
Horizon Documentation

Release 20.1.5.dev6

OpenStack Foundation

May 26, 2023

CONTENTS

1	Introduction	1
2	Using Horizon	3
2.1	Installation Guide	3
2.1.1	System Requirements	3
	System Requirements	3
2.1.2	Installing from Packages	4
	Install and configure for Debian	4
	Install and configure for openSUSE and SUSE Linux Enterprise	6
	Install and configure for Red Hat Enterprise Linux and CentOS	9
	Install and configure for Ubuntu	11
	Verify operation for Debian	14
	Verify operation for openSUSE and SUSE Linux Enterprise	14
	Verify operation for Red Hat Enterprise Linux and CentOS	14
	Verify operation for Ubuntu	14
	Next steps	14
2.1.3	Installing from Source	15
	Manual installation	15
2.1.4	Horizon plugins	19
	Plugin Registry	19
2.2	Configuration Guide	20
2.2.1	Settings Reference	20
	Introduction	20
	General Settings	20
	Service-specific Settings	37
	Django Settings	60
	Other Settings	63
2.2.2	Pluggable Panels and Groups	64
	Introduction	64
	General Pluggable Settings	64
	Pluggable Settings for Dashboards	67
	Pluggable Settings for Panels	68
	Pluggable Settings for Panel Groups	70
2.2.3	Customizing Horizon	71
	Changing the Site Title	71
	Changing the Brand Link	71
	Customizing the Footer	71
	Modifying Existing Dashboards and Panels	72
	Horizon customization module (overrides)	72

	Customize the project and user table columns	74
	Customize Angular dashboards	74
	Icons	76
	Custom Stylesheets	77
	Custom Javascript	77
	Customizing Meta Attributes	79
2.2.4	Themes	79
	Inherit from an Existing Theme	80
	Organizing Your Theme Directory	81
	Customizing the Logo	81
2.2.5	Branding Horizon	82
	Supported Components	82
	Step 1	83
	Top Navbar	83
	Side Nav	83
	Charts	83
	Tables	84
	Login	84
	Tabs	84
	Alerts	84
	Checkboxes	85
	Bootstrap and Material Design	85
	Development Tips	85
2.3	OpenStack Dashboard User Documentation	85
2.3.1	Log in to the dashboard	85
	OpenStack dashboard Project tab	86
	OpenStack dashboard Admin tab	88
	OpenStack dashboard Identity tab	89
	OpenStack dashboard Settings tab	90
2.3.2	Upload and manage images	90
	Upload an image	91
	Update an image	92
	Delete an image	93
2.3.3	Configure access and security for instances	93
	Add a rule to the default security group	94
	Add a key pair	95
	Import a key pair	95
	Allocate a floating IP address to an instance	96
2.3.4	Launch and manage instances	96
	Launch an instance	97
	Connect to your instance by using SSH	99
	Track usage for instances	100
	Create an instance snapshot	100
	Manage an instance	100
2.3.5	Create and manage networks	101
	Create a network	101
	Create a router	102
	Create a port	102
2.3.6	Create and manage object containers	103
	Create a container	103
	Upload an object	103

	Manage an object	104
2.3.7	Create and manage volumes	105
	Create a volume	105
	Attach a volume to an instance	106
	Detach a volume from an instance	106
	Create a snapshot from a volume	107
	Edit a volume	107
	Delete a volume	107
2.3.8	Supported Browsers	108
2.4	Administration Guide	108
2.4.1	Customize and configure the Dashboard	108
	Customize the Dashboard	109
	Configure the Dashboard	111
2.4.2	Set up session storage for the Dashboard	117
	Local memory cache	117
	Cached database	119
	Cookies	119
2.4.3	Create and manage images	120
	Create images	120
	Update images	122
	Delete images	122
2.4.4	Create and manage roles	122
	Create a role	122
	Edit a role	123
	Delete a role	123
2.4.5	Manage projects and users	123
	Add a new project	123
	Delete a project	124
	Update a project	124
	Add a new user	124
	Delete a new user	124
	Update a user	125
2.4.6	Manage instances	125
	Create instance snapshots	125
	Control the state of an instance	126
	Track usage	126
2.4.7	Manage flavors	126
	Create flavors	127
	Update flavors	129
	Update Metadata	129
	Delete flavors	130
2.4.8	Manage volumes and volume types	130
	Create a volume type	130
	Create an encrypted volume type	130
	Delete volume types	132
	Delete volumes	133
2.4.9	View and manage quotas	133
	View default project quotas	134
	Update project quotas	134
2.4.10	View services information	134
2.4.11	Create and manage host aggregates	135

	To create a host aggregate	135
	To manage host aggregates	135
3	Contributor Docs	137
3.1	Contributor Documentation	137
3.1.1	So You Want to Contribute	137
	Project Resources	137
	Communication	137
	Contacting the Core Team	138
	New Feature Planning	138
	Task Tracking	138
	Reporting a Bug	138
	Getting Your Patch Merged	139
	Project Team Lead Duties	139
	Etiquette	139
3.1.2	Horizon Basics	139
	Values	139
	History	140
	The Current Architecture & How It Meets Our Values	140
3.1.3	Project Policies	141
	Supported Software	141
	Horizon Teams	142
	Core Reviewer Team	143
	Horizon Bugs	144
3.1.4	Quickstart	146
	Linux Systems	146
	Setup	146
	Managing Settings	147
	Editing Horizons Source	148
	Horizons Structure	148
	Project Structure	148
	Application Design	149
3.1.5	Horizons tests and you	151
	How to run the tests	151
	tox Test Environments	152
	Writing tests	154
3.1.6	Tutorials	154
	Tutorial: Creating an Horizon Plugin	154
	Tutorial: Building a Dashboard using Horizon	163
	Tutorial: Adding a complex action to a table	173
	Extending an AngularJS Workflow	179
3.1.7	Topic Guides	183
	Code Style	183
	Workflows Topic Guide	190
	DataTables Topic Guide	192
	Horizon Policy Enforcement (RBAC: Role Based Access Control)	199
	Horizon Microversion Support	203
	AngularJS Topic Guide	204
	Testing Overview	210
	Styling in Horizon (SCSS)	221
	Release Notes	223

Translation in Horizon	224
Profiling Pages	230
Defining default settings in code	231
Packaging Software	234
DevStack for Horizon	238
3.1.8 Module Reference	239
Horizon Framework	239
openstack_auth Module	283
3.1.9 Frequently Asked Questions	291
4 Release Notes	293
5 Information	295
5.1 Glossary	295

INTRODUCTION

Horizon is the canonical implementation of [OpenStacks Dashboard](#), which provides a web based user interface to OpenStack services including Nova, Swift, Keystone, etc.

For a more in-depth look at Horizon and its architecture, see the *Horizon Basics*.

To learn what you need to know to get going, see the *Quickstart*.

USING HORIZON

How to use Horizon in your own projects.

2.1 Installation Guide

This section describes how to install and configure the dashboard on the controller node.

The only core service required by the dashboard is the Identity service. You can use the dashboard in combination with other services, such as Image service, Compute, and Networking. You can also use the dashboard in environments with stand-alone services such as Object Storage.

Note: This section assumes proper installation, configuration, and operation of the Identity service using the Apache HTTP server and Memcached service.

2.1.1 System Requirements

System Requirements

The Ussuri release of horizon has the following dependencies.

- Python 3.6 or 3.7
- Django 2.2
 - Django 3.2 support is experimental as of Xena release. (Yoga release will use Django 3.2 as the primary Django version.)
 - Django support policy is documented at *Django support*.
- An accessible [keystone](#) endpoint
- All other services are optional. Horizon supports the following services as of the Stein release. If the keystone endpoint for a service is configured, horizon detects it and enables its support automatically.
 - [cinder](#): Block Storage
 - [glance](#): Image Management
 - [neutron](#): Networking
 - [nova](#): Compute

- `swift`: Object Storage
- Horizon also supports many other OpenStack services via plugins. For more information, see the *Plugin Registry*.

2.1.2 Installing from Packages

Install and configure for Debian

This section describes how to install and configure the dashboard on the controller node.

The only core service required by the dashboard is the Identity service. You can use the dashboard in combination with other services, such as Image service, Compute, and Networking. You can also use the dashboard in environments with stand-alone services such as Object Storage.

Note: This section assumes proper installation, configuration, and operation of the Identity service using the Apache HTTP server and Memcached service.

Install and configure components

Note: Default configuration files vary by distribution. You might need to add these sections and options rather than modifying existing sections and options. Also, an ellipsis (. . .) in the configuration snippets indicates potential default configuration options that you should retain.

