pystardog Documentation

Pedro Oliveira, John Bresnahan, Stephen Nowell

Sep 18, 2023

CONTENTS:

1 pystardog **1**
- 1.1 What is it? . . . 1
- 1.2 Installation . . . 1
- 1.3 Quick Example . . . 1
- 1.4 Interactive Tutorial . . . 2
- 1.5 Documentation . . . 2
- 1.6 Contributing and Development . . . 4

2 Modules **7**
- 2.1 stardog.connection . . . 7
- 2.2 stardog.admin . . . 22
- 2.3 stardog.content . . . 48
- 2.4 stardog.exceptions . . . 52

3 Indices and tables **53**

Python Module Index **55**

Index **57**

CHAPTER

ONE

PYSTARDOG

PyPI version

a Python wrapper for communicating with the Stardog HTTP server.

Docs: http://pystardog.readthedocs.io

Requirements: Python 3.8+

1.1 What is it?

This library wraps all the functionality of a client for the Stardog Knowledge Graph, and provides access to a full set of functions such as executing SPARQL queries and many administrative tasks.

The implementation uses the HTTP protocol, since most of Stardog functionality is available using this protocol. For more information, see HTTP Programming in Stardog's documentation.

1.2 Installation

pystardog is on PyPI. To install:

```
pip install pystardog
```

1.3 Quick Example

```
import stardog

conn_details = {
  'endpoint': 'http://localhost:5820',
  'username': 'admin',
  'password': 'admin'
}

with stardog.Admin(**conn_details) as admin:
  db = admin.new_database('db')

  with stardog.Connection('db', **conn_details) as conn:
    conn.begin()
```

(continues on next page)

(continued from previous page)

```
    conn.add(stardog.content.File('./test/data/example.ttl'))
    conn.commit()
    results = conn.select('select * { ?a ?p ?o }')

  db.drop()
```

1.4 Interactive Tutorial

There is a Jupyter notebook and instructions in the `notebooks` directory of this repository.

1.5 Documentation

Documentation is available at http://pystardog.readthedocs.io

1.5.1 Build the Docs Locally

The docs can be built locally using Sphinx:

```
cd docs
pip install -r requirements.txt
make html
```

Autodoc Type Hints

The docs use `sphinx-autodoc-typehints` which allows you to omit types when documenting argument/returns types of functions. For example:

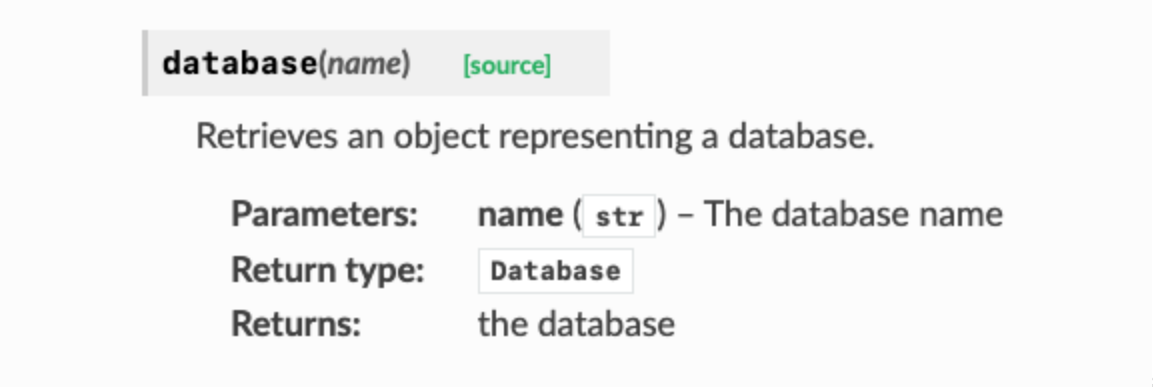
The following function:

```
def database(self, name: str) -> "Database":
    """Retrieves an object representing a database.

    :param name: The database name

    :return: the database
    """
    return Database(name, self.client)
```

will yield the following documentation after Sphinx processes it:



sphinx-autobuild-example

> **Note** Only arguments that have an existing `:param:` directive in the docstring get their respective `:type:` directives added. The `:rtype:` directive is added if and only if no existing `:rtype:` is found. See the docs for additional information on how the extension works.

Auto Build

Docs can be rebuilt automatically when saving a Python file by utilizing `sphinx-autobuild`

```
cd docs
pip install -r requirements.txt requirements-dev.txt
make livehtml
```

This should make the docs available at http://localhost:8000.

Example output after running `make livehtml`:

```
 make livehtml
sphinx-autobuild "." "_build"   --watch ../stardog/
[sphinx-autobuild] > sphinx-build /Users/frodo/projects/pystardog/docs /Users/frodo/
↪projects/pystardog/docs/_build
Running Sphinx v6.2.1
loading pickled environment... done
building [mo]: targets for 0 po files that are out of date
writing output...
building [html]: targets for 0 source files that are out of date
updating environment: 0 added, 0 changed, 0 removed
reading sources...
looking for now-outdated files... none found
no targets are out of date.
build succeeded.

The HTML pages are in _build.
[I 230710 15:26:18 server:335] Serving on http://127.0.0.1:8000
[I 230710 15:26:18 handlers:62] Start watching changes
[I 230710 15:26:18 handlers:64] Start detecting changes
```

1.6 Contributing and Development

Contrbutions are always welcome to pystardog.

To make a contribution:

1. Create a new branch off of `main`. There is no set naming convention for branches but try and keep it descriptive.

```
git checkout -b feature/add-support-for-X
```

2. Make your changes. If you are making substantive changes to pystardog, tests should be added to ensure your changes are working as expected. See *Running Tests* for additional information about running tests.
3. Format your code. All Python code should be formatted using Black. See *Formatting Your Code* for additional information.
4. Commit and push your code. Similar to branch names, there is no set structure for commit messages but try and keep your commit messages succinct and on topic.

```
git commit -am "feat: adds support for feature X"
git push origin feature/add-support-for-x
```

5. Create a pull request against `main`. All CircleCI checks should be passing in order to merge your PR. CircleCI will run tests against all supported versions of Python, single node and cluster tests for pystardog, as well as do some static analysis of the code.

1.6.1 Running Tests

Requirements:

- Docker
- Docker Compose
- Valid Stardog License

To run the tests locally, a valid Stardog license is required and placed at `dockerfiles/stardog-license-key.bin`.

1. Bring a stardog instance using docker-compose. For testing about 90% of the pystardog features, just a single node is sufficient, although we also provide a cluster set up for further testing.

```
# Bring a single node instance plus a bunch of Virtual Graphs for testing␣
↪(Recommended).
docker-compose -f docker-compose.single-node.yml up -d

# A cluster setup is also provided, if cluster only features are to be implemented␣
↪and tested.
docker-compose -f docker-compose.cluster.yml up -d
```

2. Create a virtual environment with the necessary dependencies:

```
# Create a virtualenv and activate it
virtualenv -p $(which python3) venv
source venv/bin/activate

# Install dependencies
pip install -r requirements.txt -r test-requirements.txt
```

3. Run the test suite:

```
# Run the basic test suite (covers most of the pystardog functionalities)
pytest test/test_admin_basic.py test/test_connection.py test/test_utils.py -s
```

Note Tests can be targeted against a specific Stardog endpoint by specifying an `--endpoint` option to `pytest`. Please note, that the tests will make modifications to the Stardog instance like deleting users, roles, databases, etc. By default, the `--endpoint` is set to `http://localhost:5820`, which is where the Dockerized Stardog (defined in the Docker compose files) is configured to be available at.

```
pytest test/test_connection.py -k test_queries -s --endpoint https://my-
↪other-stardog:5820
```

1.6.2 Formatting your code

To format all the Python code:

```
# make sure black is install
virtualenv -p $(which python3) venv
. venv/bin/activate
pip install -r test-requirements.txt

# run black formatter
black .
```

CHAPTER

TWO

MODULES

2.1 stardog.connection

Connect to Stardog databases.

class stardog.connection.**CommitResult**(**args*, ***kwargs*)

Bases: dict

The result of committing a transaction.

Represents the outcome of a transaction commit operation, including the counts of added and removed triples.

added: int

The amount of triples added in the transaction

removed: int

The amount of triples removed in the transaction

class stardog.connection.**Connection**(*database*, *endpoint='http://localhost:5820'*, *username='admin'*, *password='admin'*, *auth=None*, *session=None*, *run_as=None*)

Bases: object

Database Connection.

This is the entry point for all user-related operations on a Stardog database

__init__(*database*, *endpoint='http://localhost:5820'*, *username='admin'*, *password='admin'*, *auth=None*, *session=None*, *run_as=None*)

Initializes a connection to a Stardog database.

Parameters

- **database** (str) – Name of the database
- **endpoint** (Optional[str], default: 'http://localhost:5820') – URL of the Stardog server endpoint.
- **username** (Optional[str], default: 'admin') – Username to use in the connection.
- **password** (Optional[str], default: 'admin') – Password to use in the connection.
- **auth** (Optional[AuthBase], default: None) – requests.auth.AuthBase object. Used as an alternative authentication scheme. If not provided, HTTP Basic auth will be attempted with the username and password.
- **session** (Optional[Session], default: None) – requests.session.Session object
- **run_as** (Optional[str], default: None) – The user to impersonate.

