Betelgeuse Documentation

Release 1.10.0

Satellite QE

Feb 03, 2023

Contents

Topics

CHAPTER 1

What is Betelgeuse?

Betelgeuse is a Python program that reads standard Python test cases and generates XML files that are suited to be imported by Polarion importers. Possible generated XML files are:

- Requirement Importer XML
- Test Case Importer XML
- Test Run Importer XML

CHAPTER 2

Quick Start

1. Install betelguese from pypi.

```
$ pip install betelgeuse
```

2. Alternatively you can install from source:

```
$ git clone https://github.com/SatelliteQE/betelgeuse.git
$ cd betelgeuse
$ pip install -e .
```

Note: It is always recommended to use python virtual environment

CHAPTER 3

How it works?

Assuming that you have a `test_user.py` file with the following content:

```python
import entities
import unittest


class EntitiesTest(unittest.TestCase):

    def test_positive_create_user(self):
        user = entities.User(name='David', age=20)
        self.assertEqual(user.name, 'David')
        self.assertEqual(user.age, 20)

    def test_positive_create_car(self):
        car = entities.Car(make='Honda', year=2016)
        self.assertEqual(car.make, 'Honda')
        self.assertEqual(car.year, 2016)
```

Using the example above, Betelgeuse will recognize that there are 2 test cases available, and the following attributes will be derived:

- Title: this attribute will be derived from the name of the test method itself:
  - test_positive_create_user
  - test_positive_create_car
- ID: this attribute will be derived from the concatenation of the *module.test_name* or *module.ClassName.test_name* if the test method is defined within a class. In other words, *the Python import path* will be used to derived the ID. Using our example, the values generated would be:
  - test_user.EntitiesTest.test_positive_create_user
  - test_user.EntitiesTest.test_positive_create_car

By default, the values automatically derived by Betelgeuse are not very flexible, specially in the case when you rename an existing test case or move it to a different class or module. It is recommended, therefore, the use of field list fields to provide a bit more information about the tests.

```
import entities
import unittest


class EntitiesTest(unittest.TestCase):

    def test_positive_create_user(self):
        """Create a new user providing all expected attributes.

        :id: 1d73b8cc-a754-4637-8bae-d9d2aaf89003
        :title: Create a new user providing all expected attributes
        """
        user = entities.User(name='David', age=20)
        self.assertEqual(user.name, 'David')
        self.assertEqual(user.age, 20)
```

Now Betelgeuse can use the `:title:` field to derive a friendlier name for your test (instead of using *test_positive_create_user*) and a specific value for its ID. Other information can also be added to the docstring to provide more information, and this can be handled by adding more fields (named after Polarion fields and custom fields).

Note:

1. Make sure that your `IDs` are indeed unique per test case.
2. You can generate a unique UUID using the following code snippet.

   ```
   import uuid
   uuid.uuid4()
   ```

3.1 How steps and expectedresults work together

Betelgeuse will look for some fields when parsing the test cases but there is an special case: when both `steps` and `expectedresults` are defined together.

Betelgeuse will try to match both and create paired step with an expected result. For example in the following docstring:

```
"""Create a new user providing all expected attributes.

:id: 1d73b8cc-a754-4637-8bae-d9d2aaf89003
:steps: Create an user with name and email
:expectedresults: User is created without any error being raised
"""
```

A pair of `Create an user with name and email` step with `User is created without any error being raised` expected result will be created. If multiple steps and multiple expected is wanted, then a list can be used:

```
"""Create a new user providing all expected attributes.

:id: 1d73b8cc-a754-4637-8bae-d9d2aaf89003
```

(continues on next page)

(continued from previous page)

```
:steps:
    1. Open the user creation page
    2. Fill name and email
    3. Submit the form
:expectedresults:
    1. A page with a form with name and email will be displayed
    2. The fields will be populated with the information filled in
    3. User is created without any error being raised
"""
```

On the above example three pairs will be created. The first will match the first item on `steps` and first item on `expectedresults`, the second pair will be the second item on `steps` and the second item on `expectedresults`, so on and so forth.

Note: If the number of items are not the same, then only one pair will be created. The step will be the HTML generated by the value of `steps` and the expected result will be the HTML generate by the value of `expectedresults`.

CHAPTER 4

Usage Examples

Note: 1. For easy understanding of Betelgeuse, this repository is already included with `sample_project` folder. This folder contains sample tests and XML results which will help in setting up and testing Betelgeuse for your project. The sample commands used below also use this data.

2. Always run the test runner and Betelgeuse on the same directory to make sure that the test run ID mapping works fine. Otherwise Betelgeuse may report ID errors. More info can be found in *test-run command* section

4.1 help command

```
$ betelgeuse --help
```

4.2 requirement command

The `requirement` command generates an XML file suited to be imported by the Requirement XML Importer. It reads the Python test suite source code and generates a XML file with all the information necessary for the Requirement XML Importer.

```
$ betelgeuse requirement \
    --assignee assignee \
    --approver approver1 \
    --approver approver2 \
    sample_project/tests \
    PROJECT \
    betelgeuse-requirements.xml
```

Note: Requirements must be created in order to link test cases to them. Make sure to import the requirements before the test cases.

4.3 test-case command

The `test-case` command generates an XML file suited to be imported by the Test Case XML Importer. It reads the Python test suite source code and generates a XML file with all the information necessary for the Test Case XML Importer.

The `test-case` command requires you to pass:

- The path to the Python test suite source code
- The Polarion project ID
- The output XML file path (it will override if the file already exists)

Note: Even though `--response-property` is optional, it is highly recommended to pass it because will be easier to monitor the importer messages (which is not handled by Betelgeuse).

