Omniduct Documentation

Release v1.1.8

Matthew Wardrop

Sep 12, 2019

Contents

1 Supported protocols **1**

2 Installation **3**

3 Quickstart **5**
- 3.1 Task 1: Create a Presto client that connects direct to the database service 5
- 3.2 Task 2: Create a Presto client that connects via ssh to a remote server 6
- 3.3 Task 3: Persist service configuration for use in multiple sessions 7

4 Deployment **9**

5 API & IPython Magics **11**
- 5.1 Core Classes 11
- 5.2 Databases 15
- 5.3 Filesystems 91
- 5.4 Remotes 122
- 5.5 Caches 150
- 5.6 Registry Management 161

6 Extensions and Plug-ins **165**

7 Contributions **167**

8 Indices and tables **169**

Python Module Index **171**

Index **173**

CHAPTER 1

Supported protocols

The currently supported protocols are listed below. The string inside the square brackets after the protocol name (if present) indicates that support for this protocol requires external packages which are not hard-dependencies of *omniduct*. To install them with omniduct, simply add these strings to the list of desired extras as indicated in *Installation*.

- **Databases**
  - Druid [druid]
  - HiveServer2 [hiveserver2]
  - Neo4j (experimental)
  - Presto [presto]
  - PySpark [pyspark]
  - Any SQL database supported by SQL Alchemy (e.g. MySQL, Postgres, Oracle, etc) [sqlalchemy]
- **Filesystems**
  - HDFS [webhdfs]
  - S3 [s3]
  - Local filesystem
- **Remotes (also act as filesystems)**
  - SSH servers, via CLI backend [ssh] or via Paramiko backend [ssh_paramiko]
- REST Services (generic interface)

Adding support for new protocols is straightforward. If your favourite protocol is missing, feel free to contact us for help writing a patch to support it.

Within each class of protocol (database, filesystem, etc), a certain subset of functionality is guaranteed to be consistent across protocols, making them largely interchangeable programmatically. The common API for each protocol class is documented in the *API & IPython Magics* section, along with any exceptions, caveats and extensions for each implementation.

CHAPTER 2

Installation

If your company/organisation has provided a package that wraps around *omniduct* to provide a library of services, then a direct installation of *omniduct* is not required. Otherwise, you can install it using the standard Python package manager: *pip*. If you use Python 3, you may need to change *pip* references to *pip3*, depending on your system configuration.

```
pip install omniduct[<comma separated list of protocols>]
```

For example, if you want access to Presto and HiveServer2, you can run:

```
pip install omniduct[presto,hiveserver2]
```

Omitting the list of protocols (i.e. *pip install omniduct*) will mean that the external dependencies required to interface with the protocols indicated in *Supported protocols* will not be automatically installed. Attempts to use these protocols will throw an error with instructions as to which additional dependencies you will need to install.

To install *omniduct* and all possible dependencies, you can install *omniduct* using:

```
pip install omniduct[all]
```

This is only recommended for casual use, as dragging in unneeded dependencies could lead to complications with other packages on your machine (and is otherwise just generally messy!).

CHAPTER 3

Quickstart

omniduct is designed to be intuitive and uniform in its APIs. As such, insofar as possible, all *Duct* subclasses have a reasonable default configuration, making it possible to quickly create working connections to remote services. Depending on the complexity of your service configuration, it may or may not make sense to use *omniduct*'s registry utilities, and so this quickstart will show you how to directly create *Duct* instances, as well as how to work with a *Duct* registry. Though we only use *PrestoClient* explicitly in the following, since all *Duct* instances have the same basic API, the same methodology will work with all *Duct* subclasses.

If you are looking deploy *omniduct* into production or as part of a company specific package, or want to share your service configuration with others, you will likely also be interested in *Deployment*.

3.1 Task 1: Create a Presto client that connects direct to the database service

Method 1: Via PrestoClient class

```
>>> from omniduct.databases.presto import PrestoClient

>>> pc = PrestoClient(host="<host>", port=8080)

>>> pc.query("SELECT 42")
PrestoClient: Query: Complete after 0.14 sec on 2017-10-13.
    _col0
 0     42

>>> pc.register_magics('presto_local')

# The following assumes that you are using an IPython/Jupyter console
>>> %%presto_local
... {# magics are created and queries rendered using Jinja2 templating #}
... SELECT {{ 4 * 10 + 2 }}
...
```

(continues on next page)

(continued from previous page)

```
presto_local: Query: Complete after 1.20 sec on 2017-10-13.
   _col0
0     42
```

Method 2: Via Duct subclass registry

```
>>> from omniduct import Duct

>>> pc = Duct.for_protocol('presto')(host='<host>', port=8080)

>>> pc.query("SELECT 42")
# ... And all of the rest from above.
```

Method 3: Via DuctRegistry

```
>>> from omniduct import DuctRegistry

>>> duct_registry = DuctRegistry()

>>> pc = duct_registry.new(name='presto_local', protocol='presto',
...                         host='localhost', port=8080, register_magics=True)

>>> # Or: pc = duct_registry['presto_local']

>>> # Or: pc = duct_registry.get_proxy(by_kind=True).databases.presto_local

>>> pc.query("SELECT 42")
presto_local: Query: Complete after 0.14 sec on 2017-10-13.
   _col0
0     42

# The following assumes that you are using an IPython/Jupyter console
>>> %%presto_local
... {# magics are created and queries rendered using Jinja2 templating #}
... SELECT {{ 4 * 10 + 2 }}
...
presto_local: Query: Complete after 1.20 sec on 2017-10-13.
   _col0
0     42
```

3.2 Task 2: Create a Presto client that connects via ssh to a remote server

Method 1: Directly passing 'RemoteClient' instance to PrestoClient constructor

```
>>> from omniduct import Duct

>>> remote = Duct.for_protocol('ssh')(host='<remote_host>', port=22)

>>> pc = Duct.for_protocol('presto')(host='<host_relative_to_remote>',
                                     port=8080, remote=remote)

>>> pc.query("SELECT 42")  # Query sent to port-forwarded remote service
```

(continues on next page)

(continued from previous page)

```
PrestoClient: Query: Complete after 0.14 sec on 2017-10-13.
    _col0
 0     42
```

Method 2: Passing name of ‘RemoteClient‘ instance via Registry

```
>>> from omniduct import DuctRegistry

>>> duct_registry = DuctRegistry()

>>> duct_registry.new('my_server', protocol='ssh', host='<remote_host>', port=22)
<omniduct.remotes.ssh.SSHClient at 0x110bab550>

>>> duct_registry.new('presto_remote', protocol='presto', remote='my_server',
                      host='<host_relative_to_remote>', port=8080)
<omniduct.databases.presto.PrestoClient at 0x110c04a58>

# Query sent to port-forwarded remote service

>>> %%presto_remote
... SELECT 42
...
presto_remote: Query: Connecting: Connected to localhost:8080 on <remote_host>.
presto_remote: Query: Complete after 7.30 sec on 2017-10-13.
   _col0
0     42
```

3.3 Task 3: Persist service configuration for use in multiple sessions

Method 1: Manually import configuration into ‘DuctRegistry‘

```
>>> from omniduct import DuctRegistry

>>> duct_registry = DuctRegistry()

# Specify a YAML configuration verbatim (or the filename of a yaml configuration)
# In this case we create the configuration for the previous task.
>>> duct_registry.register_from_config("""
... remotes:
...     my_server:
...         protocol: ssh
...         host: <remote_host>
... databases:
...     presto_local:
...         protocol: presto
...         host: <host_relative_to_remote>
...         port: 8080
...         remote: my_server
... """)

>>> %%presto_local
... SELECT 42
...
# And so on.
```

Method 2: Save configuration to '~/.omniduct/config', and autoload

Assuming that the above YAML file has been saved to *~/.omniduct/config*, or to a file located at the location pointed to by the *OMNIDUCT_CONFIG* environment variable, you can directly restore your configuration by importing from *omniduct.session*.

```
>>> from omniduct.session import *

>>> presto_local
<omniduct.databases.presto.PrestoClient at 0x110c04a58>

>>> %%presto_local
... SELECT 42

# And so on.
```

CHAPTER 4

Deployment

While Omniduct can be used on its own by manually constructing the services that you need as part of your scripts and packages, it was designed specifically to integrate well into a organisation-specific Python wrapper package that preconfigures the services available within that organisation environment. Typically such deployments would take advantage of Omniduct's *DuctRegistry* to conveniently expose services within such a package.

An example wrapper package is provided alongside the omniduct module to help bootstrap your own wrappers.

If you need any assistance, please do not hesitate to reach out to us via the GitHub issue tracker.

CHAPTER 5

API & IPython Magics

5.1 Core Classes

All protocol implementations are subclasses (directly or indirectly) of *Duct*. This base class manages the basic life-cycle, connection management and protocol registration. When a subclass of *Duct* is loaded into memory, and has at least one protocol name in the *PROTOCOLS* attribute, then *Duct* registers that class into its subclass registry. This class can then be conveniently accessed by: *Duct.for_protocol('<protocol_name>')*. This empowers the accompanying registry tooling bundled with omniduct, as documented in *Registry Management*.

Protocol implementations may also (directly or indirectly) be subclasses of *MagicsProvider*, which provides a common API to registry IPython magics into the user's session. If implemented, the accompanying registry tooling can automatically register these magics, as documented in *Registry Management*.

5.1.1 Duct

class `omniduct.duct.`**`Duct`**(*protocol=None*, *name=None*, *registry=None*, *remote=None*, *host=None*, *port=None*, *username=None*, *password=None*, *cache=None*, *cache_namespace=None*)

Bases: `object`

The abstract base class for all protocol implementations.

This class defines the basic lifecycle of service connections, along with some magic that provides automatic registration of Duct protocol implementations. All connections made by *Duct* instances are lazy, meaning that instantiation is "free", and no protocol connections are made until required by subsequent interactions (i.e. when the value of any attribute in the list of *connection_fields* is accessed). All *Ducts* will automatically connnect and disconnect as required, and so manual intervention is not typically required to maintain connections.

Attributes

- **protocol** (*str*) – The name of the protocol for which this instance was created (especially useful if a *Duct* subclass supports multiple protocols).
- **name** (*str*) – The name given to this *Duct* instance (defaults to class name).

- **host** (*str*) – The host name providing the service (will be '127.0.0.1', if service is port forwarded from remote; use *._host* to see remote host).
- **port** (*int*) – The port number of the service (will be the port-forwarded local port, if relevant; for remote port use *._port*).
- **username** (*str, bool*) – The username to use for the service.
- **password** (*str, bool*) – The password to use for the service.
- **registry** (*None, omniduct.registry.DuctRegistry*) – A reference to a *DuctRegistry* instance for runtime lookup of other services.
- **remote** (*None, omniduct.remotes.base.RemoteClient*) – A reference to a *RemoteClient* instance to manage connections to remote services.
- **cache** (*None, omniduct.caches.base.Cache*) – A reference to a *Cache* instance to add support for caching, if applicable.
- **connection_fields** (*tuple<str>, list<str>*) – A list of instance attributes to monitor for changes, whereupon the *Duct* instance should automatically disconnect. By default, the following attributes are monitored: 'host', 'port', 'remote', 'username', and 'password'.
- **prepared_fields** (*tuple<str>, list<str>*) – A list of instance attributes to be populated (if their values are callable) when the instance first connects to a service. Refer to *Duct.prepare* and *Duct._prepare* for more details. By default, the following attributes are prepared: '_host', '_port', '_username', and '_password'.
- **Additional attributes including 'host', 'port', 'username' and 'password' are**
- **documented inline.**
- **Class Attributes** –

  AUTO_LOGGING_SCOPE (bool): Whether this class should be used by omniduct logging code as a "scope". Should be overridden by subclasses as appropriate.

  DUCT_TYPE (Duct.Type): The type of *Duct* service that is provided by this Duct instance. Should be overridden by subclasses as appropriate.

  PROTOCOLS (list<str>): The name(s) of any protocols that should be associated with this class. Should be overridden by subclasses as appropriate.

class Type

Bases: `enum.Enum`

The *Duct.Type* enum specifies all of the permissible values of *Duct.DUCT_TYPE*. Also determines the order in which ducts are loaded by DuctRegistry.

__init__(*protocol=None*, *name=None*, *registry=None*, *remote=None*, *host=None*, *port=None*, *username=None*, *password=None*, *cache=None*, *cache_namespace=None*)

protocol (str, None): Name of protocol (used by Duct registries to inform Duct instances of how they were instantiated).

name (str, None): The name to used by the *Duct* instance (defaults to class name if not specified).

registry (DuctRegistry, None): The registry to use to lookup remote and/or cache instance specified by name.

remote (str, RemoteClient): The remote by which the ducted service should be contacted.

host (str): The hostname of the service to be used by this client. port (int): The port of the service to be used by this client. username (str, bool, None): The username to authenticate with if necessary.

If True, then users will be prompted at runtime for credentials.

password (str, bool, None): The password to authenticate with if necessary. If True, then users will be prompted at runtime for credentials.

cache(Cache, None): The cache client to be attached to this instance. Cache will only used by specific methods as configured by the client.

cache_namespace(str, None): The namespace to use by default when writing to the cache.

classmethod for_protocol(*protocol*)
Retrieve a *Duct* subclass for a given protocol.

Parameters **protocol** (*str*) – The protocol of interest.

Returns

The appropriate class for the provided, partially constructed with the *protocol* keyword argument set appropriately.

Return type functools.partial object

Raises `DuctProtocolUnknown` – If no class has been defined that offers the named protocol.

prepare()
Prepare a Duct subclass for use (if not already prepared).

This method is called before the value of any of the fields referenced in *self.connection_fields* are retrieved. The fields include, by default: 'host', 'port', 'remote', 'cache', 'username', and 'password'. Subclasses may add or subtract from these special fields.

When called, it first checks whether the instance has already been prepared, and if not calls *_prepare* and then records that the instance has been successfully prepared.

This method may be overridden by subclasses, but provides the following default behaviour:

- Ensures *self.registry*, *self.remote* and *self.cache* values are instances of the right types.
- It replaces string values of *self.remote* and *self.cache* with remotes and caches looked up using *self.registry.lookup*.
- It looks through each of the fields nominated in *self.prepared_fields* and, if the corresponding value is callable, sets the value of that field to result of calling that value with a reference to *self*. By default, *prepared_fields* contains '_host', '_port', '_username', and '_password'.
- Ensures value of self.port is an integer (or None).

_prepare()
This method may be overridden by subclasses, but provides the following default behaviour:

- Ensures *self.registry*, *self.remote* and *self.cache* values are instances of the right types.
- It replaces string values of *self.remote* and *self.cache* with remotes and caches looked up using *self.registry.lookup*.
- It looks through each of the fields nominated in *self.prepared_fields* and, if the corresponding value is callable, sets the value of that field to result of calling that value with a reference to *self*. By default, *prepared_fields* contains '_host', '_port', '_username', and '_password'.
- Ensures value of self.port is an integer (or None).

reset()
Reset this *Duct* instance to its pre-preparation state.

This method disconnects from the service, resets any temporary authentication and restores the values of the attributes listed in *prepared_fields* to their values as of when *Duct.prepare* was called.

Returns A reference to this object.

Return type *Duct* instance

host

The host name providing the service, or '127.0.0.1' if *self.remote* is not *None*, whereupon the service will be port-forwarded locally. You can view the remote hostname using *duct._host*, and change the remote host at runtime using: *duct.host = '<host>'*.

Type str

port

The local port for the service. If *self.remote* is not *None*, the port will be port-forwarded from the remote host. To see the port used on the remote host refer to *duct._port*. You can change the remote port at runtime using: *duct.port = <port>*.

Type int

username

Some services require authentication in order to connect to the service, in which case the appropriate username can be specified. If not specified at instantiation, your local login name will be used. If *True* was provided, you will be prompted to type your username at runtime as necessary. If *False* was provided, then *None* will be returned. You can specify a different username at runtime using: *duct.username = '<username>'*.

Type str

password

Some services require authentication in order to connect to the service, in which case the appropriate password can be specified. If *True* was provided at instantiation, you will be prompted to type your password at runtime when necessary. If *False* was provided, then *None* will be returned. You can specify a different password at runtime using: *duct.password = '<password>'*.

Type str

connect()

Connect to the service backing this client.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Returns A reference to the current object.

Return type *Duct* instance

is_connected()

Check whether this *Duct* instances is currently connected.

This method checks to see whether a *Duct* instance is currently connected. This is performed by verifying that the remote host and port are still accessible, and then by calling *Duct._is_connected*, which should be implemented by subclasses.

Returns Whether this *Duct* instance is currently connected.

Return type bool

disconnect()

Disconnect this client from backing service.

This method is automatically called during reconnections and/or at Python interpreter shutdown. It first calls *Duct._disconnect* (which should be implemented by subclasses) and then notifies the *RemoteClient* subclass, if present, to stop port-forwarding the remote service.

Returns A reference to this object.

Return type *Duct* instance

reconnect()

Disconnects, and then reconnects, this client.

Note: This is equivalent to *duct.disconnect().connect()*.

Returns A reference to this object.

Return type *Duct* instance

5.1.2 MagicsProvider

class omniduct.utils.magics.**MagicsProvider**

Bases: `object`

5.2 Databases

All database clients are expected to be subclasses of *DatabaseClient*, and so will share a common API and inherit a suite of IPython magics. Protocol implementations are also free to add extra methods, which are documented in the "Subclass Reference" section below.

5.2.1 Common API

class omniduct.databases.base.**DatabaseClient**(*session_properties=None*, *templates=None*, *template_context=None*, *default_format_opts=None*, ***kwargs*)

Bases: *omniduct.duct.Duct*, *omniduct.utils.magics.MagicsProvider*

An abstract class providing the common API for all database clients.

Note: *DatabaseClient* subclasses are callable, so that one can use *DatabaseClient(. . .)* as a short-hand for *DatabaseClient.query(. . .)*.

Class Attributes

- **DUCT_TYPE** (*Duct.Type*) – The type of *Duct* protocol implemented by this class.
- **DEFAULT_PORT** (*int*) – The default port for the database service (defined by subclasses).
- **CURSOR_FORMATTERS** (*dict<str, CursorFormatter*) – asdsd
- **DEFAULT_CURSOR_FORMATTER** (*str*) – . . .

Attributes inherited from Duct:

protocol (str): The name of the protocol for which this instance was created (especially useful if a *Duct* subclass supports multiple protocols).

name (str): The name given to this *Duct* instance (defaults to class name).

host (str): The host name providing the service (will be '127.0.0.1', if service is port forwarded from remote; use *._host* to see remote host).

port (int): The port number of the service (will be the port-forwarded local port, if relevant; for remote port use *._port*).

username (str, bool): The username to use for the service. password (str, bool): The password to use for the service. registry (None, omniduct.registry.DuctRegistry): A reference to a

DuctRegistry instance for runtime lookup of other services.

remote (None, omniduct.remotes.base.RemoteClient): A reference to a *RemoteClient* instance to manage connections to remote services.

cache (None, omniduct.caches.base.Cache): A reference to a *Cache* instance to add support for caching, if applicable.

connection_fields (tuple<str>, list<str>): A list of instance attributes to monitor for changes, whereupon the *Duct* instance should automatically disconnect. By default, the following attributes are monitored: 'host', 'port', 'remote', 'username', and 'password'.

prepared_fields (tuple<str>, list<str>): A list of instance attributes to be populated (if their values are callable) when the instance first connects to a service. Refer to *Duct.prepare* and *Duct._prepare* for more details. By default, the following attributes are prepared: '_host', '_port', '_username', and '_password'.

Additional attributes including *host*, *port*, *username* and *password* are documented inline.

Class Attributes:

AUTO_LOGGING_SCOPE (bool): Whether this class should be used by omniduct logging code as a "scope". Should be overridden by subclasses as appropriate.

DUCT_TYPE (Duct.Type): The type of *Duct* service that is provided by this Duct instance. Should be overridden by subclasses as appropriate.

PROTOCOLS (list<str>): The name(s) of any protocols that should be associated with this class. Should be overridden by subclasses as appropriate.

`__init__`(*session_properties=None*, *templates=None*, *template_context=None*, *default_format_opts=None*, ***kwargs*)

protocol (str, None): Name of protocol (used by Duct registries to inform Duct instances of how they were instantiated).

name (str, None): The name to used by the *Duct* instance (defaults to class name if not specified).

registry (DuctRegistry, None): The registry to use to lookup remote and/or cache instance specified by name.

remote (str, RemoteClient): The remote by which the ducted service should be contacted.

host (str): The hostname of the service to be used by this client. port (int): The port of the service to be used by this client. username (str, bool, None): The username to authenticate with if necessary.

If True, then users will be prompted at runtime for credentials.

password (str, bool, None): The password to authenticate with if necessary. If True, then users will be prompted at runtime for credentials.

cache(Cache, None): The cache client to be attached to this instance. Cache will only used by specific methods as configured by the client.

cache_namespace(str, None): The namespace to use by default when writing to the cache.

DatabaseClient Quirks:

session_properties (dict): A mapping of default session properties to values. Interpretation is left up to implementations.

templates (dict): A dictionary of name to template mappings. Additional templates can be added using *.template_add*.

template_context (dict): The default template context to use when rendering templates.

default_format_opts (dict): The default formatting options passed to cursor formatter.

`session_properties`
The default session properties used in statement executions.

Type dict

`classmethod statement_hash`(*statement*, *cleanup=True*)
Retrieve the hash to use to identify query statements to the cache.

Parameters

- **`statement`** (*`str`*) – A string representation of the statement to be hashed.
- **`cleanup`** (*`bool`*) – Whether the statement should first be consistently reformatted using *statement_cleanup*.

Returns The hash used to identify a statement to the cache.

Return type str

`classmethod statement_cleanup`(*statement*)
Clean up statements prior to hash computation.

This classmethod takes an SQL statement and reformats it by consistently removing comments and replacing all whitespace. It is used by the *statement_hash* method to avoid functionally identical queries hitting different cache keys. If the statement's language is not to be SQL, this method should be overloaded appropriately.

Parameters **`statement`** (*`str`*) – The statement to be reformatted/cleaned-up.

Returns The new statement, consistently reformatted.

Return type str

`execute`(*statement*, *wait=True*, *cursor=None*, *session_properties=None*, ***kwargs*)
Execute a statement against this database and return a cursor object.

Where supported by database implementations, this cursor can the be used in future executions, by passing it as the *cursor* keyword argument.

Parameters

- **`statement`** (*`str`*) – The statement to be executed by the query client (possibly templated).
- **`wait`** (*`bool`*) – Whether the cursor should be returned before the server-side query computation is complete and the relevant results downloaded.
- **`cursor`** (*`DBAPI2 cursor`*) – Rather than creating a new cursor, execute the statement against the provided cursor.
- **`session_properties`** (*`dict`*) – Additional session properties and/or overrides to use for this query. Setting a session property value to *None* will cause it to be omitted.
- **`**kwargs`** (*`dict`*) – Extra keyword arguments to be passed on to *_execute*, as implemented by subclasses.
- **`template`** (*`bool`*) – Whether the statement should be treated as a Jinja2 template. [Used by *render_statement* decorator.]
- **`context`** (*`dict`*) – The context in which the template should be evaluated (a dictionary of parameters to values). [Used by *render_statement* decorator.]

- **use_cache** (*bool*) – True or False (default). Whether to use the cache (if present). [Used by *cached_method* decorator.]
- **renew** (*bool*) – True or False (default). If cache is being used, renew it before returning stored value. [Used by *cached_method* decorator.]
- **cleanup** (*bool*) – Whether statement should be cleaned up before computing the hash used to cache results. [Used by *cached_method* decorator.]

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

query (*statement*, *format=None*, *format_opts={}*, *use_cache=True*, ***kwargs*)
Execute a statement against this database and collect formatted data.

Parameters

- **statement** (*str*) – The statement to be executed by the query client (possibly templated).
- **format** (*str*) – A subclass of CursorFormatter, or one of: 'pandas', 'hive', 'csv', 'tuple' or 'dict'. Defaults to *self.DEFAULT_CURSOR_FORMATTER*.
- **format_opts** (*dict*) – A dictionary of format-specific options.
- **use_cache** (*bool*) – Whether to cache the cursor returned by *DatabaseClient.execute()* (overrides the default of False for *.execute()*). (default=True)
- ****kwargs** (*dict*) – Additional arguments to pass on to *DatabaseClient.execute()*.

Returns The results of the query formatted as nominated.

stream (*statement*, *format=None*, *format_opts={}*, *batch=None*, ***kwargs*)
Execute a statement against this database and stream formatted results.

This method returns a generator over objects representing rows in the result set. If *batch* is not *None*, then the iterator will be over lists of length *batch* containing formatted rows.

Parameters

- **statement** (*str*) – The statement to be executed against the database.
- **format** (*str*) – A subclass of CursorFormatter, or one of: 'pandas', 'hive', 'csv', 'tuple' or 'dict'. Defaults to *self.DEFAULT_CURSOR_FORMATTER*.
- **format_opts** (*dict*) – A dictionary of format-specific options.
- **batch** (*int*) – If not *None*, the number of rows from the resulting cursor to be returned at once.
- ****kwargs** (*dict*) – Additional keyword arguments to pass onto *DatabaseClient.execute*.

Returns

An iterator over objects of the nominated format or, if batched, a list of such objects.

Return type iterator

stream_to_file (*statement*, *file*, *format='csv'*, *fs=None*, ***kwargs*)
Execute a statement against this database and stream results to a file.

This method is a wrapper around *DatabaseClient.stream* that enables the iterative writing of cursor results to a file. This is especially useful when there are a very large number of results, and loading them all into memory would require considerable resources. Note that 'csv' is the default format for this method (rather than *pandas*).

Parameters

- **statement** (`str`) – The statement to be executed against the database.
- **file** (`str, file-like-object`) – The filename where the data should be written, or an open file-like resource.
- **format** (`str`) – The format to be used ('csv' by default). Format options can be passed via ***kwargs*.
- **fs** (`None, FileSystemClient`) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- ****kwargs** – Additional keyword arguments to pass onto *DatabaseClient.stream*.

execute_from_file(*file*, *fs=None*, ***kwargs*)
Execute a statement stored in a file.

Parameters

- **file** (`str, file-like-object`) – The path of the file containing the query statement to be executed against the database, or an open file-like resource.
- **fs** (`None, FileSystemClient`) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- ****kwargs** (`dict`) – Extra keyword arguments to pass on to *DatabaseClient.execute*.

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

query_from_file(*file*, *fs=None*, ***kwargs*)
Query using a statement stored in a file.

Parameters

- **file** (`str, file-like-object`) – The path of the file containing the query statement to be executed against the database, or an open file-like resource.
- **fs** (`None, FileSystemClient`) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- ****kwargs** (`dict`) – Extra keyword arguments to pass on to *DatabaseClient.query*.

Returns The results of the query formatted as nominated.

Return type object

template_names
A list of names associated with the templates associated with this client.

Type list

template_add(*name*, *body*)
Add a named template to the internal dictionary of templates.

Note: Templates are interpreted as *jinja2* templates. See *.template_render* for more information.

Parameters

- **name** (`str`) – The name of the template.
- **body** (`str`) – The (typically) multiline body of the template.

Returns A reference to this object.

Return type *PrestoClient*

template_get (*name*)

Retrieve a named template.

Parameters **name** (*str*) – The name of the template to retrieve.

Raises ValueError – If *name* is not associated with a template.

Returns The requested template.

Return type str

template_variables (*name_or_statement*, *by_name=False*)

Return the set of undeclared variables required for this template.

Parameters

- **name_or_statement** (*str*) – The name of a template (if *by_name* is True) or else a string representation of a *jinja2* template.
- **by_name** (*bool*) – *True* if *name_or_statement* should be interpreted as a template name, or *False* (default) if *name_or_statement* should be interpreted as a template body.

Returns A set of names which the template requires to be rendered.

Return type set<str>

template_render (*name_or_statement*, *context=None*, *by_name=False*, *cleanup=False*, *meta_only=False*)

Render a template by name or value.

In addition to the *jinja2* templating syntax, described in more detail in the official *jinja2* documentation, a meta-templating extension is also provided. This meta-templating allows you to reference other reference other templates. For example, if you had two SQL templates named 'template_a' and 'template_b', then you could render them into one SQL query using (for example):

```
.template_render('''
WITH
    a AS (
        {{{template_a}}}
    ),
    b AS (
        {{{template_b}}}
    )
SELECT *
FROM a
JOIN b ON a.x = b.x
''')
```

Note that template substitution in this way is iterative, so you can chain template embedding, provided that such embedding is not recursive.

Parameters

- **name_or_statement** (*str*) – The name of a template (if *by_name* is True) or else a string representation of a *jinja2* template.
- **context** (*dict, None*) – A dictionary to use as the template context. If not specified, an empty dictionary is used.

- **by_name** (*bool*) – *True* if *name_or_statement* should be interpreted as a template name, or *False* (default) if *name_or_statement* should be interpreted as a template body.
- **cleanup** (*bool*) – *True* if the rendered statement should be formatted, *False* (default) otherwise
- **meta_only** (*bool*) – *True* if rendering should only progress as far as rendering nested templates (i.e. don't actually substitute in variables from the context); *False* (default) otherwise.

Returns The rendered template.

Return type str

execute_from_template (*name*, *context=None*, ***kwargs*)
Render and then execute a named template.

Parameters

- **name** (*str*) – The name of the template to be rendered and executed.
- **context** (*dict*) – The context in which the template should be rendered.
- ****kwargs** (*dict*) – Additional parameters to pass to *.execute()*.

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

query_from_template (*name*, *context=None*, ***kwargs*)
Render and then query using a named tempalte.

Parameters

- **name** (*str*) – The name of the template to be rendered and used to query the database.
- **context** (*dict*) – The context in which the template should be rendered.
- ****kwargs** (*dict*) – Additional parameters to pass to *.query()*.

Returns The results of the query formatted as nominated.

Return type object

query_to_table (*statement*, *table*, *if_exists='fail'*, ***kwargs*)
Run a query and store the results in a table in this database.

Parameters

- **statement** – The statement to be executed.
- **table** (*str*) – The name of the table into which the dataframe should be uploaded.
- **if_exists** (*str*) – if nominated table already exists: 'fail' to do nothing, 'replace' to drop, recreate and insert data into new table, and 'append' to add data from this table into the existing table.
- ****kwargs** (*dict*) – Additional keyword arguments to pass onto *DatabaseClient._query_to_table*.

Returns The cursor object associated with the execution.

Return type DB-API cursor

dataframe_to_table (*df*, *table*, *if_exists='fail'*, ***kwargs*)
Upload a local pandas dataframe into a table in this database.

Parameters

- **df** (*pandas.DataFrame*) – The dataframe to upload into the database.
- **table** (*str, ParsedNamespaces*) – The name of the table into which the dataframe should be uploaded.
- **if_exists** (*str*) – if nominated table already exists: 'fail' to do nothing, 'replace' to drop, recreate and insert data into new table, and 'append' to add data from this table into the existing table.
- ****kwargs** (*dict*) – Additional keyword arguments to pass onto *DatabaseClient._dataframe_to_table*.

table_list(*namespace=None*, *renew=True*, ***kwargs*)
Return a list of table names in the data source as a DataFrame.

Parameters

- **namespace** (*str*) – The namespace in which to look for tables.
- **renew** (*bool*) – Whether to renew the table list or use cached results (default: True).
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns The names of schemas in this database.

Return type list<str>

table_exists(*table*, *renew=True*, ***kwargs*)
Check whether a table exists.

Parameters

- **table** (*str*) – The table for which to check.
- **renew** (*bool*) – Whether to renew the table list or use cached results (default: True).
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns *True* if table exists, and *False* otherwise.

Return type bool

table_drop(*table*, ***kwargs*)
Remove a table from the database.

Parameters

- **table** (*str*) – The table to drop.
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns The cursor associated with this execution.

Return type DB-API cursor

table_desc(*table*, *renew=True*, ***kwargs*)
Describe a table in the database.

Parameters

- **table** (*str*) – The table to describe.
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns A dataframe description of the table.

Return type pandas.DataFrame

`table_head`(*table*, *n=10*, *renew=True*, ***kwargs*)
Retrieve the first *n* rows from a table.

Parameters

- **`table`** (*`str`*) – The table from which to extract data.
- **`n`** (*`int`*) – The number of rows to extract.
- **`renew`** (*`bool`*) – Whether to renew the table list or use cached results (default: True).
- **`**kwargs`** (*`dict`*) – Additional arguments passed through to implementation.

Returns

A dataframe representation of the first *n* rows of the nominated table.

Return type pandas.DataFrame

`table_props`(*table*, *renew=True*, ***kwargs*)
Retrieve the properties associated with a table.

Parameters

- **`table`** (*`str`*) – The table from which to extract data.
- **`renew`** (*`bool`*) – Whether to renew the table list or use cached results (default: True).
- **`**kwargs`** (*`dict`*) – Additional arguments passed through to implementation.

Returns

A dataframe representation of the table properties.

Return type pandas.DataFrame

`connect`()
Connect to the service backing this client.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Returns A reference to the current object.

Return type *Duct* instance

`disconnect`()
Disconnect this client from backing service.

This method is automatically called during reconnections and/or at Python interpreter shutdown. It first calls *Duct._disconnect* (which should be implemented by subclasses) and then notifies the *RemoteClient* subclass, if present, to stop port-forwarding the remote service.

Returns A reference to this object.

Return type *Duct* instance

`is_connected`()
Check whether this *Duct* instances is currently connected.

This method checks to see whether a *Duct* instance is currently connected. This is performed by verifying that the remote host and port are still accessible, and then by calling *Duct._is_connected*, which should be implemented by subclasses.

Returns Whether this *Duct* instance is currently connected.

Return type bool

prepare()

Prepare a Duct subclass for use (if not already prepared).

This method is called before the value of any of the fields referenced in *self.connection_fields* are retrieved. The fields include, by default: 'host', 'port', 'remote', 'cache', 'username', and 'password'. Subclasses may add or subtract from these special fields.

When called, it first checks whether the instance has already been prepared, and if not calls *_prepare* and then records that the instance has been successfully prepared.

DatabaseClient Quirks: This method may be overridden by subclasses, but provides the following default behaviour:

- Ensures *self.registry*, *self.remote* and *self.cache* values are instances of the right types.
- It replaces string values of *self.remote* and *self.cache* with remotes and caches looked up using *self.registry.lookup*.
- It looks through each of the fields nominated in *self.prepared_fields* and, if the corresponding value is callable, sets the value of that field to result of calling that value with a reference to *self*. By default, *prepared_fields* contains '_host', '_port', '_username', and '_password'.
- Ensures value of self.port is an integer (or None).

register_magics(*base_name=None*)

The following magic functions will be registered (assuming that the base name is chosen to be 'hive'): - Cell Magics:

- *%%hive*: For querying the database.
- *%%hive.execute*: For executing a statement against the database.
- ***%%hive.stream*: For executing a statement against the database,** and streaming the results.
- *%%hive.template*: The defining a new template.
- *%%hive.render*: Render a provided query statement.

- **Line Magics:**
  - *%hive*: For querying the database using a named template.
  - ***%hive.execute*: For executing a named template statement against** the database.
  - ***%hive.stream*: For executing a named template against the database,** and streaming the results.
  - *%hive.render*: Render a provided a named template.
  - *%hive.desc*: Describe the table nominated.
  - *%hive.head*: Return the first rows in a specified table.
  - *%hive.props*: Show the properties specified for a nominated table.

Documentation for these magics is provided online.

5.2.2 IPython Magics

While it is possible in an IPython/Jupter notebook session to write code along the lines of:

```
results = db_client.query("""
SELECT *
FROM table
WHERE condition = 1
""", format='pandas', ...)
```

manually encapsulating queries in quotes quickly becomes tiresome and cluttered. We therefore expose most functionality as IPython magic functions. For example, the above code could instad be rendered (assuming magic functions have been registered under the name *db_client*):

```
%%db_client results format='pandas' ...
SELECT *
FROM table
WHERE condition = 1
```

Especially when combined with templating, this can greatly improve the readability of your code.

In the following, all of the provided magic functions are listed along with the equivalent programmatic code. Note that all arguments are passed in as space-separated tokens after the magic's name. Position-arguments are always interpreted as strings and keyword arguments are expected to be provided in the form '<key>=<value>', where the <value> will be run as Python code and the resulting value passed on to the underlying function/method as:

```
db_client.method(..., key=eval('<value>'), ...)
```

Where present in the following, arguments in square brackets after the magic name are the options specific to the magic function, and an ellipsis ('. . . ') indicates that any additional keyword arguments will be passed on to the appropriate method.

Querying

```
%%<name> [variable=None show='head' transpose=False ...]
SELECT *
FROM table
WHERE condition = 1
```

This magic is equivalent to calling `db_client.query("<sql>", ...)`, with the following magic-specific parameters offering additional flexibility:

- **variable (str):** The name of the local variable where the output should be stored (typically not referenced directly by name)
- **show (str, int, None):** What should be shown if variable is specified (if not the entire output is returned). Allowed values are 'all', 'head' (first 5 rows), 'none', or an integer which specifies the number of rows to be shown.
- **transpose (bool):** If format is pandas, whether the shown results, as defined above, should be transposed. Data stored into variable is never transposed.

There is also a line-magic version if you are querying using an existing template:

```
results = %<name> variable='<template_name>' ...
```

which is equivalent to `db_client.query_from_template('<template_name>', context=locals())`. Note that one would typically pass this the template name as a position argument, i.e. `%<name> <template_name>`.

Executing

```
%%<name>.execute [variable=None ...]
INSERT INTO database.table (field1, field2) VALUES (1, 2);
```

This magic is equivalent to `db_client.execute('<sql>', ...)`, with the *variable* argument functioning as previously for the query magic.

As for the query magic, there is also a template version:

Streaming

```
%%<name>.stream [variable=None ...]
SELECT *
FROM table
WHERE condition = 1
```

This magic is equivalent to `db_client.stream('<sql>', ...)`, with the *variable* argument functioning as previously for the query magic. Keep in mind that the value returned from this method is a generator object.

As for the query magic, there is also a template version:

Templating

To create a new template:

```
%%<name>.template <template_name>
SELECT *
FROM table
WHERE condition = 1
```

which is equivalent to `db_client.add_template("<template_name>", "<sql>")`.

You can render a template in the cell body using current context (or specified context):

```
%%<name>.render [context=None, show=True]
SELECT 1 FROM test
```

or if the template has already been created, you can render it directly by name:

```
%<name>.render [name=None, context=None, show=True]
```

In both cases, the *context* and *show* parameters respectively control the context from which template variables are extracted and whether the rendered template should be shown (printed to screen) or returned as a string.

Table properties

todo Resolve what to keep and dump here.

```
%%<name>.desc
SELECT 1 FROM test
```

```
%%<name>.head
SELECT 1 FROM test
```

```
%%<name>.props
```

5.2.3 Subclass Reference

For comprehensive documentation on any particular subclass, please refer to one of the below documents.

DruidClient

class omniduct.databases.druid.**DruidClient**(*session_properties=None*, *templates=None*, *template_context=None*, *default_format_opts=None*, ***kwargs*)

Bases: *omniduct.databases.base.DatabaseClient*

This Duct connects to a Druid server using the *pydruid* python library.

Attributes inherited from Duct:

protocol (str): The name of the protocol for which this instance was created (especially useful if a *Duct* subclass supports multiple protocols).

name (str): The name given to this *Duct* instance (defaults to class name).

host (str): The host name providing the service (will be '127.0.0.1', if service is port forwarded from remote; use *._host* to see remote host).

port (int): The port number of the service (will be the port-forwarded local port, if relevant; for remote port use *._port*).

username (str, bool): The username to use for the service. password (str, bool): The password to use for the service. registry (None, omniduct.registry.DuctRegistry): A reference to a

DuctRegistry instance for runtime lookup of other services.

remote (None, omniduct.remotes.base.RemoteClient): A reference to a *RemoteClient* instance to manage connections to remote services.

cache (None, omniduct.caches.base.Cache): A reference to a *Cache* instance to add support for caching, if applicable.

connection_fields (tuple<str>, list<str>): A list of instance attributes to monitor for changes, whereupon the *Duct* instance should automatically disconnect. By default, the following attributes are monitored: 'host', 'port', 'remote', 'username', and 'password'.

prepared_fields (tuple<str>, list<str>): A list of instance attributes to be populated (if their values are callable) when the instance first connects to a service. Refer to *Duct.prepare* and *Duct._prepare* for more details. By default, the following attributes are prepared: '_host', '_port', '_username', and '_password'.

Additional attributes including *host*, *port*, *username* and *password* are documented inline.

Class Attributes:

AUTO_LOGGING_SCOPE (bool): Whether this class should be used by omniduct logging code as a "scope". Should be overridden by subclasses as appropriate.

DUCT_TYPE (Duct.Type): The type of *Duct* service that is provided by this Duct instance. Should be overridden by subclasses as appropriate.

PROTOCOLS (list<str>): The name(s) of any protocols that should be associated with this class. Should be overridden by subclasses as appropriate.

class Type

Bases: `enum.Enum`

The *Duct.Type* enum specifies all of the permissible values of *Duct.DUCT_TYPE*. Also determines the order in which ducts are loaded by DuctRegistry.

__init__(*session_properties=None*, *templates=None*, *template_context=None*, *default_format_opts=None*, ***kwargs*)

protocol (str, None): Name of protocol (used by Duct registries to inform Duct instances of how they were instantiated).

name (str, None): The name to used by the *Duct* instance (defaults to class name if not specified).

registry (DuctRegistry, None): The registry to use to lookup remote and/or cache instance specified by name.

remote (str, RemoteClient): The remote by which the ducted service should be contacted.

host (str): The hostname of the service to be used by this client. port (int): The port of the service to be used by this client. username (str, bool, None): The username to authenticate with if necessary.

If True, then users will be prompted at runtime for credentials.

password (str, bool, None): The password to authenticate with if necessary. If True, then users will be prompted at runtime for credentials.

cache(Cache, None): The cache client to be attached to this instance. Cache will only used by specific methods as configured by the client.

cache_namespace(str, None): The namespace to use by default when writing to the cache.

DatabaseClient Quirks:

session_properties (dict): A mapping of default session properties to values. Interpretation is left up to implementations.

templates (dict): A dictionary of name to template mappings. Additional templates can be added using *.template_add*.

template_context (dict): The default template context to use when rendering templates.

default_format_opts (dict): The default formatting options passed to cursor formatter.

connect()

Connect to the service backing this client.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Returns A reference to the current object.

Return type *Duct* instance

dataframe_to_table(*df*, *table*, *if_exists='fail'*, ***kwargs*)

Upload a local pandas dataframe into a table in this database.

Parameters

- **df** (*pandas.DataFrame*) – The dataframe to upload into the database.
- **table** (*str, ParsedNamespaces*) – The name of the table into which the dataframe should be uploaded.

- **if_exists** (*str*) – if nominated table already exists: 'fail' to do nothing, 'replace' to drop, recreate and insert data into new table, and 'append' to add data from this table into the existing table.
- ****kwargs** (*dict*) – Additional keyword arguments to pass onto *DatabaseClient._dataframe_to_table*.

disconnect()

Disconnect this client from backing service.

This method is automatically called during reconnections and/or at Python interpreter shutdown. It first calls *Duct._disconnect* (which should be implemented by subclasses) and then notifies the *RemoteClient* subclass, if present, to stop port-forwarding the remote service.

Returns A reference to this object.

Return type *Duct* instance

execute(*statement*, *wait=True*, *cursor=None*, *session_properties=None*, ***kwargs*)

Execute a statement against this database and return a cursor object.

Where supported by database implementations, this cursor can the be used in future executions, by passing it as the *cursor* keyword argument.

