aioarangodb Documentation

Release 0.1.0

Joohwan Oh
Ramon Navarro

Jun 12, 2020

Contents

Welcome to the documentation for **aioarangodb**, a Python driver for ArangoDB with AsyncIO.

- Pythonic interface
- Lightweight
- High API coverage

CHAPTER 1

Compatibility

- Python versions 3.5, 3.6 and 3.7 are supported
- aioArangoDB supports ArangoDB 3.5+

CHAPTER 2

Installation

To install a stable version from PyPi:

```
~$ pip install aioarangodb
```

To install the latest version directly from GitHub:

```
~$ pip install -e git+git@github.com:bloodbare/aioarangodb.git@master#egg=aioarangodb
```

You may need to use `sudo` depending on your environment.

CHAPTER 3

Contents

3.1 Getting Started

Here is an example showing how **aioarangodb** client can be used:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient(hosts='http://localhost:8529')

# Connect to "_system" database as root user.
# This returns an API wrapper for "_system" database.
sys_db = await client.db('_system', username='root', password='passwd')

# Create a new database named "test" if it does not exist.
if not await sys_db.has_database('test'):
    await sys_db.create_database('test')

# Connect to "test" database as root user.
# This returns an API wrapper for "test" database.
db = await client.db('test', username='root', password='passwd')

# Create a new collection named "students" if it does not exist.
# This returns an API wrapper for "students" collection.
if db.has_collection('students'):
    students = db.collection('students')
else:
    students = await db.create_collection('students')

# Add a hash index to the collection.
await students.add_hash_index(fields=['name'], unique=False)

# Truncate the collection.
await students.truncate()
```

(continues on next page)

(continued from previous page)

```
# Insert new documents into the collection.
await students.insert({'name': 'jane', 'age': 19})
await students.insert({'name': 'josh', 'age': 18})
await students.insert({'name': 'jake', 'age': 21})

# Execute an AQL query. This returns a result cursor.
cursor = await db.aql.execute('FOR doc IN students RETURN doc')

# Iterate through the cursor to retrieve the documents.
student_names = [document['name'] async for document in cursor]
```

3.2 Databases

ArangoDB server can have an arbitrary number of **databases**. Each database has its own set of *collections* and *graphs*. There is a special database named `_system`, which cannot be dropped and provides operations for managing users, permissions and other databases. Most of the operations can only be executed by admin users. See *Users and Permissions* for more information.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "_system" database as root user.
# This returns an API wrapper for "_system" database.
sys_db = client.db('_system', username='root', password='passwd')

# List all databases.
await sys_db.databases()

# Create a new database named "test" if it does not exist.
# Only root user has access to it at time of its creation.
if not await sys_db.has_database('test'):
    await sys_db.create_database('test')

# Delete the database.
await sys_db.delete_database('test')

# Create a new database named "test" along with a new set of users.
# Only "jane", "john", "jake" and root user have access to it.
if not await sys_db.has_database('test'):
    await sys_db.create_database(
        name='test',
        users=[
            {'username': 'jane', 'password': 'foo', 'active': True},
            {'username': 'john', 'password': 'bar', 'active': True},
            {'username': 'jake', 'password': 'baz', 'active': True},
        ],
    )

# Connect to the new "test" database as user "jane".
```

(continues on next page)

(continued from previous page)

```
db = await client.db('test', username='jane', password='foo')

# Make sure that user "jane" has read and write permissions.
await sys_db.update_permission(username='jane', permission='rw', database='test')

# Retrieve various database and server information.
db.name
db.username
await db.version()
await db.status()
await db.details()
await db.collections()
await db.graphs()
await db.engine()

# Delete the database. Note that the new users will remain.
await sys_db.delete_database('test')
```

See *ArangoClient* and *StandardDatabase* for API specification.

3.3 Collections

A **collection** contains *documents*. It is uniquely identified by its name which must consist only of hyphen, underscore and alphanumeric characters. There are three types of collections in aioarangodb:

- **Standard Collection:** contains regular documents.
- **Vertex Collection:** contains vertex documents for graphs. See *here* for more details.
- **Edge Collection:** contains edge documents for graphs. See *here* for more details.

Here is an example showing how you can manage standard collections:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = client.db('test', username='root', password='passwd')

# List all collections in the database.
await db.collections()

# Create a new collection named "students" if it does not exist.
# This returns an API wrapper for "students" collection.
if db.has_collection('students'):
    students = db.collection('students')
else:
    students = await db.create_collection('students')

# Retrieve collection properties.
students.name
students.db_name
await students.properties()
await students.revision()
```

(continues on next page)

(continued from previous page)

```
await students.statistics()
await students.checksum()
await students.count()

# Perform various operations.
await students.load()
await students.unload()
await students.truncate()
await students.configure(journal_size=3000000)

# Delete the collection.
await db.delete_collection('students')
```

See *StandardDatabase* and *StandardCollection* for API specification.

3.4 Documents

In aioarangodb, a **document** is a Python dictionary with the following properties:

- Is JSON serializable.
- May be nested to an arbitrary depth.
- May contain lists.
- Contains the `_key` field, which identifies the document uniquely within a specific collection.
- Contains the `_id` field (also called the *handle*), which identifies the document uniquely across all collections within a database. This ID is a combination of the collection name and the document key using the format `{collection}/{key}` (see example below).
- Contains the `_rev` field. ArangoDB supports MVCC (Multiple Version Concurrency Control) and is capable of storing each document in multiple revisions. Latest revision of a document is indicated by this field. The field is populated by ArangoDB and is not required as input unless you want to validate a document against its current revision.

For more information on documents and associated terminologies, refer to ArangoDB manual. Here is an example of a valid document in "students" collection:

```
{
    '_id': 'students/bruce',
    '_key': 'bruce',
    '_rev': '_Wm3dzEi--_',
    'first_name': 'Bruce',
    'last_name': 'Wayne',
    'address': {
        'street' : '1007 Mountain Dr.',
        'city': 'Gotham',
        'state': 'NJ'
    },
    'is_rich': True,
    'friends': ['robin', 'gordon']
}
```

Edge documents (edges) are similar to standard documents but with two additional required fields `_from` and `_to`. Values of these fields must be the handles of "from" and "to" vertex documents linked by the edge document in

question (see *Graphs* for details). Edge documents are contained in *edge collections*. Here is an example of a valid edge document in "friends" edge collection:

```
{
    '_id': 'friends/001',
    '_key': '001',
    '_rev': '_Wm3d4le--_',
    '_from': 'students/john',
    '_to': 'students/jane',
    'closeness': 9.5
}
```

Standard documents are managed via collection API wrapper:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Get the API wrapper for "students" collection.
students = db.collection('students')

# Create some test documents to play around with.
lola = {'_key': 'lola', 'GPA': 3.5, 'first': 'Lola', 'last': 'Martin'}
abby = {'_key': 'abby', 'GPA': 3.2, 'first': 'Abby', 'last': 'Page'}
john = {'_key': 'john', 'GPA': 3.6, 'first': 'John', 'last': 'Kim'}
emma = {'_key': 'emma', 'GPA': 4.0, 'first': 'Emma', 'last': 'Park'}

# Insert a new document. This returns the document metadata.
metadata = await students.insert(lola)
assert metadata['_id'] == 'students/lola'
assert metadata['_key'] == 'lola'

# Check if documents exist in the collection in multiple ways.
assert await students.has('lola') and 'john' not in students

# Retrieve the total document count in multiple ways.
assert await students.count() == 1

# Insert multiple documents in bulk.
await students.import_bulk([abby, john, emma])

# Retrieve one or more matching documents.
for student in await students.find({'first': 'John'}):
    assert student['_key'] == 'john'
    assert student['GPA'] == 3.6
    assert student['last'] == 'Kim'

# Retrieve a document by key.
await students.get('john')

# Retrieve a document by ID.
await students.get('students/john')

# Retrieve a document by body with "_id" field.
```

