Mimic Documentation

Release 0.0.0

Individual Contributors

Apr 16, 2017

Contents

Mimic is an API-compatible mock service for Openstack Compute and Rackspace's implementation of Identity and Cloud Load Balancers. It is backed by in-memory data structure rather than a potentially expensive database.

Mimic helps with:

- fast set-up
- instant response
- cost efficient
- enables offline development
- enables ability to test unusual behaviors/errors of an api
- acts as a central repository for mocked responses from services

CHAPTER 1

Documentation

Development

mimic welcomes contributions!

File bugs and feature requests on our issue tracker on GitHub.

Getting started

Working on mimic requires the installation of a small number of development dependencies, which are listed in requirements/development.txt. They can be installed in a virtualenv using pip. This also installs mimic in editable mode.

For example:

```
$ # Create a virtualenv and activate it
$ pip install --requirement requirements/development.txt
```

You are now ready to run the tests and build the documentation.

Some of the tox jobs may require certain packages to be installed, so having homebrew installed would be useful if developing on Mac OS.

Running tests

mimic unit tests are found in the mimic/test/ directory. Then can be run via the built-in tox commands after setting up tox.

```
$ tox -e py27
```

You can also run the tests for other python interpreters. We use tox, which creates a virtualenv per tox job to run tests, linting, etc.:

```
$ tox
...
 py26: commands succeeded
 py27: commands succeeded
 pypy: commands succeeded
 docs: commands succeeded
 lint: commands succeeded
```

Building documentation

`mimic` documentation is stored in the `docs/` directory. It is written in reStructured Text and rendered using Sphinx.

To build the documentation, use `tox`:

```
$ tox -e docs
```

The HTML documentation index can now be found at `docs/_build/html/index.html`.

Alternately, you can use `sphinx` directly, if you would like to specify options:

```
$ sphinx-build -W -b html -d _tmp/doctrees docs docs/_build/html
```

Building a Mac application

The officially supported method of building of the application depends on the system python, and the pyobjc, and py2app libraries. Travis-CI is configured to build the mac application and run its tests.

To build the application and run its tests locally use the following commands.

```
$ cd /dir/where/mimic/lives/
$ ./build-app.sh
```

Once built, `mimic.app` can be found in the `./dist` directory. This application can be treated like any other mac application and moved into `~/Applications`. To start `mimic`, use the open command with the path to `mimic.app` , e.g. `open ./dist/mimic.app`.

When the application is running, the letter `M` will be visible in the menubar. To quit the application, simply click on the `M` and select `Quit`. You can view the application logs by opening `Applications/Utilities/Console.app`.

To run `mimic.app`'s tests use

```
$ /path/to/mimic.app/Contents/MacOS/run-tests
```

The application can also built as a standalone application that does not depend on the system python. This is *not* the officially supported method of building the application and is *not* tested by Travis-CI.

To build a standalone application, `py2app` requires the installation of a non-system framework python. In my experience, it is easiest to install a brewed 2.7 python. To install a brew python, you'll need to have homebrew installed.

The following commands will build the standalone application and run its tests.

```
$ brew install python
$ cd /dir/where/mimic/lives/

$ # build a virtualenv using the brewed python
$ virtualenv -p /usr/local/bin/python2.7 ./venv
$ source ./venv/bin/activate
$ # install mimic's dependencies including pyobjc and py2app
$ pip install -r requirements/production.txt
$ pip install -r requirements/mac-app.txt
$ python setup.py py2app
$ ./dist/mimic.app/Contents/MacOS/run-tests
```

Submitting patches

- If you have access to the mimic repository, always make a new branch for your work.
- If you don't have access to the mimic repository, working on branches in your fork is also nice because that will you can work on more than one PR at a time.
- Patches should be small to facilitate easier review.

Code

When in doubt, refer to **PEP 8** for Python code (with some exceptions). You can check if your code meets our automated requirements by running `flake8` against it. Even better would be to run the `tox` job:

```
$ tox -e lint
...
  lint: commands succeeded
  congratulations :)
```

Write comments as complete sentences.

Every Python code file must contain:

```
from __future__ import absolute_import, division
```

Tests

All code changes must be accompanied by unit tests with 100% code coverage (as measured by the tool coverage.)

