Testing using Behave in Python

Overview

In this tutorial, we will create some tests in Behave, which is a Cucumber variant for Python.

The test (specification) is initialy created in Jira as a Cucumber Test and afterwards, it is exported using the UI or the REST API.

We'll show how to use Behave JSON format and also how to generate a Cucumber JSON report, in case you need it.

- Overview
- Requirements
- Code
- Using Cucumber JSON reports
- References

Requirements

- Install Behave
- Install PyHamcrest

Code

features/tutorial01_basics.feature

```
Feature: Showing off behave (tutorial01)
@ABC-119
Scenario: Run a simple test
    Given we have behave installed
    When we implement a test
    Then behave will test it for us!
```

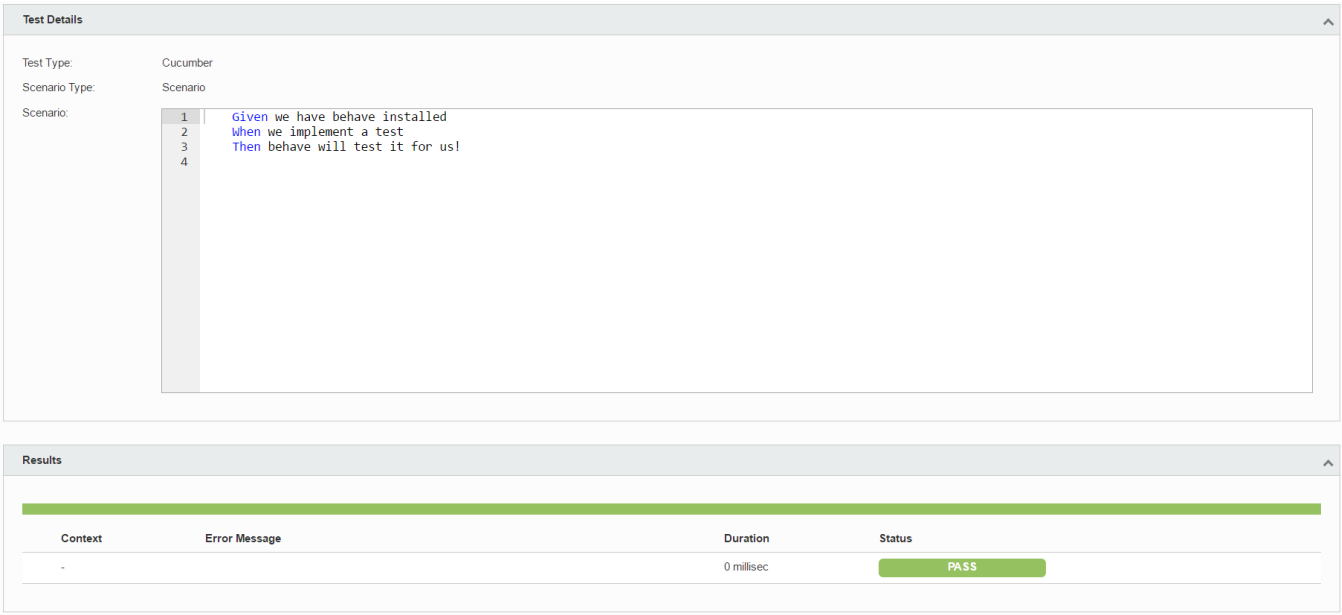
features/steps/step_tutorial01.py

```
# file:features/steps/step_tutorial01.py
# ----------------------------------------------------------------------------
# STEPS:
# ----------------------------------------------------------------------------
from behave import given, when, then
@given('we have behave installed')
def step_impl(context):
    pass
@when('we implement a test')
def step_impl(context):
    assert True is not False
@then('behave will test it for us!')
def step_impl(context):
    assert context.failed is False
```

After running the tests and generating the Behave JSON report (e.g., data.json), it can be imported to Xray via the REST API or the **Import Execution Results** action within the Test Execution.

```
behave --format=json -o data.json
```

The execution details displays the result of the Cucumber Scenario.



> **Learn more**
>
> See the available endpoints for importing Behave's results in Import Execution Results - REST.
>
> Testing in BDD with Gherkin based frameworks (e.g. Cucumber) details the typical workflow for Cucumber-related tests.