(continues on next page)

(continued from previous page)

```
await students.get({'_id': 'students/john'})

# Retrieve a document by body with "_key" field.
await students.get({'_key': 'john'})

# Retrieve multiple documents by ID, key or body.
await students.get_many(['abby', 'students/lola', {'_key': 'john'}])

# Update a single document.
lola['GPA'] = 2.6
await students.update(lola)

# Update one or more matching documents.
await students.update_match({'last': 'Park'}, {'GPA': 3.0})

# Replace a single document.
emma['GPA'] = 3.1
await students.replace(emma)

# Replace one or more matching documents.
becky = {'first': 'Becky', 'last': 'Solis', 'GPA': '3.3'}
await students.replace_match({'first': 'Emma'}, becky)

# Delete a document by key.
await students.delete('john')

# Delete a document by ID.
await students.delete('students/lola')

# Delete a document by body with "_id" or "_key" field.
await students.delete(emma)

# Delete multiple documents. Missing ones are ignored.
await students.delete_many([abby, 'john', 'students/lola'])

# Iterate through all documents and update individually.
async for student in students:
    student['GPA'] = 4.0
    student['happy'] = True
    students.update(student)
```

You can manage documents via database API wrappers also, but only simple operations (i.e. get, insert, update, replace, delete) are supported and you must provide document IDs instead of keys:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Create some test documents to play around with.
# The documents must have the "_id" field instead.
lola = {'_id': 'students/lola', 'GPA': 3.5}
abby = {'_id': 'students/abby', 'GPA': 3.2}
```

(continues on next page)

(continued from previous page)

```
john = {'_id': 'students/john', 'GPA': 3.6}
emma = {'_id': 'students/emma', 'GPA': 4.0}

# Insert a new document.
metadata = await db.insert_document('students', lola)
assert metadata['_id'] == 'students/lola'
assert metadata['_key'] == 'lola'

# Check if a document exists.
assert await db.has_document(lola) is True

# Get a document (by ID or body with "_id" field).
await db.document('students/lola')
await db.document(abby)

# Update a document.
lola['GPA'] = 3.6
await db.update_document(lola)

# Replace a document.
lola['GPA'] = 3.4
await db.replace_document(lola)

# Delete a document (by ID or body with "_id" field).
await db.delete_document('students/lola')
```

See *StandardDatabase* and *StandardCollection* for API specification.

When managing documents, using collection API wrappers over database API wrappers is recommended as more operations are available and less sanity checking is performed under the hood.

3.5 Indexes

Indexes can be added to collections to speed up document lookups. Every collection has a primary hash index on `_key` field by default. This index cannot be deleted or modified. Every edge collection has additional indexes on fields `_from` and `_to`. For more information on indexes, refer to ArangoDB manual.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Create a new collection named "cities".
cities = await db.create_collection('cities')

# List the indexes in the collection.
await cities.indexes()

# Add a new hash index on document fields "continent" and "country".
index = await cities.add_hash_index(fields=['continent', 'country'], unique=True)
```

(continues on next page)

(continued from previous page)

```
# Add new fulltext indexes on fields "continent" and "country".
index = await cities.add_fulltext_index(fields=['continent'])
index = await cities.add_fulltext_index(fields=['country'])

# Add a new skiplist index on field 'population'.
index = await cities.add_skiplist_index(fields=['population'], sparse=False)

# Add a new geo-spatial index on field 'coordinates'.
index = await cities.add_geo_index(fields=['coordinates'])

# Add a new persistent index on field 'currency'.
index = await cities.add_persistent_index(fields=['currency'], sparse=True)

# Add a new TTL (time-to-live) index on field 'currency'.
index = await cities.add_ttl_index(fields=['ttl'], expiry_time=200)

# Indexes may be added with a name that can be referred to in AQL queries.
index = await cities.add_hash_index(fields=['country'], name='my_hash_index')

# Delete the last index from the collection.
await cities.delete_index(index['id'])
```

See *StandardCollection* for API specification.

3.6 Graphs

A **graph** consists of vertices and edges. Vertices are stored as documents in *vertex collections* and edges stored as documents in *edge collections*. The collections used in a graph and their relations are specified with *edge definitions*. For more information, refer to ArangoDB manual.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# List existing graphs in the database.
await db.graphs()

# Create a new graph named "school" if it does not already exist.
# This returns an API wrapper for "school" graph.
if await db.has_graph('school'):
    school = db.graph('school')
else:
    school = await db.create_graph('school')

# Retrieve various graph properties.
school.name
school.db_name
await school.vertex_collections()
```

(continues on next page)

(continued from previous page)

```
await school.edge_definitions()

# Delete the graph.
await db.delete_graph('school')
```

3.6.1 Edge Definitions

An **edge definition** specifies a directed relation in a graph. A graph can have arbitrary number of edge definitions. Each edge definition consists of the following components:

- **From Vertex Collections:** contain "from" vertices referencing "to" vertices.
- **To Vertex Collections:** contain "to" vertices referenced by "from" vertices.
- **Edge Collection:** contains edges that link "from" and "to" vertices.

Here is an example body of an edge definition:

```
{
    'edge_collection': 'teach',
    'from_vertex_collections': ['teachers'],
    'to_vertex_collections': ['lectures']
}
```

Here is an example showing how edge definitions are managed:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Get the API wrapper for graph "school".
if await db.has_graph('school'):
    school = db.graph('school')
else:
    school = await db.create_graph('school')

# Create an edge definition named "teach". This creates any missing
# collections and returns an API wrapper for "teach" edge collection.
if not await school.has_edge_definition('teach'):
    teach = await school.create_edge_definition(
        edge_collection='teach',
        from_vertex_collections=['teachers'],
        to_vertex_collections=['teachers']
    )

# List edge definitions.
await school.edge_definitions()

# Replace the edge definition.
await school.replace_edge_definition(
    edge_collection='teach',
    from_vertex_collections=['teachers'],
```

(continues on next page)

(continued from previous page)

```
    to_vertex_collections=['lectures']
)

# Delete the edge definition (and its collections).
await school.delete_edge_definition('teach', purge=True)
```

3.6.2 Vertex Collections

A **vertex collection** contains vertex documents, and shares its namespace with all other types of collections. Each graph can have an arbitrary number of vertex collections. Vertex collections that are not part of any edge definition are called **orphan collections**. You can manage vertex documents via standard collection API wrappers, but using vertex collection API wrappers provides additional safeguards:

- All modifications are executed in transactions.
- If a vertex is deleted, all connected edges are also automatically deleted.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Get the API wrapper for graph "school".
school = db.graph('school')

