OpenCAFE Documentation

Release 0.2.0

Rackspace

Mar 27, 2018

Contents

CHAPTER 1

What is OpenCAFE

The Common Automation Framework Engine (CAFE) is the core engine/driver used to build an automated testing framework. It is designed to be used as the base engine for building an automated framework for API and non-UI resource testing. It is designed to support functional, integration and reliability testing. The engine is NOT designed to support performance or load testing.

CAFE core provides models, patterns, and supported libraries for building automated tests. It provides its own lightweight unittest based runner, however, it is designed to be modular. It can be extended to support most test case front ends/runners (nose, pytest, lettuce, testr, etc. . .) through driver plug-ins.

Note: Questions? Join us on Freenode in the #cafehub channel

CHAPTER 2

Relevant Links

- GitHub: Repository
- CI: *Coming soon*
- Coverage: *Coming soon*

CHAPTER 3

Contents

3.1 Quickstart

The goals of this section of the guide are to provide a detailed explaination of how to use some of the basic OpenCafe components, as well as to explain what types of problems are solved by using OpenCafe. Further details on each component can be found in the rest of the documentation.

3.1.1 Installation

To get started, We should install and initialize OpenCafe. We can do this by running the following commands:

```
pip install opencafe
cafe-config init
```

3.1.2 Making HTTP Requests

As an example, we can write a few tests for the GitHub API. For the sake of this example, we will assume that we don't have language bindings or SDKs for our API which is common for APIs in development. We can create a simple client using the BaseHTTPClient class provided by the OpenCafe HTTP plugin, which is a lightweight wrapper for the `requests` package. First, we'll need to install the HTTP plugin:

```
cafe-config plugin install http
```

Now we can create a simple python script to request the details of a GitHub issue:

```
import json
import os

from cafe.engine.http.client import BaseHTTPClient

# Opencafe normally expects for a configuration data file to be set beforeit is
```

```
# run. For these examples this isn't necessary, but the value still needs to be set.
# This is behavior that we should fix, but hasn't been at the time this guide was␣
↪written
os.environ['CAFE_ENGINE_CONFIG_FILE_PATH']='.'
client = BaseHTTPClient()
response = client.get('https://api.github.com/repos/cafehub/opencafe/issues/42')
print json.dumps(response.json(), indent=2)
```

This will generate some warnings that we will address later, but the outcome should be similar to the following:


```
{
    "labels": [],
    "number": 42,
    "assignee": null,
    "repository_url": "https://api.github.com/repos/CafeHub/opencafe",
    "closed_at": "2017-02-28T16:32:28Z",
    "id": 210568122,
    "title": "Adds a Travis CI config to run tox",
    "pull_request": {
        "url": "https://api.github.com/repos/CafeHub/opencafe/pulls/42",
        "diff_url": "https://github.com/CafeHub/opencafe/pull/42.diff",
        "html_url": "https://github.com/CafeHub/opencafe/pull/42",
        "patch_url": "https://github.com/CafeHub/opencafe/pull/42.patch"
    },
    "comments": 0,
    "state": "closed",
    "body": "",
    "labels_url": "https://api.github.com/repos/CafeHub/opencafe/issues/42/labels{/
↪name}",
    "events_url": "https://api.github.com/repos/CafeHub/opencafe/issues/42/events",
    "comments_url": "https://api.github.com/repos/CafeHub/opencafe/issues/42/comments
↪",
    "html_url": "https://github.com/CafeHub/opencafe/pull/42",
    "updated_at": "2017-02-28T16:32:28Z",
    "user": {
        "following_url": "https://api.github.com/users/dwalleck/following{/other_user}
↪",
        "events_url": "https://api.github.com/users/dwalleck/events{/privacy}",
        "organizations_url": "https://api.github.com/users/dwalleck/orgs",
        "url": "https://api.github.com/users/dwalleck",
        "gists_url": "https://api.github.com/users/dwalleck/gists{/gist_id}",
        "html_url": "https://github.com/dwalleck",
        "subscriptions_url": "https://api.github.com/users/dwalleck/subscriptions",
        "avatar_url": "https://avatars2.githubusercontent.com/u/843116?v=3",
        "repos_url": "https://api.github.com/users/dwalleck/repos",
        "received_events_url": "https://api.github.com/users/dwalleck/received_events
↪",
        "gravatar_id": "",
        "starred_url": "https://api.github.com/users/dwalleck/starred{/owner}{/repo}",
        "site_admin": false,
        "login": "dwalleck",
        "type": "User",
        "id": 843116,
        "followers_url": "https://api.github.com/users/dwalleck/followers"
    },
    "milestone": null,
    "closed_by": {
        "following_url": "https://api.github.com/users/jidar/following{/other_user}",
```

```
        "events_url": "https://api.github.com/users/jidar/events{/privacy}",
        "organizations_url": "https://api.github.com/users/jidar/orgs",
        "url": "https://api.github.com/users/jidar",
        "gists_url": "https://api.github.com/users/jidar/gists{/gist_id}",
        "html_url": "https://github.com/jidar",
        "subscriptions_url": "https://api.github.com/users/jidar/subscriptions",
        "avatar_url": "https://avatars2.githubusercontent.com/u/1134139?v=3",
        "repos_url": "https://api.github.com/users/jidar/repos",
        "received_events_url": "https://api.github.com/users/jidar/received_events",
        "gravatar_id": "",
        "starred_url": "https://api.github.com/users/jidar/starred{/owner}{/repo}",
        "site_admin": false,
        "login": "jidar",
        "type": "User",
        "id": 1134139,
        "followers_url": "https://api.github.com/users/jidar/followers"
    },
    "locked": false,
    "url": "https://api.github.com/repos/CafeHub/opencafe/issues/42",
    "created_at": "2017-02-27T18:34:21Z",
    "assignees": []
}
```

The BaseHTTPClient returns the same response that `requests` would, so we can treat the response similarly to view its content. At this point, it doesn't look like the OpenCafe HTTP plugin is adding any more value than `requests` would. Let's see what we can do about that. First, let's setup logging and see what happens.

```
import json
import logging
import os
import sys

from cafe.engine.http.client import BaseHTTPClient
from cafe.common.reporting import cclogging


os.environ['CAFE_ENGINE_CONFIG_FILE_PATH']='.'
cclogging.init_root_log_handler()
root_log = logging.getLogger()
root_log.addHandler(logging.StreamHandler(stream=sys.stderr))
root_log.setLevel(logging.DEBUG)

client = BaseHTTPClient()
response = client.get('https://api.github.com/repos/cafehub/opencafe/issues/42')
```

The `cclogging` package simplifies parts of working with the standard Python logger, such as creating and initializing a logger. With logging enabled, let's execute our script again to see the difference.

```
(cafe-demo) dwalleck@MINERVA:~$ python demo.py
Environment variable 'CAFE_MASTER_LOG_FILE_NAME' is not set. A null root log handler␣
↪will be used, no logs will be written.(<cafe.engine.http.client.BaseHTTPClient␣
↪object at 0x7fd2a58cf550>, 'GET', 'https://api.github.com/repos/cafehub/opencafe/
↪issues/42') {}
No section: 'PLUGIN.HTTP'.  Using default value '0' instead
Starting new HTTPS connection (1): api.github.com
https://api.github.com:443 "GET /repos/cafehub/opencafe/issues/42 HTTP/1.1" 200 None

------------
```

```
REQUEST SENT
------------
request method..: GET
request url.....: https://api.github.com/repos/cafehub/opencafe/issues/42
request params..:
request headers.: {'Connection': 'keep-alive', 'Accept-Encoding': 'gzip, deflate',
↪'Accept': '*/*', 'User-Agent': 'python-requests/2.13.0'}
request body....: None


-----------------
RESPONSE RECEIVED
-----------------
response status..: <Response [200]>
response time....: 0.35421204567
response headers.: {'X-XSS-Protection': '1; mode=block', 'Content-Security-Policy':
↪"default-src 'none'", 'Access-Control-Expose-Headers': 'ETag, Link, X-GitHub-OTP, X-
↪RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset, X-OAuth-Scopes, X-
↪Accepted-OAuth-Scopes, X-Poll-Interval', 'Transfer-Encoding': 'chunked', 'Last-
↪Modified': 'Thu, 13 Apr 2017 19:13:26 GMT', 'Access-Control-Allow-Origin': '*', 'X-
↪Frame-Options': 'deny', 'Status': '200 OK', 'X-Served-By':
↪'eef8b8685a106934dcbb4b7c59fba0bf', 'X-GitHub-Request-Id':
↪'FA86:30F6:B12CE5:ED8475:58F8F029', 'ETag': 'W/"2fbeb849316f7b18e9138ea40d150441"',
↪'Date': 'Thu, 20 Apr 2017 17:30:17 GMT', 'X-RateLimit-Remaining': '59', 'Strict-
↪Transport-Security': 'max-age=31536000; includeSubdomains; preload', 'Server':
↪'GitHub.com', 'X-GitHub-Media-Type': 'github.v3; format=json', 'X-Content-Type-
↪Options': 'nosniff', 'Content-Encoding': 'gzip', 'Vary': 'Accept, Accept-Encoding',
↪'X-RateLimit-Limit': '60', 'Cache-Control': 'public, max-age=60, s-maxage=60',
↪'Content-Type': 'application/json; charset=utf-8', 'X-RateLimit-Reset': '1492713017
↪'}
response body....: {"url":"https://api.github.com/repos/CafeHub/opencafe/issues/42",
↪"repository_url":"https://api.github.com/repos/CafeHub/opencafe","labels_url":
↪"https://api.github.com/repos/CafeHub/opencafe/issues/42/labels{/name}","comments_
↪url":"https://api.github.com/repos/CafeHub/opencafe/issues/42/comments","events_url
↪":"https://api.github.com/repos/CafeHub/opencafe/issues/42/events","html_url":
↪"https://github.com/CafeHub/opencafe/pull/42","id":210568122,"number":42,"title":
↪"Adds a Travis CI config to run tox","user":{"login":"dwalleck","id":843116,"avatar_
↪url":"https://avatars2.githubusercontent.com/u/843116?v=3","gravatar_id":"","url":
↪"https://api.github.com/users/dwalleck","html_url":"https://github.com/dwalleck",
↪"followers_url":"https://api.github.com/users/dwalleck/followers","following_url":
↪"https://api.github.com/users/dwalleck/following{/other_user}","gists_url":"https://
↪api.github.com/users/dwalleck/gists{/gist_id}","starred_url":"https://api.github.
↪com/users/dwalleck/starred{/owner}{/repo}","subscriptions_url":"https://api.github.
↪com/users/dwalleck/subscriptions","organizations_url":"https://api.github.com/users/
↪dwalleck/orgs","repos_url":"https://api.github.com/users/dwalleck/repos","events_url
↪":"https://api.github.com/users/dwalleck/events{/privacy}","received_events_url":
↪"https://api.github.com/users/dwalleck/received_events","type":"User","site_admin
↪":false},"labels":[],"state":"closed","locked":false,"assignee":null,"assignees":[],
↪"milestone":null,"comments":0,"created_at":"2017-02-27T18:34:21Z","updated_at":
↪"2017-02-28T16:32:28Z","closed_at":"2017-02-28T16:32:28Z","pull_request":{"url":
↪"https://api.github.com/repos/CafeHub/opencafe/pulls/42","html_url":"https://github.
↪com/CafeHub/opencafe/pull/42","diff_url":"https://github.com/CafeHub/opencafe/pull/
↪42.diff","patch_url":"https://github.com/CafeHub/opencafe/pull/42.patch"},"body":"",
↪"closed_by":{"login":"jidar","id":1134139,"avatar_url":"https://avatars2.
↪githubusercontent.com/u/1134139?v=3","gravatar_id":"","url":"https://api.github.com/
↪users/jidar","html_url":"https://github.com/jidar","followers_url":"https://api.
↪github.com/users/jidar/followers","following_url":"https://api.github.com/users/
↪jidar/following{/other_user}","gists_url":"https://api.github.com/users/jidar/gists
↪{/gist_id}","starred_url":"https://api.github.com/users/jidar/starred{/owner}{/repo}
↪","subscriptions_url":"https://api.github.com/users/jidar/subscriptions",
↪"organizations_url":"https://api.github.com/users/jidar/orgs","repos_url":"https://
↪api.github.com/users/jidar/repos","events_url":"https://api.github.com/users/jidar/
↪events{/privacy}","received_events_url":"https://api.github.com/users/jidar/
↪received_events","type":"User","site_admin":false}}
```

```
------------------------------------------------------------------------------
```

