arsenal Documentation

Release

Rackspace

January 11, 2017

Contents

Contents:

CHAPTER 1

Arsenal - The Ironic image caching service

1.1 About

A small, Openstack-y service designed to manage image caching to nodes in Ironic, written in Python.

Pluggable data-gathering and cache management strategy means Arsenal can be repurposed to work with other services.

1.2 Features

- Pluggable data gathering.
- Pluggable strategy/decisioning around caching images to nodes.
- Built-in objects which provide client caching and API call retries to: Ironic, Nova, and Glance.

1.3 Documentation

Hosted HTML docs for Arsenal are available at http://arsenal.readthedocs.org/

You may also build a local copy of Arsenal's documentation by using Sphinx:

```
$ sphinx-build $repo_root/docs/source $output_dir
```

Then you can read the local documentation by pointing a browser at `$output_dir/index.html`

1.4 Roadmap

See issues labeled ‘enhancements‘ on Arsenal's Github project issues page.

Installation

2.1 Deployment

The easiest way to install arsenal is through `pip`. See pip's documentation on installation to obtain it.

To install arsenal, run the following command:

```
$ pip install --pre arsenal-director
```

Here, we're instructing `pip` to install a package, namely `arsenal-director`. `arsenal-director` is the package name for Arsenal in Python's Package Index. The `--pre` option lets `pip` install pre-release versions of packages.

Currently, Arsenal is in beta. Once beta testing is complete, a full release will be made and the `--pre` option can be dropped from the above example.

2.2 Development

To obtain Arsenal for development, you should clone it directly from Github:

```
$ git clone https://github.com/rackerlabs/arsenal.git
```

For a development installation, navigate to the cloned repository's root and run:

```
$ pip install -e .
```

Should provide Arsenal to the system, while allowing development changes to your local repository to be reflected in the installed package.

Or, if you have virtualenvwrapper installed, and would like to place Arsenal in a virtual Python environment:

```
$ mkvirtualenv arsenal
$ pip install -e .
```

Should do the trick.

CHAPTER 3

Configuring Arsenal

3.1 arsenal.conf

Arsenal reads configuration variables from a single file, canonically called `arsenal.conf`. The location and name of this configuration file can be changed as long as the `--config-file` argument to `arsenal-director` is set accordingly.

3.1.1 Oslo/Config

Arsenal uses the oslo.config module to parse and load configuration. See oslo.config's documentation for detailed information on the supported syntax.

3.1.2 Basic Syntax

That said, the basic syntax of the configuration file is fairly straight-forward.

Sections

Arsenal's configuration file is separated into sections. Each section begins with a bracket enclosed string. For example:

```
[director]
```

Begins the "director" configuration section. Each line following the section directive will populate that section's configuration options, until another section directive is parsed, or the end of the file is reached.

Options

Each option has this basic format:

```
<option_name>=<value>
```

Where <option_name> is the option's name, and <value> is the value to assign to the option.

A Short Example

The following:

```
[director]
dry_run=True
```

Would set the dry_run option to the boolean value 'True', which belongs to the [director] section.

3.2 arsenal.conf Sections

There are several sections which comprise arsenal.conf. You may not need to include every available section, nor set every option. Please read through the following section descriptions to get a sense for what functionality is made available through arsenal.conf.

3.2.1 [director] Section

The [director] section contains options which affect how arsenal-director gathers information using Scouts.

Note: See scheduler.py for all [director] configuration options.

Important Section Options

- **scout** - Configures which *Scout* will be loaded by arsenal-director to gather data from services. The Scout also currently handles issuing directives to endpoints. The format is:

```
    <scout_module_name>.<ScoutClassName>
```

  For example, setting the **scout** option to:

```
    devstack_scout.DevstackScout
```

  Would cause arsenal-director to use the *DevStack Scout*, which is a Scout provided by Arsenal that is designed to work with DevStack.

- **dry_run** - A boolean option. Setting this option to True will cause arsenal-director to run in dry run mode. Which means no directives generated by the configured Strategy will be issued.

Tip: **dry_run** is a great option to use while testing Arsenal without worrying about affecting outside services beyond requesting information.