# Create a new vertex collection named "teachers" if it does not exist.
# This returns an API wrapper for "teachers" vertex collection.
if await school.has_vertex_collection('teachers'):
    teachers = school.vertex_collection('teachers')
else:
    teachers = await school.create_vertex_collection('teachers')

# List vertex collections in the graph.
await school.vertex_collections()

# Vertex collections have similar interface as standard collections.
await teachers.properties()
await teachers.insert({'_key': 'jon', 'name': 'Jon'})
await teachers.update({'_key': 'jon', 'age': 35})
await teachers.replace({'_key': 'jon', 'name': 'Jon', 'age': 36})
await teachers.get('jon')
await teachers.has('jon')
await teachers.delete('jon')
```

You can manage vertices via graph API wrappers also, but you must use document IDs instead of keys where applicable.

Example:

```
# Initialize the ArangoDB client.
client = ArangoClient()
```

(continues on next page)

(continued from previous page)

```
# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Get the API wrapper for graph "school".
school = db.graph('school')

# Create a new vertex collection named "lectures" if it does not exist.
# This returns an API wrapper for "lectures" vertex collection.
if await school.has_vertex_collection('lectures'):
    school.vertex_collection('lectures')
else:
    await school.create_vertex_collection('lectures')

# The "_id" field is required instead of "_key" field (except for insert).
await school.insert_vertex('lectures', {'_key': 'CSC101'})
await school.update_vertex({'_id': 'lectures/CSC101', 'difficulty': 'easy'})
await school.replace_vertex({'_id': 'lectures/CSC101', 'difficulty': 'hard'})
await school.has_vertex('lectures/CSC101')
await school.vertex('lectures/CSC101')
await school.delete_vertex('lectures/CSC101')
```

See *Graph* and *VertexCollection* for API specification.

3.6.3 Edge Collections

An **edge collection** contains *edge documents*, and shares its namespace with all other types of collections. You can manage edge documents via standard collection API wrappers, but using edge collection API wrappers provides additional safeguards:

- All modifications are executed in transactions.
- Edge documents are checked against the edge definitions on insert.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Get the API wrapper for graph "school".
school = db.graph('school')

# Get the API wrapper for edge collection "teach".
if school.has_edge_definition('teach'):
    teach = school.edge_collection('teach')
else:
    teach = await school.create_edge_definition(
        edge_collection='teach',
        from_vertex_collections=['teachers'],
        to_vertex_collections=['lectures']
    )
```

(continues on next page)

(continued from previous page)

```
# Edge collections have a similar interface as standard collections.
await teach.insert({
    '_key': 'jon-CSC101',
    '_from': 'teachers/jon',
    '_to': 'lectures/CSC101'
})
await teach.replace({
    '_key': 'jon-CSC101',
    '_from': 'teachers/jon',
    '_to': 'lectures/CSC101',
    'online': False
})
await teach.update({
    '_key': 'jon-CSC101',
    'online': True
})
await teach.has('jon-CSC101')
await teach.get('jon-CSC101')
await teach.delete('jon-CSC101')

# Create an edge between two vertices (essentially the same as insert).
await teach.link('teachers/jon', 'lectures/CSC101', data={'online': False})

# List edges going in/out of a vertex.
await teach.edges('teachers/jon', direction='in')
await teach.edges('teachers/jon', direction='out')
```

You can manage edges via graph API wrappers also, but you must use document IDs instead of keys where applicable.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Get the API wrapper for graph "school".
school = db.graph('school')

# The "_id" field is required instead of "_key" field.
await school.insert_edge(
    collection='teach',
    edge={
        '_id': 'teach/jon-CSC101',
        '_from': 'teachers/jon',
        '_to': 'lectures/CSC101'
    }
)
await school.replace_edge({
    '_id': 'teach/jon-CSC101',
    '_from': 'teachers/jon',
    '_to': 'lectures/CSC101',
    'online': False,
```

(continues on next page)

(continued from previous page)

```
})
await school.update_edge({
    '_id': 'teach/jon-CSC101',
    'online': True
})
await school.has_edge('teach/jon-CSC101')
await school.edge('teach/jon-CSC101')
await school.delete_edge('teach/jon-CSC101')
await school.link('teach', 'teachers/jon', 'lectures/CSC101')
await school.edges('teach', 'teachers/jon', direction='in')
```

See *Graph* and *EdgeCollection* for API specification.

3.6.4 Graph Traversals

Graph traversals are executed via the `arango.graph.Graph.traverse()` method. Each traversal can span across multiple vertex collections, and walk over edges and vertices using various algorithms.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Get the API wrapper for graph "school".
school = db.graph('school')

# Get API wrappers for "from" and "to" vertex collections.
teachers = school.vertex_collection('teachers')
lectures = school.vertex_collection('lectures')

# Get the API wrapper for the edge collection.:
teach = school.edge_collection('teach')

# Insert vertices into the graph.
await teachers.insert({'_key': 'jon', 'name': 'Professor jon'})
await lectures.insert({'_key': 'CSC101', 'name': 'Introduction to CS'})
await lectures.insert({'_key': 'MAT223', 'name': 'Linear Algebra'})
await lectures.insert({'_key': 'STA201', 'name': 'Statistics'})

# Insert edges into the graph.
await teach.insert({'_from': 'teachers/jon', '_to': 'lectures/CSC101'})
await teach.insert({'_from': 'teachers/jon', '_to': 'lectures/STA201'})
await teach.insert({'_from': 'teachers/jon', '_to': 'lectures/MAT223'})

# Traverse the graph in outbound direction, breath-first.
await school.traverse(
    start_vertex='teachers/jon',
    direction='outbound',
    strategy='bfs',
    edge_uniqueness='global',
```

(continues on next page)

(continued from previous page)

```
    vertex_uniqueness='global',
)
```

See `arango.graph.Graph.traverse()` for API specification.

3.7 AQL

ArangoDB Query Language (AQL) is used to read and write data. It is similar to SQL for relational databases, but without the support for data definition operations such as creating or deleting *databases*, *collections* or *indexes*. For more information, refer to ArangoDB manual.

3.7.1 AQL Queries

AQL queries are invoked from AQL API wrapper. Executing queries returns *result cursors*.

Example:

```
from aioarangodb import ArangoClient, AQLQueryKillError

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Insert some test documents into "students" collection.
await db.collection('students').insert_many([
    {'_key': 'Abby', 'age': 22},
    {'_key': 'John', 'age': 18},
    {'_key': 'Mary', 'age': 21}
])

# Get the AQL API wrapper.
aql = db.aql

# Retrieve the execution plan without running the query.
await aql.explain('FOR doc IN students RETURN doc')

# Validate the query without executing it.
await aql.validate('FOR doc IN students RETURN doc')

# Execute the query
cursor = await db.aql.execute(
  'FOR doc IN students FILTER doc.age < @value RETURN doc',
  bind_vars={'value': 19}
)
# Iterate through the result cursor
student_keys = [doc['_key'] async for doc in cursor]

# List currently running queries.
await aql.queries()

# List any slow queries.
```

(continues on next page)

(continued from previous page)

```
await aql.slow_queries()

# Clear slow AQL queries if any.
await aql.clear_slow_queries()

# Retrieve AQL query tracking properties.
await aql.tracking()

# Configure AQL query tracking properties.
await aql.set_tracking(
    max_slow_queries=10,
    track_bind_vars=True,
    track_slow_queries=True
)

# Kill a running query (this should fail due to invalid ID).
try:
    await aql.kill('some_query_id')
except AQLQueryKillError as err:
    assert err.http_code == 404
    assert err.error_code == 1591
    assert 'cannot kill query' in err.message
```

See *AQL* for API specification.