That's a little better. We get a verbose log entry for the details of request made and the response we received. The output from the HTTP client is meant to be human readable and to create an audit trail of what occurred while a test or script was executed.

3.1.3 Creating a Basic Application Client

Now let's add a few more requests to our script:

```
import json
import logging
import os
import sys

from cafe.engine.http.client import BaseHTTPClient
from cafe.common.reporting import cclogging


os.environ['CAFE_ENGINE_CONFIG_FILE_PATH'] ='.'
cclogging.init_root_log_handler()
root_log = logging.getLogger()
root_log.addHandler(logging.StreamHandler(stream=sys.stderr))
root_log.setLevel(logging.DEBUG)


client = BaseHTTPClient()
response = client.get('https://api.github.com/repos/cafehub/opencafe/issues/42')
response = client.get('https://api.github.com/repos/cafehub/opencafe/commits')
response = client.get('https://api.github.com/repos/cafehub/opencafe/forks')
```

As we make more requests, a few concerns come to mind. Right now we are hard-coding the base url (https://api.github.com) in each request. The organization and project names are both something that could change. At the very least, we should factor out what is common between the requests or what is likely to change as we grow this script:

```
import json
import logging
import os
import sys

from cafe.engine.http.client import BaseHTTPClient
from cafe.common.reporting import cclogging


os.environ['CAFE_ENGINE_CONFIG_FILE_PATH'] ='.'
cclogging.init_root_log_handler()
root_log = logging.getLogger()
root_log.addHandler(logging.StreamHandler(stream=sys.stderr))
root_log.setLevel(logging.DEBUG)

client = BaseHTTPClient()
base_url = 'https://api.github.com'
organization = 'cafehub'
project = 'opencafe'
issue_id = 42
```

```
response = client.get(
    '{base_url}/repos/{org}/{project}/commits'.format(
        base_url=base_url, org=organization, project=project))

response = client.get(
    '{base_url}/repos/{org}/{project}/issues/{issue_id}'.format(
        base_url=base_url, org=organization, project=project,
        issue_id=issue_id))

response = client.get(
    '{base_url}/repos/{org}/{project}/forks'.format(
        base_url=base_url, org=organization, project=project))
```

The GitHub API is expansive, so we could go on for some time defining more requests. Rather than defining these in-line, defining these functions in a common class would make more sense from an organization sense.

```
import json
import logging
import os
import sys

from cafe.engine.clients.base import BaseClient
from cafe.engine.http.client import BaseHTTPClient
from cafe.common.reporting import cclogging


class GitHubClient(BaseHTTPClient):

    def __init__(self, base_url):
        super(GitHubClient, self).__init__()
        self.base_url = base_url

    def get_project_commits(self, org_name, project_name):
        return self.get(
            '{base_url}/repos/{org}/{project}/commits'.format(
                base_url=base_url, org=org_name, project=project))

    def get_issue_by_id(self, org_name, project_name, issue_id):
        return self.get(
            '{base_url}/repos/{org}/{project}/issues/{issue_id}'.format(
                base_url=base_url, org=org_name, project=project,
                issue_id=issue_id))


    def get_project_forks(self, org_name, project_name):
        return self.get(
            '{base_url}/repos/{org}/{project}/forks'.format(
                base_url=base_url, org=org_name, project=project))

os.environ['CAFE_ENGINE_CONFIG_FILE_PATH']='.'
cclogging.init_root_log_handler()
root_log = logging.getLogger()
root_log.addHandler(logging.StreamHandler(stream=sys.stderr))
root_log.setLevel(logging.DEBUG)

base_url = 'https://api.github.com'
organization = 'cafehub'
project = 'opencafe'
issue_id = 42
```

```
client = GitHubClient(base_url)

resp1 = client.get_project_commits(org_name=organization, project_name=project)
resp2 = client.get_issue_by_id(org_name=organization, project_name=project, issue_
↪id=issue_id)
resp3 = client.get_project_forks(org_name=organization, project_name=project)
```

Our client subclasses the BaseHTTPClient, so there's no longer a need to create an instance of the client. This creates the foundation for a simple language binding for our API under test.

Now that our HTTP requests are in better shape, let's talk about dealing with the responses. The `requests` response object has a `json` method that will transform the body of the response into a Python dictionary. While treating the response content as a dictionary is good enough for quick scripts and possibly for working with very stable APIs, it has challenges that we should consider before going further.

Accessing the response as a dictionary isn't too difficult when a response body has one or two properties, but let's jump back to the first response output we looked at. It has dozens of properties, including ones that are nested. Using the response as-is requires memorizing the response structure or constantly referencing API documentation as you code. If you make a mistake with the name of a property, you may not find that out until you run the code. Also, when the name of one of the properties or the structure of the API response changes, this means tediously changing the property each place it is used or trying to do a string replace across the project, which is an error-prone process.

3.1.4 Writing Request and Response Models

An alternate approach is to deserialize the JSON response to an object. This is the approach that most SDKs and language bindings use. This greatly simplifies refactoring of response properties and has the added bonus of error detection by linters if you use an invalid property name. If you're using a code editor which offers autocomplete functionality, you can also use that when developing new tests, which removes most of the need to reference API documentation after you've done the groundwork developing the response models. Here's an example of what the response model for our first request would look like:

```
class Issue(AutoMarshallingModel):

    def __init__(self, url, repository_url, labels_url, comments_url, events_url,
                 html_url, id, number, title, user, labels, state, locked,
                 assignee, assignees, milestone, comments, created_at,
                 updated_at, closed_at, body, closed_by):


        self.url = url
        self.repository_url = repository_url
        self.labels_url = labels_url
        self.comments_url = comments_url
        self.events_url = events_url
        self.html_url = html_url
        self.id = id
        self.number = number
        self.title = title
        self.user = user
        self.labels = labels
        self.state = state
        self.locked = locked
        self.assignee = assignee
        self.assignees = assignees
        self.milestone = milestone
        self.comments = comments
        self.created_at = created_at
```

```
        self.updated_at = updated_at
        self.closed_at = closed_at
        self.body = body
        self.closed_by = closed_by

    @classmethod
    def _json_to_obj(cls, serialized_str):
        resp_dict = json.loads(serialized_str)
        user = User(**resp_dict.get('user'))

        assignees = []
        for assignee in resp_dict.get('assignees'):
            assignees.append(User(**assignee))

        assignee = None
        if resp_dict.get('assignee'):
            assignee = User(**resp_dict.get('assignee'))

        labels = []
        for label in labels:
            labels.append(Label(**label))

        return Issue(
            url=resp_dict.get('url'),
            repository_url=resp_dict.get('repository_url'),
            labels_url=resp_dict.get('labels_url'),
            comments_url=resp_dict.get('comments_url'),
            events_url=resp_dict.get('events_url'),
            html_url=resp_dict.get('html_url'),
            id=resp_dict.get('id'),
            number=resp_dict.get('number'),
            title=resp_dict.get('title'),
            user=user,
            labels=labels,
            state=resp_dict.get('state'),
            locked=resp_dict.get('locked'),
            assignee=assignee,
            assignees=assignees,
            milestone=resp_dict.get('milestone'),
            comments=resp_dict.get('comments'),
            created_at=resp_dict.get('created_at'),
            updated_at=resp_dict.get('updated_at'),
            closed_at=resp_dict.get('closed_at'),
            body=resp_dict.get('body'),
            closed_by=resp_dict.get('closed_by'))


class User(AutoMarshallingModel):

    def __init__(self, login, id, avatar_url, gravatar_id, url, html_url,
                 followers_url, following_url, gists_url, starred_url,
                 subscriptions_url, organizations_url, repos_url, events_url,
                 received_events_url, type, site_admin):

        self.login = login
        self.id = id
        self.avatar_url = avatar_url
        self.gravatar_id = gravatar_id
```

```
        self.url = url
        self.html_url = html_url
        self.followers_url = followers_url
        self.following_url = following_url
        self.gists_url = gists_url
        self.starred_url = starred_url
        self.subscriptions_url = subscriptions_url
        self.organizations_url = organizations_url
        self.repos_url = repos_url
        self.events_url = events_url
        self.received_events_url = received_events_url
        self.type = type
        self.site_admin = site_admin

    @classmethod
    def _json_to_obj(cls, serialized_str):
        resp_dict = json.loads(serialized_str)
        return User(**resp_dict)


class Label(AutoMarshallingModel):

    def __init__(self, id, url, name, color, default):

        self.id = id
        self.url = url
        self.name = name
        self.color = color
        self.default = default

    @classmethod
    def _json_to_obj(cls, serialized_str):
        resp_dict = json.loads(serialized_str)
        return Label(**resp_dict)
```

Any class that inherits from the AutoMarshallingModel class is expected to implement the _json_to_obj method, _obj_to_json method, or both. This depends on whether the model is being used to handle requests, responses, or both.