- **directive_spacing** - An integer option. Represents time in seconds. Determines how long the Director will wait between issuing new directives returned by the configured Strategy.
- **log_statistics** - A boolean option. If True, Arsenal will log detailed statistics about nodes at the INFO level every time Arsenal issues directives. Statistics include: number of provisioned nodes, number of available nodes, number of cached nodes, a breakdown of which images have been cached and at what frequency. Also logs information broken up by flavor of node. If False, no statistics will be logged. Defaults to True.

Cache Node Directive Rate Limiting

The next two options are related to limiting how many *Cache Node* directives Arsenal will issue within a given period of time. They are tightly coupled and should be set together.

- **cache_directive_rate_limit** - An integer option limiting how many cache directives Arsenal will issue within a period of time delimited by **cache_directive_limiting_period**. Defaults to 0, which indicates no rate limiting of cache directives will occur.
- **cache_directive_limiting_period** - An integer option denoting the period of time, in seconds, to limit Arsenal issuing cache directives to the limit set by **cache_directive_rate_limit**. Once this period of time passes, Arsenal will again issue cache directives (if the configured *Strategy* is returning cache directives) until the rate limit is reached, or until the current time period again passes.

Eject Node Directive Rate Limiting

The next two options are related to limiting how many *Eject Node* directives Arsenal will issue within a given period of time. They are tightly coupled and should be set together.

Tip: These two options operate identically as the cache directive rate limiting options presented above. Except they apply to ejection directives.

- **eject_directive_rate_limit** - An integer option limiting how many eject directives Arsenal will issue within a period of time delimited by **eject_directive_limiting_period**. Defaults to 0, which indicates no rate limiting of eject directives will occur.
- **eject_directive_limiting_period** - An integer option denoting the period of time, in seconds, to limit Arsenal issuing eject directives to the limit set by **eject_directive_rate_limit**. Once this period of time passes, Arsenal will again issue eject directives (if the configured *Strategy* is returning eject directives) until the rate limit is reached, or until the current time period again passes.

3.2.2 [strategy] Section

This section provides configuration options relevant to all *Strategy* objects.

module_class

The **module_class** option controls which Strategy object is loaded and subsequently used to provide Arsenal's cache decisions. The format of the **module_class** option is as follows:

```
<strategy_module_name>.<StrategyClassName>
```

For example, the default value for **module_class** is:

```
simple_proportional_strategy.SimpleProportionalStrategy
```

This causes the the class `SimpleProportionalStrategy`, which can be found in the `simple_proportional_strategy` module, to be instantiated and used by `arsenal-director` to provide cache decisions at run-time. The `simple_proportional_strategy` module is included as part of Arsenal.

Astute readers will notice the the syntax of this option matches that of **scout** from the `[director]` section.

image_weights_filename

image_weights_filename is a string option specifying the location of a text file containing a single JSON object where the object's keys are names of images as strings, and the values are the associated weights as non-negative integers. This JSON object is loaded as a Python dictionary and then referred to by Arsenal whenever a built-in image selection function, such as `arsenal.strategy.choose_weighted_images_force_distribution`, has to make a decision on which image(s) to choose to cache to available nodes.

Important: The keys of the JSON object in the file named by **image_weights_filename** must exactly match the names of images as reported by the configured Scout object. This typically means image names reported by Glance. Otherwise the configured weights will not be properly applied.

Images with higher weights will tend to be picked more frequently, and similarly those with lower weights will tend to be picked less frequently.

Note: If **image_weights_filename** is not defined, then every image will receive the weight specified by the **default_image_weight** option. Meaning every image will have an equal chance of being cached.

Example JSON object containing image weights:

```
{
    'Ubuntu': 10,
    'CoreOS': 5,
    'Windows': 2,
    'SteamOS': 1
}
```

In the above example the `Ubuntu` image will be picked twice as often as the `CoreOS` image, and ten times as often as the `SteamOS` image. If you had 18 nodes to cache, then you can reasonably expect 10 nodes to have the `Ubuntu` image cached, 5 nodes to have the `CoreOS` image cached, and so on.

