Catcher-Modules Documentation

Valerii Tikhonov

Jan 03, 2022

Contents:

External Catcher modules repository.
Besides the built-in Catcher support external modules: as python or any other executable scripts.
See Catcher documentation on installation catcher with catcher-modules.

CHAPTER 1

Usage

You can either write your own module in python or as external shell script. Both ways are covered in Catcher documentation for external modules.

Read the docs for existing modules usage info: `catcher_modules()`

CHAPTER 2

Contribution

If you believe your external python module can be useful for other people you can create a pull request here. You can find quick support in the telegram channel.

CHAPTER 3

Contributors:

- Many thanks to Ekaterina Belova for core & modules contribution.

CHAPTER 4

Additional dependencies

libclntsh.dylib is required for *oracle*. Read more here.
unixodbc-dev and driver are required for *mssql*
Webdriver is required for selenium
Webdriver bundle for your language (except python) is required for selenium.

4.1 catcher_modules.cache package

4.1.1 Submodules

4.1.2 catcher_modules.cache.redis module

class catcher_modules.cache.redis.**Redis**(***kwargs*)
Bases: catcher.steps.external_step.ExternalStep

Work with Redis cache. Put value to cache or get it, increment/decrement or delete.

Input

Conf redis configuration. Is an object.

- host: redis host. Default is localhost
- port: redis port. Default is 6379
- db: redis database number. Default is 0

<command>

- command to run. Every command can have a list of arguments.

Refer to Redis and Redis-Py

Examples

Set value (default configuration)

```
variables:
        complex:
            a: 1
            b: 'c'
            d: [1,2,4]

redis:
    request:
        set:
            key: '{{ complex }}'
```

Get value by key 'key' and register in variable 'var'

```
redis:
    request:
        get: 'key'
    register: {var: '{{ OUTPUT }}'}
```

Decrement, increment by 5 and delete

```
redis:
    actions:
        - request:
                set:
                    'foo': 11
        - request:
                decr: foo
        - request:
                incrby:
                    foo: 5
        - request:
                delete:
                    - foo
```

4.2 catcher_modules.database package

4.2.1 Submodules

4.2.2 catcher_modules.database.couchbase module

class catcher_modules.database.couchbase.**Couchbase**(***kwargs*)

Allows you to perform put/get/delete/query operations in Couchbase

Input

Conf couchbase configuration. Is an object. **Required**.

- bucket: bucket to work with
- user: database user (optional)
- host: database host (optional)
- password: user's password

Put put value in the database by the key.

Get get object by key.

Delete delete object by key.

Query query to run.

Examples

Put value by key

```
couchbase:
    request:
        conf:
            bucket: test
            host: localhost
        put:
            key: my_key
            value: {foo: bar, baz: [1,2,3,4]}
```

Get value by key

```
couchbase:
    request:
        conf:
            bucket: test
            user: test
            password: test
            host: localhost
        get:
            key: my_key
```

Delete value by key

```
couchbase:
    request:
        conf:
            bucket: test
            user: test
            password: test
            host: localhost
        delete:
            key: my_key
```

Query by foo

```
couchbase:
    request:
        conf:
            bucket: test
            user: test
            password: test
            host: localhost
        query: "select `baz` from test where `foo` = 'bar'"
```

4.2.3 catcher_modules.database.postgres module

class catcher_modules.database.postgres.**Postgres**(***kwargs*)
Allows you to run queries in Postgres

Input

Conf postgres configuration. Can be a single line string or object. Dialect is not mandatory. **Required**.

- dbname: name of the database to connect to
- user: database user
- host: database host
- password: user's password
- port: database port

Query query to run. **Deprecated since 5.2**

Sql query or sql file from resources to run. **Required**

Examples

Select all from test, use object configuration

```
postgres:
  request:
      conf:
          dbname: test
          user: user
          password: password
          host: localhost
          port: 5433
      sql: 'select count(*) from test'
  register: {documents: '{{ OUTPUT }}'}
```

Run all commands from **resources/my_ddl.sql**, using string configuration

```
postgres:
  request:
      conf: 'user:password@localhost:5432/test'
      sql: 'my_ddl.sql'
```

Insert into test, using string configuration with dialect

```
postgres:
  request:
      conf: 'postgresql://user:password@localhost:5432/test'
      sql: 'insert into test(id, num) values(3, 3);'
```

4.2.4 catcher_modules.database.mongo module

class catcher_modules.database.mongo.**Mongo**(***kwargs*)
Allows you to interact with MongoDB NoSQL database. :Input:

Conf mongodb configuration. Can be a single line, object or object with url as a parameter (for Airflow connection)

string url or kv object. **Required**.

- database: name of the database to connect to
- username: database user. Must be RFC 2396 encoded when in URI.
- host: database host
- password: user's password. Must be RFC 2396 encoded when in URI.
- port: database port
- authSource: The database to authenticate on. Default is database.

See pymongo for more options. :collection: collection to use. **Required**

Command String. Use this if you have to run command without any parameters. Where command's value is your command to run, like *command: find_one*. *Optional*

<command> Object. Use this when you have command with parameters. Where <command> key is your command name and it's value is parameter object (list or dict). *Optional* Either <command> or command should exist.

Next Run other operation just after your operation. Can be string like *next: count* or object with params *next: {'sort': 'author'}. You can chain multiple next (see example). 'Optional*

List_params Pass command params as different arguments. Useful when pymongo command takes several arguments (both **args* and ***kwargs*). **args* will be set in case of params in list while ***kwargs* will be sent in case of dict. See examples for more info.