This example creates quite a bit of boilerplate code. We used an explicit example so that it would be easy to understand what this code does. However, because these objects are explicitly defined, static analysis tools will be able to assist us going forward. It also allows code editors that support Python autocompletion to work with our models. In more practical implementations, you may want to take advantage of Python's dynamic nature to simplify the setting of properties.

3.1.5 Writing an Auto-Serializing Client

Now that we have response models, we can refactor our client to use them.

```
from cafe.engine.http.client import AutoMarshallingHTTPClient


class GitHubClient(AutoMarshallingHTTPClient):

    def __init__(self, base_url):
        super(GitHubClient, self).__init__(
```

```
            serialize_format='json', deserialize_format='json')
        self.base_url = base_url

    def get_issue_by_id(self, org_name, project_name, issue_id):
        url = '{base_url}/repos/{org}/{project}/issues/{issue_id}'.format(
            base_url=self.base_url, org=organization, project=project,
            issue_id=issue_id)
        return self.get(url, response_entity_type=Issue)
```

There's a few changes to note. The AutoMarshallingHTTPClient class replaces BaseHTTPClient as the parent class because it is aware of request and response content types. The response_entity_type parameter defines what type to expect the response to be. This together with serialization formats set when the client was instantiated determine which serialization methods are called on the response contents. This can be used to create a single API client that can handle both JSON and XML response types. This can be an extremely useful capability to have when you want to write code a single that is able to test both the JSON and XML capabilities of an API.

3.1.6 Managing Test Data

Before we start writing our tests, let's step back and deal with one more issue. In the original code, we had statically defined certain data such as the GitHub URL, the organization name, and the project name. There are many reasons why you should not hardcode these types of values in your code. Of those, the most important to us is that we should not have to make code changes whenever we want to use different test data. We should be able to provide the test data at runtime, which allows our code to be more flexible and portable.

There are many sources we could use for our test data, but for this example we will use a plain text file with headers that can be parsed by Python's `SafeConfigParser`. For this to work, we will need to create a class that represents the data that we want to store in the file.

```
from cafe.engine.models.data_interfaces import ConfigSectionInterface


class GitHubConfig(ConfigSectionInterface):

    SECTION_NAME = 'GitHub'

    @property
    def base_url(self):
        return self.get('base_url')

    @property
    def organization(self):
        return self.get('organization')

    @property
    def project(self):
        return self.get('project')

    @property
    def issue_id(self):
        return self.get('issue_id')
```

Note that there is nothing in this class that explicitly states the type of the data source. This is because the OpenCafe `data_interfaces` package provides a generic interface for data sources including environment variables and JSON data. For the purpose of this guide, we will just use plain text files.

Our class defines that there should have a section titled `GitHub` in our configuration file with four properties. The

actual configuration file would look similar to the following example:

```
[GitHub]
base_url = https://api.github.com
organization = cafehub
project = opencafe
issue_id = 42
```

3.1.7 Writing and Running a Test

From this point in the demo, you can use the opencafe-demo **project to follow along with the guide if you want to execute the steps yourself.**

Now that we have our test infrastructure in order, we can write several tests to see how OpenCafe operates.

```
from cafe.drivers.unittest.fixtures import BaseTestFixture

from opencafe_demo.github.github_client import GitHubClient
from opencafe_demo.github.github_config import GitHubConfig


class BasicGitHubTest(BaseTestFixture):

    @classmethod
    def setUpClass(cls):
        super(BasicGitHubTest, cls).setUpClass()  # Sets up logging/reporting for the␣
↪test
        cls.config_data = GitHubConfig()

        cls.organization = cls.config_data.organization
        cls.project = cls.config_data.project
        cls.issue_id = cls.config_data.issue_id
        cls.client = GitHubClient(cls.config_data.base_url)

    def test_get_issue_response_code_is_200(self):
        response = self.client.get_project_issue(
            self.organization, self.project, self.issue_id)
        self.assertEqual(response.status_code, 200)

    def test_id_is_not_null_for_get_issue_request(self):
        response = self.client.get_project_issue(
            self.organization, self.project, self.issue_id)
        # The response signature is the raw response from Requests except
        # for the `entity` property, which is the object that represents
        # the response content
        issue = response.entity
        self.assertIsNotNone(issue.id)
```

In this test class, we inherit from OpenCafe's `BaseTestFixture` class. This base class automatically handles all of the logging setup that we were previously doing by hand. The `BaseTestFixture` class inherits from Python's `unittest.TestCase`, so for all intents and purposes it behaves the same as any other unittest-based test.

Before we can run this test, we need to get our configuration data file in place. When we executed the `cafe-config init` command at the start of the guide, you may have noticed in the output that several directories were created. You should now have a `.opencafe` directory, which is where all configuration data and test logs will be stored by default (these paths can be changed in the `.opencafe/engine.config` file. See the full documentation for further details). We will need to create a directory named `GitHub` in which we will put our configuration file which

we will call `prod.config`. The names used are arbitrary, but they create a convention that will be used when we begin running our tests.

OpenCafe uses a convention based on `<product-name>` and `<config-file-name>` for finding configuration data and setting logging locations. For configuration files, the `<config-file-name>` file will be loaded from the `.opencafe/configs/<product-name>` directory. For logging, logs for each test run will be saved in a unique directory named by the date time stamp of when the tests were run in the `.opencafe/logs/<product-name>/<config-file-name>` directory.

For this guide, I'll be using OpenCafe's unittest-based runner to execute the tests. All the tests in the `github` project can be run by executing `cafe-runner github prod.config`.

```
(cafe-demo) dwalleck@minerva:~$ cafe-runner github prod.config

    ( (
    ) )
.........
|       |___
|       |_  |
|  :-)  |_| |
|       |___|
|_______|
=== CAFE Runner ===
================================================================================================================
Percolated Configuration
----------------------------------------------------------------------------------------------------
 ↪------------------------------------------------------------
BREWING FROM: ....: /home/dwalleck/cafe-demo/local/lib/python2.7/site-packages/
 ↪opencafe_demo
ENGINE CONFIG FILE: /home/dwalleck/cafe-demo/.opencafe/engine.config
TEST CONFIG FILE..: /home/dwalleck/cafe-demo/.opencafe/configs/github/prod.config
DATA DIRECTORY....: /home/dwalleck/cafe-demo/.opencafe/data
LOG PATH..........: /home/dwalleck/cafe-demo/.opencafe/logs/github/prod.config/2017-
 ↪04-19_11_26_38.698599
================================================================================================================
test_get_issue_response_code_is_200 (opencafe_demo.github.test_issues_api.
 ↪BasicGitHubTest) ... ok
test_id_is_not_null_for_get_issue_request (opencafe_demo.github.test_issues_api.
 ↪BasicGitHubTest) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.543s

OK
================================================================================================================
Detailed logs: /home/dwalleck/cafe-demo/.opencafe/logs/github/prod.config/2017-04-19_
 ↪11_26_38.698599
----------------------------------------------------------------------------------------------------
 ↪------------------------------------------------------------
```

The preamble output from the test runner pretty prints the location of all configuration files used for the test run, as well as the the location of the logs generated during the test run. Here's what the contents of the log directory look like:

```
(cafe-demo) dwalleck@minerva:~$ cd /home/dwalleck/cafe-demo/.opencafe/logs/github/
 ↪prod.config/2017-04-19_11_26_38.698599
(cafe-demo) dwalleck@minerva:~/cafe-demo/.opencafe/logs/github/prod.config/2017-04-19_
 ↪11_26_38.698599$ ls -la
total 36
```

```
drwxrwxrwx 0 dwalleck dwalleck   512 Apr 19 11:26 .
drwxrwxrwx 0 dwalleck dwalleck   512 Apr 19 11:26 ..
-rw-rw-rw- 1 dwalleck dwalleck 15613 Apr 19 11:26 cafe.master.log
-rw-rw-rw- 1 dwalleck dwalleck 15353 Apr 19 11:26 opencafe_demo.github.test_issues_
↪api.BasicGitHubTest.log
```

Two log files were generated by this test run. The second log file is named by the full package name of the test class that was run. If there had been multiple test classes loaded for execution, there would be one file per class run. The benefit of this is to be able to jump directly to the log file that you are interested in inspecting. The contents of the logs contain the HTTP requests made during test execution, but they also contain headers to mark what point the in the lifecycle of the test is being executed:

```
2017-04-19 11:26:38,838: INFO: root: ␣
↪==========================================================
2017-04-19 11:26:38,840: INFO: root: Fixture.....: opencafe_demo.github.test_issues_
↪api.BasicGitHubTest
2017-04-19 11:26:38,840: INFO: root: Created At...: 2017-04-19 11:26:38.838285
2017-04-19 11:26:38,840: INFO: root: ␣
↪==========================================================
2017-04-19 11:26:38,842: INFO: root: ␣
↪========66==================================================
2017-04-19 11:26:38,842: INFO: root: Test Case....: test_get_issue_response_code_is_
↪200
2017-04-19 11:26:38,843: INFO: root: Created At...: 2017-04-19 11:26:38.838285
2017-04-19 11:26:38,843: INFO: root: No Test description.
2017-04-19 11:26:38,843: INFO: root: ␣
↪==========================================================
```

The other file, `cafe.master.log` is a summation of the other log files in the order the tests were executed. This allows the user to consume the logs however they find easiest.