Parameters

- **statement** (*str*) – The statement to be executed by the query client (possibly templated).
- **wait** (*bool*) – Whether the cursor should be returned before the server-side query computation is complete and the relevant results downloaded.
- **cursor** (*DBAPI2 cursor*) – Rather than creating a new cursor, execute the statement against the provided cursor.
- **session_properties** (*dict*) – Additional session properties and/or overrides to use for this query. Setting a session property value to *None* will cause it to be omitted.
- ****kwargs** (*dict*) – Extra keyword arguments to be passed on to *_execute*, as implemented by subclasses.
- **template** (*bool*) – Whether the statement should be treated as a Jinja2 template. [Used by *render_statement* decorator.]
- **context** (*dict*) – The context in which the template should be evaluated (a dictionary of parameters to values). [Used by *render_statement* decorator.]
- **use_cache** (*bool*) – True or False (default). Whether to use the cache (if present). [Used by *cached_method* decorator.]
- **renew** (*bool*) – True or False (default). If cache is being used, renew it before returning stored value. [Used by *cached_method* decorator.]
- **cleanup** (*bool*) – Whether statement should be cleaned up before computing the hash used to cache results. [Used by *cached_method* decorator.]

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

execute_from_file(*file*, *fs=None*, ***kwargs*)

Execute a statement stored in a file.

Parameters

- **file** (*str, file-like-object*) – The path of the file containing the query statement to be executed against the database, or an open file-like resource.
- **fs** (*None,* FileSystemClient) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- ****kwargs** (*dict*) – Extra keyword arguments to pass on to *DatabaseClient.execute*.

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

execute_from_template(*name*, *context=None*, ***kwargs*)
Render and then execute a named template.

Parameters

- **name** (*str*) – The name of the template to be rendered and executed.
- **context** (*dict*) – The context in which the template should be rendered.
- ****kwargs** (*dict*) – Additional parameters to pass to *.execute()*.

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

classmethod for_protocol(*protocol*)
Retrieve a *Duct* subclass for a given protocol.

Parameters protocol (*str*) – The protocol of interest.

Returns

The appropriate class for the provided, partially constructed with the *protocol* keyword argument set appropriately.

Return type functools.partial object

Raises DuctProtocolUnknown – If no class has been defined that offers the named protocol.

host
The host name providing the service, or '127.0.0.1' if *self.remote* is not *None*, whereupon the service will be port-forwarded locally. You can view the remote hostname using *duct._host*, and change the remote host at runtime using: *duct.host = '<host>'*.

Type str

is_connected()
Check whether this *Duct* instances is currently connected.

This method checks to see whether a *Duct* instance is currently connected. This is performed by verifying that the remote host and port are still accessible, and then by calling *Duct._is_connected*, which should be implemented by subclasses.

Returns Whether this *Duct* instance is currently connected.

Return type bool

password
Some services require authentication in order to connect to the service, in which case the appropriate password can be specified. If *True* was provided at instantiation, you will be prompted to type your password at runtime when necessary. If *False* was provided, then *None* will be returned. You can specify a different password at runtime using: *duct.password = '<password>'*.

Type str

port
The local port for the service. If *self.remote* is not *None*, the port will be port-forwarded from the remote host. To see the port used on the remote host refer to *duct._port*. You can change the remote port at runtime using: *duct.port = <port>*.

Type int

prepare()
Prepare a Duct subclass for use (if not already prepared).

This method is called before the value of any of the fields referenced in *self.connection_fields* are retrieved. The fields include, by default: 'host', 'port', 'remote', 'cache', 'username', and 'password'. Subclasses may add or subtract from these special fields.

When called, it first checks whether the instance has already been prepared, and if not calls *_prepare* and then records that the instance has been successfully prepared.

DruidClient Quirks: This method may be overridden by subclasses, but provides the following default behaviour:

- Ensures *self.registry*, *self.remote* and *self.cache* values are instances of the right types.
- It replaces string values of *self.remote* and *self.cache* with remotes and caches looked up using *self.registry.lookup*.
- It looks through each of the fields nominated in *self.prepared_fields* and, if the corresponding value is callable, sets the value of that field to result of calling that value with a reference to *self*. By default, *prepared_fields* contains '_host', '_port', '_username', and '_password'.
- Ensures value of self.port is an integer (or None).

query(*statement*, *format=None*, *format_opts={}*, *use_cache=True*, ***kwargs*)
Execute a statement against this database and collect formatted data.

Parameters

- **statement** (`str`) – The statement to be executed by the query client (possibly templated).
- **format** (`str`) – A subclass of CursorFormatter, or one of: 'pandas', 'hive', 'csv', 'tuple' or 'dict'. Defaults to *self.DEFAULT_CURSOR_FORMATTER*.
- **format_opts** (`dict`) – A dictionary of format-specific options.
- **use_cache** (`bool`) – Whether to cache the cursor returned by *DatabaseClient.execute()* (overrides the default of False for *.execute()*). (default=True)
- ****kwargs** (`dict`) – Additional arguments to pass on to *DatabaseClient.execute()*.

Returns The results of the query formatted as nominated.

query_from_file(*file*, *fs=None*, ***kwargs*)
Query using a statement stored in a file.

Parameters

- **file** (`str, file-like-object`) – The path of the file containing the query statement to be executed against the database, or an open file-like resource.
- **fs** (`None, FileSystemClient`) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- ****kwargs** (`dict`) – Extra keyword arguments to pass on to *DatabaseClient.query*.

Returns The results of the query formatted as nominated.

Return type object

`query_from_template`(*name*, *context=None*, ***kwargs*)
Render and then query using a named tempalte.

Parameters

- **`name`** (*`str`*) – The name of the template to be rendered and used to query the database.
- **`context`** (*`dict`*) – The context in which the template should be rendered.
- **`**kwargs`** (*`dict`*) – Additional parameters to pass to *.query()*.

Returns The results of the query formatted as nominated.

Return type object

`query_to_table`(*statement*, *table*, *if_exists='fail'*, ***kwargs*)
Run a query and store the results in a table in this database.

Parameters

- **`statement`** – The statement to be executed.
- **`table`** (*`str`*) – The name of the table into which the dataframe should be uploaded.
- **`if_exists`** (*`str`*) – if nominated table already exists: 'fail' to do nothing, 'replace' to drop, recreate and insert data into new table, and 'append' to add data from this table into the existing table.
- **`**kwargs`** (*`dict`*) – Additional keyword arguments to pass onto *DatabaseClient._query_to_table*.

Returns The cursor object associated with the execution.

Return type DB-API cursor

`reconnect`()
Disconnects, and then reconnects, this client.

Note: This is equivalent to *duct.disconnect().connect()*.

Returns A reference to this object.

Return type *Duct* instance

`register_magics`(*base_name=None*)
The following magic functions will be registered (assuming that the base name is chosen to be 'hive'): - Cell Magics:

- *%%hive*: For querying the database.
- *%%hive.execute*: For executing a statement against the database.
- ***%%hive.stream*: For executing a statement against the database,** and streaming the results.
- *%%hive.template*: The defining a new template.
- *%%hive.render*: Render a provided query statement.

- **Line Magics:**
  - *%hive*: For querying the database using a named template.
  - ***%hive.execute*: For executing a named template statement against** the database.
  - ***%hive.stream*: For executing a named template against the database,** and streaming the results.

- *%hive.render*: Render a provided a named template.
- *%hive.desc*: Describe the table nominated.
- *%hive.head*: Return the first rows in a specified table.
- *%hive.props*: Show the properties specified for a nominated table.

Documentation for these magics is provided online.

reset()

Reset this *Duct* instance to its pre-preparation state.

This method disconnects from the service, resets any temporary authentication and restores the values of the attributes listed in *prepared_fields* to their values as of when *Duct.prepare* was called.

Returns A reference to this object.

Return type *Duct* instance

session_properties

The default session properties used in statement executions.

Type dict

classmethod statement_cleanup(*statement*)

Clean up statements prior to hash computation.

This classmethod takes an SQL statement and reformats it by consistently removing comments and replacing all whitespace. It is used by the *statement_hash* method to avoid functionally identical queries hitting different cache keys. If the statement's language is not to be SQL, this method should be overloaded appropriately.

Parameters **statement** (`str`) – The statement to be reformatted/cleaned-up.

Returns The new statement, consistently reformatted.

Return type str

classmethod statement_hash(*statement*, *cleanup=True*)

Retrieve the hash to use to identify query statements to the cache.

Parameters

- **statement** (`str`) – A string representation of the statement to be hashed.
- **cleanup** (`bool`) – Whether the statement should first be consistently reformatted using *statement_cleanup*.

Returns The hash used to identify a statement to the cache.

Return type str

stream(*statement*, *format=None*, *format_opts={}*, *batch=None*, ***kwargs*)

Execute a statement against this database and stream formatted results.

This method returns a generator over objects representing rows in the result set. If *batch* is not *None*, then the iterator will be over lists of length *batch* containing formatted rows.

Parameters

- **statement** (`str`) – The statement to be executed against the database.
- **format** (`str`) – A subclass of CursorFormatter, or one of: 'pandas', 'hive', 'csv', 'tuple' or 'dict'. Defaults to *self.DEFAULT_CURSOR_FORMATTER*.
- **format_opts** (`dict`) – A dictionary of format-specific options.

- **batch** (*int*) – If not *None*, the number of rows from the resulting cursor to be returned at once.
- ****kwargs** (*dict*) – Additional keyword arguments to pass onto *DatabaseClient.execute*.

Returns

An iterator over objects of the nominated format or, if batched, a list of such objects.

Return type iterator

stream_to_file(*statement*, *file*, *format='csv'*, *fs=None*, ***kwargs*)
Execute a statement against this database and stream results to a file.

This method is a wrapper around *DatabaseClient.stream* that enables the iterative writing of cursor results to a file. This is especially useful when there are a very large number of results, and loading them all into memory would require considerable resources. Note that 'csv' is the default format for this method (rather than *pandas*).

Parameters

- **statement** (*str*) – The statement to be executed against the database.
- **file** (*str, file-like-object*) – The filename where the data should be written, or an open file-like resource.
- **format** (*str*) – The format to be used ('csv' by default). Format options can be passed via ***kwargs*.
- **fs** (*None, FileSystemClient*) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- ****kwargs** – Additional keyword arguments to pass onto *DatabaseClient.stream*.

table_desc(*table*, *renew=True*, ***kwargs*)
Describe a table in the database.

Parameters

- **table** (*str*) – The table to describe.
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns A dataframe description of the table.

Return type pandas.DataFrame

table_drop(*table*, ***kwargs*)
Remove a table from the database.

Parameters

- **table** (*str*) – The table to drop.
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns The cursor associated with this execution.

Return type DB-API cursor

table_exists(*table*, *renew=True*, ***kwargs*)
Check whether a table exists.

Parameters

- **table** (*str*) – The table for which to check.

- **renew** (*bool*) – Whether to renew the table list or use cached results (default: True).
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns *True* if table exists, and *False* otherwise.

Return type bool

table_head(*table*, *n=10*, *renew=True*, ***kwargs*)
Retrieve the first *n* rows from a table.

Parameters

- **table** (*str*) – The table from which to extract data.
- **n** (*int*) – The number of rows to extract.
- **renew** (*bool*) – Whether to renew the table list or use cached results (default: True).
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns

A dataframe representation of the first *n* rows of the nominated table.

Return type pandas.DataFrame

table_list(*namespace=None*, *renew=True*, ***kwargs*)
Return a list of table names in the data source as a DataFrame.

Parameters

- **namespace** (*str*) – The namespace in which to look for tables.
- **renew** (*bool*) – Whether to renew the table list or use cached results (default: True).
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns The names of schemas in this database.

Return type list<str>

table_props(*table*, *renew=True*, ***kwargs*)
Retrieve the properties associated with a table.

Parameters

- **table** (*str*) – The table from which to extract data.
- **renew** (*bool*) – Whether to renew the table list or use cached results (default: True).
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns

A dataframe representation of the table properties.

Return type pandas.DataFrame

template_add(*name*, *body*)
Add a named template to the internal dictionary of templates.

Note: Templates are interpreted as *jinja2* templates. See *.template_render* for more information.

Parameters

- **name** (*str*) – The name of the template.
- **body** (*str*) – The (typically) multiline body of the template.

Returns A reference to this object.

Return type *PrestoClient*

template_get (*name*)

Retrieve a named template.

Parameters **name** (*str*) – The name of the template to retrieve.

Raises ValueError – If *name* is not associated with a template.

Returns The requested template.

Return type str

template_names

A list of names associated with the templates associated with this client.

Type list

template_render (*name_or_statement*, *context=None*, *by_name=False*, *cleanup=False*, *meta_only=False*)

Render a template by name or value.

In addition to the *jinja2* templating syntax, described in more detail in the official *jinja2* documentation, a meta-templating extension is also provided. This meta-templating allows you to reference other reference other templates. For example, if you had two SQL templates named 'template_a' and 'template_b', then you could render them into one SQL query using (for example):

```
.template_render('''
WITH
    a AS (
        {{{template_a}}}
    ),
    b AS (
        {{{template_b}}}
    )
SELECT *
FROM a
JOIN b ON a.x = b.x
''')
```

Note that template substitution in this way is iterative, so you can chain template embedding, provided that such embedding is not recursive.

Parameters

- **name_or_statement** (*str*) – The name of a template (if *by_name* is True) or else a string representation of a *jinja2* template.
- **context** (*dict, None*) – A dictionary to use as the template context. If not specified, an empty dictionary is used.
- **by_name** (*bool*) – *True* if *name_or_statement* should be interpreted as a template name, or *False* (default) if *name_or_statement* should be interpreted as a template body.
- **cleanup** (*bool*) – *True* if the rendered statement should be formatted, *False* (default) otherwise
- **meta_only** (*bool*) – *True* if rendering should only progress as far as rendering nested templates (i.e. don't actually substitute in variables from the context); *False* (default) otherwise.

Returns The rendered template.

Return type str

`template_variables`(*name_or_statement*, *by_name=False*)

Return the set of undeclared variables required for this template.

Parameters

- **`name_or_statement`** (`str`) – The name of a template (if *by_name* is True) or else a string representation of a *jinja2* template.
- **`by_name`** (`bool`) – *True* if *name_or_statement* should be interpreted as a template name, or *False* (default) if *name_or_statement* should be interpreted as a template body.

Returns A set of names which the template requires to be rendered.

Return type set<str>

`username`

Some services require authentication in order to connect to the service, in which case the appropriate username can be specified. If not specified at instantiation, your local login name will be used. If *True* was provided, you will be prompted to type your username at runtime as necessary. If *False* was provided, then *None* will be returned. You can specify a different username at runtime using: *duct.username = '<username>'*.

Type str

HiveServer2Client

class `omniduct.databases.hiveserver2.`**`HiveServer2Client`**(*session_properties=None*, *templates=None*, *template_context=None*, *default_format_opts=None*, ***kwargs*)

Bases: *`omniduct.databases.base.DatabaseClient`*, `omniduct.databases._schemas.SchemasMixin`

This Duct connects to an Apache HiveServer2 server instance using the *pyhive* or *impyla* libraries.

Attributes

- **schema** (*str, None*) – The default schema to use for queries (will default to server-default if not specified).
- **driver** (*str*) – One of 'pyhive' (default) or 'impyla', which specifies how the client communicates with Hive.
- **auth_mechanism** (*str*) – The authorisation protocol to use for connections. Defaults to 'NOSASL'. Authorisation methods differ between drivers. Please refer to *pyhive* and *impyla* documentation for more details.
- **push_using_hive_cli** (*bool*) – Whether the *.push()* operation should directly add files using *LOAD DATA LOCAL INPATH* rather than the *INSERT* operation via SQLAlchemy. Note that this requires the presence of the *hive* executable on the local PATH, or if connecting via a *RemoteClient* instance, on the remote's PATH. This is mostly useful for older versions of Hive which do not support the *INSERT* statement.
- **default_table_props** (*dict*) – A dictionary of table properties to use by default when creating tables.
- **connection_options** (*dict*) – Additional options to pass through to the *.connect()* methods of the drivers.

Attributes inherited from Duct:

protocol (str): The name of the protocol for which this instance was created (especially useful if a *Duct* subclass supports multiple protocols).

name (str): The name given to this *Duct* instance (defaults to class name).

host (str): The host name providing the service (will be '127.0.0.1', if service is port forwarded from remote; use *._host* to see remote host).

port (int): The port number of the service (will be the port-forwarded local port, if relevant; for remote port use *._port*).

username (str, bool): The username to use for the service. password (str, bool): The password to use for the service. registry (None, omniduct.registry.DuctRegistry): A reference to a

DuctRegistry instance for runtime lookup of other services.

remote (None, omniduct.remotes.base.RemoteClient): A reference to a *RemoteClient* instance to manage connections to remote services.

cache (None, omniduct.caches.base.Cache): A reference to a *Cache* instance to add support for caching, if applicable.

connection_fields (tuple<str>, list<str>): A list of instance attributes to monitor for changes, whereupon the *Duct* instance should automatically disconnect. By default, the following attributes are monitored: 'host', 'port', 'remote', 'username', and 'password'.

prepared_fields (tuple<str>, list<str>): A list of instance attributes to be populated (if their values are callable) when the instance first connects to a service. Refer to *Duct.prepare* and *Duct._prepare* for more details. By default, the following attributes are prepared: '_host', '_port', '_username', and '_password'.

Additional attributes including *host*, *port*, *username* and *password* are documented inline.

Class Attributes:

AUTO_LOGGING_SCOPE (bool): Whether this class should be used by omniduct logging code as a "scope". Should be overridden by subclasses as appropriate.

DUCT_TYPE (Duct.Type): The type of *Duct* service that is provided by this Duct instance. Should be overridden by subclasses as appropriate.

PROTOCOLS (list<str>): The name(s) of any protocols that should be associated with this class. Should be overridden by subclasses as appropriate.

`class Type`

Bases: `enum.Enum`

The *Duct.Type* enum specifies all of the permissible values of *Duct.DUCT_TYPE*. Also determines the order in which ducts are loaded by DuctRegistry.

`__init__`(*session_properties=None*, *templates=None*, *template_context=None*, *default_format_opts=None*, ***kwargs*)

protocol (str, None): Name of protocol (used by Duct registries to inform Duct instances of how they were instantiated).

name (str, None): The name to used by the *Duct* instance (defaults to class name if not specified).

registry (DuctRegistry, None): The registry to use to lookup remote and/or cache instance specified by name.

remote (str, RemoteClient): The remote by which the ducted service should be contacted.

host (str): The hostname of the service to be used by this client. port (int): The port of the service to be used by this client. username (str, bool, None): The username to authenticate with if necessary.

If True, then users will be prompted at runtime for credentials.

password (str, bool, None): The password to authenticate with if necessary. If True, then users will be prompted at runtime for credentials.

cache(Cache, None): The cache client to be attached to this instance. Cache will only used by specific methods as configured by the client.

cache_namespace(str, None): The namespace to use by default when writing to the cache.

DatabaseClient Quirks:

session_properties (dict): A mapping of default session properties to values. Interpretation is left up to implementations.

templates (dict): A dictionary of name to template mappings. Additional templates can be added using *.template_add*.

template_context (dict): The default template context to use when rendering templates.

default_format_opts (dict): The default formatting options passed to cursor formatter.

HiveServer2Client Quirks:

schema (str, None): The default database/schema to use for queries (will default to server-default if not specified).

driver (str): One of 'pyhive' (default) or 'impyla', which specifies how the client communicates with Hive.

auth_mechanism (str): The authorisation protocol to use for connections. Defaults to 'NOSASL'. Authorisation methods differ between drivers. Please refer to *pyhive* and *impyla* documentation for more details.

push_using_hive_cli (bool): Whether the *.push()* operation should directly add files using *LOAD DATA LOCAL INPATH* rather than the *INSERT* operation via SQLAlchemy. Note that this requires the presence of the *hive* executable on the local PATH, or if connecting via a *RemoteClient* instance, on the remote's PATH. This is mostly useful for older versions of Hive which do not support the *INSERT* statement. False by default.

default_table_props (dict): A dictionary of table properties to use by default when creating tables (default is an empty dict).

****connection_options (dict): Additional options to pass through to the** *.connect()* methods of the drivers.

`connect`()

Connect to the service backing this client.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Returns A reference to the current object.

Return type *Duct* instance

`dataframe_to_table`(*df*, *table*, *if_exists='fail'*, ***kwargs*)

Upload a local pandas dataframe into a table in this database.

Parameters

- **df** (*pandas.DataFrame*) – The dataframe to upload into the database.
- **table** (*str, ParsedNamespaces*) – The name of the table into which the dataframe should be uploaded.
- **if_exists** (*str*) – if nominated table already exists: 'fail' to do nothing, 'replace' to drop, recreate and insert data into new table, and 'append' to add data from this table into the existing table.
- ****kwargs** (*dict*) – Additional keyword arguments to pass onto *DatabaseClient._dataframe_to_table*.

HiveServer2Client Quirks: If *use_hive_cli* (or if not specified *.push_using_hive_cli*) is *True*, a *CREATE TABLE* statement will be automatically generated based on the datatypes of the DataFrame (unless overwritten by *dtype_overrides*). The *DataFrame* will then be exported to a CSV compatible with Hive and uploaded (if necessary) to the remote, before being loaded into Hive using a *LOAD DATA LOCAL INFILE …* query using the *hive* cli executable. Note that if a table is not partitioned, you cannot convert it to a parititioned table without deleting it first.

If *use_hive_cli* (or if not specified *.push_using_hive_cli*) is *False*, an attempt will be made to push the *DataFrame* to Hive using *pandas.DataFrame.to_sql* and the SQLAlchemy binding provided by *pyhive* and *impyla*. This may be slower, does not support older versions of Hive, and does not support table properties or partitioning.

If if the schema namespace is not specified, *table.schema* will be defaulted to your username.

Additional Args:

use_hive_cli (bool, None): A local override for the global *.push_using_hive_cli* attribute. If not specified, the global default is used. If True, then pushes are performed using the *hive* CLI executable on the local/remote PATH.

**kwargs (dict): Additional arguments to send to *pandas.DataFrame.to_sql*.

Further Parameters for CLI method (specifying these for the pandas method will cause a *RuntimeError* exception):

partition (dict): A mapping of column names to values that specify the partition into which the provided data should be uploaded, as well as providing the fields by which new tables should be partitioned.

sep (str): Field delimiter for data (defaults to CTRL-A, or *chr(1)*). table_props (dict): Properties to set on any newly created tables

(extends *.default_table_props*).

dtype_overrides (dict): Mapping of column names to Hive datatypes to use instead of default mapping.

disconnect()

Disconnect this client from backing service.

This method is automatically called during reconnections and/or at Python interpreter shutdown. It first calls *Duct._disconnect* (which should be implemented by subclasses) and then notifies the *RemoteClient* subclass, if present, to stop port-forwarding the remote service.

Returns A reference to this object.

Return type *Duct* instance

execute (*statement*, *wait=True*, *cursor=None*, *session_properties=None*, ***kwargs*)
Execute a statement against this database and return a cursor object.

Where supported by database implementations, this cursor can the be used in future executions, by passing it as the *cursor* keyword argument.

Parameters

- **statement** (*str*) – The statement to be executed by the query client (possibly templated).
- **wait** (*bool*) – Whether the cursor should be returned before the server-side query computation is complete and the relevant results downloaded.
- **cursor** (*DBAPI2 cursor*) – Rather than creating a new cursor, execute the statement against the provided cursor.
- **session_properties** (*dict*) – Additional session properties and/or overrides to use for this query. Setting a session property value to *None* will cause it to be omitted.
- ****kwargs** (*dict*) – Extra keyword arguments to be passed on to *_execute*, as implemented by subclasses.
- **template** (*bool*) – Whether the statement should be treated as a Jinja2 template. [Used by *render_statement* decorator.]
- **context** (*dict*) – The context in which the template should be evaluated (a dictionary of parameters to values). [Used by *render_statement* decorator.]
- **use_cache** (*bool*) – True or False (default). Whether to use the cache (if present). [Used by *cached_method* decorator.]
- **renew** (*bool*) – True or False (default). If cache is being used, renew it before returning stored value. [Used by *cached_method* decorator.]
- **cleanup** (*bool*) – Whether statement should be cleaned up before computing the hash used to cache results. [Used by *cached_method* decorator.]

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

HiveServer2Client Quirks:

Additional Args:

poll_interval (int): Default delay in seconds between consecutive query status (defaults to 1).

execute_from_file (*file*, *fs=None*, ***kwargs*)
Execute a statement stored in a file.

Parameters

- **file** (*str, file-like-object*) – The path of the file containing the query statement to be executed against the database, or an open file-like resource.
- **fs** (*None, FileSystemClient*) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- ****kwargs** (*dict*) – Extra keyword arguments to pass on to *DatabaseClient.execute*.

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

`execute_from_template`(*name*, *context=None*, ***kwargs*)
Render and then execute a named template.

Parameters

- **`name`** (*`str`*) – The name of the template to be rendered and executed.
- **`context`** (*`dict`*) – The context in which the template should be rendered.
- **`**kwargs`** (*`dict`*) – Additional parameters to pass to *.execute()*.

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

`classmethod for_protocol`(*protocol*)
Retrieve a *Duct* subclass for a given protocol.

Parameters **`protocol`** (*`str`*) – The protocol of interest.

Returns

The appropriate class for the provided, partially constructed with the *protocol* keyword argument set appropriately.

Return type functools.partial object

Raises `DuctProtocolUnknown` – If no class has been defined that offers the named protocol.

`host`
The host name providing the service, or '127.0.0.1' if *self.remote* is not *None*, whereupon the service will be port-forwarded locally. You can view the remote hostname using *duct._host*, and change the remote host at runtime using: *duct.host = '<host>'*.

Type str

`is_connected`()
Check whether this *Duct* instances is currently connected.

This method checks to see whether a *Duct* instance is currently connected. This is performed by verifying that the remote host and port are still accessible, and then by calling *Duct._is_connected*, which should be implemented by subclasses.

Returns Whether this *Duct* instance is currently connected.

Return type bool

`password`
Some services require authentication in order to connect to the service, in which case the appropriate password can be specified. If *True* was provided at instantiation, you will be prompted to type your password at runtime when necessary. If *False* was provided, then *None* will be returned. You can specify a different password at runtime using: *duct.password = '<password>'*.

Type str

`port`
The local port for the service. If *self.remote* is not *None*, the port will be port-forwarded from the remote host. To see the port used on the remote host refer to *duct._port*. You can change the remote port at runtime using: *duct.port = <port>*.

Type int

`prepare`()
Prepare a Duct subclass for use (if not already prepared).

This method is called before the value of any of the fields referenced in *self.connection_fields* are retrieved. The fields include, by default: 'host', 'port', 'remote', 'cache', 'username', and 'password'. Subclasses may add or subtract from these special fields.

When called, it first checks whether the instance has already been prepared, and if not calls *_prepare* and then records that the instance has been successfully prepared.

HiveServer2Client Quirks: This method may be overridden by subclasses, but provides the following default behaviour:

- Ensures *self.registry*, *self.remote* and *self.cache* values are instances of the right types.
- It replaces string values of *self.remote* and *self.cache* with remotes and caches looked up using *self.registry.lookup*.
- It looks through each of the fields nominated in *self.prepared_fields* and, if the corresponding value is callable, sets the value of that field to result of calling that value with a reference to *self*. By default, *prepared_fields* contains '_host', '_port', '_username', and '_password'.
- Ensures value of self.port is an integer (or None).

`query`(*statement*, *format=None*, *format_opts={}*, *use_cache=True*, ***kwargs*)
Execute a statement against this database and collect formatted data.

Parameters

- **`statement`** (`str`) – The statement to be executed by the query client (possibly templated).
- **`format`** (`str`) – A subclass of CursorFormatter, or one of: 'pandas', 'hive', 'csv', 'tuple' or 'dict'. Defaults to *self.DEFAULT_CURSOR_FORMATTER*.
- **`format_opts`** (`dict`) – A dictionary of format-specific options.
- **`use_cache`** (`bool`) – Whether to cache the cursor returned by *DatabaseClient.execute()* (overrides the default of False for *.execute()*). (default=True)
- **`**kwargs`** (`dict`) – Additional arguments to pass on to *DatabaseClient.execute()*.

Returns The results of the query formatted as nominated.

`query_from_file`(*file*, *fs=None*, ***kwargs*)
Query using a statement stored in a file.

Parameters

- **`file`** (`str, file-like-object`) – The path of the file containing the query statement to be executed against the database, or an open file-like resource.
- **`fs`** (`None, FileSystemClient`) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- **`**kwargs`** (`dict`) – Extra keyword arguments to pass on to *DatabaseClient.query*.

Returns The results of the query formatted as nominated.

Return type object

`query_from_template`(*name*, *context=None*, ***kwargs*)
Render and then query using a named tempalte.

Parameters

- **`name`** (`str`) – The name of the template to be rendered and used to query the database.
- **`context`** (`dict`) – The context in which the template should be rendered.

- **`**kwargs`** (*`dict`*) – Additional parameters to pass to *.query()*.

Returns The results of the query formatted as nominated.

Return type object

`query_to_table`(*statement*, *table*, *if_exists='fail'*, ***kwargs*)
Run a query and store the results in a table in this database.

Parameters

- **`statement`** – The statement to be executed.
- **`table`** (*`str`*) – The name of the table into which the dataframe should be uploaded.
- **`if_exists`** (*`str`*) – if nominated table already exists: 'fail' to do nothing, 'replace' to drop, recreate and insert data into new table, and 'append' to add data from this table into the existing table.
- **`**kwargs`** (*`dict`*) – Additional keyword arguments to pass onto *DatabaseClient._query_to_table*.

Returns The cursor object associated with the execution.

Return type DB-API cursor

`reconnect`()
Disconnects, and then reconnects, this client.

Note: This is equivalent to *duct.disconnect().connect()*.

Returns A reference to this object.

Return type *Duct* instance

`register_magics`(*base_name=None*)
The following magic functions will be registered (assuming that the base name is chosen to be 'hive'): - Cell Magics:

- *%%hive*: For querying the database.
- *%%hive.execute*: For executing a statement against the database.
- ***%%hive.stream*: For executing a statement against the database,** and streaming the results.
- *%%hive.template*: The defining a new template.
- *%%hive.render*: Render a provided query statement.

- **Line Magics:**
  - *%hive*: For querying the database using a named template.
  - ***%hive.execute*: For executing a named template statement against** the database.
  - ***%hive.stream*: For executing a named template against the database,** and streaming the results.
  - *%hive.render*: Render a provided a named template.
  - *%hive.desc*: Describe the table nominated.
  - *%hive.head*: Return the first rows in a specified table.
  - *%hive.props*: Show the properties specified for a nominated table.

Documentation for these magics is provided online.

reset()
Reset this *Duct* instance to its pre-preparation state.

This method disconnects from the service, resets any temporary authentication and restores the values of the attributes listed in *prepared_fields* to their values as of when *Duct.prepare* was called.

Returns A reference to this object.

Return type *Duct* instance

schemas
An object with attributes corresponding to the names of the schemas in this database.

Type object

session_properties
The default session properties used in statement executions.

Type dict

classmethod statement_cleanup(*statement*)
Clean up statements prior to hash computation.

This classmethod takes an SQL statement and reformats it by consistently removing comments and replacing all whitespace. It is used by the *statement_hash* method to avoid functionally identical queries hitting different cache keys. If the statement's language is not to be SQL, this method should be overloaded appropriately.

Parameters **statement** (*str*) – The statement to be reformatted/cleaned-up.

Returns The new statement, consistently reformatted.

Return type str

classmethod statement_hash(*statement*, *cleanup=True*)
Retrieve the hash to use to identify query statements to the cache.

Parameters

- **statement** (*str*) – A string representation of the statement to be hashed.
- **cleanup** (*bool*) – Whether the statement should first be consistently reformatted using *statement_cleanup*.

Returns The hash used to identify a statement to the cache.

Return type str

stream(*statement*, *format=None*, *format_opts={}*, *batch=None*, ***kwargs*)
Execute a statement against this database and stream formatted results.

This method returns a generator over objects representing rows in the result set. If *batch* is not *None*, then the iterator will be over lists of length *batch* containing formatted rows.

Parameters

- **statement** (*str*) – The statement to be executed against the database.
- **format** (*str*) – A subclass of CursorFormatter, or one of: 'pandas', 'hive', 'csv', 'tuple' or 'dict'. Defaults to *self.DEFAULT_CURSOR_FORMATTER*.
- **format_opts** (*dict*) – A dictionary of format-specific options.
- **batch** (*int*) – If not *None*, the number of rows from the resulting cursor to be returned at once.

- ****kwargs** (*dict*) – Additional keyword arguments to pass onto *DatabaseClient.execute*.

Returns

An iterator over objects of the nominated format or, if batched, a list of such objects.

Return type iterator

stream_to_file (*statement*, *file*, *format='csv'*, *fs=None*, ***kwargs*)
Execute a statement against this database and stream results to a file.

This method is a wrapper around *DatabaseClient.stream* that enables the iterative writing of cursor results to a file. This is especially useful when there are a very large number of results, and loading them all into memory would require considerable resources. Note that 'csv' is the default format for this method (rather than *pandas*).

Parameters

- **statement** (*str*) – The statement to be executed against the database.
- **file** (*str, file-like-object*) – The filename where the data should be written, or an open file-like resource.
- **format** (*str*) – The format to be used ('csv' by default). Format options can be passed via ***kwargs*.
- **fs** (*None*, FileSystemClient) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- ****kwargs** – Additional keyword arguments to pass onto *DatabaseClient.stream*.

table_desc (*table*, *renew=True*, ***kwargs*)
Describe a table in the database.

Parameters

- **table** (*str*) – The table to describe.
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns A dataframe description of the table.

Return type pandas.DataFrame

table_drop (*table*, ***kwargs*)
Remove a table from the database.

Parameters

- **table** (*str*) – The table to drop.
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns The cursor associated with this execution.

Return type DB-API cursor

table_exists (*table*, *renew=True*, ***kwargs*)
Check whether a table exists.

Parameters

- **table** (*str*) – The table for which to check.
- **renew** (*bool*) – Whether to renew the table list or use cached results (default: True).
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns *True* if table exists, and *False* otherwise.

Return type bool

`table_head`(*table*, *n=10*, *renew=True*, ***kwargs*)

Retrieve the first *n* rows from a table.

Parameters

- **`table`** (*`str`*) – The table from which to extract data.
- **`n`** (*`int`*) – The number of rows to extract.
- **`renew`** (*`bool`*) – Whether to renew the table list or use cached results (default: True).
- **`**kwargs`** (*`dict`*) – Additional arguments passed through to implementation.

Returns

A dataframe representation of the first *n* rows of the nominated table.

Return type pandas.DataFrame

`table_list`(*namespace=None*, *renew=True*, ***kwargs*)

Return a list of table names in the data source as a DataFrame.

Parameters

- **`namespace`** (*`str`*) – The namespace in which to look for tables.
- **`renew`** (*`bool`*) – Whether to renew the table list or use cached results (default: True).
- **`**kwargs`** (*`dict`*) – Additional arguments passed through to implementation.

Returns The names of schemas in this database.

Return type list<str>

`table_props`(*table*, *renew=True*, ***kwargs*)

Retrieve the properties associated with a table.

Parameters

- **`table`** (*`str`*) – The table from which to extract data.
- **`renew`** (*`bool`*) – Whether to renew the table list or use cached results (default: True).
- **`**kwargs`** (*`dict`*) – Additional arguments passed through to implementation.

Returns

A dataframe representation of the table properties.

Return type pandas.DataFrame

`template_add`(*name*, *body*)

Add a named template to the internal dictionary of templates.

Note: Templates are interpreted as *jinja2* templates. See *.template_render* for more information.

Parameters

- **`name`** (*`str`*) – The name of the template.
- **`body`** (*`str`*) – The (typically) multiline body of the template.

Returns A reference to this object.

Return type *PrestoClient*

template_get (*name*)

Retrieve a named template.

Parameters **name** (*str*) – The name of the template to retrieve.

Raises ValueError – If *name* is not associated with a template.

Returns The requested template.

Return type str

template_names

A list of names associated with the templates associated with this client.

Type list

template_render (*name_or_statement*, *context=None*, *by_name=False*, *cleanup=False*, *meta_only=False*)

Render a template by name or value.

In addition to the *jinja2* templating syntax, described in more detail in the official *jinja2* documentation, a meta-templating extension is also provided. This meta-templating allows you to reference other reference other templates. For example, if you had two SQL templates named 'template_a' and 'template_b', then you could render them into one SQL query using (for example):

```
.template_render('''
WITH
    a AS (
        {{{template_a}}}
    ),
    b AS (
        {{{template_b}}}
    )
SELECT *
FROM a
JOIN b ON a.x = b.x
''')
```

Note that template substitution in this way is iterative, so you can chain template embedding, provided that such embedding is not recursive.

Parameters

- **name_or_statement** (*str*) – The name of a template (if *by_name* is True) or else a string representation of a *jinja2* template.
- **context** (*dict, None*) – A dictionary to use as the template context. If not specified, an empty dictionary is used.
- **by_name** (*bool*) – *True* if *name_or_statement* should be interpreted as a template name, or *False* (default) if *name_or_statement* should be interpreted as a template body.
- **cleanup** (*bool*) – *True* if the rendered statement should be formatted, *False* (default) otherwise
- **meta_only** (*bool*) – *True* if rendering should only progress as far as rendering nested templates (i.e. don't actually substitute in variables from the context); *False* (default) otherwise.

Returns The rendered template.

Return type str

`template_variables` (*name_or_statement*, *by_name=False*)
Return the set of undeclared variables required for this template.

Parameters

- **`name_or_statement`** (`str`) – The name of a template (if *by_name* is True) or else a string representation of a *jinja2* template.
- **`by_name`** (`bool`) – *True* if *name_or_statement* should be interpreted as a template name, or *False* (default) if *name_or_statement* should be interpreted as a template body.

Returns A set of names which the template requires to be rendered.

Return type set<str>

`username`
Some services require authentication in order to connect to the service, in which case the appropriate username can be specified. If not specified at instantiation, your local login name will be used. If *True* was provided, you will be prompted to type your username at runtime as necessary. If *False* was provided, then *None* will be returned. You can specify a different username at runtime using: *duct.username = '<username>'*.

Type str

Neo4jClient

class `omniduct.databases.neo4j.`**`Neo4jClient`**(*session_properties=None*, *templates=None*, *template_context=None*, *default_format_opts=None*, ***kwargs*)

Bases: *`omniduct.databases.base.DatabaseClient`*

This Duct connects to a Neo4j graph database server using the *neo4j* python library.

Attributes inherited from Duct:

protocol (str): The name of the protocol for which this instance was created (especially useful if a *Duct* subclass supports multiple protocols).

name (str): The name given to this *Duct* instance (defaults to class name).

host (str): The host name providing the service (will be '127.0.0.1', if service is port forwarded from remote; use *._host* to see remote host).

port (int): The port number of the service (will be the port-forwarded local port, if relevant; for remote port use *._port*).

username (str, bool): The username to use for the service. password (str, bool): The password to use for the service. registry (None, omniduct.registry.DuctRegistry): A reference to a

DuctRegistry instance for runtime lookup of other services.

remote (None, omniduct.remotes.base.RemoteClient): A reference to a *RemoteClient* instance to manage connections to remote services.

cache (None, omniduct.caches.base.Cache): A reference to a *Cache* instance to add support for caching, if applicable.

connection_fields (tuple<str>, list<str>): A list of instance attributes to monitor for changes, whereupon the *Duct* instance should automatically disconnect. By default, the following attributes are monitored: 'host', 'port', 'remote', 'username', and 'password'.

prepared_fields (tuple<str>, list<str>): A list of instance attributes to be populated (if their values are callable) when the instance first connects to a service. Refer to *Duct.prepare* and *Duct._prepare* for more details. By default, the following attributes are prepared: '_host', '_port', '_username', and '_password'.

Additional attributes including *host*, *port*, *username* and *password* are documented inline.

Class Attributes:

AUTO_LOGGING_SCOPE (bool): Whether this class should be used by omniduct logging code as a "scope". Should be overridden by subclasses as appropriate.

DUCT_TYPE (Duct.Type): The type of *Duct* service that is provided by this Duct instance. Should be overridden by subclasses as appropriate.

PROTOCOLS (list<str>): The name(s) of any protocols that should be associated with this class. Should be overridden by subclasses as appropriate.

class Type

Bases: `enum.Enum`

The *Duct.Type* enum specifies all of the permissible values of *Duct.DUCT_TYPE*. Also determines the order in which ducts are loaded by DuctRegistry.

__init__(*session_properties=None*, *templates=None*, *template_context=None*, *default_format_opts=None*, ***kwargs*)

protocol (str, None): Name of protocol (used by Duct registries to inform Duct instances of how they were instantiated).

name (str, None): The name to used by the *Duct* instance (defaults to class name if not specified).

registry (DuctRegistry, None): The registry to use to lookup remote and/or cache instance specified by name.

remote (str, RemoteClient): The remote by which the ducted service should be contacted.

host (str): The hostname of the service to be used by this client. port (int): The port of the service to be used by this client. username (str, bool, None): The username to authenticate with if necessary.

If True, then users will be prompted at runtime for credentials.

password (str, bool, None): The password to authenticate with if necessary. If True, then users will be prompted at runtime for credentials.

cache(Cache, None): The cache client to be attached to this instance. Cache will only used by specific methods as configured by the client.

cache_namespace(str, None): The namespace to use by default when writing to the cache.

DatabaseClient Quirks:

session_properties (dict): A mapping of default session properties to values. Interpretation is left up to implementations.

templates (dict): A dictionary of name to template mappings. Additional templates can be added using *.template_add*.

template_context (dict): The default template context to use when rendering templates.

default_format_opts (dict): The default formatting options passed to cursor formatter.

connect()

Connect to the service backing this client.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Returns A reference to the current object.

Return type *Duct* instance

dataframe_to_table(*df*, *table*, *if_exists='fail'*, ***kwargs*)

Upload a local pandas dataframe into a table in this database.

Parameters

- **df** (`pandas.DataFrame`) – The dataframe to upload into the database.
- **table** (`str, ParsedNamespaces`) – The name of the table into which the dataframe should be uploaded.
- **if_exists** (`str`) – if nominated table already exists: 'fail' to do nothing, 'replace' to drop, recreate and insert data into new table, and 'append' to add data from this table into the existing table.
- ****kwargs** (`dict`) – Additional keyword arguments to pass onto *DatabaseClient._dataframe_to_table*.

disconnect()

Disconnect this client from backing service.

This method is automatically called during reconnections and/or at Python interpreter shutdown. It first calls *Duct._disconnect* (which should be implemented by subclasses) and then notifies the *RemoteClient* subclass, if present, to stop port-forwarding the remote service.

Returns A reference to this object.

Return type *Duct* instance

execute(*statement*, *wait=True*, *cursor=None*, *session_properties=None*, ***kwargs*)

Execute a statement against this database and return a cursor object.

Where supported by database implementations, this cursor can the be used in future executions, by passing it as the *cursor* keyword argument.

Parameters

- **statement** (`str`) – The statement to be executed by the query client (possibly templated).
- **wait** (`bool`) – Whether the cursor should be returned before the server-side query computation is complete and the relevant results downloaded.
- **cursor** (`DBAPI2 cursor`) – Rather than creating a new cursor, execute the statement against the provided cursor.
- **session_properties** (`dict`) – Additional session properties and/or overrides to use for this query. Setting a session property value to *None* will cause it to be omitted.
- ****kwargs** (`dict`) – Extra keyword arguments to be passed on to *_execute*, as implemented by subclasses.
- **template** (`bool`) – Whether the statement should be treated as a Jinja2 template. [Used by *render_statement* decorator.]

- **context** (*dict*) – The context in which the template should be evaluated (a dictionary of parameters to values). [Used by *render_statement* decorator.]
- **use_cache** (*bool*) – True or False (default). Whether to use the cache (if present). [Used by *cached_method* decorator.]
- **renew** (*bool*) – True or False (default). If cache is being used, renew it before returning stored value. [Used by *cached_method* decorator.]
- **cleanup** (*bool*) – Whether statement should be cleaned up before computing the hash used to cache results. [Used by *cached_method* decorator.]