Examples

Find one document. Use **command** key when no params.

```
mongo:
  request:
      conf:
          database: test
          username: test
          password: test
          host: localhost
          port: 27017
      collection: 'your_collection'
      command: 'find_one'
  register: {document: '{{ OUTPUT }}'}
```

Use object configuration with extra fields (for Airflow connection). This step will ignore everything except url. Inventory.yaml

```
mongo_conf:
    url: 'mongodb://username:password@host'
    type: 'mongo'
    extra: '{"key":"value"}'
```

mongo step itself

```
mongo:
    request:
        conf: '{{ mongo_conf }}'
```

(continues on next page)

(continued from previous page)

```
        collection: 'your_collection'
        command: 'find_one'
```

See more info about connections population in Catcher-Airflow **'docs https://catcher-modules.readthedocs.io/en/latest/source/airflow.html'_**

Alternatively you can use `conf: '{{ mongo_conf.url }}'`.

Insert into test, using string configuration

```
mongo:
  request:
      conf: 'mongodb://username:password@host'
      collection: 'your_collection'
      insert_one:
        'author': 'Mike'
        'text': 'My first blog post!'
        'tags': ['mongodb', 'python', 'pymongo']
        'date': '{{ NOW_DT }}'
```

Find specific document

```
mongo:
  request:
      conf:
          database: test
          username: test
          password: test
          host: localhost
          port: 27017
      collection: 'your_collection'
      find_one: {'author': 'Mike'}
  register: {document: '{{ OUTPUT }}'}
```

To find multiple documents just use **find** instead of **find_one**.

Bulk insert

```
mongo:
  request:
      conf: '{{ mongo_conf }}'
      collection: 'your_collection'
      insert_many:
        - {'foo': 'baz'}
        - {'foo': 'bar'}
```

Chaining operations: db.collection.find().sort().count()

```
mongo:
  request:
      conf:
          database: test
          username: test
          password: test
          host: localhost
          port: 27017
      collection: 'your_collection'
```

(continues on next page)

(continued from previous page)

```
      find: {'author': 'Mike'}
      next:
        sort: 'author'
        next: 'count'
  register: {document: '{{ OUTPUT }}'}
```

Will run every next operation on previous one. You can chain more than one operation.

Run operation with list parameters (***kwargs*). Is useful when calling commands with additional arguments.

```
mongo:
  request:
      conf:
          database: test
          username: test
          password: test
          host: localhost
          port: 27017
      collection: 'your_collection'
      find:
        filter: {'author': 'Mike'}
        projection: {'_id': False}
      list_params: true  # pass list arguments as separate params
  register: {document: '{{ OUTPUT }}'}
```

Run operation with list parameters (**args*). Run map-reduce.

```
mongo:
  request:
      conf:
          database: test
          username: test
          password: test
          host: localhost
          port: 27017
      collection: 'your_collection'
      map_reduce:
        - 'function () {
                this.tags.forEach(function(z) {
                    emit(z, 1);
                });
           }'
        - 'function (key, values) {
             var total = 0;
              for (var i = 0; i < values.length; i++) {
                total += values[i];
              }
              return total;
            }'
        - 'myresults'
      list_params: true  # pass list arguments as separate params
  register: {document: '{{ OUTPUT }}'}
```

4.2.5 catcher_modules.database.oracle module

class catcher_modules.database.oracle.**Oracle**(***kwargs*)

Allows you to run sql queries in OracleDB.

Input

Conf oracle configuration. Can be a single line string or object. Dialect is not mandatory. **Required**.

- dbname: name of the database to connect to
- user: database user
- host: database host
- password: user's password
- port: database port

Query query to run. **Deprecated since 5.2**

Sql query or sql file from resources to run. **Required**

Examples

Select all from test, use object configuration

```
oracle:
  request:
      conf:
          dbname: test
          user: user
          password: password
          host: localhost
          port: 1521
      sql: 'select count(*) as count from test'
  register: {documents: '{{ OUTPUT }}'}
```

Insert into test, using string configuration

```
oracle:
  request:
      conf: 'user:password@localhost:1521/test'
      sql: 'insert into test(id, num) values(3, 3);'
```

Insert into test, using string configuration with dialect

```
oracle:
  request:
      conf: 'oracle+cx_oracle://user:password@localhost:1521/test'
      sql: 'insert into test(id, num) values(3, 3);'
```

4.2.6 catcher_modules.database.sqlite module

class catcher_modules.database.sqlite.**SQLite**(***kwargs*)

Allows you to create SQLite database on your local filesystem and work with it. **Important** - for relative path use one slash /. For absolute slash - two //.

Input

Conf sqlite path string. Dialect is not mandatory. **Required**.

Query query to run. **Deprecated since 5.2**

Sql query or sql file from resources to run. **Required**

Examples

Select all from test, use relative path

```
sqlite:
  request:
      conf: '/foo.db'
      sql: 'select count(*) as count from test'
  register: {documents: '{{ OUTPUT }}'}
```

Note that we alias count. For some reason sqlalchemy for sqlite will return *count(*)* as a column name instead of *count*.

Insert into test, using string absolute path (with 2 slashes)

```
sqlite:
  request:
      conf: '//absolute/path/to/foo.db'
      sql: 'insert into test(id, num) values(3, 3);'
```

4.2.7 catcher_modules.database.mysql module

class catcher_modules.database.mysql.**MySql**(**kwargs*)

Allows you to run queries on MySQL (and all mysql compatible databases like MariaDB).

Input

Conf mysql configuration. Can be a single line string or object. Dialect is not mandatory. **Required**.

- dbname: name of the database to connect to
- user: database user
- host: database host
- password: user's password
- port: database port

Query query to run. **Deprecated since 5.2**

Sql query or sql file from resources to run. **Required**

Examples

Select all from test, use object configuration

```
mysql:
  request:
      conf:
          dbname: test
          user: user
          password: password
          host: localhost
```

(continues on next page)

(continued from previous page)

```
          port: 3306
      sql: 'select count(*) as count from test'
  register: {documents: '{{ OUTPUT }}'}
```

Note that we alias count. For some reason sqlalchemy for mysql will return *count(*)* as a column name instead of *count*.

