satellite-populate Documentation

Release 0.1.3

Bruno Rocha

Aug 18, 2017

Contents

Contents:

CHAPTER 1

Satellite-Populate

Populate and Validate the System using YAML

- Free software: GNU General Public License v3
- Documentation: https://satellite-populate.readthedocs.io.

Installation

To install latest released version:

```
pip install satellite-populate
```

To install from github master branch:

```
pip install https://github.com/SatelliteQE/satellite-populate/tarball/master
```

For development:

```
# fork https://github.com/SatelliteQE/satellite-populate/ to YOUR_GITHUB
# clone your repo locally
git clone git@github.com:YOUR_GITHUB/satellite-populate.git
cd satellite-populate

# add upstream remote
git remote add upstream git@github.com:SatelliteQE/satellite-populate.git

# create a virtualenv
mkvirtualenv satellite-populate
workon satellite-populate

# install for development (editable)
pip install -r requirements.txt
```

Testing if installation is good:

```
$ satellite-populate --test
satellite_populate.base - INFO - ECHO: Hello, if you can see this it means that I am␣
↪working!!!
```

Features

YAML based actions

Data population definition goes to YAML file e.g `office.yaml` in the following example we are going to create 2 organizations and 2 admin users using lists:

```yaml
vars:

  org_names:
    - Dunder Mifflin
    - Wernham Hogg

  user_list:
    - firstname: Michael
      lastname: Scott

    - firstname: David
      lastname: Brent

actions:

  - model: Organization
    with_items: org_names
    register: default_orgs
    data:
      name: "{{ item }}"
      label: org{{ item.replace(' ', '') }}
      description: This is a satellite organization named {{ item }}

  - model: User
    with_items: user_list
    data:
      admin: true
      firstname: "{{ item.firstname }}"
      lastname: "{{ item.lastname }}"
      login: "{{ '{0}{1}'.format(item.firstname[0], item.lastname) | lower }}"
      password:
        from_factory: alpha
      organization:
        from_registry: default_orgs
      default_organization:
        from_registry: default_orgs[loop_index]
```

On the populate file you can define CRUD actions such as **create**, **delete**, **update** if `action:` is not defined, the default will be `create`.

And also there is **special actions** and **custom actions** explained later.

Populate Satellite With Entities

Considering `office.yaml` file above you can populate satellite system with the command line:

```
$ satellite-populate office.yaml -h yourserver.com --output=office.yaml -v
```

In the above command line `-h` stands for `--hostname`, `--output` is the output file which will be written to be used to validate the system, and `-v` is the verbose level.

To see the list of available arguments please run:

```
# satellite-populate --help
```

Validate if system have entities

Once you run `satellite-populate` you can use the outputted file to validate the system. as all the output files are named as `validation_<name>.yaml` in office example you can run:

```
$ satellite-populate validation_office.yaml -v
```

Using that validation file the system will be checked for entities existence, read-only. The Validation file exists because during the population dynamic data is generated such as passwords and strings `from_factory` and also some entities can be deleted or updated so validation file takes care of it.

Special actions

Some builtin special actions are:

- assertion
- echo
- register
- unregister

In the following example we are going to run a complete test case using actions defined in YAML file, if validation fails system returns status 0 which can be used to automate tests:

```
# A TEST CASE USING SPECIAL ACTIONS
# Create a plain vanilla activation key
# Check that activation key is created and its "unlimited_hosts"
# attribute defaults to true

- action: create
  log: Create a plain vanilla activation key
  model: ActivationKey
  register: vanilla_key
  data:
     name: vanilla
     organization:
       from_registry: default_orgs[0]

- action: assertion
  log: >
    Check that activation key is created and its "unlimited_hosts"
    attribute defaults to true
```

```
  operation: eq
  register: vanilla_key_unlimited_hosts
  data:
    - from_registry: vanilla_key.unlimited_hosts
    - true

- action: echo
  log: Vanilla Key Unlimited Host is False!!!!
  level: error
  print: true
  when: vanilla_key_unlimited_hosts == False

- action: echo
  log: Vanilla Key Unlimited Host is True!!!!
  level: info
  print: true
  when: vanilla_key_unlimited_hosts

- action: register
  data:
    you_must_update_vanilla_key: true
  when: vanilla_key_unlimited_hosts == False
```

Custom actions

And you can also have special actions defined in a custom populator.