Using Cucumber JSON reports

> **Please note**
>
> Cucumber JSON reports are supported by many tools, including some results parsers used by some CI tools. Besides it, as of Xray v3.1, the internal support for Cucumber JSON is more complete giving, for example, the ability to see step level information.

Behave does not provide, as of 2018, the ability to generate compatible Cucumber JSON reports. However, it provides the mechanism to use custom formatters. Thus, we can make our own implementation of a Cucumber JSON formatter.

The following code is based on a sample code provided by an open-source contributor "fredizzimo" (see original code here), with a small changes to make it handle correctly the JSON serialization of status results. You may create this `cucumber_json.py` at the root of your project.

cucumber_json.py

```
# -*- coding: utf-8 -*-

from __future__ import absolute_import
from behave.model_core import Status
from behave.formatter.base import Formatter
import base64
import six
import copy
try:
    import json
except ImportError:
```

```
    import simplejson as json


# -----------------------------------------------------------------------------
# CLASS: JSONFormatter
# -----------------------------------------------------------------------------
class CucumberJSONFormatter(Formatter):
    name = 'json'
    description = 'JSON dump of test run'
    dumps_kwargs = {}

    json_number_types = six.integer_types + (float,)
    json_scalar_types = json_number_types + (six.text_type, bool, type(None))

    def __init__(self, stream_opener, config):
        super(CucumberJSONFormatter, self).__init__(stream_opener, config)
        # -- ENSURE: Output stream is open.
        self.stream = self.open()
        self.feature_count = 0
        self.current_feature = None
        self.current_feature_data = None
        self._step_index = 0
        self.current_background = None
        self.current_background_data = None

    def reset(self):
        self.current_feature = None
        self.current_feature_data = None
        self._step_index = 0
        self.current_background = None

    # -- FORMATTER API:
    def uri(self, uri):
        pass

    def feature(self, feature):
        self.reset()
        self.current_feature = feature
        self.current_feature_data = {
            'id': self.generate_id(feature),
            'uri': feature.location.filename,
            'line': feature.location.line,
            'description': '',
            'keyword': feature.keyword,
            'name': feature.name,
            'tags': self.write_tags(feature.tags),
            'status': feature.status.name,
        }
        element = self.current_feature_data
        if feature.description:
            element['description'] = self.format_description(feature.description)

    def background(self, background):
        element = {
            'type': 'background',
            'keyword': background.keyword,
            'name': background.name,
            'location': six.text_type(background.location),
            'steps': []
        }
        self._step_index = 0
        self.current_background = element

    def scenario(self, scenario):
        if self.current_background is not None:
            self.add_feature_element(copy.deepcopy(self.current_background))
        element = self.add_feature_element({
            'type': 'scenario',
            'id': self.generate_id(self.current_feature, scenario),
            'line': scenario.location.line,
            'description': '',
```

```
        'keyword': scenario.keyword,
        'name': scenario.name,
        'tags': self.write_tags(scenario.tags),
        'location': six.text_type(scenario.location),
        'steps': [],
    })
    if scenario.description:
        element['description'] = self.format_description(scenario.description)
    self._step_index = 0

@classmethod
def make_table(cls, table):
    table_data = {
        'headings': table.headings,
        'rows': [ list(row) for row in table.rows ]
    }
    return table_data

def step(self, step):
    s = {
        'keyword': step.keyword,
        'step_type': step.step_type,
        'name': step.name,
        'line': step.location.line,
        'result': {
            'status': 'skipped',
            'duration': 0
        }
    }

    if step.text:
        s['doc_string'] = {
            'value': step.text,
            'line': step.text.line
        }
    if step.table:
        s['rows'] = [{'cells': [heading for heading in step.table.headings]}]
        s['rows'] += [{'cells': [cell for cell in row.cells]} for row in step.table]