Insert into test, using string configuration

```
mysql:
  request:
      conf: 'user:password@localhost:3306/test'
      sql: 'insert into test(id, num) values(3, 3);'
```

Insert into test, using string configuration with dialect

```
mysql:
  request:
      conf: 'mysql+pymysql://user:password@localhost:3306/test'
      sql: 'insert into test(id, num) values(3, 3);'
```

4.2.8 catcher_modules.database.mssql module

class catcher_modules.database.mssql.**MSSql**(***kwargs*)

Allows you to run queries on Microsoft SQL Server.

Input

Conf mssql configuration. Can be a single line string or object. Dialect is not mandatory. **Required**.

- dbname: name of the database to connect to
- user: database user
- host: database host
- password: user's password
- port: database port
- **driver: odbc driver name you've installed. *Optional* If not specified, the default driver, which comes with** catcher-modules Dockerfile will be used.

Query query to run. **Deprecated since 5.2**

Sql query or sql file from resources to run. **Required**

Examples

Select all from test, use object configuration

```
mssql:
  request:
      conf:
          dbname: test
          user: user
          password: password
          host: localhost
```

(continues on next page)

(continued from previous page)

```
          port: 1433
          driver: ODBC Driver 17 for SQL Server
      sql: 'select count(*) as count from test'
  register: {documents: '{{ OUTPUT }}'}
```

Note that we alias count. For some reason sqlalchemy for mssql will return *count(*)* as a column name instead of *count*.

Insert into test, using string configuration

```
mssql:
  request:
      conf: 'user:password@localhost:5432/test'
      sql: 'insert into test(id, num) values(3, 3);'
```

Insert into test, using string configuration with pymssql (pymssql should be installed)

```
mssql:
  request:
      conf: 'mssql+pymssql://user:password@localhost:5432/test'
      sql: 'insert into test(id, num) values(3, 3);'
```

4.3 catcher_modules.frontend package

4.3.1 Submodules

4.3.2 catcher_modules.frontend.selenium module

class catcher_modules.frontend.selenium.**Selenium**(**kwargs)

Bases: catcher.steps.external_step.ExternalStep

This complex step consists of two parts. First - you need to create a Selenium script and put it in the Catcher's resources directory. Second - run the step in Catcher.

Catcher variables can be accessed from Selenium script via environment variables. All output from Selenium script is routed to Catcher **OUTPUT** variable.

If you specify java/kotlin source file as a Selenium script - Catcher will try to compile it using system's compiler

Test

- driver: path to the driver executable. *Optional*. If not specified - will try to use PATH variable.
- file: path to your file with the test
- libraries: path to selenium client libraries. *Optional*. Used for sources compilation (f.e. .java -> .class). Default is /usr/lib/java/*

Examples

Run selenium python

```
- selenium:
    test:
        driver: '/opt/bin/geckodriver'
        file: 'my_test.py'
```

Compile and run java selenium (MySeleniumTest.java should be in resource dir, selenium cliend libraries should be in /usr/share/java/)

```
- selenium:
    test:
        driver: '/usr/lib/geckodriver'
        file: MySeleniumTest.java
        libraries: '/usr/share/java/*'
```

Python, JavaScript and Jar archives with selenium tests can be stored in any directory, while Java and Kotlin source files must be stored in resources only, as they need to be compiled first.

4.4 catcher_modules.marketing package

4.4.1 Submodules

4.4.2 catcher_modules.marketing.marketo module

class catcher_modules.marketing.marketo.**Marketo**(***kwargs*)

Bases: catcher.steps.external_step.ExternalStep

Allows you to read/write/delete leads in Adobe Marketo marketing automation tool.

Config

- munchkin_id: mailserver's host
- client_id: mailserver's host. *Optional*. Default is 993.
- client_secret: your username

Read read lead record from Marketo

- conf: config object
- fields: fields to retrieve
- filter_key: field to use for filtering. *Optional*. Default is email.
- filter_value: list or single value used in filtering

Write write lead record to Marketo

- conf: config object
- action: 'createOnly', 'updateOnly', 'createOrUpdate', 'createDuplicate'
- lookupField: field to use as a key, for updating. *Optional*. Default is email.
- leads: list of dicts to write to Marketo

Delete delete lead from Marketo

- conf: config object
- by: field name by which lead will be deleted
- having: list or single value

Examples

Read lead by custom_id field

```
marketo:
    read:
        conf:
            munchkin_id: '{{ marketo_munchkin_id }}'
            client_id: '{{ marketo_client_id }}'
            client_secret: '{{ marketo_client_secret }}'
        fields: ['id', 'email', 'custom_field_1']
        filter_key: 'custom_id'
        filter_value: ['my_value_1', 'my_value_2']
    register: {leads: '{{ OUTPUT }}'}
```

Update leads in Marketo by custom_id

```
marketo:
    write:
        conf:
            munchkin_id: '{{ marketo_munchkin_id }}'
            client_id: '{{ marketo_client_id }}'
            client_secret: '{{ marketo_client_secret }}'
        action: 'updateOnly'
        lookupField: 'custom_id'
        leads:
            - custom_id: 14
              email: 'foo@bar.baz'
              custom_field_1: 'some value'
            - custom_id: 15
              email: 'foo2@bar.baz'
              custom_field_1: 'some other value'
```

Delete all leads with emails

```
marketo:
    delete:
        conf:
            munchkin_id: '{{ marketo_munchkin_id }}'
            client_id: '{{ marketo_client_id }}'
            client_secret: '{{ marketo_client_secret }}'
        by: 'custom_id'
        having: ['my_value_1', 'my_value_2']
```

4.5 catcher_modules.mq package

4.5.1 Submodules

4.5.2 catcher_modules.mq.kafka module

class catcher_modules.mq.kafka.**Kafka**(***kwargs*)

Allows you to consume/produce messages from/to Apache Kafka

Input

Consume Consume message from kafka.

- server: is the kafka host. Can be multiple, comma-separated.
- group_id: is the consumer group id. If not specified - *catcher* will be used. *Optional*
- topic: the name of the topic
- timeout: is the consumer timeout. *Optional* (default is 1 sec)
- where: search for specific message clause. *Optional*

Produce Produce message to kafka.