Lets say you have this python module in your project, properly available on PYTHONPATH:

```
from satellite_populate.api import APIPopulator

class MyPopulator(APIPopulator):
    def action_writeinfile(self, rendered_data, action_data):
        with open(rendered_data['path'], 'w') as output:
            output.write(rendered_data['content'])
```

Now go to your `test.yaml` and write:

```
config:
  populator: mine
  populators:
    mine:
      module: mypath.mymodule.MyPopulator

actions:

  - action: writeinfile
    path: /tmp/test.txt
    content: Hello World!!!
```

and run:

```
$ satellite-populate test.yaml -v
```

Decorator for test cases

Having a data_file like:

```
actions:
  - model: Organization
    register: organization_1
    data:
      name: My Org
```

Then you can use in decorators:

```
@populate_with('file.yaml')
def test_case_(self):
    'My Org exists in system test anything here'
```

And getting the populated entities inside the test_case:

```
@populate_with('file.yaml', context_name='my_context')
def test_case_(self, my_context=None):
    assert my_context.organization_1.name == 'My Org'

You can also set a customized context wrapper to the
context_wrapper argument::

    def my_custom_context_wrapper(result):
        # create an object using result
        my_context = MyResultContext(result)
        return my_context

    @populate_with('file.yaml', context_name='my_context',
                   content_wrapper=my_custom_context_wrapper)
    def test_case_(self, my_context=None):
        # assert with some expression using my_context object returned
        # my_custom_context_wrapper
        assert some_expression
```

NOTE:

```
That is important that ``context`` argument always be declared using
either a default value ``my_context=None`` or handle in ``**kwargs``
Otherwise ``py.test`` may try to use this as a fixture placeholder.

if context_wrapper is set to None, my_context will be the pure unmodified
result of populate function.
```

Satellite versions

This code is by default prepared to run against Satellite **latest** version which means the use of the **latest** master from **nailgun** repository.

If you need to run this tool in older versions e.g: to tun upgrade tests, you have to setup **nailgun** version.

You have 2 options:

Manually

before installing satellite-populate install specific nailgun version as the following list.

- Satellite 6.1.x:

```
pip install -e git+https://github.com/SatelliteQE/nailgun.git@0.28.0#egg=nailgun
pip install satellite-populate
```

- Satellite 6.2.x:

```
pip install -e git+https://github.com/SatelliteQE/nailgun.git@6.2.z#egg=nailgun
pip install satellite-populate
```

- Satellite 6.3.x (latest):

```
pip install -e git+https://github.com/SatelliteQE/nailgun.git#egg=nailgun
pip install satellite-populate
```

Docker

If you need to run `satellite-populate` in older Satellite versions you can use the `docker images` so it will manage the correct nailgun version to be used with that specific system version.

https://hub.docker.com/r/satelliteqe/satellite-populate/

First pull image from Docker Hub:

```
docker pull satelliteqe/satellite-populate:latest
```

Change `:latest` to specific tag. e.g: `:6.1` or `:6.2`

Test it:

```
docker run satelliteqe/satellite-populate --test
```

Then run:

```
docker run -v $PWD:/datafiles satelliteqe/satellite-populate /datafiles/theoffice.
↪yaml -v -h server.com
```

You must map your local folder containing datafiles

Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER 2

Usage

This section explains Satellite Populate data populate.

Contents

Commands

Using `$ satellite-populate` you can run the `populate` and `validate` commands. That commands are used to read data description from YAML file and populate the system or validate populated entities.

Having `test_data.yaml` with the following content.

```
vars:
  org_label_suffix = inc
actions:
  - model: Organization
    log: The first organization...
    register: org_1
    data:
      name: MyOrg
      label: MyOrg{{org_label_suffix}}
```

To populate the system

```
(satellite_env)[you@host]$ satellite-populate test_data.yaml -v -o validation_data.
↪yaml
2017-01-04 04:31:17 - satellite_populate.base - INFO - CREATE: The first organization.
↪..
2017-01-04 04:31:19 - satellite_populate.base - INFO - search: Organization {'query':
↪{'search': 'name=MyOrg,label=MyOrg'}} found unique item
2017-01-04 04:31:19 - satellite_populate.base - INFO - create: Entity already exists:␣
↪Organization 36
2017-01-04 04:31:19 - satellite_populate.base - INFO - registry: org_1 registered
```