1. Install the packages:

```
# apt install openstack-dashboard-apache
```

2. Respond to prompts for web server configuration.

Note: The automatic configuration process generates a self-signed SSL certificate. Consider obtaining an official certificate for production environments.

Note: There are two modes of installation. One using `/horizon` as the URL, keeping your default `vhost` and only adding an `Alias` directive: this is the default. The other mode will remove the default Apache `vhost` and install the dashboard on the `webroot`. It was the only available option before the Liberty release. If you prefer to set the Apache configuration manually, install the `openstack-dashboard` package instead of `openstack-dashboard-apache`.

3. Edit the `/etc/openstack-dashboard/local_settings.py` file and complete the following actions:
 - Configure the dashboard to use OpenStack services on the controller node:

```
OPENSTACK_HOST = "controller"
```

- In the Dashboard configuration section, allow your hosts to access Dashboard:

```
ALLOWED_HOSTS = ['one.example.com', 'two.example.com']
```

Note:

- Do not edit the ALLOWED_HOSTS parameter under the Ubuntu configuration section.
 - ALLOWED_HOSTS can also be ['*'] to accept all hosts. This may be useful for development work, but is potentially insecure and should not be used in production. See the [Django documentation](#) for further information.
-

- Configure the memcached session storage service:

```
SESSION_ENGINE = 'django.contrib.sessions.backends.cache'

CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.
↔MemcachedCache',
        'LOCATION': 'controller:11211',
    }
}
```

Note: Comment out any other session storage configuration.

- Enable the Identity API version 3:

```
OPENSTACK_KEYSTONE_URL = "http://%s/identity/v3" % OPENSTACK_HOST
```

Note: In case your keystone run at 5000 port then you would mentioned keystone port here as well i.e. OPENSTACK_KEYSTONE_URL = http://%s:5000/identity/v3 % OPENSTACK_HOST

- Enable support for domains:

```
OPENSTACK_KEYSTONE_MULTIDOMAIN_SUPPORT = True
```

- Configure API versions:

```
OPENSTACK_API_VERSIONS = {
    "identity": 3,
    "image": 2,
    "volume": 3,
}
```

- Configure Default as the default domain for users that you create via the dashboard:

```
OPENSTACK_KEYSTONE_DEFAULT_DOMAIN = "Default"
```

- Configure user as the default role for users that you create via the dashboard:

```
OPENSTACK_KEYSTONE_DEFAULT_ROLE = "user"
```

- If you chose networking option 1, disable support for layer-3 networking services:

```
OPENSTACK_NEUTRON_NETWORK = {  
    ...  
    'enable_router': False,  
    'enable_quotas': False,  
    'enable_ipv6': False,  
    'enable_distributed_router': False,  
    'enable_ha_router': False,  
    'enable_fip_topology_check': False,  
}
```

- Optionally, configure the time zone:

```
TIME_ZONE = "TIME_ZONE"
```

Replace `TIME_ZONE` with an appropriate time zone identifier. For more information, see the [list of time zones](#).

Finalize installation

- Reload the web server configuration:

```
# service apache2 reload
```

Install and configure for openSUSE and SUSE Linux Enterprise

This section describes how to install and configure the dashboard on the controller node.

The only core service required by the dashboard is the Identity service. You can use the dashboard in combination with other services, such as Image service, Compute, and Networking. You can also use the dashboard in environments with stand-alone services such as Object Storage.

Note: This section assumes proper installation, configuration, and operation of the Identity service using the Apache HTTP server and Memcached service.

Install and configure components

Note: Default configuration files vary by distribution. You might need to add these sections and options rather than modifying existing sections and options. Also, an ellipsis (. . .) in the configuration snippets indicates potential default configuration options that you should retain.

1. Install the packages:

```
# zypper install openstack-dashboard
```

2. Configure the web server:

```
# cp /etc/apache2/conf.d/openstack-dashboard.conf.sample \
/etc/apache2/conf.d/openstack-dashboard.conf
# a2enmod rewrite
```

3. Edit the `/srv/www/openstack-dashboard/openstack_dashboard/local/local_settings.py` file and complete the following actions:

- Configure the dashboard to use OpenStack services on the controller node:

```
OPENSTACK_HOST = "controller"
```

- Allow your hosts to access the dashboard:

```
ALLOWED_HOSTS = ['one.example.com', 'two.example.com']
```

Note: `ALLOWED_HOSTS` can also be `['*']` to accept all hosts. This may be useful for development work, but is potentially insecure and should not be used in production. See [Django documentation](#) for further information.

- Configure the memcached session storage service:

```
SESSION_ENGINE = 'django.contrib.sessions.backends.cache'

CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.
↪MemcachedCache',
        'LOCATION': 'controller:11211',
    }
}
```

Note: Comment out any other session storage configuration.

- Enable the Identity API version 3:

```
OPENSTACK_KEYSTONE_URL = "http://%s/identity/v3" % OPENSTACK_HOST
```

Note: In case your keystone run at 5000 port then you would mentioned keystone port here as well i.e. `OPENSTACK_KEYSTONE_URL = http://%s:5000/identity/v3 % OPENSTACK_HOST`

- Enable support for domains:

```
OPENSTACK_KEYSTONE_MULTIDOMAIN_SUPPORT = True
```

- Configure API versions:

```
OPENSTACK_API_VERSIONS = {  
    "identity": 3,  
    "image": 2,  
    "volume": 3,  
}
```

- Configure `Default` as the default domain for users that you create via the dashboard:

```
OPENSTACK_KEYSTONE_DEFAULT_DOMAIN = "Default"
```

- Configure `user` as the default role for users that you create via the dashboard:

```
OPENSTACK_KEYSTONE_DEFAULT_ROLE = "user"
```

- If you chose networking option 1, disable support for layer-3 networking services:

```
OPENSTACK_NEUTRON_NETWORK = {  
    ...  
    'enable_router': False,  
    'enable_quotas': False,  
    'enable_distributed_router': False,  
    'enable_ha_router': False,  
    'enable_fip_topology_check': False,  
}
```

- Optionally, configure the time zone:

```
TIME_ZONE = "TIME_ZONE"
```

Replace `TIME_ZONE` with an appropriate time zone identifier. For more information, see the [list of time zones](#).

Finalize installation

- Restart the web server and session storage service:

```
# systemctl restart apache2.service memcached.service
```

Note: The `systemctl restart` command starts each service if not currently running.

Install and configure for Red Hat Enterprise Linux and CentOS

This section describes how to install and configure the dashboard on the controller node.

The only core service required by the dashboard is the Identity service. You can use the dashboard in combination with other services, such as Image service, Compute, and Networking. You can also use the dashboard in environments with stand-alone services such as Object Storage.

Note: This section assumes proper installation, configuration, and operation of the Identity service using the Apache HTTP server and Memcached service.

Install and configure components

Note: Default configuration files vary by distribution. You might need to add these sections and options rather than modifying existing sections and options. Also, an ellipsis (. . .) in the configuration snippets indicates potential default configuration options that you should retain.

1. Install the packages:

```
# yum install openstack-dashboard
```

2. Edit the `/etc/openstack-dashboard/local_settings` file and complete the following actions:

- Configure the dashboard to use OpenStack services on the controller node:

```
OPENSTACK_HOST = "controller"
```

- Allow your hosts to access the dashboard:

```
ALLOWED_HOSTS = ['one.example.com', 'two.example.com']
```

Note: `ALLOWED_HOSTS` can also be `[*]` to accept all hosts. This may be useful for development work, but is potentially insecure and should not be used in production. See <https://docs.djangoproject.com/en/dev/ref/settings/#allowed-hosts> for further information.

- Configure the memcached session storage service:

```
SESSION_ENGINE = 'django.contrib.sessions.backends.cache'

CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.
↪MemcachedCache',
        'LOCATION': 'controller:11211',
    }
}
```

Note: Comment out any other session storage configuration.