- server: is the kafka host. Can be multiple, comma-separated.
- topic: the name of the topic
- data: data to be produced.
- data_from_file: File can be used as data source. *Optional* Either *data* or *data_from_file* should present.

Examples

Read message with timestamp > 1000

```
kafka:
    consume:
        server: '127.0.0.1:9092'
        group_id: 'test'
        topic: 'test_consume_with_timestamp'
        timeout: {seconds: 5}
        where:
            equals: '{{ MESSAGE.timestamp > 1000 }}'
```

Produce *data* variable as json message

```
kafka:
    produce:
        server: '127.0.0.1:9092'
        topic: 'test_produce_json'
        data: '{{ data|tojson }}'
```

4.5.3 catcher_modules.mq.rabbit module

class catcher_modules.mq.rabbit.**Rabbit**(***kwargs*)

Allows you to consume/produce messages from/to RabbitMQ

Input

Config rabbitmq config object, used in other rabbitmq commands.

- server: is the rabbit host, <rabbit-host:rabbit-port>
- username: is the username
- password: is the password
- virtualhost: virtualhost *Optional* defaults to "/"
- **sslOptions: {'ssl_version': 'PROTOCOL_TLSv1, PROTOCOL_TLSv1_1 or PROTOCOL_TLSv1_2', 'ca_certs': '/p**
  Optional object to be used only when ssl is required. If an empty object is passed ssl_version defaults to PROTOCOL_TLSv1_2 and cert_reqs defaults to CERT_NONE
- **disconnect_timeout: number of seconds to wait for a disconnect before force closing the connection. Warning! Publis**
  may fail if you use to small timeout value.

Consume Consume message from rabbit.

- config: rabbitmq config object
- queue: the name of the queue to consume from

Publish Publish message to rabbit exchange.

- config: rabbitmq config object
- exchange: exchange to publish message
- routing_key: routing key
- headers: headers json *Optional*
- data: data to be produced
- data_from_file: data to be published. File can be used as data source. *Optional* Either *data* or *data_from_file* should present.

Examples

Read message

```
variables:
    rabbitmq_config:
        server: 127.0.0.1:5672
        username: 'guest'
        password: 'guest'
steps:
    - rabbit:
        consume:
            config: '{{ rabbitmq_config }}''
            queue: 'test.catcher.queue'
```

Publish *data* variable as message

```
variables:
    rabbitmq_config:
        server: 127.0.0.1:5672
        sslOptions: {'ssl_version': 'PROTOCOL_TLSv1, PROTOCOL_TLSv1_1 or PROTOCOL_
↪TLSv1_2', 'ca_certs': '/path/to/ca_cert', 'keyfile': '/path/to/key', 'certfile
↪': '/path/to/cert'. 'cert_reqs': 'CERT_NONE, CERT_OPTIONAL or CERT_REQUIRED'}
        username: 'guest'
        password: 'guest'
steps:
    - rabbit:
        publish:
            config: '{{ rabbitmq_config }}''
            exchange: 'test.catcher.exchange'
            routing_key: 'catcher.routing.key'
            headers: {'test.header.1': 'header1', 'test.header.2': 'header1'}
            data: '{{ data|tojson }}'
```

Publish *data_from_file* variable as json message

```
variables:
    rabbitmq_config:
        server: 127.0.0.1:5672
        username: 'guest'
        password: 'guest'
steps:
    - rabbit:
        publish:
            config: '{{ rabbitmq_config }}''
            exchange: 'test.catcher.exchange'
            routing_key: 'catcher.routing.key'
            data_from_file: '{{ /path/to/file }}'
```

4.6 catcher_modules.service package

4.6.1 Submodules

4.6.2 catcher_modules.service.docker module

class catcher_modules.service.docker.**Docker**(***kwargs*)

Allows you to start/stop/disconnect/connect/exec commands, get logs and statuses of Docker containers. Is very useful when you need to run something like Mockserver and/or simulate network disconnects.

Input

Start run container. Return hash.

- image: container's image.
- name: container's name. *Optional*
- cmd: command to run in the container. *Optional*
- detached: should it be run detached? *Optional* (default is True)
- ports: dictionary of ports to bind. Keys - container ports, values - host ports.
- environment: a dictionary of environment variables

- volumes: a dictionary of volumes
- network: network name. *Optional* (default is current test's name)

Stop stop a container.

- name: container's name. *Optional*
- hash: container's hash. *Optional* Either name or hash should present
- delete: delete a container. *Optional* (default is false)

Status get the container status.

- name: container's name. *Optional*
- hash: container's hash. *Optional* Either name or hash should present

Disconnect disconnect a container from a network (network failure simulation)

- name: container's name. *Optional*
- hash: container's hash. *Optional* Either name or hash should present
- network: network name. *Optional* (default is current test's name)

Connect connect a container to a network. All containers share the same network per test.

- name: container's name. *Optional*
- hash: container's hash. *Optional* Either name or hash should present
- network: network name. *Optional* (default is current test's name)

Exec execute a command inside a running container.

- name: container's name. *Optional*
- hash: container's hash. *Optional* Either name or hash should present
- cmd: command to execute.
- dir: directory, where this command will be executed. *Optional*
- user: user to execute this command. *Optional* (default is root)
- environment: a dictionary of environment variables

Logs get container's logs.

- name: container's name. *Optional*
- hash: container's hash. *Optional* Either name or hash should present

Inspect get container's inspect information

- name: container's name. *Optional*
- hash: container's hash. *Optional* Either name or hash should present

Useful hack

if you are going to run docker step from a docker image - you'd need to mount your host's docker `/var/run/docker.sock` directory to the catcher image. docker installation is not included in the catcher's docker image to avoid docker-in-docker problem.