3.7.2 AQL User Functions

AQL User Functions are custom functions you define in Javascript to extend AQL functionality. They are somewhat similar to SQL procedures.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = client.db('test', username='root', password='passwd')

# Get the AQL API wrapper.
aql = db.aql

# Create a new AQL user function.
aql.create_function(
    # Grouping by name prefix is supported.
    name='functions::temperature::converter',
    code='function (celsius) { return celsius * 1.8 + 32; }'
)
# List AQL user functions.
aql.functions()

# Delete an existing AQL user function.
aql.delete_function('functions::temperature::converter')
```

See *AQL* for API specification.

3.7.3 AQL Query Cache

AQL Query Cache is used to minimize redundant calculation of the same query results. It is useful when read queries are issued frequently and write queries are not.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = client.db('test', username='root', password='passwd')

# Get the AQL API wrapper.
aql = db.aql

# Retrieve AQL query cache properties.
aql.cache.properties()

# Configure AQL query cache properties
aql.cache.configure(mode='demand', max_results=10000)

# Clear results in AQL query cache.
aql.cache.clear()
```

See *AQLQueryCache* for API specification.

3.8 Cursors

Many operations provided by aioarangodb (e.g. executing *AQL* queries) return result **cursors** to batch the network communication between ArangoDB server and aioarangodb client. Each HTTP request from a cursor fetches the next batch of results (usually documents). Depending on the query, the total number of items in the result set may or may not be known in advance.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Set up some test data to query against.
await db.collection('students').insert_many([
    {'_key': 'Abby', 'age': 22},
    {'_key': 'John', 'age': 18},
    {'_key': 'Mary', 'age': 21},
    {'_key': 'Suzy', 'age': 23},
    {'_key': 'Dave', 'age': 20}
])

# Execute an AQL query which returns a cursor object.
```

(continues on next page)

(continued from previous page)

```
cursor = await db.aql.execute(
    'FOR doc IN students FILTER doc.age > @val RETURN doc',
    bind_vars={'val': 17},
    batch_size=2,
    count=True
)

# Get the cursor ID.
cursor.id

# Get the items in the current batch.
await cursor.batch()

# Check if the current batch is empty.
await cursor.empty()

# Get the total count of the result set.
await cursor.count()

# Flag indicating if there are more to be fetched from server.
await cursor.has_more()

# Flag indicating if the results are cached.
await cursor.cached()

# Get the cursor statistics.
await cursor.statistics()

# Get the performance profile.
await cursor.profile()

# Get any warnings produced from the query.
await cursor.warnings()

# Return the next item from the cursor. If current batch is depleted, the
# next batch if fetched from the server automatically.
await cursor.next()

# Return the next item from the cursor. If current batch is depleted, an
# exception is thrown. You need to fetch the next batch manually.
cursor.pop()

# Fetch the next batch and add them to the cursor object.
await cursor.fetch()

# Delete the cursor from the server.
await cursor.close()
```

See *Cursor* for API specification.

If the fetched result batch is depleted while you are iterating over a cursor (or while calling the method `arango.cursor.Cursor.next()`), aioarangodb automatically sends an HTTP request to the server to fetch the next batch (just-in-time style). To control exactly when the fetches occur, you can use methods `arango.cursor.Cursor.fetch()` and `arango.cursor.Cursor.pop()` instead.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = client.db('test', username='root', password='passwd')

# Set up some test data to query against.
db.collection('students').insert_many([
    {'_key': 'Abby', 'age': 22},
    {'_key': 'John', 'age': 18},
    {'_key': 'Mary', 'age': 21}
])

# If you iterate over the cursor or call cursor.next(), batches are
# fetched automatically from the server just-in-time style.
cursor = db.aql.execute('FOR doc IN students RETURN doc', batch_size=1)
result = [doc for doc in cursor]

# Alternatively, you can manually fetch and pop for finer control.
cursor = db.aql.execute('FOR doc IN students RETURN doc', batch_size=1)
while cursor.has_more(): # Fetch until nothing is left on the server.
    cursor.fetch()
while not cursor.empty(): # Pop until nothing is left on the cursor.
    cursor.pop()
```

3.9 Asynchronous Execution

In **asynchronous execution**, aioarangodb sends API requests to ArangoDB in fire-and-forget style. The server processes the requests in the background, and the results can be retrieved once available via *AsyncJob* objects.

Example :

```
import time

from aioarangodb import (
    ArangoClient,
    AQLQueryExecuteError,
    AsyncJobCancelError,
    AsyncJobClearError
)

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Begin async execution. This returns an instance of AsyncDatabase, a
# database-level API wrapper tailored specifically for async execution.
async_db = db.begin_async_execution(return_result=True)

# Child wrappers are also tailored for async execution.
async_aql = async_db.aql
```

(continues on next page)

(continued from previous page)

```
async_col = async_db.collection('students')

# API execution context is always set to "async".
assert async_db.context == 'async'
assert async_aql.context == 'async'
assert async_col.context == 'async'

# On API execution, AsyncJob objects are returned instead of results.
job1 = await async_col.insert({'_key': 'Neal'})
job2 = await async_col.insert({'_key': 'Lily'})
job3 = await async_aql.execute('RETURN 100000')
job4 = await async_aql.execute('INVALID QUERY')  # Fails due to syntax error.

# Retrieve the status of each async job.
for job in [job1, job2, job3, job4]:
    # Job status can be "pending", "done" or "cancelled".
    assert await job.status() in {'pending', 'done', 'cancelled'}

    # Let's wait until the jobs are finished.
    while await job.status() != 'done':
        time.sleep(0.1)

# Retrieve the results of successful jobs.
metadata = job1.result()
assert metadata['_id'] == 'students/Neal'

metadata = job2.result()
assert metadata['_id'] == 'students/Lily'

cursor = job3.result()
assert await cursor.next() == 100000

# If a job fails, the exception is propagated up during result retrieval.
try:
    result = job4.result()
except AQLQueryExecuteError as err:
    assert err.http_code == 400
    assert err.error_code == 1501
    assert 'syntax error' in err.message

# Cancel a job. Only pending jobs still in queue may be cancelled.
# Since job3 is done, there is nothing to cancel and an exception is raised.
try:
    await job3.cancel()
except AsyncJobCancelError as err:
    assert err.message.endswith('job {} not found'.format(job3.id))

# Clear the result of a job from ArangoDB server to free up resources.
# Result of job4 was removed from the server automatically upon retrieval,
# so attempt to clear it raises an exception.
try:
    await job4.clear()
except AsyncJobClearError as err:
    assert err.message.endswith('job {} not found'.format(job4.id))

# List the IDs of the first 100 async jobs completed.
await db.async_jobs(status='done', count=100)
```

(continues on next page)

(continued from previous page)

```
# List the IDs of the first 100 async jobs still pending.
await db.async_jobs(status='pending', count=100)

# Clear all async jobs still sitting on the server.
await db.clear_async_jobs()
```

Note: Be mindful of server-side memory capacity when issuing a large number of async requests in small time interval.

