
medallion Documentation

Release 3.0.0

OASIS Open

Jun 06, 2023

Contents:

1	Compatibility	3
2	Custom Backends and Users	5
2.1	How to create your custom Backend	5
2.2	How to load your custom Backend	5
2.3	How to use a different authentication library	6
2.4	How to use a different backend to control users	7
3	Design of the TAXII Server Mongo DB Schema for <i>medallion</i>	9
3.1	The discovery database	9
3.2	The api root databases	10
4	Contributing	13
4.1	Setting up a development environment	13
4.2	Code style	14
4.3	Testing	14
4.4	Adding a dependency	14
5	Indices and tables	15

Medallion is a minimal implementation of a TAXII 2.0 Server in Python.

CHAPTER 1

Compatibility

Medallion does NOT support the following features of the [TAXII 2.1 specification](#):

- Content-Types other than the default
- Pending Add Objects responses (all Add Objects request process all objects before sending back a response)
- *Medallion* uses HTTP only (not HTTPS). It can be run using a WSGI server (such as Gunicorn or uWSGI) behind a production server such as Apache or NGINX that acts as a reverse proxy for *medallion*.

Its main purpose is for use in testing scenarios of STIX-based applications that use the [python-stix2 API](#). It has been developed in conjunction with [cti-taxii-client](#) but should be compatible with any TAXII client which makes HTTP requests as defined in TAXII 2.1 specification.

Custom Backends and Users

2.1 How to create your custom Backend

To create a custom Backend compatible with medallion you should subclass `medallion.backends.Backend`. This object provides the basic skeleton used to handle each of the endpoint requests. For further examples of on how to build a custom backend look under the `medallion/backends/` directory.

2.2 How to load your custom Backend

Backends are loaded based on the content of the `backend` map element in the `medallion` configuration file. Built-in backend implementations can be loaded by simply setting the `module_class` key to the subclass' names. Extra keyword arguments can be passed to the backend implementation by including them in the `backend` map element. A simple example of loading the built-in `MemoryBackend` and passing it a keyword argument looks like:

```
{
  "backend": {
    "module_class": "MemoryBackend",
    "filename": "../test/data/default_data.json"
  }
}
```

Loading custom backends can be a little more complicated depending on how the backend class is implemented. If the custom backend subclasses the base `Backend` and is somehow imported prior to starting the `medallion` flask application, the configuration file may simply refer to it by name in the `module_class` key. This might be useful in environments where it is preferable to use something like the Python `site` module rather than installing an extra package.

```
{
  "backend": {
    "module_class": "MyCustomBackend",
```

(continues on next page)

(continued from previous page)

```
}  
}
```

To make loading out-of-tree backend implementations easier, the `medallion.backends` entrypoint is defined which may be used by other packages to point to external modules or classes which should be loaded by the backend machinery. This should be defined in your package's `setup.py` like:

```
setup(  
    # ...  
    entry_points={  
        "medallion.backends": [  
            "MyEPName = mypackage.with_backends:MyCustomBackend",  
        ],  
    },  
)
```

The entrypoint will be loaded and if it refers to a class object it will be registered with the base `Backend` class using the entrypoint's name. If the entrypoint is a subclass of the base `Backend`, it will also be registered using the name of the loaded class object, supporting the use of implementation which do not subclass the base `Backend` properly. The entrypoint might also validly point to a module to be loaded, in which case any backend implementations must be subclasses of the base `Backend` and they will be registered under their class names.

A previous implementation allowed a dotted module path to be specified in the `module` key of the backend map in the configuration. This behaviour is deprecated but will continue to work with a warning. This is done to allow implementations to pivot to using the entry-point mechanism instead. An example configuration snippet for this approach looks like:

```
{  
    "backend": {  
        "module": "mypackage.with_backends",  
        "module_class": "MyCustomBackend",  
    }  
}
```

Another way to provide a custom backend using flask proxy could be:

```
import MyCustomBackend  
from flask import current_app  
from medallion import application_instance, set_config  
  
MyCustomBackend.init() # Do some setup before attaching to application... (Imagine_  
↳ other steps happening here)  
  
with application_instance.app_context():  
    current_app.medallion_backend = MyCustomBackend  
  
# Do some other stuff...  
set_config(application_instance, {...})  
application_instance.run()
```