3.2 OpenCAFE Documentation

3.2.1 cafe

cafe.common package

cafe.common.reporting package

cafe.common.reporting.base_report module

class cafe.common.reporting.base_report.**BaseReport**
Bases: object

generate_report(*result_parser*, *all_results=None*, *path=None*)

cafe.common.reporting.cclogging module

cafe.common.reporting.cclogging.**getLogger**(*log_name=None*, *log_level=None*)
Convenience function to create a logger and set it's log level at the same time. Log level defaults to logging.DEBUG Note: If the root log is accesed via this method in VERBOSE mode, the root log will be initialized and returned, if it hasn't been initialized already.

cafe.common.reporting.cclogging.**get_object_namespace**(*obj*)
Attempts to return a dotted string name representation of the general form 'package.module.class.obj' for an object that has an __mro__ attribute

Designed to let you to name loggers inside objects in such a way that the engine logger organizes them as child loggers to the modules they originate from.

So that logging doesn't cause exceptions, if the namespace cannot be extracted from the object's mro attribute, the actual name returned is set to a probably-unique string, the id() of the object passed, and is then further improved by a series of functions until one of them fails. The value of the last successful name-setting method is returned.

cafe.common.reporting.cclogging.**init_root_log_handler**(*override_handler=None*)
Setup root log handler if the root logger doesn't already have one

cafe.common.reporting.cclogging.**log_info_block**(*log*, *info*, *separator=None*, *heading=None*, *log_level=20*, *one_line=False*)
Expects info to be a list of tuples or an OrderedDict Logs info in blocks surrounded by a separator: ========== A heading will print here, with another separator below it. ========== Items are logged in order.....: Info And are separated from their info.....: Info By at least three dots.....: Info If no second value is given in the tuple, a single line is logged Lower lines will still line up correctly.....: Info The longest line dictates the dot length for all lines...: Like this ========== if one_line is true, info block will be logged as a single line, formatted using newlines. Otherwise, each line of the info block will be logged as seperate log lines (with seperate timestamps, etc.)

cafe.common.reporting.cclogging.**logsafe_str**(*data*)

cafe.common.reporting.cclogging.**parse_class_namespace_string**(*class_string*)
Parses the dotted namespace out of an object's __mro__. Returns a string

cafe.common.reporting.cclogging.**setup_new_cchandler**(*log_file_name*, *log_dir=None*, *encoding=None*, *msg_format=None*)
Creates a log handler named <log_file_name> configured to save the log in <log_dir> or <os environment variable 'CAFE_TEST_LOG_PATH'>, in that order or precedent. File handler defaults: 'a+', encoding=encoding or "UTF-8", delay=True

cafe.common.reporting.json_report module

class cafe.common.reporting.json_report.**JSONReport**
Bases: *cafe.common.reporting.base_report.BaseReport*

generate_report(*result_parser*, *all_results=None*, *path=None*)
Generates a JSON report in the specified directory.

cafe.common.reporting.metrics module

class cafe.common.reporting.metrics.**CSVWriter**(*headers*, *file_name*, *log_dir='.'*, *start_clean=False*)
Bases: object

writerow(*row_list*)

class cafe.common.reporting.metrics.**PBStatisticsLog**(*file_name=None*, *log_dir='.'*, *start_clean=False*)

Bases: *cafe.common.reporting.metrics.CSVWriter*

extends the csv writer

report(*test_result=<cafe.common.reporting.metrics.TestRunMetrics object>*)

class cafe.common.reporting.metrics.**TestResultTypes**

Bases: object

Types dictating an individual Test Case result @cvar PASSED: Test has passed @type PASSED: C{str} @cvar FAILED: Test has failed @type FAILED: C{str} @cvar SKIPPED: Test was skipped @type SKIPPED: C{str} @cvar TIMEDOUT: Test exceeded pre-defined execution time limit @type TIMEDOUT: C{str} @cvar ERRORED: Test has errored @type ERRORED: C{str} @note: This is essentially an Enumerated Type

ERRORED = 'ERRORED'

FAILED = 'Failed'

PASSED = 'Passed'

SKIPPED = 'Skipped'

TIMEDOUT = 'Timedout'

UNKNOWN = 'UNKNOWN'

class cafe.common.reporting.metrics.**TestRunMetrics**

Bases: object

Contains test timing and results metrics for a test.

class cafe.common.reporting.metrics.**TestTimer**

Bases: object

@summary: Generic Timer used to track any time span @ivar start_time: Timestamp from the start of the timer @type start_time: C{datetime} @ivar stop_time: Timestamp of the end of the timer @type stop_time: C{datetime}

get_elapsed_time()

@summary: Convenience method for total elapsed time @rtype: C{datetime} @return: Elapsed time for this timer. C{None} if timer has not started

start()

@summary: Starts this timer @return: None @rtype: None

stop()

@summary: Stops this timer @return: None @rtype: None

cafe.common.reporting.reporter module

class cafe.common.reporting.reporter.**Reporter**(*result_parser*, *all_results*)

generate_report(*result_type*, *path=None*)

Creates a report object based on what type is given and generates the report in the specified directory.

cafe.common.reporting.xml_report module

class cafe.common.reporting.xml_report.**XMLReport**
Bases: *cafe.common.reporting.base_report.BaseReport*

generate_report(*result_parser*, *all_results=None*, *path=None*)
Generates an XML report in the specified directory.

cafe.common.unicode

cafe.common.unicode.**PLANE_NAMES = <class 'cafe.common.unicode.PLANE_NAMES'>**
Namespace that defines all standard Unicode Plane names

A list-like object (UnicodeRangeList) made up of UnicodeRange objects. It covers the same total range as UNICODE_BLOCKS, but is instead organized by plane names instead of block names, which results in fewer but larger ranges.

cafe.common.unicode.**BLOCK_NAMES = <class 'cafe.common.unicode.BLOCK_NAMES'>**
Namespace that defines all standard Unicode Block names

A list-like object (UnicodeRangeList) made up of UnicodeRange objects. Each UnicodeRange object in the list corresponds to a named Unicode Block, and contains the start and end integer for that Block.

cafe.common.unicode.**UNICODE_BLOCKS**(*cafe.common.unicode.UnicodeRangeList*)
list-like object that iterates through named ranges of unicode codepoints Instantiated at runtime (when imported) near the bottom of this file

cafe.common.unicode.**UNICODE_PLANES**(*cafe.common.unicode.UnicodeRangeList*)
list-like object that iterates through ranges of ranges of unicode codepoints Instantiated at runtime (when imported) near the bottom of this file

See also:

http://en.wikipedia.org/wiki/Unicode#Architecture_and_terminology

Usage Examples:

```
# Print all the characters in the "Thai" unicode block
for c in UNICODE_BLOCKS.get_range(BLOCK_NAMES.thai).encoded_codepoints():
    print c

# Iterate through all the integer codepoints in the "Thai" unicode block
for i in UNICODE_BLOCKS.get_range(BLOCK_NAMES.thai).codepoints():
    do_something(i)

# Get a list of the names of all the characters in the "Thai" unicode block
    [n for n in UNICODE_BLOCKS.get_range(
        BLOCK_NAMES.thai).codepoint_names()]
```

cafe.common.unicode.**UNICODE_ENDING_CODEPOINT = 1114109**
Integer denoting the last unicode codepoint

cafe.common.unicode.**UNICODE_STARTING_CODEPOINT = 0**
Integer denoting the first unicode codepoint

class cafe.common.unicode.**UnicodeRange**(*start*, *end*, *name*)
Bases: object

Iterable representation of a range of unicode codepoints. This can represent a standard Unicode Block, a standard Unicode Plane, or even a custom range.

A UnicodeRange object contains a start, end, and name attribute which normally corresponds to the start and end integer for a range of Unicode codepoints.

Each UnicodeRange object includes generators for performing common functions on the codepoints in that integer range.

codepoint_names()
Generator that yields the name of each codepoint in range as a string.

If a name cannot be found, the codepoint's integer value is returned in hexidecimal format as a string.

Return type generator, returns strings

codepoints()
Generator that yields each codepoint in range as an integer.

Return type generator, returns ints

encoded_codepoints(*encoding='utf-8'*)
Generator that yields each codepoint name in range, encoded.

Parameters **encoding** (*string*) – the encoding to use on the string

Return type generator, returns unicode strings

class cafe.common.unicode.**UnicodeRangeList**
Bases: list

A list-like for containing collections of UnicodeRange objects.

Allows iteration through all codepoins in collected ranged, even if the ranges are disjointed. Useful for for creating custom ranges for specialized testing.

codepoint_names()
Generator that yields the name of each codepoint in range as a string.

If a name cannot be found, the codepoint's integer value is returned in hexidecimal format as a string.

Return type generator, returns strings

codepoints()
Generator that yields each codepoint in all ranges as an integer.

Return type generator, returns ints

encoded_codepoints(*encoding='utf-8'*)
Generator that yields each codepoint name in range, encoded.

Parameters **encoding** (*string*) – the encoding to use on the string

Return type generator, returns unicode strings

get_range(*range_name*)
Get a range of unicode codepoints by block name.

Returns a single *UnicodeRange* object representing the codepoints in the unicode block range named by range_name, if such a range exists in the instance of *UnicodeRangeList* that *get_range* is being called from.

Parameters **range_name** (*string*) – name of the requested unicode block range.

Return type *UnicodeRange* class instance, or None

get_range_list(*range_name_list*)
Get a list of ranges of unicode codepoints by block names.

Returns a single *UnicodeRangeList* object representing the codepoints in the unicode block ranges named by range_name_list, if such ranges exists in the instance of *UnicodeRangeList* that *get_range_list* is being called from.

Parameters **range_name_list** (*list of strings*) – name(s) of requested unicode block ranges.

Return type *UnicodeRangeList* class instance, or None

cafe.common.unicode.**codepoint_name**(*codepoint_integer*)

Expects a Unicode codepoint as an integer.

Returns the unicode name of codepoint_integer if valid unicode codepoint, None otherwise

If a name cannot be found, the codepoint's integer value is returned in hexidecimal format as a string.

cafe.common.unicode.**codepoint_parent_block**(*codepoint_integer*)

Expects a Unicode codepoint as an integer.

Return a UnicodeRange object representing the unicode block that codepoint_integer belongs to.

cafe.common.unicode.**codepoint_parent_plane**(*codepoint_integer*)

Expects a Unicode codepoint as an integer.

