nosqlapi

Release 1.0.2

Matteo Guadrini

Apr 05, 2022

CONTENTS:

This library is defined to encourage similarity between Python modules used to access **NOSQL** databases. In this way, I hope for consistency that leads to more easily understood modules, code that generally gets more portability across databases and a broader scope of database connectivity from Python. This document describes the *Python NOSQL database API* specification.

CHAPTER

ONE

SUPPORTED NOSQL DATABASE TYPES

NoSql databases are of four types:

- Key/Value database
- Column database
- Document database
- Graph database

For each type of database, *nosqlapi* offers standard interfaces, in order to unify as much as possible the names of methods and functions.

1.1 Key-Value database

A **key–value database**, or key–value store, is a data storage paradigm designed for storing, retrieving, and managing associative arrays, and a data structure more commonly known today as a dictionary or hash table. Dictionaries contain a collection of objects, or records, which in turn have many different fields within them, each containing data. These records are stored and retrieved using a key that uniquely identifies the record, and is used to find the data within the database.

1.2 Column database

A **column-oriented DBMS** or columnar DBMS is a database management system (DBMS) that stores data tables by column rather than by row. Practical use of a column store versus a row store differs little in the relational DBMS world. Both columnar and row databases can use traditional database query languages like SQL to load data and perform queries. Both row and columnar databases can become the backbone in a system to serve data for common extract, transform, load (ETL) and data visualization tools. However, by storing data in columns rather than rows, the database can more precisely access the data it needs to answer a query rather than scanning and discarding unwanted data in rows.

1.3 Document database

A **document-oriented database**, or document store, is a computer program and data storage system designed for storing, retrieving and managing document-oriented information, also known as semi-structured data. Document-oriented databases are one of the main categories of NoSQL databases, and the popularity of the term *"document-oriented database"* has grown with the use of the term NoSQL itself. Graph databases are similar, but add another layer, the relationship, which allows them to link documents for rapid traversal.

1.4 Graph database

Graph databases are a type of NoSQL database, created to address the limitations of relational databases. While the graph model explicitly lays out the dependencies between nodes of data, the relational model and other NoSQL database models link the data by implicit connections. In other words, relationships are a first-class citizen in a graph database and can be labelled, directed, and given properties. This is compared to relational approaches where these relationships are implied and must be reified at run-time. Graph databases are similar to 1970s network model databases in that both represent general graphs, but network-model databases operate at a lower level of abstraction and lack easy traversal over a chain of edges.

1.4.1 Interface

Access and CRUD operations in the various types of databases are standardized in two objects: `Connection` and `Session` objects.

Constructors

Access to the database is made available through `Connection` objects. The module must provide the following constructor for these:

```
>>> connection = Connection(*args, **kwargs)
```

The `Connection` object allows you to access the database and operate directly at the database server level. This object has a `connect` method that returns a `Session` object. It takes a number of parameters which are database dependent.

```
>>> session = connection.connect(*args, **kwargs)
```

With `Session` object you can operate directly on the database data specified when creating the `Connection` object.

Note: This division of objects allows you to have more flexibility in case you want to change databases. It is not necessary to create a new object to connect to a different database.

Globals

`api_level` is a global variable to check compatibility with the names defined in this document. Currently the level is *1.0*.

`CONNECTION` is a global variable where to save a global `Connection` object.

`SESSION` is a global variable where to save a global `Session` object.

Exceptions

All defined exceptions derive from the general exception `Error` based on `Exception` class, which tends not to be used directly.

```
>>> raise Error("Don't use this, but its subclasses!")
```

Name	Base	Description
`Error`	`Exception`	Exception that is the base class of all other error exceptions. Use only for checks.
`UnknownError`	`Error`	Exception raised when an unspecified error occurred.
`ConnectError`	`Error`	Exception raised for errors that are related to the database connection.
`CloseError`	`Error`	Exception raised for errors that are related to the database close connection.
`DatabaseError`	`Error`	Exception raised for errors that are related to the database, generally.
`DatabaseCreationError`	`DatabaseError`	Exception raised for errors that are related to the creation of a database.
`DatabaseDeletionError`	`DatabaseError`	Exception raised for errors that are related to the deletion of a database.
`SessionError`	`Error`	Exception raised for errors that are related to the session, generally.
`SessionInsertingError`	`SessionError`	Exception raised for errors that are related to the inserting data on a database session.
`SessionUpdatingError`	`SessionError`	Exception raised for errors that are related to the updating data on a database session.
`SessionDeletingError`	`SessionError`	Exception raised for errors that are related to the deletion data on a database session.
`SessionClosingError`	`SessionError`	Exception raised for errors that are related to the closing database session.
`SessionFindingError`	`SessionError`	Exception raised for errors that are related to the finding data on a database session.
`SessionACLError`	`SessionError`	Exception raised for errors that are related to the grant or revoke permission on a database.
`SelectorError`	`Error`	Exception raised for errors that are related to the selectors in general.
`SelectorAttributeError`	`SelectorError`	Exception raised for errors that are related to the selectors attribute.

The tree of exceptions:

```
Exception
|__Error
   |__UnknownError
   |__ConnectError
   |__CloseError
   |__DatabaseError
   |   |__DatabaseCreationError
   |   |__DatabaseDeletionError
```

(continues on next page)

(continued from previous page)

```
   |__SessionError
   |   |__SessionInsertingError
   |   |__SessionUpdatingError
   |   |__SessionDeletingError
   |   |__SessionClosingError
   |   |__SessionFindingError
   |   |__SessionACLError
   |__SelectorError
      |__SelectorAttributeError
```

Selectors

NOSQL databases do not use SQL syntax, or if they do, it is encapsulated in a shell or interpreter. The selection queries will be driven through `Selector` objects which will then be passed to the `find` method of a `Session` object.

```
>>> selector = Selector(*args, **kwargs)
>>> session.find(selector)
```

Note: A string representing the language of the selector can also be passed to the find method.

1.4.2 API

The following describes the object APIs that will build a connection to databases. All the method names that are present below are common to all Nosql database types.

Note: Some methods are specific to certain types of databases, so they will not be present in all objects.

Connection Objects

The `Connection` object creates a layer concerning the connection to the server. You can not specify any specific databases; in this way, the object will be used for the creation and management of all databases at the server level.

Connection attributes

`connected`

This read-only attribute contains a boolean value representing whether the connection has been established with the server.

Connection methods

close(**args*, ***kwargs*)

Immediate closure of the connection and session with the database. Returns `None`.

connect(**args*, ***kwargs*)

Create a persistent session with a specific database. Returns an object of type `Session`.

> **Attention:** A valid `Session` object must necessarily be instantiated with a valid *database* name.

create_database(**args*, ***kwargs*)

Create a single database. Returns an object of type `Response`.

has_database(**args*, ***kwargs*)

Check if exists a single database. Returns an object of type `Response`.

delete_database(**args*, ***kwargs*)

Delete of a single database. Returns an object of type `Response`.

databases(**args*, ***kwargs*)

List all databases. Returns an object of type `Response`.

show_database(**args*, ***kwargs*)

Show an information of a specific database. Returns an object of type `Response`.

Session Objects

The `Session` object represents the *session* to a database. With this object it is possible to operate directly on the data.

> **Warning:** This object is derived from the `connect()` method of a `Connection` object. It is also possible to instantiate the session object through a connect function, but it will be mandatory to specify a *database*.

Session attributes

connection

`Connection` or other compliant object of server in current session

description

This read-only attribute contains the *session* parameters (can be `str`, `tuple` or `dict`).

item_count

This read-only attribute contains the number of object returned of an operations (must be `int`).

database

This read-only attribute contains the name of database in current session (must be `str`).

`acl`

This read-only attribute contains the Access Control List in the current session (must be `tuple`, `dict` or `Response` object).

`indexes`

This read-only attribute contains the name of indexes of the current database (must be `tuple`, `dict` or `Response` object).

Session methods

`get`(**args*, ***kwargs*)

Get one or more data from specific database. Returns an object of type `Response`.

> **Attention:** This is the letter *R* of CRUD operations.

`insert`(**args*, ***kwargs*)

Insert one data on specific database. Returns an object of type `Response`.

> **Attention:** This is the letter *C* of CRUD operations.

`insert_many`(**args*, ***kwargs*)

Insert one or more data on specific database. Returns an object of type `Response`.

`update`(**args*, ***kwargs*)

Update one existing data on specific database. Returns an object of type `Response`.

> **Attention:** This is the letter *U* of CRUD operations.

`update_many`(**args*, ***kwargs*)

Update one or more existing data on specific database. Returns an object of type `Response`.

`delete`(**args*, ***kwargs*)

Delete one existing data on specific database. Returns an object of type `Response`.

> **Attention:** This is the letter *D* of CRUD operations.

`close`(**args*, ***kwargs*)

Close immediately current session. Returns `None`.

`find`(**args*, ***kwargs*)

Find data on specific database with `str` selector or `Selector` object. Returns an object of type `Response`.

grant(**args*, ***kwargs*)

Grant *Access Control List* on specific database. Returns an object of type `Response`.

revoke(**args*, ***kwargs*)

Revoke *Access Control List* on specific database. Returns an object of type `Response`

new_user(**args*, ***kwargs*)

Create new normal or admin user. Returns an object of type `Response`

set_user(**args*, ***kwargs*)

Modify exists user or reset password. Returns an object of type `Response`

delete_user(**args*, ***kwargs*)

Delete exists user. Returns an object of type `Response`

add_index(**args*, ***kwargs*)

Add a new index to database. Returns an object of type `Response`

delete_index(**args*, ***kwargs*)

Delete exists index to database. Returns an object of type `Response`

call(*batch*, **args*, ***kwargs*)

Call a `Batch` object to execute one or more statement. Returns an object of type `Response`

Selector Objects

The `Selector` object represents the *query* string for find data from database. Once instantiated, it can be an input to the `Session`'s find method.

Selector attributes

selector

This read/write attribute represents the selector key/value than you want search.

fields

This read/write attribute represents the fields key that returned from find operations.

partition

This read/write attribute represents the name of partition/collection in a database.

condition

This read/write attribute represents other condition to apply a selectors.

order

This read/write attribute represents order returned from find operations.

limit

This read/write attribute represents limit number of objects returned from find operations.

Selector methods

build(**args*, ***kwargs*)

Building a selector string in the dialect of a NOSQL language based on various property of the `Selector` object. Returns `str`.

Response Objects

The `Response` object represents the response that comes from an operation from the database.