    if self.current_feature.background is not None:
        element = self.current_feature_data['elements'][-2]
        if len(element['steps']) >= len(self.current_feature.background.steps):
            element = self.current_feature_element
    else:
        element = self.current_feature_element
    element['steps'].append(s)

def match(self, match):
    if match.location:
        # -- NOTE: match.location=None occurs for undefined steps.
        match_data = {
            'location': six.text_type(match.location) or "",
        }
        self.current_step['match'] = match_data

def result(self, result):
    self.current_step['result'] = {
        'status': result.status.name,
        'duration': int(round(result.duration * 1000.0 * 1000.0 * 1000.0)),
    }
    if result.error_message and result.status == Status.failed:
        # -- OPTIONAL: Provided for failed steps.
        error_message = result.error_message
        result_element = self.current_step['result']
        result_element['error_message'] = error_message
    self._step_index += 1

def embedding(self, mime_type, data):
    step = self.current_feature_element['steps'][-1]
    step['embeddings'].append({
        'mime_type': mime_type,
```

```
            'data': base64.b64encode(data).replace('\n', ''),
        })

    def eof(self):
        """
        End of feature
        """
        if not self.current_feature_data:
            return

        # -- NORMAL CASE: Write collected data of current feature.
        self.update_status_data()

        if self.feature_count == 0:
            # -- FIRST FEATURE:
            self.write_json_header()
        else:
            # -- NEXT FEATURE:
            self.write_json_feature_separator()

        self.write_json_feature(self.current_feature_data)
        self.current_feature_data = None
        self.feature_count += 1

    def close(self):
        self.write_json_footer()
        self.close_stream()

    # -- JSON-DATA COLLECTION:
    def add_feature_element(self, element):
        assert self.current_feature_data is not None
        if 'elements' not in self.current_feature_data:
            self.current_feature_data['elements'] = []
        self.current_feature_data['elements'].append(element)
        return element

    @property
    def current_feature_element(self):
        assert self.current_feature_data is not None
        return self.current_feature_data['elements'][-1]

    @property
    def current_step(self):
        step_index = self._step_index
        if self.current_feature.background is not None:
            element = self.current_feature_data['elements'][-2]
            if step_index >= len(self.current_feature.background.steps):
                step_index -= len(self.current_feature.background.steps)
                element = self.current_feature_element
        else:
            element = self.current_feature_element

        return element['steps'][step_index]

    def update_status_data(self):
        assert self.current_feature
        assert self.current_feature_data
        self.current_feature_data['status'] = self.current_feature.status.name

    def write_tags(self, tags):
        return [{'name': f'@{tag}', 'line': tag.line if hasattr(tag, 'line') else 1} for tag in tags]

    def generate_id(self, feature, scenario=None):
        def convert(name):
            return name.lower().replace(' ', '-')
        id = convert(feature.name)
        if scenario is not None:
            id += ';'
            id += convert(scenario.name)
        return id
```

```
    def format_description(self, lines):
        description = '\n'.join(lines)
        description = '<pre>%s</pre>' % description
        return description

    # -- JSON-WRITER:
    def write_json_header(self):
        self.stream.write('[\n')

    def write_json_footer(self):
        self.stream.write('\n]\n')

    def write_json_feature(self, feature_data):
        self.stream.write(json.dumps(feature_data, **self.dumps_kwargs))
        self.stream.flush()

    def write_json_feature_separator(self):
        self.stream.write(",\n\n")


# -----------------------------------------------------------------------------
# CLASS: PrettyJSONFormatter
# -----------------------------------------------------------------------------
class PrettyCucumberJSONFormatter(CucumberJSONFormatter):
    """
    Provides readable/comparable textual JSON output.
    """
    name = 'json.pretty'
    description = 'JSON dump of test run (human readable)'
    dumps_kwargs = { 'indent': 2, 'sort_keys': True }
```

