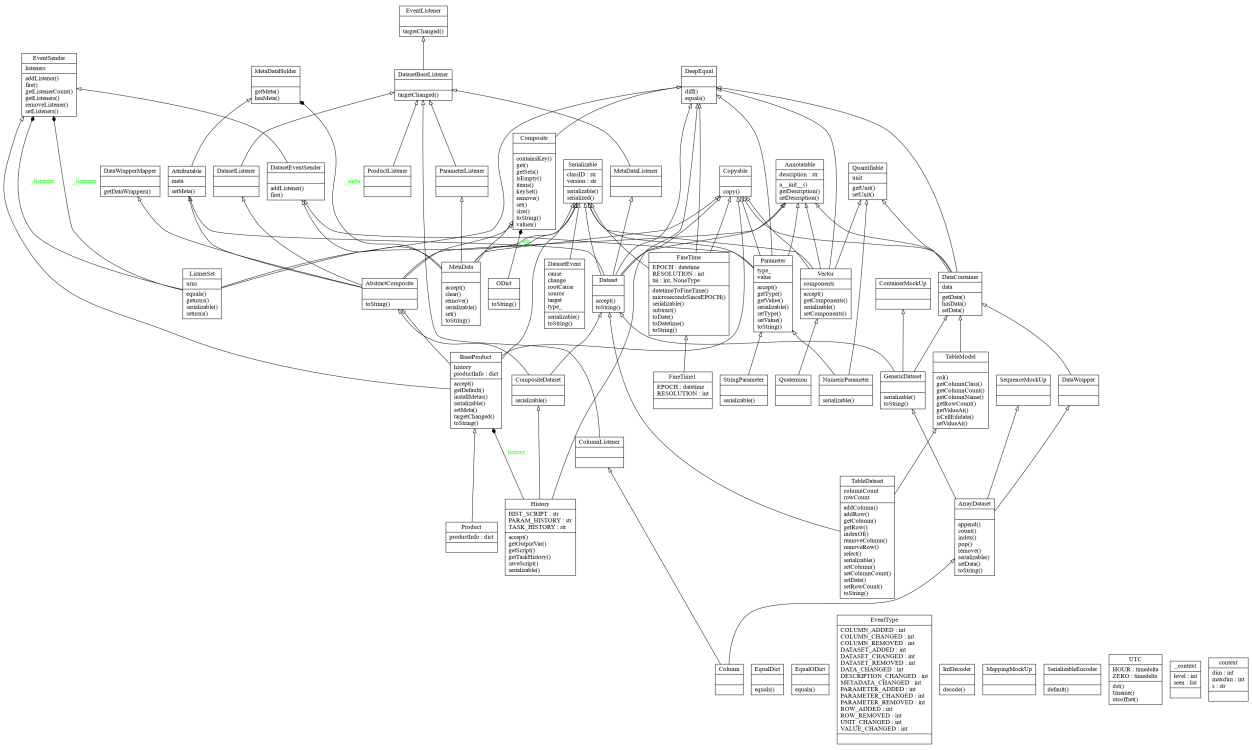
to test `BaseProduct` in sub-package `dataset`.

2.2.4 Design

Packages


fdi.dataset.odict
fdi.dataset.serializable
fdi.dataset.eq
fdi.dataset.listener
fdi.dataset.copyable
fdi.dataset.composite
fdi.dataset.quantifiable
fdi.dataset.annotatable
fdi.dataset.finetime
fdi.dataset.metadata
fdi.dataset.datatypes
fdi.dataset.datawrapper
fdi.dataset.metadataholder
fdi.dataset.attributable
fdi.dataset.ndprint
fdi.dataset.abstractcomposite
fdi.dataset.dataset
fdi.dataset.baseproduct
fdi.dataset.product
fdi.dataset.deserialize
fdi.dataset
fdi.dataset.yaml2python
fdi.dataset.collectionsMockUp

Classes



2.3 pal: Product Access Layer

Product Access Layer allows data stored logical "pools" to be accessed with light weight product refernces by data processers, data storage, and data consumers. A data product can include a context built with references of relevant data. A `ProductStorage` interface is provided to handle saving/retrieving/querying data in registered pools.

2.3.1 Rationale

In a data processing pipeline or network of processing nodes, data products are generated within a context which may include input data, reference data, and auxiliary data of many kind. It is often needed to have relevant context recorded with a product. However the context could have a large size so including their actual data as metadata of the product is often impractical.

Once FDI data are generated they can have a reference through which they can be accessed. The size of such references are typically less than a few hundred bytes, like a URL. In the product context only data references are recorded.

This package provides `MapContext`, `ProductRef`, `Urn`, `ProductStorage`, `ProductPool`, and `Query` classes (simplified but mostly API-compatible with Herschel Common Science System v15.0) for the storing, retrieving, tagging, and context creating of data product modeled in the dataset package.

2.3.2 Definitions

URN

The Universial Resource Name (URN) string has this format:

```
urn:poolname:resourceclass:serialnumber
```

where

resourceclass
: full class name of the resource (product)

poolname
: scheme + `://` + place + directory

scheme
: `file`, `mem`, `http` … etc

place
: `192.168.5.6:8080`, `c:`, an empty string … etc

directory

: **A label for the pool that is by default used as the full path where the pool is stored. ProductPool.transformpath() can used to change the directory here to other meaning.**

  - for `file` scheme: `/` + name + `/` + name + … + `/` + name
  - for `mem` scheme: `/` + name + `/`

serialnumber
: internal index. str(int).

ProductRef

This class not only holds the URN of the product it references to, but also records who (the _parents_) are keeping this reference.

Context and MapContext

Context is a Product that holds a set of `productRef` s that accessible by keys. The keys are strings for MapContext which usually maps names to product references.

ProductStorage

A centralized access place for saving/loading/querying/deleting data organized in conceptual pools. One gets a ProductRef when saving data.

ProductPool

An place where products can be saved, with a reference for the saved product generated. The product can be retrieved with the reference. Pools based on different media or networking mechanism can be implemented. Multiple pools can be registered in a ProductStorage front-end where users can do the saving, loading, querying etc. so that the pools are collectively form a larger logical storage.

Query

One can make queries to a ProductStorage and get back a list of references to products that satisfy search chriteria. Queries can be constructed using Python predicate expressions about a product and its metadata, or a function that returns True or False.

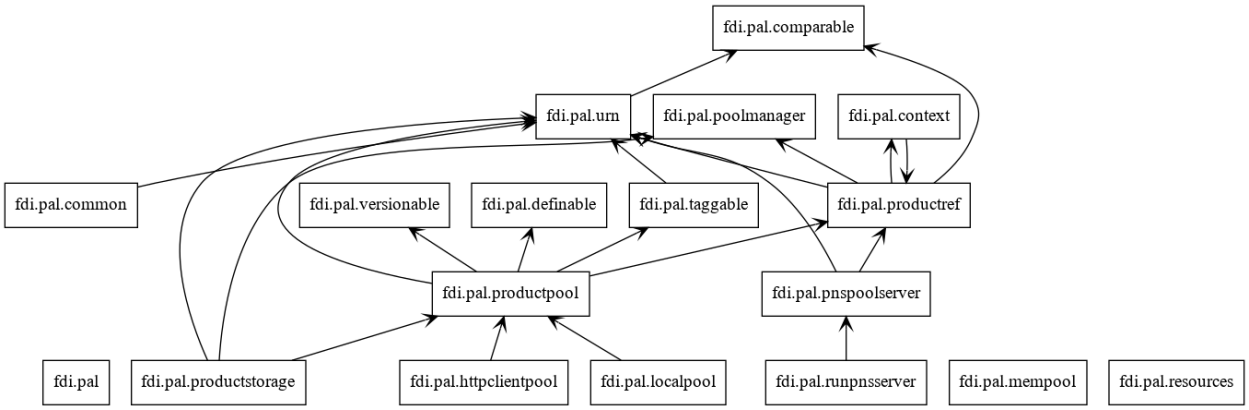
2.3.3 run tests

in the same directory:

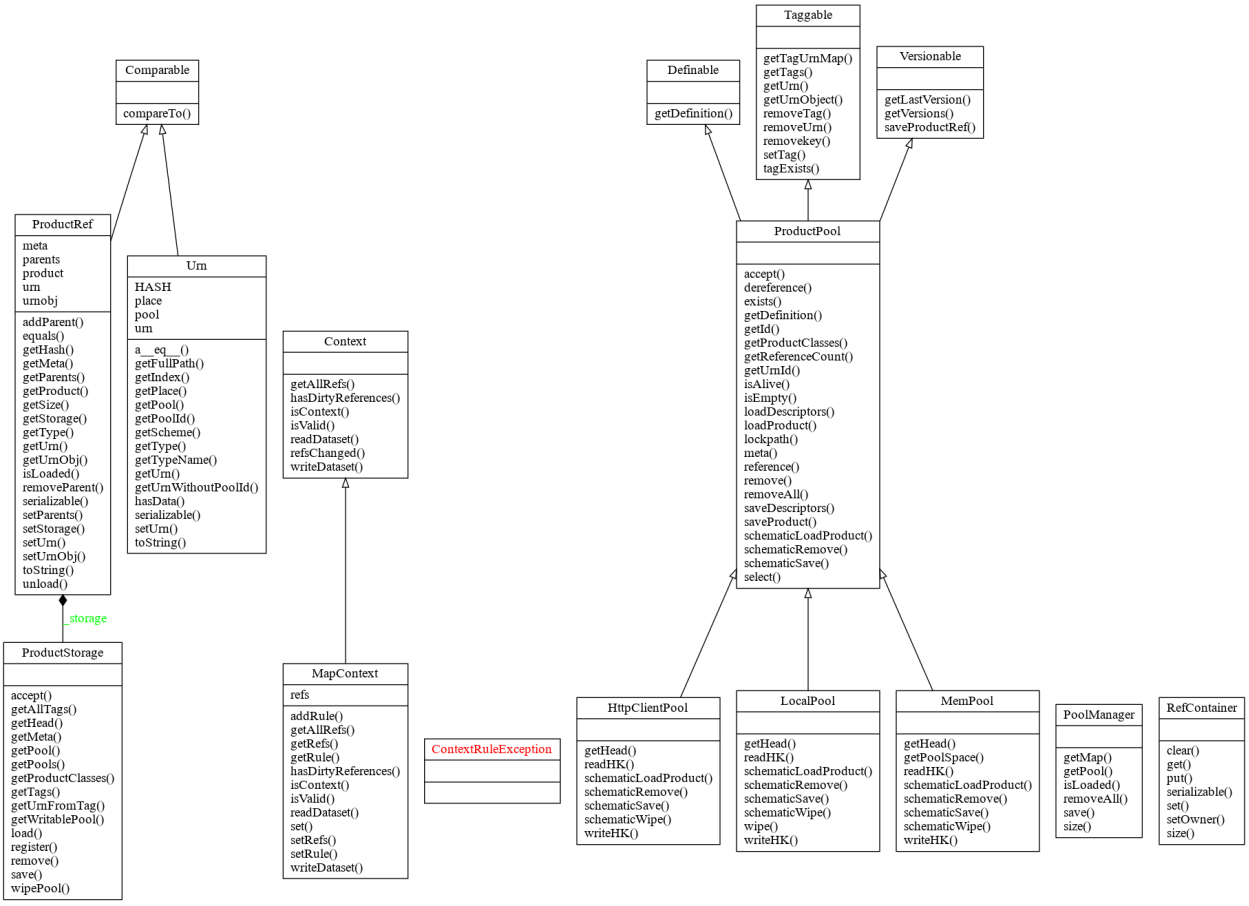
```
make test2
```

2.3.4 Design

Packages


fdi.pal.comparable
fdi.pal.urn
fdi.pal.poolmanager
fdi.pal.context
fdi.pal.common
fdi.pal.versionable
fdi.pal.definable
fdi.pal.taggable
fdi.pal.productref
fdi.pal.productpool
fdi.pal.pnspoolserver
fdi.pal
fdi.pal.productstorage
fdi.pal.httpclientpool
fdi.pal.localpool
fdi.pal.runpnsserver
fdi.pal.mempool
fdi.pal.resources

Classes


Comparable
compareTo()
Taggable
getTagUrnMap()
getTags()
getUrn()
getUrnObject()
removeTag()
removeUrn()
removekey()
setTag()
tagExists()
Definable
getDefinition()
Versionable
getLastVersion()
getVersions()
saveProductRef()
ProductRef
meta
parents
product
urn
urnobj
addParent()
equals()
getHash()
getMeta()
getParents()
getProduct()
getSize()
getStorage()
getType()
getUrn()
getUrnObj()
isLoaded()
removeParent()
serializable()
setParents()
setStorage()
setUrn()
setUrnObj()
toString()
unload()
Urn
HASH
place
pool
urn
a__eq__()
getFullPath()
getIndex()
getPlace()
getPool()
getPoolId()
getScheme()
getType()
getTypeName()
getUrn()
getUrnWithoutPoolId()
hasData()
serializable()
setUrn()
toString()
Context
getAllRefs()
hasDirtyReferences()
isContext()
isValid()
readDataset()
refsChanged()
writeDataset()
ProductPool
accept()
dereference()
exists()
getDefinition()
getId()
getProductClasses()
getReferenceCount()
getUrnId()
isAlive()
isEmpty()
loadDescriptors()
loadProduct()
lockpath()
meta()
reference()
remove()
removeAll()
saveDescriptors()
saveProduct()
schematicLoadProduct()
schematicRemove()
schematicSave()
select()
_storage
ProductStorage
accept()
getAllTags()
getHead()
getMeta()
getPool()
getPools()
getProductClasses()
getTags()
getUrnFromTag()
getWritablePool()
load()
register()
remove()
save()
wipePool()
MapContext
refs
addRule()
getAllRefs()
getRefs()
getRule()
hasDirtyReferences()
isContext()
isValid()
readDataset()
set()
setRefs()
setRule()
writeDataset()
ContextRuleException
HttpClientPool
getHead()
readHK()
schematicLoadProduct()
schematicRemove()
schematicSave()
schematicWipe()
writeHK()
LocalPool
getHead()
readHK()
schematicLoadProduct()
schematicRemove()
schematicSave()
schematicWipe()
wipe()
writeHK()
MemPool
getHead()
getPoolSpace()
readHK()
schematicLoadProduct()
schematicRemove()
schematicSave()
schematicWipe()
writeHK()
PoolManager
getMap()
getPool()
isLoaded()
removeAll()
save()
size()
RefContainer
clear()
get()
put()
serializable()
set()
setOwner()
size()

2.4 pns: Processing Node Server

2.4.1 Rationale

Many data processing pipelines need to run software that only runs on a specific combination of OS type, version, language, and library. These software could be impractical to replace or modify but need to be run side-by-side with software of incompatible environments/formats to form an integral data processing pipeline, each software being a "node" to perform a processing task. Docker containers are often the perfect solution to run software with incompatible dependencies.

PNS installed on a Docker container or a normal server allows such processing tasks to run in the PNS memory space, in a daemon process, or as an OS process receiving input and delivering output through a 'delivery man' protocol.

This Web API Server for a data processing pipeline/network node provides interfaces to configure the data processing task software (PTS) in a processing node, to make a run request, to deliver necessary input data, and to read results, all via web APIs.

The following commands are run from the fdi directory from installation.

2.4.2 Basic Configuration

When running Flask server, the host IP is `0.0.0.0` and port number `5000` by default. They are configurable in `pnsconfig.py`. Default configuration can be overridden by `~/.config/pnslocal.py`. Copy `pnsconfig.py` to `~/.config/pnslocal.py`

```
cp fdi/pns/pnsconfig.py ~/.config/pnslocal.py
```

and customize `~/.config/pnslocal.py`.

When in developement mode, set `dev` to `True` (`dev = True` or `dev = 1`) to run local server. The `serveruser` should be the name of the user of web server, usually your username if you run `make runserver`. This is the default if `dev` is true.

For production deployment the `dev` should be set false. Set `serveruser` depending which web server (e.g. `'apache'`).

The `ptsuser` is usually the user required by the processing software. It is set to `serveruser` by default. `ptsuser` must have write previlige to read and write `inputdir` and `outputdir`, which are owned by `serveruser` with mode `o0775`.

On the server side (or on your computer which can be both the server and the client) edit `Makefile` by changing the value of varible `PNSDIR` in `Makefile` the pnshome directory if you do not want the default (`~/pns`).