To validate the system use the file generated by population `validation_data.yaml`

```
(satellite_env)[you@host]$ satellite-populate validation_data.yaml
(satellite_env)[you@host]$ echo $?
0  # system validated else 1
```

Use `$ satellite-populate --help` for more info

Hostname and Credentials

Pass -h –hostname, -p –password, -u –username to the command, or this arguments to decorator:

```
@populate_with(data, username='x', password='y', hostname='server.com')
```

NOTE:

> "validation data can also be included in *config* section"

Decorator

Other way to use populate is via decorator, it is useful to decorate a test_case forcing a populate or validate operation to be performed.

Having a data_file like:

```
actions:
  - model: Organization
    register: organization_1
    data:
      name: My Org
```

Then you can use in decorators:

```
@populate_with('file.yaml')
def test_case_(self):
    'My Org exists in system test anything here'
```

And getting the populated entities inside the test_case:

```
@populate_with('file.yaml', context_name='my_context')
def test_case_(self, my_context=None):
    assert my_context.organization_1.name == 'My Org'

You can also set a customized context wrapper to the
context_wrapper argument::

    def my_custom_context_wrapper(result):
        # create an object using result
        my_context = MyResultContext(result)
        return my_context

    @populate_with('file.yaml', context_name='my_context',
                   content_wrapper=my_custom_context_wrapper)
    def test_case_(self, my_context=None):
        # assert with some expression using my_context object returned
        # my_custom_context_wrapper
        assert some_expression
```

And if you don't want to have YAML file you can provide a dict:

```
data_in_dict = {
    'actions': [
        {
            'model': 'Organization',
            'register': 'organization_1',
            'data': {
                'name': 'My Organization 1',
                'label': 'my_organization_1'
            }
        },
    ]
}


@populate_with(data_in_dict, context_name='my_context', verbose=1)
def test_org_1(my_context=None):
```

```
    """a test with populated data"""
    assert my_context.organization_1.name == "MyOrganization1"
```

And finally it also accepts bare YAML string for testing purposes:

```
data_in_string = """
actions:
- model: Organization
  registry: organization_3
  data:
    name: My Organization 3
    label: my_organization_3
"""

@populate_with(data_in_string, context_name='context', verbose=1)
def test_org_3(context=None):
    """a test with populated data"""
    assert context.organization_3.name == "My Organization 3"
    assert context.organization_3.label == "my_organization_3"
```

NOTE:

> "That is important that `context_name` argument always be declared using either a default value `my_context=None` or handle in `**kwargs` Otherwise `py.test` may try to use this as a fixture placeholder. And if context_wrapper is set to None, my_context will be the pure unmodified result of populate function."

Decorating UnitTest setUp and test_cases:

```
class MyTestCase(TestCase):
    """
    This test populates data in setUp and also in individual tests
    """
    @populate_with(data_in_string, context_name='context')
    def setUp(self, context=None):
        self.context = context

    def test_with_setup_data(self):
        self.assertEqual(
            self.context.organization_3.name, "My Organization 3"
        )

    @populate_with(data_in_dict, context_name='test_context')
    def test_with_isolated_data(self, test_context=None):
        self.assertEqual(
            test_context.organization_1.name, "My Organization 1"
        )
```

The YAML data file

In the YAML data file it is possible to specify 3 sections, `config`, `vars` and `actions`.

config

The config may be used to define special behavior of populator and its keys are:

example:

```
config:
  verbose: 3
  populator: api
  populators:
    api:
      module: satellite_populate.api.APIPopulator
    cli:
      module: satellite_populate.cli.CLIPopulator
```