Note: `Response` objects is a species of an either-data type, because contains both *success* and *error* values.

Response attributes

data

This read-only attribute represents the effective python data (`Any`) than returned from an operation from the database.

code

This read-only attribute represents a number code (`int`) of error or success in an operation.

header

This read-only attribute represents a `str` information (header) of an operation.

error

This read-only attribute represents a `str` or `Exception` object error of an operation.

dict

This read-only attribute represents a `dict` transformation of `Response` object.

Response methods

throw()

Raise exception stored in `error` property.

Batch Objects

The `Batch` object represents an aggregation of CRUD operations.

Batch attributes

session

This read/write attribute represents a `Session` object.

batch

This read/write attribute represents a *batch* operation.

Batch methods

execute(**args*, ***kwargs*)

Executing a batch operation with position and keyword arguments. Returns `tuple` or an object of type `Response`

1.4.3 nosqlapi package

The package nosqlapi is a collection of interface, utility class and functions for build your own NOSQL python package.

This library offers API-based interfaces described above, helping to build a more coherent and integrated python library for a NOSQL database, with names similar to another library using the same interfaces.

The benefit of using `nosqlapi` is to standardize the names and syntax for the end user, so as to make as few changes as possible to your software.

Installation

To install the nosqlapi package, run `pip` as follows:

```
$ pip install nosqlapi #from pypi

$ git clone https://github.com/MatteoGuadrini/nosqlapi.git #from official repo
$ cd nosqlapi
$ python setup.py install
```

nosqlapi common

In this package you will find the modules that contain common interfaces and ODMs (Object-Data Mapping), shared by all four types of NOSQL database.

core module

In the **core** module, we find the abstract classes that form the basis of the classes of all four NOSQL database types.

Note: In theory, you shouldn't use these classes to build your class module for the NOSQL database you plan to build, but you could; in all classes you have the necessary methods to iterate with the database server.

Module that contains the core objects.

class nosqlapi.common.core.**Batch**(*batch*, *session=None*)

Bases: abc.ABC

Batch abstract class

The abstract class *Batch* is used to create batch-type classes that represent a collection of operations to be performed at the same time.

__init__(*batch*, *session=None*)

Instantiate Batch object

Parameters

- **batch** – List of commands
- **session** – Session object or other compliant object

__repr__()

Return repr(self).

__str__()

Return str(self).

__weakref__

list of weak references to the object (if defined)

property batch

String batch operation

abstract execute(**args*, ***kwargs*)

Execute some batch statement

Returns Union[tuple, Response]

property session

Session object

class nosqlapi.common.core.**Connection**(*host=None*, *user=None*, *password=None*, *database=None*, *port=None*, *bind_address=None*, *read_timeout=None*, *write_timeout=None*, *ssl=None*, *ssl_ca=None*, *ssl_cert=None*, *tls=None*, *ssl_key=None*, *ssl_verify_cert=None*, *max_allowed_packet=None*)

Bases: abc.ABC

Server connection abstract class

The abstract class *Connection* is used to create connection-type classes that will allow you to work directly on the layer at the database level.

__init__(*host=None*, *user=None*, *password=None*, *database=None*, *port=None*, *bind_address=None*, *read_timeout=None*, *write_timeout=None*, *ssl=None*, *ssl_ca=None*, *ssl_cert=None*, *tls=None*, *ssl_key=None*, *ssl_verify_cert=None*, *max_allowed_packet=None*)

Instantiate Connection object

Parameters

- **host** – Name of host that contains database
- **user** – Username for connect to the host
- **password** – Password for connect to the host

- **database** – Name of database
- **port** – Tcp port
- **bind_address** – Hostname or an IP address for multiple network interfaces
- **read_timeout** – Timeout for reading from the connection in seconds
- **write_timeout** – Timeout for writing from the connection in seconds
- **ssl** – Ssl connection established
- **ssl_ca** – Ssl CA file specified
- **ssl_cert** – Ssl certificate file specified
- **tls** – Tls connection established
- **ssl_key** – Ssl private key file specified
- **ssl_verify_cert** – Verify certificate file
- **max_allowed_packet** – Max size of packet sent to server in bytes

__repr__()

Return repr(self).

__str__()

Return str(self).

__weakref__

list of weak references to the object (if defined)

abstract close(**args*, ***kwargs*)

Close connection

Returns None

abstract connect(**args*, ***kwargs*)

Connect database server

Returns Session object

property connected

Boolean representing the database connection

Returns bool

abstract create_database(**args*, ***kwargs*)

Create new database on server

Returns Union[bool, Response]

abstract databases(**args*, ***kwargs*)

Get all databases

Returns Union[tuple, list, Response]

abstract delete_database(**args*, ***kwargs*)

Delete database on server

Returns Union[bool, Response]

abstract has_database(**args*, ***kwargs*)

Check if database exists

Returns Union[bool, Response]

abstract show_database(**args*, ***kwargs*)

Show a database information

:return : Union[Any, Response]

class nosqlapi.common.core.**Response**(*data*, *code=None*, *header=None*, *error=None*)

Bases: abc.ABC

Server response abstract class

The abstract class *Response* is used to create response-type classes that represent response data to an operation.

__init__(*data*, *code=None*, *header=None*, *error=None*)

Instantiate Response object

Parameters

- **data** – Data from operation
- **code** – Exit code of operation
- **header** – Header of operation
- **error** – Error string or Exception class

__repr__()

Return repr(self).

__str__()

Return str(self).

property code

Number code of error or success in an operation

property data

The effective data than returned

property dict

dict format for Response object

property error

Error of an operation

property header

Information (header) of an operation

throw()

Raise or throw exception from error property

Returns Exception

class nosqlapi.common.core.**Selector**(*selector=None*, *fields=None*, *partition=None*, *condition=None*, *order=None*, *limit=None*)

Bases: abc.ABC

Selector abstract class

The abstract class *Selector* is used to create selector-type classes, which represent a query in the specific language of the database.

__init__(*selector=None*, *fields=None*, *partition=None*, *condition=None*, *order=None*, *limit=None*)

Instantiate Selector object

Parameters

- **selector** – Selector part of the query string
- **fields** – Return fields
- **partition** – Partition or collection of data
- **condition** – Condition of query
- **order** – Order by specific selector
- **limit** – Limit result

__repr__()

Return repr(self).

__str__()

Return str(self).

__weakref__

list of weak references to the object (if defined)

abstract build(**args*, ***kwargs*)

Build string query selector

Returns string

property condition

Other condition to apply a selectors

property fields

Key that returned from find operations

property limit

Limit number of objects returned from find operations

property order

Order returned from find operations

property partition

The name of partition or collection in a database

property selector

Value than you want search

class nosqlapi.common.core.**Session**(*connection*, *database=None*)

Bases: abc.ABC

Server session abstract class

The abstract class *Session* is used to create session-type classes that will allow you to work directly on the layer at the data level.

__init__(*connection*, *database=None*)

Instantiate Session object

Parameters

- **connection** – Connection object or other object to serve connection
- **database** – database name

__repr__()

Return repr(self).

__str__()

Return str(self).

__weakref__

list of weak references to the object (if defined)

abstract property acl

Access Control List in the current session

abstract add_index(**args*, ***kwargs*)

Add index to database

Returns Union[bool, Response]

static call(*batch*, **args*, ***kwargs*)

Call a batch

Returns Union[Any, Response]

abstract close(**args*, ***kwargs*)

Close session

Returns None

property connection

Connection of server in current session

property database

Name of database in current session

abstract delete(**args*, ***kwargs*)

Delete one value

Returns Union[bool, Response]

abstract delete_index(**args*, ***kwargs*)

Delete index to database

Returns Union[bool, Response]

abstract delete_user(**args*, ***kwargs*)

Delete exist user

Returns Union[bool, Response]

abstract property description

Contains the session parameters

abstract find(**args*, ***kwargs*)

Find data

Returns Union[tuple, Response]

abstract get(**args*, ***kwargs*)

Get one or more value

Returns Union[tuple, Response]

abstract grant(**args*, ***kwargs*)

Grant users ACLs

Returns Union[Any, Response]

abstract property indexes

Name of indexes of the current database

abstract insert(**args*, ***kwargs*)

Insert one value

Returns Union[bool, Response]

abstract insert_many(**args*, ***kwargs*)

Insert one or more value

Returns Union[bool, Response]

abstract property item_count

Number of item returned from latest CRUD operation

abstract new_user(**args*, ***kwargs*)

Create new user

Returns Union[bool, Response]

abstract revoke(**args*, ***kwargs*)

Revoke users ACLs

Returns Union[Any, Response]

abstract set_user(**args*, ***kwargs*)

Modify exist user

Returns Union[bool, Response]

abstract update(**args*, ***kwargs*)

Update one value

Returns Union[bool, Response]

abstract update_many(**args*, ***kwargs*)

Update one or more value

Returns Union[bool, Response]

core example

Classes on this module are *abstract* (interfaces), therefore, they cannot be instantiated as objects, but inherit them in their own classes.

```
# mymodule.py
import nosqlapi

# this module is my library of NOSQL database

class Connection(nosqlapi.common.Connection):
    def __init__(host=None, port=None, database=None, username=None, password=None,␣
↪ssl=None, tls=None, cert=None,
                 ca_cert=None, ca_bundle=None): ...
    def close(self, force=False): ...
    def connect(self, retry=1): ...
    def create_database(self, database, exists=False): ...
    def has_database(self, database): ...
    def databases(self, return_obj=None): ...
    def delete_database(self, force=False): ...
    def show_database(self, database): ...

conn = Connection('server.local', 1241, 'new_db', username='admin', password='pa$$w0rd',␣
↪ssl=True)
conn.create_database(conn.database)     # use 'database' attribute equals 'new_db'
if conn.has_database('new_db'):         # check if database exists
    sess = conn.connect()               # Session object
else:
    raise nosqlapi.common.exception.ConnectError(f'Connection error with database {conn.
↪database}')
...
```

exception module

In the **exception** module, we find the exceptions listed in the exceptions table in the interface documentation.

Note: You should never use the *Error* exception class as it is the basis for all exceptions. If you have a generic or unknown error, use *UnknownError* instead.

Exception module.