See *AsyncDatabase* and *AsyncJob* for API specification.

3.10 Batch Execution

In **batch execution**, requests to ArangoDB server are stored in client-side in-memory queue, and committed together in a single HTTP call. After the commit, results can be retrieved later from *BatchJob* objects.

Example:

```
from aioarangodb import ArangoClient, AQLQueryExecuteError

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Get the API wrapper for "students" collection.
students = db.collection('students')

# Begin batch execution via context manager. This returns an instance of
# BatchDatabase, a database-level API wrapper tailored specifically for
# batch execution. The batch is automatically committed when exiting the
# context. The BatchDatabase wrapper cannot be reused after commit.
with db.begin_batch_execution(return_result=True) as batch_db:

    # Child wrappers are also tailored for batch execution.
    batch_aql = batch_db.aql
    batch_col = batch_db.collection('students')

    # API execution context is always set to "batch".
    assert batch_db.context == 'batch'
    assert batch_aql.context == 'batch'
    assert batch_col.context == 'batch'

    # BatchJob objects are returned instead of results.
    job1 = await batch_col.insert({'_key': 'Kris'})
    job2 = await batch_col.insert({'_key': 'Rita'})
    job3 = await batch_aql.execute('RETURN 100000')
    job4 = await batch_aql.execute('INVALID QUERY')  # Fails due to syntax error.

# Upon exiting context, batch is automatically committed.
assert 'Kris' in students
```

(continues on next page)

(continued from previous page)

```
assert 'Rita' in students

# Retrieve the status of each batch job.
for job in batch_db.queued_jobs():
    # Status is set to either "pending" (transaction is not committed yet
    # and result is not available) or "done" (transaction is committed and
    # result is available).
    assert await job.status() in {'pending', 'done'}

# Retrieve the results of successful jobs.
metadata = await job1.result()
assert metadata['_id'] == 'students/Kris'

metadata = await job2.result()
assert metadata['_id'] == 'students/Rita'

cursor = await job3.result()
assert cursor.next() == 100000

# If a job fails, the exception is propagated up during result retrieval.
try:
    result = await  job4.result()
except AQLQueryExecuteError as err:
    assert err.http_code == 400
    assert err.error_code == 1501
    assert 'syntax error' in err.message

# Batch execution can be initiated without using a context manager.
# If return_result parameter is set to False, no jobs are returned.
batch_db = db.begin_batch_execution(return_result=False)
await batch_db.collection('students').insert({'_key': 'Jake'})
await batch_db.collection('students').insert({'_key': 'Jill'})

# The commit must be called explicitly.
await batch_db.commit()
assert 'Jake' in students
assert 'Jill' in students
```

Note:

- Be mindful of client-side memory capacity when issuing a large number of requests in single batch execution.
- *BatchDatabase* and *BatchJob* instances are stateful objects, and should not be shared across multiple threads.
- *BatchDatabase* instance cannot be reused after commit.

See *BatchDatabase* and *BatchJob* for API specification.

3.11 Transactions

In **transactions**, requests to ArangoDB server are committed as a single, logical unit of work (ACID compliant).

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')
col = db.collection('students')

# Begin a transaction. Read and write collections must be declared ahead of
# time. This returns an instance of TransactionDatabase, database-level
# API wrapper tailored specifically for executing transactions.
txn_db = await db.begin_transaction(read=col.name, write=col.name)

# The API wrapper is specific to a single transaction with a unique ID.
txn_db.transaction_id

# Child wrappers are also tailored only for the specific transaction.
txn_aql = txn_db.aql
txn_col = txn_db.collection('students')

# API execution context is always set to "transaction".
assert txn_db.context == 'transaction'
assert txn_aql.context == 'transaction'
assert txn_col.context == 'transaction'

assert '_rev' in await txn_col.insert({'_key': 'Abby'})
assert '_rev' in await txn_col.insert({'_key': 'John'})
assert '_rev' in await txn_col.insert({'_key': 'Mary'})

# Check the transaction status.
await txn_db.transaction_status()

# Commit the transaction.
await txn_db.commit_transaction()
assert await col.has('Abby')
assert await col.has('John')
assert await col.has('Mary')
assert await col.count() == 3

# Begin another transaction. Note that the wrappers above are specific to
# the last transaction and cannot be reused. New ones must be created.
txn_db = db.begin_transaction(read=col.name, write=col.name)
txn_col = txn_db.collection('students')
assert '_rev' in await txn_col.insert({'_key': 'Kate'})
assert '_rev' in await txn_col.insert({'_key': 'Mike'})
assert '_rev' in await txn_col.insert({'_key': 'Lily'})
assert await txn_col.count() == 6

# Abort the transaction
await txn_db.abort_transaction()
assert not await col.has('Kate')
assert not await col.has('Mike')
assert not await col.has('Lily')
assert await col.count() == 3  # transaction is aborted so txn_col cannot be used
```

See *TransactionDatabase* for API specification.

Alternatively, you can use `arango.database.StandardDatabase.execute_transaction()` to run raw Javascript code in a transaction.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Get the API wrapper for "students" collection.
students = db.collection('students')

# Execute transaction in raw Javascript.
result = await db.execute_transaction(
    command='''
    function () {{
        var db = require('internal').db;
        db.students.save(params.student1);
        if (db.students.count() > 1) {
            db.students.save(params.student2);
        } else {
            db.students.save(params.student3);
        }
        return true;
    }}
    ''',
    params={
        'student1': {'_key': 'Lucy'},
        'student2': {'_key': 'Greg'},
        'student3': {'_key': 'Dona'}
    },
    read='students',  # Specify the collections read.
    write='students'  # Specify the collections written.
)
assert result is True
assert await students.has('Lucy')
assert await students.has('Greg')
assert await students.has('Dona')
```

3.12 Server Administration

Python-arango provides operations for server administration and monitoring. Most of these operations can only be performed by admin users via `_system` database.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "_system" database as root user.
```

(continues on next page)

(continued from previous page)

```
sys_db = await client.db('_system', username='root', password='passwd')

# Retrieve the server version.
await sys_db.version()

# Retrieve the server details.
await sys_db.details()

# Retrieve the target DB version.
await sys_db.required_db_version()

# Retrieve the database engine.
await sys_db.engine()

# Retrieve the server time.
await sys_db.time()

# Retrieve the server role in a cluster.
await sys_db.role()

# Retrieve the server statistics.
await sys_db.statistics()

# Read the server log.
await sys_db.read_log(level="debug")

# Retrieve the log levels.
await sys_db.log_levels()

# Set the log .
await sys_db.set_log_levels(
    agency='DEBUG',
    collector='INFO',
    threads='WARNING'
)

# Echo the last request.
await sys_db.echo()

# Reload the routing collection.
await sys_db.reload_routing()

# Retrieve server metrics.
await sys_db.metrics()
```

See *StandardDatabase* for API specification.

3.13 Users and Permissions

Python-arango provides operations for managing users and permissions. Most of these operations can only be performed by admin users via `_system` database.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "_system" database as root user.
sys_db = await client.db('_system', username='root', password='passwd')

# List all users.
await sys_db.users()

# Create a new user.
await sys_db.create_user(
    username='johndoe@gmail.com',
    password='first_password',
    active=True,
    extra={'team': 'backend', 'title': 'engineer'}
)

# Check if a user exists.
await sys_db.has_user('johndoe@gmail.com')

# Retrieve details of a user.
await sys_db.user('johndoe@gmail.com')

# Update an existing user.
await sys_db.update_user(
    username='johndoe@gmail.com',
    password='second_password',
    active=True,
    extra={'team': 'frontend', 'title': 'engineer'}
)

# Replace an existing user.
await sys_db.replace_user(
    username='johndoe@gmail.com',
    password='third_password',
    active=True,
    extra={'team': 'frontend', 'title': 'architect'}
)

# Retrieve user permissions for all databases and collections.
await sys_db.permissions('johndoe@gmail.com')

# Retrieve user permission for "test" database.
await sys_db.permission(
    username='johndoe@gmail.com',
    database='test'
)

# Retrieve user permission for "students" collection in "test" database.
await sys_db.permission(
    username='johndoe@gmail.com',
    database='test',
    collection='students'
)
```