Config variables:

```
config:
  # Set verbosity to -v, -vv, -vvv, -vvvv, -vvvvv
  # int
  # range(0, 5)
  verbose: 1

  # define the default active populator name
  # str
  populator: foo

  # specify available populators
  # dict(<name>=dict(module='module_path'))
  populators:
    foo:
      module: mypack.mymodule.MyPopulatorClass
    other:
      module: otherpath.OtherClass

  # define the mode (override by argument)
  # str
  # choices: validate | populate
  mode: validate

  # http or https ? (override by argument)
  schema: http

  # Satellite system port (override by argument)
  port: 443

  # hostname (without scheme) (override by argument)
  hostname: server.com

  # Admin username (override by argument)
  username: admin

  # admin password (override by argument)
  password: changeme

  # User for ssh login (override by argument)
  ssh_user: root

  # Ssh auth (override by argument)
  # if None local ~/.ssh pub key is used
  # or password
  # or keyfile
  ssh_auth:
```

```
    password: 123456
    key_file: path/to/file.pub

  # raw search rules is a dict of rules
  # to force some transformations over nailgun
  # EntitySearchMixin
  # in the example below we are removing the password
  # field from search queries for User entity
  raw_search_rules:
    user:
      password:
        remove: true

  # In some cases a GPGKey is needed for nailgun
  gpgkey:
    content: skjfsdhbgbsdhbgsdjbg=
    docker_url: system.com:dockerport

  # inject following modules to context (import)
  add_to_context:
    path: os.path
    shortname: package.module.module.module.object
    # the above will available as {{ shortname }}
```

vars

Variables to be available in the rendering context of the YAML data every var defined here is available to be referenced using `Jinja` syntax in any action.

```
vars:
  admin_username: admin
  admin_password: changeme
  org_name_list:
    - company7
    - company8
  prefix: aaaa
  suffix: bbbb
  my_name: me
```

Actions

The actions is the most important section of the YAML, it is a list of actions being each action a dictionary containing special keys depending on the action type.

Actions are executed in the defined order and order is very important because each action can `register` its result to the internal registry to be referenced later in any other action.

The action type is defined in `action` key and available actions are:

CRUD

Crud actions takes a `model` argument, any from `nailgun.entities` is a valid model, models are passed as `CamelCasedName` of the antity class, then, depending on the populator being used, that CRUD action can be performed by API, CLI or UI.

List of possible variables for crud actions:

```
# action name - create | delete | update
action: create

# entity class
model: User

# name to register
register: my_user

# log message to output
log: Creating a new user ....

# Must iterate a list to repeat the same action?
with_items:
  - item1
  - item2
  ...

# The data to perform a search for the entity
data:

  # base types - int, str, list etc..
  name: Foo bar

  # from an available Python object
  url:
    from_object: somemodule.constants.REPO_URL

  # from a search in the system
  organization:
    from_search:
      model: Organization
      data:
        name: SomeCompanyName

  # from specific ID
  product:
    from_read:
      model: Product
      data:
        id: 1

  # from registered action
  user:
    from_registry: already_existing_user

  # from fauxfactory generator
  password:
    from_factory: alphanumeric

# If needed specify data to be used only for search (in validation)
search_query:
  field: something

# If needed custom options can be passed to nailgun search
search_options:
  filter: {}
```

```
  # should force a raw search or use attribute search?
  # note: some entities such as Organization will always be raw searched
  force_raw: true | false

# Choose which populator to use for this specific action
# NotImplementedYet
via: api | cli | ui | custom_populator

# Should errors be silenced and None registered if error?
silent_errors: true | false

# Run async?
# NotImplementedYet
async: true | false
wait: other_action_register_name

# Run only in the case of following condition
# Python allowed, registered objects allowed
# should be a Boolean operation
when: object_a == object_b and 1 > 0
```

create

Search for the new entity and creates if not found, else only register the object.

- If no action is informed **create** will be always the default
- In populate perform search then create
- In validate perform only search

Required variables:

- **model**: Nailgun Entity Class name
- **data**: a dictionary to search or populate the entity

Creating a simple Organization:

```
# a list of dictionaries
actions:

  - model: Organization  # the nailgun Entity class

    # The message to output in the log
    log: This is the first organization

    # The name which this object will be registered
    # to be referenced in other actions.
    register: my_organization

    # The data to search or populate the entity
    data:
      name: My Company
      label: mycompany
```