An *example image weight json file* is available in Arsenal's source tree.

default_image_weight

default_image_weight is an integer value which is used to weight an image with no corresponding entry in the JSON object loaded by the **image_weights_filename** option. Defaults to 1.

3.2.3 [simple_proportional_strategy] Section

Currently, the `SimpleProportionalStrategy` class is the only concrete implementation of `strategy.Strategy` provided by Arsenal.

See the *SimpleProportionalStrategy* section for more information on this *Strategy*.

Important Section Options

percentage_to_cache - A floating point number. Valid values range from 0 to 1 inclusive. 0 corresponds to 0%, and 1 corresponds to 100%. Controls the percentage of unprovisioned/available nodes of a particular flavor to be cached at a particular time.

3.2.4 [client_wrapper] Section

The `[client_wrapper]` section contains options relevant to the Openstack client wrapper provided by Arsenal. Arsenal provides service-specific client wrappers for Ironic, Nova, and Glance.

The client wrappers provided by Arsenal all provide client caching and call-retry behavior. This section provides options to configure part of that behavior as well as provide credentials to all wrappers.

Note: Please see client_wrapper.py for all `[client_wrapper]` configuration options.

Important: Credential options defined in the client_wrapper section will be used by default by every derived instance of client wrapper unless the credential is overridden in the derived client wrapper's section. For instance, if **os_username** is defined in the `[client_wrapper]` section, then the Nova client wrapper will use the `client_wrapper.os_username` value unless `nova.admin_username` is defined.

Important Section Options

- **call_max_retries** - An integer value which determines how many times an individual client will be retried, until it is successful.
- **call_retry_interval** - An integer value which Determines how long the client wrapper will wait before trying a call again.

3.2.5 [nova] Section

This section provides options mainly relating to credentials and the endpoint to use to communicate with Nova.

Note: Please see nova_client_wrapper.py for all `[nova]` configuration options.

3.2.6 [ironic] Section

This section provides options mainly relating to credentials and the endpoint to use to communicate with Ironic.

Note: Please see ironic_client_wrapper.py for all `[ironic]` configuration options.

3.2.7 [glance] Section

This section provides options mainly relating to credentials and the endpoint to use to communicate with Glance.

Note: Please see glance_client_wrapper.py for all `[glance]` configuration options.

3.3 A full example arsenal.conf file

See the example Arsenal configuration in the Arsenal source tree to see a full example configuration to use with `arsenal-director`.

CHAPTER 4

Usage

4.1 arsenal-director

Arsenal is invoked by running:

```
arsenal-director
```

With various arguments. All of `arsenal-director`'s supported arguments are documented on the command line. Run:

```
arsenal-director --help
```

To see them, and a brief explanation on each one.

A reasonable invocation for actual use looks something like:

```
arsenal-director --config-file /etc/arsenal/arsenal.conf --log-file /car/log/arsenal/arsenal-directo
```

Which would start `arsenal-director`, and it would try to load the configuration file found at `/etc/arsenal/arsenal.conf` while logging to `/var/log/arsenal/arsenal-director.log`.

`arsenal-director` will periodically gather data using the configured *Scout* object, and issuing directives returned by the configured *Strategy* object. `arsenal-director` will continue in this way indefinitely, only stopping through program termination.

Important: It's a good idea to set the *dry_run option* to `True` in order to prevent `arsenal-director` from issuing directives until you are confident that all the configuration settings appear to be correct, and the directives emitted by the configured Strategy are consistent with expected behavior.

CHAPTER 5

Design

The core of Arsenal's functionality consists of gathering data for input, through *Scout* objects, to send to Arsenal's caching *Strategy* objects, which produce directives, which in turn are currently fulfilled by *Scout* objects.

Therefore, Scouts deal with the outside world, while Strategies provide introspection on data provided by Scouts to direct image caching on nodes in some meaningful way. The Scout and Strategy objects used by `arsenal-director` can be changed through configuration options.

Arsenal's design philosophy can be summed up as: "Provide a way to do something, but make it easy to change or swap out."

