AIO Consul Documentation

Release 0.7.0

AIO Consul

November 03, 2016

Contents

AIOConsul is a Python >= 3.5 library for requesting Consul API, build on top of asyncio and aiohttp.

Currently, this library aims a full compatibility with consul 0.7.

Sources are available at https://lab.errorist.xyz/aio/aioconsul.

CHAPTER 1

Build status

CHAPTER 2

Installation

```
pip install aioconsul
```

CHAPTER 3

Tutorial

In this example I will show you how to join my cluster with another:

```
from aioconsul import Consul
client = Consul('my.node.ip')

# do I have a members?
members = await client.members.items()
assert len(members) == 1, "I am alone in my cluster"

# let's join another cluster
joined = await client.members.join('other.node.ip')
if joined:
    members = await client.members.items()
    assert len(members) > 1, "I'm not alone anymore"
```

And display the catalog:

```
datacenters = await client.catalog.datacenters()
for dc in datacenters:
    print(dc)

services, _ = await client.catalog.services()
for service, tags in services.items():
    print(service, tags)

nodes, _ = await client.catalog.nodes()
for node in nodes:
    print(node.name, node.address)
```

CHAPTER 4

Important

Version 0.7 breaks compatibility with previous versions:

- It is closer to what HTTP API returns
- It does not add consul property anymore
- Response with metadata are now a 2 items length tuple (`CollectionMeta` or `ObjectMeta`)

CHAPTER 5

Focus

5.1 Client reference

This is a high-level client for Consul HTTP API, intended to have less friction between Python idiom and Consul API. API endpoints are organised by Python endpoints, each reachable by it's client property.

If needed, you can use the `API` directly.

5.1.1 Common exceptions

5.2 Raw API

This is a low-level wrapper of Consul HTTP API, intended to be the most simple.

5.3 Client Endpoints

5.3.1 Common parameters

dc By default, the datacenter of the agent is queried ; however, the dc can be provided.

watch Performs a blocking query. Value can be a `ObjectIndex`, or a `Tuple[ObjectIndex, timedelta]`.

consistency Change client behavior about consistency.

session A session Object.

index An index Object.

token A token Object.

Most of endpoints that returns ConsulValue supports blocking queries and consistency modes.

5.3.2 ACL

Create, update, destroy, and query ACL tokens.

Most of the operations relies on ACL token with sufficant rights. Here is the way aioconsul manage tokens:

```
token = {"Name": "my-app-token"}
token_id = await client.acl.create(token)
token.update(token_id)
token.update({"Rules": """
    key "" {
        policy = "read"
    }
    key "private/" {
        policy = "deny"
    }
"""})
token_id = await client.acl.update(token)

token, meta = await client.acl.info(token_id)
clone_id, meta = await client.acl.clone(token_id)

destroyed = await client.acl.destroy(token_id)

tokens, meta = await client.acl.items()
info = await client.acl.replication()
```

5.3.3 Agent

Interact with the local Consul agent.

Example:

```
obj = await client.agent.info()
disabled = await client.agent.disable(reason="migrating")
enabled = await client.agent.enable(reason="migration done")
```

5.3.4 Agent's Checks

Manage local checks.

Example:

```
check = {
    "ID": "mem",
    "Name": "Memory utilization",
    "TTL": "15s"
    "Interval": "10s"
}
registered = await client.checks.register(check)
mapping = await client.checks.items()
marked = await client.checks.critical(check, note="Fatal")
marked = await client.checks.warning(check, note="Warning")
marked = await client.checks.passing(check, note="Back to normal")
deregistered = await client.checks.deregister(check)
```

5.3.5 Agent's Members

Manage local serf agent cluster

Example:

```
joined = await client.checks.join("10.11.12.13", wan=True)
members = await client.members.items()
leaving = await client.checks.force_leave("my-node")
```

5.3.6 Agent's Services

Manage local services.

Example:

```
service = {
    "ID": "redis1",
    "Name": "redis",
    "Tags": [
        "master",
        "v1"
    ],
}
registered = await client.services.register(service)
services = await client.services.items()
disabled = await client.services.disable(service, reason="migrating")
enabled = await client.services.enable(service, reason="migration done")
deregistered = await client.services.deregister(service)
```

5.3.7 Catalog

Manage catalog.