Then run the deployment command

```
make installpns
```

to create the pns home directory and copy the demo PTS script set.

2.4.3 Run the FLASK Server

Edit ~/.config/pnslocal.py if needed. Then

```
python3 fdi/pns/runflaskserver.py --username=<username> --password=<password> [--ip=
↪<host ip>] [--port=<port>]
```

Contents in [], like [--ip=<host ip>] [--port=<port>] above, are optional.

<> means you need to substitute with actual information (for example --port=<port> becomes --port=5000).

Or you can run

```
python3 fdi/pns/runflaskserver.py -u <username> -p <password> [-i <host ip>] [-o <port>]
```

in debugging mode:

```
python3 fdi/pns/runflaskserver.py --username=foo --password=bar -v
```

or just

```
make runserver
```

to use the defaults.

Do not run debugging mode for production use.

The username and password are used when making run requests.

2.4.4 Test and Verify Installation

To run all tests in one go:

```
make test3 [T='-u <username> -p <password> [-i <host ip>] [-o <port>] [options]']
```

Tests can be done step-by-step to pin-point possible problems:

2.4.5 1. Server Unit Test

Run this on the server host to verify that internal essential functions of the server work with current configuration. This runs without needing starting the server:

```
make test4
```

2.4.6 2. Local Flask Server Functional Tests

In ~/.config/pnslocal.py (see above for installation and customization), set dev=True and make sure the IP is local (0.0.0.0 or 127.0.0.1). Start the server fresh in one terminal (see above) and in another terminal (on the server host) run the following:

2a: test GET initPTS script to see if reading the init script back works:

```
make test3 T='getinit'
```

2b: test PUT initialization test:

```
make test3 T='-k putinittest'
```

2c1: If the test passes, you can Run all tests in one go:

```
make test3
```

2c2: Or keep on individual tests...

test POST In-server processing

```
make test3 T='-k _post'
```

test POST PTS processing

```
make test3 T='-k _run'
```

test DELETE Clean-up the server by removing the input and output dirs

```
make test3 T='-k deleteclean'
```

Now is a good time to ...

2.4.7 3. Get public access APIs and information

Suppose the server address and port are `127.0.0.1` and `5000`, respectively:

Run the Flask server in a terminal (see above) and open this in a browser. The up-to-date URL is displayed in the server stating message:

http://127.0.0.1:5000/v0.6/

An online API documentation page similar to below is shown.

```
{
"APIs": {
"DELETE": [
{
"URL": "http://127.0.0.1:5000/v0.6/clean",
"description": " Removing traces of past runnings the Processing Task Software.\n    "
}
],
"GET": [
{
"URL": "http://127.0.0.1:5000/v0.6/init",
"description": "the initPTS file"
},
{
"URL": "http://127.0.0.1:5000/v0.6/config",
"description": "the configPTS file"
},
{
"URL": "http://127.0.0.1:5000/v0.6/run",
"description": "the file running PTS"
},
{
```

(continues on next page)

(continued from previous page)


```
"URL": "http://127.0.0.1:5000/v0.6/clean",
"description": "the cleanPTS file"
},
{
"URL": "http://127.0.0.1:5000/v0.6/input",
"description": " returns names and contents of all files in the dir, 'None' if dir not␣
↪existing. "
},
{
"URL": "http://127.0.0.1:5000/v0.6/output",
"description": " returns names and contents of all files in the dir, 'None' if dir not␣
↪existing. "
},
{
"URL": "http://127.0.0.1:5000/v0.6/pnsconfig",
"description": "PNS configuration"
}
],
"POST": [
{
"URL": "http://127.0.0.1:5000/v0.6/calc",
"description": " generates result product directly using data on PNS.\n    "
},
{
"URL": "http://127.0.0.1:5000/v0.6/testcalc",
"description": " generate post test product.\n    put the 1st input (see maketestdata in␣
↪test_all.py)\n    parameter to metadata\n    and 2nd to the product's dataset\n    "
},
{
"URL": "http://127.0.0.1:5000/v0.6/echo",
"description": "Echo"
},
{
"URL": "http://127.0.0.1:5000/v0.6/run",
"description": " Generates a product by running script defined in the config under 'run'.
↪ Execution on the server host is in the pnshome directory and run result and status␣
↪are returned.\n    "
},
{
"URL": "http://127.0.0.1:5000/v0.6/testrun",
"description": "  Run 'runPTS' for testing, and as an example.\n    "
}
],
"PUT": [
{
"URL": "http://127.0.0.1:5000/v0.6/init",
"description": " Initialize the Processing Task Software by running the init script␣
↪defined in the config. Execution on the server host is in the pnshome directory and␣
↪run result and status are returned. If input/output directories cannot be created with␣
↪serveruser as owner, Error401 will be given.\n    "
},
{
```


(continues on next page)

(continued from previous page)


```
"URL": "http://127.0.0.1:5000/v0.6/config",
"description": " Configure the Processing Task Software by running the config script.␣
↪Ref init PTS.\n    "
},
{
"URL": "http://127.0.0.1:5000/v0.6/pnsconf",
"description": " Configure the PNS itself by replacing the pnsconfig var\n    "
},
{
"URL": "http://127.0.0.1:5000/v0.6/inittest",
"description": "     Renames the 'init' 'config' 'run' 'clean' scripts to \"*.save\" and␣
↪points it to the '.ori' scripts.\n    "
}
]
},
"timestamp": 1566130779.0208821
}
```


Continue with tests…

2.4.8 4. Run tests from a remote client

Install pns on a remote host, configure IP and port, then run the tests above. This proves that the server and the client have connection and fire wall configured correctly.

2.4.9 Run the local tests with Apache

Set dev=False in ~/.config/pnslocal.py (see above) and set the IP and port. Suppose the server is on CentOS. Edit pns/resources/pns.conf according to local setup, then

```
cp pns/resources/pns.conf /etc/httpd/conf.d
systemctl restart httpd
systemctl status http -l
```

then run the above with correct IP and port (edit ~/.config/pnslocal.py or specifying in command line). Start the server and run all the tests:

```
make test3
```

2.4.10 PTS Configuration

To run a PTS shell script instead of the 'hello' demo, change the `run` parameter in the config file, e.g. to run the script named `runPTS.vvpp`

```
run=[join(h, 'runPTS.vvpp'), ''],
```

restart the server. run

```
make test4
```

2.4.11 PTS API

TBW

2.4.12 Return on Common Errors

400

```
{'error': 'Bad request.', 'timestamp': ts}
```

401

```
{'error': 'Unauthorized. Authentication needed to modify.', 'timestamp': ts}
```

404

```
{'error': 'Not found.', 'timestamp': ts}
```

409

```
{'error': 'Conflict. Updating.', 'timestamp': ts}
```

Resources

TBW

2.5 fdi Quick Start

Contents:

The following demostrates important dataset and pal functionalities. It was made by running `fdi/resources/example.py` with command `elpy-shell-send-group-and-step [c-c c-y c-g]` in `emacs`.

You can copy the code from code blocks by clicking the `copy` icon on the top-right, with he prompts and results removed.

```
>>> # import these first.
... import copy
... import getpass
... import os
... from datetime import datetime
... import logging
... from fdi.dataset.product import Product
... from fdi.dataset.metadata import Parameter, NumericParameter, MetaData
... from fdi.dataset.finetime import FineTime1, utcobj
... from fdi.dataset.dataset import ArrayDataset, TableDataset, Column
... from fdi.pal.context import Context, MapContext
... from fdi.pal.productref import ProductRef
... from fdi.pal.query import MetaQuery
... from fdi.pal.poolmanager import PoolManager, DEFAULT_MEM_POOL
... from fdi.pal.productstorage import ProductStorage
```

2.5.1 dataset

ArrayDataset

```
>>> # Creation
... a1 = [1, 4.4, 5.4E3, -22, 0xa2]      # a 1D array of data
... v = ArrayDataset(data=a1, unit='ev', description='5 elements')
... v
ArrayDataset{ [1, 4.4, 5400.0, -22, 162] <ev>, description = "5 elements", meta =␣
↪MetaData{[], listeners = []}}
```

```
>>> # data access
... v[2]
5400.0
```

```
>>> v.unit
'ev'
```

```
>>> v.unit = 'm'
... v.unit
'm'
```

```
>>> # iteration
... for m in v:
...     print(m)
1
4.4
5400.0
```

