ArangoDB Python Driver Documentation

Release 0.1a

Max Klymyshyn

Sep 27, 2017

Contents

CHAPTER 1

Features support

Driver for Python is not entirely completed. It supports **Connections to ArangoDB with custom options**, **Collections**, **Documents**, **Indexes Cursors** and partially **Edges**.

Presentation about Graph Databases and Python with real-world examples how to work with **arango-python**.

> **ArangoDB** is an open-source database with a flexible data model for documents, graphs, and key-values. Build high performance applications using a convenient sql-like query language or JavaScript/Ruby extensions.

More details about **ArangoDB** on official website. Some blog posts about this driver.

CHAPTER 2

Getting started

Installation

Library is in early alpha so it's not on PyPi yet. To install use *pip*:

```
pip install arango
```

Usage example

It's quite simple to start work with **ArangoDB**:

```
from arango import create

# create connection to database
conn = create(db="test")

# create database itself
conn.database.create()

# create collection with name `test_collection`
conn.test_collection.create()
# create document
conn.test_collection.documents.create({"sample_key": "sample_value"})
# get first document
doc = conn.test_collection.documents().first
# get document body
doc.body

# get all documents in collection
for doc in conn.test_collection.query.execute():
  print doc.id
```

```
# work with AQL
conn.test_range.create()

for n in range(10):
  conn.test_range.documents.create({
    "n": n,
    "mult": n * n})

conn.test_range.query.filter(
  filter("n == 1 || n == 5")).execute()

# delete database
conn.database.delete()
```

CHAPTER 3

Contents

Collections

Collections is something similar to tables in SQL databases world. Collection consist from **documents** and *Edges*.

It's quite easy to create collection:

```
from arango import create

# here we define connection to Arango
c = create(db="test")

# make sure database exists
c.database.create()

# here we creating collection explicitly
c.test.create()

assert len(c.collections()) == 1

# here we creating edges collection
c.test_edges.create_edges()

assert len(c.collections()) == 2
```

Collection `test` being created.

Note: It's not necessary to create collection before adding documents to it. You can specify `createCollection` as keyed argument during creation of new **Document**

If you don't want to create collection explicitly use

```
# here we creating document AND collection
c.test.documents.create({"sample": 1}, createCollection=True)
```

Get list of collections

To get list of **Collections** simply call connection like *c()*

For example:

```
# here we creating collection explicitly
c.test.create()

assert c(), ["test"]
```

class arango.collection.**Collections**(*connection*)
connection) for Collections

__call__(**args*, ***kwargs*)
Return list of collections within current database

__getattr__(*name*)
Accessible as property by default.

__getitem__(*name*)
In case property used internally by `Collections` it's possible to use dict-like interface, for example `.database` used internally as link to database instance but feel free to use dict-like interface to create collection with name `database`: `voca["database"]`

Collection

Arango DB provide rich API to manipulate collections Collection instance methods is quite rich. Here is documentation which describe *Collections REST Api*

class arango.collection.**Collection**(*connection=None*, *name=None*, *id=None*, *createCollection=True*, *response=None*)
Represent single collection with certain name

__len__()
Exactly the same as `count` but it's possible to use in more convenient way

```
c.test.create()

assert c.test.count() == len(c.test)
```

cid
Get collection name

count()
Get count of all documents in collection

create(*waitForSync=False*, *type=2*, ***kwargs*)
Create new **Collection**. You can specify `waitForSync` argument (boolean) to wait until collection will be synced to disk

create_edges(**args*, ***kwargs*)
Create new **Edges Collection** - sepcial kind of collections to keep information about edges.

delete()
Delete collection

documents
Get *Documents* related to Collection.

Technically return instance of *Documents for Collection instance* object

edges
Get *Edges* related to Collection.

Technically return instance of *Edges for Collection instance* object

If this method used to query edges (or called with no arguments) it may generated exceptions:

- `DocumentIncompatibleDataType`

  In case you're not provided `VERTEX` of the Edge which should be instance or subclass od *Document*

  More about DocumentIncompatibleDataType

index
Get **Indexes** related to Collection

info(*resource=''*)
Get information about collection. Information returns **AS IS** as raw `Response` data

load()
Load collection into memory

properties(***props*)
Set or get collection properties.

If `**props` are empty eq no keyed arguments specified then this method return properties for current **Collection**.

Otherwise method will set or update properties using values from `**props`

query
Create Query Builder for current collection.

```
c.test.create()
c.test.docs.create({"name": "sample"})

assert len(c.test.query.execute()), 1
```

rename(*name=None*)
Change name of Collection to `name`.