Example:

```
definitions = {
  "Node": "foobar",
  "Address": "192.168.10.10",
  "Service": {
    "ID": "redis1",
    "Service": "redis",
    "Tags": [ "master", "v1" ],
    "Address": "127.0.0.1",
    "Port": 8000
  },
  "Check": {
    "Node": "foobar",
    "CheckID": "service:redis1",
    "Name": "Redis health check",
    "Notes": "Script based health check",
    "Status": "passing",
    "ServiceID": "redis1"
  }
}

registered = await client.catalog.register(definitions)
datacenters = await client.catalog.datacenters()
nodes, meta = await client.catalog.nodes(near="_self")
services, meta = await client.catalog.node("foobar", watch=(meta, "30s"))
services, meta = await client.catalog.services(consistency="stale")
nodes, meta = await client.catalog.service("redis1", tag="prod")
```

```
deregistered = await client.catalog.deregister({
    "Node": "foobar",
    "Address": "192.168.10.10"
})
```

5.3.8 Events

Manage events.

Example:

```
id = await client.event.fire("my-event", service="my-service")
collection, meta = await client.event.items("my-event")
```

5.3.9 Health Checks

Consult health.

Example:

```
collection, meta = await client.health.node("my-node")
collection, meta = await client.health.checks("my-service")
collection, meta = await client.health.state("passing", near="_self")
```

5.3.10 Key/Value Store

Manage kv store.

Common operations example:

```
keys, meta = await client.kv.keys("my/key", separator="/")
setted = await client.kv.set("my/key", b"my value")
obj, meta = await client.kv.get("my/key")
deleted = await client.kv.delete("my/key")
```

Tree operations example:

```
collection, meta = await client.kv.get_tree("my/key", separator="/")
deleted = await client.kv.delete_tree("my/key", separator="/")
```

CAS operations example:

```
setted = await client.kv.cas("my/key", b"my value", index=meta)
deleted = await client.kv.delete_cas("my/key", index=meta)
```

Locked operations example:

```
locked = await client.kv.lock("my/key", b"my value", session=session_id)
unlocked = await client.kv.unlock("my/key", b"my value", session=session_id)
```

5.3.11 Key/Value Transactions

These same operations can be done in a transactional way:

```
results, meta = await client.kv.prepare()\
                        .set("my/key", b"my value")\
                        .get("my/key")\
                        .execute()
```

5.3.12 Network Coordinates

Consult network coordinates.

Example:

```
datacenters = await client.coordinate.datacenters()
collection, meta = await client.coordinate.nodes()
```

5.3.13 Operator

Manage raft.

Example:

```
obj = await client.operator.configuration()
obj = await client.operator.peer_delete("10.11.12.13")
```

5.3.14 Prepared Queries

Manage prepared queries.

Example:

```
collection = await client.query.items()
obj = await client.query.create({""})
obj = await client.query.read({""})
obj = await client.query.explain({""})
updated = await client.query.update({""})
deleted = await client.query.delete({""})
results = await client.query.execute({""})
```

5.3.15 Session

Manage sessions.

Example:

```
obj = await client.session.create({""})
destroyed = await client.session.destroy({""})
obj, meta = await client.session.info({""})
obj, meta = await client.session.renew({""})
collection, meta = await client.session.node("my-node")
collection, meta = await client.session.items()
```

5.3.16 Status

Consult status.

Example:

```
addr = await client.status.leader()
addrs = await client.status.peers()
```

5.4 Misc.

5.4.1 Typing

Most of call are polymorphic and most of the types in documentation are just hints. Here is the full list of types:

class `aioconsul.typing.`**`ObjectIndex`**
Used by polymorphic functions that accept any `str` or `dict` that have an **Index** key.

For example these two objects are equivalent:

```
a = { "Index": "1234-43-6789" }
b = "1234-43-6789"
```

class `aioconsul.typing.`**`ObjectID`**
Used by polymorphic functions that accept any `str` or `dict` that have an **ID** key.

These two objects are equivalent:

```
a = { "ID": "1234-43-6789" }
b = "1234-43-6789"
```

class `aioconsul.typing.`**`Object`**
Represents a `dict` where keys are `str` and values can be any type.

For example:

```
{
  "ID": "1234-43-6789",
  "Name": "foo",
  "Value": "bar"
}
```

Depending of the endpoint, **ID** may be required or not.

class `aioconsul.typing.`**`Collection`**
Represents a `list` where values are `Object`.

For example:

```
[{"ID": "1234"}, {"ID": "5678"}]
```

class `aioconsul.typing.`**`Mapping`**
Represents a `dict` where keys are `id` and values are `Object`.

For example:

```
{"service1": {"ID": "service1"}}
```

class `aioconsul.typing.`**`Meta`**
Represents a `dict` that is associated to Response. These keys should be presents:

- **Index** — a unique identifier representing the current state of the requested resource
- **KnownLeader** — indicates if there is a known leader
- **LastContact** — Contains the time in milliseconds that a server was last contacted by the leader node

Meta can be used for blocking / watch mode.

class aioconsul.typing.**CollectionMeta**
A tuple where first value is a `Collection` and second value is `Meta`.

class aioconsul.typing.**ObjectMeta**
A tuple where first value is an `Object` and second value is `Meta`.

class aioconsul.typing.**Filter**
Regular expression to filter by.