Examples

Run blocking command in a new container and check the output.

```
steps:
    - docker:
        start:
            image: 'alpine'
            cmd: 'echo hello world'
            detached: false
        register: {echo: '{{ OUTPUT.strip() }}'}
    - check:
        equals: {the: '{{ echo }}', is: 'hello world'}
```

Start named container detached with volumes and environment.

```
- docker:
    start:
        image: 'my-backend-service'
        name: 'mock server'
        ports:
            '1080/tcp': 8000
        environment:
            POOL_SIZE: 20
            OTHER_URL: {{ service1.url }}
        volumes:
            '{{ CURRENT_DIR }}/data': '/data'
            '/tmp/logs': '/var/log/service'
```

Exec command on running container.

```
- docker:
    start:
        image: 'postgres:alpine'
        environment:
            POSTGRES_PASSWORD: test
            POSTGRES_USER: user
            POSTGRES_DB: test
    register: {hash: '{{ OUTPUT }}'}
...
- docker:
    exec:
        hash: '{{ hash }}'
        cmd: >
            psql -U user -d test -c                    "CREATE TABLE test(rno␣
↪integer, name character varying)"
    register: {create_result: '{{ OUTPUT.strip() }}'}
```

Get container's logs.

```
- docker:
    start:
        image: 'alpine'
        cmd: 'echo hello world'
```

(continues on next page)

(continued from previous page)

```
    register: {id: '{{ OUTPUT }}'}
- docker:
    logs:
        hash: '{{ id }}'
    register: {out: '{{ OUTPUT.strip() }}'}
- check:
    equals: {the: '{{ out }}', is: 'hello world'}
```

Disconnect a container from a network.

```
- docker:
    disconnect:
        hash: '{{ hash }}'
- http:
    get:
        url: 'http://localhost:8000/some/path'
        should_fail: true
- docker:
    connect:
        hash: '{{ hash }}'
```

4.6.3 catcher_modules.service.elastic module

class catcher_modules.service.elastic.**Elastic**(***kwargs*)

Allows you to get data from Elasticsearch. Useful, when your services push their logs there and you need to check the logs automatically from the test.

Input

Search search elastic

- url: RFC-1738 compatible (can contain user credentials) server url.
- index: ES index (database).
- query: your query to run.
- <other param>: you can add any param here (see Search with limiting fields for an example)

Refresh Trigger a refresh for an index.

- url: RFC-1738 compatible (can contain user credentials) server url.
- index: ES index (database).

Examples

Search with limiting fields

```
elastic:
    search:
        url: 'http://127.0.0.1:9200'
        index: test
        query:
            match: {payload : "three"}
```

(continues on next page)

(continued from previous page)

```
        _source: ['name']
    register: {docs: '{{ OUTPUT }}'}
```

Connect to multiple ES instances. One simple and one secured

```
elastic:
    search:
        url:
            - 'http://127.0.0.1:9200'
            - 'https://{{ user }}:{{ secret }}@{{ host2 }}:443'
        index: test
        query: {match_all: {}}
```

Refresh index

```
elastic:
    refresh:
        url: 'http://127.0.0.1:9092'
        index: test
```

In bool query *must* and *should* are lists

```
elastic:
    search:
        url: 'http://127.0.0.1:9200'
        index: test
        query:
            bool:
                must:
                    - term: {shape: "round"}
                    - bool:
                        should:
                            - term: {color: "red"}
                            - term: {color": "blue"}
```

4.6.4 catcher_modules.service.s3 module

class catcher_modules.service.s3.**S3**(***kwargs*)

Allows you to get/put/list/delete files in Amazon S3

Useful hint: for local testing you can use Minio run in docker as it is S3 API compatible.

Input

Config s3 config object, used in other s3 commands.

- key_id: access key id
- secret_key: secret for the access key
- region: region. *Optional*.
- url: endpoint_url url. Can be used to run against Minio. *Optional*

Put put file to s3

- config: s3 config object

- **path: path including the filename. First dir treats like a bucket.** F.e. /my_bucket/subdir/file or my_bucket/subfir/file
- content: file's content. *Optional*
- content_resource: path to a file. *Optional*. Either *content* or *content_resource* must be set.

Get Get file from s3

- config: s3 config object
- path: path including the filename

List List S3 directory

- config: s3 config object
- path: path to the directory being listed

Delete Delete file or directory from S3

- config: s3 config object
- path: path to the deleted
- **recursive: if path is directory and recursive is true - will delete directory with all content. *Optional*,** default is false.

Examples

Put data into s3

```
s3:
    put:
        config: '{{ s3_config }}'
        path: /foo/bar/file.csv
        content: '{{ my_data }}'
```

Get data from s3

```
s3:
    get:
        config: '{{ s3_config }}'
        path: /foo/bar/file.csv
    register: {csv: '{{ OUTPUT }}'}
```

List files

```
s3:
    list:
        config: '{{ s3_config }}'
        path: /foo/bar/
    register: {files: '{{ OUTPUT }}'}
```

Delete file

```
s3:
    delete:
        config: '{{ s3_config }}'
        path: '/remove/me'
        recursive: true
```

4.6.5 catcher_modules.service.prepare module

class catcher_modules.service.prepare.**Prepare**(***kwargs*)

Used for bulk actions to prepare test data. Is useful when you need to prepare a lot of data. This step consists of 3 parts:

1. write sql ddl schema file (optional) - describe all tables/schemas/privileges needed to be created
2. prepare data in a csv file (optional)
3. call Catcher's prepare step to populate csv content into the database

Both sql schema and csv file supports templates.

Important:

- populate step is designed to be supported by all steps (in future). Currently it is supported only by Postges/Oracle/MSSql/MySql/SQLite steps.
- to populate json as Postgres Json data type you need to use **use_json: true** flag

  Input

  Populate Populate existing service with predefined data.

- **<service_name>: See each own step's documentation for the parameters description and** information. Note, that not all steps are compatible with prepare step.
- variables: Variables, which will override state (only for this prepare step).