Return value is `bool` if success or error respectively.

This method may raise exceptions:

- `InvalidCollection`

  This one may be generated only in case very low-level instantiation of Collection and if base collection proxy isn't provided More about InvalidCollection

- `CollectionIdAlreadyExist`

  If Collection with new name already exist this exception will be generated. More about CollectionIdAlreadyExist

- `InvalidCollectionId`

  If Collection instantiated but name is not defined or not set. More about InvalidCollectionId

Sample usage:

```
c.test.create()

c.test.rename("test2")
assert "test2" in c()
```

truncate()
Truncate current **Collection**

unload()
Unload collection from memory

Documents

Documents in ArangoDB are JSON objects. These objects can be nested (to any depth) and may contains lists. Each document is unique identified by its document handle.

Usage example:

```
from arango import create

# connection & collection `test`
c = create(db="test")
c.database.create()  # make sure database exists
c.test.create()

# create FROM document
document = c.test.documents.create({
    "sample_key": "sample_value"
})

assert document.get("sample_key") == "sample_value"

assert c.test.documents().count != 0
```

Documents for Collection instance

Documents are accessible via collection instance for example `connection.collection.sample_collection.documents`. Usually this expressions looks lot shorter.

Basically via *docuemnts* shortcut accessible **Docuemnts Proxy** - Proxy object which have several shortcuts and produce `Resultset` object.

Below described basic method within `Documents` proxy:

class `arango.document.`**Documents**(*collection=None*)
Proxy object to handle work with documents within collection instance

count
Get count of all documents in *Collection*

create(**args*, ***kwargs*)
Shortcut for new documents creation

create_bulk(*docs*, *batch=100*)
Insert bulk of documents using **HTTP Interface for bulk imports**.

```
docs = [
    {"name": "doc1"},
    {"name": "doc2"},
    {"name": "doc3"}]
response = c.test.documents.create_bulk(docs)

assert response == {
    u'created': 3, u'errors': 0,
    u'empty': 0, u'error': False}, "Docs are not created"
```

Actually, it's possible to use **Headers and values import** in this call (and first element in `docs` have to be attribute names and every element in `docs` array have to be list). In this case you don't need to pass key/value pair for every document.

```
docs = [
    ["name"],
    ["doc1"],
    ["doc2"],
    ["doc3"]]
response = c.test.documents.create_bulk(docs)

assert response == {
    u'created': 3, u'errors': 0,
    u'empty': 0, u'error': False}, "Docs are not created"
```

delete(*ref_or_document*)

Delete document shorcut

`ref_or_document` may be either plai reference or Document instance

load(*doc_id*)

Load particular document by id `doc_id`.

Example:

```
doc_id = c.test.documents.create({"x": 2}).id
doc = c.test.documents.load(doc_id)

assert doc.body["x"] == 2
```

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

Update document

`ref_or_document` may be either plain reference or Document instance

Document

Document instance methods consist from basic **CRUD** methods and serveral shortcuts to more convenient work with documents.

class `arango.document.`**Document**(*collection=None*, *id=None*, *rev=None*, *resource_url=None*, *connection=None*)

Particular instance of Document

body

Return whole document.

This property setter also should be used if overwriting of whole document is required.

```
doc_id = c.test.documents.create({"x": 2}).id

doc = c.test.documents.load(doc_id)
assert doc.body["x"] == 2

doc.body = {"x": 1}
doc.save()

assert c.test.documents.load(doc_id).body["x"] == 1
```

create (*body*, *createCollection=False*, ***kwargs*)
Method to create new document.

Possible arguments: *waitForSync*

Read more about additional arguments *Documents REST Api*

This method may raise DocumentAlreadyCreated exception in case document already created.

Return document instance (`self`) or `None`

delete ()
Delete current document.

Return `True` if success and `False` if not

get (*name=None*, *default=None*)
This method very similar to `dict`'s `get` method. The difference is that *default* value should be specified explicitly.

To get specific value for specific key in body use and default *(fallback)* value `0`

```
c.test.documents().first.get(name="sample_key", default=0)
```

id
Id of the *Document* instance

rev
Revision of the *Document* instance

save (***kwargs*)
Method to force save of the document.

`kwargs` will be passed directly to `requests` arguments.

update (*newData*, *save=True*, ***kwargs*)
Method to update document.

This method is **NOT** for overwriting document body. In case document is `list` it will **extend** current document body. In case it's `dict` - **update** document body with new data.