In this example, we'll use a `demo.feature` file inspired in two Behave tutorials. The feature file needs to have the proper tags to the Test issue keys and, optionally, to the Test Execution in case you want to enfor the results to be submited to that same Test Executon. You may generate this feature from the UI of the Test Execution issue screen, by using the REST API.

features/demo.feature

```
@CALC-1958
Feature: demo


        @TEST_CALC-1957
        Scenario Outline: Use Blender with <thing>
                    Given I put "<thing>" in a blender
                    When I switch the blender on
                    Then it should transform into "<other thing>"

                    Examples: Amphibians
                        | thing         | other thing |
                        | Red Tree Frog | mush        |
                        | apples        | apple juice |



        @TEST_CALC-1956
        Scenario: Run a simple test
                    Given we have behave installed
                    When we implement a test
                    Then behave will test it for us!
```

The corresponding steps implementation code lives in the following files.

features/steps/blender.py

```
# file:features/steps/blender.py
# ----------------------------------------------------------------------------
# DOMAIN-MODEL:
# ----------------------------------------------------------------------------
class Blender(object):
    TRANSFORMATION_MAP = {
        "Red Tree Frog": "mush",
        "apples": "apple juice",
        "iPhone": "toxic waste",
        "Galaxy Nexus": "toxic waste",
    }
    def __init__(self):
        self.thing  = None
        self.result = None

    @classmethod
    def select_result_for(cls, thing):
        return cls.TRANSFORMATION_MAP.get(thing, "DIRT")

    def add(self, thing):
        self.thing = thing

    def switch_on(self):
        self.result = self.select_result_for(self.thing)
```

features/steps/step_tutorial01.py

```
# file:features/steps/step_tutorial01.py
# ----------------------------------------------------------------------------
# STEPS:
# ----------------------------------------------------------------------------
from behave import given, when, then
@given('we have behave installed')
def step_impl(context):
    pass
@when('we implement a test')
def step_impl(context):
    assert True is not False
@then('behave will test it for us!')
def step_impl(context):
    assert context.failed is False
```

features/steps/step_tutorial03.py

```
# file:features/steps/step_tutorial03.py
# -----------------------------------------------------------------------------
# STEPS:
# -----------------------------------------------------------------------------
from behave   import given, when, then
from hamcrest import assert_that, equal_to
from blender  import Blender

@given('I put "{thing}" in a blender')
def step_given_put_thing_into_blender(context, thing):
    context.blender = Blender()
    context.blender.add(thing)

@when('I switch the blender on')
def step_when_switch_blender_on(context):
    context.blender.switch_on()

@then('it should transform into "{other_thing}"')
def step_then_should_transform_into(context, other_thing):
    assert_that(context.blender.result, equal_to(other_thing))
```

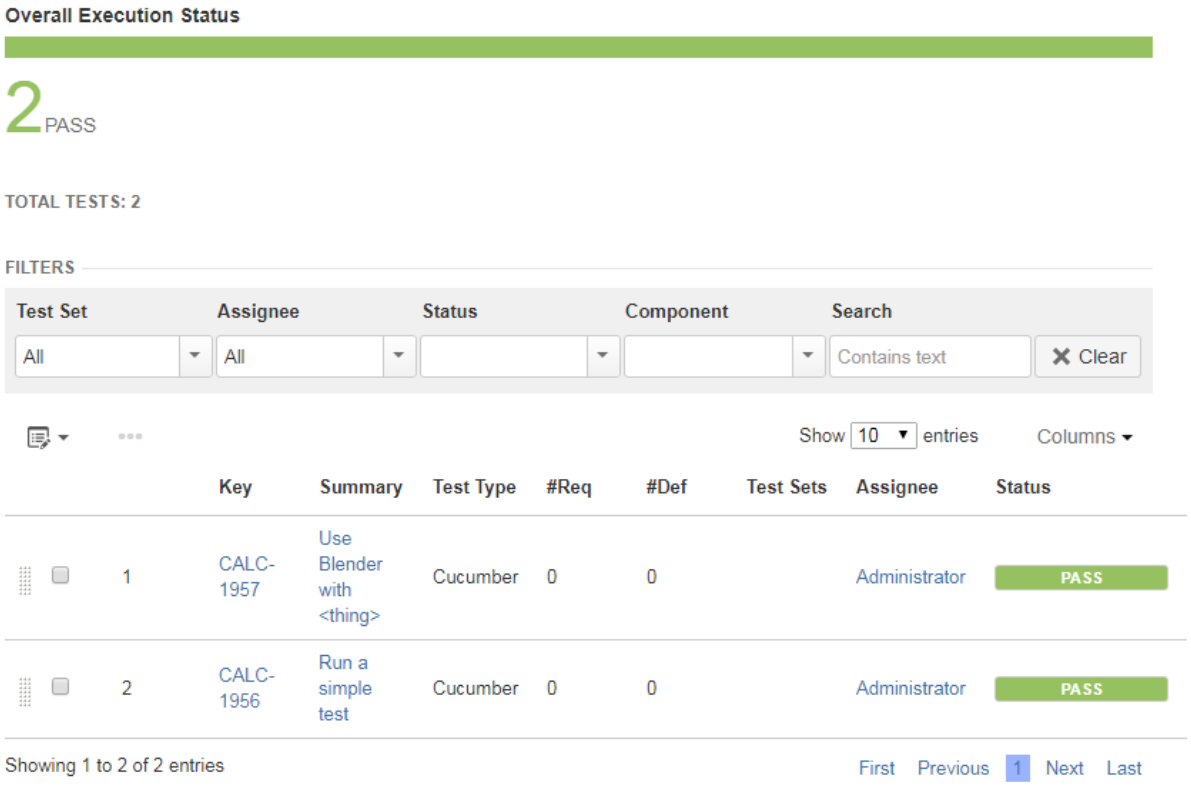
After running the tests and generating the Cucumber JSON report (e.g., cucumber.json), it can be imported to Xray via the REST API or the **Import Execution Results** action within the Test Execution.