(continues on next page)

(continued from previous page)

```
# Update user permission for "test" database.
await sys_db.update_permission(
    username='johndoe@gmail.com',
    permission='rw',
    database='test'
)

# Update user permission for "students" collection in "test" database.
await sys_db.update_permission(
    username='johndoe@gmail.com',
    permission='ro',
    database='test',
    collection='students'
)

# Reset user permission for "test" database.
await sys_db.reset_permission(
    username='johndoe@gmail.com',
    database='test'
)

# Reset user permission for "students" collection in "test" database.
await sys_db.reset_permission(
    username='johndoe@gmail.com',
    database='test',
    collection='students'
)
```

See *StandardDatabase* for API specification.

3.14 Tasks

ArangoDB can schedule user-defined Javascript snippets as one-time or periodic (re-scheduled after each execution) tasks. Tasks are executed in the context of the database they are defined in.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# List all active tasks
await db.tasks()

# Create a new task which simply prints parameters.
await db.create_task(
    name='test_task',
    command='''
        var task = function(params){
            var db = require('@arangodb');
            db.print(params);
```

(continues on next page)

(continued from previous page)

```
        }
        task(params);
    ''',
    params={'foo': 'bar'},
    offset=300,
    period=10,
    task_id='001'
)

# Retrieve details on a task by ID.
await db.task('001')

# Delete an existing task by ID.
await db.delete_task('001', ignore_missing=True)
```

Note: When deleting a database, any tasks that were initialized under its context remain active. It is therefore advisable to delete any running tasks before deleting the database.

Refer to *StandardDatabase* class for API specification.

3.15 Write-Ahead Log (WAL)

Write-Ahead Log (WAL) is a set of append-only files recording all writes on ArangoDB server. It is typically used to perform data recovery after a crash or synchronize slave databases with master databases in replicated environments. WAL operations can only be performed by admin users via `_system` database.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "_system" database as root user.
sys_db = await client.db('_system', username='root', password='passwd')

# Get the WAL API wrapper.
wal = sys_db.wal

# Configure WAL properties.
await wal.configure(
    historic_logs=15,
    oversized_ops=False,
    log_size=30000000,
    reserve_logs=5,
    throttle_limit=0,
    throttle_wait=16000
)

# Retrieve WAL properties.
await wal.properties()

# List WAL transactions.
```

(continues on next page)

(continued from previous page)

```
await wal.transactions()

# Flush WAL with garbage collection.
await wal.flush(garbage_collect=True)

# Get the available ranges of tick values.
await wal.tick_ranges()

# Get the last available tick value.
await wal.last_tick()

# Get recent WAL operations.
await wal.tail()
```

See `WriteAheadLog` for API specification.

3.16 Pregel

Python-arango provides support for **Pregel**, ArangoDB module for distributed iterative graph processing. For more information, refer to ArangoDB manual.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Get the Pregel API wrapper.
pregel = db.pregel

# Start a new Pregel job in "school" graph.
job_id = await db.pregel.create_job(
    graph='school',
    algorithm='pagerank',
    store=False,
    max_gss=100,
    thread_count=1,
    async_mode=False,
    result_field='result',
    algorithm_params={'threshold': 0.000001}
)

# Retrieve details of a Pregel job by ID.
job = await pregel.job(job_id)

# Delete a Pregel job by ID.
await pregel.delete_job(job_id)
```

See *Pregel* for API specification.

3.17 Foxx

Python-arango provides support for **Foxx**, a microservice framework which lets you define custom HTTP endpoints to extend ArangoDB's REST API. For more information, refer to ArangoDB manual.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "_system" database as root user.
db = client.db('_system', username='root', password='passwd')

# Get the Foxx API wrapper.
foxx = db.foxx

# Define the test mount point.
service_mount = '/test_mount'

# List services.
foxx.services()

# Create a service using source on server.
await foxx.create_service(
    mount=service_mount,
    source='/tmp/service.zip',
    config={},
    dependencies={},
    development=True,
    setup=True,
    legacy=True
)

# Update (upgrade) a service.
service = await db.foxx.update_service(
    mount=service_mount,
    source='/tmp/service.zip',
    config={},
    dependencies={},
    teardown=True,
    setup=True,
    legacy=False
)

# Replace (overwrite) a service.
service = await db.foxx.replace_service(
    mount=service_mount,
    source='/tmp/service.zip',
    config={},
    dependencies={},
    teardown=True,
    setup=True,
    legacy=True,
    force=False
)
```

(continues on next page)

(continued from previous page)

```
# Get service details.
await foxx.service(service_mount)

# Manage service configuration.
await foxx.config(service_mount)
await foxx.update_config(service_mount, config={})
await foxx.replace_config(service_mount, config={})

# Manage service dependencies.
await foxx.dependencies(service_mount)
await foxx.update_dependencies(service_mount, dependencies={})
await foxx.replace_dependencies(service_mount, dependencies={})

# Toggle development mode for a service.
await foxx.enable_development(service_mount)
await foxx.disable_development(service_mount)

# Other miscellaneous functions.
await foxx.readme(service_mount)
await foxx.swagger(service_mount)
await foxx.download(service_mount)
await foxx.commit(service_mount)
await foxx.scripts(service_mount)
await foxx.run_script(service_mount, 'setup', [])
await foxx.run_tests(service_mount, reporter='xunit', output_format='xml')

# Delete a service.
await foxx.delete_service(service_mount)
```

You can also manage Foxx services by using zip or Javascript files directly:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "_system" database as root user.
db = await client.db('_system', username='root', password='passwd')

# Get the Foxx API wrapper.
foxx = db.foxx

# Define the test mount point.
service_mount = '/test_mount'

# Create a service by providing a file directly.
await foxx.create_service_with_file(
    mount=service_mount,
    filename='/home/user/service.zip',
    development=True,
    setup=True,
    legacy=True
)

# Update (upgrade) a service by providing a file directly.
await foxx.update_service_with_file(
```

(continues on next page)

(continued from previous page)

```
    mount=service_mount,
    filename='/home/user/service.zip',
    teardown=False,
    setup=True,
    legacy=True,
    force=False
)

# Replace a service by providing a file directly.
await foxx.replace_service_with_file(
    mount=service_mount,
    filename='/home/user/service.zip',
    teardown=False,
    setup=True,
    legacy=True,
    force=False
)