- Enable the Identity API version 3:

```
OPENSTACK_KEYSTONE_URL = "http://%s/identity/v3" % OPENSTACK_HOST
```

Note: In case your keystone run at 5000 port then you would mentioned keystone port here as well i.e. `OPENSTACK_KEYSTONE_URL = http://%s:5000/identity/v3 % OPENSTACK_HOST`

- Enable support for domains:

```
OPENSTACK_KEYSTONE_MULTIDOMAIN_SUPPORT = True
```

- Configure API versions:

```
OPENSTACK_API_VERSIONS = {
    "identity": 3,
    "image": 2,
    "volume": 3,
}
```

- Configure `Default` as the default domain for users that you create via the dashboard:

```
OPENSTACK_KEYSTONE_DEFAULT_DOMAIN = "Default"
```

- Configure `user` as the default role for users that you create via the dashboard:

```
OPENSTACK_KEYSTONE_DEFAULT_ROLE = "user"
```

- If you chose networking option 1, disable support for layer-3 networking services:

```
OPENSTACK_NEUTRON_NETWORK = {
    ...
    'enable_router': False,
    'enable_quotas': False,
    'enable_distributed_router': False,
    'enable_ha_router': False,
```

(continues on next page)

(continued from previous page)

```
'enable_fip_topology_check': False,  
}
```

- Optionally, configure the time zone:

```
TIME_ZONE = "TIME_ZONE"
```

Replace `TIME_ZONE` with an appropriate time zone identifier. For more information, see the [list of time zones](#).

3. Add the following line to `/etc/httpd/conf.d/openstack-dashboard.conf` if not included.

```
WSGIApplicationGroup %{GLOBAL}
```

Finalize installation

- Restart the web server and session storage service:

```
# systemctl restart httpd.service memcached.service
```

Note: The `systemctl restart` command starts each service if not currently running.

Install and configure for Ubuntu

This section describes how to install and configure the dashboard on the controller node.

The only core service required by the dashboard is the Identity service. You can use the dashboard in combination with other services, such as Image service, Compute, and Networking. You can also use the dashboard in environments with stand-alone services such as Object Storage.

Note: This section assumes proper installation, configuration, and operation of the Identity service using the Apache HTTP server and Memcached service.

Install and configure components

Note: Default configuration files vary by distribution. You might need to add these sections and options rather than modifying existing sections and options. Also, an ellipsis (...) in the configuration snippets indicates potential default configuration options that you should retain.

1. Install the packages:

```
# apt install openstack-dashboard
```

2. Edit the `/etc/openstack-dashboard/local_settings.py` file and complete the following actions:

- Configure the dashboard to use OpenStack services on the controller node:

```
OPENSTACK_HOST = "controller"
```

- In the Dashboard configuration section, allow your hosts to access Dashboard:

```
ALLOWED_HOSTS = ['one.example.com', 'two.example.com']
```

Note:

- Do not edit the `ALLOWED_HOSTS` parameter under the Ubuntu configuration section.
 - `ALLOWED_HOSTS` can also be `['*']` to accept all hosts. This may be useful for development work, but is potentially insecure and should not be used in production. See the [Django documentation](#) for further information.
-

- Configure the memcached session storage service:

```
SESSION_ENGINE = 'django.contrib.sessions.backends.cache'

CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.
↪MemcachedCache',
        'LOCATION': 'controller:11211',
    }
}
```

Note: Comment out any other session storage configuration.

- Enable the Identity API version 3:

```
OPENSTACK_KEYSTONE_URL = "http://%s/identity/v3" % OPENSTACK_HOST
```

Note: In case your keystone run at 5000 port then you would mentioned keystone port here as well i.e. `OPENSTACK_KEYSTONE_URL = http://%s:5000/identity/v3 % OPENSTACK_HOST`

- Enable support for domains:

```
OPENSTACK_KEYSTONE_MULTIDOMAIN_SUPPORT = True
```

- Configure API versions:

```
OPENSTACK_API_VERSIONS = {
    "identity": 3,
```

(continues on next page)

(continued from previous page)

```
"image": 2,  
"volume": 3,  
}
```

- Configure `Default` as the default domain for users that you create via the dashboard:

```
OPENSTACK_KEYSTONE_DEFAULT_DOMAIN = "Default"
```

- Configure `user` as the default role for users that you create via the dashboard:

```
OPENSTACK_KEYSTONE_DEFAULT_ROLE = "user"
```

- If you chose networking option 1, disable support for layer-3 networking services:

```
OPENSTACK_NEUTRON_NETWORK = {  
    ...  
    'enable_router': False,  
    'enable_quotas': False,  
    'enable_ipv6': False,  
    'enable_distributed_router': False,  
    'enable_ha_router': False,  
    'enable_fip_topology_check': False,  
}
```

- Optionally, configure the time zone:

```
TIME_ZONE = "TIME_ZONE"
```

Replace `TIME_ZONE` with an appropriate time zone identifier. For more information, see the [list of time zones](#).

3. Add the following line to `/etc/apache2/conf-available/openstack-dashboard.conf` if not included.

```
WSGIApplicationGroup %{GLOBAL}
```

Finalize installation

- Reload the web server configuration:

```
# systemctl reload apache2.service
```

Verify operation for Debian

Verify operation of the dashboard.

Access the dashboard using a web browser at `http://controller/horizon/`.

Authenticate using `admin` or `demo` user and `default` domain credentials.

Verify operation for openSUSE and SUSE Linux Enterprise

Verify operation of the dashboard.

Access the dashboard using a web browser at `http://controller/`.

Authenticate using `admin` or `demo` user and `default` domain credentials.

Verify operation for Red Hat Enterprise Linux and CentOS

Verify operation of the dashboard.

Access the dashboard using a web browser at `http://controller/dashboard`.

Authenticate using `admin` or `demo` user and `default` domain credentials.

Verify operation for Ubuntu

Verify operation of the dashboard.

Access the dashboard using a web browser at `http://controller/horizon`.

Authenticate using `admin` or `demo` user and `default` domain credentials.

Next steps

Your OpenStack environment now includes the dashboard.

After you install and configure the dashboard, you can complete the following tasks:

- Provide users with a public IP address, a username, and a password so they can access the dashboard through a web browser. In case of any SSL certificate connection problems, point the server IP address to a domain name, and give users access.
- Customize your dashboard. For details, see *Customize and configure the Dashboard*.
- Set up session storage. For details, see *Set up session storage for the Dashboard*.

- To use the VNC client with the dashboard, the browser must support HTML5 Canvas and HTML5 WebSockets.

For details about browsers that support noVNC, see [README](#).

2.1.3 Installing from Source

Manual installation

This page covers the basic installation of horizon in a production environment. If you are looking for a developer environment, see [Quickstart](#).

For the system dependencies, see [System Requirements](#).

Installation

Note: In the commands below, substitute <release> for your version of choice, such as queens or rocky. If you use the development version, replace stable/<release> with master.

1. Clone Horizon

```
$ git clone https://opendev.org/openstack/horizon -b stable/<release> --  
↪depth=1  
$ cd horizon
```

2. Install the horizon python module into your system

```
$ sudo pip install -c https://opendev.org/openstack/requirements/raw/  
↪branch/stable/<release>/upper-constraints.txt .
```

Configuration

This section contains a small summary of the critical settings required to run horizon. For more details, please refer to [Settings Reference](#).

Settings

Create `openstack_dashboard/local/local_settings.py`. It is usually a good idea to copy `openstack_dashboard/local/local_settings.py.example` and edit it. As a minimum, the following settings will need to be modified:

DEBUG Set to False

ALLOWED_HOSTS Set to your domain name(s)

OPENSTACK_HOST Set to the IP of your Keystone endpoint. You may also need to alter `OPENSTACK_KEYSTONE_URL`

Note: The following steps in the Configuration section are optional, but highly recommended in production.

Translations

Compile translation message catalogs for internationalization. This step is not required if you do not need to support languages other than US English. GNU `gettext` tool is required to compile message catalogs.

```
$ sudo apt install gettext
$ ./manage.py compilemessages
```

Static Assets

Compress your static files by adding `COMPRESS_OFFLINE = True` to your `local_settings.py`, then run the following commands

```
$ ./manage.py collectstatic
$ ./manage.py compress
```

Logging

Horizons uses Django's logging configuration mechanism, which can be customized by altering the `LOGGING` dictionary in `local_settings.py`. By default, Horizons logging example sets the log level to `INFO`.

Horizon also uses a number of 3rd-party clients which log separately. The log level for these can still be controlled through Horizons `LOGGING` config, however behaviors may vary beyond Horizons control.

For more information regarding configuring logging in Horizon, please read the [Django logging directive](#) and the [Python logging directive](#) documentation. Horizon is built on Python and Django.

Session Storage

Horizon uses Django's [sessions framework](#) for handling session data. There are numerous session backends available, which are selected through the `SESSION_ENGINE` setting in your `local_settings.py` file.