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

execute_from_file (*file*, *fs=None*, ***kwargs*)
Execute a statement stored in a file.

Parameters

- **file** (*str, file-like-object*) – The path of the file containing the query statement to be executed against the database, or an open file-like resource.
- **fs** (*None, FileSystemClient*) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- ****kwargs** (*dict*) – Extra keyword arguments to pass on to *DatabaseClient.execute*.

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

execute_from_template (*name*, *context=None*, ***kwargs*)
Render and then execute a named template.

Parameters

- **name** (*str*) – The name of the template to be rendered and executed.
- **context** (*dict*) – The context in which the template should be rendered.
- ****kwargs** (*dict*) – Additional parameters to pass to *.execute()*.

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

classmethod for_protocol (*protocol*)
Retrieve a *Duct* subclass for a given protocol.

Parameters protocol (*str*) – The protocol of interest.

Returns

The appropriate class for the provided, partially constructed with the *protocol* keyword argument set appropriately.

Return type functools.partial object

Raises DuctProtocolUnknown – If no class has been defined that offers the named protocol.

host
The host name providing the service, or '127.0.0.1' if *self.remote* is not *None*, whereupon the service will be port-forwarded locally. You can view the remote hostname using *duct._host*, and change the remote host at runtime using: *duct.host = '<host>'*.

Type str

is_connected()

Check whether this *Duct* instances is currently connected.

This method checks to see whether a *Duct* instance is currently connected. This is performed by verifying that the remote host and port are still accessible, and then by calling *Duct._is_connected*, which should be implemented by subclasses.

Returns Whether this *Duct* instance is currently connected.

Return type bool

password

Some services require authentication in order to connect to the service, in which case the appropriate password can be specified. If *True* was provided at instantiation, you will be prompted to type your password at runtime when necessary. If *False* was provided, then *None* will be returned. You can specify a different password at runtime using: *duct.password = '<password>'*.

Type str

port

The local port for the service. If *self.remote* is not *None*, the port will be port-forwarded from the remote host. To see the port used on the remote host refer to *duct._port*. You can change the remote port at runtime using: *duct.port = <port>*.

Type int

prepare()

Prepare a Duct subclass for use (if not already prepared).

This method is called before the value of any of the fields referenced in *self.connection_fields* are retrieved. The fields include, by default: 'host', 'port', 'remote', 'cache', 'username', and 'password'. Subclasses may add or subtract from these special fields.

When called, it first checks whether the instance has already been prepared, and if not calls *_prepare* and then records that the instance has been successfully prepared.

Neo4jClient Quirks: This method may be overridden by subclasses, but provides the following default behaviour:

- Ensures *self.registry*, *self.remote* and *self.cache* values are instances of the right types.
- It replaces string values of *self.remote* and *self.cache* with remotes and caches looked up using *self.registry.lookup*.
- It looks through each of the fields nominated in *self.prepared_fields* and, if the corresponding value is callable, sets the value of that field to result of calling that value with a reference to *self*. By default, *prepared_fields* contains '_host', '_port', '_username', and '_password'.
- Ensures value of self.port is an integer (or None).

query(*statement*, *format=None*, *format_opts={}*, *use_cache=True*, ***kwargs*)

Execute a statement against this database and collect formatted data.

Parameters

- **statement** (*str*) – The statement to be executed by the query client (possibly templated).
- **format** (*str*) – A subclass of CursorFormatter, or one of: 'pandas', 'hive', 'csv', 'tuple' or 'dict'. Defaults to *self.DEFAULT_CURSOR_FORMATTER*.
- **format_opts** (*dict*) – A dictionary of format-specific options.

- **`use_cache`** (*`bool`*) – Whether to cache the cursor returned by *DatabaseClient.execute()* (overrides the default of False for *.execute()*). (default=True)
- **`**kwargs`** (*`dict`*) – Additional arguments to pass on to *DatabaseClient.execute()*.

Returns The results of the query formatted as nominated.

`query_from_file`(*file*, *fs=None*, ***kwargs*)
Query using a statement stored in a file.

Parameters

- **`file`** (*`str, file-like-object`*) – The path of the file containing the query statement to be executed against the database, or an open file-like resource.
- **`fs`** (*None*, `FileSystemClient`) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- **`**kwargs`** (*`dict`*) – Extra keyword arguments to pass on to *DatabaseClient.query*.

Returns The results of the query formatted as nominated.

Return type object

`query_from_template`(*name*, *context=None*, ***kwargs*)
Render and then query using a named tempalte.

Parameters

- **`name`** (*`str`*) – The name of the template to be rendered and used to query the database.
- **`context`** (*`dict`*) – The context in which the template should be rendered.
- **`**kwargs`** (*`dict`*) – Additional parameters to pass to *.query()*.

Returns The results of the query formatted as nominated.

Return type object

`query_to_table`(*statement*, *table*, *if_exists='fail'*, ***kwargs*)
Run a query and store the results in a table in this database.

Parameters

- **`statement`** – The statement to be executed.
- **`table`** (*`str`*) – The name of the table into which the dataframe should be uploaded.
- **`if_exists`** (*`str`*) – if nominated table already exists: 'fail' to do nothing, 'replace' to drop, recreate and insert data into new table, and 'append' to add data from this table into the existing table.
- **`**kwargs`** (*`dict`*) – Additional keyword arguments to pass onto *DatabaseClient._query_to_table*.

Returns The cursor object associated with the execution.

Return type DB-API cursor

`reconnect`()
Disconnects, and then reconnects, this client.

Note: This is equivalent to *duct.disconnect().connect()*.

Returns A reference to this object.

Return type *Duct* instance

register_magics(*base_name=None*)

The following magic functions will be registered (assuming that the base name is chosen to be 'hive'): - Cell Magics:

- *%%hive*: For querying the database.
- *%%hive.execute*: For executing a statement against the database.
- ***%%hive.stream*: For executing a statement against the database,** and streaming the results.
- *%%hive.template*: The defining a new template.
- *%%hive.render*: Render a provided query statement.

- **Line Magics:**
  - *%hive*: For querying the database using a named template.
  - ***%hive.execute*: For executing a named template statement against** the database.
  - ***%hive.stream*: For executing a named template against the database,** and streaming the results.
  - *%hive.render*: Render a provided a named template.
  - *%hive.desc*: Describe the table nominated.
  - *%hive.head*: Return the first rows in a specified table.
  - *%hive.props*: Show the properties specified for a nominated table.

Documentation for these magics is provided online.

reset()

Reset this *Duct* instance to its pre-preparation state.

This method disconnects from the service, resets any temporary authentication and restores the values of the attributes listed in *prepared_fields* to their values as of when *Duct.prepare* was called.

Returns A reference to this object.

Return type *Duct* instance

session_properties

The default session properties used in statement executions.

Type dict

classmethod statement_cleanup(*statement*)

Clean up statements prior to hash computation.

This classmethod takes an SQL statement and reformats it by consistently removing comments and replacing all whitespace. It is used by the *statement_hash* method to avoid functionally identical queries hitting different cache keys. If the statement's language is not to be SQL, this method should be overloaded appropriately.

Parameters **statement** (`str`) – The statement to be reformatted/cleaned-up.

Returns The new statement, consistently reformatted.

Return type str

classmethod statement_hash(*statement*, *cleanup=True*)

Retrieve the hash to use to identify query statements to the cache.

Parameters

- **`statement`** (*`str`*) – A string representation of the statement to be hashed.
- **`cleanup`** (*`bool`*) – Whether the statement should first be consistently reformatted using *statement_cleanup*.

Returns The hash used to identify a statement to the cache.

Return type str

`stream`(*statement*, *format=None*, *format_opts={}*, *batch=None*, ***kwargs*)
Execute a statement against this database and stream formatted results.

This method returns a generator over objects representing rows in the result set. If *batch* is not *None*, then the iterator will be over lists of length *batch* containing formatted rows.

Parameters

- **`statement`** (*`str`*) – The statement to be executed against the database.
- **`format`** (*`str`*) – A subclass of CursorFormatter, or one of: 'pandas', 'hive', 'csv', 'tuple' or 'dict'. Defaults to *self.DEFAULT_CURSOR_FORMATTER*.
- **`format_opts`** (*`dict`*) – A dictionary of format-specific options.
- **`batch`** (*`int`*) – If not *None*, the number of rows from the resulting cursor to be returned at once.
- **`**kwargs`** (*`dict`*) – Additional keyword arguments to pass onto *DatabaseClient.execute*.

Returns

An iterator over objects of the nominated format or, if batched, a list of such objects.

Return type iterator

`stream_to_file`(*statement*, *file*, *format='csv'*, *fs=None*, ***kwargs*)
Execute a statement against this database and stream results to a file.

This method is a wrapper around *DatabaseClient.stream* that enables the iterative writing of cursor results to a file. This is especially useful when there are a very large number of results, and loading them all into memory would require considerable resources. Note that 'csv' is the default format for this method (rather than *pandas*).

Parameters

- **`statement`** (*`str`*) – The statement to be executed against the database.
- **`file`** (*`str, file-like-object`*) – The filename where the data should be written, or an open file-like resource.
- **`format`** (*`str`*) – The format to be used ('csv' by default). Format options can be passed via ***kwargs*.
- **`fs`** (*`None, FileSystemClient`*) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- **`**kwargs`** – Additional keyword arguments to pass onto *DatabaseClient.stream*.

`table_desc`(*table*, *renew=True*, ***kwargs*)
Describe a table in the database.

Parameters

- **`table`** (*`str`*) – The table to describe.
- **`**kwargs`** (*`dict`*) – Additional arguments passed through to implementation.

Returns A dataframe description of the table.

Return type pandas.DataFrame

table_drop(*table*, ***kwargs*)
Remove a table from the database.

Parameters

- **table** (*str*) – The table to drop.
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns The cursor associated with this execution.

Return type DB-API cursor

table_exists(*table*, *renew=True*, ***kwargs*)
Check whether a table exists.

Parameters

- **table** (*str*) – The table for which to check.
- **renew** (*bool*) – Whether to renew the table list or use cached results (default: True).
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns *True* if table exists, and *False* otherwise.

Return type bool

table_head(*table*, *n=10*, *renew=True*, ***kwargs*)
Retrieve the first *n* rows from a table.

Parameters

- **table** (*str*) – The table from which to extract data.
- **n** (*int*) – The number of rows to extract.
- **renew** (*bool*) – Whether to renew the table list or use cached results (default: True).
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns

A dataframe representation of the first *n* rows of the nominated table.

Return type pandas.DataFrame

table_list(*namespace=None*, *renew=True*, ***kwargs*)
Return a list of table names in the data source as a DataFrame.

Parameters

- **namespace** (*str*) – The namespace in which to look for tables.
- **renew** (*bool*) – Whether to renew the table list or use cached results (default: True).
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns The names of schemas in this database.

Return type list<str>

table_props(*table*, *renew=True*, ***kwargs*)
Retrieve the properties associated with a table.

Parameters

- **table** (*str*) – The table from which to extract data.
- **renew** (*bool*) – Whether to renew the table list or use cached results (default: True).
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns

A dataframe representation of the table properties.

Return type pandas.DataFrame

template_add(*name*, *body*)

Add a named template to the internal dictionary of templates.

Note: Templates are interpreted as *jinja2* templates. See *.template_render* for more information.

Parameters

- **name** (*str*) – The name of the template.
- **body** (*str*) – The (typically) multiline body of the template.

Returns A reference to this object.

Return type *PrestoClient*

template_get(*name*)

Retrieve a named template.

Parameters **name** (*str*) – The name of the template to retrieve.

Raises `ValueError` – If *name* is not associated with a template.

Returns The requested template.

Return type str

template_names

A list of names associated with the templates associated with this client.

Type list

template_render(*name_or_statement*, *context=None*, *by_name=False*, *cleanup=False*, *meta_only=False*)

Render a template by name or value.

In addition to the *jinja2* templating syntax, described in more detail in the official *jinja2* documentation, a meta-templating extension is also provided. This meta-templating allows you to reference other reference other templates. For example, if you had two SQL templates named 'template_a' and 'template_b', then you could render them into one SQL query using (for example):

```
.template_render('''
WITH
    a AS (
        {{{template_a}}}
    ),
    b AS (
        {{{template_b}}}
    )
SELECT *
FROM a
JOIN b ON a.x = b.x
''')
```

Note that template substitution in this way is iterative, so you can chain template embedding, provided that such embedding is not recursive.

Parameters

- **name_or_statement** (*str*) – The name of a template (if *by_name* is True) or else a string representation of a *jinja2* template.
- **context** (*dict, None*) – A dictionary to use as the template context. If not specified, an empty dictionary is used.
- **by_name** (*bool*) – *True* if *name_or_statement* should be interpreted as a template name, or *False* (default) if *name_or_statement* should be interpreted as a template body.
- **cleanup** (*bool*) – *True* if the rendered statement should be formatted, *False* (default) otherwise
- **meta_only** (*bool*) – *True* if rendering should only progress as far as rendering nested templates (i.e. don't actually substitute in variables from the context); *False* (default) otherwise.

Returns The rendered template.

Return type str

template_variables(*name_or_statement*, *by_name=False*)

Return the set of undeclared variables required for this template.

Parameters

- **name_or_statement** (*str*) – The name of a template (if *by_name* is True) or else a string representation of a *jinja2* template.
- **by_name** (*bool*) – *True* if *name_or_statement* should be interpreted as a template name, or *False* (default) if *name_or_statement* should be interpreted as a template body.

Returns A set of names which the template requires to be rendered.

Return type set<str>

username

Some services require authentication in order to connect to the service, in which case the appropriate username can be specified. If not specified at instantiation, your local login name will be used. If *True* was provided, you will be prompted to type your username at runtime as necessary. If *False* was provided, then *None* will be returned. You can specify a different username at runtime using: *duct.username = '<username>'*.

Type str

PrestoClient

class omniduct.databases.presto.**PrestoClient**(*session_properties=None*, *templates=None*, *template_context=None*, *default_format_opts=None*, ***kwargs*)

Bases: *omniduct.databases.base.DatabaseClient*, omniduct.databases._schemas.SchemasMixin

This Duct connects to a Facebook Presto server instance using the *pyhive* library.

In addition to the standard *DatabaseClient* API, *PrestoClient* adds a *.schemas* descriptor attribute, which enables a tab completion driven exploration of a Presto database's schemas and tables.

Attributes

- **catalog** (*str*) – The default catalog to use in database queries.
- **schema** (*str*) – The default schema/database to use in database queries.
- **connection_options** (*dict*) – Additional options to pass on to *pyhive.presto.connect(. . .)*.

Attributes inherited from Duct:

protocol (str): The name of the protocol for which this instance was created (especially useful if a *Duct* subclass supports multiple protocols).

name (str): The name given to this *Duct* instance (defaults to class name).

host (str): The host name providing the service (will be '127.0.0.1', if service is port forwarded from remote; use *._host* to see remote host).

port (int): The port number of the service (will be the port-forwarded local port, if relevant; for remote port use *._port*).

username (str, bool): The username to use for the service. password (str, bool): The password to use for the service. registry (None, omniduct.registry.DuctRegistry): A reference to a

DuctRegistry instance for runtime lookup of other services.

remote (None, omniduct.remotes.base.RemoteClient): A reference to a *RemoteClient* instance to manage connections to remote services.

cache (None, omniduct.caches.base.Cache): A reference to a *Cache* instance to add support for caching, if applicable.

connection_fields (tuple<str>, list<str>): A list of instance attributes to monitor for changes, whereupon the *Duct* instance should automatically disconnect. By default, the following attributes are monitored: 'host', 'port', 'remote', 'username', and 'password'.

prepared_fields (tuple<str>, list<str>): A list of instance attributes to be populated (if their values are callable) when the instance first connects to a service. Refer to *Duct.prepare* and *Duct._prepare* for more details. By default, the following attributes are prepared: '_host', '_port', '_username', and '_password'.

Additional attributes including *host*, *port*, *username* and *password* are documented inline.

Class Attributes:

AUTO_LOGGING_SCOPE (bool): Whether this class should be used by omniduct logging code as a "scope". Should be overridden by subclasses as appropriate.

DUCT_TYPE (Duct.Type): The type of *Duct* service that is provided by this Duct instance. Should be overridden by subclasses as appropriate.

PROTOCOLS (list<str>): The name(s) of any protocols that should be associated with this class. Should be overridden by subclasses as appropriate.

`class Type`

Bases: `enum.Enum`

The *Duct.Type* enum specifies all of the permissible values of *Duct.DUCT_TYPE*. Also determines the order in which ducts are loaded by DuctRegistry.

`__init__`(*session_properties=None*, *templates=None*, *template_context=None*, *default_format_opts=None*, ***kwargs*)

protocol (str, None): Name of protocol (used by Duct registries to inform Duct instances of how they were instantiated).

name (str, None): The name to used by the *Duct* instance (defaults to class name if not specified).

registry (DuctRegistry, None): The registry to use to lookup remote and/or cache instance specified by name.

remote (str, RemoteClient): The remote by which the ducted service should be contacted.

host (str): The hostname of the service to be used by this client. port (int): The port of the service to be used by this client. username (str, bool, None): The username to authenticate with if necessary.

If True, then users will be prompted at runtime for credentials.

password (str, bool, None): The password to authenticate with if necessary. If True, then users will be prompted at runtime for credentials.

cache(Cache, None): The cache client to be attached to this instance. Cache will only used by specific methods as configured by the client.

cache_namespace(str, None): The namespace to use by default when writing to the cache.

DatabaseClient Quirks:

session_properties (dict): A mapping of default session properties to values. Interpretation is left up to implementations.

templates (dict): A dictionary of name to template mappings. Additional templates can be added using *.template_add*.

template_context (dict): The default template context to use when rendering templates.

default_format_opts (dict): The default formatting options passed to cursor formatter.

PrestoClient Quirks: catalog (str): The default catalog to use in database queries. schema (str): The default schema/database to use in database queries. server_protocol (str): The protocol over which to connect to the Presto REST

service ('http' or 'https'). (default='http')

source (str): The source of this query (by default "omniduct <version>"). If manually specified, result will be: "<source> / omniduct <version>".

connect()

Connect to the service backing this client.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Returns A reference to the current object.

Return type *Duct* instance

dataframe_to_table(*df*, *table*, *if_exists='fail'*, ***kwargs*)

Upload a local pandas dataframe into a table in this database.

Parameters

- **df** (*pandas.DataFrame*) – The dataframe to upload into the database.
- **table** (*str, ParsedNamespaces*) – The name of the table into which the dataframe should be uploaded.
- **if_exists** (*str*) – if nominated table already exists: 'fail' to do nothing, 'replace' to drop, recreate and insert data into new table, and 'append' to add data from this table into the existing table.

- ****kwargs** (*dict*) – Additional keyword arguments to pass onto *DatabaseClient._dataframe_to_table*.

PrestoClient Quirks: If if the schema namespace is not specified, *table.schema* will be defaulted to your username. Catalog overrides will be ignored, and will default to *self.catalog*.

disconnect()

Disconnect this client from backing service.

This method is automatically called during reconnections and/or at Python interpreter shutdown. It first calls *Duct._disconnect* (which should be implemented by subclasses) and then notifies the *RemoteClient* subclass, if present, to stop port-forwarding the remote service.

Returns A reference to this object.

Return type *Duct* instance

execute(*statement*, *wait=True*, *cursor=None*, *session_properties=None*, ***kwargs*)

Execute a statement against this database and return a cursor object.

Where supported by database implementations, this cursor can the be used in future executions, by passing it as the *cursor* keyword argument.

Parameters

- **statement** (*str*) – The statement to be executed by the query client (possibly templated).
- **wait** (*bool*) – Whether the cursor should be returned before the server-side query computation is complete and the relevant results downloaded.
- **cursor** (*DBAPI2 cursor*) – Rather than creating a new cursor, execute the statement against the provided cursor.
- **session_properties** (*dict*) – Additional session properties and/or overrides to use for this query. Setting a session property value to *None* will cause it to be omitted.
- ****kwargs** (*dict*) – Extra keyword arguments to be passed on to *_execute*, as implemented by subclasses.
- **template** (*bool*) – Whether the statement should be treated as a Jinja2 template. [Used by *render_statement* decorator.]
- **context** (*dict*) – The context in which the template should be evaluated (a dictionary of parameters to values). [Used by *render_statement* decorator.]
- **use_cache** (*bool*) – True or False (default). Whether to use the cache (if present). [Used by *cached_method* decorator.]
- **renew** (*bool*) – True or False (default). If cache is being used, renew it before returning stored value. [Used by *cached_method* decorator.]
- **cleanup** (*bool*) – Whether statement should be cleaned up before computing the hash used to cache results. [Used by *cached_method* decorator.]

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

PrestoClient Quirks: If something goes wrong, *PrestoClient* will attempt to parse the error log and present the user with useful debugging information. If that fails, the full traceback will be raised instead.

execute_from_file(*file*, *fs=None*, ***kwargs*)

Execute a statement stored in a file.

Parameters

- **file** (*str, file-like-object*) – The path of the file containing the query statement to be executed against the database, or an open file-like resource.
- **fs** (*None, FileSystemClient*) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- ****kwargs** (*dict*) – Extra keyword arguments to pass on to *DatabaseClient.execute*.

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

execute_from_template(*name*, *context=None*, ***kwargs*)

Render and then execute a named template.

Parameters

- **name** (*str*) – The name of the template to be rendered and executed.
- **context** (*dict*) – The context in which the template should be rendered.
- ****kwargs** (*dict*) – Additional parameters to pass to *.execute()*.

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

classmethod for_protocol(*protocol*)

Retrieve a *Duct* subclass for a given protocol.

Parameters protocol (*str*) – The protocol of interest.

Returns

The appropriate class for the provided, partially constructed with the *protocol* keyword argument set appropriately.

Return type functools.partial object

Raises `DuctProtocolUnknown` – If no class has been defined that offers the named protocol.

host

The host name providing the service, or '127.0.0.1' if *self.remote* is not *None*, whereupon the service will be port-forwarded locally. You can view the remote hostname using *duct._host*, and change the remote host at runtime using: *duct.host = '<host>'*.

Type str

is_connected()

Check whether this *Duct* instances is currently connected.

This method checks to see whether a *Duct* instance is currently connected. This is performed by verifying that the remote host and port are still accessible, and then by calling *Duct._is_connected*, which should be implemented by subclasses.

Returns Whether this *Duct* instance is currently connected.

Return type bool

password

Some services require authentication in order to connect to the service, in which case the appropriate password can be specified. If *True* was provided at instantiation, you will be prompted to type your password at runtime when necessary. If *False* was provided, then *None* will be returned. You can specify a different password at runtime using: *duct.password = '<password>'*.

Type str

port

The local port for the service. If *self.remote* is not *None*, the port will be port-forwarded from the remote host. To see the port used on the remote host refer to *duct._port*. You can change the remote port at runtime using: *duct.port = <port>*.

Type int

prepare()

Prepare a Duct subclass for use (if not already prepared).

This method is called before the value of any of the fields referenced in *self.connection_fields* are retrieved. The fields include, by default: 'host', 'port', 'remote', 'cache', 'username', and 'password'. Subclasses may add or subtract from these special fields.

When called, it first checks whether the instance has already been prepared, and if not calls *_prepare* and then records that the instance has been successfully prepared.

PrestoClient Quirks: This method may be overridden by subclasses, but provides the following default behaviour:

- Ensures *self.registry*, *self.remote* and *self.cache* values are instances of the right types.
- It replaces string values of *self.remote* and *self.cache* with remotes and caches looked up using *self.registry.lookup*.
- It looks through each of the fields nominated in *self.prepared_fields* and, if the corresponding value is callable, sets the value of that field to result of calling that value with a reference to *self*. By default, *prepared_fields* contains '_host', '_port', '_username', and '_password'.
- Ensures value of self.port is an integer (or None).

query(*statement*, *format=None*, *format_opts={}*, *use_cache=True*, ***kwargs*)

Execute a statement against this database and collect formatted data.

Parameters

- **statement** (*str*) – The statement to be executed by the query client (possibly templated).
- **format** (*str*) – A subclass of CursorFormatter, or one of: 'pandas', 'hive', 'csv', 'tuple' or 'dict'. Defaults to *self.DEFAULT_CURSOR_FORMATTER*.
- **format_opts** (*dict*) – A dictionary of format-specific options.
- **use_cache** (*bool*) – Whether to cache the cursor returned by *DatabaseClient.execute()* (overrides the default of False for *.execute()*). (default=True)
- ****kwargs** (*dict*) – Additional arguments to pass on to *DatabaseClient.execute()*.

Returns The results of the query formatted as nominated.

query_from_file(*file*, *fs=None*, ***kwargs*)

Query using a statement stored in a file.

Parameters

- **file** (*str, file-like-object*) – The path of the file containing the query statement to be executed against the database, or an open file-like resource.

- **fs** (*None*, `FileSystemClient`) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- ****kwargs** (*dict*) – Extra keyword arguments to pass on to *DatabaseClient.query*.

Returns The results of the query formatted as nominated.

Return type object

query_from_template(*name*, *context=None*, ***kwargs*)
Render and then query using a named tempalte.

Parameters

- **name** (*str*) – The name of the template to be rendered and used to query the database.
- **context** (*dict*) – The context in which the template should be rendered.
- ****kwargs** (*dict*) – Additional parameters to pass to *.query()*.

Returns The results of the query formatted as nominated.

Return type object

query_to_table(*statement*, *table*, *if_exists='fail'*, ***kwargs*)
Run a query and store the results in a table in this database.

Parameters

- **statement** – The statement to be executed.
- **table** (*str*) – The name of the table into which the dataframe should be uploaded.
- **if_exists** (*str*) – if nominated table already exists: 'fail' to do nothing, 'replace' to drop, recreate and insert data into new table, and 'append' to add data from this table into the existing table.
- ****kwargs** (*dict*) – Additional keyword arguments to pass onto *DatabaseClient._query_to_table*.

Returns The cursor object associated with the execution.

Return type DB-API cursor

reconnect()
Disconnects, and then reconnects, this client.

Note: This is equivalent to *duct.disconnect().connect()*.

Returns A reference to this object.

Return type *Duct* instance

register_magics(*base_name=None*)
The following magic functions will be registered (assuming that the base name is chosen to be 'hive'): - Cell Magics:

- *%%hive*: For querying the database.
- *%%hive.execute*: For executing a statement against the database.
- ***%%hive.stream*: For executing a statement against the database,** and streaming the results.
- *%%hive.template*: The defining a new template.
- *%%hive.render*: Render a provided query statement.

- **Line Magics:**

  - *%hive*: For querying the database using a named template.
  - ***%hive.execute*: For executing a named template statement against** the database.
  - ***%hive.stream*: For executing a named template against the database,** and streaming the results.
  - *%hive.render*: Render a provided a named template.
  - *%hive.desc*: Describe the table nominated.
  - *%hive.head*: Return the first rows in a specified table.
  - *%hive.props*: Show the properties specified for a nominated table.

Documentation for these magics is provided online.

reset()

Reset this *Duct* instance to its pre-preparation state.

This method disconnects from the service, resets any temporary authentication and restores the values of the attributes listed in *prepared_fields* to their values as of when *Duct.prepare* was called.

Returns A reference to this object.

Return type *Duct* instance

schemas

An object with attributes corresponding to the names of the schemas in this database.

Type object

session_properties

The default session properties used in statement executions.

Type dict

classmethod statement_cleanup(*statement*)

Clean up statements prior to hash computation.

This classmethod takes an SQL statement and reformats it by consistently removing comments and replacing all whitespace. It is used by the *statement_hash* method to avoid functionally identical queries hitting different cache keys. If the statement's language is not to be SQL, this method should be overloaded appropriately.

Parameters **statement** (`str`) – The statement to be reformatted/cleaned-up.

Returns The new statement, consistently reformatted.

Return type str

classmethod statement_hash(*statement*, *cleanup=True*)

Retrieve the hash to use to identify query statements to the cache.

Parameters

- **statement** (`str`) – A string representation of the statement to be hashed.
- **cleanup** (`bool`) – Whether the statement should first be consistently reformatted using *statement_cleanup*.

Returns The hash used to identify a statement to the cache.

Return type str

`stream`(*statement*, *format=None*, *format_opts={}*, *batch=None*, ***kwargs*)
Execute a statement against this database and stream formatted results.

This method returns a generator over objects representing rows in the result set. If *batch* is not *None*, then the iterator will be over lists of length *batch* containing formatted rows.

Parameters

- **`statement`** (`str`) – The statement to be executed against the database.
- **`format`** (`str`) – A subclass of CursorFormatter, or one of: 'pandas', 'hive', 'csv', 'tuple' or 'dict'. Defaults to *self.DEFAULT_CURSOR_FORMATTER*.
- **`format_opts`** (`dict`) – A dictionary of format-specific options.
- **`batch`** (`int`) – If not *None*, the number of rows from the resulting cursor to be returned at once.
- **`**kwargs`** (`dict`) – Additional keyword arguments to pass onto *DatabaseClient.execute*.

Returns

An iterator over objects of the nominated format or, if batched, a list of such objects.

Return type iterator

`stream_to_file`(*statement*, *file*, *format='csv'*, *fs=None*, ***kwargs*)
Execute a statement against this database and stream results to a file.

This method is a wrapper around *DatabaseClient.stream* that enables the iterative writing of cursor results to a file. This is especially useful when there are a very large number of results, and loading them all into memory would require considerable resources. Note that 'csv' is the default format for this method (rather than *pandas*).

Parameters

- **`statement`** (`str`) – The statement to be executed against the database.
- **`file`** (`str, file-like-object`) – The filename where the data should be written, or an open file-like resource.
- **`format`** (`str`) – The format to be used ('csv' by default). Format options can be passed via ***kwargs*.
- **`fs`** (`None, FileSystemClient`) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- **`**kwargs`** – Additional keyword arguments to pass onto *DatabaseClient.stream*.

`table_desc`(*table*, *renew=True*, ***kwargs*)
Describe a table in the database.

Parameters

- **`table`** (`str`) – The table to describe.
- **`**kwargs`** (`dict`) – Additional arguments passed through to implementation.

Returns A dataframe description of the table.

Return type pandas.DataFrame

`table_drop`(*table*, ***kwargs*)
Remove a table from the database.

Parameters

- **table** (*str*) – The table to drop.
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns The cursor associated with this execution.

Return type DB-API cursor

table_exists (*table*, *renew=True*, ***kwargs*)
Check whether a table exists.

Parameters

- **table** (*str*) – The table for which to check.
- **renew** (*bool*) – Whether to renew the table list or use cached results (default: True).
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns *True* if table exists, and *False* otherwise.

Return type bool

table_head (*table*, *n=10*, *renew=True*, ***kwargs*)
Retrieve the first *n* rows from a table.

Parameters

- **table** (*str*) – The table from which to extract data.
- **n** (*int*) – The number of rows to extract.
- **renew** (*bool*) – Whether to renew the table list or use cached results (default: True).
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns

A dataframe representation of the first *n* rows of the nominated table.

Return type pandas.DataFrame

table_list (*namespace=None*, *renew=True*, ***kwargs*)
Return a list of table names in the data source as a DataFrame.

Parameters

- **namespace** (*str*) – The namespace in which to look for tables.
- **renew** (*bool*) – Whether to renew the table list or use cached results (default: True).
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns The names of schemas in this database.

Return type list<str>

table_props (*table*, *renew=True*, ***kwargs*)
Retrieve the properties associated with a table.

Parameters

- **table** (*str*) – The table from which to extract data.
- **renew** (*bool*) – Whether to renew the table list or use cached results (default: True).
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns

A dataframe representation of the table properties.

Return type pandas.DataFrame

template_add(*name*, *body*)
Add a named template to the internal dictionary of templates.

Note: Templates are interpreted as *jinja2* templates. See *.template_render* for more information.

Parameters

- **name** (*str*) – The name of the template.
- **body** (*str*) – The (typically) multiline body of the template.

Returns A reference to this object.

Return type *PrestoClient*

template_get(*name*)
Retrieve a named template.

Parameters **name** (*str*) – The name of the template to retrieve.

Raises ValueError – If *name* is not associated with a template.

Returns The requested template.

Return type str

template_names
A list of names associated with the templates associated with this client.

Type list

template_render(*name_or_statement*, *context=None*, *by_name=False*, *cleanup=False*, *meta_only=False*)
Render a template by name or value.

In addition to the *jinja2* templating syntax, described in more detail in the official *jinja2* documentation, a meta-templating extension is also provided. This meta-templating allows you to reference other reference other templates. For example, if you had two SQL templates named 'template_a' and 'template_b', then you could render them into one SQL query using (for example):

```
.template_render('''
WITH
    a AS (
        {{{template_a}}}
    ),
    b AS (
        {{{template_b}}}
    )
SELECT *
FROM a
JOIN b ON a.x = b.x
''')
```

Note that template substitution in this way is iterative, so you can chain template embedding, provided that such embedding is not recursive.

Parameters

- **name_or_statement** (*str*) – The name of a template (if *by_name* is True) or else a string representation of a *jinja2* template.

- **`context`** (*`dict, None`*) – A dictionary to use as the template context. If not specified, an empty dictionary is used.
- **`by_name`** (*`bool`*) – *True* if *name_or_statement* should be interpreted as a template name, or *False* (default) if *name_or_statement* should be interpreted as a template body.
- **`cleanup`** (*`bool`*) – *True* if the rendered statement should be formatted, *False* (default) otherwise
- **`meta_only`** (*`bool`*) – *True* if rendering should only progress as far as rendering nested templates (i.e. don't actually substitute in variables from the context); *False* (default) otherwise.

Returns The rendered template.

Return type str

`template_variables` (*name_or_statement*, *by_name=False*)
Return the set of undeclared variables required for this template.

Parameters

- **`name_or_statement`** (*`str`*) – The name of a template (if *by_name* is True) or else a string representation of a *jinja2* template.
- **`by_name`** (*`bool`*) – *True* if *name_or_statement* should be interpreted as a template name, or *False* (default) if *name_or_statement* should be interpreted as a template body.

Returns A set of names which the template requires to be rendered.

Return type set<str>

`username`
Some services require authentication in order to connect to the service, in which case the appropriate username can be specified. If not specified at instantiation, your local login name will be used. If *True* was provided, you will be prompted to type your username at runtime as necessary. If *False* was provided, then *None* will be returned. You can specify a different username at runtime using: *duct.username = '<username>'*.

Type str

PySparkClient

class `omniduct.databases.pyspark.`**`PySparkClient`**(*session_properties=None*, *templates=None*, *template_context=None*, *default_format_opts=None*, ***kwargs*)
Bases: *`omniduct.databases.base.DatabaseClient`*

This Duct connects to a local PySpark session using the *pyspark* library.

Attributes inherited from Duct:

protocol (str): The name of the protocol for which this instance was created (especially useful if a *Duct* subclass supports multiple protocols).

name (str): The name given to this *Duct* instance (defaults to class name).

host (str): The host name providing the service (will be '127.0.0.1', if service is port forwarded from remote; use *._host* to see remote host).

port (int): The port number of the service (will be the port-forwarded local port, if relevant; for remote port use *._port*).

username (str, bool): The username to use for the service. password (str, bool): The password to use for the service. registry (None, omniduct.registry.DuctRegistry): A reference to a

DuctRegistry instance for runtime lookup of other services.

remote (None, omniduct.remotes.base.RemoteClient): A reference to a *RemoteClient* instance to manage connections to remote services.

cache (None, omniduct.caches.base.Cache): A reference to a ***Cache*** instance to add support for caching, if applicable.

connection_fields (tuple<str>, list<str>): A list of instance attributes to monitor for changes, whereupon the *Duct* instance should automatically disconnect. By default, the following attributes are monitored: 'host', 'port', 'remote', 'username', and 'password'.

prepared_fields (tuple<str>, list<str>): A list of instance attributes to be populated (if their values are callable) when the instance first connects to a service. Refer to *Duct.prepare* and *Duct._prepare* for more details. By default, the following attributes are prepared: '_host', '_port', '_username', and '_password'.

Additional attributes including *host*, *port*, *username* and *password* are documented inline.

Class Attributes:

AUTO_LOGGING_SCOPE (bool): Whether this class should be used by omniduct logging code as a "scope". Should be overridden by subclasses as appropriate.

DUCT_TYPE (Duct.Type): The type of ***Duct*** **service that is provided by** this Duct instance. Should be overridden by subclasses as appropriate.

PROTOCOLS (list<str>): The name(s) of any protocols that should be associated with this class. Should be overridden by subclasses as appropriate.

class Type

Bases: `enum.Enum`

The *Duct.Type* enum specifies all of the permissible values of *Duct.DUCT_TYPE*. Also determines the order in which ducts are loaded by DuctRegistry.

__init__(*session_properties=None*, *templates=None*, *template_context=None*, *default_format_opts=None*, ***kwargs*)

protocol (str, None): Name of protocol (used by Duct registries to inform Duct instances of how they were instantiated).

name (str, None): The name to used by the ***Duct*** **instance (defaults to** class name if not specified).

registry (DuctRegistry, None): The registry to use to lookup remote and/or cache instance specified by name.

remote (str, RemoteClient): The remote by which the ducted service should be contacted.

host (str): The hostname of the service to be used by this client. port (int): The port of the service to be used by this client. username (str, bool, None): The username to authenticate with if necessary.

If True, then users will be prompted at runtime for credentials.

password (str, bool, None): The password to authenticate with if necessary. If True, then users will be prompted at runtime for credentials.

cache(Cache, None): The cache client to be attached to this instance. Cache will only used by specific methods as configured by the client.

cache_namespace(str, None): The namespace to use by default when writing to the cache.

DatabaseClient Quirks:

session_properties (dict): A mapping of default session properties to values. Interpretation is left up to implementations.

templates (dict): A dictionary of name to template mappings. Additional templates can be added using *.template_add*.

template_context (dict): The default template context to use when rendering templates.

default_format_opts (dict): The default formatting options passed to cursor formatter.

PySparkClient Quirks:

Args: app_name (str): The application name of the SparkSession. config (dict or None): Any additional configuration to pass through

to the SparkSession builder.

master (str): The Spark master URL to connect to (only necessary if environment specified configuration is missing).

enable_hive_support (bool): Whether to enable Hive support for the Spark session.

Note: Pyspark must be installed in order to use this backend.

`connect``()`

Connect to the service backing this client.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Returns A reference to the current object.

Return type *Duct* instance

`dataframe_to_table`(*df*, *table*, *if_exists='fail'*, ***kwargs*)

Upload a local pandas dataframe into a table in this database.

Parameters

- **`df`** (*`pandas.DataFrame`*) – The dataframe to upload into the database.
- **`table`** (*`str, ParsedNamespaces`*) – The name of the table into which the dataframe should be uploaded.
- **`if_exists`** (*`str`*) – if nominated table already exists: 'fail' to do nothing, 'replace' to drop, recreate and insert data into new table, and 'append' to add data from this table into the existing table.
- **`**kwargs`** (*`dict`*) – Additional keyword arguments to pass onto *DatabaseClient._dataframe_to_table*.

`disconnect``()`

Disconnect this client from backing service.

This method is automatically called during reconnections and/or at Python interpreter shutdown. It first calls *Duct._disconnect* (which should be implemented by subclasses) and then notifies the *RemoteClient* subclass, if present, to stop port-forwarding the remote service.

Returns A reference to this object.

Return type *Duct* instance

execute(*statement*, *wait=True*, *cursor=None*, *session_properties=None*, ***kwargs*)
Execute a statement against this database and return a cursor object.

Where supported by database implementations, this cursor can the be used in future executions, by passing it as the *cursor* keyword argument.

Parameters

- **statement** (*str*) – The statement to be executed by the query client (possibly templated).
- **wait** (*bool*) – Whether the cursor should be returned before the server-side query computation is complete and the relevant results downloaded.
- **cursor** (*DBAPI2 cursor*) – Rather than creating a new cursor, execute the statement against the provided cursor.
- **session_properties** (*dict*) – Additional session properties and/or overrides to use for this query. Setting a session property value to *None* will cause it to be omitted.
- ****kwargs** (*dict*) – Extra keyword arguments to be passed on to *_execute*, as implemented by subclasses.
- **template** (*bool*) – Whether the statement should be treated as a Jinja2 template. [Used by *render_statement* decorator.]
- **context** (*dict*) – The context in which the template should be evaluated (a dictionary of parameters to values). [Used by *render_statement* decorator.]
- **use_cache** (*bool*) – True or False (default). Whether to use the cache (if present). [Used by *cached_method* decorator.]
- **renew** (*bool*) – True or False (default). If cache is being used, renew it before returning stored value. [Used by *cached_method* decorator.]
- **cleanup** (*bool*) – Whether statement should be cleaned up before computing the hash used to cache results. [Used by *cached_method* decorator.]

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

execute_from_file(*file*, *fs=None*, ***kwargs*)
Execute a statement stored in a file.

Parameters

- **file** (*str, file-like-object*) – The path of the file containing the query statement to be executed against the database, or an open file-like resource.
- **fs** (*None, FileSystemClient*) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- ****kwargs** (*dict*) – Extra keyword arguments to pass on to *DatabaseClient.execute*.

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

execute_from_template(*name*, *context=None*, ***kwargs*)
Render and then execute a named template.

Parameters

- **name** (*str*) – The name of the template to be rendered and executed.

- **context** (*dict*) – The context in which the template should be rendered.
- ****kwargs** (*dict*) – Additional parameters to pass to *.execute()*.

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

classmethod for_protocol(*protocol*)

Retrieve a *Duct* subclass for a given protocol.

Parameters protocol (*str*) – The protocol of interest.

Returns

The appropriate class for the provided, partially constructed with the *protocol* keyword argument set appropriately.

Return type functools.partial object

Raises `DuctProtocolUnknown` – If no class has been defined that offers the named protocol.

host

The host name providing the service, or '127.0.0.1' if *self.remote* is not *None*, whereupon the service will be port-forwarded locally. You can view the remote hostname using *duct._host*, and change the remote host at runtime using: *duct.host = '<host>'*.

Type str

is_connected()

Check whether this *Duct* instances is currently connected.

This method checks to see whether a *Duct* instance is currently connected. This is performed by verifying that the remote host and port are still accessible, and then by calling *Duct._is_connected*, which should be implemented by subclasses.

Returns Whether this *Duct* instance is currently connected.

Return type bool

password

Some services require authentication in order to connect to the service, in which case the appropriate password can be specified. If *True* was provided at instantiation, you will be prompted to type your password at runtime when necessary. If *False* was provided, then *None* will be returned. You can specify a different password at runtime using: *duct.password = '<password>'*.

Type str

port

The local port for the service. If *self.remote* is not *None*, the port will be port-forwarded from the remote host. To see the port used on the remote host refer to *duct._port*. You can change the remote port at runtime using: *duct.port = <port>*.

Type int

prepare()

Prepare a Duct subclass for use (if not already prepared).

This method is called before the value of any of the fields referenced in *self.connection_fields* are retrieved. The fields include, by default: 'host', 'port', 'remote', 'cache', 'username', and 'password'. Subclasses may add or subtract from these special fields.

When called, it first checks whether the instance has already been prepared, and if not calls *_prepare* and then records that the instance has been successfully prepared.

PySparkClient Quirks: This method may be overridden by subclasses, but provides the following default behaviour:

- Ensures *self.registry*, *self.remote* and *self.cache* values are instances of the right types.
- It replaces string values of *self.remote* and *self.cache* with remotes and caches looked up using *self.registry.lookup*.
- It looks through each of the fields nominated in *self.prepared_fields* and, if the corresponding value is callable, sets the value of that field to result of calling that value with a reference to *self*. By default, *prepared_fields* contains '_host', '_port', '_username', and '_password'.
- Ensures value of self.port is an integer (or None).

query (*statement*, *format=None*, *format_opts={}*, *use_cache=True*, ***kwargs*)
Execute a statement against this database and collect formatted data.