2.3 How to use a different authentication library

If you need or prefer a library different from `Flask-HTTPAuth`, you can override it by modifying the `auth` global to your preference. Now, if you want to keep changes at a minimum throughout the library. You can wrap the behavior

inside another class, but remember all changes need to be performed before the call to `run()`. For example,

```
from flask import current_app
from medallion import application_instance, auth, set_config, init_backend

# This is a dummy implementation of Flask Auth that always returns false
dummy_auth = class DummyAuth(object):

    def login_required(self, f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            return f(*args, **kwargs)
        return decorated_function

    def get_password():
        return None # Custom stuff to get password using other libraries, users_
        ↪config can go here.

# Set the default implementation to the dummy auth
auth = dummy_auth()

set_config(application_instance, {...})
init_backend(application_instance, {...})
application_instance.run()
```

2.4 How to use a different backend to control users

Our implementation of a users authentication system is not suitable for a production environment. Thus requiring to write custom code to handle credential authentication, sessions, etc. Most likely you will require the changes described in the section above on *How to use a different authentication library*, plus changing the `users_config`.

```
import MyCustomDBforUsers
from flask import current_app
from medallion import application_instance, set_config

# This is a dummy implementation of Flask Auth that always returns false
dummy_auth = class DummyAuth(object):

    def login_required(self, f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            return f(*args, **kwargs)
        return decorated_function

    def get_password():
        # Usage of MyCustomDBforUsers would likely happen here.
        return something # Custom stuff to get password using other libraries, users_
        ↪config functionality.

# Set the default implementation to the dummy auth
auth = dummy_auth()

db = MyCustomDBforUsers.init() # Do some setup before attaching to application...
        ↪(Imagine other steps happening here)
```

(continues on next page)

(continued from previous page)

```
with application_instance.app_context():
    current_app.users_config = db # This will make it available inside the Flask_
    ↪instance in case you decide to perform changes to the internal blueprints.

init_backend(application_instance, {...})
application_instance.run()
```

Design of the TAXII Server Mongo DB Schema for *medallion*

As *medallion* is a prototype TAXII server implementation, the schema design for a Mongo DB is relatively straightforward.

Each Mongo database contains one or more collections. The term “collection” in Mongo DBs is similar to the concept of a table in a relational database. Collections contain “documents”, similar to records.

It is unfortunate that the term “collection” is also used to signify something unrelated in the TAXII specification. We will use the phrase “taxii collection” to distinguish them.

An instance of this schema can be populated via the file `test/data/initialize_mongodb.py`. This instance will be used for examples below.

Utilities to initialize your own Mongo DB can be found in `test/generic_initialize_mongodb.py`.

3.1 The discovery database

Basic metadata contained in the mongo database named **discovery_database**.

The `discovery_database` contains two collections:

discovery_information. It should only contain only one “document”, which is the discovery information that would be returned from the Discovery endpoint. Here is the document from the example database.

```
{
  "title": "Some TAXII Server",
  "description": "This TAXII Server contains a listing of",
  "contact": "string containing contact information",
  "default": "http://localhost:5000/api2/",
  "api_roots": [
    "http://localhost:5000/api1/",
    "http://localhost:5000/api2/",
    "http://localhost:5000/trustgroup1/"
  ]
}
```

api_root_info contains documents that describe each **api_root**. Because the “_url” and “_name” properties are not part of the TAXII specification, they will be stripped by *medallion* before any document is returned to the client.

Here is a document from the example database:

```
{
  "title": "Malware Research Group",
  "description": "A trust group setup for malware researchers",
  "versions": [
    "taxii-2.0"
  ],
  "max_content_length": 9765625,
  "_url": "http://localhost:5000/trustgroup1/",
  "_name": "trustgroup1"
}
```

3.2 The api root databases

Each api root is contained in a separate Mongo DB database. It has four collections: **status**, **objects**, **manifests**, and **collections**. To support multiple taxii collections, any document in the **objects** and **manifests** contains an extra property, “collection_id”, to link it to the taxii collection that it is contained in. Because “_collection_id” property is not part of the TAXII specification, it will be stripped by *medallion* before any document is returned to the client.

A document from the **collections** collection:

```
{
  "id": "91a7b528-80eb-42ed-a74d-c6fbd5a26116",
  "title": "High Value Indicator Collection",
  "description": "This data collection is for collecting high value IOCs",
  "can_read": true,
  "can_write": true,
  "media_types": [
    "application/vnd.oasis.stix+json; version=2.0"
  ]
}
```