To test code coverage you'll need to install detox and coverage. They can be installed by running:

```
pip install --user requirements/toolchain.txt
```

(Or you may prefer to install those requirements in a `virtualenv`.)

Then run:

```
$ coverage erase \
  && detox \
  && coverage combine \
  && coverage html
```

And open `htmlcov/index.html` in your web browser.

Documentation

All features should be documented with prose in the `docs` section. To ensure it builds and passes style checks you can run doc8 against it or run our `tox` job to lint docs. We also provide a spell-check job for docs:

```
$ tox -e docs
  docs: commands succeeded
  congratulations :)

$ tox -e docs-spellcheck
  docs-spellcheck: commands succeeded
  congratulations :)

$ tox -e docs-linkcheck
  docs-linkcheck: commands succeeded
  congratulations :)
```

The spell-check can catch jargon or abbreviations - if you are sure it is not an error, please add that word to the `spelling_wordlist.txt` in alphabetical order.

Docstrings

Docstrings generally follow pep257, with a few exceptions. They should be written like this:

```
def some_function(some_arg):
    """
    Does some things.

    :param some_arg: Some argument.
    """
```

So, specifically:

- Always use three double quotes.
- Put the three double quotes on their own line.
- No blank line at the end.
- Use Sphinx parameter/attribute documentation syntax.

The same job that lints code also lints docstrings:

```
$ tox -e lint
...
  lint: commands succeeded
  congratulations :)
```

Documentation Documentation

Here are some things we're going to document and how we're going to document them.

There should be a document describing how to write your own API mock plugin.

- reference the Twisted documentation for writing plugins as a prerequisite

- provide an example of the file that goes into *mimic/plugins*, *dummy_plugin.py* is a good start but it fails to explain what a real resource might look like that responds to API requests
  - unlike *dummy_plugin.py*, the *IAPIMock* implementer class ought to live in an application module, the plugin should only contain the instantiation.
  - better naming might help explain why you have to make separate “API mock” and “Region” objects.
- explain what a twisted web resource is, reference twisted docs for the interface and klein docs for how to produce your own resource with nice route decorators
- implement catalog_entries
  - are you implementing a crazy endpoint that manipulates the tenant ID? manipulate it here.
    - remember tenant ID may be None, because the core needs to allocate URI prefixes before there are any tenants. maybe this really ought to be a different API; the difference in meaning is that `catalog_entries(None)` must enumerate all regions that any user might possibly ever be able to access, whereas `catalog_entries(not_none)` can return different regions for different tenants if it wants to (for example, simulating limited availability).
      - this is super confusing, and technically unnecessary; it just made the implementation slightly easier to do in the first place. really all the URI prefixes could be allocated on demand as individual tenants receive particular services/region pairs for the first time, obviating the need for this weird implementation detail. possibly file an issue for this and just fix it before writing the final docs so we don’t have to explain.
  - return entries. canonical region right now is “ORD” but hopefully we can eventually change this API at some point to support a suggested list of regions from some configuration, pass in that list and then honor it here.
- implement resource_for_region
  - this gets called on every request
  - guideline: use klein to build your hierarchy since that makes it easier
    - note that you can always use whatever twisted.web resources make sense, if you want to toss a static hierarchy in there you don’t need to use klein, just make a static.File or a static.Data (perhaps note this at the end?)
  - to-do implementation-wise: we really ought to have a JSON serializer as a decorator or something so everybody doesn’t have to actually type “`dumps`” all the time
  - note where the “region” is in the hierarchy; there will be some URI prefix which you hopefully don’t care about (passed in as the `resource_for_region` argument) but you have to handle all segments from the end of the thing that Mimic has allocated for you: including your “prefix”; so if your “prefix” to the Endpoint construction in `catalog_entries` is “v5” then your routes need to all begin “/v5/<string:tenant_id>/”. The object returned from _that_ route is actually the specific tenant’s service endpoint.
  - implementation note: t.w.resource lifecycle management is weird, and a bit hard to explain. It would be nice to tell the developer at this point that they can store some kind of state on the session or associated with the tenant, but given that each resource is implicitly created with each request, it’s a bit tricky to do that. perhaps we should expose the “session” object we’re already keeping around in MimicCore to application code, or a dictionary associated with it, so that we can easily have per-tenant state.

Change Logs

Next Version

- The Cinder V2 API now has limited support for the List volumes with details endpoint.

Index