Parameters

- **statement** (*str*) – The statement to be executed by the query client (possibly templated).
- **format** (*str*) – A subclass of CursorFormatter, or one of: 'pandas', 'hive', 'csv', 'tuple' or 'dict'. Defaults to *self.DEFAULT_CURSOR_FORMATTER*.
- **format_opts** (*dict*) – A dictionary of format-specific options.
- **use_cache** (*bool*) – Whether to cache the cursor returned by *DatabaseClient.execute()* (overrides the default of False for *.execute()*). (default=True)
- ****kwargs** (*dict*) – Additional arguments to pass on to *DatabaseClient.execute()*.

Returns The results of the query formatted as nominated.

query_from_file (*file*, *fs=None*, ***kwargs*)
Query using a statement stored in a file.

Parameters

- **file** (*str, file-like-object*) – The path of the file containing the query statement to be executed against the database, or an open file-like resource.
- **fs** (*None, FileSystemClient*) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- ****kwargs** (*dict*) – Extra keyword arguments to pass on to *DatabaseClient.query*.

Returns The results of the query formatted as nominated.

Return type object

query_from_template (*name*, *context=None*, ***kwargs*)
Render and then query using a named tempalte.

Parameters

- **name** (*str*) – The name of the template to be rendered and used to query the database.
- **context** (*dict*) – The context in which the template should be rendered.
- ****kwargs** (*dict*) – Additional parameters to pass to *.query()*.

Returns The results of the query formatted as nominated.

Return type object

query_to_table(*statement*, *table*, *if_exists='fail'*, ***kwargs*)
Run a query and store the results in a table in this database.

Parameters

- **statement** – The statement to be executed.
- **table** (*str*) – The name of the table into which the dataframe should be uploaded.
- **if_exists** (*str*) – if nominated table already exists: 'fail' to do nothing, 'replace' to drop, recreate and insert data into new table, and 'append' to add data from this table into the existing table.
- ****kwargs** (*dict*) – Additional keyword arguments to pass onto *DatabaseClient._query_to_table*.

Returns The cursor object associated with the execution.

Return type DB-API cursor

reconnect()
Disconnects, and then reconnects, this client.

Note: This is equivalent to *duct.disconnect().connect()*.

Returns A reference to this object.

Return type *Duct* instance

register_magics(*base_name=None*)
The following magic functions will be registered (assuming that the base name is chosen to be 'hive'): - Cell Magics:

- *%%hive*: For querying the database.
- *%%hive.execute*: For executing a statement against the database.
- ***%%hive.stream*: For executing a statement against the database,** and streaming the results.
- *%%hive.template*: The defining a new template.
- *%%hive.render*: Render a provided query statement.

- **Line Magics:**
  - *%hive*: For querying the database using a named template.
  - ***%hive.execute*: For executing a named template statement against** the database.
  - ***%hive.stream*: For executing a named template against the database,** and streaming the results.
  - *%hive.render*: Render a provided a named template.
  - *%hive.desc*: Describe the table nominated.
  - *%hive.head*: Return the first rows in a specified table.
  - *%hive.props*: Show the properties specified for a nominated table.

Documentation for these magics is provided online.

reset()
Reset this *Duct* instance to its pre-preparation state.

This method disconnects from the service, resets any temporary authentication and restores the values of the attributes listed in *prepared_fields* to their values as of when *Duct.prepare* was called.

Returns A reference to this object.

Return type *Duct* instance

`session_properties`
The default session properties used in statement executions.

Type dict

`classmethod statement_cleanup`(*statement*)
Clean up statements prior to hash computation.

This classmethod takes an SQL statement and reformats it by consistently removing comments and replacing all whitespace. It is used by the *statement_hash* method to avoid functionally identical queries hitting different cache keys. If the statement's language is not to be SQL, this method should be overloaded appropriately.

Parameters **`statement`** (`str`) – The statement to be reformatted/cleaned-up.

Returns The new statement, consistently reformatted.

Return type str

`classmethod statement_hash`(*statement*, *cleanup=True*)
Retrieve the hash to use to identify query statements to the cache.

Parameters

- **`statement`** (`str`) – A string representation of the statement to be hashed.
- **`cleanup`** (`bool`) – Whether the statement should first be consistently reformatted using *statement_cleanup*.

Returns The hash used to identify a statement to the cache.

Return type str

`stream`(*statement*, *format=None*, *format_opts={}*, *batch=None*, ***kwargs*)
Execute a statement against this database and stream formatted results.

This method returns a generator over objects representing rows in the result set. If *batch* is not *None*, then the iterator will be over lists of length *batch* containing formatted rows.

Parameters

- **`statement`** (`str`) – The statement to be executed against the database.
- **`format`** (`str`) – A subclass of CursorFormatter, or one of: 'pandas', 'hive', 'csv', 'tuple' or 'dict'. Defaults to *self.DEFAULT_CURSOR_FORMATTER*.
- **`format_opts`** (`dict`) – A dictionary of format-specific options.
- **`batch`** (`int`) – If not *None*, the number of rows from the resulting cursor to be returned at once.
- **`**kwargs`** (`dict`) – Additional keyword arguments to pass onto *DatabaseClient.execute*.

Returns

An iterator over objects of the nominated format or, if batched, a list of such objects.

Return type iterator

`stream_to_file`(*statement*, *file*, *format='csv'*, *fs=None*, ***kwargs*)
Execute a statement against this database and stream results to a file.

This method is a wrapper around *DatabaseClient.stream* that enables the iterative writing of cursor results to a file. This is especially useful when there are a very large number of results, and loading them all into memory would require considerable resources. Note that 'csv' is the default format for this method (rather than *pandas*).

Parameters

- **`statement`** (*`str`*) – The statement to be executed against the database.
- **`file`** (*`str, file-like-object`*) – The filename where the data should be written, or an open file-like resource.
- **`format`** (*`str`*) – The format to be used ('csv' by default). Format options can be passed via ***kwargs*.
- **`fs`** (*`None,` `FileSystemClient`*) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- **`**kwargs`** – Additional keyword arguments to pass onto *DatabaseClient.stream*.

`table_desc`(*table*, *renew=True*, ***kwargs*)

Describe a table in the database.

Parameters

- **`table`** (*`str`*) – The table to describe.
- **`**kwargs`** (*`dict`*) – Additional arguments passed through to implementation.

Returns A dataframe description of the table.

Return type pandas.DataFrame

`table_drop`(*table*, ***kwargs*)

Remove a table from the database.

Parameters

- **`table`** (*`str`*) – The table to drop.
- **`**kwargs`** (*`dict`*) – Additional arguments passed through to implementation.

Returns The cursor associated with this execution.

Return type DB-API cursor

`table_exists`(*table*, *renew=True*, ***kwargs*)

Check whether a table exists.

Parameters

- **`table`** (*`str`*) – The table for which to check.
- **`renew`** (*`bool`*) – Whether to renew the table list or use cached results (default: True).
- **`**kwargs`** (*`dict`*) – Additional arguments passed through to implementation.

Returns *True* if table exists, and *False* otherwise.

Return type bool

`table_head`(*table*, *n=10*, *renew=True*, ***kwargs*)

Retrieve the first *n* rows from a table.

Parameters

- **`table`** (*`str`*) – The table from which to extract data.

- **n** (*int*) – The number of rows to extract.
- **renew** (*bool*) – Whether to renew the table list or use cached results (default: True).
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns

A dataframe representation of the first *n* rows of the nominated table.

Return type pandas.DataFrame

table_list (*namespace=None*, *renew=True*, ***kwargs*)
Return a list of table names in the data source as a DataFrame.

Parameters

- **namespace** (*str*) – The namespace in which to look for tables.
- **renew** (*bool*) – Whether to renew the table list or use cached results (default: True).
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns The names of schemas in this database.

Return type list<str>

table_props (*table*, *renew=True*, ***kwargs*)
Retrieve the properties associated with a table.

Parameters

- **table** (*str*) – The table from which to extract data.
- **renew** (*bool*) – Whether to renew the table list or use cached results (default: True).
- ****kwargs** (*dict*) – Additional arguments passed through to implementation.

Returns

A dataframe representation of the table properties.

Return type pandas.DataFrame

template_add (*name*, *body*)
Add a named template to the internal dictionary of templates.

Note: Templates are interpreted as *jinja2* templates. See *.template_render* for more information.

Parameters

- **name** (*str*) – The name of the template.
- **body** (*str*) – The (typically) multiline body of the template.

Returns A reference to this object.

Return type *PrestoClient*

template_get (*name*)
Retrieve a named template.

Parameters **name** (*str*) – The name of the template to retrieve.

Raises `ValueError` – If *name* is not associated with a template.

Returns The requested template.

Return type str

template_names
A list of names associated with the templates associated with this client.

Type list

template_render(*name_or_statement*, *context=None*, *by_name=False*, *cleanup=False*, *meta_only=False*)
Render a template by name or value.

In addition to the *jinja2* templating syntax, described in more detail in the official *jinja2* documentation, a meta-templating extension is also provided. This meta-templating allows you to reference other reference other templates. For example, if you had two SQL templates named 'template_a' and 'template_b', then you could render them into one SQL query using (for example):

```
.template_render('''
WITH
    a AS (
        {{{template_a}}}
    ),
    b AS (
        {{{template_b}}}
    )
SELECT *
FROM a
JOIN b ON a.x = b.x
''')
```

Note that template substitution in this way is iterative, so you can chain template embedding, provided that such embedding is not recursive.

Parameters

- **name_or_statement** (*str*) – The name of a template (if *by_name* is True) or else a string representation of a *jinja2* template.
- **context** (*dict, None*) – A dictionary to use as the template context. If not specified, an empty dictionary is used.
- **by_name** (*bool*) – *True* if *name_or_statement* should be interpreted as a template name, or *False* (default) if *name_or_statement* should be interpreted as a template body.
- **cleanup** (*bool*) – *True* if the rendered statement should be formatted, *False* (default) otherwise
- **meta_only** (*bool*) – *True* if rendering should only progress as far as rendering nested templates (i.e. don't actually substitute in variables from the context); *False* (default) otherwise.

Returns The rendered template.

Return type str

template_variables(*name_or_statement*, *by_name=False*)
Return the set of undeclared variables required for this template.

Parameters

- **name_or_statement** (*str*) – The name of a template (if *by_name* is True) or else a string representation of a *jinja2* template.
- **by_name** (*bool*) – *True* if *name_or_statement* should be interpreted as a template name, or *False* (default) if *name_or_statement* should be interpreted as a template body.

Returns A set of names which the template requires to be rendered.

Return type set<str>

`username`

Some services require authentication in order to connect to the service, in which case the appropriate username can be specified. If not specified at instantiation, your local login name will be used. If *True* was provided, you will be prompted to type your username at runtime as necessary. If *False* was provided, then *None* will be returned. You can specify a different username at runtime using: *duct.username = '<username>'*.

Type str

SQLAlchemyClient

class `omniduct.databases.sqlalchemy.`**`SQLAlchemyClient`**(*session_properties=None, templates=None, template_context=None, default_format_opts=None, **kwargs*)

Bases: `omniduct.databases.base.DatabaseClient`, `omniduct.databases._schemas.SchemasMixin`

This Duct connects to several different databases using one of several SQLAlchemy drivers. In general, these are provided for their potential utility, but will be less functional than the specially crafted database clients.

Attributes inherited from Duct:

protocol (str): The name of the protocol for which this instance was created (especially useful if a *Duct* subclass supports multiple protocols).

name (str): The name given to this *Duct* instance (defaults to class name).

host (str): The host name providing the service (will be '127.0.0.1', if service is port forwarded from remote; use *._host* to see remote host).

port (int): The port number of the service (will be the port-forwarded local port, if relevant; for remote port use *._port*).

username (str, bool): The username to use for the service. password (str, bool): The password to use for the service. registry (None, omniduct.registry.DuctRegistry): A reference to a

DuctRegistry instance for runtime lookup of other services.

remote (None, omniduct.remotes.base.RemoteClient): A reference to a *RemoteClient* instance to manage connections to remote services.

cache (None, omniduct.caches.base.Cache): A reference to a *Cache* instance to add support for caching, if applicable.

connection_fields (tuple<str>, list<str>): A list of instance attributes to monitor for changes, whereupon the *Duct* instance should automatically disconnect. By default, the following attributes are monitored: 'host', 'port', 'remote', 'username', and 'password'.

prepared_fields (tuple<str>, list<str>): A list of instance attributes to be populated (if their values are callable) when the instance first connects to a service. Refer to *Duct.prepare* and *Duct._prepare* for more details. By default, the following attributes are prepared: '_host', '_port', '_username', and '_password'.

Additional attributes including *host*, *port*, *username* and *password* are documented inline.

Class Attributes:

- **AUTO_LOGGING_SCOPE (bool): Whether this class should be used by omniduct** logging code as a "scope". Should be overridden by subclasses as appropriate.
- **DUCT_TYPE (Duct.Type): The type of *Duct* service that is provided by** this Duct instance. Should be overridden by subclasses as appropriate.
- **PROTOCOLS (list<str>): The name(s) of any protocols that should be** associated with this class. Should be overridden by subclasses as appropriate.

class Type

Bases: `enum.Enum`

The *Duct.Type* enum specifies all of the permissible values of *Duct.DUCT_TYPE*. Also determines the order in which ducts are loaded by DuctRegistry.

__init__(*session_properties=None*, *templates=None*, *template_context=None*, *default_format_opts=None*, ***kwargs*)

protocol (str, None): Name of protocol (used by Duct registries to inform Duct instances of how they were instantiated).

name (str, None): The name to used by the *Duct* instance (defaults to class name if not specified).

registry (DuctRegistry, None): The registry to use to lookup remote and/or cache instance specified by name.

remote (str, RemoteClient): The remote by which the ducted service should be contacted.

host (str): The hostname of the service to be used by this client. port (int): The port of the service to be used by this client. username (str, bool, None): The username to authenticate with if necessary.

If True, then users will be prompted at runtime for credentials.

password (str, bool, None): The password to authenticate with if necessary. If True, then users will be prompted at runtime for credentials.

cache(Cache, None): The cache client to be attached to this instance. Cache will only used by specific methods as configured by the client.

cache_namespace(str, None): The namespace to use by default when writing to the cache.

DatabaseClient Quirks:

- **session_properties (dict): A mapping of default session properties** to values. Interpretation is left up to implementations.
- **templates (dict): A dictionary of name to template mappings. Additional** templates can be added using *.template_add*.
- **template_context (dict): The default template context to use when** rendering templates.
- **default_format_opts (dict): The default formatting options passed to** cursor formatter.

connect()

Connect to the service backing this client.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Returns A reference to the current object.

Return type *Duct* instance

dataframe_to_table(*df*, *table*, *if_exists='fail'*, ***kwargs*)

Upload a local pandas dataframe into a table in this database.

Parameters

- **df** (*pandas.DataFrame*) – The dataframe to upload into the database.
- **table** (*str, ParsedNamespaces*) – The name of the table into which the dataframe should be uploaded.
- **if_exists** (*str*) – if nominated table already exists: 'fail' to do nothing, 'replace' to drop, recreate and insert data into new table, and 'append' to add data from this table into the existing table.
- ****kwargs** (*dict*) – Additional keyword arguments to pass onto *DatabaseClient._dataframe_to_table*.

disconnect()

Disconnect this client from backing service.

This method is automatically called during reconnections and/or at Python interpreter shutdown. It first calls *Duct._disconnect* (which should be implemented by subclasses) and then notifies the *RemoteClient* subclass, if present, to stop port-forwarding the remote service.

Returns A reference to this object.

Return type *Duct* instance

execute(*statement*, *wait=True*, *cursor=None*, *session_properties=None*, ***kwargs*)

Execute a statement against this database and return a cursor object.

Where supported by database implementations, this cursor can the be used in future executions, by passing it as the *cursor* keyword argument.

Parameters

- **statement** (*str*) – The statement to be executed by the query client (possibly templated).
- **wait** (*bool*) – Whether the cursor should be returned before the server-side query computation is complete and the relevant results downloaded.
- **cursor** (*DBAPI2 cursor*) – Rather than creating a new cursor, execute the statement against the provided cursor.
- **session_properties** (*dict*) – Additional session properties and/or overrides to use for this query. Setting a session property value to *None* will cause it to be omitted.
- ****kwargs** (*dict*) – Extra keyword arguments to be passed on to *_execute*, as implemented by subclasses.
- **template** (*bool*) – Whether the statement should be treated as a Jinja2 template. [Used by *render_statement* decorator.]
- **context** (*dict*) – The context in which the template should be evaluated (a dictionary of parameters to values). [Used by *render_statement* decorator.]
- **use_cache** (*bool*) – True or False (default). Whether to use the cache (if present). [Used by *cached_method* decorator.]
- **renew** (*bool*) – True or False (default). If cache is being used, renew it before returning stored value. [Used by *cached_method* decorator.]
- **cleanup** (*bool*) – Whether statement should be cleaned up before computing the hash used to cache results. [Used by *cached_method* decorator.]

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

`execute_from_file`(*file*, *fs=None*, ***kwargs*)
Execute a statement stored in a file.

Parameters

- **file** (*str, file-like-object*) – The path of the file containing the query statement to be executed against the database, or an open file-like resource.
- **fs** (*None,* FileSystemClient) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- ****kwargs** (*dict*) – Extra keyword arguments to pass on to *DatabaseClient.execute*.

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

`execute_from_template`(*name*, *context=None*, ***kwargs*)
Render and then execute a named template.

Parameters

- **name** (*str*) – The name of the template to be rendered and executed.
- **context** (*dict*) – The context in which the template should be rendered.
- ****kwargs** (*dict*) – Additional parameters to pass to *.execute()*.

Returns A DBAPI2 compatible cursor instance.

Return type DBAPI2 cursor

`classmethod for_protocol`(*protocol*)
Retrieve a *Duct* subclass for a given protocol.

Parameters **protocol** (*str*) – The protocol of interest.

Returns

The appropriate class for the provided, partially constructed with the *protocol* keyword argument set appropriately.

Return type functools.partial object

Raises `DuctProtocolUnknown` – If no class has been defined that offers the named protocol.

`host`
The host name providing the service, or '127.0.0.1' if *self.remote* is not *None*, whereupon the service will be port-forwarded locally. You can view the remote hostname using *duct._host*, and change the remote host at runtime using: *duct.host = '<host>'*.

Type str

`is_connected`()
Check whether this *Duct* instances is currently connected.

This method checks to see whether a *Duct* instance is currently connected. This is performed by verifying that the remote host and port are still accessible, and then by calling *Duct._is_connected*, which should be implemented by subclasses.

Returns Whether this *Duct* instance is currently connected.

Return type bool

password
Some services require authentication in order to connect to the service, in which case the appropriate password can be specified. If *True* was provided at instantiation, you will be prompted to type your password at runtime when necessary. If *False* was provided, then *None* will be returned. You can specify a different password at runtime using: *duct.password = '<password>'*.

Type str

port
The local port for the service. If *self.remote* is not *None*, the port will be port-forwarded from the remote host. To see the port used on the remote host refer to *duct._port*. You can change the remote port at runtime using: *duct.port = <port>*.

Type int

prepare()
Prepare a Duct subclass for use (if not already prepared).

This method is called before the value of any of the fields referenced in *self.connection_fields* are retrieved. The fields include, by default: 'host', 'port', 'remote', 'cache', 'username', and 'password'. Subclasses may add or subtract from these special fields.

When called, it first checks whether the instance has already been prepared, and if not calls *_prepare* and then records that the instance has been successfully prepared.

SQLAlchemyClient Quirks: This method may be overridden by subclasses, but provides the following default behaviour:

- Ensures *self.registry*, *self.remote* and *self.cache* values are instances of the right types.
- It replaces string values of *self.remote* and *self.cache* with remotes and caches looked up using *self.registry.lookup*.
- It looks through each of the fields nominated in *self.prepared_fields* and, if the corresponding value is callable, sets the value of that field to result of calling that value with a reference to *self*. By default, *prepared_fields* contains '_host', '_port', '_username', and '_password'.
- Ensures value of self.port is an integer (or None).

query(*statement*, *format=None*, *format_opts={}*, *use_cache=True*, ***kwargs*)
Execute a statement against this database and collect formatted data.

Parameters

- **statement** (`str`) – The statement to be executed by the query client (possibly templated).
- **format** (`str`) – A subclass of CursorFormatter, or one of: 'pandas', 'hive', 'csv', 'tuple' or 'dict'. Defaults to *self.DEFAULT_CURSOR_FORMATTER*.
- **format_opts** (`dict`) – A dictionary of format-specific options.
- **use_cache** (`bool`) – Whether to cache the cursor returned by *DatabaseClient.execute()* (overrides the default of False for *.execute()*). (default=True)
- ****kwargs** (`dict`) – Additional arguments to pass on to *DatabaseClient.execute()*.

Returns The results of the query formatted as nominated.

query_from_file(*file*, *fs=None*, ***kwargs*)
Query using a statement stored in a file.

Parameters

- **file** (*str, file-like-object*) – The path of the file containing the query statement to be executed against the database, or an open file-like resource.
- **fs** (*None, FileSystemClient*) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- ****kwargs** (*dict*) – Extra keyword arguments to pass on to *DatabaseClient.query*.

Returns The results of the query formatted as nominated.

Return type object

query_from_template (*name*, *context=None*, ***kwargs*)

Render and then query using a named tempalte.

Parameters

- **name** (*str*) – The name of the template to be rendered and used to query the database.
- **context** (*dict*) – The context in which the template should be rendered.
- ****kwargs** (*dict*) – Additional parameters to pass to *.query()*.

Returns The results of the query formatted as nominated.

Return type object

query_to_table (*statement*, *table*, *if_exists='fail'*, ***kwargs*)

Run a query and store the results in a table in this database.

Parameters

- **statement** – The statement to be executed.
- **table** (*str*) – The name of the table into which the dataframe should be uploaded.
- **if_exists** (*str*) – if nominated table already exists: 'fail' to do nothing, 'replace' to drop, recreate and insert data into new table, and 'append' to add data from this table into the existing table.
- ****kwargs** (*dict*) – Additional keyword arguments to pass onto *DatabaseClient._query_to_table*.

Returns The cursor object associated with the execution.

Return type DB-API cursor

reconnect ()

Disconnects, and then reconnects, this client.

Note: This is equivalent to *duct.disconnect().connect()*.

Returns A reference to this object.

Return type *Duct* instance

register_magics (*base_name=None*)

The following magic functions will be registered (assuming that the base name is chosen to be 'hive'): - Cell Magics:

- *%%hive*: For querying the database.
- *%%hive.execute*: For executing a statement against the database.
- ***%%hive.stream*: For executing a statement against the database,** and streaming the results.
- *%%hive.template*: The defining a new template.

- *%%hive.render*: Render a provided query statement.

- **Line Magics:**
  - *%hive*: For querying the database using a named template.
  - ***%hive.execute*: For executing a named template statement against** the database.
  - ***%hive.stream*: For executing a named template against the database,** and streaming the results.
  - *%hive.render*: Render a provided a named template.
  - *%hive.desc*: Describe the table nominated.
  - *%hive.head*: Return the first rows in a specified table.
  - *%hive.props*: Show the properties specified for a nominated table.

Documentation for these magics is provided online.

`reset`()

Reset this *Duct* instance to its pre-preparation state.

This method disconnects from the service, resets any temporary authentication and restores the values of the attributes listed in *prepared_fields* to their values as of when *Duct.prepare* was called.

Returns A reference to this object.

Return type *Duct* instance

`schemas`

An object with attributes corresponding to the names of the schemas in this database.

Type object

`session_properties`

The default session properties used in statement executions.

Type dict

`classmethod statement_cleanup`(*statement*)

Clean up statements prior to hash computation.

This classmethod takes an SQL statement and reformats it by consistently removing comments and replacing all whitespace. It is used by the *statement_hash* method to avoid functionally identical queries hitting different cache keys. If the statement's language is not to be SQL, this method should be overloaded appropriately.

Parameters **`statement`** (`str`) – The statement to be reformatted/cleaned-up.

Returns The new statement, consistently reformatted.

Return type str

`classmethod statement_hash`(*statement*, *cleanup=True*)

Retrieve the hash to use to identify query statements to the cache.

Parameters

- **`statement`** (`str`) – A string representation of the statement to be hashed.
- **`cleanup`** (`bool`) – Whether the statement should first be consistently reformatted using *statement_cleanup*.

Returns The hash used to identify a statement to the cache.

Return type str

`stream`(*statement*, *format=None*, *format_opts={}*, *batch=None*, ***kwargs*)
Execute a statement against this database and stream formatted results.

This method returns a generator over objects representing rows in the result set. If *batch* is not *None*, then the iterator will be over lists of length *batch* containing formatted rows.

Parameters

- **`statement`** (`str`) – The statement to be executed against the database.
- **`format`** (`str`) – A subclass of CursorFormatter, or one of: 'pandas', 'hive', 'csv', 'tuple' or 'dict'. Defaults to *self.DEFAULT_CURSOR_FORMATTER*.
- **`format_opts`** (`dict`) – A dictionary of format-specific options.
- **`batch`** (`int`) – If not *None*, the number of rows from the resulting cursor to be returned at once.
- **`**kwargs`** (`dict`) – Additional keyword arguments to pass onto *DatabaseClient.execute*.

Returns

An iterator over objects of the nominated format or, if batched, a list of such objects.

Return type iterator

`stream_to_file`(*statement*, *file*, *format='csv'*, *fs=None*, ***kwargs*)
Execute a statement against this database and stream results to a file.

This method is a wrapper around *DatabaseClient.stream* that enables the iterative writing of cursor results to a file. This is especially useful when there are a very large number of results, and loading them all into memory would require considerable resources. Note that 'csv' is the default format for this method (rather than *pandas*).

Parameters

- **`statement`** (`str`) – The statement to be executed against the database.
- **`file`** (`str, file-like-object`) – The filename where the data should be written, or an open file-like resource.
- **`format`** (`str`) – The format to be used ('csv' by default). Format options can be passed via ***kwargs*.
- **`fs`** (`None, FileSystemClient`) – The filesystem wihin which the nominated file should be found. If *None*, the local filesystem will be used.
- **`**kwargs`** – Additional keyword arguments to pass onto *DatabaseClient.stream*.

`table_desc`(*table*, *renew=True*, ***kwargs*)
Describe a table in the database.

Parameters

- **`table`** (`str`) – The table to describe.
- **`**kwargs`** (`dict`) – Additional arguments passed through to implementation.

Returns A dataframe description of the table.

Return type pandas.DataFrame

`table_drop`(*table*, ***kwargs*)
Remove a table from the database.

Parameters

- **`table`** (*`str`*) – The table to drop.
- **`**kwargs`** (*`dict`*) – Additional arguments passed through to implementation.

Returns The cursor associated with this execution.

Return type DB-API cursor

`table_exists`(*table*, *renew=True*, ***kwargs*)
Check whether a table exists.

Parameters

- **`table`** (*`str`*) – The table for which to check.
- **`renew`** (*`bool`*) – Whether to renew the table list or use cached results (default: True).
- **`**kwargs`** (*`dict`*) – Additional arguments passed through to implementation.

Returns *True* if table exists, and *False* otherwise.

Return type bool

`table_head`(*table*, *n=10*, *renew=True*, ***kwargs*)
Retrieve the first *n* rows from a table.

Parameters

- **`table`** (*`str`*) – The table from which to extract data.
- **`n`** (*`int`*) – The number of rows to extract.
- **`renew`** (*`bool`*) – Whether to renew the table list or use cached results (default: True).
- **`**kwargs`** (*`dict`*) – Additional arguments passed through to implementation.

Returns

A dataframe representation of the first *n* rows of the nominated table.

Return type pandas.DataFrame

`table_list`(*namespace=None*, *renew=True*, ***kwargs*)
Return a list of table names in the data source as a DataFrame.

Parameters

- **`namespace`** (*`str`*) – The namespace in which to look for tables.
- **`renew`** (*`bool`*) – Whether to renew the table list or use cached results (default: True).
- **`**kwargs`** (*`dict`*) – Additional arguments passed through to implementation.

Returns The names of schemas in this database.

Return type list<str>

`table_props`(*table*, *renew=True*, ***kwargs*)
Retrieve the properties associated with a table.

Parameters

- **`table`** (*`str`*) – The table from which to extract data.
- **`renew`** (*`bool`*) – Whether to renew the table list or use cached results (default: True).
- **`**kwargs`** (*`dict`*) – Additional arguments passed through to implementation.

Returns

A dataframe representation of the table properties.

Return type pandas.DataFrame

template_add(*name*, *body*)

Add a named template to the internal dictionary of templates.

Note: Templates are interpreted as *jinja2* templates. See *.template_render* for more information.

Parameters

- **name** (*str*) – The name of the template.
- **body** (*str*) – The (typically) multiline body of the template.

Returns A reference to this object.

Return type *PrestoClient*

template_get(*name*)

Retrieve a named template.

Parameters **name** (*str*) – The name of the template to retrieve.

Raises ValueError – If *name* is not associated with a template.

Returns The requested template.

Return type str

template_names

A list of names associated with the templates associated with this client.

Type list

template_render(*name_or_statement*, *context=None*, *by_name=False*, *cleanup=False*, *meta_only=False*)

Render a template by name or value.

In addition to the *jinja2* templating syntax, described in more detail in the official *jinja2* documentation, a meta-templating extension is also provided. This meta-templating allows you to reference other reference other templates. For example, if you had two SQL templates named 'template_a' and 'template_b', then you could render them into one SQL query using (for example):

```
.template_render('''
WITH
    a AS (
        {{{template_a}}}
    ),
    b AS (
        {{{template_b}}}
    )
SELECT *
FROM a
JOIN b ON a.x = b.x
''')
```

Note that template substitution in this way is iterative, so you can chain template embedding, provided that such embedding is not recursive.

Parameters

- **name_or_statement** (*str*) – The name of a template (if *by_name* is True) or else a string representation of a *jinja2* template.
- **context** (*dict, None*) – A dictionary to use as the template context. If not specified, an empty dictionary is used.
- **by_name** (*bool*) – *True* if *name_or_statement* should be interpreted as a template name, or *False* (default) if *name_or_statement* should be interpreted as a template body.
- **cleanup** (*bool*) – *True* if the rendered statement should be formatted, *False* (default) otherwise
- **meta_only** (*bool*) – *True* if rendering should only progress as far as rendering nested templates (i.e. don't actually substitute in variables from the context); *False* (default) otherwise.

Returns The rendered template.

Return type str

template_variables(*name_or_statement*, *by_name=False*)
Return the set of undeclared variables required for this template.

Parameters

- **name_or_statement** (*str*) – The name of a template (if *by_name* is True) or else a string representation of a *jinja2* template.
- **by_name** (*bool*) – *True* if *name_or_statement* should be interpreted as a template name, or *False* (default) if *name_or_statement* should be interpreted as a template body.

Returns A set of names which the template requires to be rendered.

Return type set<str>

username
Some services require authentication in order to connect to the service, in which case the appropriate username can be specified. If not specified at instantiation, your local login name will be used. If *True* was provided, you will be prompted to type your username at runtime as necessary. If *False* was provided, then *None* will be returned. You can specify a different username at runtime using: *duct.username = '<username>'*.

Type str

5.3 Filesystems

All database clients are expected to be subclasses of *DatabaseClient*, and so will share a common API and inherit a suite of IPython magics. Protocol implementations are also free to add extra methods, which are documented in the "Subclass Reference" section below.

5.3.1 Common API

class omniduct.filesystems.base.**FileSystemClient**(*cwd=None*, *home=None*, *read_only=False*, *global_writes=False*, ***kwargs*)
Bases: `omniduct.duct.Duct`, `omniduct.utils.magics.MagicsProvider`

An abstract class providing the common API for all filesystem clients.

Class Attributes

- **DUCT_TYPE** (*Duct.Type*) – The type of *Duct* protocol implemented by this class.
- **DEFAULT_PORT** (*int*) – The default port for the filesystem service (defined by subclasses).

Attributes inherited from Duct:

protocol (str): The name of the protocol for which this instance was created (especially useful if a *Duct* subclass supports multiple protocols).

name (str): The name given to this *Duct* instance (defaults to class name).

host (str): The host name providing the service (will be '127.0.0.1', if service is port forwarded from remote; use *._host* to see remote host).

port (int): The port number of the service (will be the port-forwarded local port, if relevant; for remote port use *._port*).

username (str, bool): The username to use for the service. password (str, bool): The password to use for the service. registry (None, omniduct.registry.DuctRegistry): A reference to a

DuctRegistry instance for runtime lookup of other services.

remote (None, omniduct.remotes.base.RemoteClient): A reference to a *RemoteClient* instance to manage connections to remote services.

cache (None, omniduct.caches.base.Cache): A reference to a *Cache* instance to add support for caching, if applicable.

connection_fields (tuple<str>, list<str>): A list of instance attributes to monitor for changes, whereupon the *Duct* instance should automatically disconnect. By default, the following attributes are monitored: 'host', 'port', 'remote', 'username', and 'password'.

prepared_fields (tuple<str>, list<str>): A list of instance attributes to be populated (if their values are callable) when the instance first connects to a service. Refer to *Duct.prepare* and *Duct._prepare* for more details. By default, the following attributes are prepared: '_host', '_port', '_username', and '_password'.

Additional attributes including *host*, *port*, *username* and *password* are documented inline.

Class Attributes:

AUTO_LOGGING_SCOPE (bool): Whether this class should be used by omniduct logging code as a "scope". Should be overridden by subclasses as appropriate.

DUCT_TYPE (Duct.Type): The type of *Duct* service that is provided by this Duct instance. Should be overridden by subclasses as appropriate.

PROTOCOLS (list<str>): The name(s) of any protocols that should be associated with this class. Should be overridden by subclasses as appropriate.

`__init__`(*cwd=None*, *home=None*, *read_only=False*, *global_writes=False*, ***kwargs*)

protocol (str, None): Name of protocol (used by Duct registries to inform Duct instances of how they were instantiated).

name (str, None): The name to used by the *Duct* instance (defaults to class name if not specified).

registry (DuctRegistry, None): The registry to use to lookup remote and/or cache instance specified by name.

remote (str, RemoteClient): The remote by which the ducted service should be contacted.

host (str): The hostname of the service to be used by this client. port (int): The port of the service to be used by this client. username (str, bool, None): The username to authenticate with if necessary.

If True, then users will be prompted at runtime for credentials.

password (str, bool, None): The password to authenticate with if necessary. If True, then users will be prompted at runtime for credentials.

cache(Cache, None): The cache client to be attached to this instance. Cache will only used by specific methods as configured by the client.

cache_namespace(str, None): The namespace to use by default when writing to the cache.

FileSystemClient Quirks:

cwd (None, str): The path prefix to use as the current working directory (if None, the user's home directory is used where that makes sense).

home (None, str): The path prefix to use as the current users' home directory. If not specified, it will default to an implementation- specific value (often '/').

read_only (bool): Whether the filesystem should only be able to perform read operations.

global_writes (bool): Whether to allow writes outside of the user's home folder.

**kwargs (dict): Additional keyword arguments to passed on to subclasses.

`path_home`

The path prefix to use as the current users' home directory. Unless *cwd* is set, this will be the prefix to use for all non-absolute path references on this filesystem. This is assumed not to change between connections, and so will not be updated on client reconnections. Unless *global_writes* is set to *True*, this will be the only folder into which this client is permitted to write.

Type str

`path_cwd`

The path prefix associated with the current working directory. If not otherwise set, it will be the users' home directory, and will be the prefix used by all non-absolute path references on this filesystem.

Type str

`path_separator`

The character(s) to use in separating path components. Typically this will be '/'.

Type str

`path_join`(*path*, **components*)

Generate a new path by joining together multiple paths.

If any component starts with *self.path_separator* or '~', then all previous path components are discarded, and the effective base path becomes that component (with '~' expanding to *self.path_home*). Note that this method does *not* simplify paths components like '..'. Use *self.path_normpath* for this purpose.

Parameters

- **`path`** (*`str`*) – The base path to which components should be joined.
- **`*components`** (*`str`*) – Any additional components to join to the base path.

Returns The path resulting from joining all of the components nominated, in order, to the base path.

Return type str

path_basename (*path*)
Extract the last component of a given path.

Components are determined by splitting by *self.path_separator*. Note that if a path ends with a path separator, the basename will be the empty string.

Parameters **path** (*str*) – The path from which the basename should be extracted.

Returns The extracted basename.

Return type str

path_dirname (*path*)
Extract the parent directory for provided path.

This method returns the entire path except for the basename (the last component), where components are determined by splitting by *self.path_separator*.

Parameters **path** (*str*) – The path from which the directory path should be extracted.

Returns The extracted directory path.

Return type str

path_normpath (*path*)
Normalise a pathname.

This method returns the normalised (absolute) path corresponding to *path* on this filesystem.

Parameters **path** (*str*) – The path to normalise (make absolute).

Returns The normalised path.

Return type str

read_only
Whether this filesystem client should be permitted to attempt any write operations.

Type bool

global_writes
Whether writes should be permitted outside of home directory. This write-lock is designed to prevent inadvertent scripted writing in potentially dangerous places.

Type bool

exists (*path*)
Check whether nominated path exists on this filesytem.

Parameters **path** (*str*) – The path for which to check existence.

Returns

True **if file/folder exists at nominated path, and** ***False*** otherwise.

Return type bool

isdir (*path*)
Check whether a nominated path is directory.

Parameters **path** (*str*) – The path for which to check directory nature.

Returns *True* if folder exists at nominated path, and *False* otherwise.

Return type bool

isfile (*path*)
Check whether a nominated path is a file.

Parameters **path** (*str*) – The path for which to check file nature.

Returns *True* if a file exists at nominated path, and *False* otherwise.

Return type bool

dir(*path=None*)
Retrieve information about the children of a nominated directory.

This method returns a generator over *FileSystemFileDesc* objects that represent the files/directories that a present as children of the nominated path. If *path* is not a directory, an exception is raised. The path is interpreted as being relative to the current working directory (on remote filesytems, this will typically be the home folder).

Parameters **path** (*str*) – The path to examine for children.

Returns The children of *path* represented as *FileSystemFileDesc* objects.

Return type generator<FileSystemFileDesc>

This method should return a generator over *FileSystemFileDesc* objects.

listdir(*path=None*)
Retrieve the names of the children of a nomianted directory.

This method inspects the contents of a directory using *.dir(path)*, and returns the names of child members as strings. *path* is interpreted relative to the current working directory (on remote filesytems, this will typically be the home folder).

Parameters **path** (*str*) – The path of the directory from which to enumerate filenames.

Returns The names of all children of the nominated directory.

Return type list<str>

showdir(*path=None*)
Return a dataframe representation of a directory.

This method returns a *pandas.DataFrame* representation of the contents of a path, which are retrieved using *.dir(path)*. The exact columns will vary from filesystem to filesystem, depending on the fields returned by *.dir()*, but the returned DataFrame is guaranteed to at least have the columns: 'name' and 'type'.

Parameters **path** (*str*) – The path of the directory from which to show contents.

Returns A DataFrame representation of the contents of the nominated directory.

Return type pandas.DataFrame

walk(*path=None*)
Explore the filesystem tree starting at a nominated path.

This method returns a generator which recursively walks over all paths that are children of *path*, one result for each directory, of form: (<path name>, [<directory 1>, . . .], [<file 1>, . . .])

Parameters **path** (*str*) – The path of the directory from which to enumerate contents.

Returns A generator of tuples, each tuple being associated with one directory that is either *path* or one of its descendants.

Return type generator<tuple>

find(*path_prefix=None*, ***attrs*)
Find a file or directory based on certain attributes.

This method searches for files or folders which satisfy certain constraints on the attributes of the file (as encoded into *FileSystemFileDesc*). Note that without attribute constraints, this method will function identically to *self.dir*.

Parameters

- **`path_prefix`** (*`str`*) – The path under which files/directories should be found.
- **`**attrs`** (*`dict`*) – Constraints on the fields of the *FileSystemFileDesc* objects associated with this filesystem, as constant values or callable objects (in which case the object will be called and should return True if attribute value is match, and False otherwise).

Returns

A generator over *FileSystemFileDesc* objects that are descendents of *path_prefix* and which statisfy provided constraints.

Return type generator<FileSystemFileDesc>

`mkdir`(*path*, *recursive=True*, *exist_ok=False*)

Create a directory at the given path.

Parameters

- **`path`** (*`str`*) – The path of the directory to create.
- **`recursive`** (*`bool`*) – Whether to recursively create any parents of this path if they do not already exist.

Note: *exist_ok* is passed onto subclass implementations of *_mkdir* rather that implementing the existence check using *.exists* so that they can avoid the overhead associated with multiple operations, which can be costly in some cases.

`remove`(*path*, *recursive=False*)

Remove file(s) at a nominated path.

Directories (and their contents) will not be removed unless *recursive* is set to *True*.

Parameters

- **`path`** (*`str`*) – The path of the file/directory to be removed.
- **`recursive`** (*`bool`*) – Whether to remove directories and all of their contents.

`open`(*path*, *mode='rt'*)

Open a file for reading and/or writing.

This method opens the file at the given path for reading and/or writing operations. The object returned is programmatically interchangeable with any other Python file-like object, including specification of file modes. If the file is opened in write mode, changes will only be flushed to the source filesystem when the file is closed.

Parameters

- **`path`** (*`str`*) – The path of the file to open.
- **`mode`** (*`str`*) – All standard Python file modes.

Returns An opened file-like object.

Return type *FileSystemFile* or file-like

`download`(*source*, *dest=None*, *overwrite=False*, *fs=None*)

Download files to another filesystem.

This method (recursively) downloads a file/folder from path *source* on this filesystem to the path *dest* on filesytem *fs*, overwriting any existing file if *overwrite* is *True*.

Parameters

- **source** (*str*) – The path on this filesystem of the file to download to the nominated filesystem (*fs*). If *source* ends with '/' then contents of the the *source* directory will be copied into destination folder, and will throw an error if path does not resolve to a directory.
- **dest** (*str*) – The destination path on filesystem (*fs*). If not specified, the file/folder is downloaded into the default path, usually one's home folder. If *dest* ends with '/', and corresponds to a directory, the contents of source will be copied instead of copying the entire folder. If *dest* is otherwise a directory, an exception will be raised.
- **overwrite** (*bool*) – *True* if the contents of any existing file by the same name should be overwritten, *False* otherwise.
- **fs** (*FileSystemClient*) – The FileSystemClient into which the nominated file/folder *source* should be downloaded. If not specified, defaults to the local filesystem.

upload(*source*, *dest=None*, *overwrite=False*, *fs=None*)

Upload files from another filesystem.

This method (recursively) uploads a file/folder from path *source* on filesystem *fs* to the path *dest* on this filesytem, overwriting any existing file if *overwrite* is *True*. This is equivalent to *fs.download(…, fs=self)*.

Parameters

- **source** (*str*) – The path on the specified filesystem (*fs*) of the file to upload to this filesystem. If *source* ends with '/', and corresponds to a directory, the contents of source will be copied instead of copying the entire folder.
- **dest** (*str*) – The destination path on this filesystem. If not specified, the file/folder is uploaded into the default path, usually one's home folder, on this filesystem. If *dest* ends with '/' then file will be copied into destination folder, and will throw an error if path does not resolve to a directory.
- **overwrite** (*bool*) – *True* if the contents of any existing file by the same name should be overwritten, *False* otherwise.
- **fs** (*FileSystemClient*) – The FileSystemClient from which to load the file/folder at *source*. If not specified, defaults to the local filesystem.

connect()

Connect to the service backing this client.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Returns A reference to the current object.

Return type *Duct* instance

disconnect()

Disconnect this client from backing service.

This method is automatically called during reconnections and/or at Python interpreter shutdown. It first calls *Duct._disconnect* (which should be implemented by subclasses) and then notifies the *RemoteClient* subclass, if present, to stop port-forwarding the remote service.

Returns A reference to this object.

Return type *Duct* instance

is_connected()

Check whether this *Duct* instances is currently connected.

This method checks to see whether a *Duct* instance is currently connected. This is performed by verifying that the remote host and port are still accessible, and then by calling *Duct._is_connected*, which should be implemented by subclasses.

Returns Whether this *Duct* instance is currently connected.