Examples

```
>>> conn = Connection('db', endpoint='http://localhost:9999',
                      username='admin', password='admin')
```

add(*content*, *graph_uri=None*, *server_side=False*)

Adds data to the database.

Parameters

- **content** (Union[*Content*, str]) – Data to add to a graph.
- **graph_uri** (Optional[str], default: None) – Named graph into which to add the data. If no named graph is provided, the data will be loaded into the default graph.
- **server_side** (bool, default: False) – Whether the file to load is located on the same file system as the Stardog server.

Return type
None

Raises
stardog.exceptions.TransactionException – If currently not in a transaction

Examples

Loads example.ttl from the current directory

```
>>> conn.add(File('example.ttl'), graph_uri='urn:graph')
```

Loads /tmp/example.ttl which exists on the same file system as the Stardog server, and loads it in the default graph.

```
>>> conn.add(File('/tmp/example.ttl'), server_side=True)
```

ask(*query*, *base_uri=None*, *limit=None*, *offset=None*, *timeout=None*, *reasoning=None*, *bindings=None*, *default_graph_uri=None*, *named_graph_uri=None*, ***kwargs*)

Executes a SPARQL ASK query.

Parameters

- **query** (str) – SPARQL query
- **base_uri** (Optional[str], default: None) – Base URI for the parsing of the query
- **limit** (Optional[int], default: None) – Maximum number of results to return
- **offset** (Optional[int], default: None) – Offset into the result set
- **timeout** (Optional[int], default: None) – Number of ms after which the query should timeout. 0 or less implies no timeout
- **reasoning** (Optional[bool], default: None) – Enable reasoning for the query
- **bindings** (Optional[Dict[str, str]], default: None) – Map between query variables and their values
- **default_graph_uri** (Optional[List[str]], default: None) – URI(s) to be used as the default graph (equivalent to FROM)

- **named_graph_uri** (Optional[List[str]], default: None) – URI(s) to be used as named graphs (equivalent to FROM NAMED)

Return type
bool

Returns
whether the query pattern has a solution or not

Examples:

Listing 1: ASK query with reasoning enabled.

```
pattern_exists = conn.ask('ask {:subj :pred :obj}', reasoning=True)
```

See also:

SPARQL Spec - ASK Queries

begin(***kwargs*)

Begins a transaction.

Parameters
reasoning (*bool, optional*) – Enable reasoning for all queries inside the transaction. If the transaction does not have reasoning enabled, queries within will not be able to use reasoning.

Returns
Transaction ID

Return type
str

Raises
stardog.exceptions.TransactionException – If already in a transaction

clear(*graph_uri=None*)

Removes all data from the database or specific named graph.

Parameters
graph_uri (Optional[str], default: None) – Named graph from which to remove data.

Return type
None

Warning: If no graph_uri is specified, the entire database will be cleared.

Raises
stardog.exceptions.TransactionException – If currently not in a transaction

Examples

clear a specific named graph

```
>>> conn.clear('urn:graph')
```

clear the whole database

```
>>> conn.clear()
```

close()

Close the underlying HTTP connection.

Return type
None

commit()

Commits the current transaction.

Return type
CommitResult

Raises
stardog.exceptions.TransactionException – If currently not in a transaction

docs()

Makes a document storage object.

Return type
Docs

explain(*query*, *base_uri=None*)

Explains the evaluation of a SPARQL query.

Parameters

- **query** (str) – the SPARQL query to explain
- **base_uri** (Optional[str], default: None) – base URI for the parsing of the query

Return type
str

Returns
Query explanation

explain_inconsistency(*graph_uri=None*)

Explains why the database or a named graph is inconsistent.

Parameters
graph_uri (Optional[str], default: None) – the URI of the named graph for which to explain inconsistency

Return type
dict

Returns
explanation results

explain_inference(*content*)

Explains the given inference results.

Parameters
content (*Content*) – data from which to provide explanations

Return type
dict

Returns
explanation results

Examples

```
>>> conn.explain_inference(File('inferences.ttl'))
```

export(*content_type='text/turtle'*, *stream=False*, *chunk_size=10240*, *graph_uri=None*)

Exports the contents of the database.

Parameters

- **content_type** (str, default: 'text/turtle') – RDF content type.
- **stream** (bool, default: False) – Stream and chunk results?. See the note below for additional information.
- **chunk_size** (int, default: 10240) – Number of bytes to read per chunk when streaming.
- **graph_uri** (Optional[str], default: None) – URI of the named graph to export

Return type
Union[str, Iterator[bytes]]

Note: If stream=False (default), the contents of the database or named graph will be returned as a str. If stream=True, an iterable that yields chunks of content as bytes will be returned.

Examples

No streaming

```
>>> contents = conn.export()
```

Streaming

```
>>> with conn.export(stream=True) as stream:
      contents = ''.join(stream)
```

graph(*query*, *base_uri=None*, *limit=None*, *offset=None*, *timeout=None*, *reasoning=None*, *bindings=None*, *content_type='text/turtle'*, *default_graph_uri=None*, *named_graph_uri=None*, ***kwargs*)

Executes a SPARQL graph (CONSTRUCT) query.

Parameters

- **query** (str) – SPARQL query
- **base_uri** (Optional[str], default: None) – Base URI for the parsing of the query

- **limit** (Optional[int], default: None) – Maximum number of results to return
- **offset** (Optional[int], default: None) – Offset into the result set
- **timeout** (Optional[int], default: None) – Number of ms after which the query should timeout. 0 or less implies no timeout
- **reasoning** (Optional[bool], default: None) – Enable reasoning for the query
- **bindings** (Optional[Dict[str, str]], default: None) – Map between query variables and their values
- **content_type** (default: 'text/turtle') – Content type for results.
- **default_graph_uri** (Optional[List[str]], default: None) – URI(s) to be used as the default graph (equivalent to FROM)
- **named_graph_uri** (Optional[List[str]], default: None) – URI(s) to be used as named graphs (equivalent to FROM NAMED)

Return type
bytes

Returns
the query results

Examples:

Listing 2: Simple CONSTRUCT (graph) query utilizing limit and offset to restrict the result set with reasoning enabled.

```
results = conn.graph('select * {?s ?p ?o}',
            offset=100,
            limit=100,
            reasoning=True
          )
```

Listing 3: CONSTRUCT (graph) query utilizing bindings to bind the query variable o to a value of <urn:a>.

```
results = conn.graph('select * {?s ?p ?o}', bindings={'o': '<urn:a>'})
```

graphql()

Makes a GraphQL object.

Return type
GraphQL

icv()

Makes an integrity constraint validation (ICV) object.

Return type
ICV

is_consistent(*graph_uri=None*)

Checks if the database or named graph is consistent with respect to its schema.

Parameters
graph_uri (Optional[str], default: None) – the URI of the graph to check

Return type
bool

Returns
database consistency state

paths(*query*, *base_uri=None*, *limit=None*, *offset=None*, *timeout=None*, *reasoning=None*, *bindings=None*, *content_type='application/sparql-results+json'*, *default_graph_uri=None*, *named_graph_uri=None*, ***kwargs*)

Executes a SPARQL paths query.

Parameters

- **query** (str) – SPARQL query
- **base_uri** (Optional[str], default: None) – Base URI for the parsing of the query
- **limit** (Optional[int], default: None) – Maximum number of results to return
- **offset** (Optional[int], default: None) – Offset into the result set
- **timeout** (Optional[int], default: None) – Number of ms after which the query should timeout. 0 or less implies no timeout
- **reasoning** (Optional[bool], default: None) – Enable reasoning for the query
- **bindings** (Optional[Dict[str, str]], default: None) – Map between query variables and their values
- **content_type** (default: 'application/sparql-results+json') – Content type for results.
- **default_graph_uri** (Optional[List[str]], default: None) – URI(s) to be used as the default graph (equivalent to FROM)
- **named_graph_uri** (Optional[List[str]], default: None) – URI(s) to be used as named graphs (equivalent to FROM NAMED)

Return type
Union[Dict, bytes]

Returns
If content_type='application/sparql-results+json', results will be returned as a Dict , else results will be returned as bytes.

Examples:

Listing 4: Simple PATHS query with reasoning enabled.

```
results = conn.paths('paths start ?x = :subj end ?y = :obj via ?p',␣
↪reasoning=True)
```

See also:

Stardog Docs - PATH Queries

remove(*content*, *graph_uri=None*)

Removes data from the database.

Parameters

- **content** (Content) – Data to add
- **graph_uri** (*str, optional*) – Named graph from which to remove the data

Raises
stardog.exceptions.TransactionException – If currently not in a transaction

Examples

```
>>> conn.remove(File('example.ttl'), graph_uri='urn:graph')
```

rollback()

Rolls back the current transaction.

Raises

stardog.exceptions.TransactionException – If currently not in a transaction

select(*query*, *base_uri=None*, *limit=None*, *offset=None*, *timeout=None*, *reasoning=None*, *bindings=None*, *content_type='application/sparql-results+json'*, *default_graph_uri=None*, *named_graph_uri=None*, ***kwargs*)

Executes a SPARQL select query.