Return a UnicodeRangeList of UnicodeRange objects representing the unicode plane that codepoint_integer belongs to.

cafe.configurator package

cafe.configurator.cli module

class cafe.configurator.cli.**ConfiguratorCLI**

Bases: object

CLI for future engine management and configuration options.

classmethod run()

class cafe.configurator.cli.**EngineActions**

Bases: object

class Init(*args*)

Bases: object

class InitInstall(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Bases: argparse.Action

class cafe.configurator.cli.**PluginActions**

Bases: object

class InstallPlugin(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Bases: argparse.Action

class ListPlugins(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Bases: argparse.Action

cafe.configurator.cli.**entry_point**()

cafe.configurator.managers module

class cafe.configurator.managers.**EngineConfigManager**
Bases: object

ENGINE_CONFIG_PATH = '/home/docs/checkouts/readthedocs.org/user_builds/opencafe/envs/s

classmethod build_engine_config()

classmethod generate_default_engine_config()

classmethod install_optional_configs(*source_directory*, *print_progress=True*)

static read_config_file(*path*)

static rename_section(*config_parser_object*, *current_section_name*, *new_section_name*)

static rename_section_option(*config_parser_object*, *section_name*, *current_option_name*, *new_option_name*)

classmethod update_engine_config()
Applies to an existing engine.config file all modifications made to the default engine.config file since opencafe's release in the order those modification where added.

wrapper = <textwrap.TextWrapper instance>

static write_and_chown_config(*config_parser_object*, *path*)

classmethod write_config_backup(*config*)

class cafe.configurator.managers.**EngineDirectoryManager**
Bases: object

OPENCAFE_ROOT_DIR = '/home/docs/checkouts/readthedocs.org/user_builds/opencafe/envs/st

OPENCAFE_SUB_DIRS = {'DATA_DIR': '/home/docs/checkouts/readthedocs.org/user_builds/ope

classmethod build_engine_directories()
Updates, creates, and owns (as needed) all default directories

classmethod create_engine_directories()

classmethod set_engine_directory_permissions()
Recursively changes permissions default engine directory so that everything is user-owned

wrapper = <textwrap.TextWrapper instance>

class cafe.configurator.managers.**EnginePluginManager**
Bases: object

classmethod install_plugin(*plugin_name*)
Install a single plugin by name into the current environment

classmethod install_plugins(*plugin_names*)
Installs a list of plugins into the current environment

classmethod list_plugins()
Lists all plugins currently available in user's .opencafe cache

exception cafe.configurator.managers.**PackageNotFoundError**
Bases: exceptions.Exception

class cafe.configurator.managers.**PlatformManager**
Bases: object

Methods for dealing with the OS cafe is running on

USING_VIRTUALENV = True

USING_WINDOWS = False

classmethod get_current_user()
Returns the name of the current user. For Linux, always tries to return a user other than 'root' if it can.

classmethod get_user_gid()

classmethod get_user_home_path()

classmethod get_user_uid()

classmethod safe_chown(*path*)

classmethod safe_create_dir(*directory_path*)

class cafe.configurator.managers.**TestEnvManager**(*product_name*, *test_config_file_name*, *engine_config_path=None*, *test_repo_package_name=None*)

Bases: object

Sets all environment variables used by cafe and its implementations.

Wraps all internally-set and config-controlled environment variables in read-only properties for easy access. Useful for writing bootstrappers for runners and scripts.

Set the environment variable "CAFE_ALLOW_MANAGED_ENV_VAR_OVERRIDES" to any value to enable overrides for derived environment variables. (The full list of these is available in the attribute MANAGED_VARS)

NOTE: The TestEnvManager is only responsible for setting these vars, it has no control over how they are used by the engine or its implementations, so override them at your own risk!

USAGE HINTS: If you set CAFE_TEST_REPO_PATH, you should also set the CAFE_TEST_REPO_PACKAGE accordingly, as having them point to different things could cause undefined behavior. (The path is normally derived from the package).

MANAGED_VARS = {'test_data_directory': 'CAFE_DATA_DIR_PATH', 'test_config_file_path':

finalize(*create_log_dirs=True*)
Calls all lazy_properties in the TestEnvManager.

Sets all lazy_properties to their configured or derived values. To override this behavior, simply don't call finalize(): note that unless you manually set the os environment variables yourself this will result in undefined behavior. Creates all log directories, overridden by making create_log_dirs=False. Checks that all set paths exists, raises exception if they don't.

test_config_file_path = None

test_data_directory = None

test_log_dir = None

test_logging_verbosity = None

test_master_log_file_name = None

test_repo_package = None

test_repo_path = None

test_root_log_dir = None

cafe.drivers package

Subpackages

cafe.drivers.behave package

cafe.drivers.behave.runner module

cafe.drivers.behave.runner.**entry_point**()

cafe.drivers.behave.runner.**print_mug**(*base_dir*)

cafe.drivers.lettuce package

Module contents

cafe.drivers.pyvows package

cafe.drivers.pyvows.runner module

cafe.drivers.pyvows.runner.**entry_point**()

cafe.drivers.pyvows.runner.**print_mug**(*base_dir*)

cafe.drivers.specter package

cafe.drivers.specter.runner module

cafe.drivers.specter.runner.**entry_point**()

cafe.drivers.unittest package

cafe.drivers.unittest.datasets module

class cafe.drivers.unittest.datasets.**DatasetFileLoader**(*file_object*)

Bases: *cafe.drivers.unittest.datasets.DatasetList*

Reads a file object's contents in as json and converts them to lists of Dataset objects. Files should be opened in 'rb' (read binady) mode. File should be a list of dictionaries following this format: [{'name':"dataset_name", 'data':{key:value, key:value, ...}},] if name is ommited, it is replaced with that dataset's location in the load order, so that not all datasets need to be named.

class cafe.drivers.unittest.datasets.**DatasetGenerator**(*list_of_dicts*, *base_dataset_name=None*)

Bases: *cafe.drivers.unittest.datasets.DatasetList*

Generates Datasets from a list of dictionaries, which are named numericaly according to the source dictionary's order in the source list. If a base_dataset_name is provided, that is used as the base name postfix for all tests before they are numbered.

class cafe.drivers.unittest.datasets.**DatasetList**
Bases: list

Specialized list-like object that holds Dataset objects

append(*dataset*)

append_new_dataset(*name*, *data_dict*, *tags=None*, *decorators=None*)
Creates and appends a new Dataset

When including a *decorators* value (a list), make sure that the functions provided in the list are provided in the order in which they should be executed. When comparing them to typical stacked decorators, order them from bottom to top.

apply_test_decorators(**decorators*)
Applies decorators to all tests in dataset list

apply_test_tags(**tags*)
Applies tags to all tests in dataset list

dataset_name_map()
Creates a dictionary with key=count and value=dataset name

dataset_names()
Gets a list of dataset names from dataset list

extend(*dataset_list*)

extend_new_datasets(*dataset_list*)
Creates and extends a new DatasetList

merge_dataset_tags(**dataset_lists*)

static replace_invalid_characters(*string*, *new_char='_'*)
This functions corrects string so the following is true Identifiers (also referred to as names) are described by the following lexical definitions: identifier ::= (letter|"_") (letter | digit | "_")* letter ::= lowercase | uppercase lowercase ::= "a"…"z" uppercase ::= "A"…"Z" digit ::= "0"…"9"

class cafe.drivers.unittest.datasets.**DatasetListCombiner**(**datasets*)
Bases: *cafe.drivers.unittest.datasets.DatasetList*

Class that can be used to combine multiple DatasetList objects together. Produces the product of combining every dataset from each list together with the names merged together. The data is overridden in a cascading fashion, similar to CSS, where the last dataset takes priority.

class cafe.drivers.unittest.datasets.**TestMultiplier**(*num_range*)
Bases: *cafe.drivers.unittest.datasets.DatasetList*

Creates num_range number of copies of the source test, and names the new tests numerically. Does not generate Datasets.

cafe.drivers.unittest.decorators module

cafe.drivers.unittest.decorators.**DataDrivenClass**(**dataset_lists*)
Use data driven class decorator. designed to be used on a fixture.

cafe.drivers.unittest.decorators.**DataDrivenFixture**(*cls*)
Generates new unittest test methods from methods defined in the decorated class

exception cafe.drivers.unittest.decorators.**DataDrivenFixtureError**
Bases: exceptions.Exception

Error if you apply DataDrivenClass to class that isn't a TestCase

exception cafe.drivers.unittest.decorators.**EmptyDSLError**(*dsl_namespace*, *original_test_list*)
Bases: exceptions.Exception

Custom exception to allow errors in Datadriven classes with no data.

cafe.drivers.unittest.decorators.**data_driven_test**(**dataset_sources*, ***kwargs*)
Used to define the data source for a data driven test in a DataDrivenFixture decorated Unittest TestCase class

class cafe.drivers.unittest.decorators.**memoized**(*func*)
Bases: object

Decorator. @see: https://wiki.python.org/moin/PythonDecoratorLibrary#Memoize Caches a function's return value each time it is called. If called later with the same arguments, the cached value is returned (not reevaluated).

Adds and removes handlers to root log for the duration of the function call, or logs return of cached result.

cafe.drivers.unittest.decorators.**skip_open_issue**(*type*, *bug_id*)

cafe.drivers.unittest.decorators.**tags**(**tags*, ***attrs*)
Adds tags and attributes to tests, which are interpreted by the cafe-runner at run time

cafe.drivers.unittest.fixtures module

@summary: Base Classes for Test Fixtures @note: Corresponds DIRECTLY TO A unittest.TestCase @see: http://docs.python.org/library/unittest.html#unittest.TestCase

class cafe.drivers.unittest.fixtures.**BaseBurnInTestFixture**(*methodName='runTest'*)
Bases: *cafe.drivers.unittest.fixtures.BaseTestFixture*

@summary: Base test fixture that allows for Burn-In tests

classmethod addTest(*test_case*)
@summary: Adds a test case

classmethod setUpClass()
@summary: inits burning testing variables

class cafe.drivers.unittest.fixtures.**BaseTestFixture**(*methodName='runTest'*)
Bases: unittest.case.TestCase

@summary: This should be used as the base class for any unittest tests, meant to be used instead of unittest.TestCase.