Return type bool

prepare()

Prepare a Duct subclass for use (if not already prepared).

This method is called before the value of any of the fields referenced in *self.connection_fields* are retrieved. The fields include, by default: 'host', 'port', 'remote', 'cache', 'username', and 'password'. Subclasses may add or subtract from these special fields.

When called, it first checks whether the instance has already been prepared, and if not calls *_prepare* and then records that the instance has been successfully prepared.

FileSystemClient Quirks: This method may be overridden by subclasses, but provides the following default behaviour:

- Ensures *self.registry*, *self.remote* and *self.cache* values are instances of the right types.
- It replaces string values of *self.remote* and *self.cache* with remotes and caches looked up using *self.registry.lookup*.
- It looks through each of the fields nominated in *self.prepared_fields* and, if the corresponding value is callable, sets the value of that field to result of calling that value with a reference to *self*. By default, *prepared_fields* contains '_host', '_port', '_username', and '_password'.
- Ensures value of self.port is an integer (or None).

class omniduct.filesystems.base.**FileSystemFile**(*fs*, *path*, *mode='r'*)

Bases: `object`

A file-like implementation that is interchangeable with native Python file objects, allowing remote files to be treated identically to local files both by omniduct, the user and other libraries.

__init__(*fs*, *path*, *mode='r'*)

Initialize self. See help(type(self)) for accurate signature.

class omniduct.filesystems.base.**FileSystemFileDesc**

Bases: `omniduct.filesystems.base.Node`

A representation of a file/directory stored within an Omniduct FileSystemClient.

5.3.2 Subclass Reference

For comprehensive documentation on any particular subclass, please refer to one of the below documents.

LocalFsClient

class omniduct.filesystems.local.**LocalFsClient**(*cwd=None*, *home=None*, *read_only=False*, *global_writes=False*, ***kwargs*)

Bases: `omniduct.filesystems.base.FileSystemClient`

LocalFsClient is a *Duct* that implements the *FileSystemClient* common API, and exposes the local filesystem.

Unlike most other filesystems, *LocalFsClient* defaults to the current working directory on the local machine, rather than the home directory as used on remote filesystems. To change this, you can always execute: ` local_fs.path_cwd = local_fs.path_home `

Attributes inherited from Duct:

protocol (str): The name of the protocol for which this instance was created (especially useful if a *Duct* subclass supports multiple protocols).

name (str): The name given to this *Duct* instance (defaults to class name).

host (str): The host name providing the service (will be '127.0.0.1', if service is port forwarded from remote; use *._host* to see remote host).

port (int): The port number of the service (will be the port-forwarded local port, if relevant; for remote port use *._port*).

username (str, bool): The username to use for the service. password (str, bool): The password to use for the service. registry (None, omniduct.registry.DuctRegistry): A reference to a

DuctRegistry instance for runtime lookup of other services.

remote (None, omniduct.remotes.base.RemoteClient): A reference to a *RemoteClient* instance to manage connections to remote services.

cache (None, omniduct.caches.base.Cache): A reference to a *Cache* instance to add support for caching, if applicable.

connection_fields (tuple<str>, list<str>): A list of instance attributes to monitor for changes, whereupon the *Duct* instance should automatically disconnect. By default, the following attributes are monitored: 'host', 'port', 'remote', 'username', and 'password'.

prepared_fields (tuple<str>, list<str>): A list of instance attributes to be populated (if their values are callable) when the instance first connects to a service. Refer to *Duct.prepare* and *Duct._prepare* for more details. By default, the following attributes are prepared: '_host', '_port', '_username', and '_password'.

Additional attributes including *host*, *port*, *username* and *password* are documented inline.

Class Attributes:

AUTO_LOGGING_SCOPE (bool): Whether this class should be used by omniduct logging code as a "scope". Should be overridden by subclasses as appropriate.

DUCT_TYPE (Duct.Type): The type of *Duct* service that is provided by this Duct instance. Should be overridden by subclasses as appropriate.

PROTOCOLS (list<str>): The name(s) of any protocols that should be associated with this class. Should be overridden by subclasses as appropriate.

`class Type`

Bases: `enum.Enum`

The *Duct.Type* enum specifies all of the permissible values of *Duct.DUCT_TYPE*. Also determines the order in which ducts are loaded by DuctRegistry.

`__init__`(*cwd=None*, *home=None*, *read_only=False*, *global_writes=False*, ***kwargs*)

protocol (str, None): Name of protocol (used by Duct registries to inform Duct instances of how they were instantiated).

name (str, None): The name to used by the *Duct* instance (defaults to class name if not specified).

registry (DuctRegistry, None): The registry to use to lookup remote and/or cache instance specified by name.

remote (str, RemoteClient): The remote by which the ducted service should be contacted.

host (str): The hostname of the service to be used by this client. port (int): The port of the service to be used by this client. username (str, bool, None): The username to authenticate with if necessary.

If True, then users will be prompted at runtime for credentials.

password (str, bool, None): The password to authenticate with if necessary. If True, then users will be prompted at runtime for credentials.

cache(Cache, None): The cache client to be attached to this instance. Cache will only used by specific methods as configured by the client.

cache_namespace(str, None): The namespace to use by default when writing to the cache.

FileSystemClient Quirks:

cwd (None, str): The path prefix to use as the current working directory (if None, the user's home directory is used where that makes sense).

home (None, str): The path prefix to use as the current users' home directory. If not specified, it will default to an implementation- specific value (often '/').

read_only (bool): Whether the filesystem should only be able to perform read operations.

global_writes (bool): Whether to allow writes outside of the user's home folder.

**kwargs (dict): Additional keyword arguments to passed on to subclasses.

connect()

Connect to the service backing this client.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Returns A reference to the current object.

Return type *Duct* instance

dir(*path=None*)

Retrieve information about the children of a nominated directory.

This method returns a generator over *FileSystemFileDesc* objects that represent the files/directories that a present as children of the nominated path. If *path* is not a directory, an exception is raised. The path is interpreted as being relative to the current working directory (on remote filesytems, this will typically be the home folder).

Parameters **path** (*str*) – The path to examine for children.

Returns The children of *path* represented as *FileSystemFileDesc* objects.

Return type generator<FileSystemFileDesc>

disconnect()

Disconnect this client from backing service.

This method is automatically called during reconnections and/or at Python interpreter shutdown. It first calls *Duct._disconnect* (which should be implemented by subclasses) and then notifies the *RemoteClient* subclass, if present, to stop port-forwarding the remote service.

Returns A reference to this object.

Return type *Duct* instance

download(*source*, *dest=None*, *overwrite=False*, *fs=None*)
Download files to another filesystem.

This method (recursively) downloads a file/folder from path *source* on this filesystem to the path *dest* on filesytem *fs*, overwriting any existing file if *overwrite* is *True*.

Parameters

- **source** (*str*) – The path on this filesystem of the file to download to the nominated filesystem (*fs*). If *source* ends with '/' then contents of the the *source* directory will be copied into destination folder, and will throw an error if path does not resolve to a directory.
- **dest** (*str*) – The destination path on filesystem (*fs*). If not specified, the file/folder is downloaded into the default path, usually one's home folder. If *dest* ends with '/', and corresponds to a directory, the contents of source will be copied instead of copying the entire folder. If *dest* is otherwise a directory, an exception will be raised.
- **overwrite** (*bool*) – *True* if the contents of any existing file by the same name should be overwritten, *False* otherwise.
- **fs** (FileSystemClient) – The FileSystemClient into which the nominated file/folder *source* should be downloaded. If not specified, defaults to the local filesystem.

exists(*path*)
Check whether nominated path exists on this filesytem.

Parameters **path** (*str*) – The path for which to check existence.

Returns

True* if file/folder exists at nominated path, and *False otherwise.

Return type bool

find(*path_prefix=None*, ***attrs*)
Find a file or directory based on certain attributes.

This method searches for files or folders which satisfy certain constraints on the attributes of the file (as encoded into *FileSystemFileDesc*). Note that without attribute constraints, this method will function identically to *self.dir*.

Parameters

- **path_prefix** (*str*) – The path under which files/directories should be found.
- ****attrs** (*dict*) – Constraints on the fields of the *FileSystemFileDesc* objects associated with this filesystem, as constant values or callable objects (in which case the object will be called and should return True if attribute value is match, and False otherwise).

Returns

A generator over *FileSystemFileDesc* objects that are descendents of *path_prefix* and which statisfy provided constraints.

Return type generator<FileSystemFileDesc>

classmethod for_protocol(*protocol*)
Retrieve a *Duct* subclass for a given protocol.

Parameters **protocol** (*str*) – The protocol of interest.

Returns

The appropriate class for the provided, partially constructed with the *protocol* keyword argument set appropriately.

Return type functools.partial object

Raises `DuctProtocolUnknown` – If no class has been defined that offers the named protocol.

global_writes

Whether writes should be permitted outside of home directory. This write-lock is designed to prevent inadvertent scripted writing in potentially dangerous places.

Type bool

host

The host name providing the service, or '127.0.0.1' if *self.remote* is not *None*, whereupon the service will be port-forwarded locally. You can view the remote hostname using *duct._host*, and change the remote host at runtime using: *duct.host = '<host>'*.

Type str

is_connected()

Check whether this *Duct* instances is currently connected.

This method checks to see whether a *Duct* instance is currently connected. This is performed by verifying that the remote host and port are still accessible, and then by calling *Duct._is_connected*, which should be implemented by subclasses.

Returns Whether this *Duct* instance is currently connected.

Return type bool

isdir(*path*)

Check whether a nominated path is directory.

Parameters **path** (`str`) – The path for which to check directory nature.

Returns *True* if folder exists at nominated path, and *False* otherwise.

Return type bool

isfile(*path*)

Check whether a nominated path is a file.

Parameters **path** (`str`) – The path for which to check file nature.

Returns *True* if a file exists at nominated path, and *False* otherwise.

Return type bool

listdir(*path=None*)

Retrieve the names of the children of a nomianted directory.

This method inspects the contents of a directory using *.dir(path)*, and returns the names of child members as strings. *path* is interpreted relative to the current working directory (on remote filesytems, this will typically be the home folder).

Parameters **path** (`str`) – The path of the directory from which to enumerate filenames.

Returns The names of all children of the nominated directory.

Return type list<str>

mkdir(*path*, *recursive=True*, *exist_ok=False*)

Create a directory at the given path.

Parameters

- **path** (*str*) – The path of the directory to create.
- **recursive** (*bool*) – Whether to recursively create any parents of this path if they do not already exist.

Note: *exist_ok* is passed onto subclass implementations of *_mkdir* rather that implementing the existence check using *.exists* so that they can avoid the overhead associated with multiple operations, which can be costly in some cases.

open (*path*, *mode='rt'*)
Open a file for reading and/or writing.

This method opens the file at the given path for reading and/or writing operations. The object returned is programmatically interchangeable with any other Python file-like object, including specification of file modes. If the file is opened in write mode, changes will only be flushed to the source filesystem when the file is closed.

Parameters

- **path** (*str*) – The path of the file to open.
- **mode** (*str*) – All standard Python file modes.

Returns An opened file-like object.

Return type *FileSystemFile* or file-like

password
Some services require authentication in order to connect to the service, in which case the appropriate password can be specified. If *True* was provided at instantiation, you will be prompted to type your password at runtime when necessary. If *False* was provided, then *None* will be returned. You can specify a different password at runtime using: *duct.password = '<password>'*.

Type str

path_basename (*path*)
Extract the last component of a given path.

Components are determined by splitting by *self.path_separator*. Note that if a path ends with a path separator, the basename will be the empty string.

Parameters **path** (*str*) – The path from which the basename should be extracted.

Returns The extracted basename.

Return type str

path_cwd
The path prefix associated with the current working directory. If not otherwise set, it will be the users' home directory, and will be the prefix used by all non-absolute path references on this filesystem.

Type str

path_dirname (*path*)
Extract the parent directory for provided path.

This method returns the entire path except for the basename (the last component), where components are determined by splitting by *self.path_separator*.

Parameters **path** (*str*) – The path from which the directory path should be extracted.

Returns The extracted directory path.

Return type str

path_home
The path prefix to use as the current users' home directory. Unless *cwd* is set, this will be the prefix to use for all non-absolute path references on this filesystem. This is assumed not to change between connections, and so will not be updated on client reconnections. Unless *global_writes* is set to *True*, this will be the only folder into which this client is permitted to write.

Type str

path_join(*path*, **components*)
Generate a new path by joining together multiple paths.

If any component starts with *self.path_separator* or '~', then all previous path components are discarded, and the effective base path becomes that component (with '~' expanding to *self.path_home*). Note that this method does *not* simplify paths components like '..'. Use *self.path_normpath* for this purpose.

Parameters

- **path** (`str`) – The base path to which components should be joined.
- ***components** (`str`) – Any additional components to join to the base path.

Returns The path resulting from joining all of the components nominated, in order, to the base path.

Return type str

path_normpath(*path*)
Normalise a pathname.

This method returns the normalised (absolute) path corresponding to *path* on this filesystem.

Parameters **path** (`str`) – The path to normalise (make absolute).

Returns The normalised path.

Return type str

path_separator
The character(s) to use in separating path components. Typically this will be '/'.

Type str

port
The local port for the service. If *self.remote* is not *None*, the port will be port-forwarded from the remote host. To see the port used on the remote host refer to *duct._port*. You can change the remote port at runtime using: *duct.port = <port>*.

Type int

prepare()
Prepare a Duct subclass for use (if not already prepared).

This method is called before the value of any of the fields referenced in *self.connection_fields* are retrieved. The fields include, by default: 'host', 'port', 'remote', 'cache', 'username', and 'password'. Subclasses may add or subtract from these special fields.

When called, it first checks whether the instance has already been prepared, and if not calls *_prepare* and then records that the instance has been successfully prepared.

read_only
Whether this filesystem client should be permitted to attempt any write operations.

Type bool

reconnect()
Disconnects, and then reconnects, this client.

Note: This is equivalent to *duct.disconnect().connect()*.

Returns A reference to this object.

Return type *Duct* instance

remove(*path*, *recursive=False*)
Remove file(s) at a nominated path.

Directories (and their contents) will not be removed unless *recursive* is set to *True*.

Parameters

- **path** (*str*) – The path of the file/directory to be removed.
- **recursive** (*bool*) – Whether to remove directories and all of their contents.

reset()
Reset this *Duct* instance to its pre-preparation state.

This method disconnects from the service, resets any temporary authentication and restores the values of the attributes listed in *prepared_fields* to their values as of when *Duct.prepare* was called.

Returns A reference to this object.

Return type *Duct* instance

showdir(*path=None*)
Return a dataframe representation of a directory.

This method returns a *pandas.DataFrame* representation of the contents of a path, which are retrieved using *.dir(path)*. The exact columns will vary from filesystem to filesystem, depending on the fields returned by *.dir()*, but the returned DataFrame is guaranteed to at least have the columns: 'name' and 'type'.

Parameters path (*str*) – The path of the directory from which to show contents.

Returns A DataFrame representation of the contents of the nominated directory.

Return type pandas.DataFrame

upload(*source*, *dest=None*, *overwrite=False*, *fs=None*)
Upload files from another filesystem.

This method (recursively) uploads a file/folder from path *source* on filesystem *fs* to the path *dest* on this filesytem, overwriting any existing file if *overwrite* is *True*. This is equivalent to *fs.download(. . . , fs=self)*.

Parameters

- **source** (*str*) – The path on the specified filesystem (*fs*) of the file to upload to this filesystem. If *source* ends with '/', and corresponds to a directory, the contents of source will be copied instead of copying the entire folder.
- **dest** (*str*) – The destination path on this filesystem. If not specified, the file/folder is uploaded into the default path, usually one's home folder, on this filesystem. If *dest* ends with '/' then file will be copied into destination folder, and will throw an error if path does not resolve to a directory.
- **overwrite** (*bool*) – *True* if the contents of any existing file by the same name should be overwritten, *False* otherwise.
- **fs** (FileSystemClient) – The FileSystemClient from which to load the file/folder at *source*. If not specified, defaults to the local filesystem.

`username`
Some services require authentication in order to connect to the service, in which case the appropriate username can be specified. If not specified at instantiation, your local login name will be used. If *True* was provided, you will be prompted to type your username at runtime as necessary. If *False* was provided, then *None* will be returned. You can specify a different username at runtime using: *duct.username = '<username>'*.

Type str

`walk`(*path=None*)
Explore the filesystem tree starting at a nominated path.

This method returns a generator which recursively walks over all paths that are children of *path*, one result for each directory, of form: (<path name>, [<directory 1>, . . .], [<file 1>, . . .])

Parameters **`path`** (*`str`*) – The path of the directory from which to enumerate contents.

Returns A generator of tuples, each tuple being associated with one directory that is either *path* or one of its descendants.

Return type generator<tuple>

S3Client

class `omniduct.filesystems.s3.`**`S3Client`**(*cwd=None*, *home=None*, *read_only=False*, *global_writes=False*, ***kwargs*)
Bases: *`omniduct.filesystems.base.FileSystemClient`*

This Duct connects to an Amazon S3 bucket instance using the *boto3* library. Authentication is (optionally) handled using *opinel*.

Attributes

- **bucket** (*str*) – The name of the Amazon S3 bucket to use.
- **aws_profile** (*str*) – The name of configured AWS profile to use. This should refer to the name of a profile configured in, for example, *~/.aws/credentials*. Authentication is handled by the *opinel* library, which is also aware of environment variables.

Attributes inherited from Duct:

protocol (str): The name of the protocol for which this instance was created (especially useful if a *Duct* subclass supports multiple protocols).

name (str): The name given to this *Duct* instance (defaults to class name).

host (str): The host name providing the service (will be '127.0.0.1', if service is port forwarded from remote; use *._host* to see remote host).

port (int): The port number of the service (will be the port-forwarded local port, if relevant; for remote port use *._port*).

username (str, bool): The username to use for the service. password (str, bool): The password to use for the service. registry (None, omniduct.registry.DuctRegistry): A reference to a

DuctRegistry instance for runtime lookup of other services.

remote (None, omniduct.remotes.base.RemoteClient): A reference to a *RemoteClient* instance to manage connections to remote services.

cache (None, omniduct.caches.base.Cache): A reference to a *Cache* instance to add support for caching, if applicable.

connection_fields (tuple<str>, list<str>): A list of instance attributes to monitor for changes, whereupon the *Duct* instance should automatically disconnect. By default, the following attributes are monitored: 'host', 'port', 'remote', 'username', and 'password'.

prepared_fields (tuple<str>, list<str>): A list of instance attributes to be populated (if their values are callable) when the instance first connects to a service. Refer to *Duct.prepare* and *Duct._prepare* for more details. By default, the following attributes are prepared: '_host', '_port', '_username', and '_password'.

Additional attributes including *host*, *port*, *username* and *password* are documented inline.

Class Attributes:

AUTO_LOGGING_SCOPE (bool): Whether this class should be used by omniduct logging code as a "scope". Should be overridden by subclasses as appropriate.

DUCT_TYPE (Duct.Type): The type of *Duct* service that is provided by this Duct instance. Should be overridden by subclasses as appropriate.

PROTOCOLS (list<str>): The name(s) of any protocols that should be associated with this class. Should be overridden by subclasses as appropriate.

class Type
Bases: `enum.Enum`

The *Duct.Type* enum specifies all of the permissible values of *Duct.DUCT_TYPE*. Also determines the order in which ducts are loaded by DuctRegistry.

__init__(*cwd=None*, *home=None*, *read_only=False*, *global_writes=False*, ***kwargs*)

protocol (str, None): Name of protocol (used by Duct registries to inform Duct instances of how they were instantiated).

name (str, None): The name to used by the *Duct* instance (defaults to class name if not specified).

registry (DuctRegistry, None): The registry to use to lookup remote and/or cache instance specified by name.

remote (str, RemoteClient): The remote by which the ducted service should be contacted.

host (str): The hostname of the service to be used by this client. port (int): The port of the service to be used by this client. username (str, bool, None): The username to authenticate with if necessary.

If True, then users will be prompted at runtime for credentials.

password (str, bool, None): The password to authenticate with if necessary. If True, then users will be prompted at runtime for credentials.

cache(Cache, None): The cache client to be attached to this instance. Cache will only used by specific methods as configured by the client.

cache_namespace(str, None): The namespace to use by default when writing to the cache.

FileSystemClient Quirks:

cwd (None, str): The path prefix to use as the current working directory (if None, the user's home directory is used where that makes sense).

home (None, str): The path prefix to use as the current users' home directory. If not specified, it will default to an implementation- specific value (often '/').

read_only (bool): Whether the filesystem should only be able to perform read operations.

global_writes (bool): Whether to allow writes outside of the user's home folder.

**kwargs (dict): Additional keyword arguments to passed on to subclasses.

S3Client Quirks: bucket (str): The name of the Amazon S3 bucket to use. aws_profile (str): The name of configured AWS profile to use. This should

refer to the name of a profile configured in, for example, *~/.aws/credentials*. Authentication is (optionally) handled by the *opinel* library, which is also aware of environment variables.

use_opinel (bool): Use Opinel to extract AWS credentials. This is mainly useful if you have used opinel to set up MFA. Note: Opinel must be installed manually alongside omniduct to take advantage of this feature.

session (botocore.session.Session): A pre-configured botocore Session instance to use instead of creating a new one when this client connects.

path_separator (str): Amazon S3 is essentially a key-based storage system, and so one is free to choose an arbitrary "directory" separator. This defaults to '/' for consistency with other filesystems.

skip_hadoop_artifacts (bool): Whether to skip hadoop artifacts like '*_$folder$' when enumerating directories (default=True).

Note 1: aws_profile, if specified, should be the name of a profile as specified in ~/.aws/credentials. Authentication is handled by the *opinel* library, which is also aware of environment variables. Set up your command line aws client, and if it works, this should too.

Note 2: Some institutions have nuanced AWS configurations that with configurations that are generated by scripts. It may be useful in these environments to subclass *S3Client* and override the *_get_boto3_session* method to suit your needs.

`connect`()

Connect to the service backing this client.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Returns A reference to the current object.

Return type *Duct* instance

`dir`(*path=None*)

Retrieve information about the children of a nominated directory.

This method returns a generator over *FileSystemFileDesc* objects that represent the files/directories that a present as children of the nominated path. If *path* is not a directory, an exception is raised. The path is interpreted as being relative to the current working directory (on remote filesytems, this will typically be the home folder).

Parameters **`path`** (*`str`*) – The path to examine for children.

Returns The children of *path* represented as *FileSystemFileDesc* objects.

Return type generator<FileSystemFileDesc>

`disconnect`()

Disconnect this client from backing service.

This method is automatically called during reconnections and/or at Python interpreter shutdown. It first calls *Duct._disconnect* (which should be implemented by subclasses) and then notifies the *RemoteClient* subclass, if present, to stop port-forwarding the remote service.

Returns A reference to this object.

Return type *Duct* instance

download(*source*, *dest=None*, *overwrite=False*, *fs=None*)
Download files to another filesystem.

This method (recursively) downloads a file/folder from path *source* on this filesystem to the path *dest* on filesytem *fs*, overwriting any existing file if *overwrite* is *True*.

Parameters

- **source** (*str*) – The path on this filesystem of the file to download to the nominated filesystem (*fs*). If *source* ends with '/' then contents of the the *source* directory will be copied into destination folder, and will throw an error if path does not resolve to a directory.
- **dest** (*str*) – The destination path on filesystem (*fs*). If not specified, the file/folder is downloaded into the default path, usually one's home folder. If *dest* ends with '/', and corresponds to a directory, the contents of source will be copied instead of copying the entire folder. If *dest* is otherwise a directory, an exception will be raised.
- **overwrite** (*bool*) – *True* if the contents of any existing file by the same name should be overwritten, *False* otherwise.
- **fs** (*FileSystemClient*) – The FileSystemClient into which the nominated file/folder *source* should be downloaded. If not specified, defaults to the local filesystem.

exists(*path*)
Check whether nominated path exists on this filesytem.

Parameters **path** (*str*) – The path for which to check existence.

Returns

True* if file/folder exists at nominated path, and *False otherwise.

Return type bool

find(*path_prefix=None*, ***attrs*)
Find a file or directory based on certain attributes.

This method searches for files or folders which satisfy certain constraints on the attributes of the file (as encoded into *FileSystemFileDesc*). Note that without attribute constraints, this method will function identically to *self.dir*.

Parameters

- **path_prefix** (*str*) – The path under which files/directories should be found.
- ****attrs** (*dict*) – Constraints on the fields of the *FileSystemFileDesc* objects associated with this filesystem, as constant values or callable objects (in which case the object will be called and should return True if attribute value is match, and False otherwise).

Returns

A generator over *FileSystemFileDesc* objects that are descendents of *path_prefix* and which statisfy provided constraints.

Return type generator<FileSystemFileDesc>

classmethod for_protocol(*protocol*)
Retrieve a *Duct* subclass for a given protocol.

Parameters **protocol** (*str*) – The protocol of interest.

Returns

The appropriate class for the provided, partially constructed with the *protocol* keyword argument set appropriately.

Return type functools.partial object

Raises `DuctProtocolUnknown` – If no class has been defined that offers the named protocol.

global_writes

Whether writes should be permitted outside of home directory. This write-lock is designed to prevent inadvertent scripted writing in potentially dangerous places.

Type bool

host

The host name providing the service, or '127.0.0.1' if *self.remote* is not *None*, whereupon the service will be port-forwarded locally. You can view the remote hostname using *duct._host*, and change the remote host at runtime using: *duct.host = '<host>'*.

Type str

is_connected()

Check whether this *Duct* instances is currently connected.

This method checks to see whether a *Duct* instance is currently connected. This is performed by verifying that the remote host and port are still accessible, and then by calling *Duct._is_connected*, which should be implemented by subclasses.

Returns Whether this *Duct* instance is currently connected.

Return type bool

isdir(*path*)

Check whether a nominated path is directory.

Parameters **path** (*str*) – The path for which to check directory nature.

Returns *True* if folder exists at nominated path, and *False* otherwise.

Return type bool

isfile(*path*)

Check whether a nominated path is a file.

Parameters **path** (*str*) – The path for which to check file nature.

Returns *True* if a file exists at nominated path, and *False* otherwise.

Return type bool

listdir(*path=None*)

Retrieve the names of the children of a nomianted directory.

This method inspects the contents of a directory using *.dir(path)*, and returns the names of child members as strings. *path* is interpreted relative to the current working directory (on remote filesytems, this will typically be the home folder).

Parameters **path** (*str*) – The path of the directory from which to enumerate filenames.

Returns The names of all children of the nominated directory.

Return type list<str>

mkdir(*path*, *recursive=True*, *exist_ok=False*)

Create a directory at the given path.

Parameters

- **path** (*str*) – The path of the directory to create.
- **recursive** (*bool*) – Whether to recursively create any parents of this path if they do not already exist.

Note: *exist_ok* is passed onto subclass implementations of *_mkdir* rather that implementing the existence check using *.exists* so that they can avoid the overhead associated with multiple operations, which can be costly in some cases.

open (*path*, *mode='rt'*)

Open a file for reading and/or writing.

This method opens the file at the given path for reading and/or writing operations. The object returned is programmatically interchangeable with any other Python file-like object, including specification of file modes. If the file is opened in write mode, changes will only be flushed to the source filesystem when the file is closed.

Parameters

- **path** (*str*) – The path of the file to open.
- **mode** (*str*) – All standard Python file modes.

Returns An opened file-like object.

Return type *FileSystemFile* or file-like

password

Some services require authentication in order to connect to the service, in which case the appropriate password can be specified. If *True* was provided at instantiation, you will be prompted to type your password at runtime when necessary. If *False* was provided, then *None* will be returned. You can specify a different password at runtime using: *duct.password = '<password>'*.

Type str

path_basename (*path*)

Extract the last component of a given path.

Components are determined by splitting by *self.path_separator*. Note that if a path ends with a path separator, the basename will be the empty string.

Parameters **path** (*str*) – The path from which the basename should be extracted.

Returns The extracted basename.

Return type str

path_cwd

The path prefix associated with the current working directory. If not otherwise set, it will be the users' home directory, and will be the prefix used by all non-absolute path references on this filesystem.

Type str

path_dirname (*path*)

Extract the parent directory for provided path.

This method returns the entire path except for the basename (the last component), where components are determined by splitting by *self.path_separator*.

Parameters **path** (*str*) – The path from which the directory path should be extracted.

Returns The extracted directory path.

Return type str

path_home

The path prefix to use as the current users' home directory. Unless *cwd* is set, this will be the prefix to use for all non-absolute path references on this filesystem. This is assumed not to change between connections, and so will not be updated on client reconnections. Unless *global_writes* is set to *True*, this will be the only folder into which this client is permitted to write.

Type str

path_join(*path*, **components*)

Generate a new path by joining together multiple paths.

If any component starts with *self.path_separator* or '~', then all previous path components are discarded, and the effective base path becomes that component (with '~' expanding to *self.path_home*). Note that this method does *not* simplify paths components like '..'. Use *self.path_normpath* for this purpose.

Parameters

- **path** (*str*) – The base path to which components should be joined.
- ***components** (*str*) – Any additional components to join to the base path.

Returns The path resulting from joining all of the components nominated, in order, to the base path.

Return type str

path_normpath(*path*)

Normalise a pathname.

This method returns the normalised (absolute) path corresponding to *path* on this filesystem.

Parameters **path** (*str*) – The path to normalise (make absolute).

Returns The normalised path.

Return type str

path_separator

The character(s) to use in separating path components. Typically this will be '/'.

Type str

port

The local port for the service. If *self.remote* is not *None*, the port will be port-forwarded from the remote host. To see the port used on the remote host refer to *duct._port*. You can change the remote port at runtime using: *duct.port = <port>*.

Type int

prepare()

Prepare a Duct subclass for use (if not already prepared).

This method is called before the value of any of the fields referenced in *self.connection_fields* are retrieved. The fields include, by default: 'host', 'port', 'remote', 'cache', 'username', and 'password'. Subclasses may add or subtract from these special fields.

When called, it first checks whether the instance has already been prepared, and if not calls *_prepare* and then records that the instance has been successfully prepared.

S3Client Quirks: This method may be overridden by subclasses, but provides the following default behaviour:

- Ensures *self.registry*, *self.remote* and *self.cache* values are instances of the right types.

- It replaces string values of *self.remote* and *self.cache* with remotes and caches looked up using *self.registry.lookup*.
- It looks through each of the fields nominated in *self.prepared_fields* and, if the corresponding value is callable, sets the value of that field to result of calling that value with a reference to *self*. By default, *prepared_fields* contains '_host', '_port', '_username', and '_password'.
- Ensures value of self.port is an integer (or None).

`read_only`
Whether this filesystem client should be permitted to attempt any write operations.

Type bool

`reconnect``()`
Disconnects, and then reconnects, this client.

Note: This is equivalent to *duct.disconnect().connect()*.

Returns A reference to this object.

Return type *Duct* instance

`remove``(`*path*, *recursive=False*`)`
Remove file(s) at a nominated path.

Directories (and their contents) will not be removed unless *recursive* is set to *True*.

Parameters

- **`path`** (*`str`*) – The path of the file/directory to be removed.
- **`recursive`** (*`bool`*) – Whether to remove directories and all of their contents.

`reset``()`
Reset this *Duct* instance to its pre-preparation state.

This method disconnects from the service, resets any temporary authentication and restores the values of the attributes listed in *prepared_fields* to their values as of when *Duct.prepare* was called.

Returns A reference to this object.

Return type *Duct* instance

`showdir``(`*path=None*`)`
Return a dataframe representation of a directory.

This method returns a *pandas.DataFrame* representation of the contents of a path, which are retrieved using *.dir(path)*. The exact columns will vary from filesystem to filesystem, depending on the fields returned by *.dir()*, but the returned DataFrame is guaranteed to at least have the columns: 'name' and 'type'.

Parameters **`path`** (*`str`*) – The path of the directory from which to show contents.

Returns A DataFrame representation of the contents of the nominated directory.

Return type pandas.DataFrame

`upload``(`*source*, *dest=None*, *overwrite=False*, *fs=None*`)`
Upload files from another filesystem.

This method (recursively) uploads a file/folder from path *source* on filesystem *fs* to the path *dest* on this filesytem, overwriting any existing file if *overwrite* is *True*. This is equivalent to *fs.download(. . . , fs=self)*.

Parameters

- **source** (*str*) – The path on the specified filesystem (*fs*) of the file to upload to this filesystem. If *source* ends with '/', and corresponds to a directory, the contents of source will be copied instead of copying the entire folder.
- **dest** (*str*) – The destination path on this filesystem. If not specified, the file/folder is uploaded into the default path, usually one's home folder, on this filesystem. If *dest* ends with '/' then file will be copied into destination folder, and will throw an error if path does not resolve to a directory.
- **overwrite** (*bool*) – *True* if the contents of any existing file by the same name should be overwritten, *False* otherwise.
- **fs** (FileSystemClient) – The FileSystemClient from which to load the file/folder at *source*. If not specified, defaults to the local filesystem.

username

Some services require authentication in order to connect to the service, in which case the appropriate username can be specified. If not specified at instantiation, your local login name will be used. If *True* was provided, you will be prompted to type your username at runtime as necessary. If *False* was provided, then *None* will be returned. You can specify a different username at runtime using: *duct.username = '<username>'*.

Type str

walk (*path=None*)

Explore the filesystem tree starting at a nominated path.

This method returns a generator which recursively walks over all paths that are children of *path*, one result for each directory, of form: (<path name>, [<directory 1>, . . .], [<file 1>, . . .])

Parameters **path** (*str*) – The path of the directory from which to enumerate contents.

Returns A generator of tuples, each tuple being associated with one directory that is either *path* or one of its descendants.

Return type generator<tuple>

WebHdfsClient

class omniduct.filesystems.webhdfs.**WebHdfsClient** (*cwd=None*, *home=None*, *read_only=False*, *global_writes=False*, ***kwargs*)

Bases: *omniduct.filesystems.base.FileSystemClient*

This Duct connects to an Apache WebHDFS server using the *pywebhdfs* library.

Attributes namenodes (*list<str>*) – A list of hosts that are acting as namenodes for the HDFS cluster in form "<hostname>:<port>".

Attributes inherited from Duct:

protocol (str): The name of the protocol for which this instance was created (especially useful if a *Duct* subclass supports multiple protocols).

name (str): The name given to this *Duct* instance (defaults to class name).

host (str): The host name providing the service (will be '127.0.0.1', if service is port forwarded from remote; use *._host* to see remote host).

port (int): The port number of the service (will be the port-forwarded local port, if relevant; for remote port use *._port*).

username (str, bool): The username to use for the service. password (str, bool): The password to use for the service. registry (None, omniduct.registry.DuctRegistry): A reference to a

DuctRegistry instance for runtime lookup of other services.

remote (None, omniduct.remotes.base.RemoteClient): A reference to a *RemoteClient* instance to manage connections to remote services.

cache (None, omniduct.caches.base.Cache): A reference to a *Cache* instance to add support for caching, if applicable.

connection_fields (tuple<str>, list<str>): A list of instance attributes to monitor for changes, whereupon the *Duct* instance should automatically disconnect. By default, the following attributes are monitored: 'host', 'port', 'remote', 'username', and 'password'.

prepared_fields (tuple<str>, list<str>): A list of instance attributes to be populated (if their values are callable) when the instance first connects to a service. Refer to *Duct.prepare* and *Duct._prepare* for more details. By default, the following attributes are prepared: '_host', '_port', '_username', and '_password'.

Additional attributes including *host*, *port*, *username* and *password* are documented inline.

Class Attributes:

AUTO_LOGGING_SCOPE (bool): Whether this class should be used by omniduct logging code as a "scope". Should be overridden by subclasses as appropriate.

DUCT_TYPE (Duct.Type): The type of *Duct* service that is provided by this Duct instance. Should be overridden by subclasses as appropriate.

PROTOCOLS (list<str>): The name(s) of any protocols that should be associated with this class. Should be overridden by subclasses as appropriate.

class Type

Bases: `enum.Enum`

The *Duct.Type* enum specifies all of the permissible values of *Duct.DUCT_TYPE*. Also determines the order in which ducts are loaded by DuctRegistry.

__init__(*cwd=None*, *home=None*, *read_only=False*, *global_writes=False*, ***kwargs*)

protocol (str, None): Name of protocol (used by Duct registries to inform Duct instances of how they were instantiated).

name (str, None): The name to used by the *Duct* instance (defaults to class name if not specified).

registry (DuctRegistry, None): The registry to use to lookup remote and/or cache instance specified by name.

remote (str, RemoteClient): The remote by which the ducted service should be contacted.

host (str): The hostname of the service to be used by this client. port (int): The port of the service to be used by this client. username (str, bool, None): The username to authenticate with if necessary.

If True, then users will be prompted at runtime for credentials.

password (str, bool, None): The password to authenticate with if necessary. If True, then users will be prompted at runtime for credentials.

cache(Cache, None): The cache client to be attached to this instance. Cache will only used by specific methods as configured by the client.

cache_namespace(str, None): The namespace to use by default when writing to the cache.

FileSystemClient Quirks:

cwd (None, str): The path prefix to use as the current working directory (if None, the user's home directory is used where that makes sense).

home (None, str): The path prefix to use as the current users' home directory. If not specified, it will default to an implementation- specific value (often '/').

read_only (bool): Whether the filesystem should only be able to perform read operations.

global_writes (bool): Whether to allow writes outside of the user's home folder.

**kwargs (dict): Additional keyword arguments to passed on to subclasses.

WebHdfsClient Quirks:

namenodes (list<str>): A list of hosts that are acting as namenodes for the HDFS cluster in form "<hostname>:<port>".

auto_conf (bool): Whether to automatically extract host, port and namenode information from Cloudera configuration files. If True, automatically extracted values will override other passed values.

auto_conf_cluster (str): The name of the cluster for which to extract configuration.

auto_conf_path (str): The path of the *hdfs-site.xml* file in which the HDFS configuration is stored (on the remote filesystem if *remote* is specified, and on the local filesystem otherwise). Defaults to '/etc/hadoop/conf.cloudera.hdfs2/hdfs-site.xml'.

**kwargs (dict): Additional arguments to pass onto the WebHdfs client.

connect()

Connect to the service backing this client.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Returns A reference to the current object.

Return type *Duct* instance

dir(*path=None*)

Retrieve information about the children of a nominated directory.

This method returns a generator over *FileSystemFileDesc* objects that represent the files/directories that a present as children of the nominated path. If *path* is not a directory, an exception is raised. The path is interpreted as being relative to the current working directory (on remote filesytems, this will typically be the home folder).

Parameters **path** (*str*) – The path to examine for children.

Returns The children of *path* represented as *FileSystemFileDesc* objects.

Return type generator<FileSystemFileDesc>

disconnect()

Disconnect this client from backing service.

This method is automatically called during reconnections and/or at Python interpreter shutdown. It first calls *Duct._disconnect* (which should be implemented by subclasses) and then notifies the *RemoteClient* subclass, if present, to stop port-forwarding the remote service.

Returns A reference to this object.

Return type *Duct* instance

download(*source*, *dest=None*, *overwrite=False*, *fs=None*)
Download files to another filesystem.

This method (recursively) downloads a file/folder from path *source* on this filesystem to the path *dest* on filesytem *fs*, overwriting any existing file if *overwrite* is *True*.

Parameters

- **source** (*str*) – The path on this filesystem of the file to download to the nominated filesystem (*fs*). If *source* ends with '/' then contents of the the *source* directory will be copied into destination folder, and will throw an error if path does not resolve to a directory.
- **dest** (*str*) – The destination path on filesystem (*fs*). If not specified, the file/folder is downloaded into the default path, usually one's home folder. If *dest* ends with '/', and corresponds to a directory, the contents of source will be copied instead of copying the entire folder. If *dest* is otherwise a directory, an exception will be raised.
- **overwrite** (*bool*) – *True* if the contents of any existing file by the same name should be overwritten, *False* otherwise.
- **fs** (FileSystemClient) – The FileSystemClient into which the nominated file/folder *source* should be downloaded. If not specified, defaults to the local filesystem.

exists(*path*)
Check whether nominated path exists on this filesytem.

Parameters **path** (*str*) – The path for which to check existence.

Returns

True* if file/folder exists at nominated path, and *False otherwise.

Return type bool

find(*path_prefix=None*, ***attrs*)
Find a file or directory based on certain attributes.

This method searches for files or folders which satisfy certain constraints on the attributes of the file (as encoded into *FileSystemFileDesc*). Note that without attribute constraints, this method will function identically to *self.dir*.

Parameters

- **path_prefix** (*str*) – The path under which files/directories should be found.
- ****attrs** (*dict*) – Constraints on the fields of the *FileSystemFileDesc* objects associated with this filesystem, as constant values or callable objects (in which case the object will be called and should return True if attribute value is match, and False otherwise).

Returns

A generator over *FileSystemFileDesc* objects that are descendents of *path_prefix* and which statisfy provided constraints.

Return type generator<FileSystemFileDesc>

classmethod for_protocol(*protocol*)
Retrieve a *Duct* subclass for a given protocol.

Parameters **protocol** (*str*) – The protocol of interest.

Returns

The appropriate class for the provided, partially constructed with the *protocol* keyword argument set appropriately.

Return type functools.partial object

Raises `DuctProtocolUnknown` – If no class has been defined that offers the named protocol.

`global_writes`
Whether writes should be permitted outside of home directory. This write-lock is designed to prevent inadvertent scripted writing in potentially dangerous places.

Type bool

`host`
The host name providing the service, or '127.0.0.1' if *self.remote* is not *None*, whereupon the service will be port-forwarded locally. You can view the remote hostname using *duct._host*, and change the remote host at runtime using: *duct.host = '<host>'*.

Type str

`is_connected`()
Check whether this *Duct* instances is currently connected.

This method checks to see whether a *Duct* instance is currently connected. This is performed by verifying that the remote host and port are still accessible, and then by calling *Duct._is_connected*, which should be implemented by subclasses.

Returns Whether this *Duct* instance is currently connected.

Return type bool

`isdir`(*path*)
Check whether a nominated path is directory.

Parameters **`path`** (*`str`*) – The path for which to check directory nature.

Returns *True* if folder exists at nominated path, and *False* otherwise.

Return type bool

`isfile`(*path*)
Check whether a nominated path is a file.

Parameters **`path`** (*`str`*) – The path for which to check file nature.

Returns *True* if a file exists at nominated path, and *False* otherwise.

Return type bool

`listdir`(*path=None*)
Retrieve the names of the children of a nomianted directory.

This method inspects the contents of a directory using *.dir(path)*, and returns the names of child members as strings. *path* is interpreted relative to the current working directory (on remote filesytems, this will typically be the home folder).

Parameters **`path`** (*`str`*) – The path of the directory from which to enumerate filenames.

Returns The names of all children of the nominated directory.

Return type list<str>

`mkdir`(*path*, *recursive=True*, *exist_ok=False*)
Create a directory at the given path.

Parameters

- **`path`** (*`str`*) – The path of the directory to create.

- **recursive** (*bool*) – Whether to recursively create any parents of this path if they do not already exist.

Note: *exist_ok* is passed onto subclass implementations of *_mkdir* rather that implementing the existence check using *.exists* so that they can avoid the overhead associated with multiple operations, which can be costly in some cases.

open (*path*, *mode='rt'*)
Open a file for reading and/or writing.

This method opens the file at the given path for reading and/or writing operations. The object returned is programmatically interchangeable with any other Python file-like object, including specification of file modes. If the file is opened in write mode, changes will only be flushed to the source filesystem when the file is closed.

Parameters

- **path** (*str*) – The path of the file to open.
- **mode** (*str*) – All standard Python file modes.

Returns An opened file-like object.

Return type *FileSystemFile* or file-like

password
Some services require authentication in order to connect to the service, in which case the appropriate password can be specified. If *True* was provided at instantiation, you will be prompted to type your password at runtime when necessary. If *False* was provided, then *None* will be returned. You can specify a different password at runtime using: *duct.password = '<password>'*.

Type str

path_basename (*path*)
Extract the last component of a given path.