To overwrite document body use `body` setter.

In case `save` argument set to `False` document will not be updated until `save()` method will be called.

This method may raise EdgeNotYetCreated exception in case you trying to update edge which is not saved yet.

Exception DocumentIncompatibleDataType will be raised in case body of the document isn't either `dict` or `list`.

AQL Queries

Query Builder is abstraction layer around **AQL** to work with it in more *pythonic* way.

Simplest start point is to use *arango.collection.Collection.query*.

Simple example:

```
from arango import collection as c

# create collection
c.test1.create()

c.test1.docs.create({"name": "John", "email": "john@example.com"})
c.test1.docs.create({"name": "Jane", "email": "jane@example.com"})

c.test1.query.filter("obj.name == 'John'").build_query()

c.test1.delete()
```

will generate AQL query:

```
FOR obj IN test
    FILTER obj.name == 'John'
RETURN
    obj
```

AQL Query Builder API

This API typically accesible via `query` method of collection instance.

Builder methods to generate AQL query:

class `arango.aql.`**`AQLQuery`**(*connection=None*, *collection=None*, *no_cache=False*)

An abstraction layer to generate simple AQL queries.

`bind`(***kwargs*)

Bind some data to AQL Query. Technically it's just a proxy to *arango.cursor.Cursor.bind* method which attach variables to the `Cursor`.

It's mostly for avoding any kind of query injetions.

```
data = c.test.query.filter("obj.name == @name")\
                   .bind(name="Jane")\
                   .execute().first

assert data != None
assert data.body["name"] == "Jane"
```

`build_query`()

Build AQL query and return it as a string. This is good start to debug generated AQL queries.

`collect`(**pairs*, ***kwargs*)

Specify `COLLECT` operators, it's possible to use it multiple times

```
COLLECT variable-name = expression
COLLECT variable-name = expression INTO groups
```

In python

```
c.test.query
      .collect("emails", "u.email")
      .collect("names", "u.name", into="eml")
      .result(emails="eml",
              names="names")
```

cursor(***kwargs*)
Method to provide custom arguments for *arango.cursor.Cursor* instance. All keywords arguments except `bindVars` may be changed.

execute(*wrapper=None*)
Execute query: create cursor, put binded variables and return instance of *arango.cursor.Cursor* object

filter(*condition*)
Filter query by condition `condition`. It's possible to add multiple filter expressions.

```
FILTER condition
```

For exmaple code in python

```
c.test.query
      .filter("a==b && c==d")
      .filter("d == m")
```

iter(*name*)
`FOR` cycle temporary variable, `variable-name` in AQL expression:

```
FOR variable-name IN expression
```

let(*name*, *value*)
Add `LET` operation

```
LET variable-name = expression
```

limit(*count*, *offset=None*)
Limit results with `count` items. By default `offset` is `0`.

```
query.limit(100, offset=10)
```

over(*expression*)
`expression` in `FOR` cycle

```
FOR variable-name IN expression
```

result(**args*, ***kwargs*)
Expression which will be added as `RETURN` of **AQL**. You can specify:

- single name, like `q.result("u")`
- named arguments, like `q.result(users="u", members="m")` which transform into `RETURN {users: u, members: m}`
- `fields` named argument, like `q.result(fields={"key-a": "a"})` to work with names which are not supported by Python syntax.

sort(**args*)
Sort result by criterias from `args`.

```
query.sort("u.email", "u.first_name DESC")
     .sort("u.last_name")
```

Helpers to work with query variables and functions.

`arango.aql.`**V**(*name*)
Factory for defining variables in requests. By default in functions arguments which are dicts all fields wrapped with double quoutes `"`. To specify members of variables defined above `V` factory should be used.

```
expect = 'MERGE({"user1": u.name}, {"user1": "name"})'
assert F.MERGE(
    {"user1": V("u.name")},
    {"user1": "name"}).build_query() == expect
```

class `arango.aql.`**FuncFactory**
AQL Function factory. This is `F` object in `arango.aql` module.

```
from arango.aql import F

c.test.query.over(F.PATH("a", "b", "c")).execute()
```

Execute query:

```
FOR obj IN PATH(a, b, c)
RETURN obj
```

Making raw queries with AQL

Now it's possible to querieng database by using *Arango Query Language (AQL)*.

This functionality implementation based on *HTTP Interface for AQL Query Cursors* and provide lazy iterator over dataset and with ability to customize (wrap) result item by custom wrapper.