Creating 2 organizations and 2 users from lists and referencing objects from the registry:

```yaml
vars:

  # a list with data for 2 users
  user_list:
    - firstname: Michael
      lastname: Scott
    - firstname: David
      lastname: Brent

  # a list of company names
  company_names:
    - Dunder Mifflin
    - Wernham Hogg

actions:

  # create all the organizations listed above
  - model: Organization

    # iterate specified list and repeats the action for each
    with_items: company_names

    # include the result in registry
    # if `with_items` is used, the registered object will be a list
    register: companies

    # give the data
    data:
      name: "{{item}}"
      label: "{{item.replace(' ', '')}}"  # transform name in a valid label

  # Create one user as admin for each organization
  - model: User
    with_items: user_list
    data:
      admin: true
      # refer to loop iteration using `items` object
      firstname: "{{item.firstname}}"
      lastname: "{{item.lastname}}"

      # Use object methods and Jinja filters to transform data
      # the following gives us mscott and dbrent
      login: "{{ '{0}{1}'.format(item.firstname[0], item.lastname) | lower }}"

      # generate a random password using builtin fauxfactory
      password:
        from_factory: alpha

      # Set the organizations to existing list of orgs
      organization:
        from_registry: companies

      # Set as default org the same positioned in the loop
      default_organization:
        from_registry: companies[loop_index]
```

update

Get some existing entity and updates it with provided data.

- Executed only in populate mode
- In validate mode it only searches for updated entity

Required variables:

- **model**: Nailgun Entity Class name
- **registry** The name registry object
- **data**: a dictionary to search

Updating the product named *old_name* with *new_name*:

```
actions:
   - action: update
     model: Product
     register: some_product
     data:
       name: new_name
     search_query:
       name: old_name
       organization:
           from_search:
             model: Organization
             data:
               name: Default Organization
```

If the *some_product* already exists in registry you can omit the search:

```
actions:
   - action: update
     model: Product
     register: some_product
     data:
       name: new_name
```

delete

Deletes existing entity.

- Executed only in populate mode
- In validate mode it only searches for updated entity

Required variables:

- **model**: Nailgun Entity Class name
- **registry** The name registry object
- **data**: a dictionary to search

Deleting the product named *new_name*:

```
actions:
   - action: delete
     model: Product
     search_query:
       name: new_name
       organization:
           from_search:
             model: Organization
             data:
               name: Default Organization
```

If the *some_product* already exists in registry you can omit the search:

```
actions:
   - action: delete
     model: Product
     register: some_product
```

Note:

> "delete action perform a DELETE call to the api and removes the entity from the system, while unregister action only removes it from runtime registry"

OTHER

This are other built-in actions

echo

Outputs a message to the LOG and also to stdout.

Required variables:

- **log**: The message to be logged

Examples:

```
actions:
   - action: echo
     log: Hello World
   - action: echo
     log: This an error
     level: error
   - action echo
     log: This message goes also to the stdout
     print: true
   - action: echo
     log: I can read variables, you are {{ env.USER }}
```

Which outputs:

```
2017-01-20 00:10:53 - satellite_populate.base - INFO - ECHO: Hello World
2017-01-20 00:10:53 - satellite_populate.base - ERROR - ECHO: This an error
2017-01-20 00:10:53 - satellite_populate.base - INFO - ECHO: This message goes also␣
↪to the stdout
This message goes also to the stdout
2017-01-20 00:10:53 - satellite_populate.base - INFO - ECHO: I can read variables,␣
↪you are root
```

register

Register variables to the runtime registry

Required variables:

- **data**: A dictionary

Examples:

```
- action: register
  data:
    name: Michael Scott
    preferred_organization:
      from_search:
        model: Organization
        data:
          name: My prefered Organization
    repo_url:
      from_object: "http://" + file.constants.REPO_BASE_URL
```

All variables registered above will be available for the next executed actions.

unregister

Removes variables from runtime register.

Required variables:

- **data**: A list of variable names

Examples:

```
- action: unregister
  data:
    - name
    - preferred_organization
    - repo_url
```

All variables unregistered above will be not available for the next executed actions.

Unregister is useful for actions using *when:* conditions.

assertion

Execute predefined assertion operations and fails the validation if assertion returns False.

Required variables:

- **operator**: Logical operator mapped to a function returning Boolean
- **data**: A list of two elements to be tested

Built in operators:

- eq # the default

- ne
- gt
- lt
- gte
- lte
- identity

Examples:

```
- action: assertion
  log: Check if current user is root
  operator: eq
  data:
    - root
    - "{{ env.USER }}"
```

If returns False, the validation ends with exit code 1

Custom Populators can also include custom operators for assertion.