Components are determined by splitting by *self.path_separator*. Note that if a path ends with a path separator, the basename will be the empty string.

Parameters path (*str*) – The path from which the basename should be extracted.

Returns The extracted basename.

Return type str

path_cwd
The path prefix associated with the current working directory. If not otherwise set, it will be the users' home directory, and will be the prefix used by all non-absolute path references on this filesystem.

Type str

path_dirname (*path*)
Extract the parent directory for provided path.

This method returns the entire path except for the basename (the last component), where components are determined by splitting by *self.path_separator*.

Parameters path (*str*) – The path from which the directory path should be extracted.

Returns The extracted directory path.

Return type str

path_home
The path prefix to use as the current users' home directory. Unless *cwd* is set, this will be the prefix to use

for all non-absolute path references on this filesystem. This is assumed not to change between connections, and so will not be updated on client reconnections. Unless *global_writes* is set to *True*, this will be the only folder into which this client is permitted to write.

Type str

path_join (*path*, **components*)

Generate a new path by joining together multiple paths.

If any component starts with *self.path_separator* or '~', then all previous path components are discarded, and the effective base path becomes that component (with '~' expanding to *self.path_home*). Note that this method does *not* simplify paths components like '..'. Use *self.path_normpath* for this purpose.

Parameters

- **path** (*str*) – The base path to which components should be joined.
- ***components** (*str*) – Any additional components to join to the base path.

Returns The path resulting from joining all of the components nominated, in order, to the base path.

Return type str

path_normpath (*path*)

Normalise a pathname.

This method returns the normalised (absolute) path corresponding to *path* on this filesystem.

Parameters **path** (*str*) – The path to normalise (make absolute).

Returns The normalised path.

Return type str

path_separator

The character(s) to use in separating path components. Typically this will be '/'.

Type str

port

The local port for the service. If *self.remote* is not *None*, the port will be port-forwarded from the remote host. To see the port used on the remote host refer to *duct._port*. You can change the remote port at runtime using: *duct.port = <port>*.

Type int

prepare ()

Prepare a Duct subclass for use (if not already prepared).

This method is called before the value of any of the fields referenced in *self.connection_fields* are retrieved. The fields include, by default: 'host', 'port', 'remote', 'cache', 'username', and 'password'. Subclasses may add or subtract from these special fields.

When called, it first checks whether the instance has already been prepared, and if not calls *_prepare* and then records that the instance has been successfully prepared.

WebHdfsClient Quirks: This method may be overridden by subclasses, but provides the following default behaviour:

- Ensures *self.registry*, *self.remote* and *self.cache* values are instances of the right types.
- It replaces string values of *self.remote* and *self.cache* with remotes and caches looked up using *self.registry.lookup*.

- It looks through each of the fields nominated in *self.prepared_fields* and, if the corresponding value is callable, sets the value of that field to result of calling that value with a reference to *self*. By default, *prepared_fields* contains '_host', '_port', '_username', and '_password'.
- Ensures value of self.port is an integer (or None).

`read_only`
Whether this filesystem client should be permitted to attempt any write operations.

Type bool

`reconnect`()
Disconnects, and then reconnects, this client.

Note: This is equivalent to *duct.disconnect().connect()*.

Returns A reference to this object.

Return type *Duct* instance

`remove`(*path*, *recursive=False*)
Remove file(s) at a nominated path.

Directories (and their contents) will not be removed unless *recursive* is set to *True*.

Parameters

- **`path`** (*`str`*) – The path of the file/directory to be removed.
- **`recursive`** (*`bool`*) – Whether to remove directories and all of their contents.

`reset`()
Reset this *Duct* instance to its pre-preparation state.

This method disconnects from the service, resets any temporary authentication and restores the values of the attributes listed in *prepared_fields* to their values as of when *Duct.prepare* was called.

Returns A reference to this object.

Return type *Duct* instance

`showdir`(*path=None*)
Return a dataframe representation of a directory.

This method returns a *pandas.DataFrame* representation of the contents of a path, which are retrieved using *.dir(path)*. The exact columns will vary from filesystem to filesystem, depending on the fields returned by *.dir()*, but the returned DataFrame is guaranteed to at least have the columns: 'name' and 'type'.

Parameters **`path`** (*`str`*) – The path of the directory from which to show contents.

Returns A DataFrame representation of the contents of the nominated directory.

Return type pandas.DataFrame

`upload`(*source*, *dest=None*, *overwrite=False*, *fs=None*)
Upload files from another filesystem.

This method (recursively) uploads a file/folder from path *source* on filesystem *fs* to the path *dest* on this filesytem, overwriting any existing file if *overwrite* is *True*. This is equivalent to *fs.download(…, fs=self)*.

Parameters

- **`source`** (*`str`*) – The path on the specified filesystem (*fs*) of the file to upload to this filesystem. If *source* ends with '/', and corresponds to a directory, the contents of source will be copied instead of copying the entire folder.

- **dest** (*str*) – The destination path on this filesystem. If not specified, the file/folder is uploaded into the default path, usually one's home folder, on this filesystem. If *dest* ends with '/' then file will be copied into destination folder, and will throw an error if path does not resolve to a directory.
- **overwrite** (*bool*) – *True* if the contents of any existing file by the same name should be overwritten, *False* otherwise.
- **fs** (`FileSystemClient`) – The FileSystemClient from which to load the file/folder at *source*. If not specified, defaults to the local filesystem.

username
Some services require authentication in order to connect to the service, in which case the appropriate username can be specified. If not specified at instantiation, your local login name will be used. If *True* was provided, you will be prompted to type your username at runtime as necessary. If *False* was provided, then *None* will be returned. You can specify a different username at runtime using: *duct.username = '<username>'*.

Type str

walk (*path=None*)
Explore the filesystem tree starting at a nominated path.

This method returns a generator which recursively walks over all paths that are children of *path*, one result for each directory, of form: (<path name>, [<directory 1>, . . .], [<file 1>, . . .])

Parameters **path** (*str*) – The path of the directory from which to enumerate contents.

Returns A generator of tuples, each tuple being associated with one directory that is either *path* or one of its descendants.

Return type generator<tuple>

5.4 Remotes

All remote clients are expected to be subclasses of *RemoteClient*, and so will share a common API. Protocol implementations are also free to add extra methods, which are documented in the "Subclass Reference" section below.

5.4.1 Common API

class omniduct.remotes.base.**RemoteClient** (*smartcards=None*, ***kwargs*)
Bases: *omniduct.filesystems.base.FileSystemClient*

An abstract class providing the common API for all remote clients.

Attributes **smartcard** (*dict*) – Mapping of smartcard names to system libraries compatible with *ssh-add -s '<system library>'* . . . .

Attributes **smartcard** (*dict*) – Mapping of smartcard names to system libraries compatible with *ssh-add -s '<system library>'* . . . .

Attributes inherited from Duct:

protocol (str): The name of the protocol for which this instance was created (especially useful if a *Duct* subclass supports multiple protocols).

name (str): The name given to this *Duct* instance (defaults to class name).

host (str): The host name providing the service (will be '127.0.0.1', if service is port forwarded from remote; use *._host* to see remote host).

port (int): The port number of the service (will be the port-forwarded local port, if relevant; for remote port use *._port*).

username (str, bool): The username to use for the service. password (str, bool): The password to use for the service. registry (None, omniduct.registry.DuctRegistry): A reference to a

DuctRegistry instance for runtime lookup of other services.

remote (None, omniduct.remotes.base.RemoteClient): A reference to a *RemoteClient* instance to manage connections to remote services.

cache (None, omniduct.caches.base.Cache): A reference to a *Cache* instance to add support for caching, if applicable.

connection_fields (tuple<str>, list<str>): A list of instance attributes to monitor for changes, whereupon the *Duct* instance should automatically disconnect. By default, the following attributes are monitored: 'host', 'port', 'remote', 'username', and 'password'.

prepared_fields (tuple<str>, list<str>): A list of instance attributes to be populated (if their values are callable) when the instance first connects to a service. Refer to *Duct.prepare* and *Duct._prepare* for more details. By default, the following attributes are prepared: '_host', '_port', '_username', and '_password'.

Additional attributes including *host*, *port*, *username* and *password* are documented inline.

Class Attributes:

AUTO_LOGGING_SCOPE (bool): Whether this class should be used by omniduct logging code as a "scope". Should be overridden by subclasses as appropriate.

DUCT_TYPE (Duct.Type): The type of *Duct* service that is provided by this Duct instance. Should be overridden by subclasses as appropriate.

PROTOCOLS (list<str>): The name(s) of any protocols that should be associated with this class. Should be overridden by subclasses as appropriate.

`__init__`(*smartcards=None*, ***kwargs*)

protocol (str, None): Name of protocol (used by Duct registries to inform Duct instances of how they were instantiated).

name (str, None): The name to used by the *Duct* instance (defaults to class name if not specified).

registry (DuctRegistry, None): The registry to use to lookup remote and/or cache instance specified by name.

remote (str, RemoteClient): The remote by which the ducted service should be contacted.

host (str): The hostname of the service to be used by this client. port (int): The port of the service to be used by this client. username (str, bool, None): The username to authenticate with if necessary.

If True, then users will be prompted at runtime for credentials.

password (str, bool, None): The password to authenticate with if necessary. If True, then users will be prompted at runtime for credentials.

cache(Cache, None): The cache client to be attached to this instance. Cache will only used by specific methods as configured by the client.

cache_namespace(str, None): The namespace to use by default when writing to the cache.

FileSystemClient Quirks:

cwd (None, str): The path prefix to use as the current working directory (if None, the user's home directory is used where that makes sense).

home (None, str): The path prefix to use as the current users' home directory. If not specified, it will default to an implementation- specific value (often '/').

read_only (bool): Whether the filesystem should only be able to perform read operations.

global_writes (bool): Whether to allow writes outside of the user's home folder.

**kwargs (dict): Additional keyword arguments to passed on to subclasses.

RemoteClient Quirks:

Args:

smartcards (dict): Mapping of smartcard names to system libraries compatible with *ssh-add -s '<system library>' ...*.

connect()

Connect to the service backing this client.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Returns A reference to the current object.

Return type *Duct* instance

RemoteClient Quirks: Connect to the remote server.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Compared to base *Duct.connect*, this method will automatically catch the first *DuctAuthenticationError* error triggered by *Duct.connect*, and (if smartcards have been configured) attempt to re-initialise the smartcards before trying once more.

Returns: *Duct* instance: A reference to the current object.

prepare_smartcards()

Prepare smartcards for use in authentication.

This method checks attempts to ensure that the each of the nominated smartcards is available and prepared for use. This may result in interactive requests for pin confirmation, depending on the card.

Returns

Returns *True* if at least one smartcard was activated, and *False* otherwise.

Return type bool

execute(*cmd*, ***kwargs*)

Execute a command on the remote server.

Parameters

- **cmd** (`str`) – The command to run on the remote associated with this instance.
- ****kwargs** (`dict`) – Additional keyword arguments to be passed on to *._execute*.

Returns The result of the execution.

Return type SubprocessResults

port_forward(*remote_host*, *remote_port=None*, *local_port=None*)
Initiate a port forward connection.

This method establishes a local port forwarding from a local port *local* to remote port *remote*. If *local* is *None*, an available local port is automatically chosen. If the remote port is already forwarded, a new connection is not established.

Parameters

- **remote_host** (*str*) – The hostname of the remote host in form: 'hostname(:port)'.
- **remote_port** (*int, None*) – The remote port of the service.
- **local_port** (*int, None*) – The port to use locally (automatically determined if not specified).

Returns The local port which is port forwarded to the remote service.

Return type int

has_port_forward(*remote_host=None*, *remote_port=None*, *local_port=None*)
Check whether a port forward connection exists.

Parameters

- **remote_host** (*str*) – The hostname of the remote host in form: 'hostname(:port)'.
- **remote_port** (*int, None*) – The remote port of the service.
- **local_port** (*int, None*) – The port used locally.

Returns

Whether a port-forward for this remote service exists, or if local port is specified, whether that port is locally used for port forwarding.

Return type bool

port_forward_stop(*local_port=None*, *remote_host=None*, *remote_port=None*)
Disconnect an existing port forward connection.

If a local port is provided, then the forwarding (if any) associated with that port is found and stopped; otherwise any established port forwarding associated with the nominated remote service is stopped.

Parameters

- **remote_host** (*str*) – The hostname of the remote host in form: 'hostname(:port)'.
- **remote_port** (*int, None*) – The remote port of the service.
- **local_port** (*int, None*) – The port used locally.

port_forward_stopall()
Disconnect all existing port forwarding connections.

get_local_uri(*uri*)
Convert a remote uri to a local one.

This method takes a remote service uri accessible to the remote host and returns a local uri accessible directly on the local host, establishing any necessary port forwarding in the process.

Parameters **uri** (*str*) – The remote uri to be made local.

Returns A local uri that tunnels all traffic to the remote host.

Return type str

show_port_forwards()
Print to stdout the active port forwards associated with this client.

is_port_bound(*host*, *port*)
Check whether a port on a remote host is accessible.

This method checks to see whether a particular port is active on a given host by attempting to establish a connection with it.

Parameters

- **host** (*str*) – The hostname of the target service.
- **port** (*int*) – The port of the target service.

Returns Whether the port is active and accepting connections.

Return type bool

dir(*path=None*)
Retrieve information about the children of a nominated directory.

This method returns a generator over *FileSystemFileDesc* objects that represent the files/directories that a present as children of the nominated path. If *path* is not a directory, an exception is raised. The path is interpreted as being relative to the current working directory (on remote filesytems, this will typically be the home folder).

Parameters **path** (*str*) – The path to examine for children.

Returns The children of *path* represented as *FileSystemFileDesc* objects.

Return type generator<FileSystemFileDesc>

RemoteClient Quirks: This method should return a generator over *FileSystemFileDesc* objects.

disconnect()
Disconnect this client from backing service.

This method is automatically called during reconnections and/or at Python interpreter shutdown. It first calls *Duct._disconnect* (which should be implemented by subclasses) and then notifies the *RemoteClient* subclass, if present, to stop port-forwarding the remote service.

Returns A reference to this object.

Return type *Duct* instance

download(*source*, *dest=None*, *overwrite=False*, *fs=None*)
Download files to another filesystem.

This method (recursively) downloads a file/folder from path *source* on this filesystem to the path *dest* on filesytem *fs*, overwriting any existing file if *overwrite* is *True*.

Parameters

- **source** (*str*) – The path on this filesystem of the file to download to the nominated filesystem (*fs*). If *source* ends with '/' then contents of the the *source* directory will be copied into destination folder, and will throw an error if path does not resolve to a directory.
- **dest** (*str*) – The destination path on filesystem (*fs*). If not specified, the file/folder is downloaded into the default path, usually one's home folder. If *dest* ends with '/', and corresponds to a directory, the contents of source will be copied instead of copying the entire folder. If *dest* is otherwise a directory, an exception will be raised.

- **overwrite** (*bool*) – *True* if the contents of any existing file by the same name should be overwritten, *False* otherwise.
- **fs** (FileSystemClient) – The FileSystemClient into which the nominated file/folder *source* should be downloaded. If not specified, defaults to the local filesystem.

exists(*path*)

Check whether nominated path exists on this filesytem.

Parameters **path** (*str*) – The path for which to check existence.

Returns

True **if file/folder exists at nominated path, and** ***False*** otherwise.

Return type bool

find(*path_prefix=None*, ***attrs*)

Find a file or directory based on certain attributes.

This method searches for files or folders which satisfy certain constraints on the attributes of the file (as encoded into *FileSystemFileDesc*). Note that without attribute constraints, this method will function identically to *self.dir*.

Parameters

- **path_prefix** (*str*) – The path under which files/directories should be found.
- ****attrs** (*dict*) – Constraints on the fields of the *FileSystemFileDesc* objects associated with this filesystem, as constant values or callable objects (in which case the object will be called and should return True if attribute value is match, and False otherwise).

Returns

A generator over ***FileSystemFileDesc*** objects that are descendents of *path_prefix* and which statisfy provided constraints.

Return type generator<FileSystemFileDesc>

is_connected()

Check whether this *Duct* instances is currently connected.

This method checks to see whether a *Duct* instance is currently connected. This is performed by verifying that the remote host and port are still accessible, and then by calling *Duct._is_connected*, which should be implemented by subclasses.

Returns Whether this *Duct* instance is currently connected.

Return type bool

isdir(*path*)

Check whether a nominated path is directory.

Parameters **path** (*str*) – The path for which to check directory nature.

Returns *True* if folder exists at nominated path, and *False* otherwise.

Return type bool

isfile(*path*)

Check whether a nominated path is a file.

Parameters **path** (*str*) – The path for which to check file nature.

Returns *True* if a file exists at nominated path, and *False* otherwise.

Return type bool

mkdir (*path*, *recursive=True*, *exist_ok=False*)
Create a directory at the given path.

Parameters

- **path** (*str*) – The path of the directory to create.
- **recursive** (*bool*) – Whether to recursively create any parents of this path if they do not already exist.

Note: *exist_ok* is passed onto subclass implementations of *_mkdir* rather that implementing the existence check using *.exists* so that they can avoid the overhead associated with multiple operations, which can be costly in some cases.

open (*path*, *mode='rt'*)
Open a file for reading and/or writing.

This method opens the file at the given path for reading and/or writing operations. The object returned is programmatically interchangeable with any other Python file-like object, including specification of file modes. If the file is opened in write mode, changes will only be flushed to the source filesystem when the file is closed.

Parameters

- **path** (*str*) – The path of the file to open.
- **mode** (*str*) – All standard Python file modes.

Returns An opened file-like object.

Return type *FileSystemFile* or file-like

path_home
The path prefix to use as the current users' home directory. Unless *cwd* is set, this will be the prefix to use for all non-absolute path references on this filesystem. This is assumed not to change between connections, and so will not be updated on client reconnections. Unless *global_writes* is set to *True*, this will be the only folder into which this client is permitted to write.

Type str

path_separator
The character(s) to use in separating path components. Typically this will be '/'.

Type str

prepare ()
Prepare a Duct subclass for use (if not already prepared).

This method is called before the value of any of the fields referenced in *self.connection_fields* are retrieved. The fields include, by default: 'host', 'port', 'remote', 'cache', 'username', and 'password'. Subclasses may add or subtract from these special fields.

When called, it first checks whether the instance has already been prepared, and if not calls *_prepare* and then records that the instance has been successfully prepared.

RemoteClient Quirks: This method may be overridden by subclasses, but provides the following default behaviour:

- Ensures *self.registry*, *self.remote* and *self.cache* values are instances of the right types.
- It replaces string values of *self.remote* and *self.cache* with remotes and caches looked up using *self.registry.lookup*.

- It looks through each of the fields nominated in *self.prepared_fields* and, if the corresponding value is callable, sets the value of that field to result of calling that value with a reference to *self*. By default, *prepared_fields* contains '_host', '_port', '_username', and '_password'.
- Ensures value of self.port is an integer (or None).

remove(*path*, *recursive=False*)

Remove file(s) at a nominated path.

Directories (and their contents) will not be removed unless *recursive* is set to *True*.

Parameters

- **path** (*str*) – The path of the file/directory to be removed.
- **recursive** (*bool*) – Whether to remove directories and all of their contents.

walk(*path=None*)

Explore the filesystem tree starting at a nominated path.

This method returns a generator which recursively walks over all paths that are children of *path*, one result for each directory, of form: (<path name>, [<directory 1>, ...], [<file 1>, ...])

Parameters **path** (*str*) – The path of the directory from which to enumerate contents.

Returns A generator of tuples, each tuple being associated with one directory that is either *path* or one of its descendants.

Return type generator<tuple>

5.4.2 Subclass Reference

For comprehensive documentation on any particular subclass, please refer to one of the below documents.

SSHClient

class omniduct.remotes.ssh.**SSHClient**(*smartcards=None*, ***kwargs*)

Bases: *omniduct.remotes.base.RemoteClient*

An implementation of the *RemoteClient Duct*, offering a persistent connection to remote hosts over SSH via the CLI. As such, it requires that *ssh* be installed and on your executable path.

To speed up connections we use control sockets, which allows all connections to share one SSH transport. For more details, refer to: https://puppetlabs.com/blog/speed-up-ssh-by-reusing-connections

Attributes interactive (*bool*) – Whether *SSHClient* should ask the user questions, if necessary, to establish the connection. Production deployments using this client should set this to False. (default: *False*)

Attributes inherited from RemoteClient:

smartcard (dict): Mapping of smartcard names to system libraries compatible with *ssh-add -s '<system library>'* ....

Attributes inherited from Duct:

protocol (str): The name of the protocol for which this instance was created (especially useful if a *Duct* subclass supports multiple protocols).

name (str): The name given to this *Duct* instance (defaults to class name).

host (str): The host name providing the service (will be '127.0.0.1', if service is port forwarded from remote; use *._host* to see remote host).

port (int): The port number of the service (will be the port-forwarded local port, if relevant; for remote port use *._port*).

username (str, bool): The username to use for the service. password (str, bool): The password to use for the service. registry (None, omniduct.registry.DuctRegistry): A reference to a

DuctRegistry instance for runtime lookup of other services.

remote (None, omniduct.remotes.base.RemoteClient): A reference to a *RemoteClient* instance to manage connections to remote services.

cache (None, omniduct.caches.base.Cache): A reference to a *Cache* instance to add support for caching, if applicable.

connection_fields (tuple<str>, list<str>): A list of instance attributes to monitor for changes, whereupon the *Duct* instance should automatically disconnect. By default, the following attributes are monitored: 'host', 'port', 'remote', 'username', and 'password'.

prepared_fields (tuple<str>, list<str>): A list of instance attributes to be populated (if their values are callable) when the instance first connects to a service. Refer to *Duct.prepare* and *Duct._prepare* for more details. By default, the following attributes are prepared: '_host', '_port', '_username', and '_password'.

Additional attributes including *host*, *port*, *username* and *password* are documented inline.

Class Attributes:

AUTO_LOGGING_SCOPE (bool): Whether this class should be used by omniduct logging code as a "scope". Should be overridden by subclasses as appropriate.

DUCT_TYPE (Duct.Type): The type of *Duct* service that is provided by this Duct instance. Should be overridden by subclasses as appropriate.

PROTOCOLS (list<str>): The name(s) of any protocols that should be associated with this class. Should be overridden by subclasses as appropriate.

class Type

Bases: `enum.Enum`

The *Duct.Type* enum specifies all of the permissible values of *Duct.DUCT_TYPE*. Also determines the order in which ducts are loaded by DuctRegistry.

__init__(*smartcards=None*, ***kwargs*)

protocol (str, None): Name of protocol (used by Duct registries to inform Duct instances of how they were instantiated).

name (str, None): The name to used by the *Duct* instance (defaults to class name if not specified).

registry (DuctRegistry, None): The registry to use to lookup remote and/or cache instance specified by name.

remote (str, RemoteClient): The remote by which the ducted service should be contacted.

host (str): The hostname of the service to be used by this client. port (int): The port of the service to be used by this client. username (str, bool, None): The username to authenticate with if necessary.

If True, then users will be prompted at runtime for credentials.

password (str, bool, None): The password to authenticate with if necessary. If True, then users will be prompted at runtime for credentials.

cache(Cache, None): The cache client to be attached to this instance. Cache will only used by specific methods as configured by the client.

cache_namespace(str, None): The namespace to use by default when writing to the cache.

FileSystemClient Quirks:

cwd (None, str): The path prefix to use as the current working directory (if None, the user's home directory is used where that makes sense).

home (None, str): The path prefix to use as the current users' home directory. If not specified, it will default to an implementation- specific value (often '/').

read_only (bool): Whether the filesystem should only be able to perform read operations.

global_writes (bool): Whether to allow writes outside of the user's home folder.

**kwargs (dict): Additional keyword arguments to passed on to subclasses.

RemoteClient Quirks:

Args:

smartcards (dict): Mapping of smartcard names to system libraries compatible with *ssh-add -s '<system library>'* ....

SSHClient Quirks:

interactive (bool): Whether *SSHClient* should ask the user questions, if necessary, to establish the connection. Production deployments using this client should set this to False, which is the default.

check_known_hosts (bool): Whether *SSHClient* should check the known hosts file when establishing the connection. This option should only be set to False in trusted environments.

`connect()`

Connect to the service backing this client.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Returns A reference to the current object.

Return type *Duct* instance

RemoteClient Quirks: Connect to the remote server.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Compared to base *Duct.connect*, this method will automatically catch the first *DuctAuthenticationError* error triggered by *Duct.connect*, and (if smartcards have been configured) attempt to re-initialise the smartcards before trying once more.

Returns: *Duct* instance: A reference to the current object.

`dir`(*path=None*)

Retrieve information about the children of a nominated directory.

This method returns a generator over *FileSystemFileDesc* objects that represent the files/directories that a present as children of the nominated path. If *path* is not a directory, an exception is raised. The path is interpreted as being relative to the current working directory (on remote filesytems, this will typically be the home folder).

Parameters **path** (*str*) – The path to examine for children.

Returns The children of *path* represented as *FileSystemFileDesc* objects.

Return type generator<FileSystemFileDesc>

disconnect()

Disconnect this client from backing service.

This method is automatically called during reconnections and/or at Python interpreter shutdown. It first calls *Duct._disconnect* (which should be implemented by subclasses) and then notifies the *RemoteClient* subclass, if present, to stop port-forwarding the remote service.

Returns A reference to this object.

Return type *Duct* instance

download(*source*, *dest=None*, *overwrite=False*, *fs=None*)

Download files to another filesystem.

This method (recursively) downloads a file/folder from path *source* on this filesystem to the path *dest* on filesytem *fs*, overwriting any existing file if *overwrite* is *True*.

Parameters

- **source** (*str*) – The path on this filesystem of the file to download to the nominated filesystem (*fs*). If *source* ends with '/' then contents of the the *source* directory will be copied into destination folder, and will throw an error if path does not resolve to a directory.
- **dest** (*str*) – The destination path on filesystem (*fs*). If not specified, the file/folder is downloaded into the default path, usually one's home folder. If *dest* ends with '/', and corresponds to a directory, the contents of source will be copied instead of copying the entire folder. If *dest* is otherwise a directory, an exception will be raised.
- **overwrite** (*bool*) – *True* if the contents of any existing file by the same name should be overwritten, *False* otherwise.
- **fs** (FileSystemClient) – The FileSystemClient into which the nominated file/folder *source* should be downloaded. If not specified, defaults to the local filesystem.

SSHClient Quirks: Download files to another filesystem.

This method (recursively) downloads a file/folder from path *source* on this filesystem to the path *dest* on filesytem *fs*, overwriting any existing file if *overwrite* is *True*.

Args:

source (str): The path on this filesystem of the file to download to the nominated filesystem (*fs*). If *source* ends with '/' then contents of the the *source* directory will be copied into destination folder, and will throw an error if path does not resolve to a directory.

dest (str): The destination path on filesystem (*fs*). If not specified, the file/folder is uploaded into the default path, usually one's home folder. If *dest* ends with '/', and corresponds to a directory, the contents of source will be copied instead of copying the entire folder. If *dest* is otherwise a directory, an exception will be raised.

overwrite (bool): *True* if the contents of any existing file by the same name should be overwritten, *False* otherwise.

fs (FileSystemClient): The FileSystemClient into which the nominated file/folder *source* should be downloaded. If not specified, defaults to the local filesystem.

SSHClient Quirks: This method is overloaded so that remote-to-local downloads can be handled specially using *scp*. Downloads to any non-local filesystem are handled using the standard implementation.

`execute`(*cmd*, ***kwargs*)
Execute a command on the remote server.

Parameters

- **`cmd`** (*`str`*) – The command to run on the remote associated with this instance.
- **`**kwargs`** (*`dict`*) – Additional keyword arguments to be passed on to *._execute*.

Returns The result of the execution.

Return type SubprocessResults

SSHClient Quirks:

Additional Args:

skip_cwd (bool): Whether to skip changing to the current working directory associated with this client before executing the command. This is mainly useful to methods internal to this class.

`exists`(*path*)
Check whether nominated path exists on this filesytem.

Parameters **`path`** (*`str`*) – The path for which to check existence.

Returns

True **if file/folder exists at nominated path, and *False*** otherwise.

Return type bool

`find`(*path_prefix=None*, ***attrs*)
Find a file or directory based on certain attributes.

This method searches for files or folders which satisfy certain constraints on the attributes of the file (as encoded into *FileSystemFileDesc*). Note that without attribute constraints, this method will function identically to *self.dir*.

Parameters

- **`path_prefix`** (*`str`*) – The path under which files/directories should be found.
- **`**attrs`** (*`dict`*) – Constraints on the fields of the *FileSystemFileDesc* objects associated with this filesystem, as constant values or callable objects (in which case the object will be called and should return True if attribute value is match, and False otherwise).

Returns

A generator over *FileSystemFileDesc* objects that are descendents of *path_prefix* and which statisfy provided constraints.

Return type generator<FileSystemFileDesc>

`classmethod for_protocol`(*protocol*)
Retrieve a *Duct* subclass for a given protocol.

Parameters **`protocol`** (*`str`*) – The protocol of interest.

Returns

The appropriate class for the provided, partially constructed with the *protocol* keyword argument set appropriately.

Return type functools.partial object

Raises `DuctProtocolUnknown` – If no class has been defined that offers the named protocol.

get_local_uri(*uri*)
Convert a remote uri to a local one.

This method takes a remote service uri accessible to the remote host and returns a local uri accessible directly on the local host, establishing any necessary port forwarding in the process.

Parameters **uri** (*str*) – The remote uri to be made local.

Returns A local uri that tunnels all traffic to the remote host.

Return type str

global_writes
Whether writes should be permitted outside of home directory. This write-lock is designed to prevent inadvertent scripted writing in potentially dangerous places.

Type bool

has_port_forward(*remote_host=None*, *remote_port=None*, *local_port=None*)
Check whether a port forward connection exists.

Parameters

- **remote_host** (*str*) – The hostname of the remote host in form: 'hostname(:port)'.
- **remote_port** (*int, None*) – The remote port of the service.
- **local_port** (*int, None*) – The port used locally.

Returns

Whether a port-forward for this remote service exists, or if local port is specified, whether that port is locally used for port forwarding.

Return type bool

host
The host name providing the service, or '127.0.0.1' if *self.remote* is not *None*, whereupon the service will be port-forwarded locally. You can view the remote hostname using *duct._host*, and change the remote host at runtime using: *duct.host = '<host>'*.

Type str

is_connected()
Check whether this *Duct* instances is currently connected.

This method checks to see whether a *Duct* instance is currently connected. This is performed by verifying that the remote host and port are still accessible, and then by calling *Duct._is_connected*, which should be implemented by subclasses.

Returns Whether this *Duct* instance is currently connected.

Return type bool

is_port_bound(*host*, *port*)
Check whether a port on a remote host is accessible.

This method checks to see whether a particular port is active on a given host by attempting to establish a connection with it.

Parameters

- **host** (*str*) – The hostname of the target service.
- **port** (*int*) – The port of the target service.

Returns Whether the port is active and accepting connections.

Return type bool

isdir(*path*)

Check whether a nominated path is directory.

Parameters **path** (*str*) – The path for which to check directory nature.

Returns *True* if folder exists at nominated path, and *False* otherwise.

Return type bool

isfile(*path*)

Check whether a nominated path is a file.

Parameters **path** (*str*) – The path for which to check file nature.

Returns *True* if a file exists at nominated path, and *False* otherwise.

Return type bool

listdir(*path=None*)

Retrieve the names of the children of a nomianted directory.

This method inspects the contents of a directory using *.dir(path)*, and returns the names of child members as strings. *path* is interpreted relative to the current working directory (on remote filesytems, this will typically be the home folder).

Parameters **path** (*str*) – The path of the directory from which to enumerate filenames.

Returns The names of all children of the nominated directory.

Return type list<str>

mkdir(*path*, *recursive=True*, *exist_ok=False*)

Create a directory at the given path.

Parameters

- **path** (*str*) – The path of the directory to create.
- **recursive** (*bool*) – Whether to recursively create any parents of this path if they do not already exist.

Note: *exist_ok* is passed onto subclass implementations of *_mkdir* rather that implementing the existence check using *.exists* so that they can avoid the overhead associated with multiple operations, which can be costly in some cases.

open(*path*, *mode='rt'*)

Open a file for reading and/or writing.

This method opens the file at the given path for reading and/or writing operations. The object returned is programmatically interchangeable with any other Python file-like object, including specification of file modes. If the file is opened in write mode, changes will only be flushed to the source filesystem when the file is closed.

Parameters

- **path** (*str*) – The path of the file to open.

- **mode** (*str*) – All standard Python file modes.

Returns An opened file-like object.

Return type *FileSystemFile* or file-like

password

Some services require authentication in order to connect to the service, in which case the appropriate password can be specified. If *True* was provided at instantiation, you will be prompted to type your password at runtime when necessary. If *False* was provided, then *None* will be returned. You can specify a different password at runtime using: *duct.password = '<password>'*.

Type str

path_basename (*path*)

Extract the last component of a given path.

Components are determined by splitting by *self.path_separator*. Note that if a path ends with a path separator, the basename will be the empty string.

Parameters path (*str*) – The path from which the basename should be extracted.

Returns The extracted basename.

Return type str

path_cwd

The path prefix associated with the current working directory. If not otherwise set, it will be the users' home directory, and will be the prefix used by all non-absolute path references on this filesystem.

Type str

path_dirname (*path*)

Extract the parent directory for provided path.

This method returns the entire path except for the basename (the last component), where components are determined by splitting by *self.path_separator*.

Parameters path (*str*) – The path from which the directory path should be extracted.

Returns The extracted directory path.

Return type str

path_home

The path prefix to use as the current users' home directory. Unless *cwd* is set, this will be the prefix to use for all non-absolute path references on this filesystem. This is assumed not to change between connections, and so will not be updated on client reconnections. Unless *global_writes* is set to *True*, this will be the only folder into which this client is permitted to write.

Type str

path_join (*path*, **components*)

Generate a new path by joining together multiple paths.

If any component starts with *self.path_separator* or '~', then all previous path components are discarded, and the effective base path becomes that component (with '~' expanding to *self.path_home*). Note that this method does *not* simplify paths components like '..'. Use *self.path_normpath* for this purpose.

Parameters

- **path** (*str*) – The base path to which components should be joined.
- ***components** (*str*) – Any additional components to join to the base path.

Returns The path resulting from joining all of the components nominated, in order, to the base path.

Return type str

path_normpath(*path*)
Normalise a pathname.

This method returns the normalised (absolute) path corresponding to *path* on this filesystem.

Parameters **path** (*str*) – The path to normalise (make absolute).

Returns The normalised path.

Return type str

path_separator
The character(s) to use in separating path components. Typically this will be '/'.

Type str

port
The local port for the service. If *self.remote* is not *None*, the port will be port-forwarded from the remote host. To see the port used on the remote host refer to *duct._port*. You can change the remote port at runtime using: *duct.port = <port>*.

Type int

port_forward(*remote_host*, *remote_port=None*, *local_port=None*)
Initiate a port forward connection.

This method establishes a local port forwarding from a local port *local* to remote port *remote*. If *local* is *None*, an available local port is automatically chosen. If the remote port is already forwarded, a new connection is not established.

Parameters

- **remote_host** (*str*) – The hostname of the remote host in form: 'hostname(:port)'.
- **remote_port** (*int, None*) – The remote port of the service.
- **local_port** (*int, None*) – The port to use locally (automatically determined if not specified).

Returns The local port which is port forwarded to the remote service.

Return type int

port_forward_stop(*local_port=None*, *remote_host=None*, *remote_port=None*)
Disconnect an existing port forward connection.

If a local port is provided, then the forwarding (if any) associated with that port is found and stopped; otherwise any established port forwarding associated with the nominated remote service is stopped.

Parameters

- **remote_host** (*str*) – The hostname of the remote host in form: 'hostname(:port)'.
- **remote_port** (*int, None*) – The remote port of the service.
- **local_port** (*int, None*) – The port used locally.

port_forward_stopall()
Disconnect all existing port forwarding connections.

prepare()
Prepare a Duct subclass for use (if not already prepared).

This method is called before the value of any of the fields referenced in *self.connection_fields* are retrieved. The fields include, by default: 'host', 'port', 'remote', 'cache', 'username', and 'password'. Subclasses may add or subtract from these special fields.

When called, it first checks whether the instance has already been prepared, and if not calls *_prepare* and then records that the instance has been successfully prepared.

SSHClient Quirks: This method may be overridden by subclasses, but provides the following default behaviour:

- Ensures *self.registry*, *self.remote* and *self.cache* values are instances of the right types.
- It replaces string values of *self.remote* and *self.cache* with remotes and caches looked up using *self.registry.lookup*.
- It looks through each of the fields nominated in *self.prepared_fields* and, if the corresponding value is callable, sets the value of that field to result of calling that value with a reference to *self*. By default, *prepared_fields* contains '_host', '_port', '_username', and '_password'.
- Ensures value of self.port is an integer (or None).

prepare_smartcards()
Prepare smartcards for use in authentication.

This method checks attempts to ensure that the each of the nominated smartcards is available and prepared for use. This may result in interactive requests for pin confirmation, depending on the card.

Returns

Returns *True* if at least one smartcard was activated, and *False* otherwise.

Return type bool

read_only
Whether this filesystem client should be permitted to attempt any write operations.

Type bool

reconnect()
Disconnects, and then reconnects, this client.

Note: This is equivalent to *duct.disconnect().connect()*.

Returns A reference to this object.

Return type *Duct* instance

remove(*path*, *recursive=False*)
Remove file(s) at a nominated path.

Directories (and their contents) will not be removed unless *recursive* is set to *True*.

Parameters

- **path** (*str*) – The path of the file/directory to be removed.
- **recursive** (*bool*) – Whether to remove directories and all of their contents.

reset()
Reset this *Duct* instance to its pre-preparation state.

This method disconnects from the service, resets any temporary authentication and restores the values of the attributes listed in *prepared_fields* to their values as of when *Duct.prepare* was called.

Returns A reference to this object.

Return type *Duct* instance

show_port_forwards()
Print to stdout the active port forwards associated with this client.

showdir(*path=None*)
Return a dataframe representation of a directory.

This method returns a *pandas.DataFrame* representation of the contents of a path, which are retrieved using *.dir(path)*. The exact columns will vary from filesystem to filesystem, depending on the fields returned by *.dir()*, but the returned DataFrame is guaranteed to at least have the columns: 'name' and 'type'.

Parameters **path** (*str*) – The path of the directory from which to show contents.

Returns A DataFrame representation of the contents of the nominated directory.

Return type pandas.DataFrame

update_host_keys()
Update host keys associated with this remote.

This method updates the SSH host-keys stored in *~/.ssh/known_hosts*, allowing one to successfully connect to hosts when servers are, for example, redeployed and have different host keys.

upload(*source*, *dest=None*, *overwrite=False*, *fs=None*)
Upload files from another filesystem.

This method (recursively) uploads a file/folder from path *source* on filesystem *fs* to the path *dest* on this filesytem, overwriting any existing file if *overwrite* is *True*. This is equivalent to *fs.download(. . . , fs=self)*.

Parameters

- **source** (*str*) – The path on the specified filesystem (*fs*) of the file to upload to this filesystem. If *source* ends with '/', and corresponds to a directory, the contents of source will be copied instead of copying the entire folder.
- **dest** (*str*) – The destination path on this filesystem. If not specified, the file/folder is uploaded into the default path, usually one's home folder, on this filesystem. If *dest* ends with '/' then file will be copied into destination folder, and will throw an error if path does not resolve to a directory.
- **overwrite** (*bool*) – *True* if the contents of any existing file by the same name should be overwritten, *False* otherwise.
- **fs** (FileSystemClient) – The FileSystemClient from which to load the file/folder at *source*. If not specified, defaults to the local filesystem.

SSHClient Quirks: Upload files from another filesystem.

This method (recursively) uploads a file/folder from path *source* on filesystem *fs* to the path *dest* on this filesytem, overwriting any existing file if *overwrite* is *True*. This is equivalent to *fs.download(. . . , fs=self)*.

Args:

source (str): The path on the specified filesystem (*fs*) of the file to upload to this filesystem. If *source* ends with '/', and corresponds to a directory, the contents of source will be copied instead of copying the entire folder.

dest (str): The destination path on this filesystem. If not specified, the file/folder is uploaded into the default path, usually one's home folder, on this filesystem. If *dest* ends with '/' then file will be copied into destination folder, and will throw an error if path does not resolve to a directory.

overwrite (bool): *True* if the contents of any existing file by the same name should be overwritten, *False* otherwise.

fs (FileSystemClient): The FileSystemClient from which to load the file/folder at *source*. If not specified, defaults to the local filesystem.

SSHClient Quirks: This method is overloaded so that local-to-remote uploads can be handled specially using *scp*. Uploads to any non-local filesystem are handled using the standard implementation.

`username`

Some services require authentication in order to connect to the service, in which case the appropriate username can be specified. If not specified at instantiation, your local login name will be used. If *True* was provided, you will be prompted to type your username at runtime as necessary. If *False* was provided, then *None* will be returned. You can specify a different username at runtime using: *duct.username = '<username>'*.

Type str

`walk`(*path=None*)

Explore the filesystem tree starting at a nominated path.

This method returns a generator which recursively walks over all paths that are children of *path*, one result for each directory, of form: (<path name>, [<directory 1>, . . .], [<file 1>, . . .])

Parameters **`path`** (*`str`*) – The path of the directory from which to enumerate contents.

Returns A generator of tuples, each tuple being associated with one directory that is either *path* or one of its descendants.

Return type generator<tuple>

ParamikoSSHClient

`class` `omniduct.remotes.ssh_paramiko.`**`ParamikoSSHClient`**(*smartcards=None*, ***kwargs*)

Bases: *`omniduct.remotes.base.RemoteClient`*

An experimental SSH client that uses a *paramiko* rather than command-line SSH backend. This client has been fully implemented and should work as is, but until it receives further testing, we recommend using the cli backed SSH client.

Attributes inherited from RemoteClient:

smartcard (dict): Mapping of smartcard names to system libraries compatible with *ssh-add -s '<system library>'* . . . .

Attributes inherited from Duct:

protocol (str): The name of the protocol for which this instance was created (especially useful if a *Duct* subclass supports multiple protocols).

name (str): The name given to this *Duct* instance (defaults to class name).

host (str): The host name providing the service (will be '127.0.0.1', if service is port forwarded from remote; use *._host* to see remote host).

port (int): The port number of the service (will be the port-forwarded local port, if relevant; for remote port use *._port*).

username (str, bool): The username to use for the service. password (str, bool): The password to use for the service. registry (None, omniduct.registry.DuctRegistry): A reference to a

DuctRegistry instance for runtime lookup of other services.

remote (None, omniduct.remotes.base.RemoteClient): A reference to a *RemoteClient* instance to manage connections to remote services.

cache (None, omniduct.caches.base.Cache): A reference to a *Cache* instance to add support for caching, if applicable.

connection_fields (tuple<str>, list<str>): A list of instance attributes to monitor for changes, whereupon the *Duct* instance should automatically disconnect. By default, the following attributes are monitored: 'host', 'port', 'remote', 'username', and 'password'.

prepared_fields (tuple<str>, list<str>): A list of instance attributes to be populated (if their values are callable) when the instance first connects to a service. Refer to *Duct.prepare* and *Duct._prepare* for more details. By default, the following attributes are prepared: '_host', '_port', '_username', and '_password'.

Additional attributes including *host*, *port*, *username* and *password* are documented inline.

Class Attributes:

AUTO_LOGGING_SCOPE (bool): Whether this class should be used by omniduct logging code as a "scope". Should be overridden by subclasses as appropriate.

DUCT_TYPE (Duct.Type): The type of *Duct* service that is provided by this Duct instance. Should be overridden by subclasses as appropriate.

PROTOCOLS (list<str>): The name(s) of any protocols that should be associated with this class. Should be overridden by subclasses as appropriate.