Parameters

- **query** (str) – SPARQL query
- **base_uri** (Optional[str], default: None) – Base URI for the parsing of the query
- **limit** (Optional[int], default: None) – Maximum number of results to return
- **offset** (Optional[int], default: None) – Offset into the result set
- **timeout** (Optional[int], default: None) – Number of ms after which the query should timeout. 0 or less implies no timeout
- **reasoning** (Optional[bool], default: None) – Enable reasoning for the query
- **bindings** (Optional[Dict[str, str]], default: None) – Map between query variables and their values
- **content_type** (str, default: 'application/sparql-results+json') – Content type for results.
- **default_graph_uri** (Optional[List[str]], default: None) – URI(s) to be used as the default graph (equivalent to FROM)
- **named_graph_uri** (Optional[List[str]], default: None) – URI(s) to be used as named graphs (equivalent to FROM NAMED)

Return type

Union[bytes, Dict]

Returns

If content_type='application/sparql-results+json', results will be returned as a Dict, else results will be returned as bytes.

Examples:

Listing 5: Simple SELECT query utilizing limit and offset to restrict the result set with reasoning enabled.

```
results = conn.select('select * {?s ?p ?o}',
            offset=100,
            limit=100,
            reasoning=True
          )
```

Listing 6: Query utilizing `bindings` to bind the query variable `o` to a value of `<urn:a>`.

```
results = conn.select('select * {?s ?p ?o}', bindings={'o': '<urn:a>'})
```

size(*exact=False*)

Calculate the size of the database.

Parameters

exact (bool, default: False) – calculate the size of the database exactly. If False (default), the size will be an estimate; this should take less time to calculate especially if the database is large.

Return type

int

Returns

the number of triples in the database

update(*query*, *base_uri=None*, *limit=None*, *offset=None*, *timeout=None*, *reasoning=None*, *bindings=None*, *using_graph_uri=None*, *using_named_graph_uri=None*, *remove_graph_uri=None*, *insert_graph_uri=None*, ***kwargs*)

Executes a SPARQL update query.

Parameters

- **query** (str) – SPARQL query
- **base_uri** (Optional[str], default: None) – Base URI for the parsing of the query
- **limit** (Optional[int], default: None) – Maximum number of results to return
- **offset** (Optional[int], default: None) – Offset into the result set
- **timeout** (Optional[int], default: None) – Number of ms after which the query should timeout. 0 or less implies no timeout
- **reasoning** (Optional[bool], default: None) – Enable reasoning for the query
- **bindings** (Optional[Dict[str, str]], default: None) – Map between query variables and their values
- **using_graph_uri** (Optional[List[str]], default: None) – URI(s) to be used as the default graph (equivalent to USING)
- **using_named_graph_uri** (Optional[List[str]], default: None) – URI(s) to be used as named graphs (equivalent to USING NAMED)
- **remove_graph_uri** (Optional[str], default: None) – URI of the graph to be removed from
- **insert_graph_uri** (Optional[str], default: None) – URI of the graph to be inserted into

Return type

None

Examples

```
>>> conn.update('delete where {?s ?p ?o}')
```

class stardog.connection.**Docs**(*client*)

Bases: object

BITES: Document Storage.

See also:

Stardog Docs - BITES

__init__(*client*)

Initializes a Docs.

Use *stardog.connection.Connection.docs()* instead of constructing manually.

add(*name*, *content*)

Adds a document to the store.

Parameters

- **name** (str) – Name of the document
- **content** (*Content*) – Contents of the document

Return type
None

Examples

```
>>> docs.add('example', File('example.pdf'))
```

clear()

Removes all documents from the store.

Return type
None

delete(*name*)

Deletes a document from the store.

Parameters
name (str) – Name of the document to delete

Return type
None

get(*name*, *stream=False*, *chunk_size=10240*)

Gets a document from the store.

Parameters

- **name** (str) – Name of the document
- **stream** (bool, default: False) – If document should be streamed back as chunks of bytes or as one string .
- **chunk_size** (int, default: 10240) – Number of bytes to read per chunk when streaming.

Return type
Union[str, Iterator[bytes]]

Note: If `stream=False`, the contents of the document will be returned as a `str`. If `stream=True`, an iterable that yields chunks of content as `bytes` will be returned.

Examples

No streaming

```
>>> contents = docs.get('example')
```

Streaming

```
>>> with docs.get('example', stream=True) as stream:
                    contents = ''.join(stream)
```

size()

Calculates document store size.

Return type
int

Returns
Number of documents in the store

class stardog.connection.**GraphQL**(*conn*)

Bases: object

See also:

Stardog Docs - GraphQL

__init__(*conn*)

Initializes a GraphQL.

Use *stardog.connection.Connection.graphql()* instead of constructing manually.

add_schema(*name*, *content*)

Adds a schema to the database.

Parameters

- **name** (str) – Name of the schema
- **content** (*Content*) – Schema data

Return type
None

Examples

```
>>> gql.add_schema('people', content=File('people.graphql'))
```

clear_schemas()

Deletes all schemas.

Return type
None

query(*query*, *variables=None*)

Executes a GraphQL query.

Parameters

- **query** (str) – GraphQL query
- **variables** (Optional[dict], default: None) – GraphQL variables. Keys: @reasoning (bool) to enable reasoning, @schema (str) to define schemas

Return type
dict

Returns
Query results

Examples

with schema and reasoning

```
>>> gql.query('{ Person {name} }',
              variables={'@reasoning': True, '@schema': 'people'})
```

with named variables

```
>>> gql.query(
      'query getPerson($id: Int) { Person(id: $id) {name} }',
      variables={'id': 1000})
```

remove_schema(*name*)

Removes a schema from the database.

Parameters
name (str) – Name of the schema

Return type
None

schema(*name*)

Gets schema information.

Parameters
name (str) – Name of the schema

Return type
str

Returns
GraphQL schema

schemas()

Retrieves all available schemas.

Return type
Dict

Returns
All schemas

class stardog.connection.**ICV**(*conn*)

Bases: object

Integrity Constraint Validation.

See also:

Stardog Docs - Data Quality Constraints

__init__(*conn*)

Initializes an ICV.

Use *stardog.connection.Connection.icv()* instead of constructing manually.

add(*content*)

Adds integrity constraints to the database.

Parameters
content (*Content*) – Data to add

Return type
None

Warning: Deprecated: *stardog.connection.Connection.add()* should be preferred. *stardog.connection.ICV.add()* will be removed in the next major version.

Examples

```
>>> icv.add(File('constraints.ttl'))
```

clear()

Removes all integrity constraints from the database.

Return type
None

convert(*content*, *graph_uri=None*)

Converts given integrity constraints to a SPARQL query.

Parameters
content (*Content*) – Integrity constraints

Graph_uri
Named graph from which to apply constraints

Return type
str

Returns
SPARQL query

Warning: Deprecated: *stardog.connection.ICV.convert()* was meant as a debugging tool, and will be removed in the next major version.

Examples

```
>>> icv.convert(File('constraints.ttl'), graph_uri='urn:graph')
```

explain_violations(*content*, *graph_uri=None*)

Explains violations of the given integrity constraints.

Parameters
content (*Content*) – Data containing constraints

Graph_uri
Named graph from which to check for violations

Return type
Dict

Returns
the violations

Warning: Deprecated: *stardog.connection.ICV.report()* should be preferred. *stardog.connection.ICV.explain_violations()* will be removed in the next major version.

Examples

```
>>> icv.explain_violations(File('constraints.ttl'),
                           graph_uri='urn:graph')
```

is_valid(*content*, *graph_uri=None*)

Checks if given integrity constraints are valid.

Parameters

- **content** (*Content*) – Data to check validity (with respect to constraints) against
- **graph_uri** (Optional[str], default: None) – URI of the named graph to check for validity

Return type
bool

Returns
whether the data is valid with respect to the constraints

Examples

```
>>> icv.is_valid(File('constraints.ttl'), graph_uri='urn:graph')
```

list()

List all integrity constraints from the database.

Return type
str

remove(*content*)

Removes integrity constraints from the database.

Parameters
content (*Content*) – Data to remove

Return type
None

Warning: Deprecated: *stardog.connection.Connection.remove()* should be preferred. *stardog.connection.ICV.remove()* will be removed in the next major version.

Examples

```
>>> icv.remove(File('constraints.ttl'))
```

report(***kwargs*)

Produces a SHACL validation report.

Keyword Arguments

- **shapes** (*str, optional*) – SHACL shapes to validate
- **shacl.shape.graphs** (*str, optional*) – SHACL shape graphs to validate
- **nodes** (*str, optional*) – SHACL focus node(s) to validate
- **countLimit** (*str, optional*) – Maximum number of violations to report
- **shacl.targetClass.simple** (*bool, optional*) – If True, sh:targetClass will be evaluated based on rdf:type triples only, without following rdfs:subClassOf relations
- **shacl.violation.limit.shape** (*str, optional*) – number of violation limits per SHACL shapes
- **graph-uri** (*str, optional*) – Named Graph
- **reasoning** (*bool, optional*) – If True, enable reasoning.

Return type
str

Returns
SHACL validation report

Examples

```
>>> icv.report()
```

2.2 stardog.admin

Administer a Stardog server.

class stardog.admin.**Admin**(*endpoint='http://localhost:5820'*, *username='admin'*, *password='admin'*, *auth=None*, *run_as=None*)

Bases: object

Admin Connection.

This is the entry point for admin-related operations on a Stardog server. .. seealso:: Stardog Docs - Operating Stardog

__init__(*endpoint='http://localhost:5820'*, *username='admin'*, *password='admin'*, *auth=None*, *run_as=None*)

Initializes an admin connection to a Stardog server.