CUSTOM

And you can also have special actions defined in a custom populator.

Lets say you have this python module in your project, properly available on PYTHONPATH:

```
from satellite_populate.api import APIPopulator

class MyPopulator(APIPopulator):
    def action_writeinfile(self, rendered_data, action_data):
        with open(rendered_data['path'], 'w') as output:
            output.write(rendered_data['content'])
```

Now go to your `test.yaml` and write:

```
config:
  populator: mine
  populators:
    mine:
      module: mypath.mymodule.MyPopulator

actions:

  - action: writeinfile
    path: /tmp/test.txt
    content: Hello World!!!
```

and run:

$ satellite-populate test.yaml -v

Dynamic Data

There are some ways to fetch dynamic data in action definitions, it depends on the action type.

For any key you can use `Jinja` to provide a dynamic value as in:

```
value: "{{ get_something }}"
value: "{{ fauxfactory.gen_string('alpha') }}"
value: user_{{ item }}
```

For some actions you can provide a `data` key, that data is used to create new entities and also to perform searches or build the action function.

Every `data` key accepts 4 special reference directives in its sub-keys.

- from_registry

  Gets anything from registry:

  ```
  data:
    organization:
      from_registry: default_org
    name:
      from_registry: my_name
  ```

- from_object

  Gets any Python object available in the environment:

  ```
  data:
    url:
      from_object:
        name: robottelo.constants.FAKE_0_YUM_REPO
  ```

- from_search

  Perform a search and return its result:

  ```
  data:
    organization:
      from_search:
        model: Organization
        data:
          name: Default Organization
  ```

- from_read

  Perform a read operation, which is useful when we have unique data or id:

  ```
  data:
    organization:
      from_read:
        model: Organization
        data:
          id: 1
  ```

The internal registry

Every action which returns a result can write its result to the registry, so it is available to be accessed by other actions.

Provide a `register` unique name in `action` definition.

The actions that support `register` are:

- create
- update
- register
- assertion

All dynamic directives `from_*` supports the use of `register`

Example:

```
- action: create
  model: Organization
  register: my_org
  data:
    name: my_org

- model: User
  log: Creating user under {{ register.my_org.name }}
  data:
    organization:
      from_registry: my_org
```

CHAPTER 3

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/SatelliteQE/satellite-populate/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

satellite-populate could always use more documentation, whether as part of the official satellite-populate docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/SatelliteQE/satellite-populate/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up *satellite_populate* for local development.

1. Fork the *satellite-populate* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/satellite-populate.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv satellite-populate
$ cd satellite-populate/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 satellite-populate tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, 3.4 and 3.5, and for PyPy. Check https://travis-ci.org/SatelliteQE/satellite-populate/pull_requests and make sure that the tests pass for all supported Python versions.

Tips

To run a subset of tests:

```
$ py.test tests.test_satellite_populate
```

CHAPTER 4

History

0.1.3 (2017-01-13)

- Docker support

0.1.2 (2017-01-12)

- Fix decorators.

0.1.0 (2017-01-10)

- First release on PyPI.

CHAPTER 5

satellite_populate

satellite_populate package

Submodules

satellite_populate.api module

Implements API populator using Nailgun

class satellite_populate.api.**APIPopulator**(*data*, *verbose=None*, *mode=None*, *config=None*)
Bases: *satellite_populate.base.BasePopulator*

Populates system using API/Nailgun

action_create(*rendered_action_data*, *action_data*, *search*, *model*, *silent_errors*)
Creates new entity if does not exists or get existing entity and return Entity object

action_delete(*rendered_action_data*, *action_data*, *search*, *model*, *silent_errors*)
Deletes an existing entity

action_update(*rendered_action_data*, *action_data*, *search*, *model*, *silent_errors*)
Updates an existing entity

add_and_log_error(*action_data*, *rendered_action_data*, *search*, *e=None*)
Add to validation errors and outputs error

populate(*rendered_action_data*, *action_data*, *search*, *action*)
Populates the System using Nailgun based on value provided in *action* argument gets the proper CRUD method to execute dynamically

validate(*rendered_action_data*, *action_data*, *search*, *action*)
Based on action fields or using action_data['search_query'] searches the system and validates the existence of all entities