This module contains the hierarchy of exceptions included in the NOSQL API.

exception nosqlapi.common.exception.**CloseError**

Bases: *nosqlapi.common.exception.ConnectError*

Exception raised for errors that are related to the database close connection.

exception nosqlapi.common.exception.**ConnectError**

Bases: *nosqlapi.common.exception.Error*

Exception raised for errors that are related to the database connection.

exception nosqlapi.common.exception.**DatabaseCreationError**

Bases: *nosqlapi.common.exception.DatabaseError*

Exception raised for errors that are related to the creation of a database.

exception nosqlapi.common.exception.**DatabaseDeletionError**

Bases: *nosqlapi.common.exception.DatabaseError*

Exception raised for errors that are related to the deletion of a database.

exception nosqlapi.common.exception.**DatabaseError**

Bases: *nosqlapi.common.exception.Error*

Exception raised for errors that are related to the database, generally.

exception nosqlapi.common.exception.**Error**

Bases: Exception

Error that is the base class of all other error exceptions. You can use this to catch all errors with one single except statement.

exception nosqlapi.common.exception.**SelectorAttributeError**

Bases: *nosqlapi.common.exception.SelectorError*

Exception raised for errors that are related to the selectors attribute.

exception nosqlapi.common.exception.**SelectorError**

Bases: *nosqlapi.common.exception.Error*

Exception raised for errors that are related to the selectors in general.

exception nosqlapi.common.exception.**SessionACLError**

Bases: *nosqlapi.common.exception.SessionError*

Exception raised for errors that are related to the grant or revoke permission on a database.

exception nosqlapi.common.exception.**SessionClosingError**

Bases: *nosqlapi.common.exception.SessionError*

Exception raised for errors that are related to the closing database session.

exception nosqlapi.common.exception.**SessionDeletingError**

Bases: *nosqlapi.common.exception.SessionError*

Exception raised for errors that are related to the deletion data on a database session.

exception nosqlapi.common.exception.**SessionError**

Bases: *nosqlapi.common.exception.Error*

Exception raised for errors that are related to the session, generally.

exception nosqlapi.common.exception.**SessionFindingError**

Bases: *nosqlapi.common.exception.SessionError*

Exception raised for errors that are related to the finding data on a database session.

exception nosqlapi.common.exception.**SessionInsertingError**

Bases: *nosqlapi.common.exception.SessionError*

Exception raised for errors that are related to the inserting data on a database session.

exception nosqlapi.common.exception.**SessionUpdatingError**

Bases: *nosqlapi.common.exception.SessionError*

Exception raised for errors that are related to the updating data on a database session.

exception nosqlapi.common.exception.**UnknownError**

Bases: *nosqlapi.common.exception.Error*

Exception raised when an unspecified error occurred.

exception example

The exceptions stated in this form are used in certain circumstances. See the table of exceptions to find out more.

```
import nosqlapi

raise nosqlapi.common.exception.UnknownError('I have no idea what the hell happened!')
raise nosqlapi.UnknownError('in short')
```

odm module

In the **odm** module, we find generic classes represent real objects present in various NOSQL databases.

Note: These objects are not mandatory for the implementation of their own NOSQL module. Rather they serve to help the end user have some consistency in python.

Module that contains some ODM common objects.

nosqlapi.common.odm.**Array**

alias of *nosqlapi.common.odm.List*

class nosqlapi.common.odm.**Ascii**(*value=''*)

Bases: `str`

Represents ASCII string

__init__(*value=''*)

ASCII string

Parameters **value** – String printable characters

__weakref__

list of weak references to the object (if defined)

class nosqlapi.common.odm.**Blob**

Bases: `bytes`

Represents bytes

class nosqlapi.common.odm.**Boolean**(*value*)

Bases: `object`

Represents bool

__init__(*value*)

Boolean object

Parameters **value** – True of False

__repr__()

Return repr(self).

__weakref__

list of weak references to the object (if defined)

class nosqlapi.common.odm.**Counter**(*value=0*)

Bases: object

Represents integer counter

__init__(*value=0*)

Counter object

Parameters **value** – Integer (default 0)

__repr__()

Return repr(self).

__weakref__

list of weak references to the object (if defined)

decrement(*value=1*)

Decrement number

Parameters **value** – Number (default 1)

Returns None

increment(*value=1*)

Increment number

Parameters **value** – Number (default 1)

Returns None

class nosqlapi.common.odm.**Date**

Bases: datetime.date

Represents date in format %Y-%m-%d

__repr__()

Return repr(self).

__weakref__

list of weak references to the object (if defined)

class nosqlapi.common.odm.**Decimal**(*value='0'*, *context=None*)

Bases: decimal.Decimal

Represents decimal number

__weakref__

list of weak references to the object (if defined)

class nosqlapi.common.odm.**Double**(*x=0, /*)

Bases: float

Represents float

__weakref__

list of weak references to the object (if defined)

class nosqlapi.common.odm.**Duration**

Bases: datetime.timedelta

Represents duration ISO 8601 format: P[n]Y[n]M[n]DT[n]H[n]M[n]S

__repr__()

Return repr(self).

__weakref__

list of weak references to the object (if defined)

string_format()

ISO 8601 format: P[n]Y[n]M[n]DT[n]H[n]M[n]S

Returns str

class nosqlapi.common.odm.**Float**(*x=0, /*)

Bases: float

Represents float

__weakref__

list of weak references to the object (if defined)

class nosqlapi.common.odm.**Inet**(*ip*)

Bases: object

Represents ip address version 4 or 6 like string

__init__(*ip*)

Network ip address object

Parameters **ip** – String ip value

__repr__()

Return repr(self).

__weakref__

list of weak references to the object (if defined)

class nosqlapi.common.odm.**Int**(*number*)

Bases: int

Represents integer

__init__(*number*)

Integer object

Parameters **number** – Integer

__repr__()

Return repr(self).

class nosqlapi.common.odm.**List**(*iterable=()*, */*)

Bases: list

Represents list of objects

__weakref__

list of weak references to the object (if defined)

class nosqlapi.common.odm.**Map**

Bases: *dict*

Represents dict of objects

__weakref__

list of weak references to the object (if defined)

class nosqlapi.common.odm.**Null**

Bases: object

Represents None

__repr__()

Return repr(self).

__weakref__

list of weak references to the object (if defined)

class nosqlapi.common.odm.**SmallInt**(*number*)

Bases: *nosqlapi.common.odm.Int*

Represents small integer: -32767 to 32767

__init__(*number*)

Integer number from -32767 to 32767

Parameters **number** – Integer

class nosqlapi.common.odm.**Text**

Bases: str

Represents str

__weakref__

list of weak references to the object (if defined)

class nosqlapi.common.odm.**Time**

Bases: datetime.time

Represents time

__repr__()

Return repr(self).

__weakref__

list of weak references to the object (if defined)

class nosqlapi.common.odm.**Timestamp**

Bases: datetime.datetime

Represents datetime timestamp

__repr__()

Return repr(self).

__weakref__

list of weak references to the object (if defined)

class nosqlapi.common.odm.**Uuid**

Bases: object

Represents uuid version 1

__init__()

Uuid1 object

__repr__()

Return repr(self).

__weakref__

list of weak references to the object (if defined)

nosqlapi.common.odm.**Varchar**

alias of *nosqlapi.common.odm.Text*

nosqlapi.common.odm.**Varint**

alias of *nosqlapi.common.odm.Int*

odm example

The classes within the module are python representations of real objects in the database.

```
import nosqlapi

null = nosqlapi.common.odm.Null()     # compare with null object into database
null = nosqlapi.Null()                # in short
map_ = nosqlapi.Map()                 # like dict
inet = nosqlapi.Inet('192.168.1.1')   # ipv4/ipv6 addresses
```

utils module

In the **utils** module, we find classes and functions that help the end user's work.

Note: The functions and classes in this module are meant for the work of the end user and not for those who build libraries.

Utils function and classes for any type of NOSQL database

class nosqlapi.common.utils.**Manager**(*connection*, **args*, ***kwargs*)

Bases: object

Manager class for api compliant nosql database connection

__init__(*connection*, **args*, ***kwargs*)

`__repr__`()
Return repr(self).

`__str__`()
Return str(self).

`__weakref__`
list of weak references to the object (if defined)

`add_index`(**args*, ***kwargs*)
Add index to database

Returns Union[bool, Response]

`call`(**args*, ***kwargs*)
Call a batch

Returns Union[Any, Response]

`change`(*connection*, **args*, ***kwargs*)
Change connection type

Parameters

- **connection** – Connection object
- **args** – positional args of Connection object
- **kwargs** – keywords args of Connection object

Returns None

`close`(**args*, ***kwargs*)
Close session

Returns None

`create_database`(**args*, ***kwargs*)
Create new database

Returns Union[bool, Response]

`databases`(**args*, ***kwargs*)
Get all databases

Returns Union[tuple, list, Response]

`delete`(**args*, ***kwargs*)
Delete one value

Returns Union[bool, Response]

`delete_database`(**args*, ***kwargs*)
Delete database on server

Returns Union[bool, Response]

`delete_index`(**args*, ***kwargs*)
Delete index to database

Returns Union[bool, Response]

delete_user(**args*, ***kwargs*)

Delete exist user

Returns Union[bool, Response]

find(**args*, ***kwargs*)

Find data

Returns Union[tuple, Response]

get(**args*, ***kwargs*)

Get one or more value

Returns Union[tuple, Response]

grant(**args*, ***kwargs*)

Grant users ACLs

Returns Union[Any, Response]

has_database(**args*, ***kwargs*)

Check if database exists

Returns Union[bool, Response]

insert(**args*, ***kwargs*)

Insert one value

Returns Union[bool, Response]

insert_many(**args*, ***kwargs*)

Insert one or more value

Returns Union[bool, Response]

new_user(**args*, ***kwargs*)

Create new user

Returns Union[bool, Response]

revoke(**args*, ***kwargs*)

Revoke users ACLs

Returns Union[Any, Response]

set_user(**args*, ***kwargs*)

Modify exist user

Returns Union[bool, Response]

show_database(**args*, ***kwargs*)

Show a database information

:return : Union[Any, Response]

update(**args*, ***kwargs*)

Update one value

Returns Union[bool, Response]

update_many(**args*, ***kwargs*)

Update one or more value

Returns Union[bool, Response]

nosqlapi.common.utils.**api**(***methods*)

Decorator function to transform standard classes into API compliant classes

Parameters **methods** – method names that you want to bind to the methods of API compliant classes

Returns class

nosqlapi.common.utils.**apply_vendor**(*name*)

Apply new name of api name

Parameters **name** – name of vendor

Returns None

nosqlapi.common.utils.**cursor_response**(*resp*)

Transform nosql Response object to list of tuple, like a response of sql cursor object