Alternative way to do such king of functionality is by using *Documents REST Api* which is not implemented in driver.

class `arango.cursor.`**Cursor**(*connection*, *query*, *count=True*, *batchSize=None*, *bindVars=None*, *wrapper=<bound method type.load of <class 'arango.document.Document'>>*)
Work with **Cursors** in ArangoDB. At the moment, it's common routine to work with **AQL** from this driver.

Note: the server will also destroy abandoned cursors automatically after a certain server-controlled timeout to avoid resource leakage.

- `query` - contains the query string to be executed (mandatory)
- **`count` - boolean flag that indicates whether the** number of documents found should be returned as "count" attribute in the result set (optional). Calculating the "count" attribute might have a performance penalty for some queries so this option is turned off by default.
- **`batchSize` - maximum number of result documents to be** transferred from the server to the client in one roundtrip (optional). If this attribute is not set, a server-controlled default value will be used.

- `bindVars` - key/value list of bind parameters (optional).
- **`wrapper` - by default it's `Document.load`** class, wrap result into

bind(*bind_vars*)
Bind variables to the cursor

first
Get first element from resultset

last
Return last element from `current bulk`. It's **NOT** last result in *entire dataset*.

Custom data wrapper for raw queries

It's not necessary to wrap all documents within `Document` object. `Cursor` do it by default but you can provide custom wrapper by overriding `wrapper` argument during execution of `connection.query` method.

Note: Also it's possible to provide custom wrapper via *`arango.aql.AQLQuery.cursor`* method during building of the **AQL** query:

```
c.test1.query.cursor(wrapper=lambda conn, item: item)
             .filter("obj.name == 'John'").build_query()
```

`wrapper` should accept two arguments:

- `connection` - first argument, current connection instnace
- `item` - dictionary with data provided from ArangoDB query

```
from arango import c

wrapper = lambda conn, item: item

c.collection.test.create()
c.collection.test.documents.create({"1": 2})

# create connection to database
for item in c.query("FOR d in test RETURN d", wrapper=wrapper):
    item
```

Indexes

Indexes are used to allow fast access to documents. For each collection there is always the primary index which is a hash index for the document identifier.

Usage example:

```
from arango import create

# here we define connection to Arango
c = create()

# here we creating collection explicitly
```

```
c.test.create()

# create `hash` index for two fields: `name` and `num`,
# not unque
c.test.index.create(["name", "num"])

# create unique `hash` index for `slug` field
c.test.index.create(
    "slug",
    index_type="hash",
    unique=True
)
```

class `arango.index.`**Index**(*collection=None*)

Interface to work with Indexes

__call__()

Get list of all available indexes. Returns tuple with indentyfiers and original response

```
...
c.test.index()
```

create(*fields*, *index_type='hash'*, *unique=False*)

Create new index. By default type is *hash* and *unique=False*

`fields` may be either `str`, `list` or `tuple`.

This method may generate WrongIndexType exception in case `index_type` isn't allowed for Arango DB

delete(*field_id*)

Return tuple of two values: - bool success or not deletion - original response

get(*field_id*, *force_read=False*)

Get index by `id`

Edges

An edge is an entity that represents a connection between two documents. The main idea of edges is that you can build your own graph (tree) between sets of documents and then perform searches within the document hierarchy.

In order to define a vertex, `from_document` and `to_document` should be specified during the creation of the edge:

```
from arango import create

c = create()
c.test.create()

# create FROM document
from_doc = c.test.documents.create({
    "sample_key": "sample_value"
})

# create TO document
to_doc = c.test.documents.create({
    "sample_key1": "sample_value1"
```

```
})

# creating edge with custom data
c.test.edges.create(from_doc, to_doc, {"custom": 1})
```

Warning: Code below should be implemented by using `AQL` (*AQL Queries*). Not implemented at the moment.

```
# getting edge by document
# c.test.edges(from_doc)

# getting with direction
# c.test.edges(from_doc, direction="in")

# assert c.test.edges(from_doc).first.from_document == from_doc
# assert c.test.edges(from_doc).first.to_document == to_doc
```

Edges for Collection instance

Edges are accessible via a collection instance, for example `connection.collection.sample_collection.edges`. Usually this expressions looks lot shorter.

Basically via *edges* shortcut accessible **Edges Proxy** - Proxy object which have several shortcuts and produce `Resultset` object.