satellite_populate.assertion_operators module

Implement basic assertions to be used in assertion action

satellite_populate.assertion_operators.**eq**(*value*, *other*)
Equal

satellite_populate.assertion_operators.**gt**(*value*, *other*)
Greater than

satellite_populate.assertion_operators.**gte**(*value*, *other*)
Greater than or equal

satellite_populate.assertion_operators.**identity**(*value*, *other*)
Identity check using ID

satellite_populate.assertion_operators.**lt**(*value*, *other*)
Lower than

satellite_populate.assertion_operators.**lte**(*value*, *other*)
Lower than or equal

satellite_populate.assertion_operators.**ne**(*value*, *other*)
Not equal

satellite_populate.base module

Base module for satellite_populate reads the YAML definition and perform all the rendering and basic actions.

class satellite_populate.base.**BasePopulator**(*data*, *verbose=None*, *mode=None*, *config=None*)
Bases: object

Base class for API and CLI populators

action_assertion(*rendered_action_data*, *action_data*)
Run assert operations

action_echo(*rendered_action_data*, *action_data*)
After message is echoed to log, check if needs print

action_register(*rendered_action_data*, *action_data*)
Register arbitrary items to the registry

action_unregister(*rendered_action_data*, *action_data*)
Remove data from registry

add_modules_to_context()
Add modules dynamically to render context

add_rendered_action(*action_data*, *rendered_action_data*)
Add rendered action to be written in validation file

add_to_registry(*action_data*, *result*, *append=True*)
Add objects to the internal registry

build_raw_query(*data*, *action_data*)
Builds nailgun raw_query for search

build_search(*rendered_action_data*, *action_data*, *context=None*)
Build search data and returns a dict containing elements

- data Dictionary of parsed rendered_action_data to be used to i nstantiate an object to searched without raw_query.
- options if *search_options* are specified it is passed to *.search(**options)*
- searchable Returns boolean True if model inherits from EntitySearchMixin, else alternative search must be implemented.

if *search_query* is available in action_data it will be used instead of rendered_action_data.

build_search_options(*data*, *action_data*)
Builds nailgun options for search raw_query: Some API endpoints demands a raw_query, so build it as in example: *{'query': {'search':'name=name,label=label,id=28'}}*

force_raw: Returns a boolean if action_data.force_raw is explicitly specified

config
Return config dynamically because it can be overwritten by user in datafile or by custom populator

crud_actions
Return a list of crud_actions, actions that gets *data* and perform nailgun crud operations so custom populators can overwrite this list to add new crud actions.

execute(*mode=None*)
Iterates the entities property described in YAML file and parses its values, variables and substitutions depending on *mode* execute *populate* or *validate*

from_factory(*action_data*, *context*)
Generates random content using fauxfactory

from_read(*action_data*, *context*)
Gets fields and perform a read to return Entity object used when 'from_read' directive is used in YAML file

from_search(*action_data*, *context*)
Gets fields and perform a search to return Entity object used when 'from_search' directive is used in YAML file

get_search_result(*model*, *search*, *unique=False*, *silent_errors=False*)
Perform a search

load_raw_search_rules()
Reads default search rules then update first with custom populator defined rules and then user defined in datafile.

populate(*rendered_action_data*, *raw_entity*, *search_query*, *action*)
Should be implemented in sub classes

populate_modelname(*rendered_action_data*, *action_data*, *search_query*, *action*)
Example on how to implement custom populate methods e.g: *def populate_organization* This method should take care of all validations and errors.

raw_search_rules
Subclasses of custom populators can extend this rules

render(*action_data*, *action*)
Takes an entity description and strips 'data' out to perform single rendering and also handle repetitions defined in *with_items*

render_action_data(*data*, *context*)
Gets a single action_data and perform inplace template rendering or reference evaluation depending on directive being used.

render_assertion_data(*action_data*, *rendered_action_data*)
Render items on assertion data

resolve_result(*data*, *from_where*, *k*, *v*, *result*)
Used in *from_search* and *from_object* to get specific attribute from object e.g: name. Or to invoke a method when attr is a dictionary of parameters.

set_gpgkey()
Set gpgkey

validate(*rendered_action_data*, *raw_entity*, *search_query*, *action*)
Should be implemented in sub classes

validate_modelname(*rendered_action_data*, *action_data*, *search_query*, *action*)
Example on how to implement custom validate methods e.g:: *def validate_organization* This method should take care of all validations and errors.

satellite_populate.cli module

To be implemented: a populator using CLI

satellite_populate.commands module

This module contains commands to interact with satellite populator and validator.