@see: http://docs.python.org/library/unittest.html#unittest.TestCase

classmethod addClassCleanup(*function*, **args*, ***kwargs*)
@summary: Named to match unittest's addCleanup. ClassCleanup tasks run if setUpClass fails, or after tearDownClass. (They don't depend on tearDownClass running)

classmethod assertClassSetupFailure(*message*)

@summary: Use this if you need to fail from a Test Fixture's setUpClass() method

classmethod assertClassTeardownFailure(*message*)

@summary: Use this if you need to fail from a Test Fixture's tearUpClass() method

logDescription()
@summary: Returns a formatted description from the _testMethodDoc

setUp()
@summary: Logs test metrics

classmethod setUpClass()
@summary: Adds logging/reporting to Unittest setUpClass

shortDescription()
@summary: Returns a formatted description of the test

tearDown()

@todo: This MUST be upgraded this from resultForDoCleanups into a better pattern or working with the result object directly. This is related to the todo in L{TestRunMetrics}

classmethod tearDownClass()
@summary: Adds stop reporting to Unittest setUpClass

cafe.drivers.unittest.parsers module

class cafe.drivers.unittest.parsers.**Result**(*test_class_name*, *test_method_name*, *failure_trace=None*, *skipped_msg=None*, *error_trace=None*, *test_time=0*)
Bases: object

Result object used to create the json and xml results

class cafe.drivers.unittest.parsers.**SummarizeResults**(*result_dict*, *tests*, *execution_time*, *datagen_time=None*)
Bases: object

Reads in vars dict from suite and builds a Summarized results obj

gather_results()
Gets a result obj for all tests ran and failed setup classes

get_passed_tests()
Gets a list of results objects for passed tests

summary_result()
Returns a dictionary containing counts of tests and statuses

cafe.drivers.unittest.runner module

class cafe.drivers.unittest.runner.**OpenCafeParallelTextTestRunner**(*stream=<open file '<stderr>', mode 'w'>*, *descriptions=1*, *verbosity=1*)
Bases: unittest.runner.TextTestRunner

run(*test*)
Run the given test case or test suite.

class cafe.drivers.unittest.runner.**SuiteBuilder**(*cl_args*, *test_repo_name*)
Bases: object

build_suite(*module_path*)
loads the found tests and builds the test suite

generate_suite()
creates a single unittest test suite

generate_suite_list()
creates a list containing unittest test suites

get_classes(*loaded_module*)
finds all the classes in a loaded module calculates full path to class

get_modules()
walks all modules in the test repo, filters by product and module filter. Filter passed in with -m returns a list of module dotpath strings

class cafe.drivers.unittest.runner.**UnittestRunner**
Bases: object

static dump_results(*start*, *finish*, *results*)

static execute_test(*runner*, *test_id*, *test*, *results*)

static get_runner(*cl_args*)

log_errors(*label*, *result*, *errors*)

static print_mug_and_paths(*test_env*)

run()
loops through all the packages, modules, and methods sent in from the command line and runs them

run_parallel(*test_suites*, *test_runner*, *result_type=None*, *results_path=None*)

run_serialized(*master_suite*, *test_runner*, *result_type=None*, *results_path=None*)

cafe.drivers.unittest.runner.**entry_point**()

cafe.drivers.unittest.runner.**tree**(*directory*, *padding*, *print_files=False*)
creates an ascii tree for listing files or configs

cafe.drivers.unittest.suite module

Contains a monkeypatched version of unittest's TestSuite class that supports a version of addCleanup that can be used in classmethods. This allows a more granular approach to teardown to be used in setUpClass and classmethod helper methods

class cafe.drivers.unittest.suite.**OpenCafeUnittestTestSuite**(*tests=()*)
Bases: unittest.suite.TestSuite

cafe.drivers.base module

class cafe.drivers.base.**FixtureReporter**(*parent_object*)
Bases: object

Provides logging and metrics reporting for any test fixture

start()
Starts logging and metrics reporting for the fixture

start_test_metrics(*class_name*, *test_name*, *test_description=None*)
Creates a new Metrics object and starts reporting to it. Useful for creating metrics for individual tests.

stop()
Logs all collected metrics and stats, then stops logging and metrics reporting for the fixture.

stop_test_metrics(*test_name*, *test_result*)
Stops reporting on the Metrics object created in start_test_metrics. Logs all collected metrics. Useful for logging metrics for individual test at the test's conclusion

cafe.drivers.base.**get_error**(*exception=None*)
Gets errno from exception or returns one

cafe.drivers.base.**parse_runner_args**(*arg_parser*)
Generic CAFE args for external runners

cafe.drivers.base.**print_exception**(*file_=None*, *method=None*, *value=None*, *exception=None*)
Prints exceptions in a standard format to stderr.

cafe.drivers.base.**print_mug**(*name*, *brewing_from*)
Generic CAFE mug

cafe.engine package

Subpackages

cafe.engine.clients package

cafe.engine.clients.base module

class cafe.engine.clients.base.**BaseClient**
Bases: object

cafe.engine.clients.commandline module

class cafe.engine.clients.commandline.**BaseCommandLineClient**(*base_command=None*, *env_var_dict=None*)
Bases: *cafe.engine.clients.base.BaseClient*

Provides low level connectivity to the commandline via popen()

Primarily intended to serve as base classes for a specific command line client Class. This class is dependent on a local installation of the wrapped client process. The thing you run has to be there!

run_command(*cmd*, **args*)
Sends a command directly to this instance's command line @param cmd: Command to sent to command line @type cmd: C{str} @param args: Optional list of args to be passed with the command @type args: C{list} @raise exception: If unable to close process after running the command @return: The full response details from the command line @rtype: L{CommandLineResponse} @note: PRIVATE. Can be overridden in a child class

run_command_async(*cmd*, **args*)
Running a command asynchronously returns a CommandLineResponse objecct with a running subprocess.Process object in it. This process needs to be closed or killed manually after execution.

set_environment_variables(*env_var_dict=None*)
Sets all os environment variables provided in env_var_dict

unset_environment_variables(*env_var_list=None*)
Unsets all os environment variables provided in env_var_dict by default. If env_var_list is passed, attempts to unset all environment vars in list

update_environment_variables(*env_var_dict=None*)
Sets all os environment variables provided in env_var_dict

cafe.engine.clients.ping module

class cafe.engine.clients.ping.**PingClient**
Bases: object

@summary: Client to ping windows or linux servers

DEFAULT_NUM_PINGS = 3

PING_IPV4_COMMAND_LINUX = 'ping -c {num_pings}'

PING_IPV4_COMMAND_WINDOWS = 'ping -c {num_pings}'

PING_IPV6_COMMAND_LINUX = 'ping6 -c {num_pings}'

PING_IPV6_COMMAND_WINDOWS = 'ping -6 -c {num_pings}'

PING_PACKET_LOSS_REGEX = '(\\d{1,3})\\.?\\d*\\%.*loss'

classmethod ping(*ip*, *ip_address_version*, *num_pings=3*)
@summary: Ping an IP address, return if replies were received or not. @param ip: IP address to ping @type ip: string @param ip_address_version: IP Address version (4 or 6) @type ip_address_version: int @param num_pings: Number of pings to attempt @type num_pings: int @return: True if the server was reachable, False otherwise @rtype: bool

classmethod ping_percent_loss(*ip*, *ip_address_version*, *num_pings=3*)

@summary: Ping an IP address, return the percent of replies not returned

@param ip: IP address to ping @type ip: string @param ip_address_version: IP Address version (4 or 6) @type ip_address_version: int @param num_pings: Number of pings to attempt @type num_pings: int @return: Percent of responses not received, based on number of requests @rtype: int

classmethod ping_percent_success(*ip*, *ip_address_version*, *num_pings=3*)
@summary: Ping an IP address, return the percent of replies received @param ip: IP address to ping @type ip: string @param ip_address_version: IP Address version (4 or 6) @type ip_address_version: int @param num_pings: Number of pings to attempt @type num_pings: int @return: Percent of responses received, based on number of requests @rtype: int

cafe.engine.clients.sql module

class cafe.engine.clients.sql.**BaseSQLClient**
Bases: *cafe.engine.clients.base.BaseClient*

Base support client for DBAPI 2.0 clients.

This client is not meant to be used directly. New clients will extend this client and live inside of the individual CAFE.

For more information on the DBAPI 2.0 standard please visit: .. seealso:: http://www.python.org/dev/peps/pep-0249

close()
Closes the connection

connect(*data_source_name=None*, *user=None*, *password=None*, *host=None*, *database=None*)
Connects to self._driver with passed parameters

Parameters

- **data_source_name** (*string*) – The data source name
- **user** (*string*) – Username
- **password** (*string*) – Password
- **host** (*string*) – Hostname
- **database** (*string*) – Database Name

execute(*operation*, *parameters=None*, *cursor=None*)
Calls execute with operation & parameters sent in on either the passed cursor or a new cursor

For more information on the execute command see: http://www.python.org/dev/peps/pep-0249/#id15

Parameters

- **operation** (*string*) – The operation being executed
- **parameters** (*string or dictionary*) – Sequence or map that wil be bound to variables in the operation
- **cursor** (*cursor object*) – A pre-existing cursor

execute_many(*operation*, *seq_of_parameters=None*, *cursor=None*)
Calls executemany with operation & parameters sent in on either the passed cursor or a new cursor

For more information on the execute command see: http://www.python.org/dev/peps/pep-0249/#executemany

Parameters

- **operation** (*string*) – The operation being executed
- **seq_of_parameters** (*string or object*) – The sequence or mappings that will be run against the operation
- **cursor** (*cursor object*) – A pre-existing cursor

exception cafe.engine.clients.sql.**SQLClientException**
Bases: exceptions.Exception

cafe.engine.models package

cafe.engine.models.base module

class cafe.engine.models.base.**AutoMarshallingDictModel**
Bases: dict, *cafe.engine.models.base.AutoMarshallingModel*

Dict-like AutoMarshallingModel used for some special cases

class cafe.engine.models.base.**AutoMarshallingListModel**
Bases: list, *cafe.engine.models.base.AutoMarshallingModel*

List-like AutoMarshallingModel used for some special cases

class cafe.engine.models.base.**AutoMarshallingModel**
Bases: *cafe.engine.models.base.BaseModel*, *cafe.engine.models.base.CommonToolsMixin*, *cafe.engine.models.base.JSON_ToolsMixin*, *cafe.engine.models.base.XML_ToolsMixin*

@summary: A class used as a base to build and contain the logic necessary to automatically create serialized requests and automatically deserialize responses in a format-agnostic way.