`class Type`

Bases: `enum.Enum`

The *Duct.Type* enum specifies all of the permissible values of *Duct.DUCT_TYPE*. Also determines the order in which ducts are loaded by DuctRegistry.

`__init__`(*smartcards=None*, ***kwargs*)

protocol (str, None): Name of protocol (used by Duct registries to inform Duct instances of how they were instantiated).

name (str, None): The name to used by the *Duct* instance (defaults to class name if not specified).

registry (DuctRegistry, None): The registry to use to lookup remote and/or cache instance specified by name.

remote (str, RemoteClient): The remote by which the ducted service should be contacted.

host (str): The hostname of the service to be used by this client. port (int): The port of the service to be used by this client. username (str, bool, None): The username to authenticate with if necessary.

If True, then users will be prompted at runtime for credentials.

password (str, bool, None): The password to authenticate with if necessary. If True, then users will be prompted at runtime for credentials.

cache(Cache, None): The cache client to be attached to this instance. Cache will only used by specific methods as configured by the client.

cache_namespace(str, None): The namespace to use by default when writing to the cache.

FileSystemClient Quirks:

cwd (None, str): The path prefix to use as the current working directory (if None, the user's home directory is used where that makes sense).

home (None, str): The path prefix to use as the current users' home directory. If not specified, it will default to an implementation- specific value (often '/').

read_only (bool): Whether the filesystem should only be able to perform read operations.

global_writes (bool): Whether to allow writes outside of the user's home folder.

**kwargs (dict): Additional keyword arguments to passed on to subclasses.

RemoteClient Quirks:

Args:

smartcards (dict): Mapping of smartcard names to system libraries compatible with *ssh-add -s '<system library>'* ….

connect()

Connect to the service backing this client.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Returns A reference to the current object.

Return type *Duct* instance

RemoteClient Quirks: Connect to the remote server.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Compared to base *Duct.connect*, this method will automatically catch the first *DuctAuthenticationError* error triggered by *Duct.connect*, and (if smartcards have been configured) attempt to re-initialise the smartcards before trying once more.

Returns: *Duct* instance: A reference to the current object.

dir(*path=None*)

Retrieve information about the children of a nominated directory.

This method returns a generator over *FileSystemFileDesc* objects that represent the files/directories that a present as children of the nominated path. If *path* is not a directory, an exception is raised. The path is interpreted as being relative to the current working directory (on remote filesytems, this will typically be the home folder).

Parameters **path** (*str*) – The path to examine for children.

Returns The children of *path* represented as *FileSystemFileDesc* objects.

Return type generator<FileSystemFileDesc>

disconnect()

Disconnect this client from backing service.

This method is automatically called during reconnections and/or at Python interpreter shutdown. It first calls *Duct._disconnect* (which should be implemented by subclasses) and then notifies the *RemoteClient* subclass, if present, to stop port-forwarding the remote service.

Returns A reference to this object.

Return type *Duct* instance

download(*source*, *dest=None*, *overwrite=False*, *fs=None*)
Download files to another filesystem.

This method (recursively) downloads a file/folder from path *source* on this filesystem to the path *dest* on filesytem *fs*, overwriting any existing file if *overwrite* is *True*.

Parameters

- **source** (*str*) – The path on this filesystem of the file to download to the nominated filesystem (*fs*). If *source* ends with '/' then contents of the the *source* directory will be copied into destination folder, and will throw an error if path does not resolve to a directory.
- **dest** (*str*) – The destination path on filesystem (*fs*). If not specified, the file/folder is downloaded into the default path, usually one's home folder. If *dest* ends with '/', and corresponds to a directory, the contents of source will be copied instead of copying the entire folder. If *dest* is otherwise a directory, an exception will be raised.
- **overwrite** (*bool*) – *True* if the contents of any existing file by the same name should be overwritten, *False* otherwise.
- **fs** (FileSystemClient) – The FileSystemClient into which the nominated file/folder *source* should be downloaded. If not specified, defaults to the local filesystem.

execute(*cmd*, ***kwargs*)
Execute a command on the remote server.

Parameters

- **cmd** (*str*) – The command to run on the remote associated with this instance.
- ****kwargs** (*dict*) – Additional keyword arguments to be passed on to *._execute*.

Returns The result of the execution.

Return type SubprocessResults

exists(*path*)
Check whether nominated path exists on this filesytem.

Parameters **path** (*str*) – The path for which to check existence.

Returns

True **if file/folder exists at nominated path, and** ***False*** otherwise.

Return type bool

find(*path_prefix=None*, ***attrs*)
Find a file or directory based on certain attributes.

This method searches for files or folders which satisfy certain constraints on the attributes of the file (as encoded into *FileSystemFileDesc*). Note that without attribute constraints, this method will function identically to *self.dir*.

Parameters

- **path_prefix** (*str*) – The path under which files/directories should be found.
- ****attrs** (*dict*) – Constraints on the fields of the *FileSystemFileDesc* objects associated with this filesystem, as constant values or callable objects (in which case the object will be called and should return True if attribute value is match, and False otherwise).

Returns

A generator over *FileSystemFileDesc* objects that are descendents of *path_prefix* and which statisfy provided constraints.

Return type generator<FileSystemFileDesc>

classmethod for_protocol(*protocol*)

Retrieve a *Duct* subclass for a given protocol.

Parameters **protocol** (*str*) – The protocol of interest.

Returns

The appropriate class for the provided, partially constructed with the *protocol* keyword argument set appropriately.

Return type functools.partial object

Raises `DuctProtocolUnknown` – If no class has been defined that offers the named protocol.

get_local_uri(*uri*)

Convert a remote uri to a local one.

This method takes a remote service uri accessible to the remote host and returns a local uri accessible directly on the local host, establishing any necessary port forwarding in the process.

Parameters **uri** (*str*) – The remote uri to be made local.

Returns A local uri that tunnels all traffic to the remote host.

Return type str

global_writes

Whether writes should be permitted outside of home directory. This write-lock is designed to prevent inadvertent scripted writing in potentially dangerous places.

Type bool

has_port_forward(*remote_host=None*, *remote_port=None*, *local_port=None*)

Check whether a port forward connection exists.

Parameters

- **remote_host** (*str*) – The hostname of the remote host in form: 'hostname(:port)'.
- **remote_port** (*int, None*) – The remote port of the service.
- **local_port** (*int, None*) – The port used locally.

Returns

Whether a port-forward for this remote service exists, or if local port is specified, whether that port is locally used for port forwarding.

Return type bool

host

The host name providing the service, or '127.0.0.1' if *self.remote* is not *None*, whereupon the service will be port-forwarded locally. You can view the remote hostname using *duct._host*, and change the remote host at runtime using: *duct.host = '<host>'*.

Type str

is_connected()

Check whether this *Duct* instances is currently connected.

This method checks to see whether a *Duct* instance is currently connected. This is performed by verifying that the remote host and port are still accessible, and then by calling *Duct._is_connected*, which should be implemented by subclasses.

Returns Whether this *Duct* instance is currently connected.

Return type bool

`is_port_bound`(*host*, *port*)

Check whether a port on a remote host is accessible.

This method checks to see whether a particular port is active on a given host by attempting to establish a connection with it.

Parameters

- **`host`** (*`str`*) – The hostname of the target service.
- **`port`** (*`int`*) – The port of the target service.

Returns Whether the port is active and accepting connections.

Return type bool

`isdir`(*path*)

Check whether a nominated path is directory.

Parameters **`path`** (*`str`*) – The path for which to check directory nature.

Returns *True* if folder exists at nominated path, and *False* otherwise.

Return type bool

`isfile`(*path*)

Check whether a nominated path is a file.

Parameters **`path`** (*`str`*) – The path for which to check file nature.

Returns *True* if a file exists at nominated path, and *False* otherwise.

Return type bool

`listdir`(*path=None*)

Retrieve the names of the children of a nomianted directory.

This method inspects the contents of a directory using *.dir(path)*, and returns the names of child members as strings. *path* is interpreted relative to the current working directory (on remote filesytems, this will typically be the home folder).

Parameters **`path`** (*`str`*) – The path of the directory from which to enumerate filenames.

Returns The names of all children of the nominated directory.

Return type list<str>

`mkdir`(*path*, *recursive=True*, *exist_ok=False*)

Create a directory at the given path.

Parameters

- **`path`** (*`str`*) – The path of the directory to create.
- **`recursive`** (*`bool`*) – Whether to recursively create any parents of this path if they do not already exist.

Note: *exist_ok* is passed onto subclass implementations of *_mkdir* rather that implementing the existence check using *.exists* so that they can avoid the overhead associated with multiple operations, which can be costly in some cases.

open(*path*, *mode='rt'*)
Open a file for reading and/or writing.

This method opens the file at the given path for reading and/or writing operations. The object returned is programmatically interchangeable with any other Python file-like object, including specification of file modes. If the file is opened in write mode, changes will only be flushed to the source filesystem when the file is closed.

Parameters

- **path** (`str`) – The path of the file to open.
- **mode** (`str`) – All standard Python file modes.

Returns An opened file-like object.

Return type *FileSystemFile* or file-like

ParamikoSSHClient Quirks: Paramiko offers a complete file-like abstraction for files opened over sftp, so we use that abstraction rather than a *FileSystemFile*. Results should be indistinguishable.

password
Some services require authentication in order to connect to the service, in which case the appropriate password can be specified. If *True* was provided at instantiation, you will be prompted to type your password at runtime when necessary. If *False* was provided, then *None* will be returned. You can specify a different password at runtime using: *duct.password = '<password>'*.

Type str

path_basename(*path*)
Extract the last component of a given path.

Components are determined by splitting by *self.path_separator*. Note that if a path ends with a path separator, the basename will be the empty string.

Parameters **path** (`str`) – The path from which the basename should be extracted.

Returns The extracted basename.

Return type str

path_cwd
The path prefix associated with the current working directory. If not otherwise set, it will be the users' home directory, and will be the prefix used by all non-absolute path references on this filesystem.

Type str

path_dirname(*path*)
Extract the parent directory for provided path.

This method returns the entire path except for the basename (the last component), where components are determined by splitting by *self.path_separator*.

Parameters **path** (`str`) – The path from which the directory path should be extracted.

Returns The extracted directory path.

Return type str

`path_home`

The path prefix to use as the current users' home directory. Unless *cwd* is set, this will be the prefix to use for all non-absolute path references on this filesystem. This is assumed not to change between connections, and so will not be updated on client reconnections. Unless *global_writes* is set to *True*, this will be the only folder into which this client is permitted to write.

Type str

`path_join`(*path*, **components*)

Generate a new path by joining together multiple paths.

If any component starts with *self.path_separator* or '~', then all previous path components are discarded, and the effective base path becomes that component (with '~' expanding to *self.path_home*). Note that this method does *not* simplify paths components like '..'. Use *self.path_normpath* for this purpose.

Parameters

- **`path`** (*`str`*) – The base path to which components should be joined.
- **`*components`** (*`str`*) – Any additional components to join to the base path.

Returns The path resulting from joining all of the components nominated, in order, to the base path.

Return type str

`path_normpath`(*path*)

Normalise a pathname.

This method returns the normalised (absolute) path corresponding to *path* on this filesystem.

Parameters **`path`** (*`str`*) – The path to normalise (make absolute).

Returns The normalised path.

Return type str

`path_separator`

The character(s) to use in separating path components. Typically this will be '/'.

Type str

`port`

The local port for the service. If *self.remote* is not *None*, the port will be port-forwarded from the remote host. To see the port used on the remote host refer to *duct._port*. You can change the remote port at runtime using: *duct.port = <port>*.

Type int

`port_forward`(*remote_host*, *remote_port=None*, *local_port=None*)

Initiate a port forward connection.

This method establishes a local port forwarding from a local port *local* to remote port *remote*. If *local* is *None*, an available local port is automatically chosen. If the remote port is already forwarded, a new connection is not established.

Parameters

- **`remote_host`** (*`str`*) – The hostname of the remote host in form: 'hostname(:port)'.
- **`remote_port`** (*`int, None`*) – The remote port of the service.
- **`local_port`** (*`int, None`*) – The port to use locally (automatically determined if not specified).

Returns The local port which is port forwarded to the remote service.

Return type int

`port_forward_stop`(*local_port=None*, *remote_host=None*, *remote_port=None*)
Disconnect an existing port forward connection.

If a local port is provided, then the forwarding (if any) associated with that port is found and stopped; otherwise any established port forwarding associated with the nominated remote service is stopped.

Parameters

- **`remote_host`** (*`str`*) – The hostname of the remote host in form: 'hostname(:port)'.
- **`remote_port`** (*`int, None`*) – The remote port of the service.
- **`local_port`** (*`int, None`*) – The port used locally.

`port_forward_stopall`()
Disconnect all existing port forwarding connections.

`prepare`()
Prepare a Duct subclass for use (if not already prepared).

This method is called before the value of any of the fields referenced in *self.connection_fields* are retrieved. The fields include, by default: 'host', 'port', 'remote', 'cache', 'username', and 'password'. Subclasses may add or subtract from these special fields.

When called, it first checks whether the instance has already been prepared, and if not calls *_prepare* and then records that the instance has been successfully prepared.

ParamikoSSHClient Quirks: This method may be overridden by subclasses, but provides the following default behaviour:

- Ensures *self.registry*, *self.remote* and *self.cache* values are instances of the right types.
- It replaces string values of *self.remote* and *self.cache* with remotes and caches looked up using *self.registry.lookup*.
- It looks through each of the fields nominated in *self.prepared_fields* and, if the corresponding value is callable, sets the value of that field to result of calling that value with a reference to *self*. By default, *prepared_fields* contains '_host', '_port', '_username', and '_password'.
- Ensures value of self.port is an integer (or None).

`prepare_smartcards`()
Prepare smartcards for use in authentication.

This method checks attempts to ensure that the each of the nominated smartcards is available and prepared for use. This may result in interactive requests for pin confirmation, depending on the card.

Returns

Returns *True* if at least one smartcard was activated, and *False* otherwise.

Return type bool

`read_only`
Whether this filesystem client should be permitted to attempt any write operations.

Type bool

`reconnect`()
Disconnects, and then reconnects, this client.

Note: This is equivalent to *duct.disconnect().connect()*.

Returns A reference to this object.

Return type *Duct* instance

`remove`(*path*, *recursive=False*)
Remove file(s) at a nominated path.

Directories (and their contents) will not be removed unless *recursive* is set to *True*.

Parameters

- **`path`** (`str`) – The path of the file/directory to be removed.
- **`recursive`** (`bool`) – Whether to remove directories and all of their contents.

`reset`()
Reset this *Duct* instance to its pre-preparation state.

This method disconnects from the service, resets any temporary authentication and restores the values of the attributes listed in *prepared_fields* to their values as of when *Duct.prepare* was called.

Returns A reference to this object.

Return type *Duct* instance

`show_port_forwards`()
Print to stdout the active port forwards associated with this client.

`showdir`(*path=None*)
Return a dataframe representation of a directory.

This method returns a *pandas.DataFrame* representation of the contents of a path, which are retrieved using *.dir(path)*. The exact columns will vary from filesystem to filesystem, depending on the fields returned by *.dir()*, but the returned DataFrame is guaranteed to at least have the columns: 'name' and 'type'.

Parameters **`path`** (`str`) – The path of the directory from which to show contents.

Returns A DataFrame representation of the contents of the nominated directory.

Return type pandas.DataFrame

`upload`(*source*, *dest=None*, *overwrite=False*, *fs=None*)
Upload files from another filesystem.

This method (recursively) uploads a file/folder from path *source* on filesystem *fs* to the path *dest* on this filesytem, overwriting any existing file if *overwrite* is *True*. This is equivalent to *fs.download(. . . , fs=self)*.

Parameters

- **`source`** (`str`) – The path on the specified filesystem (*fs*) of the file to upload to this filesystem. If *source* ends with '/', and corresponds to a directory, the contents of source will be copied instead of copying the entire folder.
- **`dest`** (`str`) – The destination path on this filesystem. If not specified, the file/folder is uploaded into the default path, usually one's home folder, on this filesystem. If *dest* ends with '/' then file will be copied into destination folder, and will throw an error if path does not resolve to a directory.
- **`overwrite`** (`bool`) – *True* if the contents of any existing file by the same name should be overwritten, *False* otherwise.
- **`fs`** (`FileSystemClient`) – The FileSystemClient from which to load the file/folder at *source*. If not specified, defaults to the local filesystem.

`username`
Some services require authentication in order to connect to the service, in which case the appropriate username can be specified. If not specified at instantiation, your local login name will be used. If *True*

was provided, you will be prompted to type your username at runtime as necessary. If *False* was provided, then *None* will be returned. You can specify a different username at runtime using: *duct.username = '<username>'*.

Type str

walk(*path=None*)

Explore the filesystem tree starting at a nominated path.

This method returns a generator which recursively walks over all paths that are children of *path*, one result for each directory, of form: (<path name>, [<directory 1>, . . .], [<file 1>, . . .])

Parameters **path** (*str*) – The path of the directory from which to enumerate contents.

Returns A generator of tuples, each tuple being associated with one directory that is either *path* or one of its descendants.

Return type generator<tuple>

5.5 Caches

All remote clients are expected to be subclasses of *Cache*, and so will share a common API. Protocol implementations are also free to add extra methods, which are documented in the "Subclass Reference" section below.

5.5.1 Common API

omniduct.caches.base.**cached_method**(*key*, *namespace=<function <lambda>>*, *cache=<function <lambda>>*, *use_cache=<function <lambda>>*, *renew=<function <lambda>>*, *serializer=<function <lambda>>*, *metadata=<function <lambda>>*)

Wrap a method of a *Duct* class and add caching capabilities.

All arguments of this function are expected to be functions taking two arguments: a reference to current instance of the class (*self*) and a dictionary of arguments passed to the function (*kwargs*).

Parameters

- **key** (*function -> str*) – The key under which the value returned by the wrapped function should be stored.
- **namespace** (*function -> str*) – The namespace under which the key should be stored (default: *"<duct class name>.<duct instance name>"*).
- **cache** (*function -> Cache*) – The instance of cache via which to store the output of the wrapped function (default: *self.cache*).
- **use_cache** (*function -> bool*) – Whether or not to use the caching functionality (default: *True*).
- **renew** (*function -> bool*) – Whether to renew the stored cache, overriding if a value has already been stored (default: *False*).
- **serializer** (*function -> Serializer*) – The *Serializer* subclass to use when storing the return object (default: *PickleSerializer*).
- **metadata** (*function -> None, dict*) – A dictionary of additional metadata to be stored alongside the wrapped function's output (default: *None*).

Returns

The (potentially cached) object returned when calling the wrapped function.

Return type object

Raises `Exception` – If cache fails to store the output of the wrapped function, and the omniduct configuration key *cache_fail_hard* is *True*, then the underlying exceptions raised by the Cache instance will be reraised.

class `omniduct.caches.base.`**`Cache`**(***kwargs*)

Bases: `omniduct.duct.Duct`

An abstract class providing the common API for all cache clients.

Attributes inherited from Duct:

protocol (str): The name of the protocol for which this instance was created (especially useful if a *Duct* subclass supports multiple protocols).

name (str): The name given to this *Duct* instance (defaults to class name).

host (str): The host name providing the service (will be '127.0.0.1', if service is port forwarded from remote; use *._host* to see remote host).

port (int): The port number of the service (will be the port-forwarded local port, if relevant; for remote port use *._port*).

username (str, bool): The username to use for the service. password (str, bool): The password to use for the service. registry (None, omniduct.registry.DuctRegistry): A reference to a

DuctRegistry instance for runtime lookup of other services.

remote (None, omniduct.remotes.base.RemoteClient): A reference to a *RemoteClient* instance to manage connections to remote services.

cache (None, omniduct.caches.base.Cache): A reference to a *Cache* instance to add support for caching, if applicable.

connection_fields (tuple<str>, list<str>): A list of instance attributes to monitor for changes, whereupon the *Duct* instance should automatically disconnect. By default, the following attributes are monitored: 'host', 'port', 'remote', 'username', and 'password'.

prepared_fields (tuple<str>, list<str>): A list of instance attributes to be populated (if their values are callable) when the instance first connects to a service. Refer to *Duct.prepare* and *Duct._prepare* for more details. By default, the following attributes are prepared: '_host', '_port', '_username', and '_password'.

Additional attributes including *host*, *port*, *username* and *password* are documented inline.

Class Attributes:

AUTO_LOGGING_SCOPE (bool): Whether this class should be used by omniduct logging code as a "scope". Should be overridden by subclasses as appropriate.

DUCT_TYPE (Duct.Type): The type of *Duct* service that is provided by this Duct instance. Should be overridden by subclasses as appropriate.

PROTOCOLS (list<str>): The name(s) of any protocols that should be associated with this class. Should be overridden by subclasses as appropriate.

`__init__`(***kwargs*)

protocol (str, None): Name of protocol (used by Duct registries to inform Duct instances of how they were instantiated).

name (str, None): The name to used by the *Duct* instance (defaults to class name if not specified).

registry (DuctRegistry, None): The registry to use to lookup remote and/or cache instance specified by name.

remote (str, RemoteClient): The remote by which the ducted service should be contacted.

host (str): The hostname of the service to be used by this client. port (int): The port of the service to be used by this client. username (str, bool, None): The username to authenticate with if necessary.

If True, then users will be prompted at runtime for credentials.

password (str, bool, None): The password to authenticate with if necessary. If True, then users will be prompted at runtime for credentials.

cache(Cache, None): The cache client to be attached to this instance. Cache will only used by specific methods as configured by the client.

cache_namespace(str, None): The namespace to use by default when writing to the cache.

`set` (*key*, *value*, *namespace=None*, *serializer=None*, *metadata=None*)
Set the value of a key.

Parameters

- **`key`** (*`str`*) – The key for which *value* should be stored.
- **`value`** (*`object`*) – The value to be stored.
- **`namespace`** (*`str, None`*) – The namespace to be used.
- **`serializer`** (*`Serializer`*) – The *Serializer* subclass to use for the serialisation of value into the cache. (default=PickleSerializer)
- **`metadata`** (*`dict, None`*) – Additional metadata to be stored with the value in the cache. Values must be serializable via *yaml.safe_dump*.

`set_metadata` (*key*, *metadata*, *namespace=None*, *replace=False*)
Set the metadata associated with a stored key, creating the key if it is missing.

Parameters

- **`key`** (*`str`*) – The key for which *value* should be stored.
- **`metadata`** (*`dict, None`*) – Additional/override metadata to be stored for *key* in the cache. Values must be serializable via *yaml.safe_dump*.
- **`namespace`** (*`str, None`*) – The namespace to be used.
- **`replace`** (*`bool`*) – Whether the provided metadata should entirely replace any existing metadata, or just update it. (default=False)

`get` (*key*, *namespace=None*, *serializer=None*)
Retrieve the value associated with the nominated key from the cache.

Parameters

- **`key`** (*`str`*) – The key for which *value* should be retrieved.
- **`namespace`** (*`str, None`*) – The namespace to be used.
- **`serializer`** (*`Serializer`*) – The *Serializer* subclass to use for the deserialisation of value from the cache. (default=PickleSerializer)

Returns The (appropriately deserialized) object stored in the cache.

Return type object

`get_bytecount`(*key*, *namespace=None*)

Retrieve the number of bytes used by a stored key.

This bytecount may or may not include metadata storage, depending on the backend.

Parameters

- **key** (*str*) – The key for which to extract the bytecount.
- **namespace** (*str, None*) – The namespace to be used.

Returns

The number of bytes used by the stored value associated with the nominated key and namespace.

Return type int

`get_metadata`(*key*, *namespace=None*)

Retrieve metadata associated with the nominated key from the cache.

Parameters

- **key** (*str*) – The key for which to extract metadata.
- **namespace** (*str, None*) – The namespace to be used.

Returns The metadata associated with this namespace and key.

Return type dict

`unset`(*key*, *namespace=None*)

Remove the nominated key from the cache.

Parameters

- **key** (*str*) – The key which should be unset.
- **namespace** (*str, None*) – The namespace to be used.

`unset_namespace`(*namespace=None*)

Remove an entire namespace from the cache.

Parameters **namespace** (*str, None*) – The namespace to be removed.

`namespaces`

A list of the namespaces stored in the cache.

Type list <str,None>

`has_namespace`(*namespace=None*)

Check whether the cache has the nominated namespace.

Parameters **namespace** (*str,None*) – The namespace for which to check for existence.

Returns Whether the cache has the nominated namespaces.

Return type bool

`keys`(*namespace=None*)

Collect a list of all the keys present in the nominated namespaces.

Parameters **namespace** (*str,None*) – The namespace from which to extract all of the keys.

Returns The keys stored in the cache for the nominated namespace.

Return type list<str>

has_key (*key*, *namespace=None*)
Check whether the cache as a nominated key.

Parameters

- **key** (`str`) – The key for which to check existence.
- **namespace** (`str, None`) – The namespace from which to extract all of the keys.

Returns

Whether the cache has a value for the nominated namespace and key.

Return type bool

get_total_bytecount (*namespaces=None*)
Retrieve the total number of bytes used by the cache.

This method iterates over all (nominated) namespaces and the keys therein, summing the result of *.get_bytecount(. . .)* on each.

Parameters namespaces (`list<str,None>`) – The namespaces to which the bytecount should be restricted.

Returns The total number of bytes used by the nominated namespaces.

Return type int

describe (*namespaces=None*)
Return a pandas DataFrame showing all keys and their metadata.

Parameters namespaces (`list<str,None>`) – The namespaces to which the summary should be restricted.

Returns

A representation of keys in the cache. Will include at least the following columns: ['bytes', 'namespace', 'key', 'created', 'last_accessed']. Any additional metadata for keys will be appended to these columns.

Return type pandas.DataFrame

prune (*namespaces=None*, *max_age=None*, *max_bytes=None*, *total_bytes=None*)
Remove keys from the cache in order to satisfy nominated constraints.

Parameters

- **namespaces** (`list<str, None>`) – The namespaces to consider for pruning.
- **max_age** (`None, int, timedelta, relativedelta, date, datetime`) – The number of days, a timedelta, or a relativedelta, indicating the maximum age of items in the cache (based on last accessed date). Deltas are expected to be positive.
- **max_bytes** (`None, int`) – The maximum number of bytes for *each* key, allowing the pruning of larger keys.
- **total_bytes** (`None, int`) – The total number of bytes for the entire cache. Keys will be removed from least recently accessed to most recently accessed until the constraint is satisfied. This constraint will be applied after max_age and max_bytes.

connect ()
Connect to the service backing this client.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Returns A reference to the current object.

Return type *Duct* instance

disconnect()

Disconnect this client from backing service.

This method is automatically called during reconnections and/or at Python interpreter shutdown. It first calls *Duct._disconnect* (which should be implemented by subclasses) and then notifies the *RemoteClient* subclass, if present, to stop port-forwarding the remote service.

Returns A reference to this object.

Return type *Duct* instance

is_connected()

Check whether this *Duct* instances is currently connected.

This method checks to see whether a *Duct* instance is currently connected. This is performed by verifying that the remote host and port are still accessible, and then by calling *Duct._is_connected*, which should be implemented by subclasses.

Returns Whether this *Duct* instance is currently connected.

Return type bool

prepare()

Prepare a Duct subclass for use (if not already prepared).

This method is called before the value of any of the fields referenced in *self.connection_fields* are retrieved. The fields include, by default: 'host', 'port', 'remote', 'cache', 'username', and 'password'. Subclasses may add or subtract from these special fields.

When called, it first checks whether the instance has already been prepared, and if not calls *_prepare* and then records that the instance has been successfully prepared.

Cache Quirks: This method may be overridden by subclasses, but provides the following default behaviour:

- Ensures *self.registry*, *self.remote* and *self.cache* values are instances of the right types.
- It replaces string values of *self.remote* and *self.cache* with remotes and caches looked up using *self.registry.lookup*.
- It looks through each of the fields nominated in *self.prepared_fields* and, if the corresponding value is callable, sets the value of that field to result of calling that value with a reference to *self*. By default, *prepared_fields* contains '_host', '_port', '_username', and '_password'.
- Ensures value of self.port is an integer (or None).

5.5.2 Subclass Reference

For comprehensive documentation on any particular subclass, please refer to one of the below documents.

FileSystemCache

class omniduct.caches.filesystem.**FileSystemCache**(***kwargs*)

Bases: *omniduct.caches.base.Cache*

An implementation of *Cache* that wraps around a *FilesystemClient*.

Attributes inherited from Duct:

protocol (str): The name of the protocol for which this instance was created (especially useful if a *Duct* subclass supports multiple protocols).

name (str): The name given to this *Duct* instance (defaults to class name).

host (str): The host name providing the service (will be '127.0.0.1', if service is port forwarded from remote; use *._host* to see remote host).

port (int): The port number of the service (will be the port-forwarded local port, if relevant; for remote port use *._port*).

username (str, bool): The username to use for the service. password (str, bool): The password to use for the service. registry (None, omniduct.registry.DuctRegistry): A reference to a

DuctRegistry instance for runtime lookup of other services.

remote (None, omniduct.remotes.base.RemoteClient): A reference to a *RemoteClient* instance to manage connections to remote services.

cache (None, omniduct.caches.base.Cache): A reference to a *Cache* instance to add support for caching, if applicable.

connection_fields (tuple<str>, list<str>): A list of instance attributes to monitor for changes, whereupon the *Duct* instance should automatically disconnect. By default, the following attributes are monitored: 'host', 'port', 'remote', 'username', and 'password'.

prepared_fields (tuple<str>, list<str>): A list of instance attributes to be populated (if their values are callable) when the instance first connects to a service. Refer to *Duct.prepare* and *Duct._prepare* for more details. By default, the following attributes are prepared: '_host', '_port', '_username', and '_password'.

Additional attributes including *host*, *port*, *username* and *password* are documented inline.

Class Attributes:

AUTO_LOGGING_SCOPE (bool): Whether this class should be used by omniduct logging code as a "scope". Should be overridden by subclasses as appropriate.

DUCT_TYPE (Duct.Type): The type of *Duct* service that is provided by this Duct instance. Should be overridden by subclasses as appropriate.

PROTOCOLS (list<str>): The name(s) of any protocols that should be associated with this class. Should be overridden by subclasses as appropriate.

class Type

Bases: `enum.Enum`

The *Duct.Type* enum specifies all of the permissible values of *Duct.DUCT_TYPE*. Also determines the order in which ducts are loaded by DuctRegistry.

`__init__`(***kwargs*)

protocol (str, None): Name of protocol (used by Duct registries to inform Duct instances of how they were instantiated).

name (str, None): The name to used by the *Duct* instance (defaults to class name if not specified).

registry (DuctRegistry, None): The registry to use to lookup remote and/or cache instance specified by name.

remote (str, RemoteClient): The remote by which the ducted service should be contacted.

host (str): The hostname of the service to be used by this client. port (int): The port of the service to be used by this client. username (str, bool, None): The username to authenticate with if necessary.

If True, then users will be prompted at runtime for credentials.

password (str, bool, None): The password to authenticate with if necessary. If True, then users will be prompted at runtime for credentials.

cache(Cache, None): The cache client to be attached to this instance. Cache will only used by specific methods as configured by the client.

cache_namespace(str, None): The namespace to use by default when writing to the cache.

FileSystemCache Quirks: path (str): The top-level path of the cache in the filesystem. fs (FileSystemClient, str): The filesystem client to use as the

datastore of this cache. If not specified, this will default to the local filesystem using *LocalFsClient*. If specified as a string, and connected to a *DuctRegistry*, upon first use an attempt will be made to look up a *FileSystemClient* instance in the registry by this name.

`connect`()

Connect to the service backing this client.

It is not normally necessary for a user to manually call this function, since when a connection is required, it is automatically created.

Returns A reference to the current object.

Return type *Duct* instance

`describe`(*namespaces=None*)

Return a pandas DataFrame showing all keys and their metadata.

Parameters **`namespaces`** (*`list<str,None>`*) – The namespaces to which the summary should be restricted.

Returns

A representation of keys in the cache. Will include at least the following columns: [‘bytes’, ‘namespace’, ‘key’, ‘created’, ‘last_accessed’]. Any additional metadata for keys will be appended to these columns.

Return type pandas.DataFrame

`disconnect`()

Disconnect this client from backing service.

This method is automatically called during reconnections and/or at Python interpreter shutdown. It first calls *Duct._disconnect* (which should be implemented by subclasses) and then notifies the *RemoteClient* subclass, if present, to stop port-forwarding the remote service.

Returns A reference to this object.

Return type *Duct* instance

`classmethod for_protocol`(*protocol*)

Retrieve a *Duct* subclass for a given protocol.

Parameters **`protocol`** (*`str`*) – The protocol of interest.

Returns

The appropriate class for the provided, partially constructed with the *protocol* keyword argument set appropriately.

Return type functools.partial object

Raises `DuctProtocolUnknown` – If no class has been defined that offers the named protocol.

get (*key*, *namespace=None*, *serializer=None*)

Retrieve the value associated with the nominated key from the cache.

Parameters

- **key** (*str*) – The key for which *value* should be retrieved.
- **namespace** (*str, None*) – The namespace to be used.
- **serializer** (*Serializer*) – The *Serializer* subclass to use for the deserialisation of value from the cache. (default=PickleSerializer)

Returns The (appropriately deserialized) object stored in the cache.

Return type object

get_bytecount (*key*, *namespace=None*)

Retrieve the number of bytes used by a stored key.

This bytecount may or may not include metadata storage, depending on the backend.

Parameters

- **key** (*str*) – The key for which to extract the bytecount.
- **namespace** (*str, None*) – The namespace to be used.

Returns

The number of bytes used by the stored value associated with the nominated key and namespace.

Return type int

get_metadata (*key*, *namespace=None*)

Retrieve metadata associated with the nominated key from the cache.

Parameters

- **key** (*str*) – The key for which to extract metadata.
- **namespace** (*str, None*) – The namespace to be used.

Returns The metadata associated with this namespace and key.

Return type dict

get_total_bytecount (*namespaces=None*)

Retrieve the total number of bytes used by the cache.

This method iterates over all (nominated) namespaces and the keys therein, summing the result of *.get_bytecount(. . .)* on each.

Parameters namespaces (*list<str,None>*) – The namespaces to which the bytecount should be restricted.

Returns The total number of bytes used by the nominated namespaces.

Return type int

has_key (*key*, *namespace=None*)

Check whether the cache as a nominated key.

Parameters

- **key** (*str*) – The key for which to check existence.
- **namespace** (*str,None*) – The namespace from which to extract all of the keys.

Returns

Whether the cache has a value for the nominated namespace and key.

Return type bool

has_namespace(*namespace=None*)
Check whether the cache has the nominated namespace.

Parameters **namespace** (*str, None*) – The namespace for which to check for existence.

Returns Whether the cache has the nominated namespaces.

Return type bool

host
The host name providing the service, or '127.0.0.1' if *self.remote* is not *None*, whereupon the service will be port-forwarded locally. You can view the remote hostname using *duct._host*, and change the remote host at runtime using: *duct.host = '<host>'*.

Type str

is_connected()
Check whether this *Duct* instances is currently connected.

This method checks to see whether a *Duct* instance is currently connected. This is performed by verifying that the remote host and port are still accessible, and then by calling *Duct._is_connected*, which should be implemented by subclasses.

Returns Whether this *Duct* instance is currently connected.

Return type bool

keys(*namespace=None*)
Collect a list of all the keys present in the nominated namespaces.

Parameters **namespace** (*str, None*) – The namespace from which to extract all of the keys.

Returns The keys stored in the cache for the nominated namespace.

Return type list<str>

namespaces
A list of the namespaces stored in the cache.

Type list <str,None>

password
Some services require authentication in order to connect to the service, in which case the appropriate password can be specified. If *True* was provided at instantiation, you will be prompted to type your password at runtime when necessary. If *False* was provided, then *None* will be returned. You can specify a different password at runtime using: *duct.password = '<password>'*.

Type str

port
The local port for the service. If *self.remote* is not *None*, the port will be port-forwarded from the remote host. To see the port used on the remote host refer to *duct._port*. You can change the remote port at runtime using: *duct.port = <port>*.

Type int

prepare()
Prepare a Duct subclass for use (if not already prepared).

This method is called before the value of any of the fields referenced in *self.connection_fields* are retrieved. The fields include, by default: 'host', 'port', 'remote', 'cache', 'username', and 'password'. Subclasses may add or subtract from these special fields.

When called, it first checks whether the instance has already been prepared, and if not calls *_prepare* and then records that the instance has been successfully prepared.

prune(*namespaces=None*, *max_age=None*, *max_bytes=None*, *total_bytes=None*)
Remove keys from the cache in order to satisfy nominated constraints.

Parameters

- **namespaces** (*list<str, None>*) – The namespaces to consider for pruning.
- **max_age** (*None, int, timedelta, relativedelta, date, datetime*) – The number of days, a timedelta, or a relativedelta, indicating the maximum age of items in the cache (based on last accessed date). Deltas are expected to be positive.
- **max_bytes** (*None, int*) – The maximum number of bytes for *each* key, allowing the pruning of larger keys.
- **total_bytes** (*None, int*) – The total number of bytes for the entire cache. Keys will be removed from least recently accessed to most recently accessed until the constraint is satisfied. This constraint will be applied after max_age and max_bytes.

reconnect()
Disconnects, and then reconnects, this client.

Note: This is equivalent to *duct.disconnect().connect()*.

Returns A reference to this object.

Return type *Duct* instance

reset()
Reset this *Duct* instance to its pre-preparation state.

This method disconnects from the service, resets any temporary authentication and restores the values of the attributes listed in *prepared_fields* to their values as of when *Duct.prepare* was called.

Returns A reference to this object.

Return type *Duct* instance

set(*key*, *value*, *namespace=None*, *serializer=None*, *metadata=None*)
Set the value of a key.

Parameters

- **key** (*str*) – The key for which *value* should be stored.
- **value** (*object*) – The value to be stored.
- **namespace** (*str, None*) – The namespace to be used.
- **serializer** (*Serializer*) – The *Serializer* subclass to use for the serialisation of value into the cache. (default=PickleSerializer)
- **metadata** (*dict, None*) – Additional metadata to be stored with the value in the cache. Values must be serializable via *yaml.safe_dump*.

set_metadata(*key*, *metadata*, *namespace=None*, *replace=False*)
Set the metadata associated with a stored key, creating the key if it is missing.

Parameters

- **`key`** (*`str`*) – The key for which *value* should be stored.
- **`metadata`** (*`dict, None`*) – Additional/override metadata to be stored for *key* in the cache. Values must be serializable via *yaml.safe_dump*.
- **`namespace`** (*`str, None`*) – The namespace to be used.
- **`replace`** (*`bool`*) – Whether the provided metadata should entirely replace any existing metadata, or just update it. (default=False)

`unset`(*key*, *namespace=None*)
Remove the nominated key from the cache.

Parameters

- **`key`** (*`str`*) – The key which should be unset.
- **`namespace`** (*`str, None`*) – The namespace to be used.

`unset_namespace`(*namespace=None*)
Remove an entire namespace from the cache.

Parameters **`namespace`** (*`str, None`*) – The namespace to be removed.

`username`
Some services require authentication in order to connect to the service, in which case the appropriate username can be specified. If not specified at instantiation, your local login name will be used. If *True* was provided, you will be prompted to type your username at runtime as necessary. If *False* was provided, then *None* will be returned. You can specify a different username at runtime using: *duct.username = '<username>'*.

Type str

5.6 Registry Management

Omniduct provides some simple tooling to help manage ensembles of *Duct* configurations. The primary tool is the *DuctRegistry*, which manages *Duct* instances and allows them to communicate with each other.

For some simple examples on its use, please refer to the *Quickstart*.

`class` `omniduct.registry.`**`DuctRegistry`**(*config=None*)
Bases: `object`

A convenient registry for *Duct* instances.

This class provides a simple interface to a pool of configured services, allowing convenient lookups of available services and the creation of new ones. It also allows for the batch creation of services from a shared configuration, which is especially useful in a company deployment.

`class ServicesProxy`(*registry*, *by_kind=True*)
Bases: `omniduct.utils.proxies.NestedDictObjectProxy`

A wrapper around *NestedDictObjectProxy* which is used to expose the services attached to a *DuctRegistry* as attributes on an object, optionally nested by service type.

`__init__`(*registry*, *by_kind=True*)
Initialize self. See help(type(self)) for accurate signature.

`registry`
The registry which hosts the services.

Type *DuctRegistry*

`__init__`(*config=None*)

Parameters **config** (*iterable, dict, str, None*) – Refer to *.import_from_config* for more details (default: *None*).

`register`(*duct*, *name=None*, *override=False*, *register_magics=True*)
Register an existing Duct instance into the registry.

Names of ducts can consist of any valid Python identifier, and multiple names can be provided as a comma separated list in which case the names will be aliases referring to the same *Duct* instance. Keep in mind that any name must uniquely identify one *Duct* instance.

Parameters

- **duct** (Duct) – The *Duct* instance to be registered.
- **name** (*str*) – An optional name to use when registering. If not provided this will fall back to *duct.name*. If neither is configured, an error will be thrown. Name can be a comma-separated list of names, in which case the names are aliases and will point to the same *Duct* instance.
- **override** (*bool*) – Whether to override any existing *Duct* instance of the same name. If *False*, any overrides will result in an exception.

Returns The *Duct* instance being registered.

Return type *Duct*

`new`(*name*, *protocol*, *override=False*, *register_magics=True*, ***kwargs*)
Create a new service and register it into the registry.

Parameters

- **name** (*str*) – The name (or names) of the target service. If multiple aliases are to be used, names should be a comma separated list. See *.register* for more details.
- **protocol** (*str*) – The protocol of the new service.
- **override** (*bool*) – Whether to override any existing *Duct* instance of the same name. If *False*, any overrides will result in an exception.
- **register_magics** (*bool*) – Whether to register the magics if running in and IPython session (default: *True*).
- ****kwargs** (*dict*) – Additional arguments to pass to the constructor of the class associated with the nominated protocol.

Returns The *Duct* instance registered into the registry.

Return type *Duct*

`names`
The names of all ducts in the registry.

Type list

`lookup`(*name*, *kind=None*)
Look up an existing registered *Duct* by name and (optionally) kind.

Parameters

- **name** (*str*) – The name of the *Duct* instance.
- **kind** (*str*, Duct.Type) – The kind of *Duct* to which the lookup should be restricted.

Returns The looked up *Duct* instance.

Return type *Duct*

Raises `DuctNotFound` – If no *Duct* can be found for requested name and/or type.

populate_namespace(*namespace=None*, *names=None*, *kinds=None*)
Populate a nominated namespace with references to a subset of ducts.

While a registry object is a great way to store and configure *Duct* instances, it is sometimes desirable to surface frequently used instances in other more convenient namespaces (such as the globals of your module).

Parameters

- **namespace** (*dict, None*) – The namespace to populate. If using from a module you can pass *globals()*. If *None*, a new dictionary is created, populated and then returned.
- **names** (*list<str>, None*) – The names to include in the population. If not specified then all names will be exported.
- **kinds** (*list<str>, None*) – The kinds of ducts to include in the population. If not specified, all kinds will be exported.

Returns The populated namespace.

Return type dict

get_proxy(*by_kind=True*)
Return a structured proxy object for easy exploration of services.

This method returns a proxy object to the registry upon which the *Duct* instances are available as attributes. This object is also by default structured such that one first accesses an attribute associated with a kind, which makes larger collections of services more easily navigatable.

For example, if you have *DatabaseClient* subclass registered as 'my_service', you could access it on the proxy using: >>> proxy = registry.get_proxy(by_kind=True) >>> proxy.databases.my_service

Parameters **by_kind** (*bool*) – Whether to nest proxy of *Duct* instances by kind.

Returns The proxy object.

Return type *ServicesProxy*

register_from_config(*config*)
Register a collection of Duct service configurations.

The configuration format must be one of the following: - An iterable sequence of dictionaries containing a mapping between the

keyword arguments required to instantiate the *Duct* subclass.