(continues on next page)

(continued from previous page)

```
-22
162
```

```
>>> [m**3 for m in v if m > 0 and m < 40]
[1, 85.18400000000003]
```

```
>>> # slice
... v[1:3]
[4.4, 5400.0]
```

```
>>> v[2:-1]
[5400.0, -22]
```

```
>>> v.data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
... v[0:2]
[[1, 2, 3], [4, 5, 6]]
```

```
>>> # Run this to see a demo of the ``toString()`` function::
... # make a 4-D array: a list of 2 lists of 3 lists of 4 lists of 5 elements.
... s = [[[[i + j + k + l for i in range(5)] for j in range(4)]
...       for k in range(3)] for l in range(2)]
... x = ArrayDataset(data=s)
... print(x.toString())
```

```
# ArrayDataset
# description = "UNKNOWN"
# meta = MetaData{[], listeners = []}
# unit = "None"
# data =

0 1 2 3
1 2 3 4
2 3 4 5
3 4 5 6
4 5 6 7


1 2 3 4
2 3 4 5
3 4 5 6
4 5 6 7
5 6 7 8


2 3 4 5
3 4 5 6
4 5 6 7
5 6 7 8
6 7 8 9
```

(continues on next page)

(continued from previous page)

```
#=== dimension 4

1 2 3 4
2 3 4 5
3 4 5 6
4 5 6 7
5 6 7 8


2 3 4 5
3 4 5 6
4 5 6 7
5 6 7 8
6 7 8 9


3 4 5 6
4 5 6 7
5 6 7 8
6 7 8 9
7 8 9 10


#=== dimension 4
```

TableDataset

```
>>> # Creation
... a1 = [dict(name='col1', unit='eV', column=[1, 4.4, 5.4E3]),
...       dict(name='col2', unit='cnt', column=[0, 43.2, 2E3])
...       ]
... v = TableDataset(data=a1)
... v
TableDataset{ description = "UNKNOWN", meta = MetaData{[], listeners = []}, data = "OD{
↪'col1':Column{ [1, 4.4, 5400.0] <eV>, description = "UNKNOWN", meta = MetaData{[],␣
↪listeners = []}}, 'col2':Column{ [0, 43.2, 2000.0] <cnt>, description = "UNKNOWN",␣
↪meta = MetaData{[], listeners = []}}}"}
```

```
>>> # many other ways to create a TableDataset
... v3 = TableDataset(data=[('col1', [1, 4.4, 5.4E3], 'eV'),
...                         ('col2', [0, 43.2, 2E3], 'cnt')])
... v == v3
True
```

```
>>> # quick and dirty. data are list of lists without names or units
... a5 = [[1, 4.4, 5.4E3], [0, 43.2, 2E3]]
... v5 = TableDataset(data=a5)
... print(v5.toString())
```

```
# TableDataset
# description = "UNKNOWN"
# meta = MetaData{[], listeners = []}
# data =

# col1 col2
# None None
1 0
4.4 43.2
5400.0 2000.0
```

```
>>> # access
... # get names of all column
... v5.data.keys()
odict_keys(['col1', 'col2'])
```

```
>>> # get a list of all columns' data
... [c.data for c in v5.data.values()]   # == a5
[[1, 4.4, 5400.0], [0, 43.2, 2000.0]]
```

```
>>> # get column by name
... c_1 = v5['col1']
... c_1
Column{ [1, 4.4, 5400.0] <None>, description = "UNKNOWN", meta = MetaData{[], listeners␣
↪= []}}
```

```
>>> #  indexOf
... v5.indexOf('col1')  # == u.indexOf(c_1)
0
```

```
>>> v5.indexOf(c_1)
0
```

```
>>> # get a cell
... v5['col2'][1]    # 43.2
43.2
```

```
>>> # set cell value
... v5['col2'][1] = 123
... v5['col2'][1]    # 123
123
```

```
>>> v5.setValueAt(aValue=42, rowIndex=1, columnIndex=1)
... v5.getValueAt(rowIndex=1, columnIndex=1)    # 42
42
```

```
>>> # unit access
... v3['col1'].unit  # == 'eV'
'eV'
```

```
>>> # add, set, and replace columns and rows
... # column set / get
... u = TableDataset()
... c1 = Column([1, 4], 'sec')
... u.addColumn('col3', c1)
... u.columnCount        # 1
1
```

```
>>> # for non-existing names set is addColum.
... c2 = Column([2, 3], 'eu')
... u['col4'] = c2
... u['col4'][0]    # 2
2
```

```
>>> u.columnCount        # 2
2
```

```
>>> # replace column for existing names
... c3 = Column([5, 7], 'j')
... u['col4'] = c3
... u['col4'][0]    # c3.data[0]
5
```

```
>>> # addRow
... u.rowCount    # 2
2
```

```
>>> cc = copy.deepcopy(c1)
... c33, c44 = 3.3, 4.4
... cc.append(c33)
... u.addRow({'col4': c44, 'col3': c33})
... u.rowCount    # 3
3
```

```
>>> u['col3']    # cc
Column{ [1, 4, 3.3] <sec>, description = "UNKNOWN", meta = MetaData{[], listeners = []}}
```

```
>>> # removeRow
... u.removeRow(u.rowCount - 1)    # [c33, c44]
[3.3, 4.4]
```

```
>>> u.rowCount    # 2
2
```

```
>>> # syntax ``in``
... [c for c in u]  # list of column names ['col1', 'col2']
['col3', 'col4']
```

```
>>> # run this to see ``toString()``
... ELECTRON_VOLTS = 'eV'
```

(continues on next page)

(continued from previous page)

```
... SECONDS = 'sec'
... t = [x * 1.0 for x in range(10)]
... e = [2 * x + 100 for x in t]
... # creating a table dataset to hold the quantified data
... x = TableDataset(description="Example table")
... x["Time"] = Column(data=t, unit=SECONDS)
... x["Energy"] = Column(data=e, unit=ELECTRON_VOLTS)
... print(x.toString())
```

```
# TableDataset
# description = "Example table"
# meta = MetaData{[], listeners = []}
# data =

# Time Energy
# sec eV
0.0 100.0
1.0 102.0
2.0 104.0
3.0 106.0
4.0 108.0
5.0 110.0
6.0 112.0
7.0 114.0
8.0 116.0
9.0 118.0
```

Parameter

```
>>> # Creation
... # standard way -- with keyword arguments
... a1 = 'a test parameter'
... a2 = 300
... a3 = 'integer'
... v = Parameter(description=a1, value=a2, type_=a3)
... v.description   # == a1
'a test parameter'
```

```
>>> v.value   # == a2
300
```

```
>>> v.type_   # == a3
'integer'
```

```
>>> # with no argument
... v = Parameter()
... v.description   # == 'UNKNOWN# inherited from Anotatable
'UNKNOWN'
```

```
>>> v.value   # is None
```

```
>>> v.type_   # == ''
''
```

```
>>> # make a blank one then set attributes
... v = Parameter(description=a1)
... v.description   # == a1
'a test parameter'
```

```
>>> v.value    # is None
```

```
>>> v.type_   # == ''
''
```

```
>>> v.setValue(a2)
... v.setType(a3)
... v.description   # == a1
'a test parameter'
```

```
>>> v.value   # == a2
300
```

```
>>> v.type_   # == a3
'integer'
```

```
>>> # test equivalence of v.setXxxx(a) and v.xxx = a
... a1 = 'test score'
... a2 = 98
... v = Parameter()
... v.description = a1
... v.value = a2
... v.description   # == a1
'test score'
```

```
>>> v.value   # == a2
98
```

```
>>> # test equals
... b1 = ''.join(a1)  # make a new string copy
... b2 = a2 + 0  # make a copy
... v1 = Parameter(description=b1, value=b2)
... v.equals(v1)
True
```

```
>>> v == v1
True
```

```
>>> v1.value = -4
... v.equals(v1)   # False
False
```

```
>>> v != v1  # True
True
```