The example below shows how to run the command:

```
$ betelgeuse test-case \
    --automation-script-format "https://github.com/SatelliteQE/betelgeuse/tree/master/
↪{path}#L{line_number}" \
    sample_project/tests \
    PROJECT \
    betelgeuse-test-cases.xml
```

4.4 test-results command

Gives a nice summary of test cases/results in the given jUnit XML file.

```
$ betelgeuse test-results --path \
sample_project/results/sample-junit-result.xml

Passed: 1
```

4.5 test-run command

The `test-run` command generates an XML file suited to be imported by the Test Run XML importer. It takes:

- A valid xUnit XML file
- A Python test suite where test case IDs can be found

And generates a resulting XML file with all the information necessary for the Test Run XML importer.

The `test-run` command only requires you to pass:

- The path to the xUnit XML file

- The path to the Python test suite source code
- The Polarion user ID
- The Polarion project ID
- The output XML file path (it will override if the file already exists)

Note: Even though `--response-property` is optional, it is highly recommended to pass it because will be easier to monitor the importer messages (which is not handled by Betelgeuse).

The example below shows how to run `test-run` command:

```
$ betelgeuse test-run \
    --response-property property_key=property_value \
    sample_project/results/sample-junit-result.xml \
    sample_project/tests/ \
    testuser \
    PROJECT \
    betelgeuse-test-run.xml
```

Polarion custom fields can be set by using the `--custom-fields` option. There are two ways to define custom fields:

`key=value` format This a shortcut when you want to define plain strings as the value of a custom field.

JSON format This approach suits better when the type of the custom field matters. For example, if a custom field expects a boolean as a value.

Example using `key=value` format:

```
$ betelgeuse test-run \
    --custom-fields arch=x8664 \
    --custom-fields variant=server \
    --response-property property_key=property_value \
    sample_project/results/sample-junit-result.xml \
    sample_project/tests/ \
    testuser \
    PROJECT \
    betelgeuse-test-run.xml
```

Example using JSON format:

```
$ betelgeuse test-run \
    --custom-fields '{"isautomated":"true","arch":"x8664"}' \
    --response-property property_key=property_value \
    sample_project/results/sample-junit-result.xml \
    sample_project/tests/ \
    testuser \
    PROJECT \
    betelgeuse-test-run.xml
```

Warning: Make sure to pass the the custom field ID (same as in Polarion) and its value. Also, pass custom field values as string since they will be converted to XML where there is no type information.

CHAPTER 5

Case Study - A real world sample Test Case

Field list fields can be used to provide more information about a test case. The more information one provides via these fields, the more accurate the data being imported into Polarion. For example:

```python
import entities
import unittest

class EntitiesTest(unittest.TestCase):

    def test_positive_create_user(self):
        """Create a new user providing all expected attributes.

        :id: 1d73b8cc-a754-4637-8bae-d9d2aaf89003
        :expectedresults: User is successfully created
        :requirement: User Management
        :caseautomation: Automated
        :caselevel: Acceptance
        :casecomponent: CLI
        :testtype: Functional
        :caseimportance: High
        :upstream: No
        """
        user = entities.User(name='David', age=20)
        self.assertEqual(user.name, 'David')
        self.assertEqual(user.age, 20)
```

When the above test case is collected, Betelgeuse will make use of all 9 fields provided and generates a more meaningful test case.

Ok, this is cool. But wait, there is more! Betelgeuse will reuse fields defined in different levels, namely:

- function level
- class level
- module level
- package level

This feature can be leveraged to minimize the amount of information that needs to be written for each test case. Since most of the time, test cases grouped in a module usually share the same generic information, one could move most of these fields to the `module` level and every single test case found by Betelgeuse will inherit these attributes. For example:

```
"""Test cases for entities.

:caseautomation: Automated
:casecomponent: CLI
:caseimportance: High
:caselevel: Acceptance
:requirement: User Management
:testtype: functional
:upstream: no
"""

import entities
import unittest


class EntitiesTest(unittest.TestCase):

    def test_positive_create_user(self):
        """Create a new user providing all expected attributes.

        :id: 1d73b8cc-a754-4637-8bae-d9d2aaf89003
        :expectedresults: User is successfully created
        """
        user = entities.User(name='David', age=20)
        self.assertEqual(user.name, 'David')
        self.assertEqual(user.age, 20)


    def test_positive_create_car(self):
        """Create a new car providing all expected attributes.

        :id: 71b9b000-b978-4a95-b6f8-83c09ed39c01
        :caseimportance: Medium
        :expectedresults: Car is successfully created and has no owner
        """
        car = entities.Car(make='Honda', year=2016)
        self.assertEqual(car.make, 'Honda')
        self.assertEqual(car.year, 2016)
```

Now all discovered test cases will inherit the attributes defined at the module level. Furthermore, the test case attributes can be overridden at the *class level* or at the *test case level*. Using the example above, since `test_positive_create_car` has its own *caseimportance* field defined, Betelgeuse will use its value of *Medium* for this test case alone while all other test cases will have a value of *High*, derived from the module.

Betelgeuse is able to handle `pytest` parametrized tests and, in order to do so, set the `parametrized` field to `yes` on all tests that make use of the `@pytest.parametrize` decorator or a parametrized fixture. With that, the `test-case` command will generate an XML that instructs the importer to set the test case as being parametrized. And the `test-run` command will generate an XML that instructs the importer to set the result as an iteration result, if it finds a test result with `[<pytest-parameters>]` on its name and the `parametrized` set to `yes` on the source code.

CHAPTER 6

Advanced Usage

Betelgeuse allows configuring the field processing to your own needs, check the Betelgeuse Configuration Module documentation for more information.