classmethod deserialize(*serialized_str*, *format_type*)

serialize(*format_type*)

class cafe.engine.models.base.**BaseModel**
Bases: object

class cafe.engine.models.base.**CommonToolsMixin**
Bases: object

Methods used to make building data models easier, common to all types

class cafe.engine.models.base.**JSON_ToolsMixin**
Bases: object

Methods used to make building json data models easier

class cafe.engine.models.base.**XML_ToolsMixin**
Bases: object

Methods used to make building xml data models easier

xml_header

cafe.engine.models.behavior_response module

class cafe.engine.models.behavior_response.**BehaviorResponse**
Bases: object

An object to represent the result of behavior. @ivar response: Last response returned from last client call @ivar ok: Represents the success state of the behavior call @type ok:C{bool} @ivar entity: Data model created via behavior calls, if applicable

cafe.engine.models.commandline_response module

@summary: Responses directly from the command line

class cafe.engine.models.commandline_response.**CommandLineResponse**
Bases: *cafe.engine.models.base.BaseModel*

Bare bones object for any Command Line Connector response @ivar Command: The full original command string for this response @type Command: C{str} @ivar StandardOut: The Standard Out generated by this command @type StandardOut: C{list} of C{str} @ivar StandardError: The Standard Error generated by this command @type StandardError: C{list} of C{str} @ivar ReturnCode: The command's return code @type ReturnCode: C{int}

cafe.engine.models.data_interfaces module

class cafe.engine.models.data_interfaces.**BaseConfigSectionInterface**(*config_file_path*, *section_name*)

Bases: object

Base class for building an interface for the data contained in a SafeConfigParser object, as loaded from the config file as defined by the engine's config file.

get(*item_name*, *default=None*)

get_boolean(*item_name*, *default=None*)

get_json(*item_name*, *default=None*)

get_raw(*item_name*, *default=None*)

exception cafe.engine.models.data_interfaces.**ConfigDataException**

Bases: exceptions.Exception

exception cafe.engine.models.data_interfaces.**ConfigEnvironmentVariableError**

Bases: exceptions.Exception

class cafe.engine.models.data_interfaces.**ConfigParserDataSource**(*config_file_path*, *section_name*)

Bases: *cafe.engine.models.data_interfaces.DataSource*

get(*item_name*, *default=None*)

get_boolean(*item_name*, *default=None*)

get_json(*item_name*, *default=None*)

get_raw(*item_name*, *default=None*)

class cafe.engine.models.data_interfaces.**ConfigSectionInterface**(*config_file_path=None*, *section_name=None*)

Bases: *cafe.engine.models.data_interfaces.BaseConfigSectionInterface*

class cafe.engine.models.data_interfaces.**DataSource**

Bases: object

get(*item_name*, *default=None*)

get_boolean(*item_name*, *default=None*)

get_json(*item_name*, *default=None*)

get_raw(*item_name*, *default=None*)

class cafe.engine.models.data_interfaces.**DictionaryDataSource**

Bases: *cafe.engine.models.data_interfaces.DataSource*

get(*item_name*, *default=None*)

get_boolean(*item_name*, *default=None*)

get_json(*item_name*, *default=None*)

get_raw(*item_name*, *default=None*)

class cafe.engine.models.data_interfaces.**EnvironmentVariableDataSource**(*section_name*)

Bases: *cafe.engine.models.data_interfaces.DataSource*

get(*item_name*, *default=None*)

get_boolean(*item_name*, *default=None*)

get_json(*item_name*, *default=None*)

get_raw(*item_name*, *default=None*)

class cafe.engine.models.data_interfaces.**JSONDataSource**(*config_file_path*, *section_name*)
Bases: *cafe.engine.models.data_interfaces.DictionaryDataSource*

class cafe.engine.models.data_interfaces.**MongoDataSource**(*hostname*, *db_name*, *username*, *password*, *config_name*, *section_name*)
Bases: *cafe.engine.models.data_interfaces.DictionaryDataSource*

exception cafe.engine.models.data_interfaces.**NonExistentConfigPathError**
Bases: exceptions.Exception

cafe.engine.models.data_interfaces.**expected_values**(**values*)

cafe.engine.behaviors module

class cafe.engine.behaviors.**BaseBehavior**
Bases: object

exception cafe.engine.behaviors.**RequiredClientNotDefinedError**
Bases: exceptions.Exception

Raised when a behavior method call can't find a required client

cafe.engine.behaviors.**behavior**(**required_clients*)
Decorator that tags method as a behavior, and optionally adds required client objects to an internal attribute. Causes calls to this method to throw RequiredClientNotDefinedError exception if the containing class does not have the proper client instances defined.

cafe.engine.config module

class cafe.engine.config.**EngineConfig**(*config_file_path=None*)
Bases: *cafe.engine.models.data_interfaces.ConfigSectionInterface*

SECTION_NAME = 'OPENCAFE_ENGINE'

config_directory
Provided as the default location for test config files.

data_directory
Provided as the default location for data required by tests.

default_test_repo
Provided as the default name of the python package containing tests to be run. This package must be in your python path under the name provided here.

log_directory
Provided as the default location for logs. It is recommended that the default log directory be used as a root directory for subdirectories of logs.

logging_verbosity
Used by the engine to determine which loggers to add handlers to by default

master_log_file_name
Used by the engine logger as the default name for the file written to by the handler on the root python log (since the root python logger doesn't have a name by default).

temp_directory
Provided as the default location for temp files and other temporary output generated by tests (not for logs).

cafe.engine.provider module

class cafe.engine.provider.**BaseProvider**
Bases: object

3.2.2 Index

- genindex

3.3 Getting Started

3.3.1 Setting up a Development Environment

OpenCAFE strongly recommends the use of Python virtual environments for development.

While not required, if you wish to manage your Python versions as well as virtual environments, we suggest the use of pyenv. Instructions on installing pyenv can be found here: *Installing pyenv*

Building your virtual environment

If you wish to run OpenCAFE in a virtual environment, then you will need to install virtualenv

For easier management of your virtual environments, it is recommended that you install virtualenvwrapper as well.

```
sudo pip install virtualenvwrapper
```

Creating virtualenv and installing OpenCAFE

```
# Requires virtualenv and virtualenvwrapper to be installed
mkvirtualenv OpenCAFE

# Clone OpenCAFE Repo
git clone git@github.com:openstack/opencafe.git

# Change directory into the newly cloned repository
cd opencafe

# Install OpenCAFE into your virtualenv
pip install . --upgrade
```

Information regarding the operation of your virtual environment can be found here: *Working with virtual environments*

3.3.2 Installing OpenCAFE

Currently OpenCAFE is not available on PyPI *(coming soon)*. As a result, you will need to install via pip through either cloning the repository or downloading a snapshot archive.

Install OpenCAFE

The preferred method of installing OpenCAFE is using pip.

```
# Clone Repository
git clone git@github.com:CafeHub/opencafe.git

# Change current directory
cd opencafe

# Install OpenCAFE
pip install . --upgrade
```

If you don't wish to use Git, then you can download and uncompress a snapshot archive in place of cloning the repository.

Installing Plugins

When you install OpenCAFE, you only install the base framework. All other functionality is packaged into plugins.

Install

To install a plugin, use the following command:

```
# Installing the mongo plugin
cafe-config plugins install mongo
```

List

Retrieves a list of all plugins available to install

```
cafe-config plugins list
```

3.3.3 Using OpenCAFE

3.3.4 Working with pyenv

Installing pyenv

The official installation guide is available here pyenv on github. The official guide includes instructions for Mac OS X as well.

Installing pyenv on Ubuntu:

```
sudo apt-get install git python-pip make build-essential libssl-dev zlib1g-dev libbz2-
↪dev libreadline-dev libsqlite3-dev
sudo pip install virtualenvwrapper

git clone https://github.com/yyuu/pyenv.git ~/.pyenv
git clone https://github.com/yyuu/pyenv-virtualenvwrapper.git ~/.pyenv/plugins/pyenv-
↪virtualenvwrapper

echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.bashrc
echo 'export PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.bashrc
echo 'eval "$(pyenv init -)"' >> ~/.bashrc
echo 'pyenv virtualenvwrapper' >> ~/.bashrc

exec $SHELL
```

Installing a fresh version of Python

Now that pyenv and the virtualenvwrapper plugin is installed. We need to download and install a version of python. In this case, we're going to install Python 2.7.6.

```
# Install
pyenv install 2.7.6

# Set 2.7.6 as your shell default
pyenv global 2.7.6
```

3.3.5 Working with virtual environments

For a complete guide on using virtualenv, please refer to the virtualenv and virtualenvwrapper documentation

Using virtualenvwrapper

There are four primary commands that are used to manage your virtual environments with virtualenvwrapper.

Create

When you create a virtualenv with virtualenvwrapper it creates a virtual environment within the $HOME/.virtualenvs folder.

```
# Allows for you to new create a virtual environment based
# on your current python version(s)
mkvirtualenv CloudCAFE
```

Activate Environment

```
# Activates a virtual environment within your current shell.
workon CloudCAFE
```

Deactivate Environment

```
# Deactivates a virtual environment within your current shell.
deactivate
```

Remove Environment

```
# Deletes a virtual environment from your system.
rmvirtualenv CloudCAFE
```

3.4 Contributing

3.4.1 Development Process

Starting a Feature

To give your development visibility, we strongly recommend creating a GitHub issue describing your change before making any non-trivial change. This will also give other contributors an early opportunity to provide feedback on your change, which allows for questions that may have come up during the review process be addressed earlier.

All development should occur in feature branches. The name of the feature branch should be a short, meaningful name helps a reviewer understand the purpose of the request. The scope of a feature branch should be relatively narrow and granular. It should either cover a small, standalone feature or one aspect of a larger feature. By keeping the scope of individual changes small, it encourages the size of pull requests to stay small as well. While there is no hard limit on the number of lines in a change, in general a review should not be larger than several hundred lines of code. If it grows larger than that, consider re-evaluating what the change is trying to accomplish to determine if it can be broken up into smaller chunks.

Maintaining a Feature Branch

During the lifetime of a branch, you will likely want to perform commits as your code progresses. However, when you submit your feature, your intent will be to submit the entirety of your work as one logical change. There are several strategies that can be used to handle this problem. The first is to commit to the feature branch as you normally would, and then squash the commits before submitting the branch for review. Another option is to make an initial commit to your branch and then amend all additional changes to that commit. We recommend the first approach as it allows you to have a history of your changes while working on the branch.