Parameters

- **endpoint** (Optional[str], default: 'http://localhost:5820') – URL of the Stardog server endpoint.
- **username** (Optional[str], default: 'admin') – Username to use in the connection.
- **password** (Optional[str], default: 'admin') – Password to use in the connection.
- **auth** (Optional[AuthBase], default: None) – requests.auth.AuthBase object. Used as an alternative authentication scheme. If not provided, HTTP Basic auth will be attempted with the username and password.
- **run_as** (Optional[str], default: None) – the user to impersonate

Note: auth and username/password should not be used together. If they are, the value of auth will take precedent.

Examples

```
>>> admin = Admin(endpoint='http://localhost:9999',
                  username='admin', password='admin')
```

alive()

Determine whether the server is running

Return type
bool

Returns
is the server alive?

backup_all(*location=None*)

Create a backup of all databases on the server. This is also known as a **server backup**.

Parameters
location (Optional[str], default: None) – where to write the server backup to on the Stardog server's file system.

Note: By default, backups are stored in the .backup directory in $STARDOG_HOME, but you can use the backup.dir property in your stardog.properties file to specify a different location for backups or you can override it using the location parameter.

---

cache(*name*)

Retrieve an object representing a cached dataset.

Parameters
name (str) – the name of the cache to retrieve

Return type
Cache

cache_status(**names*)

Retrieves the status of one or more cached graphs.

Parameters
names (str) – Names of the cached graphs to retrieve status for

Return type
List[Dict]

Returns
list of statuses

cache_targets()

Retrieves all cache targets.

Return type
List[*CacheTarget*]

cached_graphs()

Retrieves all cached graphs.

Return type
List[*Cache*]

cached_queries()

Retrieves all cached queries.

Warning: This method is deprecated in Stardog 8+

Return type
List[*Cache*]

Returns
cached queries

cached_status()

Retrieves all cached graphs.

Return type
List[*Cache*]

clear_stored_queries()

Remove all stored queries on the server.

Return type
None

cluster_coordinator_check()

Determine if a specific cluster node is the cluster coordinator

Return type
bool

Returns
whether the node is a coordinator or not.

cluster_info()

Prints info about the nodes in the Stardog cluster.

Return type
Dict

Returns
information about nodes in the cluster

cluster_join()

Instruct a standby node to join its cluster as a full node

Return type
None

cluster_list_standby_nodes()

List standby nodes

Return type
Dict

Returns
all standby nodes in the cluster

cluster_revoke_standby_access(*registry_id*)

Instruct a standby node to stop syncing

Parameters
registry_id (str) – ID of the standby node.

Return type
None

cluster_shutdown()

Shutdown all nodes in the cluster

Return type
bool

Returns
whether the cluster was shutdown successfully or not.

`cluster_start_readonly()`

Start read only mode

Return type
`None`

`cluster_status()`

Prints status information for each node in the cluster

Return type
`Dict`

Returns
status information about each node in the cluster

`cluster_stop_readonly()`

Stops read only mode

Return type
`None`

`database(name)`

Retrieves an object representing a database.

Parameters
name (`str`) – The database name

Return type
Database

Returns
the database

`databases()`

Retrieves all databases.

Return type
`List[Database]`

`datasource(name)`

Retrieves an object representing a DataSource.

Parameters
name (`str`) – The name of the data source

Return type
DataSource

`datasources()`

Retrieves all data sources.

Return type
`List[DataSource]`

`datasources_info()`

List all data sources with their details

Return type
`List[Dict]`

Returns
a list of data sources with their details

`get_all_metadata_properties()`

Get information on all database metadata properties, including description and example values

Return type
`Dict`

Returns
Metadata properties

See also:

HTTP API - Get all database metadata properties

`get_prometheus_metrics()`

Return type
`str`

`get_server_metrics()`

Returns metric information from the registry in JSON format

Return type
`Dict`

Returns
Server metrics

See also:

HTTP API - Server metrics

`get_server_properties()`

Get the value of any set server-level properties

Return type
`Dict`

Returns
server properties

`healthcheck()`

Determine whether the server is running and able to accept traffic

Return type
`bool`

Returns
is the server accepting traffic?

`import_file(db, mappings, input_file, options=None, named_graph=None)`

Import a JSON or CSV file.

Parameters

- **db** (`str`) – Name of the database to import the data
- **mappings** (`Union[MappingRaw, MappingFile]`) – Mappings specifying how to import the data contained in the CSV/JSON.
- **input_file** (`Union[ImportFile, ImportRaw]`) – the JSON or CSV file to import
- **options** (`Optional[Dict]`, default: `None`) – Options for the import.
- **named_graph** (`Optional[str]`, default: `None`) – The named graph to import the mapped CSV/JSON into.

Return type
bool

Returns
was the import successful?

import_virtual_graph(*db*, *mappings*, *named_graph*, *remove_all*, *options*)

Import (materialize) a virtual graph directly into the Stardog database.

> **Warning: Deprecated**: *stardog.admin.Admin.materialize_virtual_graph()* should be preferred.

Parameters

- **db** (str) – The database into which to import the graph
- **mappings** (Union[*MappingRaw*, *MappingFile*, str]) – New mapping contents. An empty string can be passed for autogenerated mappings.
- **named_graph** (str) – Name of the graph to import the virtual graph into.
- **remove_all** (Optional[bool]) – Should the target named graph be cleared before importing?
- **options** (Dict) – Options for the new virtual graph. See Stardog Docs - Virtual Graph Properties for all available options.

Return type
None

Examples:

Listing 7: Import a MySQL virtual graph into the `db-name` database using the mappings specified in `mappings.ttl`. The virtual graph will be imported into the named graph `my-graph` and prior to the import will have its contents cleared.

```
admin.import_virtual_graph(
      'db-name',
      mappings=File('mappings.ttl'),
      named_graph='my-graph',
      remove_all=True,
      options={'jdbc.driver': 'com.mysql.jdbc.Driver'}
)
```

kill_query(*id*)

Kills a running query.

Parameters
id (str) – ID of the query to kill

Return type
None

materialize_virtual_graph(*db*, *mappings*, *data_source=None*, *options=None*, *named_graph='tag:stardog:api:context:default'*, *remove_all=False*)

Import (materialize) a virtual graph directly into a database.

Parameters

- **db** (str) – The database into which to import the graph
- **mappings** (Union[*MappingFile*, *MappingRaw*, str]) – New mapping contents. An empty string can be passed for autogenerated mappings.
- **data_source** (Optional[str], default: None) – The datasource to load from
- **options** (Optional[Dict], default: None) – Options for the new virtual graph, See Stardog Docs - Virtual Graph Properties for all available options.
- **named_graph** (Optional[str], default: 'tag:stardog:api:context:default') – Name of the graph into which import the virtual graph.
- **remove_all** (bool, default: False) – Should the target named graph be cleared before importing?

Note: data_source or options must be provided.

new_cache_target(*name*, *hostname*, *port*, *username*, *password*, *use_existing_db=False*)

Creates a new cache target.

Parameters

- **name** (str) – The name of the cache target
- **hostname** (str) – The hostname of the cache target server
- **port** (int) – The port of the cache target server
- **username** (str) – The username for the cache target
- **password** (str) – The password for the cache target
- **use_existing_db** (bool, default: False) – If True, check for an existing cache database to use before creating a new one

Return type
CacheTarget

Returns
the new CacheTarget

new_cached_graph(*name*, *target*, *graph*, *database=None*, *refresh_script=None*, *register_only=False*)

Creates a new cached graph.

Parameters

- **name** (str) – The name (URI) of the cached query
- **target** (str) – The name (URI) of the cache target
- **graph** (str) – The name of the graph to cache
- **database** (Optional[str], default: None) – The name of the database. Optional for virtual graphs, required for named graphs.
- **refresh_script** (Optional[str], default: None) – An optional SPARQL update query to run when refreshing the cache.
- **register_only** (bool, default: False) – An optional value that if True, register a cached dataset without loading data from the source graph or query into the cache target's databases.

Return type
Cache

Returns
The new Cache

new_cached_query(*name*, *target*, *query*, *database*, *refresh_script=None*, *register_only=False*)

Creates a new cached query.

> **Warning:** This method is deprecated in Stardog 8+

Parameters

- **name** (str) – The name (URI) of the cached query
- **target** (str) – The name (URI) of the cache target
- **query** (str) – The query to cache
- **database** (str) – The name of the database
- **refresh_script** (Optional[str], default: None) – A SPARQL insert query to run when refreshing the cache
- **register_only** (bool, default: False) – If True, register a cached dataset without loading data from the source graph or query into the cache target's databases

Return type
Cache

Returns
the new Cache

new_database(*name*, *options=None*, **contents*, ***kwargs*)

Creates a new database.

Parameters

- **name** (str) – the database name
- **options** (Optional[Dict], default: None) – Dictionary with database options
- **contents** (Union[*Content*, Tuple[*Content*, str], None]) – Datasets to perform bulk-load with. Named graphs are made with tuples of Content and the name.

Keyword Arguments
copy_to_server – . If True, sends the files to the Stardog server; if running as a cluster, data will be replicated to all nodes in the cluster.

Return type
Database

Examples

Options

```
>>> admin.new_database('db', {'search.enabled': True})
```

bulk-load

```
>>> admin.new_database('db', {},
                       File('example.ttl'), File('test.rdf'))
```

bulk-load to named graph

```
>>> admin.new_database('db', {}, (File('test.rdf'), 'urn:context'))
```

new_datasource(*name*, *options*)

Creates a new DataSource.