Memcached

```
SESSION_ENGINE = 'django.contrib.sessions.backends.cache'
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': 'controller:11211',
    }
}
```

External caching using an application such as memcached offers persistence and shared storage, and can be very useful for small-scale deployment and/or development. However, for distributed and high-availability scenarios memcached has inherent problems which are beyond the scope of this documentation.

Requirements:

- Memcached service running and accessible
- Python memcached module installed

Database

```
SESSION_ENGINE = 'django.core.cache.backends.db.DatabaseCache'
DATABASES = {
    'default': {
        # Database configuration here
    }
}
```

Database-backed sessions are scalable (using an appropriate database strategy), persistent, and can be made high-concurrency and highly-available.

The downside to this approach is that database-backed sessions are one of the slower session storages, and incur a high overhead under heavy usage. Proper configuration of your database deployment can also be a substantial undertaking and is far beyond the scope of this documentation.

Cached Database

To mitigate the performance issues of database queries, you can also consider using Django's `cached_db` session backend which utilizes both your database and caching infrastructure to perform write-through caching and efficient retrieval. You can enable this hybrid setting by configuring both your database and cache as discussed above and then using

```
SESSION_ENGINE = "django.contrib.sessions.backends.cached_db"
```

Deployment

1. Set up a web server with WSGI support. For example, install Apache web server on Ubuntu

```
$ sudo apt install apache2 libapache2-mod-wsgi
```

You can either use the provided `openstack_dashboard/wsgi.py` or generate a `openstack_dashboard/horizon_wsgi.py` file with the following command (which detects if you use a virtual environment or not to automatically build an adapted WSGI file)

```
$ ./manage.py make_web_conf --wsgi
```

Then configure the web server to host OpenStack dashboard via WSGI. For apache2 web server, you may need to create `/etc/apache2/sites-available/horizon.conf`. The template in DevStack is a good example of the file. <https://opendev.org/openstack/devstack/src/branch/master/files/apache-horizon.template> Or you can automatically generate an apache configuration file. If you previously generated an `openstack_dashboard/horizon_wsgi.py` file it will use that, otherwise will default to using `openstack_dashboard/wsgi.py`

```
$ ./manage.py make_web_conf --apache > /etc/apache2/sites-available/  
↪horizon.conf
```

Same as above but if you want SSL support

```
$ ./manage.py make_web_conf --apache --ssl --sslkey=/path/to/ssl/key --  
↪sslcert=/path/to/ssl/cert > /etc/apache2/sites-available/horizon.conf
```

By default the apache configuration will launch a number of apache processes equal to the number of CPUs + 1 of the machine on which you launch the `make_web_conf` command. If the target machine is not the same or if you want to specify the number of processes, add the `--processes` option

```
$ ./manage.py make_web_conf --apache --processes 10 > /etc/apache2/sites-  
↪available/horizon.conf
```

2. Enable the above configuration and restart the web server

```
$ sudo a2ensite horizon  
$ sudo service apache2 restart
```

Next Steps

- *Settings Reference* lists the available settings for horizon.
- *Customizing Horizon* describes how to customize horizon.

2.1.4 Horizon plugins

There are a number of horizon plugins for various useful features. You can get dashboard supports for them by installing corresponding horizon plugins.

Plugin Registry

This is a list of horizon plugins which are part of the official OpenStack releases.

Note: Currently, Horizon plugins are responsible for their own compatibility. Check the individual repos for information on support.

Plugin	Repository	Bug Tracker
adjutant-ui	openstack/adjutant-ui	storyboard openstack/adjutant-ui
blazar-dashboard	openstack/blazar-dashboard	launchpad blazar
cloudkitty-dashboard	openstack/cloudkitty-dashboard	storyboard openstack/cloudkitty-dashboard
designate-dashboard	openstack/designate-dashboard	launchpad designate-dashboard
freezer-web-ui	openstack/freezer-web-ui	storyboard openstack/freezer-web-ui
heat-dashboard	openstack/heat-dashboard	storyboard openstack/heat-dashboard
ironic-ui	openstack/ironic-ui	storyboard openstack/ironic-ui
magnum-ui	openstack/magnum-ui	launchpad magnum-ui
manila-ui	openstack/manila-ui	launchpad manila-ui
masakari-dashboard	openstack/masakari-dashboard	launchpad masakari
mistral-dashboard	openstack/mistral-dashboard	launchpad mistral
monasca-ui	openstack/monasca-ui	launchpad monasca
murano-dashboard	openstack/murano-dashboard	launchpad murano
networking-bgpvpn	openstack/networking-bgpvpn	launchpad bgpvpn
neutron-vpnaas-dashboard	openstack/neutron-vpnaas-dashboard	launchpad neutron-vpnaas-dashboard
octavia-dashboard	openstack/octavia-dashboard	storyboard openstack/octavia-dashboard
sahara-dashboard	openstack/sahara-dashboard	storyboard openstack/sahara-dashboard
senlin-dashboard	openstack/senlin-dashboard	launchpad senlin-dashboard
solum-dashboard	openstack/solum-dashboard	launchpad solum
tacker-horizon	openstack/tacker-horizon	launchpad tacker
trove-dashboard	openstack/trove-dashboard	launchpad trove-dashboard
vitrage-dashboard	openstack/vitrage-dashboard	storyboard openstack/vitrage-dashboard
watcher-dashboard	openstack/watcher-dashboard	launchpad watcher-dashboard
zaqar-ui	openstack/zaqar-ui	launchpad zaqar-ui
zun-ui	openstack/zun-ui	launchpad zun-ui

2.2 Configuration Guide

2.2.1 Settings Reference

Introduction

Horizons settings broadly fall into four categories:

- *General Settings*: this includes visual settings like the modal backdrop style, bug url and theme configuration, as well as settings that affect every service, such as page sizes on API requests.
- *Service-specific Settings*: Many services that Horizon consumes, such as Nova and Neutron, don't advertise their capabilities via APIs, so Horizon carries configuration for operators to enable or disable many items. This section covers all settings that are specific to a single service.
- *Django Settings*, which are common to all Django applications. The only ones documented here are those that Horizon alters by default; however, you should read the [Django settings documentation](#) to see the other options available to you.
- *Other Settings*: settings which do not fall into any of the above categories.

To modify your settings, you have two options:

- **Preferred:** Add `.py` settings snippets to the `openstack_dashboard/local/local_settings.d/` directory. Several example files (appended with `.example`) can be found there. These must start with an underscore, and are evaluated alphabetically, after `local_settings.py`.
- Modify your `openstack_dashboard/local/local_settings.py`. There is an file found at `openstack_dashboard/local/local_settings.py.example`.

General Settings

ANGULAR_FEATURES

New in version 10.0.0(Newton).

Default:

```
{
    'images_panel': True,
    'key_pairs_panel': True,
    'flavors_panel': False,
    'domains_panel': False,
    'users_panel': False,
    'groups_panel': False,
    'roles_panel': True
}
```

A dictionary of currently available AngularJS features. This allows simple toggling of legacy or rewritten features, such as new panels, workflows etc.

Note: If you toggle `domains_panel` to `True`, you also need to enable the setting of `OPENSTACK_KEYSTONE_DEFAULT_DOMAIN` and add `OPENSTACK_KEYSTONE_DEFAULT_DOMAIN` to `REST_API_REQUIRED_SETTINGS`.

API_RESULT_LIMIT

New in version 2012.1(Essex).

Default: 1000

The maximum number of objects (e.g. Swift objects or Glance images) to display on a single page before providing a paging element (a more link) to paginate results.

API_RESULT_PAGE_SIZE

New in version 2012.2(Folsom).

Default: 20

Similar to `API_RESULT_LIMIT`. This setting controls the number of items to be shown per page if API pagination support for this exists.

AVAILABLE_THEMES

New in version 9.0.0(Mitaka).

Default:

```
AVAILABLE_THEMES = [
    ('default', 'Default', 'themes/default'),
    ('material', 'Material', 'themes/material'),
]
```

This setting tells Horizon which themes to use.

A list of tuples which define multiple themes. The tuple format is ('`theme_name`', '`theme_label`', '`theme_path`').

The `theme_name` is the name used to define the directory which the theme is collected into, under /`THEME_COLLECTION_DIR`. It also specifies the key by which the selected theme is stored in the browsers cookie.

The `theme_label` is the user-facing label that is shown in the theme picker. The theme picker is only visible if more than one theme is configured, and shows under the topnavs user menu.

By default, the `theme_path` is the directory that will serve as the static root of the theme and the entire contents of the directory is served up at /`THEME_COLLECTION_DIR`/`theme_name`. If you wish to include content other than static files in a theme directory, but do not wish that content to be served up, then you can create a sub directory named `static`. If the theme folder contains a sub-directory with the name `static`, then `static/custom/static` will be used as the root for the content served at `/static/custom`.