- A dictionary mapping names of *Duct* instances to dictionaries of keyword arguments.
- A dictionary mapping Duct types ('databases', 'filesystems', etc) to mappings like those immediately above.
- A string YAML representation of one of the above (with at least one newline character).
- A string filename containing such a YAML representation.

There are three special keyword arguments that are required by the *DuctRegistry* instance: - name: Should be present only in the configuration dictionary when

config is provided as an iterable sequence of dictionaries.

- protocol: Which specifies which *Duct* subclass to fetch. Failure to correctly set this will result in a warning and an ignoring of this configuration.
- register_magics (optional): A boolean flag indicating whether to register any magics defined by this Duct class (default: True).

Parameters **`config`** (*`iterable, dict, str, None`*) – A configuration specified in one of the above described formats.

Omniduct's API has been designed to ensure that ducts which provide the same type of service (i.e. database querying, filesystem grokking, etc) also provide a programmatically similar API. As such, all protocol implementations are subclasses of a generic abstract class *Duct* via a protocol type-specific subclass (such as *DatabaseClient* for database protocols). This ensures that the core API is consistent between all instances of the same protocol type. These type-specific classes may also derive from *omniduct.utils.magics.MagicsProvider*, and provide IPython magic functions to provide convenient access to these protocols in IPython sessions. Protocol implementations can also have protocol-specific additions to the core API.

The *Duct* class provides the scaffolding for connection management and other "magic" such as the automatic creation of a registry of the protocols handled by subclasses. This class is described in more detail in *Core Classes*, along with the *MagicsProvider* class.

The protocol-specific subclasses of *Duct* that provide the shared APIs (including any IPython magics) for each protocol type are detailed in dedicated pages; i.e. *Databases*, *Filesystems*, *Remotes*, and *Caches*.

Lastly, utility classes and methods are provided to help manage registries of connections to various services. These are documented in *Registry Management*.

Note Omniduct does not guarantee a stable API between major versions. However, we do commit to ensuring that version *x.y.z* of *omniduct* is API forward-compatible with all future minor versions *x.y.**. While there is no guarantee of APIs remaining fixed between major versions, we expect that in practice these breaking API changes will be small, and in all cases will be documented in the release notes. As such, if you are using Omniduct in a production environment, we recommend installing using a static pinned version or something like *omniduct>=1.2.3<1.3*, where 1.2.3 is the version found to work well in your environment.

CHAPTER 6

Extensions and Plug-ins

Extending Omniduct to support additional services is relatively straightforward, requiring you only to subclass *Duct* or one of the protocol specific common API subclasses (a template for each of these is provided as a *stub.py* file in the appropriate subpackage, e.g. https://github.com/airbnb/omniduct/blob/master/omniduct/databases/stub.py).

As soon as your subclass is in memory, it will integrate automatically with the rest of the Omniduct ecosystem, and be instantiatable by protocol name through the *DuctRegistry* or *Duct.for_protocol()* systems.

If you would like to contribute this extension into the upstream Omniduct library, we welcome your contribution. This would entail simply adding a module containing your subclass to the appropriate Omniduct subpackage, and then (if it is stable and ready for broad usage) importing that subpackage from *omniduct.protocols*. Once your module is merged into the master branch of Omniduct, maintainance will fall to the core Omniduct maintainers, though you are of course welcome to continue submitting patches to improve it or any other aspect of Omniduct.

If you need further assistance, please do not hesitate to open an issue on our issue tracker: https://github.com/airbnb/omniduct/issues .

CHAPTER 7

Contributions

Contributions of any nature are welcome, including software patches, improvements to documentation, bug reports, or feature requests. One of the most useful contributions will undoubtedly be support for new protocols, and so we look forward to seeing your patches to support your favourite services.

For documentation on how to contribute support for new protocols, please refer to *Extensions and Plug-ins*.

Omniduct is an extensible Python library that provides uniform interfaces to a wide variety of (potentially) remote data providers such as databases, filesystems, and REST services. Its primary objective is to simplify the process of collecting and analysing data in a heterogeneous data environment, and is suitable for deployment in interactive and production environments. To that end, it offers the following features:

- A generic plugin-based programmatic API to access data in a consistent manner across different services (see *Supported protocols*).
- A framework for lazily connecting to data sources and maintaining these connections during the entire lifetime of the relevant Python session.
- Automatic port forwarding of remote services over SSH where connections cannot be made directly.
- Convenient IPython magic functions for interfacing with data providers from within IPython and Jupyter Notebook sessions.
- Utility classes and methods to assist in maintaining registries of useful services.

Omniduct has been designed such that it is convenient to use directly (each user can configure their own service definitions) or via another package (which can create a library of pre-defined services, such as for a company). For more information on how to deploy *omniduct* refer to *Deployment*.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

omniduct.caches.base, 150
omniduct.filesystems.base, 91
omniduct.registry, 161

Index

Symbols

`__init__()` (*omniduct.caches.base.Cache method*), 151
`__init__()` (*omniduct.caches.filesystem.FileSystemCache method*), 156
`__init__()` (*omniduct.databases.base.DatabaseClient method*), 16
`__init__()` (*omniduct.databases.druid.DruidClient method*), 28
`__init__()` (*omniduct.databases.hiveserver2.HiveServer2Client method*), 38
`__init__()` (*omniduct.databases.neo4j.Neo4jClient method*), 50
`__init__()` (*omniduct.databases.presto.PrestoClient method*), 60
`__init__()` (*omniduct.databases.pyspark.PySparkClient method*), 71
`__init__()` (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 82
`__init__()` (*omniduct.duct.Duct method*), 12
`__init__()` (*omniduct.filesystems.base.FileSystemClient method*), 92
`__init__()` (*omniduct.filesystems.base.FileSystemFile method*), 98
`__init__()` (*omniduct.filesystems.local.LocalFsClient method*), 99
`__init__()` (*omniduct.filesystems.s3.S3Client method*), 107
`__init__()` (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 115
`__init__()` (*omniduct.registry.DuctRegistry method*), 161
`__init__()` (*omniduct.registry.DuctRegistry.ServicesProxy method*), 161
`__init__()` (*omniduct.remotes.base.RemoteClient method*), 123
`__init__()` (*omniduct.remotes.ssh.SSHClient method*), 130
`__init__()` (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 141
`_prepare()` (*omniduct.duct.Duct method*), 13

C

`Cache` (*class in omniduct.caches.base*), 151
`cached_method()` (*in module omniduct.caches.base*), 150
`connect()` (*omniduct.caches.base.Cache method*), 154
`connect()` (*omniduct.caches.filesystem.FileSystemCache method*), 157
`connect()` (*omniduct.databases.base.DatabaseClient method*), 23
`connect()` (*omniduct.databases.druid.DruidClient method*), 28
`connect()` (*omniduct.databases.hiveserver2.HiveServer2Client method*), 39
`connect()` (*omniduct.databases.neo4j.Neo4jClient method*), 50
`connect()` (*omniduct.databases.presto.PrestoClient method*), 61
`connect()` (*omniduct.databases.pyspark.PySparkClient method*), 72
`connect()` (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 82
`connect()` (*omniduct.duct.Duct method*), 14
`connect()` (*omniduct.filesystems.base.FileSystemClient method*), 97
`connect()` (*omniduct.filesystems.local.LocalFsClient method*), 100
`connect()` (*omniduct.filesystems.s3.S3Client method*), 108
`connect()` (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 116
`connect()` (*omniduct.remotes.base.RemoteClient method*), 124
`connect()` (*omniduct.remotes.ssh.SSHClient method*), 131
`connect()` (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 142

D

DatabaseClient (*class in omniduct.databases.base*), 15
dataframe_to_table() (*omniduct.databases.base.DatabaseClient method*), 21
dataframe_to_table() (*omniduct.databases.druid.DruidClient method*), 28
dataframe_to_table() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 39
dataframe_to_table() (*omniduct.databases.neo4j.Neo4jClient method*), 51
dataframe_to_table() (*omniduct.databases.presto.PrestoClient method*), 61
dataframe_to_table() (*omniduct.databases.pyspark.PySparkClient method*), 72
dataframe_to_table() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 82
describe() (*omniduct.caches.base.Cache method*), 154
describe() (*omniduct.caches.filesystem.FileSystemCache method*), 157
dir() (*omniduct.filesystems.base.FileSystemClient method*), 95
dir() (*omniduct.filesystems.local.LocalFsClient method*), 100
dir() (*omniduct.filesystems.s3.S3Client method*), 108
dir() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 116
dir() (*omniduct.remotes.base.RemoteClient method*), 126
dir() (*omniduct.remotes.ssh.SSHClient method*), 131
dir() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 142
disconnect() (*omniduct.caches.base.Cache method*), 155
disconnect() (*omniduct.caches.filesystem.FileSystemCache method*), 157
disconnect() (*omniduct.databases.base.DatabaseClient method*), 23
disconnect() (*omniduct.databases.druid.DruidClient method*), 29
disconnect() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 40
disconnect() (*omniduct.databases.neo4j.Neo4jClient method*), 51
disconnect() (*omniduct.databases.presto.PrestoClient method*), 62
disconnect() (*omniduct.databases.pyspark.PySparkClient method*), 72
disconnect() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 83
disconnect() (*omniduct.duct.Duct method*), 14
disconnect() (*omniduct.filesystems.base.FileSystemClient method*), 97
disconnect() (*omniduct.filesystems.local.LocalFsClient method*), 100
disconnect() (*omniduct.filesystems.s3.S3Client method*), 108
disconnect() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 116
disconnect() (*omniduct.remotes.base.RemoteClient method*), 126
disconnect() (*omniduct.remotes.ssh.SSHClient method*), 132
disconnect() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 142
download() (*omniduct.filesystems.base.FileSystemClient method*), 96
download() (*omniduct.filesystems.local.LocalFsClient method*), 101
download() (*omniduct.filesystems.s3.S3Client method*), 109
download() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 116
download() (*omniduct.remotes.base.RemoteClient method*), 126
download() (*omniduct.remotes.ssh.SSHClient method*), 132
download() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 143
DruidClient (*class in omniduct.databases.druid*), 27
DruidClient.Type (*class in omniduct.databases.druid*), 28
Duct (*class in omniduct.duct*), 11
Duct.Type (*class in omniduct.duct*), 12
DuctRegistry (*class in omniduct.registry*), 161
DuctRegistry.ServicesProxy (*class in omniduct.registry*), 161

E

execute() (*omniduct.databases.base.DatabaseClient method*), 17
execute() (*omniduct.databases.druid.DruidClient method*), 29
execute() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 40
execute() (*omniduct.databases.neo4j.Neo4jClient method*), 51
execute() (*omniduct.databases.presto.PrestoClient method*), 62
execute() (*omniduct.databases.pyspark.PySparkClient method*), 72
execute() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 83
execute() (*omniduct.remotes.base.RemoteClient method*), 124
execute() (*omniduct.remotes.ssh.SSHClient method*), 133
execute() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 143
execute_from_file() (*omniduct.databases.base.DatabaseClient method*), 19
execute_from_file() (*omniduct.databases.druid.DruidClient method*), 29
execute_from_file() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 41
execute_from_file() (*omniduct.databases.neo4j.Neo4jClient method*), 52
execute_from_file() (*omniduct.databases.presto.PrestoClient method*), 62
execute_from_file() (*omniduct.databases.pyspark.PySparkClient method*), 73
execute_from_file() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 84
execute_from_template() (*omniduct.databases.base.DatabaseClient method*), 21
execute_from_template() (*omniduct.databases.druid.DruidClient method*), 30
execute_from_template() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 41
execute_from_template() (*omniduct.databases.neo4j.Neo4jClient method*), 52
execute_from_template() (*omniduct.databases.presto.PrestoClient method*), 63
execute_from_template() (*omniduct.databases.pyspark.PySparkClient method*), 73
execute_from_template() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 84
exists() (*omniduct.filesystems.base.FileSystemClient method*), 94
exists() (*omniduct.filesystems.local.LocalFsClient method*), 101
exists() (*omniduct.filesystems.s3.S3Client method*), 109
exists() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 117
exists() (*omniduct.remotes.base.RemoteClient method*), 127
exists() (*omniduct.remotes.ssh.SSHClient method*), 133
exists() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 143

F

FileSystemCache (*class in omniduct.caches.filesystem*), 155
FileSystemCache.Type (*class in omniduct.caches.filesystem*), 156
FileSystemClient (*class in omniduct.filesystems.base*), 91
FileSystemFile (*class in omniduct.filesystems.base*), 98
FileSystemFileDesc (*class in omniduct.filesystems.base*), 98
find() (*omniduct.filesystems.base.FileSystemClient method*), 95
find() (*omniduct.filesystems.local.LocalFsClient method*), 101
find() (*omniduct.filesystems.s3.S3Client method*), 109
find() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 117
find() (*omniduct.remotes.base.RemoteClient method*), 127
find() (*omniduct.remotes.ssh.SSHClient method*), 133
find() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 143
for_protocol() (*omniduct.caches.filesystem.FileSystemCache class method*), 157
for_protocol() (*omniduct.databases.druid.DruidClient class method*), 30

for_protocol() (*omniduct.databases.hiveserver2.HiveServer2Client class method*), 42
for_protocol() (*omniduct.databases.neo4j.Neo4jClient class method*), 52
for_protocol() (*omniduct.databases.presto.PrestoClient class method*), 63
for_protocol() (*omniduct.databases.pyspark.PySparkClient class method*), 74
for_protocol() (*omniduct.databases.sqlalchemy.SQLAlchemyClient class method*), 84
for_protocol() (*omniduct.duct.Duct class method*), 13
for_protocol() (*omniduct.filesystems.local.LocalFsClient class method*), 101
for_protocol() (*omniduct.filesystems.s3.S3Client class method*), 109
for_protocol() (*omniduct.filesystems.webhdfs.WebHdfsClient class method*), 117
for_protocol() (*omniduct.remotes.ssh.SSHClient class method*), 133
for_protocol() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient class method*), 144

G

get() (*omniduct.caches.base.Cache method*), 152
get() (*omniduct.caches.filesystem.FileSystemCache method*), 157
get_bytecount() (*omniduct.caches.base.Cache method*), 153
get_bytecount() (*omniduct.caches.filesystem.FileSystemCache method*), 158
get_local_uri() (*omniduct.remotes.base.RemoteClient method*), 125
get_local_uri() (*omniduct.remotes.ssh.SSHClient method*), 134
get_local_uri() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 144
get_metadata() (*omniduct.caches.base.Cache method*), 153
get_metadata() (*omniduct.caches.filesystem.FileSystemCache method*), 158
get_proxy() (*omniduct.registry.DuctRegistry method*), 163
get_total_bytecount() (*omniduct.caches.base.Cache method*), 154
get_total_bytecount() (*omniduct.caches.filesystem.FileSystemCache method*), 158
global_writes (*omniduct.filesystems.base.FileSystemClient attribute*), 94
global_writes (*omniduct.filesystems.local.LocalFsClient attribute*), 102
global_writes (*omniduct.filesystems.s3.S3Client attribute*), 110
global_writes (*omniduct.filesystems.webhdfs.WebHdfsClient attribute*), 118
global_writes (*omniduct.remotes.ssh.SSHClient attribute*), 134
global_writes (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient attribute*), 144

H

has_key() (*omniduct.caches.base.Cache method*), 153
has_key() (*omniduct.caches.filesystem.FileSystemCache method*), 158
has_namespace() (*omniduct.caches.base.Cache method*), 153
has_namespace() (*omniduct.caches.filesystem.FileSystemCache method*), 159
has_port_forward() (*omniduct.remotes.base.RemoteClient method*), 125
has_port_forward() (*omniduct.remotes.ssh.SSHClient method*), 134
has_port_forward() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 144
HiveServer2Client (*class in omniduct.databases.hiveserver2*), 37
HiveServer2Client.Type (*class in omniduct.databases.hiveserver2*), 38
host (*omniduct.caches.filesystem.FileSystemCache attribute*), 159
host (*omniduct.databases.druid.DruidClient attribute*), 30
host (*omniduct.databases.hiveserver2.HiveServer2Client attribute*), 42
host (*omniduct.databases.neo4j.Neo4jClient attribute*), 52

host (*omniduct.databases.presto.PrestoClient attribute*), 63
host (*omniduct.databases.pyspark.PySparkClient attribute*), 74
host (*omniduct.databases.sqlalchemy.SQLAlchemyClient attribute*), 84
host (*omniduct.duct.Duct attribute*), 14
host (*omniduct.filesystems.local.LocalFsClient attribute*), 102
host (*omniduct.filesystems.s3.S3Client attribute*), 110
host (*omniduct.filesystems.webhdfs.WebHdfsClient attribute*), 118
host (*omniduct.remotes.ssh.SSHClient attribute*), 134
host (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient attribute*), 144

I

is_connected() (*omniduct.caches.base.Cache method*), 155
is_connected() (*omniduct.caches.filesystem.FileSystemCache method*), 159
is_connected() (*omniduct.databases.base.DatabaseClient method*), 23
is_connected() (*omniduct.databases.druid.DruidClient method*), 30
is_connected() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 42
is_connected() (*omniduct.databases.neo4j.Neo4jClient method*), 53
is_connected() (*omniduct.databases.presto.PrestoClient method*), 63
is_connected() (*omniduct.databases.pyspark.PySparkClient method*), 74
is_connected() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 84
is_connected() (*omniduct.duct.Duct method*), 14
is_connected() (*omniduct.filesystems.base.FileSystemClient method*), 97
is_connected() (*omniduct.filesystems.local.LocalFsClient method*), 102
is_connected() (*omniduct.filesystems.s3.S3Client method*), 110
is_connected() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 118
is_connected() (*omniduct.remotes.base.RemoteClient method*), 127
is_connected() (*omniduct.remotes.ssh.SSHClient method*), 134
is_connected() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 144
is_port_bound() (*omniduct.remotes.base.RemoteClient method*), 126
is_port_bound() (*omniduct.remotes.ssh.SSHClient method*), 134
is_port_bound() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 145
isdir() (*omniduct.filesystems.base.FileSystemClient method*), 94
isdir() (*omniduct.filesystems.local.LocalFsClient method*), 102
isdir() (*omniduct.filesystems.s3.S3Client method*), 110
isdir() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 118
isdir() (*omniduct.remotes.base.RemoteClient method*), 127
isdir() (*omniduct.remotes.ssh.SSHClient method*), 135
isdir() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 145
isfile() (*omniduct.filesystems.base.FileSystemClient method*), 94
isfile() (*omniduct.filesystems.local.LocalFsClient method*), 102
isfile() (*omniduct.filesystems.s3.S3Client method*), 110
isfile() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 118
isfile() (*omniduct.remotes.base.RemoteClient method*), 127
isfile() (*omniduct.remotes.ssh.SSHClient method*), 135
isfile() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 145

K

keys() (*omniduct.caches.base.Cache method*), 153
keys() (*omniduct.caches.filesystem.FileSystemCache method*), 159

L

listdir() (*omniduct.filesystems.base.FileSystemClient method*), 95

listdir() (*omniduct.filesystems.local.LocalFsClient method*), 102
listdir() (*omniduct.filesystems.s3.S3Client method*), 110
listdir() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 118
listdir() (*omniduct.remotes.ssh.SSHClient method*), 135
listdir() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 145
LocalFsClient (*class in omniduct.filesystems.local*), 98
LocalFsClient.Type (*class in omniduct.filesystems.local*), 99
lookup() (*omniduct.registry.DuctRegistry method*), 162

M

MagicsProvider (*class in omniduct.utils.magics*), 15
mkdir() (*omniduct.filesystems.base.FileSystemClient method*), 96
mkdir() (*omniduct.filesystems.local.LocalFsClient method*), 102
mkdir() (*omniduct.filesystems.s3.S3Client method*), 110
mkdir() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 118
mkdir() (*omniduct.remotes.base.RemoteClient method*), 127
mkdir() (*omniduct.remotes.ssh.SSHClient method*), 135
mkdir() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 145

N

names (*omniduct.registry.DuctRegistry attribute*), 162
namespaces (*omniduct.caches.base.Cache attribute*), 153
namespaces (*omniduct.caches.filesystem.FileSystemCache attribute*), 159
Neo4jClient (*class in omniduct.databases.neo4j*), 49
Neo4jClient.Type (*class in omniduct.databases.neo4j*), 50
new() (*omniduct.registry.DuctRegistry method*), 162

O

omniduct.caches.base (*module*), 150
omniduct.filesystems.base (*module*), 91
omniduct.registry (*module*), 161
open() (*omniduct.filesystems.base.FileSystemClient method*), 96
open() (*omniduct.filesystems.local.LocalFsClient method*), 103
open() (*omniduct.filesystems.s3.S3Client method*), 111
open() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 119
open() (*omniduct.remotes.base.RemoteClient method*), 128
open() (*omniduct.remotes.ssh.SSHClient method*), 135
open() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 146

P

ParamikoSSHClient (*class in omniduct.remotes.ssh_paramiko*), 140
ParamikoSSHClient.Type (*class in omniduct.remotes.ssh_paramiko*), 141
password (*omniduct.caches.filesystem.FileSystemCache attribute*), 159
password (*omniduct.databases.druid.DruidClient attribute*), 30
password (*omniduct.databases.hiveserver2.HiveServer2Client attribute*), 42
password (*omniduct.databases.neo4j.Neo4jClient attribute*), 53
password (*omniduct.databases.presto.PrestoClient attribute*), 63
password (*omniduct.databases.pyspark.PySparkClient attribute*), 74
password (*omniduct.databases.sqlalchemy.SQLAlchemyClient attribute*), 84
password (*omniduct.duct.Duct attribute*), 14
password (*omniduct.filesystems.local.LocalFsClient attribute*), 103
password (*omniduct.filesystems.s3.S3Client attribute*), 111
password (*omniduct.filesystems.webhdfs.WebHdfsClient attribute*), 119
password (*omniduct.remotes.ssh.SSHClient attribute*), 136
password (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient attribute*), 146
path_basename() (*omniduct.filesystems.base.FileSystemClient method*), 93
path_basename() (*omniduct.filesystems.local.LocalFsClient method*), 103
path_basename() (*omniduct.filesystems.s3.S3Client method*), 111
path_basename() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 119
path_basename() (*omniduct.remotes.ssh.SSHClient method*), 136
path_basename() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 146

path_cwd (*omniduct.filesystems.base.FileSystemClient attribute*), 93
path_cwd (*omniduct.filesystems.local.LocalFsClient attribute*), 103
path_cwd (*omniduct.filesystems.s3.S3Client attribute*), 111
path_cwd (*omniduct.filesystems.webhdfs.WebHdfsClient attribute*), 119
path_cwd (*omniduct.remotes.ssh.SSHClient attribute*), 136
path_cwd (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient attribute*), 146
path_dirname() (*omniduct.filesystems.base.FileSystemClient method*), 94
path_dirname() (*omniduct.filesystems.local.LocalFsClient method*), 103
path_dirname() (*omniduct.filesystems.s3.S3Client method*), 111
path_dirname() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 119
path_dirname() (*omniduct.remotes.ssh.SSHClient method*), 136
path_dirname() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 146
path_home (*omniduct.filesystems.base.FileSystemClient attribute*), 93
path_home (*omniduct.filesystems.local.LocalFsClient attribute*), 103
path_home (*omniduct.filesystems.s3.S3Client attribute*), 111
path_home (*omniduct.filesystems.webhdfs.WebHdfsClient attribute*), 119
path_home (*omniduct.remotes.base.RemoteClient attribute*), 128
path_home (*omniduct.remotes.ssh.SSHClient attribute*), 136
path_home (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient attribute*), 146
path_join() (*omniduct.filesystems.base.FileSystemClient method*), 93
path_join() (*omniduct.filesystems.local.LocalFsClient method*), 104
path_join() (*omniduct.filesystems.s3.S3Client method*), 112
path_join() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 120
path_join() (*omniduct.remotes.ssh.SSHClient method*), 136
path_join() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 147
path_normpath() (*omniduct.filesystems.base.FileSystemClient method*), 94
path_normpath() (*omniduct.filesystems.local.LocalFsClient method*), 104
path_normpath() (*omniduct.filesystems.s3.S3Client method*), 112
path_normpath() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 120
path_normpath() (*omniduct.remotes.ssh.SSHClient method*), 137
path_normpath() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 147
path_separator (*omniduct.filesystems.base.FileSystemClient attribute*), 93
path_separator (*omniduct.filesystems.local.LocalFsClient attribute*), 104
path_separator (*omniduct.filesystems.s3.S3Client attribute*), 112
path_separator (*omniduct.filesystems.webhdfs.WebHdfsClient attribute*), 120
path_separator (*omniduct.remotes.base.RemoteClient attribute*), 128
path_separator (*omniduct.remotes.ssh.SSHClient attribute*), 137
path_separator (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient attribute*), 147
populate_namespace() (*omniduct.registry.DuctRegistry method*), 163
port (*omniduct.caches.filesystem.FileSystemCache attribute*), 159
port (*omniduct.databases.druid.DruidClient attribute*), 30
port (*omniduct.databases.hiveserver2.HiveServer2Client attribute*), 42
port (*omniduct.databases.neo4j.Neo4jClient attribute*), 53
port (*omniduct.databases.presto.PrestoClient attribute*), 64
port (*omniduct.databases.pyspark.PySparkClient attribute*), 74
port (*omniduct.databases.sqlalchemy.SQLAlchemyClient attribute*), 85
port (*omniduct.duct.Duct attribute*), 14
port (*omniduct.filesystems.local.LocalFsClient attribute*), 104

port (*omniduct.filesystems.s3.S3Client attribute*), 112
port (*omniduct.filesystems.webhdfs.WebHdfsClient attribute*), 120
port (*omniduct.remotes.ssh.SSHClient attribute*), 137
port (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient attribute*), 147
port_forward() (*omniduct.remotes.base.RemoteClient method*), 124
port_forward() (*omniduct.remotes.ssh.SSHClient method*), 137
port_forward() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 147
port_forward_stop() (*omniduct.remotes.base.RemoteClient method*), 125
port_forward_stop() (*omniduct.remotes.ssh.SSHClient method*), 137
port_forward_stop() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 148
port_forward_stopall() (*omniduct.remotes.base.RemoteClient method*), 125
port_forward_stopall() (*omniduct.remotes.ssh.SSHClient method*), 137
port_forward_stopall() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 148
prepare() (*omniduct.caches.base.Cache method*), 155
prepare() (*omniduct.caches.filesystem.FileSystemCache method*), 159
prepare() (*omniduct.databases.base.DatabaseClient method*), 23
prepare() (*omniduct.databases.druid.DruidClient method*), 31
prepare() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 42
prepare() (*omniduct.databases.neo4j.Neo4jClient method*), 53
prepare() (*omniduct.databases.presto.PrestoClient method*), 64
prepare() (*omniduct.databases.pyspark.PySparkClient method*), 74
prepare() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 85
prepare() (*omniduct.duct.Duct method*), 13
prepare() (*omniduct.filesystems.base.FileSystemClient method*), 98
prepare() (*omniduct.filesystems.local.LocalFsClient method*), 104
prepare() (*omniduct.filesystems.s3.S3Client method*), 112
prepare() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 120
prepare() (*omniduct.remotes.base.RemoteClient method*), 128
prepare() (*omniduct.remotes.ssh.SSHClient method*), 137
prepare() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 148
prepare_smartcards() (*omniduct.remotes.base.RemoteClient method*), 124
prepare_smartcards() (*omniduct.remotes.ssh.SSHClient method*), 138
prepare_smartcards() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 148
PrestoClient (*class in omniduct.databases.presto*), 59
PrestoClient.Type (*class in omniduct.databases.presto*), 60
prune() (*omniduct.caches.base.Cache method*), 154
prune() (*omniduct.caches.filesystem.FileSystemCache method*), 160
PySparkClient (*class in omniduct.databases.pyspark*), 70
PySparkClient.Type (*class in omniduct.databases.pyspark*), 71

Q

query() (*omniduct.databases.base.DatabaseClient method*), 18
query() (*omniduct.databases.druid.DruidClient method*), 31
query() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 43
query() (*omniduct.databases.neo4j.Neo4jClient method*), 53
query() (*omniduct.databases.presto.PrestoClient method*), 64
query() (*omniduct.databases.pyspark.PySparkClient method*), 75
query() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 85
query_from_file() (*omniduct.databases.base.DatabaseClient method*), 19
query_from_file() (*omniduct.databases.druid.DruidClient method*), 31
query_from_file() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 43
query_from_file() (*omniduct.databases.neo4j.Neo4jClient method*),

54
query_from_file() (omniduct.databases.presto.PrestoClient method), 64
query_from_file() (omniduct.databases.pyspark.PySparkClient method), 75
query_from_file() (omniduct.databases.sqlalchemy.SQLAlchemyClient method), 85
query_from_template() (omniduct.databases.base.DatabaseClient method), 21
query_from_template() (omniduct.databases.druid.DruidClient method), 32
query_from_template() (omniduct.databases.hiveserver2.HiveServer2Client method), 43
query_from_template() (omniduct.databases.neo4j.Neo4jClient method), 54
query_from_template() (omniduct.databases.presto.PrestoClient method), 65
query_from_template() (omniduct.databases.pyspark.PySparkClient method), 75
query_from_template() (omniduct.databases.sqlalchemy.SQLAlchemyClient method), 86
query_to_table() (omniduct.databases.base.DatabaseClient method), 21
query_to_table() (omniduct.databases.druid.DruidClient method), 32
query_to_table() (omniduct.databases.hiveserver2.HiveServer2Client method), 44
query_to_table() (omniduct.databases.neo4j.Neo4jClient method), 54
query_to_table() (omniduct.databases.presto.PrestoClient method), 65
query_to_table() (omniduct.databases.pyspark.PySparkClient method), 75
query_to_table() (omniduct.databases.sqlalchemy.SQLAlchemyClient method), 86

R

read_only (omniduct.filesystems.base.FileSystemClient attribute), 94
read_only (omniduct.filesystems.local.LocalFsClient attribute), 104
read_only (omniduct.filesystems.s3.S3Client attribute), 113
read_only (omniduct.filesystems.webhdfs.WebHdfsClient attribute), 121
read_only (omniduct.remotes.ssh.SSHClient attribute), 138
read_only (omniduct.remotes.ssh_paramiko.ParamikoSSHClient attribute), 148
reconnect() (omniduct.caches.filesystem.FileSystemCache method), 160
reconnect() (omniduct.databases.druid.DruidClient method), 32
reconnect() (omniduct.databases.hiveserver2.HiveServer2Client method), 44
reconnect() (omniduct.databases.neo4j.Neo4jClient method), 54
reconnect() (omniduct.databases.presto.PrestoClient method), 65
reconnect() (omniduct.databases.pyspark.PySparkClient method), 76
reconnect() (omniduct.databases.sqlalchemy.SQLAlchemyClient method), 86
reconnect() (omniduct.duct.Duct method), 15
reconnect() (omniduct.filesystems.local.LocalFsClient method), 104
reconnect() (omniduct.filesystems.s3.S3Client method), 113
reconnect() (omniduct.filesystems.webhdfs.WebHdfsClient method), 121
reconnect() (omniduct.remotes.ssh.SSHClient method), 138
reconnect() (omniduct.remotes.ssh_paramiko.ParamikoSSHClient method), 148
register() (omniduct.registry.DuctRegistry method), 162
register_from_config() (omniduct.registry.DuctRegistry method), 163
register_magics() (omniduct.databases.base.DatabaseClient method), 24
register_magics() (omniduct.databases.druid.DruidClient method), 32
register_magics() (omniduct.databases.hiveserver2.HiveServer2Client method), 44
register_magics() (omniduct.databases.neo4j.Neo4jClient method), 54

register_magics() (*omniduct.databases.presto.PrestoClient method*), 65
register_magics() (*omniduct.databases.pyspark.PySparkClient method*), 76
register_magics() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 86
registry (*omniduct.registry.DuctRegistry.ServicesProxy attribute*), 161
RemoteClient (*class in omniduct.remotes.base*), 122
remove() (*omniduct.filesystems.base.FileSystemClient method*), 96
remove() (*omniduct.filesystems.local.LocalFsClient method*), 105
remove() (*omniduct.filesystems.s3.S3Client method*), 113
remove() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 121
remove() (*omniduct.remotes.base.RemoteClient method*), 129
remove() (*omniduct.remotes.ssh.SSHClient method*), 138
remove() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 149
reset() (*omniduct.caches.filesystem.FileSystemCache method*), 160
reset() (*omniduct.databases.druid.DruidClient method*), 33
reset() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 44
reset() (*omniduct.databases.neo4j.Neo4jClient method*), 55
reset() (*omniduct.databases.presto.PrestoClient method*), 66
reset() (*omniduct.databases.pyspark.PySparkClient method*), 76
reset() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 87
reset() (*omniduct.duct.Duct method*), 13
reset() (*omniduct.filesystems.local.LocalFsClient method*), 105
reset() (*omniduct.filesystems.s3.S3Client method*), 113
reset() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 121
reset() (*omniduct.remotes.ssh.SSHClient method*), 138
reset() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 149

S

S3Client (*class in omniduct.filesystems.s3*), 106
S3Client.Type (*class in omniduct.filesystems.s3*), 107
schemas (*omniduct.databases.hiveserver2.HiveServer2Client attribute*), 45
schemas (*omniduct.databases.presto.PrestoClient attribute*), 66
schemas (*omniduct.databases.sqlalchemy.SQLAlchemyClient attribute*), 87
session_properties (*omniduct.databases.base.DatabaseClient attribute*), 17
session_properties (*omniduct.databases.druid.DruidClient attribute*), 33
session_properties (*omniduct.databases.hiveserver2.HiveServer2Client attribute*), 45
session_properties (*omniduct.databases.neo4j.Neo4jClient attribute*), 55
session_properties (*omniduct.databases.presto.PrestoClient attribute*), 66
session_properties (*omniduct.databases.pyspark.PySparkClient attribute*), 77
session_properties (*omniduct.databases.sqlalchemy.SQLAlchemyClient attribute*), 87
set() (*omniduct.caches.base.Cache method*), 152
set() (*omniduct.caches.filesystem.FileSystemCache method*), 160
set_metadata() (*omniduct.caches.base.Cache method*), 152
set_metadata() (*omniduct.caches.filesystem.FileSystemCache method*), 160
show_port_forwards() (*omniduct.remotes.base.RemoteClient method*), 125
show_port_forwards() (*omniduct.remotes.ssh.SSHClient method*), 139
show_port_forwards() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 149
showdir() (*omniduct.filesystems.base.FileSystemClient method*), 95
showdir() (*omniduct.filesystems.local.LocalFsClient method*), 105
showdir() (*omniduct.filesystems.s3.S3Client method*), 113
showdir() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 121
showdir() (*omniduct.remotes.ssh.SSHClient method*),

139
showdir() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 149
SQLAlchemyClient (*class in omniduct.databases.sqlalchemy*), 81
SQLAlchemyClient.Type (*class in omniduct.databases.sqlalchemy*), 82
SSHClient (*class in omniduct.remotes.ssh*), 129
SSHClient.Type (*class in omniduct.remotes.ssh*), 130
statement_cleanup() (*omniduct.databases.base.DatabaseClient class method*), 17
statement_cleanup() (*omniduct.databases.druid.DruidClient class method*), 33
statement_cleanup() (*omniduct.databases.hiveserver2.HiveServer2Client class method*), 45
statement_cleanup() (*omniduct.databases.neo4j.Neo4jClient class method*), 55
statement_cleanup() (*omniduct.databases.presto.PrestoClient class method*), 66
statement_cleanup() (*omniduct.databases.pyspark.PySparkClient class method*), 77
statement_cleanup() (*omniduct.databases.sqlalchemy.SQLAlchemyClient class method*), 87
statement_hash() (*omniduct.databases.base.DatabaseClient class method*), 17
statement_hash() (*omniduct.databases.druid.DruidClient class method*), 33
statement_hash() (*omniduct.databases.hiveserver2.HiveServer2Client class method*), 45
statement_hash() (*omniduct.databases.neo4j.Neo4jClient class method*), 55
statement_hash() (*omniduct.databases.presto.PrestoClient class method*), 66
statement_hash() (*omniduct.databases.pyspark.PySparkClient class method*), 77
statement_hash() (*omniduct.databases.sqlalchemy.SQLAlchemyClient class method*), 87
stream() (*omniduct.databases.base.DatabaseClient method*), 18
stream() (*omniduct.databases.druid.DruidClient method*), 33
stream() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 45
stream() (*omniduct.databases.neo4j.Neo4jClient method*), 56
stream() (*omniduct.databases.presto.PrestoClient method*), 66
stream() (*omniduct.databases.pyspark.PySparkClient method*), 77
stream() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 88
stream_to_file() (*omniduct.databases.base.DatabaseClient method*), 18
stream_to_file() (*omniduct.databases.druid.DruidClient method*), 34
stream_to_file() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 46
stream_to_file() (*omniduct.databases.neo4j.Neo4jClient method*), 56
stream_to_file() (*omniduct.databases.presto.PrestoClient method*), 67
stream_to_file() (*omniduct.databases.pyspark.PySparkClient method*), 77
stream_to_file() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 88

T

table_desc() (*omniduct.databases.base.DatabaseClient method*), 22
table_desc() (*omniduct.databases.druid.DruidClient method*), 34
table_desc() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 46
table_desc() (*omniduct.databases.neo4j.Neo4jClient method*), 56
table_desc() (*omniduct.databases.presto.PrestoClient method*), 67
table_desc() (*omniduct.databases.pyspark.PySparkClient method*), 78

table_desc() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 88
table_drop() (*omniduct.databases.base.DatabaseClient method*), 22
table_drop() (*omniduct.databases.druid.DruidClient method*), 34
table_drop() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 46
table_drop() (*omniduct.databases.neo4j.Neo4jClient method*), 57
table_drop() (*omniduct.databases.presto.PrestoClient method*), 67
table_drop() (*omniduct.databases.pyspark.PySparkClient method*), 78
table_drop() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 88
table_exists() (*omniduct.databases.base.DatabaseClient method*), 22
table_exists() (*omniduct.databases.druid.DruidClient method*), 34
table_exists() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 46
table_exists() (*omniduct.databases.neo4j.Neo4jClient method*), 57
table_exists() (*omniduct.databases.presto.PrestoClient method*), 68
table_exists() (*omniduct.databases.pyspark.PySparkClient method*), 78
table_exists() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 89
table_head() (*omniduct.databases.base.DatabaseClient method*), 22
table_head() (*omniduct.databases.druid.DruidClient method*), 35
table_head() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 47
table_head() (*omniduct.databases.neo4j.Neo4jClient method*), 57
table_head() (*omniduct.databases.presto.PrestoClient method*), 68
table_head() (*omniduct.databases.pyspark.PySparkClient method*), 78
table_head() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 89
table_list() (*omniduct.databases.base.DatabaseClient method*), 22
table_list() (*omniduct.databases.druid.DruidClient method*), 35
table_list() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 47
table_list() (*omniduct.databases.neo4j.Neo4jClient method*), 57
table_list() (*omniduct.databases.presto.PrestoClient method*), 68
table_list() (*omniduct.databases.pyspark.PySparkClient method*), 79
table_list() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 89
table_props() (*omniduct.databases.base.DatabaseClient method*), 23
table_props() (*omniduct.databases.druid.DruidClient method*), 35
table_props() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 47
table_props() (*omniduct.databases.neo4j.Neo4jClient method*), 57
table_props() (*omniduct.databases.presto.PrestoClient method*), 68
table_props() (*omniduct.databases.pyspark.PySparkClient method*), 79
table_props() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 89

template_add() (*omniduct.databases.base.DatabaseClient method*), 19
template_add() (*omniduct.databases.druid.DruidClient method*), 35
template_add() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 47
template_add() (*omniduct.databases.neo4j.Neo4jClient method*), 58
template_add() (*omniduct.databases.presto.PrestoClient method*), 69
template_add() (*omniduct.databases.pyspark.PySparkClient method*), 79
template_add() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 90
template_get() (*omniduct.databases.base.DatabaseClient method*), 20
template_get() (*omniduct.databases.druid.DruidClient method*), 36
template_get() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 47
template_get() (*omniduct.databases.neo4j.Neo4jClient method*), 58
template_get() (*omniduct.databases.presto.PrestoClient method*), 69
template_get() (*omniduct.databases.pyspark.PySparkClient method*), 79
template_get() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 90
template_names (*omniduct.databases.base.DatabaseClient attribute*), 19
template_names (*omniduct.databases.druid.DruidClient attribute*), 36
template_names (*omniduct.databases.hiveserver2.HiveServer2Client attribute*), 48
template_names (*omniduct.databases.neo4j.Neo4jClient attribute*), 58
template_names (*omniduct.databases.presto.PrestoClient attribute*), 69
template_names (*omniduct.databases.pyspark.PySparkClient attribute*), 79
template_names (*omniduct.databases.sqlalchemy.SQLAlchemyClient attribute*), 90
template_render() (*omniduct.databases.base.DatabaseClient method*), 20
template_render() (*omniduct.databases.druid.DruidClient method*), 36
template_render() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 48
template_render() (*omniduct.databases.neo4j.Neo4jClient method*), 58
template_render() (*omniduct.databases.presto.PrestoClient method*), 69
template_render() (*omniduct.databases.pyspark.PySparkClient method*), 80
template_render() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 90
template_variables() (*omniduct.databases.base.DatabaseClient method*), 20
template_variables() (*omniduct.databases.druid.DruidClient method*), 37
template_variables() (*omniduct.databases.hiveserver2.HiveServer2Client method*), 48
template_variables() (*omniduct.databases.neo4j.Neo4jClient method*), 59
template_variables() (*omniduct.databases.presto.PrestoClient method*), 70
template_variables() (*omniduct.databases.pyspark.PySparkClient method*), 80
template_variables() (*omniduct.databases.sqlalchemy.SQLAlchemyClient method*), 91

U

unset() (*omniduct.caches.base.Cache method*), 153

unset() (*omniduct.caches.filesystem.FileSystemCache method*), 161
unset_namespace() (*omniduct.caches.base.Cache method*), 153
unset_namespace() (*omniduct.caches.filesystem.FileSystemCache method*), 161
update_host_keys() (*omniduct.remotes.ssh.SSHClient method*), 139
upload() (*omniduct.filesystems.base.FileSystemClient method*), 97
upload() (*omniduct.filesystems.local.LocalFsClient method*), 105
upload() (*omniduct.filesystems.s3.S3Client method*), 113
upload() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 121
upload() (*omniduct.remotes.ssh.SSHClient method*), 139
upload() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 149
username (*omniduct.caches.filesystem.FileSystemCache attribute*), 161
username (*omniduct.databases.druid.DruidClient attribute*), 37
username (*omniduct.databases.hiveserver2.HiveServer2Client attribute*), 49
username (*omniduct.databases.neo4j.Neo4jClient attribute*), 59
username (*omniduct.databases.presto.PrestoClient attribute*), 70
username (*omniduct.databases.pyspark.PySparkClient attribute*), 81
username (*omniduct.databases.sqlalchemy.SQLAlchemyClient attribute*), 91
username (*omniduct.duct.Duct attribute*), 14
username (*omniduct.filesystems.local.LocalFsClient attribute*), 105
username (*omniduct.filesystems.s3.S3Client attribute*), 114
username (*omniduct.filesystems.webhdfs.WebHdfsClient attribute*), 122
username (*omniduct.remotes.ssh.SSHClient attribute*), 140
username (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient attribute*), 149

W

walk() (*omniduct.filesystems.base.FileSystemClient method*), 95
walk() (*omniduct.filesystems.local.LocalFsClient method*), 106
walk() (*omniduct.filesystems.s3.S3Client method*), 114
walk() (*omniduct.filesystems.webhdfs.WebHdfsClient method*), 122
walk() (*omniduct.remotes.base.RemoteClient method*), 129
walk() (*omniduct.remotes.ssh.SSHClient method*), 140
walk() (*omniduct.remotes.ssh_paramiko.ParamikoSSHClient method*), 150
WebHdfsClient (*class in omniduct.filesystems.webhdfs*), 114
WebHdfsClient.Type (*class in omniduct.filesystems.webhdfs*), 115