A document from the **objects** collection:

```
{
  "created": "2014-05-08T09:00:00.000Z",
  "id": "indicator--a932fcc6-e032-176c-126f-cb970a5a1ade",
  "labels": [
    "file-hash-watchlist"
  ],
  "modified": "2014-05-08T09:00:00.000Z",
  "name": "File hash for Poison Ivy variant",
  "pattern": "[file:hashes.'SHA-256' =
  ↪ 'ef537f25c895bfa782526529a9b63d97aa631564d5d789c2b765448c8635fb6c']",
  "type": "indicator",
  "valid_from": "2014-05-08T09:00:00.000000Z",
  "_collection_id": "91a7b528-80eb-42ed-a74d-c6fbd5a26116"
}
```

A document from the **status** collection:

```
{
  "id": "2d086da7-4bdc-4f91-900e-d77486753710",
  "status": "pending",
  "request_timestamp": "2016-11-02T12:34:34.12345Z",
  "total_count": 4,
  "success_count": 1,
  "successes": [
    "indicator--a932fcc6-e032-176c-126f-cb970a5alade"
  ],
  "failure_count": 1,
  "failures": [
    {
      "id": "malware--664fa29d-bf65-4f28-a667-bdb76f29ec98",
      "message": "Unable to process object"
    }
  ],
  "pending_count": 2,
  "pendings": [
    "indicator--252c7c11-daf2-42bd-843b-be65edca9f61",
    "relationship--045585ad-a22f-4333-af33-bfd503a683b5"
  ]
}
```

A document from the **manifest** collection:

```
{
  "id": "indicator--a932fcc6-e032-176c-126f-cb970a5alade",
  "date_added": "2016-11-01T10:29:05Z",
  "versions": [
    "2014-05-08T09:00:00.000Z"
  ],
  "media_types": [
    "application/vnd.oasis.stix+json; version=2.0"
  ],
  "_collection_id": "91a7b528-80eb-42ed-a74d-c6fbd5a26116"
}
```


We're thrilled that you're interested in contributing to medallion! Here are some things you should know:

- contribution-guide.org has great ideas for contributing to any open-source project (not just this one).
- All contributors must sign a Contributor License Agreement. See [CONTRIBUTING.md](#) in the project repository for specifics.
- If you are planning to implement a major feature (vs. fixing a bug), please discuss with a project maintainer first to ensure you aren't duplicating the work of someone else, and that the feature is likely to be accepted.

Now, let's get started!

4.1 Setting up a development environment

We recommend using a [virtualenv](#).

1. Clone the repository. If you're planning to make pull request, you should fork the repository on GitHub and clone your fork instead of the main repo:

```
$ git clone https://github.com/yourusername/cti-taxii-server.git
```

2. Install development-related dependencies:

```
$ cd cti-taxii-server
$ pip install -r requirements.txt
```

3. Install [pre-commit](#) git hooks:

```
$ pre-commit install
```

At this point you should be able to make changes to the code.

4.2 Code style

All code should follow [PEP 8](#). We allow for line lengths up to 160 characters, but any lines over 80 characters should be the exception rather than the rule. PEP 8 conformance will be tested automatically by Tox and Travis-CI (see below).

4.3 Testing

Note: All of the tools mentioned in this section are installed when you run `pip install -r requirements.txt`.

This project uses [pytest](#) for testing. We encourage the use of test-driven development (TDD), where you write (failing) tests that demonstrate a bug or proposed new feature before writing code that fixes the bug or implements the features. Any code contributions should come with new or updated tests.

To run the tests in your current Python environment, use the `pytest` command from the root project directory:

```
$ pytest
```

This should show all of the tests that ran, along with their status.

You can run a specific test file by passing it on the command line:

```
$ pytest medallion/test/test_<xxx>.py
```

To ensure that the test you wrote is running, you can deliberately add an `assert False` statement at the beginning of the test. This is another benefit of TDD, since you should be able to see the test failing (and ensure it's being run) before making it pass.

[tox](#) allows you to test a package across multiple versions of Python. Setting up multiple Python environments is beyond the scope of this guide, but feel free to ask for help setting them up. Tox should be run from the root directory of the project:

```
$ tox
```

We aim for high test coverage, using the [coverage.py](#) library. Though it's not an absolute requirement to maintain 100% coverage, all code contributions must be accompanied by tests. To run coverage and look for untested lines of code, run:

```
$ pytest --cov=medallion
$ coverage html
```

then look at the resulting report in `htmlcov/index.html`.

All commits pushed to the `master` branch or submitted as a pull request are tested with [Travis-CI](#) automatically.

4.4 Adding a dependency

One of the pre-commit hooks we use in our development environment enforces a consistent ordering to imports. If you need to add a new library as a dependency please add it to the `known_third_party` section of `.isort.cfg` to make sure the import is sorted correctly.

CHAPTER 5

Indices and tables

- `genindex`
- `modindex`
- `search`