Parameters **resp** – Response object or other compliant object

Returns List[tuple]

nosqlapi.common.utils.**global_session**(*connection*, **args*, ***kwargs*)

Global session

Parameters

- **connection** – Connection object or other compliant object
- **args** – positional arguments of connect method on Connection object
- **kwargs** – keywords arguments of connect method on Connection object

Returns None

nosqlapi.common.utils.**response**(*data*)

Simple response with data only

Parameters **data** – Data

Returns Response

utils example

This is an example of a `Manager` object, used to change manage multiple sessions to different types of databases.

```
import nosqlapi
import mymodule
import othermodule

connection = mymodule.Connection('server.local', 1241, 'new_db', username='admin',␣
↪password='pa$$w0rd', ssl=True)
manager = nosqlapi.common.utils.Manager(connection)

# Change connection and session
new_connection = othermodule.Connection('server2.local', 1241, 'other_db', username=
↪'admin', password='pa$$w0rd', ssl=True)
```

(continues on next page)

(continued from previous page)

```
manager.change(new_connection)

# use connection methods
manager.create_database('db')
manager.databases()

# use session methods
manager.acl
manager.get('key')
```

The `api` decorator function allows you to return existing classes so that the methods can match the NOSQL api described in this documentation.

```
import nosqlapi
import pymongo

@nosqlapi.common.utils.api(database_names='databases', drop_database='delete_database',␣
↪close_cursor='close')
class ApiConnection(pymongo.Connection): ...

connection = ApiConnection('localhost', 27017, 'test_database')

print(hasattr(connection, 'databases'))     # True
```

The `global_session` function allows you to instantiate a global connection and session.

```
import nosqlapi
import mymodule

connection = mymodule.Connection('server.local', 1241, 'new_db', username='admin',␣
↪password='pa$$w0rd', ssl=True)
nosqlapi.common.utils.global_session(connection)

print(nosqlapi.CONNECTION)  # Connection object
print(nosqlapi.SESSION)     # Session object
```

The `cursor_response` function allows you to convert a Response object into a classic list of tuples.

```
import nosqlapi
import mymodule

connection = mymodule.Connection('server.local', 1241, 'new_db', username='admin',␣
↪password='pa$$w0rd', ssl=True)
resp = connection.databases()

print(resp)                     # Response object
print(cursor_response(resp))    # [('db1', 'db2')]
```

The `apply_vendor` function allows you to rename representation object from *nosqlapi* to other name

```
>>> import nosqlapi
>>> class Response(nosqlapi.Response): ...
>>> resp = Response('some data')
```

(continues on next page)

(continued from previous page)

```
>>> resp
<nosqlapi Response object>
>>> nosqlapi.apply_vendor('pymongo')
>>> resp
<pymongo Response object>
```

nosqlapi key-value

In this package we find abstract classes and ODM classes concerning the **Key-Value** database types.

client module

The **client** module contains the specific classes for *key-value* databases and they are based on the core classes.

Client module for key-value NOSQL database.

class nosqlapi.kvdb.client.**KVBatch**(*batch*, *session=None*)

Bases: *nosqlapi.common.core.Batch*, abc.ABC

Key-value NOSQL database Batch class

class nosqlapi.kvdb.client.**KVConnection**(**args*, ***kwargs*)

Bases: *nosqlapi.common.core.Connection*, abc.ABC

Key-value NOSQL database Connection class

__init__(**args*, ***kwargs*)

Instantiate Connection object

Parameters

- **host** – Name of host that contains database
- **user** – Username for connect to the host
- **password** – Password for connect to the host
- **database** – Name of database
- **port** – Tcp port
- **bind_address** – Hostname or an IP address for multiple network interfaces
- **read_timeout** – Timeout for reading from the connection in seconds
- **write_timeout** – Timeout for writing from the connection in seconds
- **ssl** – Ssl connection established
- **ssl_ca** – Ssl CA file specified
- **ssl_cert** – Ssl certificate file specified
- **tls** – Tls connection established
- **ssl_key** – Ssl private key file specified
- **ssl_verify_cert** – Verify certificate file
- **max_allowed_packet** – Max size of packet sent to server in bytes

class nosqlapi.kvdb.client.**KVResponse**(*data*, *code=None*, *header=None*, *error=None*)

Bases: *nosqlapi.common.core.Response*, abc.ABC

Key-value NOSQL database Response class

__weakref__

list of weak references to the object (if defined)

class nosqlapi.kvdb.client.**KVSelector**(**args*, ***kwargs*)

Bases: *nosqlapi.common.core.Selector*, abc.ABC

Key-value NOSQL database Selector class

__init__(**args*, ***kwargs*)

Instantiate Selector object

Parameters

- **selector** – Selector part of the query string
- **fields** – Return fields
- **partition** – Partition or collection of data
- **condition** – Condition of query
- **order** – Order by specific selector
- **limit** – Limit result

__str__()

Return str(self).

abstract first_greater_or_equal(*key*)

First greater or equal key by selector key

Parameters **key** – key to search

Returns Union[str, list, tuple]

abstract first_greater_than(*key*)

First greater key by selector key

Parameters **key** – key to search

Returns Union[str, list, tuple]

abstract last_less_or_equal(*key*)

Last less or equal key by selector key

Parameters **key** – key to search

Returns Union[str, list, tuple]

abstract last_less_than(*key*)

Last less key by selector key

Parameters **key** – key to search

Returns Union[str, list, tuple]

class nosqlapi.kvdb.client.**KVSession**(*connection*, *database=None*)

Bases: *nosqlapi.common.core.Session*, abc.ABC

Key-value NOSQL database Session class

abstract copy(**args*, ***kwargs*)

Copy key to other key

Returns Union[bool, Response]

client example

This is an example of a library for connecting to a redis server.

```
# mykvdb.py
import nosqlapi

# this module is my library of NOSQL key-value database, like Redis database

class Connection(nosqlapi.kvdb.KVConnection):
    def __init__(host='localhost', port=6379, database=0, username=None, password=None,␣
↪ssl=None, tls=None,
                 cert=None, ca_cert=None, ca_bundle=None): ...
    def close(self): ...
    def connect(self, *args, **kwargs): ...
    def create_database(self):
        raise nosqlapi.DatabaseCreationError('See your server configuration file.')
    def has_database(self, database): ...
    def databases(self): ...
    def delete_database(self): ...
    def show_database(self, database): ...


class Session(nosqlapi.kvdb.KVSession):
    # define here all methods
    pass

conn = Connection('myredis.local', password='pa$$w0rd')
print(conn.databases())                 # (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
sess = conn.connect()                   # Session object
...
```

odm module

The **odm** module contains the specific object for *key-value* databases.

ODM module for key-value NOSQL database.

class nosqlapi.kvdb.odm.**ExpiredItem**(*key*, *value=None*, *ttl=None*)

Bases: *nosqlapi.kvdb.odm.Item*

Represents Item object with ttl expired time

__init__(*key*, *value=None*, *ttl=None*)

ExpiredItem object

Parameters

- **key** – Key of item

- **value** – Value of item
- **ttl** – Time to live of item

__repr__()

Return repr(self).

property ttl

Time to live of item

class nosqlapi.kvdb.odm.**Index**(*name*, *key*)

Bases: tuple

__getnewargs__()

Return self as a plain tuple. Used by copy and pickle.

static __new__(*_cls*, *name*, *key*)

Create new instance of Index(name, key)

__repr__()

Return a nicely formatted representation string

property key

Alias for field number 1

property name

Alias for field number 0

class nosqlapi.kvdb.odm.**Item**(*key*, *value=None*)

Bases: object

Represents key/value like a dictionary

__init__(*key*, *value=None*)

Item object

Parameters

- **key** – Key of item
- **value** – Value of item

__repr__()

Return repr(self).

__str__()

Return str(self).

__weakref__

list of weak references to the object (if defined)

get()

Get item

Returns dict

property key

Key of item

set(*key*, *value=None*)

Set item

Parameters

- **key** – Key of item
- **value** – Value of the key

Returns None

property value

Value of the key

class nosqlapi.kvdb.odm.**Keyspace**(*name*, *exists=False*)

Bases: object

Represents keyspace like database

__init__(*name*, *exists=False*)

Keyspace object

Parameters

- **name** – Name of keyspace
- **exists** – Existing keyspace (default False)

__repr__()

Return repr(self).

__str__()

Return str(self).

__weakref__

list of weak references to the object (if defined)

append(*item*)

Append item into store

Parameters **item** – Key/value item

Returns None

property exists

Existence of keyspace

property name

Name of keyspace

pop(*item=- 1*)

Remove item from the store

Parameters **item** – Index of item to remove

Returns None

property store

List of object into keyspace

class nosqlapi.kvdb.odm.**Subspace**(*name*, *sub=None*, *sep='.'*)

Bases: *nosqlapi.kvdb.odm.Keyspace*

Represents subspace of the keyspace

__init__(*name*, *sub=None*, *sep='.'*)

Subspace object

Parameters

- **name** – Name of keyspace
- **sub** – Subname of keyspace
- **sep** – Separation character

class nosqlapi.kvdb.odm.**Transaction**(*commands=None*)

Bases: object

Represents group of commands in a single step

__init__(*commands=None*)

Transaction object

Parameters **commands** – List of commands

__repr__()

Return repr(self).

__str__()

Return str(self).

__weakref__

list of weak references to the object (if defined)

add(*command*, *index=- 1*)

Add command to commands list

Parameters

- **command** – Command string
- **index** – Index to append command

Returns None

property commands

Command list

delete(*index=- 1*)

Remove command to command list

Parameters **index** – Index to remove command

Returns None

odm example

These objects represent the respective *key-value* in databases.

```
import nosqlapi

transaction = nosqlapi.kvdb.odm.Transaction()       # in short -> nosqlapi.kvdb.
↪Transaction()
# Add commands
transaction.add('ACL LIST')
transaction.add('ACL DELUSER test')
# Remove commands
transaction.delete(1)

item = nosqlapi.kvdb.Item('key', 'value')            # item key=value
```

nosqlapi column

In this package we find abstract classes and ODM classes concerning the **column** database types.

client module

The **client** module contains the specific classes for *column* databases and they are based on the core classes.