The static root of the theme folder must always contain a `_variables.scss` file and a `_styles.scss` file. These must contain or import all the bootstrap and horizon specific variables and styles which are used to style the GUI. For example themes, see: `/horizon/openstack_dashboard/themes/`

Horizon ships with two themes configured. `default` is the default theme, and `material` is based on Google's Material Design.

DEFAULT_POLICY_FILES

New in version 19.1.0(Wallaby).

Default:

```
{
  'identity': 'default_policies/keystone.yaml',
  'compute': 'default_policies/nova.yaml',
  'volume': 'default_policies/cinder.yaml',
  'image': 'default_policies/glance.yaml',
  'network': 'default_policies/neutron.yaml',
}
```

This is a mapping from service types to YAML files including default policy definitions. Values of this mapping should be relative paths to `POLICY_FILES_PATH` or absolute paths. Policy files specified in this setting are generated from default policies of back-end services, so you rarely need to configure it. If you would like to override the default policies, consider customizing files under `POLICY_FILES`.

DEFAULT_THEME

New in version 9.0.0(Mitaka).

Default: `"default"`

This setting tells Horizon which theme to use if the user has not yet selected a theme through the theme picker and therefore set the cookie value. This value represents the `theme_name` key that is used from `AVAILABLE_THEMES`. To use this setting, the theme must also be configured inside of `AVAILABLE_THEMES`. Your default theme must be configured as part of `SELECTABLE_THEMES`. If it is not, then your `DEFAULT_THEME` will default to the first theme in `SELECTABLE_THEMES`.

DISALLOW_IFRAME_EMBED

New in version 8.0.0(Liberty).

Default: `True`

This setting can be used to defend against Clickjacking and prevent Horizon from being embedded within an iframe. Legacy browsers are still vulnerable to a Cross-Frame Scripting (XFS) vulnerability, so this option allows extra security hardening where iframes are not used in deployment. When set to true, a `"frame-buster"` script is inserted into the template header that prevents the web page from being framed and therefore defends against clickjacking.

For more information see: <http://tinyurl.com/anticlickjack>

Note: If your deployment requires the use of iframes, you can set this setting to `False` to exclude the frame-busting code and allow iframe embedding.

DROPDOWN_MAX_ITEMS

New in version 2015.1(Kilo).

Default: 30

This setting sets the maximum number of items displayed in a dropdown. Dropdowns that limit based on this value need to support a way to observe the entire list.

FILTER_DATA_FIRST

New in version 10.0.0(Newton).

Default:

```
{
  'admin.instances': False,
  'admin.images': False,
  'admin.networks': False,
  'admin.routers': False,
  'admin.volumes': False
}
```

If the dict key-value is `True`, when the view loads, an empty table will be rendered and the user will be asked to provide a search criteria first (in case no search criteria was provided) before loading any data.

Examples:

Override the dict:

```
{
  'admin.instances': True,
  'admin.images': True,
  'admin.networks': False,
  'admin.routers': False,
  'admin.volumes': False
}
```

Or, if you want to turn this on for an specific panel/view do:

```
FILTER_DATA_FIRST['admin.instances'] = True
```

HORIZON_CONFIG

A dictionary of some Horizon configuration values. These are primarily separated for historic design reasons.

Default:

```
HORIZON_CONFIG = {
    'user_home': 'openstack_dashboard.views.get_user_home',
    'ajax_queue_limit': 10,
    'auto_fade_alerts': {
        'delay': 3000,
        'fade_duration': 1500,
        'types': [
            'alert-success',
            'alert-info'
        ]
    },
    'bug_url': None,
    'help_url': "https://docs.openstack.org/",
    'exceptions': {
        'recoverable': exceptions.RECOVERABLE,
        'not_found': exceptions.NOT_FOUND,
        'unauthorized': exceptions.UNAUTHORIZED
    },
    'modal_backdrop': 'static',
    'angular_modules': [],
    'js_files': [],
    'js_spec_files': [],
    'external_templates': [],
}
```

ajax_poll_interval

New in version 2012.1(Essex).

Default: 2500

How frequently resources in transition states should be polled for updates, expressed in milliseconds.

ajax_queue_limit

New in version 2012.1(Essex).

Default: 10

The maximum number of simultaneous AJAX connections the dashboard may try to make. This is particularly relevant when monitoring a large number of instances, volumes, etc. which are all actively trying to update/change state.

angular_modules

New in version 2014.2(Juno).

Default: []

A list of AngularJS modules to be loaded when Angular bootstraps. These modules are added as dependencies on the root Horizon application horizon.

auto_fade_alerts

New in version 2013.2(Havana).

Default:

```
{
  'delay': 3000,
  'fade_duration': 1500,
  'types': []
}
```

If provided, will auto-fade the alert types specified. Valid alert types include: [alert-default, alert-success, alert-info, alert-warning, alert-danger] Can also define the delay before the alert fades and the fade out duration.

bug_url

New in version 9.0.0(Mitaka).

Default: None

If provided, a Report Bug link will be displayed in the site header which links to the value of this setting (ideally a URL containing information on how to report issues).

disable_password_reveal

New in version 2015.1(Kilo).

Default: False

Setting this to True will disable the reveal button for password fields, including on the login form.

exceptions

New in version 2012.1(Essex).

Default:

```
{
  'unauthorized': [],
  'not_found': [],
}
```

(continues on next page)

(continued from previous page)

```
'recoverable': []  
}
```

A dictionary containing classes of exceptions which Horizons centralized exception handling should be aware of. Based on these exception categories, Horizon will handle the exception and display a message to the user.

help_url

New in version 2012.2(Folsom).

Default: None

If provided, a Help link will be displayed in the site header which links to the value of this setting (ideally a URL containing help information).

js_files

New in version 2014.2(Juno).

Default: []

A list of javascript source files to be included in the compressed set of files that are loaded on every page. This is needed for AngularJS modules that are referenced in `angular_modules` and therefore need to be include in every page.

js_spec_files

New in version 2015.1(Kilo).

Default: []

A list of javascript spec files to include for integration with the Jasmine spec runner. Jasmine is a behavior-driven development framework for testing JavaScript code.

modal_backdrop

New in version 2014.2(Kilo).

Default: "static"

Controls how bootstrap backdrop element outside of modals looks and feels. Valid values are "true" (show backdrop element outside the modal, close the modal after clicking on backdrop), "false" (do not show backdrop element, do not close the modal after clicking outside of it) and "static" (show backdrop element outside the modal, do not close the modal after clicking on backdrop).

password_autocomplete

New in version 2013.1(Grizzly).

Default: "off"

Controls whether browser autocompletion should be enabled on the login form. Valid values are "on" and "off".

password_validator

New in version 2012.1(Essex).

Default:

```
{
  'regex': '.*',
  'help_text': _("Password is not accepted")
}
```

A dictionary containing a regular expression which will be used for password validation and help text which will be displayed if the password does not pass validation. The help text should describe the password requirements if there are any.

This setting allows you to set rules for passwords if your organization requires them.

user_home

New in version 2012.1(Essex).

Default: `settings.LOGIN_REDIRECT_URL`

This can be either a literal URL path (such as the default), or Python's dotted string notation representing a function which will evaluate what URL a user should be redirected to based on the attributes of that user.

MESSAGES_PATH

New in version 9.0.0(Mitaka).

Default: None

The absolute path to the directory where message files are collected.

When the user logs in to horizon, the message files collected are processed and displayed to the user. Each message file should contain a JSON formatted data and must have a .json file extension. For example:

```
{
  "level": "info",
  "message": "message of the day here"
}
```

Possible values for level are: success, info, warning and error.

NG_TEMPLATE_CACHE_AGE

New in version 10.0.0(Newton).

Angular Templates are cached using this duration (in seconds) if *DEBUG* is set to False. Default value is 2592000 (or 30 days).

OPENSTACK_API_VERSIONS

New in version 2013.2(Havana).

Default:

```
{
  "identity": 3,
  "volume": 3,
  "compute": 2
}
```

Overrides for OpenStack API versions. Use this setting to force the OpenStack dashboard to use a specific API version for a given service API.

Note: The version should be formatted as it appears in the URL for the service API. For example, the identity service APIs have inconsistent use of the decimal point, so valid options would be 3. For example:

```
OPENSTACK_API_VERSIONS = {
  "identity": 3,
  "volume": 3,
  "compute": 2
}
```

OPENSTACK_CLOUDS_YAML_CUSTOM_TEMPLATE

New in version 15.0.0(Stein).