Metadata

```
>>> # Creation
... a1 = 'age'
... a2 = NumericParameter(description='since 2000',
...                       value=20, unit='year', type_='integer')
... v = MetaData()
... v.set(a1, a2)
... v.get(a1)   # == a2
NumericParameter{ 20 (year) <integer>, "since 2000"}
```

```
>>> # add more parameter
... a3 = 'Bob'
... v.set(name='name', newParameter=Parameter(a3))
... v.get('name').value   # == a3
'Bob'
```

```
>>> # access parameters in metadata
... v = MetaData()
... # a more readable way to set a parameter
... v[a1] = a2  # DRM doc case
... # a more readable way to get a parameter
... v[a1]   # == a2
NumericParameter{ 20 (year) <integer>, "since 2000"}
```

```
>>> v.get(a1)   # == a2
NumericParameter{ 20 (year) <integer>, "since 2000"}
```

```
>>> v['date'] = Parameter(description='take off at',
...                       value=FineTime1.datetimeToFineTime(datetime.now(tz=utcobj)))
... # names of all parameters
... [n for n in v]   # == [a1, 'date']
['age', 'date']
```

```
>>> print(v.toString())
MetaData{[age = NumericParameter{ 20 (year) <integer>, "since 2000"}, date = Parameter{␣
↪108120221290 <integer>, "take off at"}, ], listeners = []}
```

```
>>> # remove parameter
... v.remove(a1)  # inherited from composite
... print(v.size())  # == 1
1
```

Product

```
>>> # Creation:
... x = Product(description="product example with several datasets",
...             instrument="Crystal-Ball", modelName="Mk II")
... x.meta['description'].value  # == "product example with several datasets"
'product example with several datasets'
```

```
>>> x.instrument  # == "Crystal-Ball"
'Crystal-Ball'
```

```
>>> # ways to add datasets
... i0 = 6
... i1 = [[1, 2, 3], [4, 5, i0], [7, 8, 9]]
... i2 = 'ev'                  # unit
... i3 = 'image1'     # description
... image = ArrayDataset(data=i1, unit=i2, description=i3)
... x["RawImage"] = image
... x["RawImage"].data  # == [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
>>> # no unit or description. different syntax but same function as above
... x.set('QualityImage', ArrayDataset(
...     [[0.1, 0.5, 0.7], [4e3, 6e7, 8], [-2, 0, 3.1]]))
... x["QualityImage"].unit  # is None
```

```
>>> # add a tabledataset
... s1 = [('col1', [1, 4.4, 5.4E3], 'eV'),
...       ('col2', [0, 43.2, 2E3], 'cnt')]
... x["Spectrum"] = TableDataset(data=s1)
... print(x["Spectrum"].toString())
```

```
# TableDataset
# description = "UNKNOWN"
# meta = MetaData{[], listeners = []}
# data =

# col1 col2
# eV cnt
1 0
4.4 43.2
5400.0 2000.0
```

```
>>> # mandatory properties are also in metadata
... # test mandatory BaseProduct properties that are also metadata
... x.creator = ""
... a0 = "Me, myself and I"
... x.creator = a0
... x.creator   # == a0
'Me, myself and I'
```

```
>>> # metada by the same name is also set
... x.meta["creator"].value   # == a0
'Me, myself and I'
```

```
>>> # change the metadata
... a1 = "or else"
... x.meta["creator"] = Parameter(a1)
... # metada changed
... x.meta["creator"].value   # == a1
'or else'
```

```
>>> # so did the property
... x.creator   # == a1
'or else'
```

```
>>> # Demo ``toString()`` function. The result should be ::
... print(x.toString())
```

```
# Product
# description = "product example with several datasets"
# meta = MetaData{[description = Parameter{ product example with several datasets
↪<string>, "Description of this product"}, type = Parameter{ Product <string>, "Product␣
↪Type identification. Fully qualified Python class name or CARD."}, creator = Parameter
↪{ or else <string>, "UNKNOWN"}, creationDate = Parameter{ 2017-01-01T00:00:00.000000␣
↪TAI(0) <finetime>, "Creation date of this product"}, rootCause = Parameter{ UNKOWN
↪<string>, "Reason of this run of pipeline."}, schema = Parameter{ 0.3 <string>,
↪"Version of product schema"}, startDate = Parameter{ 2017-01-01T00:00:00.000000 TAI(0)
↪<finetime>, "Nominal start time  of this product."}, endDate = Parameter{ 2017-01-
↪01T00:00:00.000000 TAI(0) <finetime>, "Nominal end time  of this product."},␣
↪instrument = Parameter{ Crystal-Ball <string>, "Instrument that generated data of this␣
↪product"}, modelName = Parameter{ Mk II <string>, "Model name of the instrument of␣
↪this product"}, mission = Parameter{ _AGS <string>, "Name of the mission."}, ],␣
↪listeners = []}
# History
# description = "UNKNOWN"
# meta = MetaData{[], listeners = []}
# data =

# data =


# [ RawImage ]
# ArrayDataset
# description = "image1"
# meta = MetaData{[], listeners = []}
# unit = "ev"
# data =

1 4 7
2 5 8
3 6 9
```

(continues on next page)

(continued from previous page)

```
# [ QualityImage ]
# ArrayDataset
# description = "UNKNOWN"
# meta = MetaData{[], listeners = []}
# unit = "None"
# data =

0.1 4000.0 -2
0.5 60000000.0 0
0.7 8 3.1


# [ Spectrum ]
# TableDataset
# description = "UNKNOWN"
# meta = MetaData{[], listeners = []}
# data =

# col1 col2
# eV cnt
1 0
4.4 43.2
5400.0 2000.0
```

2.5.2 pal

Store a Product in a Pool and Get a Reference Back

Create a product and a productStorage with a pool registered

```
>>> # disable debugging messages
... logger = logging.getLogger('')
... logger.setLevel(logging.WARNING)
```

```
>>> # a pool for demonstration will be create here
... demopoolpath = '/tmp/demopool_' + getpass.getuser()
... demopool = 'file://' + demopoolpath
... # clean possible data left from previous runs
... os.system('rm -rf ' + demopoolpath)
... PoolManager.getPool(DEFAULT_MEM_POOL).removeAll()
... PoolManager.removeAll()
```

```
>>> # create a prooduct and save it to a pool
... x = Product(description='in store')
... # add a tabledataset
... s1 = [('energy', [1, 4.4, 5.6], 'eV'), ('freq', [0, 43.2, 2E3], 'Hz')]
... x["Spectrum"] = TableDataset(data=s1)
... # create a product store
... pstore = ProductStorage(pool=demopool)
```

(continues on next page)

(continued from previous page)

```
... pstore
ProductStorage { pool= OD{'file:///tmp/demopool_mh':LocalPool { pool= file:///tmp/
↪demopool_mh }} }
```

```
>>> # save the product and get a reference
... prodref = pstore.save(x)
... print(prodref)
ProductRef{ ProductURN=urn:file:///tmp/demopool_mh:fdi.dataset.product.Product:0,␣
↪meta=MetaData{[description = Parameter{ in store <string>, "Description of this product
↪"}, type = Parameter{ Product <string>, "Product Type identificat...}
```

```
>>> # get the urn string
... urn = prodref.urn
... print(urn)    # urn:file:///tmp/demopool_mh:fdi.dataset.product.Product:0
urn:file:///tmp/demopool_mh:fdi.dataset.product.Product:0
```

```
>>> newp = ProductRef(urn).product
... # the new and the old one are equal
... print(newp == x)   # == True
True
```