Below described basic method within `Edges` proxy:

class `arango.edge.`**Edges**(*collection=None*)

Proxy objects between `Collection` and `Edge`. Edges in general very related to `Document`.

create(**args*, ***kwargs*)

Create new Edge

delete(*ref*)

Delete Edge by reference

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

Update Edge by reference

Making queries

Warning: This functionality not implmented yet. Use **AQL** - *AQL Queries* section with custom wrapper to work with Edges.

More details in *Edges REST Api* documentation of **ArangoDB**

Edge

Edge instance methods consist from basic **CRUD** methods and additional methods specific obly for **Edges**:

class `arango.edge.`**Edge**(*collection=None*, *_id=None*, *_rev=None*, *_from=None*, *_to=None*, ***kwargs*)

Edge instance object

body
This property return Edge content

create(*from_doc*, *to_doc*, *body=None*, ***kwargs*)
Method to create new edge. `from_doc` and `to_doc` may be both **document-handle** or instances of `Document` object.

Possible arguments: *waitForSync*

Read more about additional arguments *Edges REST Api*

This method may raise EdgeAlreadyCreated exception in case edge already created.

Return edge instance (`self`) or `None`

delete()
Method to delete current edge. If edge deleted this method return `True` and in other case `False`

from_document
From vertex, return instance of `Document` or `None`

get(*name=None*, *default=None*)
This method very similar to `dict`'s `get` method. The difference is that *default* value should be specified explicitly.

To get specific value for specific key in body use and default *(fallback)* value `0`:

```
edge.get(name="sample_key", default=0)
```

id
Id of the *Edge* instance

rev
Revision of the *Edge* instance

save(***kwargs*)
Method to save Edge. This is useful when edge udpated several times via `update`

Possible arguments: *waitForSync*

Read more about additional arguments *Edges REST Api*

to_document
To vertex, return instance of `Document` or `None`

update(*body*, *from_doc=None*, *to_doc=None*, *save=True*, ***kwargs*)
Method to update edge. In case **from_doc** or **do_doc** not specified or equal to `None` then current `from_document` and `to_document` will be used.

In case `save` argument set to `False` edge will not be updated until `save()` method will be called.

This method may raise EdgeNotYetCreated exception in case you trying to update edge which is not saved yet.

Exception EdgeIncompatibleDataType will be raised in case body of the edge isn't `dict`.

Database

Database is abstraction over single one database within ArangoDB. With basic API you can **create**, **delete** or **get details** about particular database.

Note: Currently ArangoDB REST API support of getting list of databases. Driver doesn't support this functionality at the moment. However it's quite easy to implement using `conn.connection.client` and `conn.url(db_prefix=False)`.

```
from arango import create

c = create(db="test")
c.database.create()

c.database.info["name"] == "test"
c.database.delete()
```

class `arango.db.`**`Database`**(*connection*, *name*)

ArangoDB starting from version 1.4 work with multiple databases. This is abstraction to manage multiple databases and work within documents.

`create`(*ignore_exist=True*)

Create new database and return instance

`delete`(*ignore_exist=True*)

Delete database

`info`

Get info about database

Exceptions

Exceptions Overview

All `arango-python` exceptions are placed into `arango.exceptions` module. Feel free to imprort it like this:

```
from arango.exceptions import InvalidCollection
```

List of exceptions

Glossary

Arango Query Language (AQL) Querying documents and graphs in one database with AQL

Collections REST Api This is an introduction to ArangoDB's Http Interface for collections. HttpCollection

Documents REST Api Arango DB Http Interface for Documents. More details in RestDocument

Edges REST Api REST API for manipulate I trought HTTP interface of ArangoDB. Documentation about RestEdge

HTTP Interface for AQL Query Cursors *Description of HTTP Cursor REST API* on **ArangoDB** website: *HttpCursor* <http://www.arangodb.org/manuals/current/HttpCursor.html>`_

Indexes REST Api ArangoDB's Http Interface for Indexes. HttpIndex

waitForSync This argument may be `True` or `False`. If it's `True` then you'll get response from the server when *Document*, *Edge* or *Collection* will be saved on disk.

Guidelines

There's several simple rules which I want to follow during development of this project:

- All new features should be documented
- All new features should have *unit* and *integration* tests
- Code Coverage should be hight, at least 95%
- If something might be property it **have to be** property

CHAPTER 4

Arango versions, Platforms and Python versions

Supported versions of ArangoDB: **1.1x** and **1.2x**

This release support **Python 3.3**, *Python 2.7*, *PyPy 1.9*.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

L

O

P

Q

R

S

T

U

V

W