Default: None

Example:: `my-clouds.yaml.template`

A template name for a custom users `clouds.yaml` file. None means the default template for `clouds.yaml` is used.

If the default template is not suitable for your deployment, you can provide your own `clouds.yaml` by specifying this setting.

The default template is defined as `clouds.yaml.template` and available context parameters are found in `_get_openrc_credentials()` and `download_clouds_yaml_file()` functions in `openstack_dashboard/dashboards/project/api_access/views.py`.

Note: Your template needs to be placed in the search paths of Django templates. You may need to configure `ADD_TEMPLATE_DIRS` setting to contain a path where your template exists.

OPENSTACK_CLOUDS_YAML_NAME

New in version 12.0.0(Pike).

Default: "openstack"

The name of the entry to put into the users clouds.yaml file.

OPENSTACK_CLOUDS_YAML_PROFILE

New in version 12.0.0(Pike).

Default: None

If set, the name of the [vendor profile](#) from `os-client-config`.

OPENSTACK_ENDPOINT_TYPE

New in version 2012.1(Essex).

Default: "publicURL"

A string which specifies the endpoint type to use for the endpoints in the Keystone service catalog. The default value for all services except for identity is "publicURL". The default value for the identity service is "internalURL".

OPENSTACK_HOST

New in version 2012.1(Essex).

Default: "127.0.0.1"

The hostname of the Keystone server used for authentication if you only have one region. This is often the **only** setting that needs to be set for a basic deployment.

If you have multiple regions you should use the [AVAILABLE_REGIONS](#) setting instead.

OPENRC_CUSTOM_TEMPLATE

New in version 15.0.0(Stein).

Default: None

Example:: `my-openrc.sh.template`

A template name for a custom users `openrc` file. None means the default template for `openrc` is used.

If the default template is not suitable for your deployment, for example, if your deployment uses saml2, openid and so on for authentication, the default `openrc` would not be sufficient. You can provide your own `clouds.yaml` by specifying this setting.

The default template is defined as `openrc.sh.template` and available context parameters are found in `_get_openrc_credentials()` and `download_rc_file()` functions in `openstack_dashboard/dashboards/project/api_access/views.py`.

Note: Your template needs to be placed in the search paths of Django templates. Check `TEMPLATES[0]['DIRS']`. You may need to specify somewhere your template exist to `DIRS` in `TEMPLATES` setting.

OPENSTACK_PROFILER

New in version 11.0.0(Ocata).

Default: `{"enabled": False}`

Various settings related to integration with `osprofiler` library. Since it is a developer feature, it starts as disabled. To enable it, more than a single "enabled" key should be specified. Additional keys that should be specified in that dictionary are:

- "keys" is a list of strings, which are secret keys used to encode/decode the profiler data contained in request headers. Encryption is used for security purposes, other OpenStack components that are expected to profile themselves with `osprofiler` using the data from the request that Horizon initiated must share a common set of keys with the ones in Horizon config. List of keys is used so that security keys could be changed in non-obtrusive manner for every component in the cloud. Example: `"keys": ["SECRET_KEY", "MORE_SECRET_KEY"]`. For more details see [osprofiler documentation](#).
- "notifier_connection_string" is a url to which trace messages are sent by Horizon. For other components it is usually the only URL specified in config, because other components act mostly as traces producers. Example: `"notifier_connection_string": "mongodb://%s" % OPENSTACK_HOST`.
- "receiver_connection_string" is a url from which traces are retrieved by Horizon, needed because Horizon is not only the traces producer, but also a consumer. Having 2 settings which usually contain the same value is legacy feature from older versions of `osprofiler` when OpenStack components could use `oslo.messaging` for notifications and the trace client used `ceilometer` as a receiver backend. By default Horizon uses the same URL pointing to a MongoDB cluster for both purposes. Example: `"receiver_connection_string": "mongodb://%s" % OPENSTACK_HOST`.

OPENSTACK_SSL_CACERT

New in version 2013.2(Havana).

Default: None

When unset or set to None the default CA certificate on the system is used for SSL verification.

When set with the path to a custom CA certificate file, this overrides use of the default system CA certificate. This custom certificate is used to verify all connections to openstack services when making API calls.

OPENSTACK_SSL_NO_VERIFY

New in version 2012.2(Folsom).

Default: False

Disable SSL certificate checks in the OpenStack clients (useful for self-signed certificates).

OPERATION_LOG_ENABLED

New in version 10.0.0(Newton).

Default: False

This setting can be used to enable logging of all operations carried out by users of Horizon. The format of the logs is configured via *OPERATION_LOG_OPTIONS*

Note: If you use this feature, you need to configure the logger setting like an outputting path for operation log in `local_settings.py`.

OPERATION_LOG_OPTIONS

New in version 10.0.0(Newton).

Changed in version 12.0.0(Pike): Added `ignored_urls` parameter and added `%(client_ip)s` to format

Default:

```

{
    'mask_fields': ['password'],
    'target_methods': ['POST'],
    'ignored_urls': ['/js/', '/static/', '^/api/'],
    'format': ("[(domain_name)s] [(domain_id)s] [(project_name)s]"
              " [(project_id)s] [(user_name)s] [(user_id)s] [(request_scheme)s]"
              " [(referer_url)s] [(request_url)s] [(message)s] [(method)s]"
              " [(http_status)s] [(param)s]"),
}

```

This setting controls the behavior of the operation log.

- `mask_fields` is a list of keys of post data which should be masked from the point of view of security. Fields like `password` should be included. The fields specified in `mask_fields` are logged as `*****`.
- `target_methods` is a request method which is logged to an operation log. The valid methods are POST, GET, PUT, DELETE.
- `ignored_urls` is a list of request URLs to be hidden from a log.
- `format` defines the operation log format. Currently you can use the following keywords. The default value contains all keywords.

- `%(client_ip)s`
- `%(domain_name)s`
- `%(domain_id)s`
- `%(project_name)s`
- `%(project_id)s`
- `%(user_name)s`
- `%(user_id)s`
- `%(request_scheme)s`
- `%(referer_url)s`
- `%(request_url)s`
- `%(message)s`
- `%(method)s`
- `%(http_status)s`
- `%(param)s`

OVERVIEW_DAYS_RANGE

New in version 10.0.0(Newton).

Default:: 1

When set to an integer N (as by default), the start date in the Overview panel meters will be today minus N days. This setting is used to limit the amount of data fetched by default when rendering the Overview panel. If set to `None` (which corresponds to the behavior in past Horizon versions), the start date will be from the beginning of the current month until the current date. The legacy behaviour is not recommended for large deployments as Horizon suffers significant lag in this case.

POLICY_CHECK_FUNCTION

New in version 2013.2(Havana).

Default:: `openstack_auth.policy.check`

This value should not be changed, although removing it or setting it to `None` would be a means to bypass all policy checks.

POLICY_DIRS

New in version 13.0.0(Queens).

Default:

```
{
  'compute': ['nova_policy.d'],
  'volume': ['cinder_policy.d'],
}
```

Specifies a list of policy directories per service types. The directories are relative to *POLICY_FILES_PATH*. Services whose additional policies are defined here must be defined in *POLICY_FILES* too. Otherwise, additional policies specified in *POLICY_DIRS* are not loaded.

Note: `cinder_policy.d` and `nova_policy.d` are registered by default to maintain policies which have ben dropped from nova and cinder but horizon still uses. We recommend not to drop them.

POLICY_FILES

New in version 2013.2(Havana).

Changed in version 19.1.0(Wallaby): The default files are changed to YAML format. JSON format still continues to be supported.

Default:

```
{
  'compute': 'nova_policy.yaml',
  'identity': 'keystone_policy.yaml',
  'image': 'glance_policy.yaml',
  'network': 'neutron_policy.yaml',
  'volume': 'cinder_policy.yaml',
}
```

This should essentially be the mapping of the contents of *POLICY_FILES_PATH* to service types. When policy files are added to *POLICY_FILES_PATH*, they should be included here too.

POLICY_FILES_PATH

New in version 2013.2(Havana).

Default: `os.path.join(ROOT_PATH, "conf")`

Specifies where service based policy files are located. These are used to define the policy rules actions are verified against.

REST_API_REQUIRED_SETTINGS

New in version 2014.2(Kilo).