Please, keep it mind, that resources directory is used for all data and schema files.

Populate existing postgres with data from *pg_data_file*.

```
steps:
    - prepare:
        populate:
            postgres:
                conf: {{ pg_conf }}
                schema: {{ pg_schema_file }}
                data: {{ pg_data_file }}
```

Multiple populates and can be run at the same time. This will populate existing s3 with data, start local salesforce and postgres in docker and populates them as well.

```
steps:
    - prepare:
        populate:
            s3:
                conf: {{ s3_url }}
                path: {{ s3_path }}
                data: {{ s3_data }}
```

(continues on next page)

(continued from previous page)

```
            postgres:
                conf: {{ pg_conf }}
                schema: {{ pg_schema_file }}
                data: {{ pg_data_file }}
```

Prepare step with variables override.

```
- prepare:
    populate:
      postgres:
           conf: '{{ postgres_conf }}'
           schema: create_personal_data_customer.sql
      variables:
           email: '{{ random("email") }}'
```

4.6.6 catcher_modules.service.expect module

class catcher_modules.service.expect.**Expect**(***kwargs*)

This is the opposite for prepare. It compares expected data from csv to what you have in the database. csv file supports templates.

Important:

- populate step is designed to be supported by all steps (in future). Currently it is supported only by Postges/Oracle/MSSql/MySql/SQLite steps.
- Schema comparison is not implemented.
- You can use strict comparison (only data from csv should be in the table, in the same order as csv) or the default one (just check if the data is there)

  Input

  Compare Compare the existing data with expected one.

- **<service_name>: See each own step's documentation for the parameters description and** information. Note, that not all steps are compatible with prepare step.

Check expected schema and data in postgres.

```
steps:
    - expect:
        compare:
            postgres:
                url: {{ pg_conf }}
                schema: {{ expected_schema_file }}
                data: {{ expected_data_file }}
```

Check data in s3 and redshift.

```
steps:
    - expect:
        compare:
            s3:
                url: {{ s3_url }}
                path: {{ expected_path }}
```

(continues on next page)

(continued from previous page)

```
                csv:
                    header: true
                    headers: {{ expected_headers }}
            redshift:
                url: {{ redshift_url }}
                schema: {{ expected_schema }}
                data: {{ expected_data }}
```

4.6.7 catcher_modules.service.email module

class catcher_modules.service.email.**Message**(*message*)

class catcher_modules.service.email.**Email**(***kwargs*)

Allows you to send and receive emails via IMAP protocol.

Config

- host: mailserver's host
- port: mailserver's host. *Optional*. Default is 993.
- user: your username
- pass: your password
- ssl: use tls. *Optional* Default is true.
- starttls: use starttls. *Optional* Default is false.

Filter search filter object. All fields are optional. For more details and filter options please see the readme's of https://github.com/martinrusev/imbox library.

- unread: boolean. If true will get only unread messages. Default is false.
- sent_from: Get only messages sent from this address.
- sent_to: Get only messages sent to this address.
- date__lt: Get messages received before specific date.
- date__gt: Get messages received after specific date.
- date__on: Get messages received on a specific date.
- subject: Get messages whose subjects contain specified string.
- folder: Get messages from a specific folder.

Input

Receive get a list of messages, matching search criteria. From recent to old.

- config: email's config object.
- filter: add search filter. *Optional*.
- ack: mark as read. *Optional* Default is false.
- **limit: limit return result to N messages. *Optional* Default is unlimited.** Only messages who fit the limit will be marked as read, if ack is true.

Send send an email

- config: email's config object.
- from: from email
- to: to email or list of emails
- cc: list of cc. *Optional*
- bcc: list of bcc. *Optional*
- subject: subject. *Optional* Default is empty string.
- plain: message's text. *Optional*
- html: message's text in html format. *Optional* Either *plain* or *html* should present.
- attachments: list with attachment filenames from resources dir. *Optional*

Message for fields, available in message please see *Message*

Examples

Read all messages, take the last one and check subject

```
variables:
    email_config:
        host: 'imap.google.com'
        user: 'my_user@google.com'
        pass: 'my_pass'
steps:
    - email:
        receive:
            conf: '{{ email_config }}'
        register: {last_mail: '{{ OUTPUT[0] }}'}
    - check: {equals: {the: '{{ last_mail.subject }}', is: 'Test Subject'}}
```

Read 2 last unread messages and mark them read

```
- email:
      receive:
          config: '{{ email_conf }}'
          filter: {unread: true}
          ack: true
          limit: 2
```

Find unread message containing blog name in subject and mark as read

```
- email:
      receive:
          config: '{{ email_conf }}'
          filter: {unread: true, subject: 'justtech.blog'}
          ack: true
          limit: 1
```

Send message in html format

```
- email:
      send:
          config: '{{ email_conf }}'
```

(continues on next page)

(continued from previous page)

```
          to: 'test@test.com'
          from: 'me@test.com'
          subject: 'test_subject'
          html: '
          <html>
              <body>
                <p>Hi,<br>
                   How are you?<br>
                   <a href="http://example.com">Link</a>
                </p>
              </body>
          </html>'
```

4.6.8 catcher_modules.service.salesforce module

class catcher_modules.service.salesforce.**Salesforce**(***kwargs*)

Allows you to work with Salesforce.

Config

- instance: your Salesforce instance. *Optional*. Either instance or instance_url must exist.
- instance_url: full URL of your instance. *Optional*. Either instance or instance_url must exist.
- username: your username *Optional*.
- password: your password *Optional*.
- security_token: token is usually provided when you change your password *Optional*.
- organizationId: whitelisted Organization ID. *Optional*.
- consumer_key: consumer key from your app for JWT auth. *Optional*.
- privatekey_file: path to the private key file (with resources as root) for the JWT auth. *Optional*
- client_id: used for requests tracking. *Optional*.
- domain: domain to be used. *Optional*.
- session: Salesforce session. *Optional*.