Context: a Product with References

```
>>> # the reference can be stored in another product of Context class
... p1 = Product(description='p1')
... p2 = Product(description='p2')
... # create an empty mapcontext that can carry references with name labels
... map1 = MapContext(description='product with refs 1')
... # A ProductRef created from a lone product will use a mempool
... pref1 = ProductRef(p1)
... pref1
ProductRef{ ProductURN=urn:mem:///default:fdi.dataset.product.Product:0, meta=None}
```

```
>>> # A productStorage with a pool on disk
... pref2 = pstore.save(p2)
... pref2
ProductRef{ ProductURN=urn:file:///tmp/demopool_mh:fdi.dataset.product.Product:1,␣
↪meta=MetaData{[description = Parameter{ p2 <string>, "Description of this p...
```

```
>>> # how many prodrefs do we have? (do not use len() due to classID, version)
... map1['refs'].size()   # == 0
0
```

```
>>> len(pref1.parents)   # == 0
0
```

```
>>> len(pref2.parents)   # == 0
0
```

```
>>> # add a ref to the contex. every ref has a name in mapcontext
... map1['refs']['spam'] = pref1
... # add the second one
... map1['refs']['egg'] = pref2
... # how many prodrefs do we have? (do not use len() due to classID, version)
... map1['refs'].size()   # == 2
2
```

```
>>> len(pref2.parents)   # == 1
1
```

```
>>> pref2.parents[0] == map1
True
```

```
>>> pref1.parents[0] == map1
True
```

```
>>> # remove a ref
... del map1['refs']['spam']
... # how many prodrefs do we have? (do not use len() due to classID, version)
... map1.refs.size()   # == 1
1
```

```
>>> len(pref1.parents)   # == 0
0
```

```
>>> # add ref2 to another map
... map2 = MapContext(description='product with refs 2')
... map2.refs['also2'] = pref2
... map2['refs'].size()   # == 1
1
```

```
>>> # two parents
... len(pref2.parents)   # == 2
2
```

```
>>> pref2.parents[1] == map2
True
```

Query a ProductStorage

```
>>> # clean possible data left from previous runs
... defaultpoolpath = '/tmp/pool_' + getpass.getuser()
... newpoolpath = '/tmp/newpool_' + getpass.getuser()
... os.system('rm -rf ' + defaultpoolpath)
... os.system('rm -rf ' + newpoolpath)
... PoolManager.getPool(DEFAULT_MEM_POOL).removeAll()
... PoolManager.removeAll()
... # make a productStorage
```

(continues on next page)

(continued from previous page)

```
... defaultpool = 'file://'+defaultpoolpath
... pstore = ProductStorage(defaultpool)
... # make another
... newpoolname = 'file://' + newpoolpath
... pstore2 = ProductStorage(newpoolname)
```

```
>>> # add some products to both storages
... n = 7
... for i in range(n):
...     a0, a1, a2 = 'desc %d' % i, 'fatman %d' % (i*4), 5000+i
...     if i < 3:
...         x = Product(description=a0, instrument=a1)
...         x.meta['extra'] = Parameter(value=a2)
...     elif i < 5:
...         x = Context(description=a0, instrument=a1)
...         x.meta['extra'] = Parameter(value=a2)
... ...
...         x = MapContext(description=a0, instrument=a1)
...         x.meta['extra'] = Parameter(value=a2)
...         x.meta['time'] = Parameter(value=FineTime1(a2))
...     if i < 4:
...         r = pstore.save(x)
...     else:
...         r = pstore2.save(x)
...     print(r.urn)
... # Two pools, 7 products
... # [P P P C] [C M M]
urn:file:///tmp/pool_mh:fdi.dataset.product.Product:0
urn:file:///tmp/pool_mh:fdi.dataset.product.Product:1
urn:file:///tmp/pool_mh:fdi.dataset.product.Product:2
urn:file:///tmp/pool_mh:fdi.pal.context.Context:0
urn:file:///tmp/newpool_mh:fdi.pal.context.Context:0
urn:file:///tmp/newpool_mh:fdi.pal.context.MapContext:0
urn:file:///tmp/newpool_mh:fdi.pal.context.MapContext:1
```

```
>>> # register the new pool above to the  1st productStorage
... pstore.register(newpoolname)
... len(pstore.getPools())   # == 2
2
```

```
>>> # make a query on product metadata, which is the variable 'm'
... # in the query expression, i.e. ``m = product.meta; ...``
... # But '5000 < m["extra"]' does not work. see tests/test.py.
... q = MetaQuery(Product, 'm["extra"] > 5001 and m["extra"] <= 5005')
... # search all pools registered on pstore
... res = pstore.select(q)
... # [2,3,4,5]
... len(res)   # == 4
... [r.product.description for r in res]
['desc 2', 'desc 3', 'desc 4', 'desc 5']
```

```
>>> def t(m):
...     # query is a function
...     import re
...     return re.match('.*n.1.*', m['instrument'].value)
```

```
>>> q = MetaQuery(Product, t)
... res = pstore.select(q)
... # [3,4]
... [r.product.instrument for r in res]
['fatman 12', 'fatman 16']
```

```
>>> # same as above but query is on the product. this is slow.
... q = AbstractQuery(Product, 'p', '"n 1" in p.instrument')
... res = pstore.select(q)
... # [3,4]
... [r.product.instrument for r in res]
['fatman 12', 'fatman 16']
```

```
>>>
```

2.5.3 pns

See the installation and testing sections of the pns page.