Parameters

- **name** (str) – The name of the data source
- **options** (Dict) – Data Source options

Return type
DataSource

Returns
The new DataSource object

new_role(*name*)

Creates a new role.

Parameters
name (str) – the name of the new role

Returns
the new Role object

new_stored_query(*name*, *query*, *options=None*)

Creates a new Stored Query.

Parameters

- **name** (str) – the name of the stored query
- **query** (str) – the query to store
- **options** (Optional[Dict], default: None) – Additional options (e.g. {"shared": True, "database": "myDb" })

Return type
StoredQuery

Returns
the new StoredQuery object

Examples:

Listing 8: Create a new stored query named `all triples` and make it only executable against the database `mydb`.

```
new_stored_query = admin.new_stored_query(
    'all triples',
    'select * where { ?s ?p ?o . }',
    { 'database': 'mydb' }
)
```

new_user(*username*, *password*, *superuser=False*)

Creates a new user.

Parameters

- **username** (`str`) – The username
- **password** (`str`) – The password
- **superuser** (`bool`, default: `False`) – Create the user as a superuser. Only superusers can make other superusers.

Return type
User

Returns
The new User object

new_virtual_graph(*name*, *mappings=None*, *options=None*, *datasource=None*, *db=None*)

Creates a new Virtual Graph.

Parameters

- **name** (`str`) – The name of the virtual graph.
- **mappings** (`Union[MappingFile, MappingRaw, None]`, default: `None`) – New mapping contents. If `None` provided, mappings will be autogenerated.
- **options** (`Optional[Dict]`, default: `None`) – Options for the new virtual graph. If `None` provided, then a `datasource` must be specified.
- **datasource** (`Optional[str]`, default: `None`) – Name of the datasource to use. If `None` provided, `options` with a `datasource` key must be set.
- **db** (`Optional[str]`, default: `None`) – Name of the database to associate the virtual graph. If `None` provided, the virtual graph will be associated with all databases.

Return type
VirtualGraph

Returns
the new VirtualGraph

Examples:

Listing 9: Create a new virtual graph named `users` and associate it with all databases. The SMS2 mappings are provided in the `mappings.ttl` file.

```
new_vg = admin.new_virtual_graph(
            name='users',
```

(continues on next page)

(continued from previous page)

```
            mappings=MappingFile('mappings.ttl','SMS2'),
            datasource='my_datasource'
        )
```

queries()

Gets information about all running queries.

Return type
Dict

Returns
information about all running queries

query(*id*)

Gets information about a running query.

Parameters
id (str) – Query ID

Return type
Dict

Returns
Query information

restore(*from_path*, *, *name=None*, *force=False*)

Restore a database.

Parameters

- **from_path** (str) – the full path on the server's file system to the backup
- **name** (Optional[str], default: None) – the name of the database to restore to if different from the backup
- **force** (Optional[bool], default: False) – by default, a backup will not be restored in place of an existing database of the same name; the force parameter should be used to overwrite the database

Return type
None

Examples:

Listing 10: simple restore

```
admin.restore("/data/stardog/.backup/db/2019-12-01")
```

Listing 11: restore the backup and overwrite db2 database

```
admin.restore("/data/stardog/.backup/db/2019-11-05",
              name="db2", force=True)
```

See also:

Stardog Docs - Restoring a Database

HTTP API - Restore a Database

role(*name*)

Retrieves a Role.

Parameters
name (str) – The name of the role

Return type
Role

roles()

Retrieves all roles.

Return type
List[*Role*]

shutdown()

Shuts down the server.

Return type
None

standby_node_pause(*pause*)

Pause/Unpause standby node

Parameters
pause (bool) – True should be provided to pause the standby node. False should be provided to unpause.

Return type
bool

Returns
whether the pause status was successfully changed or not.

standby_node_pause_status()

Get the pause status of a standby node

Return type
Dict

Returns
Pause status of a standby node, possible values are: WAITING, SYNCING, PAUSING, PAUSED

See also:

HTTP API - Get Paused State of Standby Node

stored_queries()

Retrieves all stored queries.

Return type
List[*StoredQuery*]

stored_query(*name*)

Retrieves a Stored Query.

Parameters
name (str) – The name of the stored query to retrieve

Return type
StoredQuery

user(*name*)

Retrieves a User object.

Parameters
name (str) – The name of the user

Return type
User

users()

Retrieves all users.

Return type
List[*User*]

validate()

Validates an admin connection.

Return type
bool

Returns
whether the connection is valid or not

virtual_graph(*name*)

Retrieves a Virtual Graph.

Parameters
name (str) – The name of the virtual graph to retrieve

Return type
VirtualGraph

virtual_graphs()

Retrieves all virtual graphs.

Return type
List[*VirtualGraph*]

class stardog.admin.**Cache**(*name*, *client*)

Bases: object

Cached data

A cached dataset from a query or named/virtual graph.

See also:

Stardog Docs - Cache Management

__init__(*name*, *client*)

Initializes a new cached dataset from a query or named/virtual graph.

Use *stardog.admin.Admin.new_cached_graph()* or *stardog.admin.Admin.new_cached_query()* instead of constructing manually.

drop()

Drops the cache.

Return type
None

refresh()

Refreshes the cache.

Return type
None

status()

Retrieves the status of the cache.

Return type
Dict

class stardog.admin.**CacheTarget**(*name*, *client*)

Bases: object

Cache Target Server

__init__(*name*, *client*)

Initializes a cache target.

Use *stardog.admin.Admin.new_cache_target()* instead of constructing manually.

info()

Get info for the cache target

Returns
Info

Return type
dict

property name

The name (URI) of the cache target.

orphan()

Orphans the cache target but do not destroy its contents.

remove()

Removes the cache target and destroy its contents.

class stardog.admin.**DataSource**(*name*, *client*)

Bases: object

Initializes a DataSource

See also:

Stardog Docs - Data Sources

__init__(*name*, *client*)

Initializes a DataSource.

Use stardog.admin.Admin.data_source(), stardog.admin.Admin.data_sources(), or stardog.admin.Admin.new_data_source() instead of constructing manually.

available()

Checks if the data source is available.

Return type
bool

Returns
whether the data source is available or not

`delete`()

Deletes a data source

Return type
`None`

`get_options`()

Get data source options

Return type
`List[Dict]`

`info`()

Get data source info

Return type
`Dict`

Returns
data source information

`property name: str`

The name of the data source.

`online`()

Online a data source

Return type
`None`

`refresh_count`(*meta=None*)

Refresh table row-count estimates

Parameters
meta (`Optional[Dict]`, default: `None`) – dict containing the table to refresh. Examples: `{"name": "catalog.schema.table"}`, `{"name": "schema.table"}`, `{"name": "table"}`

Return type
`None`

See also:

HTTP API - Refresh table row-count estimates

`refresh_metadata`(*meta=None*)

Refresh metadata for one or all tables that are accessible to a data source. Will clear the saved metadata for a data source and reload all of its dependent virtual graphs with fresh metadata.

Parameters
meta (`Optional[Dict]`, default: `None`) – dict containing the table to refresh. Examples: `{"name": "catalog.schema.table"}`, `{"name": "schema.table"}`, `{"name": "table"}`

Return type
`None`

See also:

Stardog Docs - Refreshing Data Source Metadata

HTTP API - Refresh Data Source Metadata

Stardog CLI - stardog-admin data-source refresh-metadata

share()

Share a private data source. When a virtual graph is created without specifying a data source name, a private data source is created for that, and only that virtual graph. This methods makes the data source available to other virtual graphs, as well as decouples the data source life cycle from the original virtual graph.

Return type
None

update(*options=None*, *force=False*)

Update data source

Parameters

- **options** (Optional[Dict], default: None) – Dict with data source options
- **force** (bool, default: False) – a data source will not be updated while in use unless force=True

Return type
None

Examples

```
>>> admin.update({"sql.dialect": "MYSQL"})
>>> admin.update({"sql.dialect": "MYSQL"}, force=True)
```

See also:

Stardog Docs - DataSource options

class stardog.admin.**Database**(*name*, *client*)

Bases: object

Represents a Stardog database.

__init__(*name*, *client*)

Initializes a Database.

Use *stardog.admin.Admin.database()*, *stardog.admin.Admin.databases()*, or *stardog.admin.Admin.new_database()* instead of constructing manually.

add_namespace(*prefix*, *iri*)

Adds a specific namespace to a database

Parameters

- **prefix** (str) – the prefix of the namespace to be added
- **iri** (str) – the iri associated with the prefix to be added

Return type
bool

Returns
whether the operation succeeded or not.

backup(*, *to=None*)

Create a backup of a database on the server.

Parameters
to (Optional[str], default: None) – specify a path on the Stardog server's file system to store the backup

Return type
None

See also:

Stardog Docs - Backup a Database

copy(*to*)

Makes a copy of this database under another name.

> **Warning:** This method is deprecated and not valid for Stardog versions 6+.

The database must be offline.

Parameters
to (*str*) – Name of the new database to be created

Returns
The new Database

Return type
Database

drop()

Drops the database.