Client module for column NOSQL database.

class nosqlapi.columndb.client.**ColumnBatch**(*batch*, *session=None*)

Bases: *nosqlapi.common.core.Batch*, abc.ABC

Column NOSQL database Batch class

class nosqlapi.columndb.client.**ColumnConnection**(**args*, ***kwargs*)

Bases: *nosqlapi.common.core.Connection*, abc.ABC

Column NOSQL database Connection class

__init__(**args*, ***kwargs*)

Instantiate Connection object

Parameters

- **host** – Name of host that contains database
- **user** – Username for connect to the host
- **password** – Password for connect to the host
- **database** – Name of database
- **port** – Tcp port
- **bind_address** – Hostname or an IP address for multiple network interfaces
- **read_timeout** – Timeout for reading from the connection in seconds
- **write_timeout** – Timeout for writing from the connection in seconds
- **ssl** – Ssl connection established

- **ssl_ca** – Ssl CA file specified
- **ssl_cert** – Ssl certificate file specified
- **tls** – Tls connection established
- **ssl_key** – Ssl private key file specified
- **ssl_verify_cert** – Verify certificate file
- **max_allowed_packet** – Max size of packet sent to server in bytes

class nosqlapi.columndb.client.**ColumnResponse**(*data*, *code=None*, *header=None*, *error=None*)

Bases: *nosqlapi.common.core.Response*, abc.ABC

Column NOSQL database Response class

__weakref__

list of weak references to the object (if defined)

class nosqlapi.columndb.client.**ColumnSelector**(**args*, ***kwargs*)

Bases: *nosqlapi.common.core.Selector*, abc.ABC

Column NOSQL database Selector class

__init__(**args*, ***kwargs*)

Instantiate Selector object

Parameters

- **selector** – Selector part of the query string
- **fields** – Return fields
- **partition** – Partition or collection of data
- **condition** – Condition of query
- **order** – Order by specific selector
- **limit** – Limit result

abstract alias(**args*, ***kwargs*)

Aliases the selector: SELECT col1 AS person

abstract all()

Star selector: SELECT *

abstract cast(**args*, ***kwargs*)

Casts a selector to a type: SELECT CAST(a AS double)

abstract count()

Selects the count of all returned rows: SELECT count(*)

property filtering

Filter data

class nosqlapi.columndb.client.**ColumnSession**(*connection*, *database=None*)

Bases: *nosqlapi.common.core.Session*, abc.ABC

Column NOSQL database Session class

abstract alter_table(**args*, ***kwargs*)

Alter table or rename

abstract compact(**args*, ***kwargs*)

Compact table or database

abstract create_table(**args*, ***kwargs*)

Create table on database

abstract delete_table(**args*, ***kwargs*)

Create table on database

abstract truncate(**args*, ***kwargs*)

Delete all data into a table

client example

This is an example of a library for connecting to a cassandra server.

```
# mycolumndb.py
import nosqlapi

# this module is my library of NOSQL column database, like Cassandra database

class Connection(nosqlapi.columndb.ColumnConnection):
    def __init__(host='localhost', port=7000, database=0, username=None, password=None,␣
↪ssl=None, tls=None,
                 cert=None, ca_cert=None, ca_bundle=None): ...
    def close(self): ...
    def connect(self, *args, **kwargs): ...
    def create_database(self, database): ...
    def has_database(self, database): ...
    def databases(self): ...
    def delete_database(self): ...
    def show_database(self, database): ...


class Session(nosqlapi.columndb.ColumnSession):
    # define here all methods
    pass

conn = Connection('mycassandra.local', 'testdb', username='admin', password='pa$$w0rd',␣
↪tls=True)
print(conn.show_database('testdb')      # ('testdb', 17983788, 123, False) -- name, size,␣
↪rows, readonly
sess = conn.connect()                   # Session object
...
```

odm module

The **odm** module contains the specific object for *column* databases.

ODM module for column NOSQL database.

class nosqlapi.columndb.odm.**Column**(*name*, *data=None*, *of_type=None*, *max_len=None*, *auto_increment=False*, *primary_key=False*, *default=None*)

Bases: object

Represents column as container of values

__init__(*name*, *data=None*, *of_type=None*, *max_len=None*, *auto_increment=False*, *primary_key=False*, *default=None*)

Column object

Parameters **name** – Name of column

:param data. Data list or tuple :param of_type: Type of column :param max_len: Max length of column :param auto_increment: Boolean value (default False) :param primary_key: Set this column like a primary key :param default: Default function for generate data

__repr__()

Return repr(self).

__str__()

Return str(self).

__weakref__

list of weak references to the object (if defined)

append(*data=None*)

Appending data to column. If auto_increment is True, the value is incremented automatically.

Parameters **data** – Any type of data

Returns None

property auto_increment

Auto-increment value

property data

List of values

property of_type

Type of column

pop(*index=- 1*)

Deleting value

Parameters **index** – Number of index

Returns None

class nosqlapi.columndb.odm.**Index**(*name*, *table*, *column*)

Bases: tuple

__getnewargs__()

Return self as a plain tuple. Used by copy and pickle.

static __new__(*_cls*, *name*, *table*, *column*)

Create new instance of Index(name, table, column)

__repr__()

Return a nicely formatted representation string

property column

Alias for field number 2

property name

Alias for field number 0

property table

Alias for field number 1

class nosqlapi.columndb.odm.**Keyspace**(*name*, *exists=False*)

Bases: *nosqlapi.kvdb.odm.Keyspace*

Represents keyspace like database

class nosqlapi.columndb.odm.**Table**(*name*, **columns*, ***options*)

Bases: object

Represents table as container of columns

__init__(*name*, **columns*, ***options*)

Table object

Parameters

- **name** – Name of table
- **columns** – Columns
- **options** – Options

__repr__()

Return repr(self).

__str__()

Return str(self).

__weakref__

list of weak references to the object (if defined)

add_column(**columns*)

Adding one or more column object to table

Parameters **columns** – Column objects

Returns None

add_index(*index*)

Adding index to index property

Parameters **index** – Name or Index object

Returns None

add_row(**rows*)

Add one or more row into columns

Parameters **rows** – Tuple of objects

Returns None

property columns

List of columns

delete_column(*index=- 1*)

Deleting one column to table

Parameters **index** – Number of index

Returns None

delete_index(*index=- 1*)

Deleting index to index property

Parameters **index** – Name or Index object

Returns None

delete_row(*row=- 1*)

Delete one row into columns

Parameters **row** – Index of row

Returns None

get_rows()

Getting all rows

Returns List[tuple]

property index

List of indexes

property name

Name of table

property options

Options

set_option(*option*)

Update options

Parameters **option** – Dict options

Returns None

nosqlapi.columndb.odm.**column**(*func*)

Decorator function to transform list or tuple object to Column object

Parameters **func** – function to decorate

Returns Column object

odm example

These objects represent the respective *column* in databases.

```
import nosqlapi
import mycolumndb

keyspace = nosqlapi.columndb.odm.Keyspace('new_db')     # in short -> nosqlapi.columndb.
↪Keyspace('new_db')
# Make columns
id = nosqlapi.columndb.Column('id', of_type=int)
id.auto_increment = True                                # increment automatically
name = nosqlapi.columndb.Column('name', of_type=str)
# Set data
id.append()
name.append('Matteo Guadrini')
# Make table
table = nosqlapi.columndb.Table('peoples', id, name)
# Create Column object from column decorator
@nosqlapi.columndb.column
def salary(file, limit=5000):
    return [line
            for line in open(file, 'rt').readlines()
            if int(line) < 5000]

# Add column to table
table.add_column(id, name, salary('/tmp/salary.log'))
# Add table to keyspace
keyspace.append(table)

# Create database and insert data
mycolumndb.conn.create_database(keyspace)
mycolumndb.sess.create_table(table)
# Insert new data
mycolumndb.sess.insert('people', (None, 'Arthur Dent', 4000))
```

nosqlapi document

In this package we find abstract classes and ODM classes concerning the **document** database types.

client module

The **client** module contains the specific classes for *document* databases and they are based on the core classes.

Client module for document NOSQL database.

class nosqlapi.docdb.client.**DocBatch**(*batch*, *session=None*)

Bases: *nosqlapi.common.core.Batch*, abc.ABC

Document NOSQL database Batch class

class nosqlapi.docdb.client.**DocConnection**(**args*, ***kwargs*)

Bases: *nosqlapi.common.core.Connection*, abc.ABC

Document NOSQL database Connection class

__init__(**args*, ***kwargs*)

Instantiate Connection object

Parameters

- **host** – Name of host that contains database
- **user** – Username for connect to the host
- **password** – Password for connect to the host
- **database** – Name of database
- **port** – Tcp port
- **bind_address** – Hostname or an IP address for multiple network interfaces
- **read_timeout** – Timeout for reading from the connection in seconds
- **write_timeout** – Timeout for writing from the connection in seconds
- **ssl** – Ssl connection established
- **ssl_ca** – Ssl CA file specified
- **ssl_cert** – Ssl certificate file specified
- **tls** – Tls connection established
- **ssl_key** – Ssl private key file specified
- **ssl_verify_cert** – Verify certificate file
- **max_allowed_packet** – Max size of packet sent to server in bytes

abstract copy_database(**args*, ***kwargs*)

Copy database

:return : Union[Any, Response]

class nosqlapi.docdb.client.**DocResponse**(*data*, *code=None*, *header=None*, *error=None*)

Bases: *nosqlapi.common.core.Response*, abc.ABC

Document NOSQL database Response class

__weakref__

list of weak references to the object (if defined)

class nosqlapi.docdb.client.**DocSelector**(**args*, ***kwargs*)

Bases: *nosqlapi.common.core.Selector*, abc.ABC

Document NOSQL database Selector class

__init__(**args*, ***kwargs*)

Instantiate Selector object

Parameters

- **selector** – Selector part of the query string
- **fields** – Return fields

- **partition** – Partition or collection of data
- **condition** – Condition of query
- **order** – Order by specific selector
- **limit** – Limit result

class nosqlapi.docdb.client.**DocSession**(*connection*, *database=None*)

Bases: *nosqlapi.common.core.Session*, abc.ABC

Document NOSQL database Session class

abstract compact(**args*, ***kwargs*)

Compact data or database

client example

This is an example of a library for connecting to a mongodb server.

```
# mydocdb.py
import nosqlapi

# this module is my library of NOSQL document database, like MongoDB database

class Connection(nosqlapi.docdb.DocConnection):
    def __init__(host='localhost', port=27017, database=None, username=None,␣
↪password=None, ssl=None, tls=None,
                 cert=None, ca_cert=None, ca_bundle=None): ...
    def close(self): ...
    def connect(self, *args, **kwargs): ...
    def create_database(self, database): ...
    def has_database(self, database): ...
    def databases(self): ...
    def delete_database(self): ...
    def show_database(self, database): ...
    def copy_database(self, source, destination, force=False): ...


class Session(nosqlapi.docdb.DocSession):
    # define here all methods
    pass

conn = Connection('mymongo.local', username='admin', password='pa$$w0rd')
print(conn.databases())                 # {"databases": [{"name": "admin", "sizeOnDisk":␣
↪978944, "empty": False}], "totalSize": 1835008, "ok": 1}
sess = conn.connect()                   # Session object
...
```

odm module

The **odm** module contains the specific object for *document* databases.