5.1 Scout

The responsibility of Scouts are to gather data from various outside sources, like Ironic, Nova, and Glance, convert that data to a form suitable for Strategy object consumption, as well as issue directives to endpoints, such as Ironic.

All of Arsenal's Scout objects are derived from an abstract base class called Scout, which is defined in scout.py.

Tip: If you are thinking about defining your own Scout object, reading scout.py is a good place to start.

A couple of pre-made Scouts are currently included in Arsenal.

5.1.1 Openstack Scout

The Openstack Scout will communicate with Ironic, Nova, and Glance services, and handle fulfilling Strategy actions by talking to Ironic.

Most Scouts hoping to be used with Openstack services will derive from this Scout while passing filtration functions for flavors and images to OpenstackScout via a super() call during __init__.

For more information see, openstack_scout.py.

5.1.2 DevStack Scout

This Scout is designed to be used with the DevStack project, which provides a relatively easy way to setup an Openstack-based environment on a single machine, typically for testing purposes.

See Ironic documentation on how to configure virtual baremetal nodes for use with DevStack.

For more information see, devstack_scout.py.

5.1.3 OnMetal Scouts

The OnMetal Scouts are designed to work with Rackspace's OnMetal product. While these specific Scouts will probably not be directly useful to anyone outside of Rackspace, it can still be instructive to view fully functional implementations of *Openstack Scout* with filters.

For more information, see onmetal_scout.py.

5.2 Strategy

A Strategy's role lies in consuming data provided by Scouts, and then emitting directives to manage imaging caching on nodes.

5.2.1 Directives

Currently, two directives are used by Arsenal's strategies to manage the cache. *Cache Node*, which adds a node to the cache, and *Eject Node*, which will remove a node from the cache.

Cache Node

CacheNode instructs the endpoint to cache a specific image onto a specific node. This is the main mechanism used to build a fleet of cached nodes.

Eject Node

The second is **EjectNode**, which instructs the endpoint to do whatever is necessary to put a previously cached node back into an uncached state. This directive is necessary if an image cached to a node becomes out-of-date.

Tip: If you are thinking about defining your own Strategy object, reading strategy/base.py is a good place to start.

5.2.2 SimpleProportionalStrategy

Currently, SimpleProportionalStrategy is the only Strategy shipping with Arsenal.

This object implements a fairly straight-forward strategy: For each available flavor of node, use a constant proportion of available nodes for caching.

SimpleProportionalStrategy randomly picks available, uncached nodes to cache. The random selection is designed to level wear across nodes.

Image selection is handled by `choose_weighted_images_force_distribution` found in the `arsenal.strategy.base` module. This means SimpleProportionalStrategy will pick images by weights pulled from the `strategy.image_weights_filename` option. See the *image_weights_filename* option section for more details on how image weighting works in Arsenal.

See the *[simple_proportional_strategy] Section* for information on how to configure this Strategy.

CHAPTER 6

Contributing

Contributions are encouraged and welcome!

For any type of change, please follow this general workflow:

1. Open an issue in Arsenal's Github issue tracker. Describe the issue and tag it accordingly. That is, if the issue is a bug, please tag the issue as a bug. If an issue already exists, skip this step.
2. Clone Arsenal's repository locally.
3. Create a topic branch for the changes you plan to make in regards to the issue you're working on: `git checkout -b your_branch_name`
4. Make your changes.
5. Add appropriate unit-tests. If your change addresses a bug, please add a unit test that proves the bug is fixed by your change. For enhancements, try to thoroughly test all cases the new code will face.
6. Make sure all unit tests pass. `tox -epy27,pep8` should exercise all unit-tests and check for pep8 related style issues.
7. Commit your changes to your local repository and reference the appropriate Github issue in your commit message, if appropriate.
8. Push your topic branch: `your_banch_name` to Github.
9. Create a pull request using the `your_branch_name` branch.

At that point, a repository maintainer will need to review and approve the pull-request. You may be asked to make additional changes to your pull-request before it is merged.

Please note that any contributions will fall under the Apache 2.0 license governing this project.

Thanks for contributing!

CHAPTER 7

Indices and tables

- genindex
- modindex
- search