CHAPTER

THREE

API DOCUMENT

3.1 API Reference

3.1.1 fdi.dataset package

Submodules

fdi.dataset.abstractcomposite module

fdi.dataset.annotatable module

fdi.dataset.attributable module

fdi.dataset.baseproduct module

fdi.dataset.classes module

fdi.dataset.collectionsMockUp module

fdi.dataset.composite module

fdi.dataset.copyable module

fdi.dataset.dataset module

fdi.dataset.datatypes module

fdi.dataset.datawrapper module

fdi.dataset.deserialize module

fdi.dataset.eq module

fdi.dataset.finetime module

fdi.dataset.listener module

fdi.dataset.metadata module

fdi.dataset.metadataholder module

fdi.dataset.ndprint module

fdi.dataset.odict module

fdi.dataset.product module

fdi.dataset.quantifiable module

fdi.dataset.serializable module

fdi.dataset.yaml2python module

3.1.2 fdi.pal package

Subpackages

fdi.pal.resources package

Submodules

fdi.pal.common module

fdi.pal.comparable module

fdi.pal.context module

fdi.pal.definable module

fdi.pal.httpclientpool module

fdi.pal.localpool module

fdi.pal.mempool module

fdi.pal.pnspoolserver module

fdi.pal.poolmanager module