Return type
None

get_all_options()

Get the value of every metadata option for a database

Return type
Dict

Returns
All database metadata

get_options(**options*)

Get the value of specific metadata options for a database

Parameters
options (str) – Database option names

Return type
Dict

Returns
Database options

Examples

```
>>> db.get_options('search.enabled', 'spatial.enabled')
```

import_namespaces(*content*)

Imports namespace prefixes from an RDF file that contains prefix declarations into the database, overriding any previous mappings for those prefixes. Only the prefix declarations in the file are processed, the rest of the file is not parsed.

Parameters
content (*Content*) – RDF File containing prefix declarations

Return type
Dict

Returns
Dictionary with all namespaces after import

property name: str

The name of the database.

namespaces()

Retrieve the namespaces stored in the database

Return type
Dict

Returns
A dict listing the prefixes and IRIs of the stored namespaces

See also:

HTTPI API - Get Namespaces

offline()

Sets a database to offline state.

Return type
None

online()

Sets a database to online state.

Return type
None

optimize()

Optimizes a database.

Return type
None

remove_namespace(*prefix*)

Removes a specific namespace from a database

Parameters
prefix (str) – the prefix of the namespace to be removed

Return type
bool

Returns
whether the operation succeeded or not.

repair()

Attempt to recover a corrupted database.

Note: The database must be offline.

Return type
bool

Returns
whether the database was successfully repaired or not

set_options(*options*)

Sets database options.

Parameters
options (Dict) – Database options

Return type
None

Note: The database must be offline to set some options (e.g. search.enabled).

Examples

```
>>> db.set_options({'spatial.enabled': False})
```

verify()

verifies a database.

Return type
None

class stardog.admin.**Role**(*name*, *client*)

Bases: object

See also:

Stardog Docs - Authorization

__init__(*name*, *client*)

Initializes a Role.

Use *stardog.admin.Admin.role()*, *stardog.admin.Admin.roles()*, or *stardog.admin.Admin.new_role()* instead of constructing manually.

add_permission(*action*, *resource_type*, *resource*)

Adds a permission to the role.

See also:

Stardog Docs - Grant Permissions to a Role

HTTP API - Grant Permission to Role

Parameters

- **action** (*str*) – Action type (e.g., 'read', 'write')
- **resource_type** (*str*) – Resource type (e.g., 'user', 'db')
- **resource** (*str*) – Target resource (e.g., 'username', '*')

Examples

```
>>> role.add_permission('read', 'user', 'username')
>>> role.add_permission('write', '*', '*')
```

delete(*force=None*)

Deletes the role.

Parameters
force (*bool*) – Force deletion of the role

property name

The name of the Role.

permissions()

Gets the role permissions.

See also:

HTTP API - Get Role Permissions

Returns
Role permissions

Return type
dict

remove_permission(*action*, *resource_type*, *resource*)

Removes a permission from the role.

See also:

HTTP API - Revoke Role Permission

Parameters

- **action** (*str*) – Action type (e.g., 'read', 'write')
- **resource_type** (*str*) – Resource type (e.g., 'user', 'db')
- **resource** (*str*) – Target resource (e.g., 'username', '*')

Examples

```
>>> role.remove_permission('read', 'user', 'username')
>>> role.remove_permission('write', '*', '*')
```

users()

Lists the users for this role.

Returns
list[User]

class stardog.admin.**StoredQuery**(*name*, *client*, *details=None*)

Bases: object

Stored Query

See also:

Stardog Docs - Stored Queries

__init__(*name*, *client*, *details=None*)

Initializes a stored query.

Use *stardog.admin.Admin.stored_query()*, *stardog.admin.Admin.stored_queries()*, or *stardog.admin.Admin.new_stored_query()* instead of constructing manually.

property creator: str

The creator of the stored query.

property database: str

The database the stored query applies to.

delete()

Deletes the Stored Query.

Return type
None

property description: str

The description of the stored query.

property name: str

The name of the stored query.

property query: str

The text of the stored query.

property reasoning: bool

The value of the reasoning property.

property shared: bool

The value of the shared property.

update(***options*)

Updates the Stored Query.

Return type
None

Parameters
****options** (*str*) – Named arguments to update.

Examples

Update description

```
>>> stored_query.update(description='this query finds all the relevant...')
```

class stardog.admin.**User**(*name*, *client*)

Bases: object

Represents a Stardog user

__init__(*name*, *client*)

Initializes a User.

Use *stardog.admin.Admin.user()*, *stardog.admin.Admin.users()*, or *stardog.admin.Admin.new_user()* instead of constructing manually.

add_permission(*action*, *resource_type*, *resource*)

Add a permission to the user.

Parameters

- **action** (str) – Action type (e.g., read, write)
- **resource_type** (str) – Resource type (e.g., user, db)
- **resource** (str) – Target resource (e.g., username, *)

Return type
None

See also:

Stardog Docs - Grant Permissions to a User

HTTP API - Grant permission to a User

Examples

```
>>> user.add_permission('read', 'user', 'username')
>>> user.add_permission('write', '*', '*')
```

add_role(*role*)

Adds an existing role to the user.

Parameters
role (Union[*Role*, str]) – The *stardog.admin.Role* or name of the role to add

Return type
None

Examples

```
>>> user.add_role('reader')
>>> user.add_role(admin.role('reader'))
```

delete()

Deletes the user.

Return type
None

effective_permissions()

Gets the user's effective permissions.

Return type
Dict

Returns
User's effective permissions

is_enabled()

Checks if the user is enabled.

Return type
bool

Returns
whether the user is enabled or not

is_superuser()

Checks if the user is a superuser.

Return type
bool

Returns
whether the user is a superuser or not.

property name: str

The username.

permissions()

Gets the user permissions.

See also:

Stardog Docs - Permissions

Return type
Dict

Returns
user permissions

remove_permission(*action*, *resource_type*, *resource*)

Removes a permission from the user.

Parameters

- **action** (str) – Action type (e.g., read, write)

- **resource_type** (str) – Resource type (e.g., user, db)
- **resource** (str) – Target resource (e.g., username, *)

See also:

HTTP API - Revoke User Permission

Examples

```
>>> user.remove_permission('read', 'user', 'username')
>>> user.remove_permission('write', '*', '*')
```

remove_role(*role*)

Removes a role from the user.

Parameters
role (Union[str, *Role*]) – The *stardog.admin.Role* or name of the role to remove

Return type
None

Examples

```
>>> user.remove_role('reader')
>>> user.remove_role(admin.role('reader'))
```

roles()

Gets all the User's roles.

Return type
List[*Role*]

set_enabled(*enabled*)

Enables or disables the user.

Parameters
enabled (bool) – Desired state. True for enabled, False for disabled.

Return type
None

set_password(*password*)

Sets a new password.

Parameters
password (str) – the new password for the user

Return type
None

set_roles(**roles*)

Sets the roles of the user.

Parameters
roles (Union[str, *Role*]) – The *stardog.admin.Role* (s) or name of the role(s) to add to the user

Return type
None

Examples

```
>>> user.set_roles('reader', admin.role('writer'))
```

class stardog.admin.**VirtualGraph**(*name*, *client*)

Bases: object

Virtual Graph

See also:

Stardog Docs - Virtual Graphs

__init__(*name*, *client*)

Initializes a virtual graph.

Use *stardog.admin.Admin.virtual_graph()*, *stardog.admin.Admin.virtual_graphs()*, or *stardog.admin.Admin.new_virtual_graph()* instead of constructing manually.

available()

Checks if the virtual graph is available.

Return type
bool

Returns
whether the virtual graph is available or not

delete()

Deletes the virtual graph.

Return type
None

get_database()

Gets database associated with the virtual graph.

Return type
str

Returns
the database name

get_datasource()

Gets datasource associated with the virtual graph

Return type
str

Returns
datasource name with `data-source://` prefix removed

info()

Gets virtual graph info.

Return type
Dict

mappings(*content_type='text/turtle'*)

Gets the Virtual Graph mappings

> **Warning:** **Deprecated**: *stardog.admin.VirtualGraph.mappings_string()* should be preferred.

Parameters
content_type (str, default: 'text/turtle') – Content type for mappings.

Return type
bytes

Returns
Mappings in given content type

mappings_string(*syntax='STARDOG'*)

Returns graph mappings from virtual graph

Parameters
syntax (str, default: 'STARDOG') – The desired syntax of the mappings ('STARDOG', 'R2RML', or 'SMS2').

Returns
Mappings in desired syntax

property name: str

The name of the virtual graph.

options()

Gets virtual graph options.

Return type
Dict

update(*name*, *mappings*, *options={}*, *datasource=None*, *db=None*)

Updates the Virtual Graph.

Parameters

- **name** (str) – The new name
- **mappings** (*Content*) – New mapping contents
- **options** (Dict, default: {}) – New virtual graph options
- **datasource** (Optional[str], default: None) – new data source for the virtual graph
- **db** (Optional[str], default: None) – the database to associate with the virtual graph

Return type
None

Examples:

```
vg.update(
        name='users',
        mappings=File('mappings.ttl'),
        options={'jdbc.driver': 'com.mysql.jdbc.Driver'}
)
```

2.3 stardog.content

Content that can be loaded into Stardog.

class stardog.content.**Content**

Bases: object

Content base class.

class stardog.content.**File**(*file=None*, *content_type=None*, *content_encoding=None*, *name=None*, *fname=None*)

Bases: *Content*

File-based content.

__init__(*file=None*, *content_type=None*, *content_encoding=None*, *name=None*, *fname=None*)

Initializes a File object.