Another issue to consider while working in a feature branch is that other submissions may be merged before you submit your changes. These merged changes may modify some of the same code that you are also changing, leading to a conflict when your change is merged. To avoid this, you should be updating your master branch regularly to determine if changes have been made that will conflict with your feature branch. To sync your fork's master branch with OpenCafe's master, use the following steps:

```
0. git remote add upstream https://github.com/CafeHub/opencafe (this step
   only needs to be performed the first time)
1. git checkout master
2. git fetch upstream (you need to set the
3. git merge upstream/master
```

Once you have the upstream changes in your local repository, you can merge any changes back into your feature branch by rebasing. If any conflicts occurs during the rebase, you will need to resolve those issues before the rebase can finish the process. If your master branch is up to date, your feature branch can be updated using the following steps:

```
1. git checkout <your_branch>
2. git rebase -i master
3. Git should complain about conflicting changes to resolve
4. Resolve any merge conflicts
5. git rebase --continue
```

Committing Changes for Review

Once you have completed development of your feature, you should squash your commits to a single change to keep the commit history of the project clean. Your commit message should be informative and reference any GitHub issues that you have worked on in this branch. The following is one example:

```
fixes an issue with JSON serialization. Addresses issue #145.
```

3.4.2 Coding Standards

OpenCAFE standards are intended to allow flexability in solving coding issues, while maintaining uniformity and overall code quality.

Rules of Law

1. If a base class exists, use it. If the base class is missing features that you need, make improvements to the base class before implementing a new one.
2. Functions should only return one type. If a function can return a single item or a list of items, choose to return a list of items always, even if that means returning a list with a single item.
3. All code should be as explicit as possible. Favor readability/clarity over brevity.
4. Once you have submitted a branch for review, the only changes that should be made to that branch are changes requested by reviewers or functional issues. Any follow on work should be submitted in a new branch/pull request. **Failure to comply will result in the pull request being abandoned.**
5. If you want to change the rules of law, do lots of reviews, get added to core and make a pull request!

Development Standards

- It is **HIGHLY** encouraged that if you have not already read (and even if it's been a while since you have) the Python Enhancement Proposals (PEPs) PEP-8 and PEP 20 that you do so.
- Guidelines here are intended to help encourage code unity, they are not unbreakable rules, but should be adhered to failing a good reason not to. When in doubt, **ALL** code should conform either directly to or in the spirit of Python PEP 20, if you are still in doubt, go with Python PEP-8.
- If you really are still in doubt, see Guideline 2. Base Classes are your friend. Use them when they make sense.
- Always use **SPACES**. **NEVER TABS**. All block indention should be four (4) spaces.
- Avoid single letter variable names except in the case of iterators, in which case a descriptive variable name would still be preferable if possible.

- Do not leave trailing whitespace or whitespace in blank lines.
- Put two newlines between top-level code (funcs, classes, etc).
- Use only UNIX style newlines (“n”), not Windows style (“rn”).
- Follow the ordering/spacing guidelines described in PEP8 for imports.
- Put one newline between methods in classes and anywhere else.
- Avoid using line continuations unless absolutely necessary. Preferable alternatives are to wrap long lines in parenthesis, or line breaking on the open parenthesis of a function call.
- Long strings should be handled by wrapping the string in parenthesis and having quote delimited strings per line within.

Example::

long_string = (‘I cannot fit this whole phrase on one ‘ ‘line, but I can properly format this string ‘ ‘by using this type of structure.’)

- Do not write “except:”, use “except Exception:” at the very least
- Use try/except where logical. Avoid wrapping large blocks of code in in huge try/except blocks.
- Blocks of code should either be self documenting and/or well commented, especially in cases of non-standard code.
- Use Python list comprehensions when possible. They can make large blocks of code collapse to a single line.
- Use Enumerated Types where logical to pass around string constants or magic numbers between Functions, Methods, Classes and Packages. Python does not provide a default Enumerated Type data type, CloudCafe uses Class structs by naming convention in their place.

Example::

class ComputeServerStates(object): ACTIVE = “ACTIVE” BUILD = “BUILD” REBUILD = “REBUILD” ERROR = “ERROR” DELETING = “DELETING” DELETED = “DELETED” RESCUE = “RESCUE” PREP_RESCUE = “PREP_RESCUE” RESIZE = “RESIZE” VERIFY_RESIZE = “VERIFY_RESIZE”

3.4.3 Code Review Process

The goal of the code review process is to provide constructive feedback to contributors and to ensure that any changes follow the direction of the OpenCafe project. While a reviewer will look for any obvious logical flaws, the primary purpose of code reviews is **not** to verify that the submitted code functions correctly. We understand that the original design of OpenCafe did not lend itself well to unit testing, but we encourage that submissions include tests when possible.

Process Overview

Two steps will occur before a pull request is merged. First, our CI will use tox to run our style checks and all unit tests against the proposed branch. Once these checks pass, the pull request needs to reviewed and approved by at least one OpenCafe maintainer (the current list of maintainers can be viewed in the AUTHORS file of this project).

Review Etiquette

- Keep feedback constructive. Comment on the code and not the person. All comments should be civil.

- Review comments should be specific and descriptive. If alternative implementations need to be proposed, describe alternatives either as an inline snippet in the review comments or in a linked gist.
- All standards that contributors are held to should be documented. Reviewers should be able to point a contributor to a coding standard (either in PEP 8 or the OpenCafe development guidelines) that supports the concern. If you find that something is missing from the documented coding standards, open a pull request to our documentation with your clarifications.

Review Guidelines

Reviewers are encouraged to use their own judgement and express concerns or recommend alternatives as part of the review process. There is not a definitive checklist that reviewers use to evaluate a submission, but the following are some basic criteria that a reviewer would be likely to use:

- Does this submission follow standard Python and OpenCafe coding standards?
- Does the architecture of the solution make sense? Is there either a more simple or scalable solution? Does the solution add unnecessary complexity?
- Do the names of classes, functions, and variables impact the readability of the code?
- Do all classes and functions have docstrings?
- Are there sections of code whose purpose is unclear? Would additional comments or refactoring make it more clear?
- Were tests added for cases created by the code submission? (where applicable)

Merging

Once a pull request has passed our CI and code review, the reviewer will try to merge the pull request to the master branch. If conflicting changes have occurred during the time your pull request was open, a rebase may be required before the pull request can be merged successfully.

3.5 OpenCAFE Architecture

3.5.1 Models

One of the challenges in testing non-UI based applications is handling communication protocols between the test harness and the application under test. Abstracting this layer between the application and harness not only removes the concern of how communication occurs from the perspective of the test developer, but also makes it easier for the harness to adapt to changes in the structure of communication.

As part of the OpenCafe design strategy, we wanted to define a standard way of handling data serialization that was also generic enough to be used with any protocol. Doing so enabled us make other design decisions, such as making the serialization process transparent to the test developer (this is explained in detail in the clients section).

Design

Models in OpenCafe are very similar to data transfer objects (DTOs). The purpose of any methods defined by a model are in general limited to converting the model to and from another format. For example, a model that will be used in requests to a REST API would have methods to convert the object to JSON and XML, while an object that represents REST responses would contain methods to convert JSON or XML back to an object. By convention, these methods

are named _obj_to_<format> and _<format>_to_obj. This convention is used by other elements in the framework to determine at execution time which serialization format should be used.

For convenience, you may want to implement the __eq__ and __ne__ methods to allow standard comparison functions such as "in" and "not in" to be used in relation to the model. If you do this, make sure to implement both methods. Implementing __eq__ without implementing __ne__ will cause comparisons to not work as expected.

Example - Models for a REST API

In the example where the application under test is a REST API, the formats the application is likely to understand would be JSON and possibly XML. Our tests will need to be able to send and receive requests in both formats to be able to handle all possible scenarios.

For the purpose of this example, we'll focus on a basic authentication request. Per our specification, our system is expecting a request in one of the following formats:

```
JSON: { "auth": { "username" <user>, "api_key": <key>, "tenant_id": <tenant_id> }}
XML: <auth api_key="user" tenant_id="user" username="user" />
```

Based on the specification, the model should have three fields: username, api_key, and tenant_id. Since this is a request object, we will have to implement _obj_to_<format> methods for each possible request format, which in this case is JSON and XML. With those facts in mind, the following model could be derived:

```
import json
import xml.etree.ElementTree as ET

class AuthRequest(AutoMarshallingModel):

    def __init__(self, username, api_key, tenant_id):
        self.username = username
        self.api_key = api_key
        self.tenant_id = tenant_id

def _obj_to_json(self):
    body = {
        'username': self.username,
        'api_key': self.api_key,
        'tenant_id': tenant_id
    }
    return json.dumps({"auth": body})

def _obj_to_xml(self):
    element = ET.Element('auth')
    element.set('username', self.username)
    element.set('api_key', self.api_key)
    element.set('tenant', self.tenant_id)
    return ET.tostring(element)
```

Note that this model inherits from one of the OpenCafe base classes, AutoMarshallingModel. This class exposes the "serialize" and "deserialize" methods, which work in concert with the _obj_to_<format> methods to enable seamless data serialization.

3.5.2 Clients

OpenCAFE strives to provide a standard way of interacting with as many technologies as possible, in order to make functional testing of highly integrated systems easy to write, manage, and understand. Clients provide easy interaction

with myriad technologies via foreign function interfaces for RESTfull APIs, command line tools, databases and the like.

Design

- Clients should be simple and focused on providing native access to foreign functionality in a clean and easy to understand way.
- A client should not make assumptions about how it will be used, beyond those mandated by the foreign functionality.
- A client should be able to stand on it's own, without requiring any configuration or information beyond what is required for instantiation.

Examples

- The HTTP client itself doesn't require any information to instantiate, but an API client built using the HTTP client might require a URL and an auth token, since it's purpose is to interact solely with the API located at that URL.
- The commandline client offers logging, a uniform request/response model, and both synchronous and asynchronous requests on top of python's Popen method, but doesn't seek to expose functionality beyond running cli commands. The client deals with Popen and provides a simple way to get stdout, stderr, and stdin from a single command send to the local commandline. The client itself can be instantiated with a base command and used as an ad hoc interface for a specific commandline program, or left without a base command and used as an interface for the underlying shell.

3.5.3 Behaviors

3.6 Plugin Development

3.6.1 Writing your first plugin

3.6.2 Overview

3.7 FAQs

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

D

E

F

G

R

S

T

U

W

X