fdi.pal.productpool module

fdi.pal.productref module

fdi.pal.productstorage module

fdi.pal.query module

fdi.pal.runpnsserver module

fdi.pal.taggable module

fdi.pal.urn module

fdi.pal.versionable module

3.1.3 fdi.pns package

Subpackages

fdi.pns.resources package

Submodules

fdi.pns.jsonio module

fdi.pns.logdict module

fdi.pns.pnsconfig module

fdi.pns.runflaskserver module

fdi.pns.server module

3.1.4 fdi.utils package

Submodules

fdi.utils.checkjson module

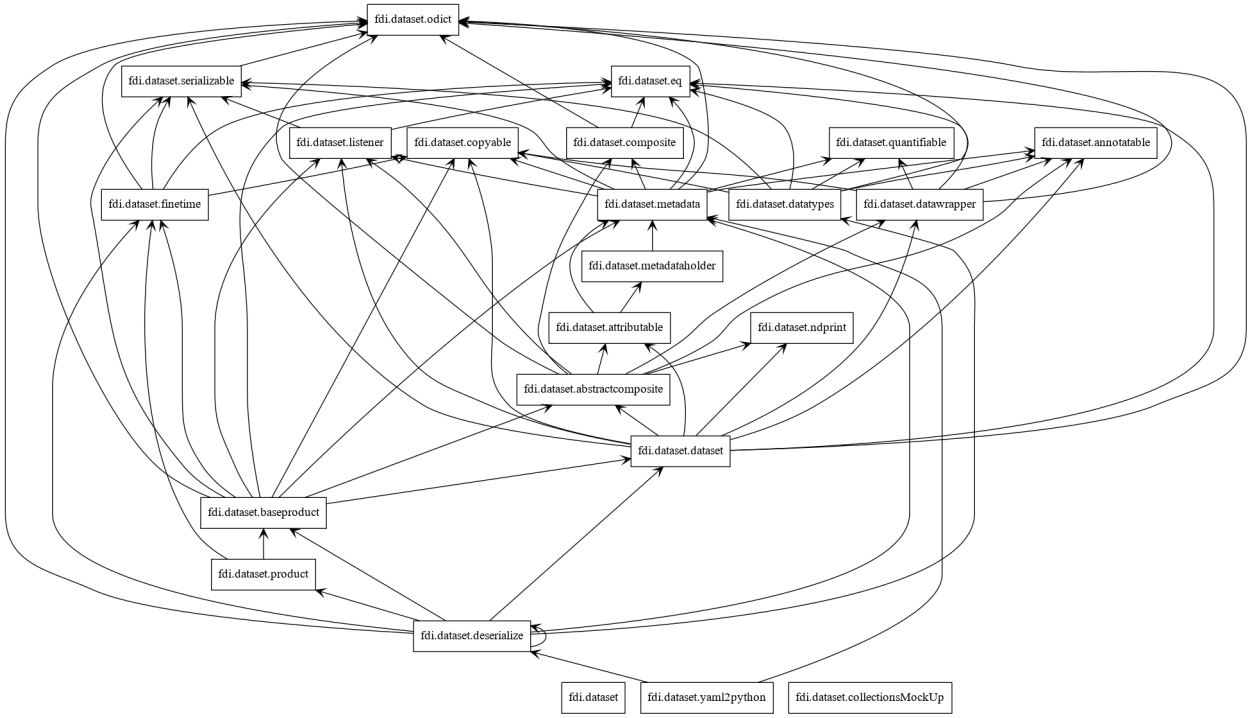
fdi.utils.common module

fdi.utils.options module

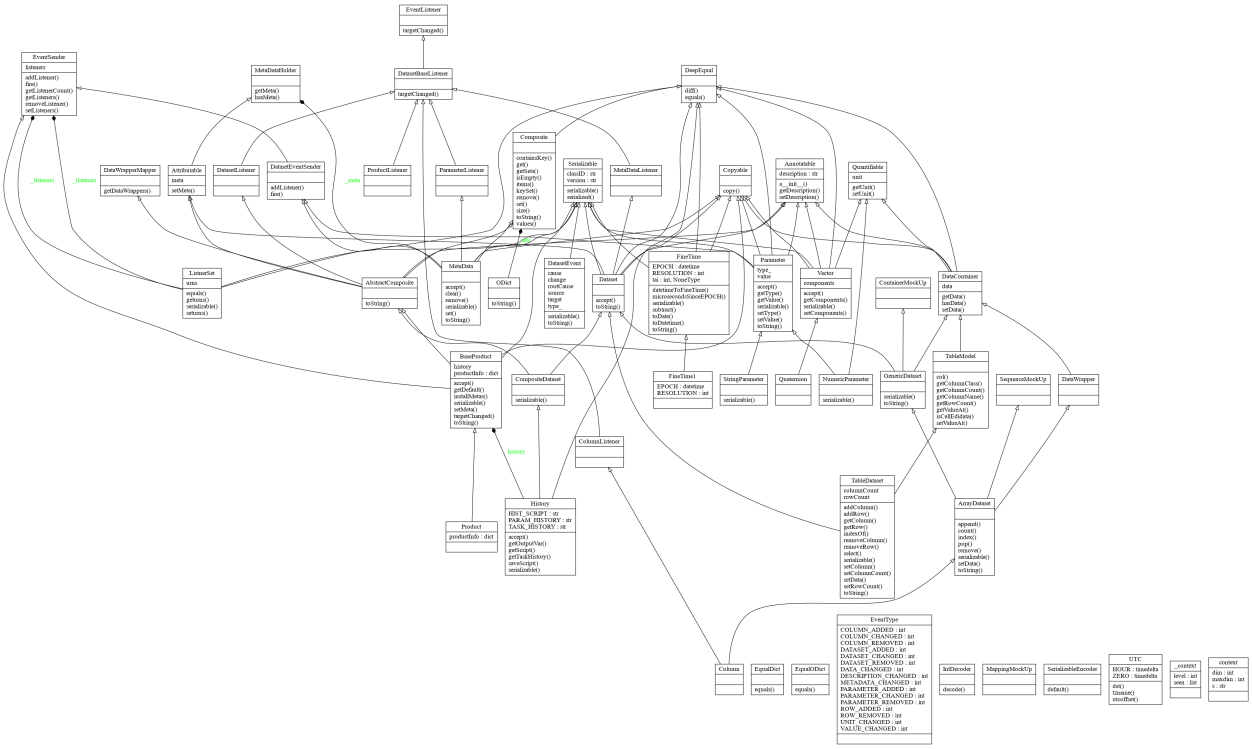
fdi.utils.ydump module

3.1.5 Diagrams

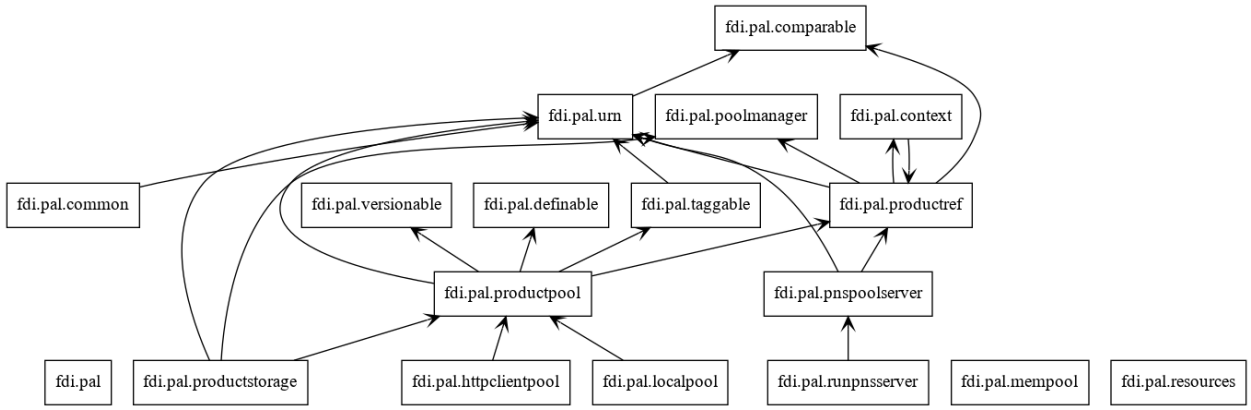
packages_dataset.png



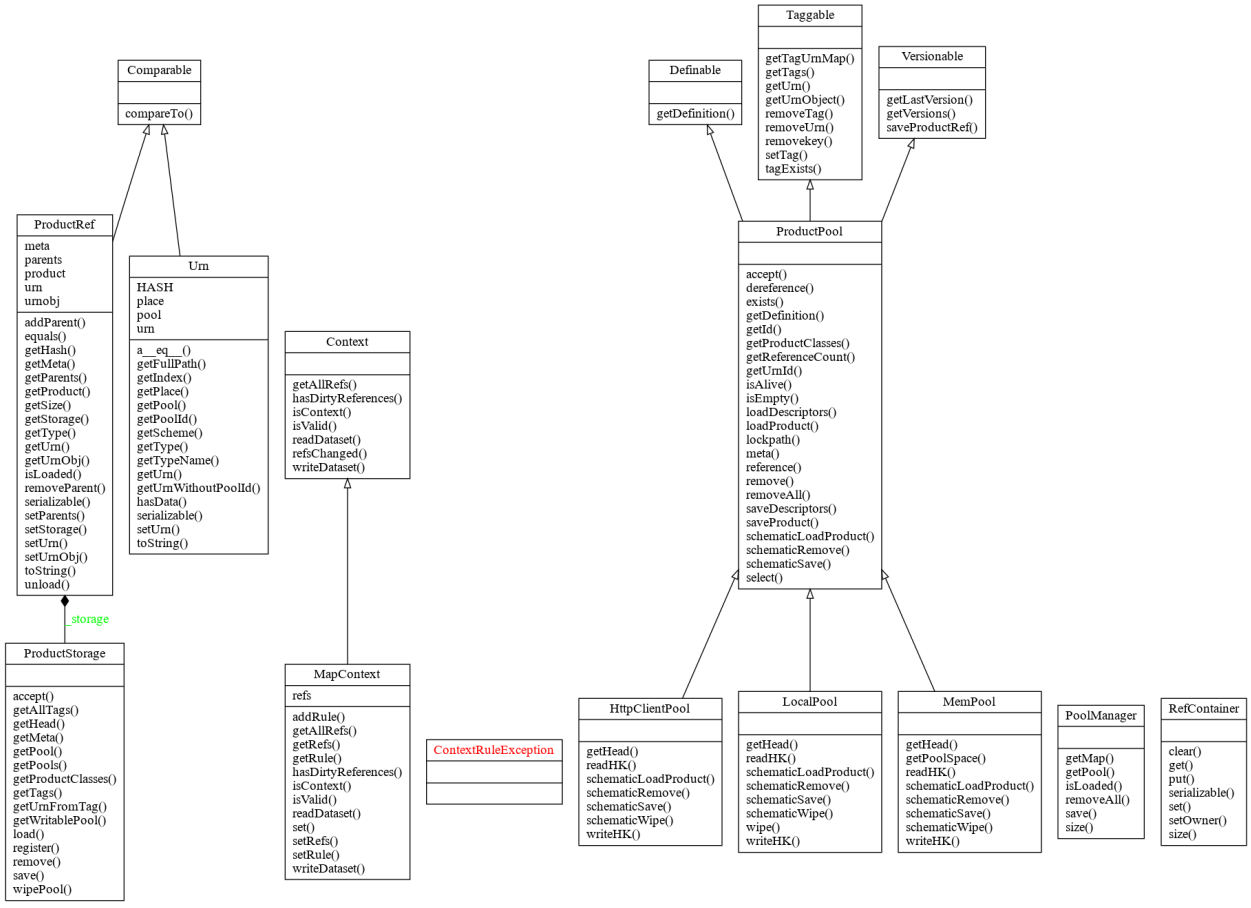
classes_dataset.png



packages_pal.png



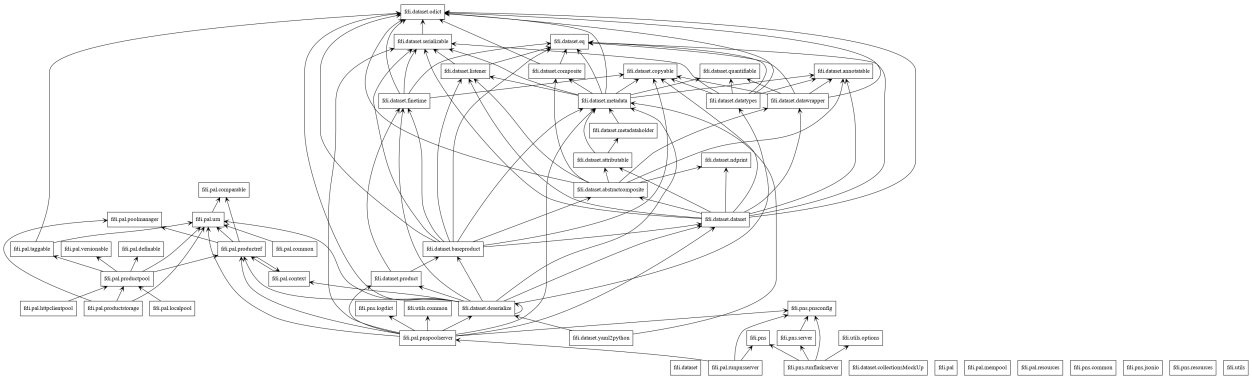
classes_pal.png



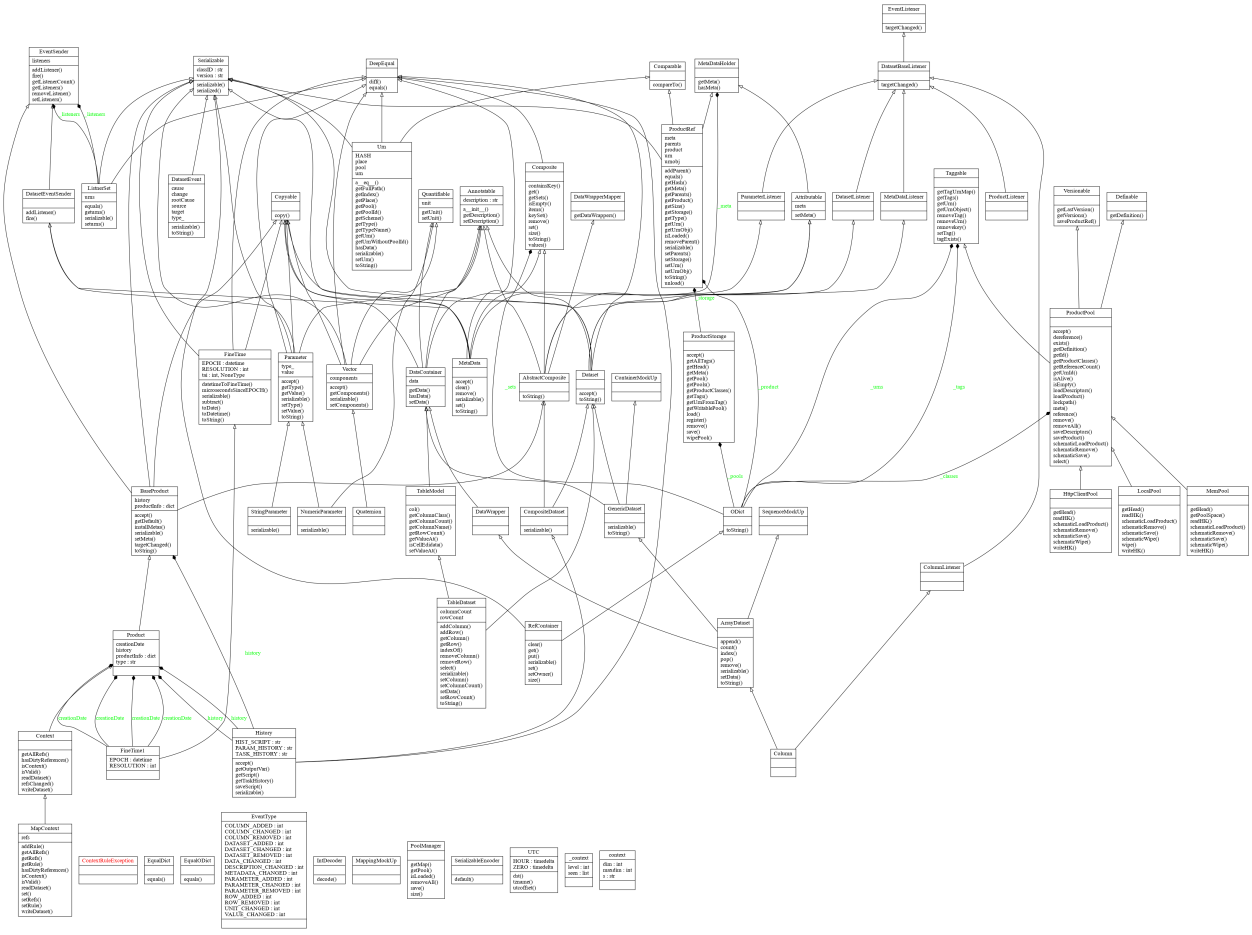
packages_pns.png

classes_pns.png

packages_all.png

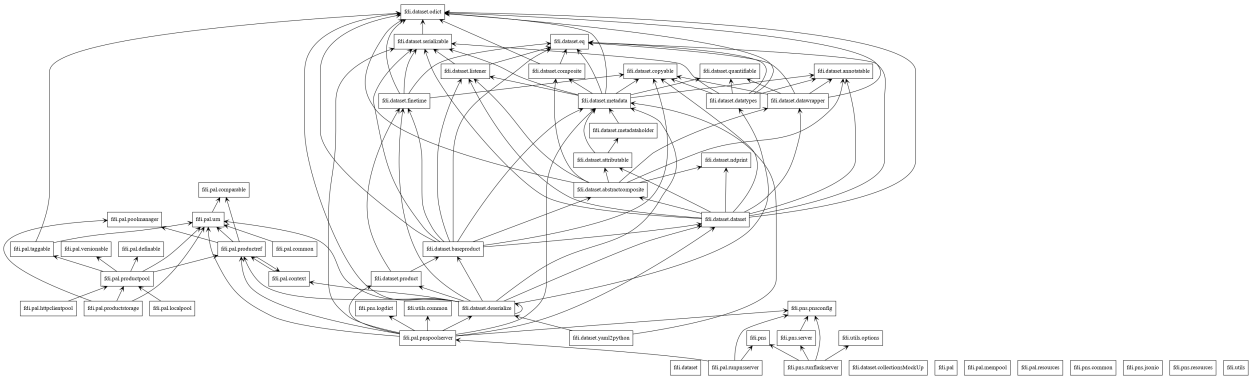


classes_all.png



3.1.6 Indices and tables

- genindex
- modindex
- search



CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search