# Delete a service.
await foxx.delete_service(service_mount)
```

See *Foxx* for API specification.

3.18 Views and ArangoSearch

Python-arango supports **view** management. For more information on view properties, refer to ArangoDB manual.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Retrieve list of views.
await db.views()

# Create a view.
await db.create_view(
    name='foo',
    view_type='arangosearch',
    properties={
        'cleanupIntervalStep': 0,
        'consolidationIntervalMsec': 0
    }
)

# Rename a view.
await db.rename_view('foo', 'bar')

# Retrieve view properties.
await db.view('bar')
```

(continues on next page)

(continued from previous page)

```
# Partially update view properties.
await db.update_view(
    name='bar',
    properties={
        'cleanupIntervalStep': 1000,
        'consolidationIntervalMsec': 200
    }
)

# Replace view properties. Unspecified ones are reset to default.
await db.replace_view(
    name='bar',
    properties={'cleanupIntervalStep': 2000}
)

# Delete a view.
await db.delete_view('bar')
```

Python-arango also supports **ArangoSearch** views.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Create an ArangoSearch view.
await db.create_arangosearch_view(
    name='arangosearch_view',
    properties={'cleanupIntervalStep': 0}
)

# Partially update an ArangoSearch view.
await db.update_arangosearch_view(
    name='arangosearch_view',
    properties={'cleanupIntervalStep': 1000}
)

# Replace an ArangoSearch view.
await db.replace_arangosearch_view(
    name='arangosearch_view',
    properties={'cleanupIntervalStep': 2000}
)

# ArangoSearch views can be retrieved or deleted using regular view API
await db.view('arangosearch_view')
await db.delete_view('arangosearch_view')
```

For more information on the content of view **properties**, see https://www.arangodb.com/docs/stable/http/views-arangosearch.html

Refer to *StandardDatabase* class for API specification.

3.19 Analyzers

Python-arango supports **analyzers**. For more information on analyzers, refer to ArangoDB manual.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Retrieve list of analyzers.
await db.analyzers()

# Create an analyzer.
await db.create_analyzer(
    name='test_analyzer',
    analyzer_type='identity',
    properties={},
    features=[]
)

# Delete an analyzer.
await db.delete_analyzer('test_analyzer', ignore_missing=True)
```

Refer to *StandardDatabase* class for API specification.

3.20 Multithreading

There are a few things you should consider before using aioarangodb in a multithreaded (or multiprocess) architecture.

3.20.1 Stateful Objects

Instances of the following classes are considered *stateful*, and should not be accessed across multiple threads without locks in place:

- *BatchDatabase* (see *Batch Execution*)
- *BatchJob* (see *Batch Execution*)
- *Cursor* (see *Cursors*)

3.20.2 HTTP Sessions

When *ArangoClient* is initialized, a **'aiohttp.ClientSession'_** instance is created per ArangoDB host connected. HTTP requests to a host are sent using only its corresponding session. For more information on how to override this behaviour, see http.

3.21 Error Handling

All aioarangodb exceptions inherit `aioarangodb.exceptions.ArangoError`, which splits into subclasses `aioarangodb.exceptions.ArangoServerError` and `aioarangodb.exceptions.ArangoClientError`.

3.21.1 Server Errors

`aioarangodb.exceptions.ArangoServerError` exceptions lightly wrap non-2xx HTTP responses coming from ArangoDB. Each exception object contains the error message, error code and HTTP request response details.

Example:

```
from aioarangodb import ArangoClient, ArangoServerError, DocumentInsertError

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Get the API wrapper for "students" collection.
students = db.collection('students')

try:
    await students.insert({'_key': 'John'})
    await students.insert({'_key': 'John'})  # duplicate key error

except DocumentInsertError as exc:

    assert isinstance(exc, ArangoServerError)
    assert exc.source == 'server'

    exc.message           # Exception message usually from ArangoDB
    exc.error_message     # Raw error message from ArangoDB
    exc.error_code        # Error code from ArangoDB
    exc.url               # URL (API endpoint)
    exc.http_method       # HTTP method (e.g. "POST")
    exc.http_headers      # Response headers
    exc.http_code         # Status code (e.g. 200)

    # You can inspect the ArangoDB response directly.
    response = exc.response
    response.method       # HTTP method (e.g. "POST")
    response.headers      # Response headers
    response.url          # Full request URL
    response.is_success   # Set to True if HTTP code is 2XX
    response.body         # JSON-deserialized response body
    response.raw_body     # Raw string response body
    response.status_text  # Status text (e.g "OK")
    response.status_code  # Status code (e.g. 200)
    response.error_code   # Error code from ArangoDB

    # You can also inspect the request sent to ArangoDB.
    request = exc.request
    request.method        # HTTP method (e.g. "post")
```

(continues on next page)

(continued from previous page)

```
    request.endpoint       # API endpoint starting with "/_api"
    request.headers        # Request headers
    request.params         # URL parameters
    request.data           # Request payload
```

See *Response* and *Request* for reference.

3.21.2 Client Errors

`aioarangodb.exceptions.ArangoClientError` exceptions originate from aioarangodb client itself. They do not contain error codes nor HTTP request response details.

Example:

```
from aioarangodb import ArangoClient, ArangoClientError, DocumentParseError

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Get the API wrapper for "students" collection.
students = db.collection('students')

try:
    await students.get({'_id': 'invalid_id'})  # malformed document

except DocumentParseError as exc:

    assert isinstance(exc, ArangoClientError)
    assert exc.source == 'client'

    # Only the error message is set.
    error_message = exc.message
    assert exc.error_code is None
    assert exc.error_message is None
    assert exc.url is None
    assert exc.http_method is None
    assert exc.http_code is None
    assert exc.http_headers is None
    assert exc.response is None
    assert exc.request is None
```

3.21.3 Exceptions

Below are all exceptions from aioarangodb.

3.21.4 Error Codes

The *errno* module contains a constant mapping to ArangoDB's error codes.

3.22 Replication

Replication allows you to replicate data onto another machine. It forms the basis of all disaster recovery and failover features ArangoDB offers. For more information, refer to ArangoDB manual.

Example:

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "test" database as root user.
db = await client.db('test', username='root', password='passwd')

# Get the Replication API wrapper.
replication = db.replication

# Create a new dump batch.
batch = await replication.create_dump_batch(ttl=1000)

# Extend an existing dump batch.
await replication.extend_dump_batch(batch['id'], ttl=1000)

# Get an overview of collections and indexes.
await replication.inventory(
    batch_id=batch['id'],
    include_system=True,
    all_databases=True
)

# Get an overview of collections and indexes in a cluster.
await replication.cluster_inventory(include_system=True)

# Get the events data for given collection.
await replication.dump(
    collection='students',
    batch_id=batch['id'],
    lower=0,
    upper=1000000,
    chunk_size=0,
    include_system=True,
    ticks=0,
    flush=True,
)

# Delete an existing dump batch.
await replication.delete_dump_batch(batch['id'])

# Get the logger state.
await replication.logger_state()

# Get the logger first tick value.
await replication.logger_first_tick()

# Get the replication applier configuration.
await replication.applier_config()
```

(continues on next page)

(continued from previous page)