Default:

```
[
    'CREATE_IMAGE_DEFAULTS',
    'DEFAULT_BOOT_SOURCE',
    'ENFORCE_PASSWORD_CHECK',
    'LAUNCH_INSTANCE_DEFAULTS',
    'OPENSTACK_HYPERVISOR_FEATURES',
    'OPENSTACK_IMAGE_FORMATS',
    'OPENSTACK_KEYSTONE_BACKEND',
    'OPENSTACK_KEYSTONE_DEFAULT_DOMAIN',
]
```

This setting allows you to expose configuration values over Horizons internal REST API, so that the AngularJS panels can access them. Please be cautious about which values are listed here (and thus exposed on the frontend). For security purpose, this exposure of settings should be recognized explicitly by operator. So `REST_API_REQUIRED_SETTINGS` is not set by default. Please refer `local_settings.py.example` and confirm your `local_settings.py`.

SELECTABLE_THEMES

New in version 12.0.0(Pike).

Default: `AVAILABLE_THEMES`

This setting tells Horizon which themes to expose to the user as selectable in the theme picker widget. This value defaults to all themes configured in `AVAILABLE_THEMES`, but a brander may wish to simply inherit from an existing theme and not allow that parent theme to be selected by the user. `SELECTABLE_THEMES` takes the exact same format as `AVAILABLE_THEMES`.

SESSION_REFRESH

New in version 15.0.0(Stein).

Default: True

Control whether the SESSION_TIMEOUT period is refreshed due to activity. If False, SESSION_TIMEOUT acts as a hard limit.

SESSION_TIMEOUT

New in version 2013.2(Havana).

Default: "3600"

This SESSION_TIMEOUT is a method to supercede the token timeout with a shorter horizon session timeout (in seconds). If SESSION_REFRESH is True (the default) SESSION_TIMEOUT acts like an idle timeout rather than being a hard limit, but will never exceed the token expiry. If your token expires in 60 minutes, a value of 1800 will log users out after 30 minutes of inactivity, or 60 minutes with activity. Setting SESSION_REFRESH to False will make SESSION_TIMEOUT act like a hard limit on session times.

MEMOIZED_MAX_SIZE_DEFAULT

New in version 15.0.0(Stein).

Default: "25"

MEMOIZED_MAX_SIZE_DEFAULT allows setting a global default to help control memory usage when caching. It should at least be 2 x the number of threads with a little bit of extra buffer.

SHOW_OPENRC_FILE

New in version 15.0.0(Stein).

Default:: True

Controls whether the keystone openrc file is accesible from the user menu and the api access panel.

See also:

OPENRC_CUSTOM_TEMPLATE to provide a custom openrc.

SHOW_OPENSTACK_CLOUDS_YAML

New in version 15.0.0(Stein).

Default:: True

Controls whether clouds.yaml is accesible from the user menu and the api access panel.

See also:

OPENSTACK_CLOUDS_YAML_CUSTOM_TEMPLATE to provide a custom clouds.yaml.

THEME_COLLECTION_DIR

New in version 9.0.0(Mitaka).

Default: "themes"

This setting tells Horizon which static directory to collect the available themes into, and therefore which URL points to the theme collection root. For example, the default theme would be accessible via `/{{ STATIC_URL }}/themes/default`.

THEME_COOKIE_NAME

New in version 9.0.0(Mitaka).

Default: "theme"

This setting tells Horizon in which cookie key to store the currently set theme. The cookie expiration is currently set to a year.

USER_MENU_LINKS

New in version 13.0.0(Queens).

Default:

```
[
  {'name': _('OpenStack RC File'),
   'icon_classes': ['fa-download', ],
   'url': 'horizon:project:api_access:openrc',
   'external': False,
  }
]
```

This setting controls the additional links on the user drop down menu. A list of dictionaries defining all of the links should be provided. This defaults to the standard OpenStack RC files.

Each dictionary should contain these values:

name The name of the link

url Either the reversible Django url name or an absolute url

external True for absolute URLs, False for django urls.

icon_classes A list of classes for the icon next to the link. If None or an empty list is provided a download icon will show

WEBROOT

New in version 2015.1(Kilo).

Default: "/"

Specifies the location where the access to the dashboard is configured in the web server.

For example, if you're accessing the Dashboard via `https://<your server>/dashboard`, you would set this to `"/dashboard/"`.

Note: Additional settings may be required in the config files of your webserver of choice. For example to make `"/dashboard/"` the web root in Apache, the `"sites-available/horizon.conf"` requires a couple of additional aliases set:

```
Alias /dashboard/static %HORIZON_DIR%/static

Alias /dashboard/media %HORIZON_DIR%/openstack_dashboard/static
```

Apache also requires changing your `WSGIScriptAlias` to reflect the desired path. For example, you'd replace `/` with `/dashboard` for the alias.

Service-specific Settings

The following settings inform the OpenStack Dashboard of information about the other OpenStack projects which are part of this cloud and control the behavior of specific dashboards, panels, API calls, etc.

Cinder

OPENSTACK_CINDER_FEATURES

New in version 2014.2(Juno).

Default: `{ 'enable_backup': False }`

A dictionary of settings which can be used to enable optional services provided by cinder. Currently only the backup service is available.

Glance

CREATE_IMAGE_DEFAULTS

New in version 12.0.0(Pike).

Default:

```
{
    'image_visibility': "public",
}
```

A dictionary of default settings for create image modal.

The `image_visibility` setting specifies the default visibility option. Valid values are "public" and "private". By default, the visibility option is public on create image modal. If its set to "private", the default visibility option is private.

HORIZON_IMAGES_UPLOAD_MODE

New in version 10.0.0(Newton).

Default: "legacy"

Valid values are "direct", "legacy" (default) and "off". "off" disables the ability to upload images via Horizon. `legacy` enables local file upload by piping the image file through the Horizons web-server. `direct` sends the image file directly from the web browser to Glance. This bypasses Horizon web-server which both reduces network hops and prevents filling up Horizon web-servers filesystem. `direct` is the preferred mode, but due to the following requirements it is not the default. The `direct` setting requires a modern web browser, network access from the browser to the public Glance endpoint, and CORS support to be enabled on the Glance API service. Without CORS support, the browser will forbid the PUT request to a location different than the Horizon server. To enable CORS support for Glance API service, you will need to edit `[cors]` section of `glance-api.conf` file (see [here](#) how to do it). Set `allowed_origin` to the full hostname of Horizon web-server (e.g. `http://<HOST_IP>/dashboard`) and restart `glance-api` process.

IMAGE_CUSTOM_PROPERTY_TITLES

New in version 2014.1(Icehouse).

Default:

```
{
  "architecture": _("Architecture"),
  "kernel_id": _("Kernel ID"),
  "ramdisk_id": _("Ramdisk ID"),
  "image_state": _("Euca2ools state"),
  "project_id": _("Project ID"),
  "image_type": _("Image Type")
}
```

Used to customize the titles for image custom property attributes that appear on image detail pages.

IMAGE_RESERVED_CUSTOM_PROPERTIES

New in version 2014.2(Juno).

Default: []

A list of image custom property keys that should not be displayed in the Update Metadata tree.

This setting can be used in the case where a separate panel is used for managing a custom property or if a certain custom property should never be edited.

IMAGES_ALLOW_LOCATION

New in version 10.0.0(Newton).

Default: False

If set to True, this setting allows users to specify an image location (URL) as the image source when creating or updating images. Depending on the Glance version, the ability to set an image location is controlled by policies and/or the Glance configuration. Therefore IMAGES_ALLOW_LOCATION should only be set to True if Glance is configured to allow specifying a location. This setting has no effect when the Keystone catalog doesn't contain a Glance v2 endpoint.

IMAGES_LIST_FILTER_TENANTS

New in version 2013.1(Grizzly).

Default: None

A list of dictionaries to add optional categories to the image fixed filters in the Images panel, based on project ownership.

Each dictionary should contain a *tenant* attribute with the project id, and optionally a *text* attribute specifying the category name, and an *icon* attribute that displays an icon in the filter button. The icon names are based on the default icon theme provided by Bootstrap.

Example:

```
[{'text': 'Official',
  'tenant': '27d0058849da47c896d205e2fc25a5e8',
  'icon': 'fa-check'}]
```

OPENSTACK_IMAGE_BACKEND

New in version 2013.2(Havana).

Default:

```
{
  'image_formats': [
    ('', _('Select format')),
    ('aki', _('AKI - Amazon Kernel Image')),
    ('ami', _('AMI - Amazon Machine Image')),
    ('ari', _('ARI - Amazon Ramdisk Image')),
    ('docker', _('Docker')),
    ('iso', _('ISO - Optical Disk Image')),
    ('qcow2', _('QCOW2 - QEMU Emulator')),
    ('raw', _('Raw')),
    ('vdi', _('VDI')),
    ('vhd', _('VHD')),
    ('vmdk', _('VMDK'))
  ]
}
```

Used to customize features related to the image service, such as the list of supported image formats.