ODM module for document NOSQL database.

class nosqlapi.docdb.odm.**Collection**(*name*, **docs*)

Bases: object

Represents collection of documents

__init__(*name*, **docs*)

Collection object

Parameters

- **name** – Name of collection
- **docs** – Documents like dict

__repr__()

Return repr(self).

__str__()

Return str(self).

__weakref__

list of weak references to the object (if defined)

append(*doc*)

Append document to collection

Parameters **doc** – Documents like dict

Returns None

property docs

Documents of collection

pop(*doc=- 1*)

Delete document from collection

Parameters **doc** – Number of document to remove

Returns None

class nosqlapi.docdb.odm.**Database**(*name*, *exists=False*)

Bases: *nosqlapi.kvdb.odm.Keyspace*

Represents database

class nosqlapi.docdb.odm.**Document**(*value=None*, *oid=None*, ***values*)

Bases: object

Represents document

__init__(*value=None*, *oid=None*, ***values*)

Document object

Parameters

- **value** – Body of document like dict
- **oid** – String id (default uuid1 string)

- **values** – Additional values of body

__repr__()
Return repr(self).

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

property body
Elements of document

property id
Document unique id

to_json(*indent=2*)
Transform document into json

Parameters **indent** – Number of indentation

Returns str

class nosqlapi.docdb.odm.**Index**(*name*, *data*)
Bases: object

Represents document index

__init__(*name*, *data*)
Index object

Parameters

- **name** – Name of index
- **data** – Data of index like dict

__repr__()
Return repr(self).

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

nosqlapi.docdb.odm.**document**(*func*)
Decorator function to transform dictionary object to Document object

Parameters **func** – function to decorate

Returns Document object

odm example

These objects represent the respective *document* in databases.

```
import nosqlapi
import mydocdb

# Create database
db = nosqlapi.docdb.odm.Database('test')
# Create documents
doc1 = nosqlapi.docdb.Document(oid=1)
doc2 = nosqlapi.docdb.Document(oid=2)
# Add documents to database
db.append(doc1)
db.append(doc2)
# Create Document object from document decorator
@nosqlapi.docdb.document
def user(name, age, salary):
    return {'name': name, 'age': age, 'salary': salary}

# Create database with docs
mydocdb.conn.create_database(db)
# Add more doc
mydocdb.sess.insert(user('Matteo Guadrini', 36, 25000))
```

nosqlapi graph

In this package we find abstract classes and ODM classes concerning the **graph** database types.

client module

The **client** module contains the specific classes for *graph* databases and they are based on the core classes.

Client module for graph NOSQL database.

class nosqlapi.graphdb.client.**GraphBatch**(*batch*, *session=None*)

Bases: *nosqlapi.common.core.Batch*, abc.ABC

Batch NOSQL database Session class

class nosqlapi.graphdb.client.**GraphConnection**(**args*, ***kwargs*)

Bases: *nosqlapi.common.core.Connection*, abc.ABC

Graph NOSQL database Connection class

__init__(**args*, ***kwargs*)

Instantiate Connection object

Parameters

- **host** – Name of host that contains database
- **user** – Username for connect to the host
- **password** – Password for connect to the host

- **database** – Name of database
- **port** – Tcp port
- **bind_address** – Hostname or an IP address for multiple network interfaces
- **read_timeout** – Timeout for reading from the connection in seconds
- **write_timeout** – Timeout for writing from the connection in seconds
- **ssl** – Ssl connection established
- **ssl_ca** – Ssl CA file specified
- **ssl_cert** – Ssl certificate file specified
- **tls** – Tls connection established
- **ssl_key** – Ssl private key file specified
- **ssl_verify_cert** – Verify certificate file
- **max_allowed_packet** – Max size of packet sent to server in bytes

class nosqlapi.graphdb.client.**GraphResponse**(*data*, *code=None*, *header=None*, *error=None*)

Bases: *nosqlapi.common.core.Response*, abc.ABC

Response NOSQL database Session class

__weakref__

list of weak references to the object (if defined)

class nosqlapi.graphdb.client.**GraphSelector**(**args*, ***kwargs*)

Bases: *nosqlapi.common.core.Selector*, abc.ABC

Graph NOSQL database Selector class

__init__(**args*, ***kwargs*)

Instantiate Selector object

Parameters

- **selector** – Selector part of the query string
- **fields** – Return fields
- **partition** – Partition or collection of data
- **condition** – Condition of query
- **order** – Order by specific selector
- **limit** – Limit result

class nosqlapi.graphdb.client.**GraphSession**(*connection*, *database=None*)

Bases: *nosqlapi.common.core.Session*, abc.ABC

Graph NOSQL database Session class

abstract detach(**args*, ***kwargs*)

Detach node

Returns Response

abstract link(**args*, ***kwargs*)

Link node to another

Returns Response

client example

This is an example of a library for connecting to a neo4j server.

```
# mygraphdb.py
import nosqlapi

# this module is my library of NOSQL graph database, like Neo4J database

class Connection(nosqlapi.graphdb.GraphConnection):
    def __init__(host='localhost', port=7474, database=None, username=None,␣
↪password=None, ssl=None, tls=None,
                 cert=None, ca_cert=None, ca_bundle=None): ...
    def close(self): ...
    def connect(self, *args, **kwargs): ...
    def create_database(self, database): ...
    def has_database(self, database): ...
    def databases(self): ...
    def delete_database(self): ...
    def show_database(self, database): ...


class Session(nosqlapi.graphdb.GraphSession):
    # define here all methods
    pass

conn = Connection('myneo.local', 'mydb', username='admin', password='pa$$w0rd')
print(conn.show_database('mydb'))       # {"name": "mydb", "address": "myneo.local:7474",
↪ "role": "standalone", "requestedStatus": "online", "currentStatus": "online", "error
↪": "", "default": False}
sess = conn.connect()                   # Session object
...
```

odm module

The **odm** module contains the specific object for *graph* databases.

ODM module for graph NOSQL database.

class nosqlapi.graphdb.odm.**Database**(*name*, *address=None*, *role=None*, *status='online'*, *default=False*)

Bases: *nosqlapi.kvdb.odm.Keyspace*

Represents database

__init__(*name*, *address=None*, *role=None*, *status='online'*, *default=False*)

Database object

Parameters

- **name** – Name of database
- **address** – Address of database server
- **role** – Role of database
- **status** – Status of database (default online)

- **default** – Default value or function

property online

Status of database

class nosqlapi.graphdb.odm.**Index**(*name*, *node*, *properties*, *options*)

Bases: tuple

__getnewargs__()

Return self as a plain tuple. Used by copy and pickle.

static __new__(*_cls*, *name*, *node*, *properties*, *options=None*)

Create new instance of Index(name, node, properties, options)

__repr__()

Return a nicely formatted representation string

property name

Alias for field number 0

property node

Alias for field number 1

property options

Alias for field number 3

property properties

Alias for field number 2

class nosqlapi.graphdb.odm.**Label**

Bases: *nosqlapi.common.odm.Text*

Label of node

class nosqlapi.graphdb.odm.**Node**(*labels*, *properties=None*, *var=''*)

Bases: object

Represents node

__init__(*labels*, *properties=None*, *var=''*)

Node object

Parameters

- **labels** – Label of node
- **properties** – Properties of Node
- **var** – Name variable

__repr__()

Return repr(self).

__str__()

Return str(self).

__weakref__

list of weak references to the object (if defined)

add_label(*label*)

Add label to node

Parameters **label** – Label string or object

Returns None

delete_label(*index=- 1*)

Delete label

Parameters **index** – Number of index

Returns None

class nosqlapi.graphdb.odm.**Property**

Bases: *dict*

Property of node

__repr__()

Return repr(self).

__weakref__

list of weak references to the object (if defined)

class nosqlapi.graphdb.odm.**Relationship**(*labels*, *properties=None*, *var=''*)

Bases: *nosqlapi.graphdb.odm.Node*

Represents relationship among nodes

__repr__()

Return repr(self).

__str__()

Return str(self).

class nosqlapi.graphdb.odm.**RelationshipType**

Bases: *nosqlapi.graphdb.odm.Label*

Type of relationship like a label

nosqlapi.graphdb.odm.**node**(*func*)

Decorator function to transform dictionary object to Node object

Parameters **func** – function to decorate

Returns Node object

nosqlapi.graphdb.odm.**prop**(*func*)

Decorator function to transform dictionary object to Property object

Parameters **func** – function to decorate

Returns Property object

odm example

These objects represent the respective *graph* in databases.

```
import nosqlapi
import mygraphdb

# Create database
db = nosqlapi.graphdb.odm.Database('test')
# Create nodes
node1 = nosqlapi.graphdb.Node(var='n1', labels=[nosqlapi.graphdb.Label('Person')],
                              properties=nosqlapi.graphdb.Property({'name': 'Matteo',
↪'age': 35}))
node2 = nosqlapi.graphdb.Node(var='n2', labels=[nosqlapi.graphdb.Label('Person')],
                              properties=nosqlapi.graphdb.Property({'name': 'Julio', 'age
↪': 53}))
# Add nodes to database
db.append(node1)
db.append(node2)

# Create Node object from node decorator
@nosqlapi.graphdb.node
def Person(name, age):
    return {'name': name, 'age': age}

# Create Property object from prop decorator
@nosqlapi.graphdb.prop
def user(name, age):
    return {'name': name, 'age': age}

# Create database with nodes
mydocdb.conn.create_database(db)
# Add other nodes
mydocdb.sess.insert(label='Person', properties=user('Matteo Guadrini', 36))
mydocdb.sess.insert(Person('Julio Artes', 29))          # Labels = ['Person']
```

1.4.4 Build a library

In this section we will build a **NOSQL API** compliant library, using the `nosqlapi` library. We will first build the core objects that will allow us to connect and operate with our database.