Running tests

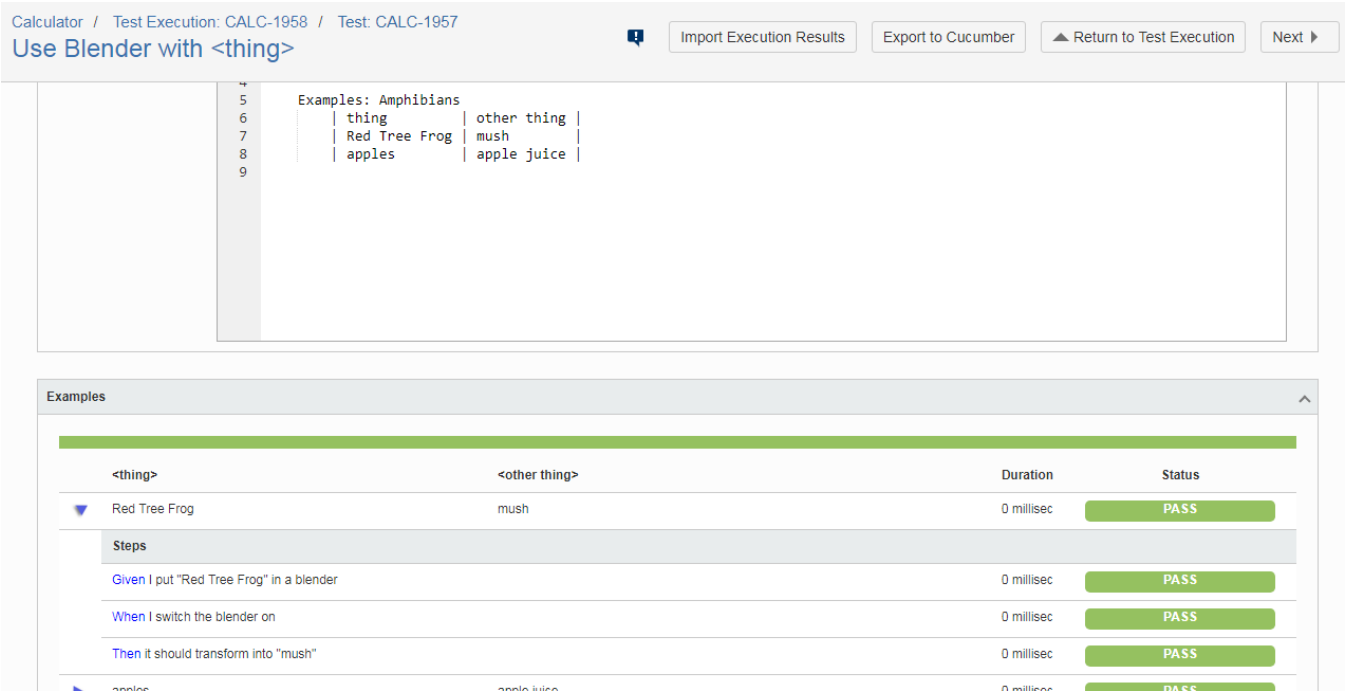
```
behave --format=cucumber_json:PrettyCucumberJSONFormatter -o cucumber.json  --format=json -o behave.json
features/demo.feature
```

Import results via REST API

```
curl -H "Content-Type: application/json" -X POST -u user:password --data @cucumber.json https://sandbox.xpand-
addons.com/rest/raven/1.0/import/execution/cucumber
```



The execution page provides detailed information, which in this case includes the results for the different examples along with the respective step results.



References

- https://behave.readthedocs.io
- https://jenisys.github.io/behave.example/tutorials/tutorial01.html
- https://jenisys.github.io/behave.example/tutorials/tutorial04.html

- https://github.com/behave/behave/issues/267