It accepts a `re.Pattern` or a `str`. These two objects are equivalent:

```
a = re.compile("node-\d+")
b = "node-\d+"
```

class aioconsul.typing.**Consistency**
Most of the read query endpoints support multiple levels of consistency. Since no policy will suit all clients' needs, these consistency modes allow the user to have the ultimate say in how to balance the trade-offs inherent in a distributed system. The three read modes are:

default — If not specified, the default is strongly consistent in almost all cases. However, there is a small window in which a new leader may be elected during which the old leader may service stale values. The trade-off is fast reads but potentially stale values. The condition resulting in stale reads is hard to trigger, and most clients should not need to worry about this case. Also, note that this race condition only applies to reads, not writes.

consistent — This mode is strongly consistent without caveats. It requires that a leader verify with a quorum of peers that it is still leader. This introduces an additional round-trip to all server nodes. The trade-off is increased latency due to an extra round trip. Most clients should not use this unless they cannot tolerate a stale read.

stale — This mode allows any server to service the read regardless of whether it is the leader. This means reads can be arbitrarily stale; however, results are generally consistent to within 50 milliseconds of the leader. The trade-off is very fast and scalable reads with a higher likelihood of stale values. Since this mode allows reads without a leader, a cluster that is unavailable will still be able to respond to queries.

class aioconsul.typing.**Duration**
Defines a duration. Can be specified in the form of "10s" or "5m" or a `datetime.timedelta`.

For example, these objects are equivalent:

```
timedelta(seconds=150)
"2m30s"
```

class aioconsul.typing.**Blocking**
Defines a blocking query.

It must be a *ObjectIndex* or better a tuple where second value is a *Duration*.

For example these values are equivalent:

```
a = {"Index": 1}
b = 1
c = ({"Index": 1}, None)
```

For adding a wait, just set the second value of tuple.

class `aioconsul.typing.`**`Payload`**

Currently only `bytes` and `bytearray` are allowed. It may vary on **Flags** value in the futur.

Internally, Payload will be base64 encoded/decoded when the endpoint requires it. End user does not have to do it.

The `KVEndpoint` is build on top of:

- `PUT /v1/kv/(key)` - **Body** will be encoded
- `GET /v1/kv/(key)` - **Value** will be decoded

For example:

```
PAYLOAD = b"bar"
await client.kv.set("foo", PAYLOAD)
response, _ = await client.kv.get("foo")
assert response["Value"] == PAYLOAD
```

The `EventEndpoint` is build on top of:

- `PUT /v1/event/fire/(name)` - **Payload** will be kept as is
- `GET /v1/event/list` - **Payload** will be base64 encoded

For example:

```
PAYLOAD = b"qux"
response = await client.event.fire("baz", PAYLOAD)
assert response["Payload"] == PAYLOAD
responses, _ = await client.kv.items("baz")
assert responses[0]["Payload"] == PAYLOAD
```

5.5 Contributing

The main repository is hosted on https://lab.errorist.xyz/aio/aioconsul and is mirrored into Github https://github.com/johnnoone/aioconsul.

We'd love for you to contribute to our source code and to make AIOConsul even better than it is today! Here are the guidelines we'd like you to follow:

5.5.1 Got a Question or Problem?

If you have questions about how to use AIOConsul, please send directly an email to main commiters of repository.

5.5.2 Found an Issue?

If you find a bug in the source code or a mistake in the documentation, you can help us by submitting an issue to our GitHub Repository. Even better you can submit a Pull Request with a fix.

5.5.3 Want a Feature?

You can request a new feature by submitting an issue to our GitHub Repository. If you would like to implement a new feature then consider what kind of change it is:

- **Major Changes** that you wish to contribute to the project should be discussed first so that we can better coordinate our efforts, prevent duplication of work, and help you to craft the change so that it is successfully accepted into the project.
- **Small** Changes can be crafted and submitted to the GitHub Repository as a Pull Request.

Want a Doc Fix?

If you want to help improve the docs, it's a good idea to let others know what you're working on to minimize duplication of effort. Create a new issue (or comment on a related existing one) to let others know what you're working on.

For large fixes, please build and test the documentation before submitting the PR to be sure you haven't accidentally introduced any layout or formatting issues.

If you're just making a small change, don't worry about filing an issue first. Use the friendly blue "Improve this doc" button at the top right of the doc page to fork the repository in-place and make a quick change on the fly.

Coding Rules

To ensure consistency throughout the source code, keep these rules in mind as you are working:

- All features or bug fixes must be tested.
- All public API methods must be documented using RestructuredText in conjonction of Napoleon.
- All code msut be linted with flake8

A