```
# Update the replication applier configuration.
result = await replication.set_applier_config(
    endpoint='http://127.0.0.1:8529',
    database='test',
    username='root',
    password='passwd',
    max_connect_retries=120,
    connect_timeout=15,
    request_timeout=615,
    chunk_size=0,
    auto_start=True,
    adaptive_polling=False,
    include_system=True,
    auto_resync=True,
    auto_resync_retries=3,
    initial_sync_max_wait_time=405,
    connection_retry_wait_time=25,
    idle_min_wait_time=2,
    idle_max_wait_time=3,
    require_from_present=False,
    verbose=True,
    restrict_type='include',
    restrict_collections=['students']
)

# Get the replication applier state.
await replication.applier_state()

# Start the replication applier.
await replication.start_applier()

# Stop the replication applier.
await replication.stop_applier()

# Get the server ID.
await replication.server_id()

# Synchronize data from a remote (master) endpoint
await replication.synchronize(
    endpoint='tcp://master:8500',
    database='test',
    username='root',
    password='passwd',
    include_system=False,
    incremental=False,
    restrict_type='include',
    restrict_collections=['students']
)
```

See *Replication* for API specification.

3.23 Clusters

aioarangodb provides APIs for working with ArangoDB clusters. For more information on the design and architecture, refer to ArangoDB manual.

3.23.1 Coordinators

To connect to multiple ArangoDB coordinators, you must provide either a list of host strings or a comma-separated string during client initialization.

Example:

```
from aioarangodb import ArangoClient

# Single host
client = ArangoClient(hosts='http://localhost:8529')

# Multiple hosts (option 1: list)
client = ArangoClient(hosts=['http://host1:8529', 'http://host2:8529'])

# Multiple hosts (option 2: comma-separated string)
client = ArangoClient(hosts='http://host1:8529,http://host2:8529')
```

By default, a **'aiohttp.ClientSession'_** instance is created per coordinator. HTTP requests to a host are sent using only its corresponding session. For more information on how to override this behaviour, see http.

3.23.2 Load-Balancing Strategies

There are two load-balancing strategies available: "roundrobin" and "random" (defaults to "roundrobin" if unspecified).

Example:

```
from aioarangodb import ArangoClient

hosts = ['http://host1:8529', 'http://host2:8529']

# Round-robin
client = ArangoClient(hosts=hosts, host_resolver='roundrobin')

# Random
client = ArangoClient(hosts=hosts, host_resolver='random')
```

3.23.3 Administration

Below is an example on how to manage clusters using aioarangodb.

```
from aioarangodb import ArangoClient

# Initialize the ArangoDB client.
client = ArangoClient()

# Connect to "_system" database as root user.
```

(continues on next page)

(continued from previous page)

```
sys_db = await client.db('_system', username='root', password='passwd')

# Get the Cluster API wrapper.
cluster = sys_db.cluster

# Get this server's ID.
await cluster.server_id()

# Get this server's role.
await cluster.server_role()

# Get the cluster health.
await cluster.health()

# Get statistics for a specific server.
server_id = await cluster.server_id()
await cluster.statistics(server_id)

# Toggle maintenance mode (allowed values are "on" and "off").
await cluster.toggle_maintenance_mode('on')
await cluster.toggle_maintenance_mode('off')
```

See *ArangoClient* and *Cluster* for API specification.

3.24 JSON Serialization

You can provide your own JSON serializer and deserializer during client initialization. They must be callables that take a single argument.

Example:

```
import json

from aioarangodb import ArangoClient

# Initialize the ArangoDB client with custom serializer and deserializer.
client = ArangoClient(
    hosts='http://localhost:8529',
    serializer=json.dumps,
    deserializer=json.loads
)
```

See *ArangoClient* for API specification.

3.25 Error Codes

Python-arango provides ArangoDB error code constants for convenience.

Example

```
from aioarangodb import errno

# Some examples
```

(continues on next page)

(continued from previous page)

```
assert errno.NOT_IMPLEMENTED == 9
assert errno.DOCUMENT_REV_BAD == 1239
assert errno.DOCUMENT_NOT_FOUND == 1202
```

For more information, refer to ArangoDB manual.

3.26 Contributing

3.26.1 Requirements

Before submitting a pull request on GitHub, please make sure you meet the following requirements:

- The pull request points to dev branch.
- Changes are squashed into a single commit. I like to use git rebase for this.
- Commit message is in present tense. For example, "Fix bug" is good while "Fixed bug" is not.
- Sphinx-compatible docstrings.
- PEP8 compliance.
- No missing docstrings or commented-out lines.
- Test coverage remains at %100. If a piece of code is trivial and does not need unit tests, use this to exclude it from coverage.
- No build failures on Travis CI. Builds automatically trigger on pull request submissions.
- Documentation is kept up-to-date with the new changes (see below).

Warning: The dev branch is occasionally rebased, and its commit history may be overwritten in the process. Before you begin your feature work, git fetch or pull to ensure that your local branch has not diverged. If you see git conflicts and want to start with a clean slate, run the following commands:

```
~$ git checkout dev
~$ git fetch origin
~$ git reset --hard origin/dev  # THIS WILL WIPE ALL LOCAL CHANGES
```

3.26.2 Style

To ensure PEP8 compliance, run flake8:

```
~$ pip install flake8
~$ git clone https://github.com/bloodbare/aioarangodb.git
~$ cd aioarangodb
~$ flake8
```

If there is a good reason to ignore a warning, see here on how to exclude it.

3.26.3 Testing

To test your changes, you can run the integration test suite that comes with **aioarangodb**. It uses pytest and requires an actual ArangoDB instance.

To run the test suite (use your own host, port and root password):

```
~$ pip install pytest
~$ git clone https://github.com/bloodbare/aioarangodb.git
~$ cd aioarangodb
~$ py.test --complete --host=127.0.0.1 --port=8529 --passwd=passwd
```

To run the test suite with coverage report:

```
~$ pip install coverage pytest pytest-cov
~$ git clone https://github.com/bloodbare/aioarangodb.git
~$ cd aioarangodb
~$ py.test --complete --host=127.0.0.1 --port=8529 --passwd=passwd --cov=kq
```

As the test suite creates real databases and jobs, it should only be run in development environments.

3.26.4 Documentation

The documentation including the README is written in reStructuredText and uses Sphinx. To build an HTML version on your local machine:

```
~$ pip install sphinx sphinx_rtd_theme
~$ git clone https://github.com/bloodbare/aioarangodb.git
~$ cd aioarangodb/docs
~$ sphinx-build . build  # Open build/index.html in a browser
```

As always, thank you for your contribution!

3.27 API Specification

This page contains the specification for all classes and methods available in aioarangodb.

3.27.1 ArangoClient

3.27.2 AsyncDatabase

3.27.3 AsyncJob

3.27.4 AQL

3.27.5 AQLQueryCache

3.27.6 BatchDatabase

3.27.7 BatchJob

3.27.8 Cluster

3.27.9 Cursor

3.27.10 DefaultHTTPClient

3.27.11 StandardCollection

3.27.12 StandardDatabase

3.27.13 EdgeCollection

3.27.14 Foxx

3.27.15 Graph

3.27.16 HTTPClient

3.27.17 Pregel

3.27.18 Request

3.27.19 Response

3.27.20 Replication

3.27.21 TransactionDatabase

3.27.22 VertexCollection

3.27.23 WAL