Note: The following example is designed for a NOSQL database of the *Document* type; by changing the application logic of the various methods in the classes you can build a library for another type of NOSQL database in the same way. The procedures are the same.

Warning: The purpose of this document is to explain how to use API class interfaces in the real world. Do not use the following example as a production library because it is very simplified and does not reflect all possible operations on a CouchDB server.

Let's prepare the steps:

- Create a package (directory with an `__init__.py` file) named **pycouch**.
- write `core.py`, with core classes and functions.
- write `utils.py`, with utility classes and function, like ODM objects.

Core

The **core** classes must provide a connection and operation interface towards the database. All *CRUD* operations will need to be supported. We can implement new ones, based on the type of database we support.

Create a `core.py` module.

```
#!/usr/bin/env python3
# -*- encoding: utf-8 -*-
# core.py

"""Python library for document CouchDB server"""

import nosqlapi
import urllib.request
import json
```

Connection class

Let's build the server connection class. Since we are connecting to a CouchDB database, we will take advantage of the *http API*.

We define the `__init__` constructor with all the info needed to perform operations with the database and create a `Session` object using the `connect()` method.

```
class Connection(nosqlapi.DocConnection):
    """CouchDB connection class; connect to CouchDB server."""
    def __init__(self, host='localhost', port=5984, username=None, password=None,␣
↪ssl=None, tls=None, cert=None,
                 database=None, ca_cert=None, ca_bundle=None):
        super().__init__(self, host=host, port=port, username=username,␣
↪password=password, ssl=ssl, tls=tls,
                         cert=cert, ca_cert=ca_cert, ca_bundle=ca_bundle)
        self.method = 'https://' if self.port == 6984 or self.port == 443 else 'http://'
        if self.username and self.password:
            auth = f'{self.username}:{self.password}@'
        self.url = f'{self.method}{auth}{self.host}:{self.port}'
```

Now let's define the `close` and `connect` methods, to create the database connection.

```
def close(self, clean=False):
    self._connected = False
    if clean:
        self.database = None
        self.host = None
        self.url = None

def connect(self):
```

(continues on next page)

(continued from previous page)

```
    session_url = self.url + f'/{self.database}'
    if urllib.request.head(session_url).status_code == 200:
        session = Session(connection=self, database=session_url)
        self._connected = True
        return session
    else:
        raise nosqlapi.ConnectError(f'I cannot connect to the server: {self.url}')
```

Now let's all define methods that operate at the database level.

```
def create_database(self, name, shards=8, replicas=3, partitioned=False):
    data = {"w": shards, "n": replicas}
    if partitioned:
        data['partitioned'] = partitioned
    req = urllib.request.Request(self.url + f'/{name}',
                                 data=json.dumps(data).encode('utf8'),
                                 method='PUT')
    req.add_header('Content-Type', 'application/json')
    response = urllib.request.urlopen(req)
    if response.status_code == 201:
        return Response(data=json.loads(response.read()),
                        code=response.status_code,
                        error=None,
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.DatabaseCreationError(f'Database creation␣
↪unsuccessfully: {name}'),
                        header=response.header_items())

def has_database(self, name):
    response = urllib.request.urlopen(self.url + f'/{name}')
    if response.status_code == 200:
        return Response(data=json.loads(response.read()),
                        code=response.status_code,
                        error=None,
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.DatabaseError(f'Database not found: {name}'),
                        header=response.header_items())

def delete_database(self, name):
    req = urllib.request.Request(self.url + f'/{name}', method='DELETE')
    req.add_header('Content-Type', 'application/json')
    response = urllib.request.urlopen(req)
    if response.status_code == 200:
        return Response(data=bool(json.loads(response.read())),
                        code=response.status_code,
                        error=None,
```

(continues on next page)

(continued from previous page)

```
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.DatabaseDeletionError(f'Database deletion␣
↪unsuccessfully: {name}'),
                        header=response.header_items())

def databases(self):
    response = urllib.request.urlopen(self.url + '/_all_dbs')
    if response.status_code == 200:
        return Response(data=json.loads(response.read()),
                        code=response.status_code,
                        error=None,
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.DatabaseError(f'Database not found: {name}'),
                        header=response.header_items())

def show_database(self, name):
    response = urllib.request.urlopen(self.url + f'/{name}')
    if response.status_code == 200:
        return Response(data=json.loads(response.read()),
                        code=response.status_code,
                        error=None,
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.DatabaseError(f'Database not found: {name}'),
                        header=response.header_items())

def copy_database(self, source, target, host, user=None, password=None, create_
↪target=True, continuous=True):
    data = {
        "_id": f"{source}to{target}",
        "source": source,
        "target": {
            "url": target,
            "auth": {
                "basic": {
                    "username": f"{user}",
                    "password": f"{password}"
                }
            }
        },
        "create_target":  create_target,
        "continuous": continuous
    }
    req = urllib.request.Request(self.url + '/_replicator', data=json.dumps(data).encode(
↪'utf8'), method='POST')
```

(continues on next page)

(continued from previous page)

```
    req.add_header('Content-Type', 'application/json')
    response = urllib.request.urlopen(req)
    if response.status_code == 200:
        return Response(data=json.loads(response.read()),
                        code=response.status_code,
                        error=None,
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.DatabaseError(f'Database copy error: {name}'),
                        header=response.header_items())
```

Response class

There isn't much to do with the `Response` class. We inherit directly from the `nosqlapi.docdb.DocResponse` class.

```
class Response(nosqlapi.DocResponse):
    """CouchDB response class; information about a certain transaction."""
    ...
```

Session class

Ok, now build the `Session` class. This class used for CRUD operation on the specific database. The `acl` property is used to retrieve the Access Control lists of the current database and therefore the read/write permissions of the current session. The property `indexes` is used to retrieve all the indexes created for the current database.

```
class Session(nosqlapi.DocSession):
    """CouchDB session class; CRUD operation on the database."""
    @property
    def acl(self):
        response = urllib.request.urlopen(self.database + f'/_security')
        if response.status_code == 200:
            return Response(data=json.loads(response.read()),
                            code=response.status_code,
                            error=None,
                            header=response.header_items())
        else:
            return Response(data=None,
                            code=response.status_code,
                            error=noslapi.SessionACLError(f'ACLs error'),
                            header=response.header_items())

    @property
    def indexes(self):
        response = urllib.request.urlopen(self.database + f'/_index')
        if response.status_code == 200:
            return Response(data=json.loads(response.read()),
                            code=response.status_code,
```

(continues on next page)

(continued from previous page)

```
                            error=None,
                            header=response.header_items())
        else:
            return Response(data=None,
                            code=response.status_code,
                            error=noslapi.SessionError(f'Index error'),
                            header=response.header_items())
```

Now let's define the `item_count` and `description` properties. `item_count` will be used to indicate a counter of each CRUD operation that will be impacted. `description` instead will contain the database info.

```
@property
def item_count(self):
    return self._item_count

@property
def description(self):
    response = urllib.request.urlopen(self.database + f'/_index')
    if response.status_code == 200:
        return self._description

    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.SessionError(f'Get description failed'),
                        header=response.header_items())
```

Now let's all define CRUD (Create, Read, Update, Delete) methods. *Create* word is associated to `insert` and `insert_many` methods; *Read* to `get` method, *Update* to `update` and `update_many` methods and *Delete* to `delete` method. Each *CRUD* method is created to directly manage the data in the database to which the connection was created via the `Connection` object.

```
def get(self, document='_all_docs', rev=None, attachment=None, partition=None,␣
↪local=False, key=None):
    url = self.database
    if partition:
        url += url + f'/{partition}/{document}'
    else:
        url += url + f'/{document}'
    if attachment:
        url += url + f'/{attachment}'
    if rev:
        url += url + f'?rev={rev}'
    elif key:
        url += url + f'?key={key}'
    if bool(local):
        url = self.database + f'/_local_docs'
    response = urllib.request.urlopen(url)
    if response.status_code == 200:
        return Response(data=json.loads(response.read()),
                        code=response.status_code,
                        error=None,
```

(continues on next page)

(continued from previous page)

```
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.SessionFindingError(f'Document not found: {url}'),
                        header=response.header_items())

def insert(self, name, data=None, attachment=None, partition=None):
    url = self.database + f'/{name}'
    if attachment:
        url += f"/{attachment}"
    id = f"{partition}:{name}" if partition else name
    data = {"_id": id}
    if data:
        data.update(data)
    req = urllib.request.Request(url,
                                 data=json.dumps(data).encode('utf8'),
                                 method='PUT')
    req.add_header('Content-Type', 'application/json')
    response = urllib.request.urlopen(req)
    if response.status_code == 201:
        return Response(data=json.loads(response.read()),
                        code=response.status_code,
                        error=None,
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.SessionInsertingError(f'Insert document {url}␣
↪with data {data} failed'),
                        header=response.header_items())

def insert_many(self, *documents):
    url = f"{self.database}/_bulk_docs"
    data = {"docs": []}
    if documents:
        data['docs'].extend(documents)
    req = urllib.request.Request(url,
                                 data=json.dumps(data).encode('utf8'),
                                 method='POST')
    req.add_header('Content-Type', 'application/json')
    response = urllib.request.urlopen(req)
    if response.status_code == 201:
        return Response(data=json.loads(response.read()),
                        code=response.status_code,
                        error=None,
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.SessionInsertingError('Bulk insert document failed
↪'),
```

(continues on next page)

(continued from previous page)

```
                        header=response.header_items())

def update(self, name, rev, data=None, partition=None):
    url = self.database + f'/{name}?rev={rev}'
    id = f"{partition}:{name}" if partition else name
    data = {"_id": id}
    if data:
        data.update(data)
    req = urllib.request.Request(url,
                            data=json.dumps(data).encode('utf8'),
                            method='PUT')
    req.add_header('Content-Type', 'application/json')
    response = urllib.request.urlopen(req)
    if response.status_code == 201:
        return Response(data=json.loads(response.read()),
                        code=response.status_code,
                        error=None,
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.SessionUpdatingError(f'Update document {url} with␣
↪data {data} failed'),
                        header=response.header_items())

def update_many(self, *documents):
    url = f"{self.database}/_bulk_docs"
    data = {"docs": []}
    if documents:
        data['docs'].extend(documents)
    req = urllib.request.Request(url,
                            data=json.dumps(data).encode('utf8'),
                            method='POST')
    req.add_header('Content-Type', 'application/json')
    response = urllib.request.urlopen(req)
    if response.status_code == 201:
        return Response(data=json.loads(response.read()),
                        code=response.status_code,
                        error=None,
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.SessionUpdatingError('Bulk update document failed
↪'),
                        header=response.header_items())

def delete(self, name, rev, partition=None):
    url = self.database + f'/{name}?rev={rev}'
    id = f"{partition}:{name}" if partition else name
    req = urllib.request.Request(url,
                            data=json.dumps(data).encode('utf8'),
```