Parameters

- **file** (Optional[str], default: None) – the filename/path of the file
- **content_type** (Optional[str], default: None) – Content type. It will be automatically detected from the filename
- **content_encoding** (Optional[str], default: None) – Content encoding. It will be automatically detected from the filename
- **name** (Optional[str], default: None) – Name of the file object. It will be automatically detected from the filename
- **fname** (Optional[str], default: None) – backward compatible parameter for file

Examples

```
>>> File('data.ttl')
>>> File('data.doc', 'application/msword')
```

data()

class stardog.content.**ImportFile**(*file*, *input_type=None*, *content_type=None*, *content_encoding=None*, *separator=None*, *name=None*)

Bases: *Content*

File-based content for Delimited and JSON file.

__init__(*file*, *input_type=None*, *content_type=None*, *content_encoding=None*, *separator=None*, *name=None*)

Initializes a File object.

Parameters

- **file** (str) – filename/path of the file
- **input_type** (Optional[str], default: None) – 'DELIMITED' or 'JSON'
- **content_type** (Optional[str], default: None) – Content type
- **content_encoding** (Optional[str], default: None) – Content encoding

- **separator** (Optional[str], default: None) – Required if input_type is 'DELIMITED'. Use ',' for a CSV. Use \\t for a TSV.
- **name** (Optional[str], default: None) – Object name. It will be automatically detected from the file if omitted.

Note: If file has a recognized extension (i.e. 'data.csv', 'data.tsv', or 'data.json'), it will auto-detect most required parameters (input_type, separator, content_type, content_encoding) - otherwise you must specify them.

Examples

```
>>> ImportFile('data.csv')
>>> ImportFile('data.tsv')
>>> ImportFile('data.txt','DELIMITED',"\\t" )
>>> ImportFile('data.json')
```

data()

class stardog.content.**ImportRaw**(*content*, *input_type=None*, *separator=None*, *content_type=None*, *content_encoding=None*, *name=None*)

Bases: *Content*

User-defined content.

__init__(*content*, *input_type=None*, *separator=None*, *content_type=None*, *content_encoding=None*, *name=None*)

Initializes a Raw object.

Parameters

- **content** (object) – Object representing the content (e.g., str, file)
- **input_type** (Optional[str], default: None) – 'DELIMITED' or 'JSON'
- **separator** (Optional[str], default: None) – Required if input_type is 'DELIMITED'. Use ',' for a CSV. Use \\t for a TSV.
- **content_type** (Optional[str], default: None) – Content type
- **content_encoding** (Optional[str], default: None) – Content encoding
- **name** (Optional[str], default: None) – Object name

Note: if name is provided like a pseudo filename (i.e. 'data.csv', 'data.tsv', or 'data.json'), it will auto-detect most required parameters (input_type, separator, content_type, content_encoding) - otherwise you must specify them.

Examples

```
>>> ImportRaw('a,b,c',  name='data.csv')
>>> ImportRaw('a      b       c', name='data.tsv')
>>> ImportRaw({'foo':'bar'}, name='data.json')
```

data()

class stardog.content.**MappingFile**(*file*, *syntax=None*, *name=None*)

Bases: *Content*

File-based content.

__init__(*file*, *syntax=None*, *name=None*)

Initializes a File object.

Parameters

- **file** (str) – the filename/path of the file
- **syntax** (Optional[str], default: None) – The mapping syntax ('STARDOG', 'R2RML', or 'SMS2') If not provided, it will try to detect it from the file's extension.
- **name** (Optional[str], default: None) – the name of the object. If not provided, will fall back to the basename of the file.

Examples

```
>>> MappingFile('data.sms')
>>> MappingFile('data.sms2')
>>> MappingFile('data.rq')
>>> MappingFile('data.r2rml')
```

data()

class stardog.content.**MappingRaw**(*content*, *syntax=None*, *name=None*)

Bases: *Content*

User-defined Mapping.

__init__(*content*, *syntax=None*, *name=None*)

Initializes a MappingRaw object.

Parameters

- **content** (str) – the actual mapping content (e.g. 'MAPPING\n FROM SQL ...')
- **syntax** (Optional[str], default: None) – The mapping syntax ('STARDOG', 'R2RML', or 'SMS2') If not provided, it will try to detect it from name if provided, otherwise from the content itself
- **name** (Optional[str], default: None) – name of object

Examples

```
>>> mapping = '''
MAPPING
FROM SQL {
  SELECT *
  FROM `benchmark`.`person`
}
TO {
  ?subject rdf:type :person
} WHERE {
  BIND(template("http://api.stardog.com/person/nr={nr}") AS ?subject)
}
'''
>>> MappingRaw(mapping)
```

data()

class stardog.content.**Raw**(*content*, *content_type=None*, *content_encoding=None*, *name=None*)

Bases: *Content*

User-defined content.

__init__(*content*, *content_type=None*, *content_encoding=None*, *name=None*)

Initializes a Raw object.

Parameters

- **content** (object) – Object representing the content (e.g., str, file)
- **content_type** (Optional[str], default: None) – Content type
- **content_encoding** (Optional[str], default: None) – Content encoding
- **name** (Optional[str], default: None) – Object name

Examples

```
>>> Raw(':luke a :Human', 'text/turtle', name='data.ttl')
>>> Raw(':üäöoñr a :Employee .'.encode('utf-8'), 'text/turtle')
```

data()

class stardog.content.**URL**(*url*, *content_type=None*, *content_encoding=None*, *name=None*)

Bases: *Content*

Url-based content.

__init__(*url*, *content_type=None*, *content_encoding=None*, *name=None*)

Initializes a URL object.

Parameters

- **url** (str) – URL to the content
- **content_type** (Optional[str], default: None) – Content type. It will be automatically detected from the url if not provided.

- **content_encoding** (Optional[str], default: None) – Content encoding. It will be automatically detected from the url if not provided.
- **name** (Optional[str], default: None) – Object name. It will be automatically detected from the url if not provided.

Examples

```
>>> URL('http://example.com/data.ttl')
>>> URL('http://example.com/data.doc', 'application/msword')
```

data()

2.4 stardog.exceptions

exception stardog.exceptions.**StardogException**(*message*, *http_code=None*, *stardog_code=None*)

Bases: Exception

General Stardog Exceptions

exception stardog.exceptions.**TransactionException**(*message*, *http_code=None*, *stardog_code=None*)

Bases: *StardogException*

Transaction Exceptions

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

stardog.admin, 22
stardog.connection, 7
stardog.content, 48
stardog.exceptions, 52

INDEX

Symbols

__init__() (*stardog.admin.Admin method*), 22
__init__() (*stardog.admin.Cache method*), 34
__init__() (*stardog.admin.CacheTarget method*), 35
__init__() (*stardog.admin.DataSource method*), 35
__init__() (*stardog.admin.Database method*), 37
__init__() (*stardog.admin.Role method*), 40
__init__() (*stardog.admin.StoredQuery method*), 42
__init__() (*stardog.admin.User method*), 43
__init__() (*stardog.admin.VirtualGraph method*), 46
__init__() (*stardog.connection.Connection method*), 7
__init__() (*stardog.connection.Docs method*), 16
__init__() (*stardog.connection.GraphQL method*), 17
__init__() (*stardog.connection.ICV method*), 19
__init__() (*stardog.content.File method*), 48
__init__() (*stardog.content.ImportFile method*), 48
__init__() (*stardog.content.ImportRaw method*), 49
__init__() (*stardog.content.MappingFile method*), 50
__init__() (*stardog.content.MappingRaw method*), 50
__init__() (*stardog.content.Raw method*), 51
__init__() (*stardog.content.URL method*), 51

A

add() (*stardog.connection.Connection method*), 8
add() (*stardog.connection.Docs method*), 16
add() (*stardog.connection.ICV method*), 19
add_namespace() (*stardog.admin.Database method*), 37
add_permission() (*stardog.admin.Role method*), 40
add_permission() (*stardog.admin.User method*), 43
add_role() (*stardog.admin.User method*), 43
add_schema() (*stardog.connection.GraphQL method*), 17
added (*stardog.connection.CommitResult attribute*), 7
Admin (*class in stardog.admin*), 22
alive() (*stardog.admin.Admin method*), 22
ask() (*stardog.connection.Connection method*), 8
available() (*stardog.admin.DataSource method*), 35
available() (*stardog.admin.VirtualGraph method*), 46

B

backup() (*stardog.admin.Database method*), 37
backup_all() (*stardog.admin.Admin method*), 22
begin() (*stardog.connection.Connection method*), 9

C

Cache (*class in stardog.admin*), 34
cache() (*stardog.admin.Admin method*), 23
cache_status() (*stardog.admin.Admin method*), 23
cache_targets() (*stardog.admin.Admin method*), 23
cached_graphs() (*stardog.admin.Admin method*), 23
cached_queries() (*stardog.admin.Admin method*), 23
cached_status() (*stardog.admin.Admin method*), 23
CacheTarget (*class in stardog.admin*), 35
clear() (*stardog.connection.Connection method*), 9
clear() (*stardog.connection.Docs method*), 16
clear() (*stardog.connection.ICV method*), 19
clear_schemas() (*stardog.connection.GraphQL method*), 18
clear_stored_queries() (*stardog.admin.Admin method*), 24
close() (*stardog.connection.Connection method*), 10
cluster_coordinator_check() (*stardog.admin.Admin method*), 24
cluster_info() (*stardog.admin.Admin method*), 24
cluster_join() (*stardog.admin.Admin method*), 24
cluster_list_standby_nodes() (*stardog.admin.Admin method*), 24
cluster_revoke_standby_access() (*stardog.admin.Admin method*), 24
cluster_shutdown() (*stardog.admin.Admin method*), 24
cluster_start_readonly() (*stardog.admin.Admin method*), 24
cluster_status() (*stardog.admin.Admin method*), 25
cluster_stop_readonly() (*stardog.admin.Admin method*), 25
commit() (*stardog.connection.Connection method*), 10
CommitResult (*class in stardog.connection*), 7
Connection (*class in stardog.connection*), 7
Content (*class in stardog.content*), 48
convert() (*stardog.connection.ICV method*), 19
copy() (*stardog.admin.Database method*), 38
creator (*stardog.admin.StoredQuery property*), 42