Input

Query run SOQL query

- soql: query to run
- config: Config object

<action> record's possible action: create/update/upsert/get/get_by_custom_id/delete/deleted/updated

- <record>: <data>

where **<record>** is SF record name. and **<data>** is an action param. In case of multiple params use list (see examples)

Examples

Run SOQL query

```
salesforce:
    query:
        config:
            password='password'
            username='myemail@example.com'
            organizationId='OrgId'
        soql: "SELECT Id, Email FROM Contact WHERE LastName = 'Jones'"
    register: {contacts: '{{ OUTPUT }}'}
```

Create new record

```
salesforce:
    create:
        config:
            password='password'
            username='myemail@example.com'
            organizationId='OrgId'
        Contact:
            LastName: Smith
            Email: example@example.com
```

Upsert a record. First param is the upsert id, second is the record

```
salesforce:
    upsert:
        config:
            password='password'
            username='myemail@example.com'
            organizationId='OrgId'
        Contact:
            - customExtIdField__c/11999
            - LastName: Smith
              Email: example@example.com
```

4.7 catcher_modules.pipeline package

4.7.1 Submodules

4.7.2 catcher_modules.pipeline.airflow module

4.8 Prepare

There is a special step prepare, which allows you to populate your data sources with batch operations. It is extremely useful when testing big data pipelines or when you need some amount of data pre-populated.

4.8.1 How it worked before?

You have to populate the database manually:

```
---
variables:
```

(continues on next page)

(continued from previous page)

```
    username: 'test'
steps:
    - postgres:
        actions:
            - request:
                conf: '{{ postgres }}'
                query: "CREATE TABLE foo(user_id integer primary key, email␣
↪varchar(36) NOT NULL);"
              name: 'create foo table'
            - request:
                conf: '{{ postgres }}'
                query: "CREATE TABLE bar(key varchar(36) primary key, value␣
↪varchar(36) NOT NULL);"
              name: 'create bar table'
            - request:
                conf: '{{ postgres }}'
                query: "INSERT INTO foo values (1,"{{username}}1@test.com"),(2,"{
↪{username}}2@test.com");
              name: 'populate foo table'
            - request:
                conf: '{{ postgres }}'
                query: "INSERT INTO bar values ("key1","value1"),("key2","value2");
              name: 'populate bar table'
    ... and now you can run your test steps
```

It is not so nice, as you have to write a lot of steps just to pre-populate your data source.

4.8.2 How it works now?

For the postgres example above.
1. You create a schema file with DDL.

resources/schema.sql:

```
CREATE TABLE foo(
    user_id      varchar(36)    primary key,
    email        varchar(36)    NOT NULL
);

CREATE TABLE bar(
    key           varchar(36)    primary key,
    value         varchar(36)    NOT NULL
);
```

2. You create a data file for each table you'd like to populate.

resources/foo.csv:

```
user_id,email
1,{{username}}1@test.com
2,{{username}}2@test.com
```

resources/bar.csv:

```
key,value
k1,v1
k2,v2
```

3. You call one prepare step for postgres.

test.yml:

```
---
steps:
    - prepare:
        populate:
            postgres:
                conf: '{{ postgres }}'
                schema: pg_schema.sql
                data:
                    foo: foo.csv
                    bar: bar.csv
    ... and now you can run your test steps
```

In this step you prepare all the data needed. Tables will be created and populated.
Note on templating - it is fully supported. Even new rows can be generated in the csv files.

foo.csv:

```
user_id,email
{% for user in users %}
{{ loop.index }},{{ user }}
{% endfor %}
4,other_email
```

4.9 Expect

There is a special step expect, which allows you to check your data sources with predefined data. It is extremely useful when testing big data pipelines or when your services produce a lot of data to be checked.

4.9.1 How it worked before?

You have to check the database manually:

```
steps:
    - ... calls to your services producing data
    - postgres:
        request:
            conf: '{{ postgres }}'
            query: 'select count(*) from foo'
```

(continues on next page)

(continued from previous page)

```
        register: {documents: '{{ OUTPUT.count }}'}
    - check:
        equals: {the: '{{ documents }}', is: 2}
    - postgres:
        request:
            conf: '{{ postgres }}'
            query: 'select count(*) from bar'
        register: {documents: '{{ OUTPUT.count }}'}
    - check:
        equals: {the: '{{ documents }}', is: 2}
```

Even if you run postgres + check steps as a registered include it is still a lot of unnecessary (from now) steps.

4.9.2 How it works now?

For the postgres example above.
1. Create csv with expected data for the tables.

resources/foo.csv:

```
user_id,email
1,test1@test.com
2,test2@test.com
```

resources/bar.csv:

```
key,value
k1,v1
k2,v2
```

2. Run the expect step.

test.yml:

```
steps:
    - ... calls to your services producing data
    - expect:
        compare:
            postgres:
                conf: 'test:test@localhost:5433/test'
                data:
                    foo: foo.csv
                    bar: bar.csv
```

Note that not all steps support prepare-expect for now.
Note on templating - it is fully supported. Even new rows can be generated in the csv files.

4.10 Airflow

4.10.1 Configure connections

`catcher_modules.pipeline.airflow()` step allows you to configure airflow connections based on your inventory. To do this you need to specify **populate_connections** and **fernet_key** in airflow step configuration:

```
steps:
      - airflow:
          run:
            config:
              db_conf: '{{ airflow_db }}'
              url: '{{ airflow_web }}'
              populate_connections: true
              fernet_key: '{{ airflow_fernet }}'
            dag_id: '{{ pipeline }}'
            sync: true
            wait_timeout: 150
          name: 'Trigger pipeline {{ pipeline }}'
```

For every connection from the inventory having **type** parameter catcher will try to create the connection with the same name in Airflow, if it **does not** already exists.