(continues on next page)

(continued from previous page)

```
                              method='DELETE')
    req.add_header('Content-Type', 'application/json')
    response = urllib.request.urlopen(req)
    if response.status_code == 201:
        return Response(data=json.loads(response.read()),
                        code=response.status_code,
                        error=None,
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.SessionDeletingError(f'Delete document {name}␣
↪failed'),
                        header=response.header_items())
```

The `close` method will only close the session, but not the connection.

```
def close(self):
    self.database = None
    self._connection = None
```

The `find` method is the one that allows searching for data in the database. This method can accept strings or `Selector` objects, which help in the construction of the query in the database language.

```
def find(self, query):
    url = f"{self.database}/_find"
    if isinstance(query, Selector):
        query = query.build()
    req = urllib.request.Request(url,
                                 data=query.encode('utf8'),
                                 method='POST')
    req.add_header('Content-Type', 'application/json')
    response = urllib.request.urlopen(req)
    if response.status_code == 200:
        return Response(data=json.loads(response.read()),
                        code=response.status_code,
                        error=None,
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.SessionFindingError(f'Find documents failed: {json.
↪loads(response.read())}'),
                        header=response.header_items())
```

The `grant` and `revoke` methods are specific for enabling and revoking permissions on the current database.

```
def grant(self, admins, members):
    url = f"{self.database}/_security"
    data = {"admins": admins, "members": members}
    req = urllib.request.Request(url,
                                 data=json.dumps(data).encode('utf8'),
                                 method='PUT')
```

(continues on next page)

(continued from previous page)

```
    req.add_header('Content-Type', 'application/json')
    response = urllib.request.urlopen(req)
    if response.status_code == 200:
        return Response(data=json.loads(response.read()),
                        code=response.status_code,
                        error=None,
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.SessionACLError(f'Grant failed: {json.
↪loads(response.read())}'),
                        header=response.header_items())

def revoke(self):
    url = f"{self.database}/_security"
    data = {"admins": {"names": [], "roles": []}, "members": {"names": [], "roles": []}}
    req = urllib.request.Request(url,
                                 data=json.dumps(data).encode('utf8'),
                                 method='PUT')
    req.add_header('Content-Type', 'application/json')
    response = urllib.request.urlopen(req)
    if response.status_code == 200:
        return Response(data=json.loads(response.read()),
                        code=response.status_code,
                        error=None,
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.SessionACLError(f'Revoke failed: {json.
↪loads(response.read())}'),
                        header=response.header_items())
```

Now let's write the three methods for creating, modifying (also password reset) and deleting a user respectively: `new_user`, `set_user` and `delete_user`.

```
def new_user(self, name, password, roles=None, type='user'):
    if roles is None:
        roles = []
    server = self.database.split('/')
    url = f"{server[0]}//{server[2]}/_users/org.couchdb.user:{name}"
    data = {"name": name, "password": password, "roles": roles, "type": type}
    req = urllib.request.Request(url,
                                 data=json.dumps(data).encode('utf8'),
                                 method='PUT')
    req.add_header('Content-Type', 'application/json')
    response = urllib.request.urlopen(req)
    if response.status_code == 201:
        return Response(data=json.loads(response.read()),
                        code=response.status_code,
                        error=None,
```

(continues on next page)

(continued from previous page)

```
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.SessionACLError(f'New user failed: {json.
↪loads(response.read())}'),
                        header=response.header_items())

def set_user(self, name, password, rev, roles=None, type='user'):
    if roles is None:
        roles = []
    server = self.database.split('/')
    url = f"{server[0]}//{server[2]}/_users/org.couchdb.user:{name}"
    data = {"name": name, "password": password, "roles": roles, "type": type}
    req = urllib.request.Request(url,
                                 data=json.dumps(data).encode('utf8'),
                                 method='PUT')
    req.add_header('Content-Type', 'application/json')
    req.add_header(f"If-Match: {rev}")
    response = urllib.request.urlopen(req)
    if response.status_code == 201:
        return Response(data=json.loads(response.read()),
                        code=response.status_code,
                        error=None,
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.SessionACLError(f'Modify user or password failed:
↪{json.loads(response.read())}'),
                        header=response.header_items())

def delete_user(self, name, rev, admin=False):
    server = self.database.split('/')
    if admin:
        url = f"{server[0]}//{server[2]}/_users/org.couchdb.user:{name}"
    else:
        url = f"{server[0]}//{server[2]}/_config/admins/{name}"
    req = urllib.request.Request(url, method='DELETE')
    req.add_header('Content-Type', 'application/json')
    req.add_header(f"If-Match: {rev}")
    response = urllib.request.urlopen(req)
    if response.status_code == 200:
        return Response(data=json.loads(response.read()),
                        code=response.status_code,
                        error=None,
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.SessionACLError(f'Delete user failed: {json.
↪loads(response.read())}'),
```

(continues on next page)

(continued from previous page)

```
                        header=response.header_items())
```

We will now write the `add_index` and `delete_index` methods, which are mainly concerned with creating indexes for the database.

```
def add_index(self, name, fields):
    url = f"{self.database}/_index"
    data = {"name": name, 'type': 'json', "index": {'fields': fields}}
    req = urllib.request.Request(url,
                                 data=json.dumps(data).encode('utf8'),
                                 method='POST')
    req.add_header('Content-Type', 'application/json')
    response = urllib.request.urlopen(req)
    if response.status_code == 201:
        return Response(data=json.loads(response.read()),
                        code=response.status_code,
                        error=None,
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.SessionError(f'Index creation error: {json.
↪loads(response.read())}'),
                        header=response.header_items())

def delete_index(self, ddoc, name):
    url = f"{self.database}/_index/{ddoc}/json/{name}"
    req = urllib.request.Request(url, method='DELETE')
    req.add_header('Content-Type', 'application/json')
    response = urllib.request.urlopen(req)
    if response.status_code == 200:
        return Response(data=json.loads(response.read()),
                        code=response.status_code,
                        error=None,
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.SessionError(f'Index deletion error: {json.
↪loads(response.read())}'),
                        header=response.header_items())
```

Finally, let's add the database `compact` method, which is useful after inserting a lot of data into the database to reduce the physical disk space.

```
def compact(self):
    url = f"{self.database}/_compact"
    req = urllib.request.Request(url, method='POST')
    req.add_header('Content-Type', 'application/json')
    response = urllib.request.urlopen(req)
    if response.status_code == 200:
        return Response(data=json.loads(response.read()),
```

(continues on next page)

(continued from previous page)

```
                        code=response.status_code,
                        error=None,
                        header=response.header_items())
    else:
        return Response(data=None,
                        code=response.status_code,
                        error=noslapi.SessionError(f'Compaction error: {json.
↪loads(response.read())}'),
                        header=response.header_items())
```

Batch class

Since we have already defined *"bulk"* operations in the `insert_many` and in the `update_many` in the `Session` class, we can define the get bulk through a `Batch` class.

```
class Batch(nosqlapi.DocBatch):
    """CouchDB batch class; multiple get from session."""

    def execute(self):
        data = {"docs": self.batch}
        url = f"{self.session.database}/_bulk_get"
        req = urllib.request.Request(url, method='POST')
        req.add_header('Content-Type', 'application/json')
        response = urllib.request.urlopen(req)
        if response.status_code == 200:
            return Response(data=json.loads(response.read()),
                            code=response.status_code,
                            error=None,
                            header=response.header_items())
        else:
            return Response(data=None,
                            code=response.status_code,
                            error=noslapi.SessionError(f'Get multiple document error:
↪{json.loads(response.read())}'),
                            header=response.header_items())
```

Selector class

Now instead, let's define the last class that will represent the query shape for our CouchDB server, the `Selector` class.

```
class Selector(nosqlapi.DocSelector):
    """CouchDB selector class; query representation."""
    pass
```

Utils

The **utils** classes and functions they map objects that represent data on the CouchDB server. These types of objects are called *ODMs*.

Create a `utils.py` module.

```
#!/usr/bin/env python3
# -*- encoding: utf-8 -*-
# utils.py

"""Python utility library for document CouchDB server"""

from nosqlapi.docdb import Database, Document, Index
import core
import json
```

connect function

Let's create a simple function that will create a `Connection` object for us and return a `Session` object. We will call it `connect()`.

```
def connect(host='localhost', port=5984, username=None, password=None, ssl=None,␣
↪tls=None, cert=None,
                database=None, ca_cert=None, ca_bundle=None):
    conn = core.Connection(host='localhost', port=5984, username=None, password=None,␣
↪ssl=None, tls=None, cert=None,
                database=None, ca_cert=None, ca_bundle=None)
    return conn.connect()
```

ODM classes

Now let's define a `DesignDocument` class, which will represent a design document in the CouchDB server.

```
class DesignDocument(Document):
    """Design document"""

    def __init__(self, oid=None, views=None, updates=None, filters=None, validate_doc_
↪update=None):
        super().__init__(oid)
        self._id = self['_id'] = f'_design/{self.id}'
        self["language"] = "javascript"
```

(continues on next page)

(continued from previous page)

```
        self['views'] = {}
        self['updates'] = {}
        self['filters'] = {}
        if views:
            self['views'].update(views)
        if updates:
            self['updates'].update(updates)
        if filters:
            self['filters'].update(filters)
        if validate_doc_update:
            self['validate_doc_update'] = validate_doc_update
```

Now let's define a `PermissionDocument` class, which will represent a permission document in the CouchDB server.

```
class PermissionDocument(Document):
    """Permission document"""

    def __init__(self, admins=None, members=None):
        super().__init__()
        self._id = None
        del self['_id']
        self["admins"] = {"names": [], "roles": []} if not admins else admins
        self['members'] = {"names": [], "roles": []} if not members else members
```

Note: Now that we have defined some classes that represent documents, we can adapt our methods of the Session class around these ODM types.

If you want to see more examples, clone the official repository of `nosqlapi` and find in the *tests* folder all the examples for each type of database.

```
$ git clone https://github.com/MatteoGuadrini/nosqlapi.git
$ cd nosqlapi
$ python -m unittest discover tests
$ ls -l tests
```

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

n

INDEX

Symbols

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V