D

data() (*stardog.content.File method*), 48
data() (*stardog.content.ImportFile method*), 49
data() (*stardog.content.ImportRaw method*), 50
data() (*stardog.content.MappingFile method*), 50
data() (*stardog.content.MappingRaw method*), 51
data() (*stardog.content.Raw method*), 51
data() (*stardog.content.URL method*), 52
Database (*class in stardog.admin*), 37
database (*stardog.admin.StoredQuery property*), 42
database() (*stardog.admin.Admin method*), 25
databases() (*stardog.admin.Admin method*), 25
DataSource (*class in stardog.admin*), 35
datasource() (*stardog.admin.Admin method*), 25
datasources() (*stardog.admin.Admin method*), 25
datasources_info() (*stardog.admin.Admin method*), 25
delete() (*stardog.admin.DataSource method*), 35
delete() (*stardog.admin.Role method*), 41
delete() (*stardog.admin.StoredQuery method*), 42
delete() (*stardog.admin.User method*), 44
delete() (*stardog.admin.VirtualGraph method*), 46
delete() (*stardog.connection.Docs method*), 16
description (*stardog.admin.StoredQuery property*), 42
Docs (*class in stardog.connection*), 16
docs() (*stardog.connection.Connection method*), 10
drop() (*stardog.admin.Cache method*), 34
drop() (*stardog.admin.Database method*), 38

E

effective_permissions() (*stardog.admin.User method*), 44
explain() (*stardog.connection.Connection method*), 10
explain_inconsistency() (*stardog.connection.Connection method*), 10
explain_inference() (*stardog.connection.Connection method*), 10
explain_violations() (*stardog.connection.ICV method*), 20
export() (*stardog.connection.Connection method*), 11

F

File (*class in stardog.content*), 48

G

get() (*stardog.connection.Docs method*), 16
get_all_metadata_properties() (*stardog.admin.Admin method*), 25
get_all_options() (*stardog.admin.Database method*), 38
get_database() (*stardog.admin.VirtualGraph method*), 46
get_datasource() (*stardog.admin.VirtualGraph method*), 46
get_options() (*stardog.admin.Database method*), 38
get_options() (*stardog.admin.DataSource method*), 36
get_prometheus_metrics() (*stardog.admin.Admin method*), 26
get_server_metrics() (*stardog.admin.Admin method*), 26
get_server_properties() (*stardog.admin.Admin method*), 26
graph() (*stardog.connection.Connection method*), 11
GraphQL (*class in stardog.connection*), 17
graphql() (*stardog.connection.Connection method*), 12

H

healthcheck() (*stardog.admin.Admin method*), 26

I

ICV (*class in stardog.connection*), 19
icv() (*stardog.connection.Connection method*), 12
import_file() (*stardog.admin.Admin method*), 26
import_namespaces() (*stardog.admin.Database method*), 39
import_virtual_graph() (*stardog.admin.Admin method*), 27
ImportFile (*class in stardog.content*), 48
ImportRaw (*class in stardog.content*), 49
info() (*stardog.admin.CacheTarget method*), 35
info() (*stardog.admin.DataSource method*), 36
info() (*stardog.admin.VirtualGraph method*), 46
is_consistent() (*stardog.connection.Connection method*), 12
is_enabled() (*stardog.admin.User method*), 44
is_superuser() (*stardog.admin.User method*), 44
is_valid() (*stardog.connection.ICV method*), 20

K

kill_query() (*stardog.admin.Admin method*), 27

L

list() (*stardog.connection.ICV method*), 21

M

MappingFile (*class in stardog.content*), 50
MappingRaw (*class in stardog.content*), 50
mappings() (*stardog.admin.VirtualGraph method*), 46
mappings_string() (*stardog.admin.VirtualGraph method*), 47
materialize_virtual_graph() (*stardog.admin.Admin method*), 27
module
    stardog.admin, 22

stardog.connection, 7
stardog.content, 48
stardog.exceptions, 52

N

name (*stardog.admin.CacheTarget property*), 35
name (*stardog.admin.Database property*), 39
name (*stardog.admin.DataSource property*), 36
name (*stardog.admin.Role property*), 41
name (*stardog.admin.StoredQuery property*), 42
name (*stardog.admin.User property*), 44
name (*stardog.admin.VirtualGraph property*), 47
namespaces() (*stardog.admin.Database method*), 39
new_cache_target() (*stardog.admin.Admin method*), 28
new_cached_graph() (*stardog.admin.Admin method*), 28
new_cached_query() (*stardog.admin.Admin method*), 29
new_database() (*stardog.admin.Admin method*), 29
new_datasource() (*stardog.admin.Admin method*), 30
new_role() (*stardog.admin.Admin method*), 30
new_stored_query() (*stardog.admin.Admin method*), 30
new_user() (*stardog.admin.Admin method*), 31
new_virtual_graph() (*stardog.admin.Admin method*), 31

O

offline() (*stardog.admin.Database method*), 39
online() (*stardog.admin.Database method*), 39
online() (*stardog.admin.DataSource method*), 36
optimize() (*stardog.admin.Database method*), 39
options() (*stardog.admin.VirtualGraph method*), 47
orphan() (*stardog.admin.CacheTarget method*), 35

P

paths() (*stardog.connection.Connection method*), 13
permissions() (*stardog.admin.Role method*), 41
permissions() (*stardog.admin.User method*), 44

Q

queries() (*stardog.admin.Admin method*), 32
query (*stardog.admin.StoredQuery property*), 42
query() (*stardog.admin.Admin method*), 32
query() (*stardog.connection.GraphQL method*), 18

R

Raw (*class in stardog.content*), 51
reasoning (*stardog.admin.StoredQuery property*), 42
refresh() (*stardog.admin.Cache method*), 34
refresh_count() (*stardog.admin.DataSource method*), 36
refresh_metadata() (*stardog.admin.DataSource method*), 36
remove() (*stardog.admin.CacheTarget method*), 35
remove() (*stardog.connection.Connection method*), 13
remove() (*stardog.connection.ICV method*), 21
remove_namespace() (*stardog.admin.Database method*), 39
remove_permission() (*stardog.admin.Role method*), 41
remove_permission() (*stardog.admin.User method*), 44
remove_role() (*stardog.admin.User method*), 45
remove_schema() (*stardog.connection.GraphQL method*), 18
removed (*stardog.connection.CommitResult attribute*), 7
repair() (*stardog.admin.Database method*), 40
report() (*stardog.connection.ICV method*), 21
restore() (*stardog.admin.Admin method*), 32
Role (*class in stardog.admin*), 40
role() (*stardog.admin.Admin method*), 32
roles() (*stardog.admin.Admin method*), 33
roles() (*stardog.admin.User method*), 45
rollback() (*stardog.connection.Connection method*), 14

S

schema() (*stardog.connection.GraphQL method*), 18
schemas() (*stardog.connection.GraphQL method*), 18
select() (*stardog.connection.Connection method*), 14
set_enabled() (*stardog.admin.User method*), 45
set_options() (*stardog.admin.Database method*), 40
set_password() (*stardog.admin.User method*), 45
set_roles() (*stardog.admin.User method*), 45
share() (*stardog.admin.DataSource method*), 37
shared (*stardog.admin.StoredQuery property*), 42
shutdown() (*stardog.admin.Admin method*), 33
size() (*stardog.connection.Connection method*), 15
size() (*stardog.connection.Docs method*), 17
standby_node_pause() (*stardog.admin.Admin method*), 33
standby_node_pause_status() (*stardog.admin.Admin method*), 33
stardog.admin
    module, 22
stardog.connection
    module, 7
stardog.content
    module, 48
stardog.exceptions
    module, 52
StardogException, 52
status() (*stardog.admin.Cache method*), 35
stored_queries() (*stardog.admin.Admin method*), 33
stored_query() (*stardog.admin.Admin method*), 33

StoredQuery (*class in stardog.admin*), 41

T

TransactionException, 52

U

update() (*stardog.admin.DataSource method*), 37
update() (*stardog.admin.StoredQuery method*), 42
update() (*stardog.admin.VirtualGraph method*), 47
update() (*stardog.connection.Connection method*), 15
URL (*class in stardog.content*), 51
User (*class in stardog.admin*), 43
user() (*stardog.admin.Admin method*), 33
users() (*stardog.admin.Admin method*), 34
users() (*stardog.admin.Role method*), 41

V

validate() (*stardog.admin.Admin method*), 34
verify() (*stardog.admin.Database method*), 40
virtual_graph() (*stardog.admin.Admin method*), 34
virtual_graphs() (*stardog.admin.Admin method*), 34
VirtualGraph (*class in stardog.admin*), 46