Then in your inventory you can have:

```
psql_conf:
  url: 'postgresql://postgres:postgres@custom_postgres_1:5432/postgres'
  type: 'postgres'
airflow_db:
  url: 'airflow:airflow@postgres:5432/airflow'
  type: 'postgres'
airflow_web:
  url: 'http://webserver:8080'
  type: 'http'
airflow_fernet: 'zp8kV516l9tKzqq9pJ2Y6cXbM3bgEWIapGwzQs6jio4='
s3_config:
    url: http://minio:9000
    key_id: minio
    secret_key: minio123
    type: 'aws'
```

type parameter should be the same as Airflow's *Conn Type* field.

And in your pipeline:

```
postgres_conn_id = 'psql_conf'
mysql_conn_id = 'mysql_conf'
aws_conn_id = 's3_config'

def my_step():
    psql_hook = PostgresHook(postgres_conn_id=postgres_conn_id)
```

You can check *Admin -> Connections* for newly created connections. Catcher will create psql_conf, s3_config and

airflow_web. Airflow_db is skipped, as it was already created before and airflow_fernet is not a connection.
You can safely use these connections in your pipeline.

Extra

If you specify **extra** field in your inventory - it will be populated to airflow. If there is no **extra** field, but it is needed (f.e. for aws field) it will be computed based on the configuration.

config:

```
s3_config:
    url: http://minio:9000
    key_id: minio
    secret_key: minio123
    type: 'aws'
```

Will have these extra json:

```
{
    'host': s3_config.get('url'),
    'aws_access_key_id': s3_config.get('key_id'),
    'aws_secret_access_key': s3_config.get('secret_key'),
    'region_name': s3_config.get('region')
}
```

Alternatively you can set up extra yourself:

```
s3_config:
    type: 'aws'
    extra: '{"host": "http://minio:9000","aws_access_key_id":"minio","aws_secret_
↪access_key":"minio123"}'
```

Although in this case it will be valid for Airflow population only. `catcher_modules.service.s3()` step won't be able to use such configuration.

Catcher will encrypt extra automatically.

4.10.2 Docker usage

If you run your airflow in docker-compose locally with all the dependencies available via **localhost:<docker_forwarded_port>** you probably use local inventory for Catcher with something like

```
psql_conf:
    url: 'postgres:postgres@localhost:5432/postgres'
    type: 'postgres'
```

If you ask Catcher to initialize Airflow's connections based on your local inventory - it will create connection with **localhost** host, which won't work in docker, as localhost will point within the container, but not the host.
Workaround for it is to run Catcher in docker within the same network as your Airflow and have another inventory which will point to containers

```
psql_conf:
    url: 'postgresql://postgres:postgres@custom_postgres_1:5432/postgres'
    type: 'postgres'
```

Checklist:

- make sure you specify inventory to run
- make sure you set proper fernet key
- make sure you set populate_connections: true and proper connection type
- make sure connections with same names do not exist in airflow

4.11 Front End testing with Selenium

4.11.1 The test

You can run Selenium steps to involve your front-end applications in the end-to-end test. Selenium step consists of two parts - selenium test and Catcher's `catcher_modules.frontend.selenium()` step definition which will trigger it.

Selenium step can look like:

```
- selenium:
    test:
        file: resources/MySeleniumTest.java
        libraries: '/usr/share/java/*'
```

file argument should point to your Selenium test. Catcher currently support JavaScript, Python, Java, Kotlin and Jar-files (in case you already prepared tests and don't need to compile them every time you run the step).
This example will run your Java-based Selenium test. In case of Java/Kotlin it will try to compile it using provided **libraries** argument.

Java source file looks like:

```
package selenium;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;


public class MySeleniumTest {

    public static void main(String[] args) {
        WebDriver driver = new FirefoxDriver();
        try {
            driver.get("http://www.google.com/ncr");
            WebElement element = driver.findElement(By.name("q"));
            element.sendKeys("Cheese!");
            element.submit();
```

(continues on next page)

(continued from previous page)

```
        } finally {
            driver.quit();
        }
    }
}
```

It will go to google and search by "Cheese!" string. Catcher's step will pass if Selenium test passes.

4.11.2 Variables and output

Catcher pushes it's context variables to Selenium steps as environment variables, so you can access them from tests. For example:

```
variables:
    site_url: 'http://www.google.com/ncr'
steps:
    - selenium:
        test:
            driver: '/usr/lib/geckodriver'
            file: resources/test.js
        register: {title: '{{ OUTPUT.title }}'}
    - echo: {from: '{{ title }}', to: variable.output}
```

Here Catcher's context has **site_url** variable which is accessible within JavaScript (for other languages similarly). As well as page's title is accessible for Catcher as an output variable.

test.js:

```
const {Builder, By, Key, until} = require('selenium-webdriver');
    async function basicExample(){
        let driver = await new Builder().forBrowser('firefox').build();
        try{
            await driver.get(process.env.site_url);
            await driver.findElement(By.name('q')).sendKeys('webdriver', Key.RETURN);
            await driver.wait(until.titleContains('webdriver'), 1000);
            await driver.getTitle().then(function(title) {
                        console.log('{\"title\":\"' + title + '\"}')
                });
            driver.quit();
        }
        catch(err) {
            console.error(err);
            process.exitCode = 1;
            driver.quit();
          }
    }
    basicExample();
```

To output any value simply use stdout. In case of Json you can access fields directly (f.e. *OUTPUT.title* json output). If Catcher fails to parse json - it will use the whole output as a plain text.

4.11.3 Libraries and Dependencies

Catcher dockerfile comes with **nodeJS**, **Java1.8**, **Kotlin** and **Python** Selenium libraries installed, as well as **Firefox**, **Chrome** and **Opera** drivers.

To run Selenium steps locally you need all dependencies, libraries and drivers be installed in your system. Driver should be included in your PATH or passed as a **driver** argument.

In case of Java/Kotlin scr compilation you should have **javac/kotlinc** installed in your system.

In case of jar files running - **java** should be installed.

In case of JavaScript - **nodeJs** should be installed.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index