Commands included:

satellite-populate

A command to populate the system based in an YAML file describing the entities:

```
$ satellite-populate file.yaml -h myhost.com -o /tmp/validation.yaml
```

validate

A command to validate the system based in an validation file generated by the populate or a YAML file with mode: validation:

```
$ satellite-populate /tmp/validation.yaml
```

Use `$ satellite-populate --help` for more info

satellite_populate.commands.**configure**()
Read satellite-populate settings file.

satellite_populate.commands.**execute_populate**(*datafile*, *verbose*, *output*, *mode*, *scheme*, *port*, *hostname*, *username*, *password*, *report=True*, *enable_output=True*)
Populate using the data described in *datafile*:

satellite_populate.constants module

Default base config values

satellite_populate.decorators module

decorators for populate feature

satellite_populate.decorators.**populate_with**(*data*, *context_name=None*, *context_wrapper=<function default_context_wrapper>*, ***extra_options*)

To be used in test cases as a decorator

Having a data_file like:

```
actions:
  - model: Organization
    register: organization_1
    data:
      name: My Org
```

Then you can use in decorators:

```
@populate_with('file.yaml')
def test_case_(self):
    'My Org exists in system test anything here'
```

And getting the populated entities inside the test_case:

```
@populate_with('file.yaml', context_name='my_context')
def test_case_(self, my_context=None):
    assert my_context.organization_1.name == 'My Org'
```

You can also set a customized context wrapper to the context_wrapper argument:

```
def my_custom_context_wrapper(result):
    # create an object using result
    my_context = MyResultContext(result)
    return my_context

@populate_with('file.yaml', context_name='my_context',
               content_wrapper=my_custom_context_wrapper)
def test_case_(self, my_context=None):
    # assert with some expression using my_context object returned
    # my_custom_context_wrapper
    assert some_expression
```

NOTE:

```
That is important that ``context_name`` argument always be declared
using either a default value ``my_context=None`` or handle in
``**kwargs`` Otherwise ``py.test`` may try to use this as a fixture
 placeholder.

if context_wrapper is set to None, my_context will be the pure
unmodified result of populate function.
```

satellite_populate.main module

Point of entry for populate and validate used in scripts

satellite_populate.main.**default_context_wrapper**(*result*)
Takes the result of populator and keeps only useful data e.g. in decorators context.registered_name, context.config.verbose and context.vars.admin_username will all be available.

satellite_populate.main.**get_populator**(*data*, ***kwargs*)
Gets an instance of populator dynamically

satellite_populate.main.**load_data**(*datafile*)
Loads YAML file as a dictionary

satellite_populate.main.**populate**(*data*, ***kwargs*)
Loads and execute populator in populate mode

satellite_populate.main.**save_rendered_data**(*result*, *filepath*)
Save the result of rendering in a new file to be used for validation

satellite_populate.main.**setup_yaml**()
Set YAML to use OrderedDict http://stackoverflow.com/a/8661021

satellite_populate.utils module

class satellite_populate.utils.**SmartDict**(**args*, ***kwargs*)
Bases: dict

A Dict which is accessible via attribute dot notation

copy()

satellite_populate.utils.**format_result**(*result*)
format result to show in logs

satellite_populate.utils.**import_from_string**(*import_name*, **args*, ***kwars*)
Try import string and then try builtins

satellite_populate.utils.**remove_keys**(*data*, **args*, ***kwargs*)
remove keys from dictionary d = {'item': 1, 'other': 2, 'keep': 3} remove_keys(d, 'item', 'other') d -> {'keep': 3} deep = True returns a deep copy of data.

satellite_populate.utils.**remove_nones**(*data*)
remove nones from data

satellite_populate.utils.**set_logger**(*verbose*)
Set logger verbosity used when client is called with -vvvvv

Module contents

This package contains tools to populate and validate the system

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

I

L

N

P

R

S

V