Keystone

ALLOW_USERS_CHANGE_EXPIRED_PASSWORD

New in version 16.0.0(Train).

Default: True

When enabled, this setting lets users change their password after it has expired or when it is required to be changed on first use. Disabling it will force such users to either use the command line interface to change their password, or contact the system administrator.

AUTHENTICATION_PLUGINS

New in version 2015.1(Kilo).

Default:

```
[
    'openstack_auth.plugin.password.PasswordPlugin',
    'openstack_auth.plugin.token.TokenPlugin'
]
```

A list of authentication plugins to be used. In most cases, there is no need to configure this.

AUTHENTICATION_URLS

New in version 2015.1(Kilo).

Default: ['openstack_auth.urls']

A list of modules from which to collate authentication URLs from. The default option adds URLs from the django-openstack-auth module however others will be required for additional authentication mechanisms.

AVAILABLE_REGIONS

New in version 2012.1(Essex).

Default: None

A list of tuples which define multiple regions. The tuple format is ('http://{{ keystone_host }}/identity/v3', '{{ region_name }}'). If any regions are specified the login form will have a dropdown selector for authenticating to the appropriate region, and there will be a region switcher dropdown in the site header when logged in.

You should also define *OPENSTACK_KEYSTONE_URL* to indicate which of the regions is the default one.

DEFAULT_SERVICE_REGIONS

New in version 12.0.0(Pike).

Default: {}

The default service region is set on a per-endpoint basis, meaning that once the user logs into some Keystone endpoint, if a default service region is defined for it in this setting and exists within Keystone catalog, it will be set as the initial service region in this endpoint. By default it is an empty dictionary because upstream can neither predict service region names in a specific deployment, nor tell whether this behavior is desired. The key of the dictionary is a full url of a Keystone endpoint with version suffix, the value is a region name.

Example:

```
DEFAULT_SERVICE_REGIONS = {
    OPENSTACK_KEYSTONE_URL: 'RegionOne'
}
```

As of Rocky you can optionally you can set '*' as the key, and if no matching endpoint is found this will be treated as a global default.

Example:

```
DEFAULT_SERVICE_REGIONS = {
    '*': 'RegionOne',
    OPENSTACK_KEYSTONE_URL: 'RegionTwo'
}
```

ENABLE_CLIENT_TOKEN

New in version 10.0.0(Newton).

Default: True

This setting will Enable/Disable access to the Keystone Token to the browser.

ENFORCE_PASSWORD_CHECK

New in version 2015.1(Kilo).

Default: False

This setting will display an Admin Password field on the Change Password form to verify that it is indeed the admin logged-in who wants to change the password.

KEYSTONE_PROVIDER_IDP_ID

New in version 11.0.0(Ocata).

Default: "localkeystone"

This ID is only used for comparison with the service provider IDs. This ID should not match any service provider IDs.

KEYSTONE_PROVIDER_IDP_NAME

New in version 11.0.0(Ocata).

Default: "Local Keystone"

The Keystone Provider drop down uses Keystone to Keystone federation to switch between Keystone service providers. This sets the display name for the Identity Provider (dropdown display name).

OPENSTACK_KEYSTONE_ADMIN_ROLES

New in version 2015.1(Kilo).

Default: ["admin"]

The list of roles that have administrator privileges in this OpenStack installation. This check is very basic and essentially only works with keystone v3 with the default policy file. The setting assumes there is a common `admin` like role(s) across services. Example uses of this setting are:

- to rename the `admin` role to `cloud-admin`
- allowing multiple roles to have administrative privileges, like ["admin", "cloud-admin", "net-op"]

OPENSTACK_KEYSTONE_BACKEND

New in version 2012.1(Essex).

Default:

```
{
  'name': 'native',
  'can_edit_user': True,
  'can_edit_group': True,
  'can_edit_project': True,
  'can_edit_domain': True,
  'can_edit_role': True,
}
```

A dictionary containing settings which can be used to identify the capabilities of the auth backend for Keystone.

If Keystone has been configured to use LDAP as the auth backend then set `can_edit_user` and `can_edit_project` to `False` and `name` to "ldap".

OPENSTACK_KEYSTONE_DEFAULT_DOMAIN

New in version 2013.2(Havana).

Default: "Default"

Overrides the default domain used when running on single-domain model with Keystone V3. All entities will be created in the default domain.

OPENSTACK_KEYSTONE_DEFAULT_ROLE

New in version 2011.3(Diablo).

Default: "_member_"

The name of the role which will be assigned to a user when added to a project. This value must correspond to an existing role name in Keystone. In general, the value should match the `member_role_name` defined in `keystone.conf`.

OPENSTACK_KEYSTONE_DOMAIN_CHOICES

New in version 12.0.0(Pike).

Default:

```
(  
    ('Default', 'Default'),  
)
```

If `OPENSTACK_KEYSTONE_DOMAIN_DROPDOWN` is enabled, this option can be used to set the available domains to choose from. This is a list of pairs whose first value is the domain name and the second is the display name.

OPENSTACK_KEYSTONE_DOMAIN_DROPDOWN

New in version 12.0.0(Pike).

Default: False

Set this to True if you want available domains displayed as a dropdown menu on the login screen. It is strongly advised NOT to enable this for public clouds, as advertising enabled domains to unauthenticated customers irresponsibly exposes private information. This should only be used for private clouds where the dashboard sits behind a corporate firewall.

OPENSTACK_KEYSTONE_FEDERATION_MANAGEMENT

New in version 9.0.0(Mitaka).

Default: False

Set this to True to enable panels that provide the ability for users to manage Identity Providers (IdPs) and establish a set of rules to map federation protocol attributes to Identity API attributes. This extension requires v3.0+ of the Identity API.

OPENSTACK_KEYSTONE_MULTIDOMAIN_SUPPORT

New in version 2013.2(Havana).

Default: False

Set this to True if running on multi-domain model. When this is enabled, it will require user to enter the Domain name in addition to username for login.

OPENSTACK_KEYSTONE_URL

New in version 2011.3(Diablo).

Changed in version 17.1.0(Ussuri): The default value was changed to "http://%s/identity/v3" % OPENSTACK_HOST

See also:

Horizons *OPENSTACK_HOST* documentation

Default: "http://%s/identity/v3" % OPENSTACK_HOST

The full URL for the Keystone endpoint used for authentication. Unless you are using HTTPS, running your Keystone server on a nonstandard port, or using a nonstandard URL scheme you shouldnt need to touch this setting.

PASSWORD_EXPIRES_WARNING_THRESHOLD_DAYS

New in version 12.0.0(Pike).

Default: -1

Password will have an expiration date when using keystone v3 and enabling the feature. This setting allows you to set the number of days that the user will be alerted prior to the password expiration. Once the password expires keystone will deny the access and users must contact an admin to change their password. Setting this value to N days means the user will be alerted when the password expires in less than N+1 days. -1 disables the feature.

PROJECT_TABLE_EXTRA_INFO

New in version 10.0.0(Newton).

See also:

USER_TABLE_EXTRA_INFO for the equivalent setting on the Users table

Default: {}

Adds additional information for projects as extra attributes. Projects can have extra attributes as defined by Keystone v3. This setting allows those attributes to be shown in Horizon.

For example:

```
PROJECT_TABLE_EXTRA_INFO = {  
    'phone_num': _('Phone Number'),  
}
```

SECURE_PROXY_ADDR_HEADER

Default: False

If horizon is behind a proxy server and the proxy is configured, the IP address from request is passed using header variables inside the request. The header name depends on a proxy or a load-balancer. This setting specifies the name of the header with remote IP address. The main use is for authentication log (success or fail) displaying the IP address of the user. The common value for this setting is HTTP_X_REAL_IP or HTTP_X_FORWARDED_FOR. If not present, then REMOTE_ADDR header is used. (REMOTE_ADDR is the field of Django HttpRequest object which contains IP address of the client.)

TOKEN_DELETION_DISABLED

New in version 10.0.0(Newton).

Default: False

This setting allows deployers to control whether a token is deleted on log out. This can be helpful when there are often long running processes being run in the Horizon environment.

TOKEN_TIMEOUT_MARGIN

Default: 0

A time margin in seconds to subtract from the real tokens validity. An example use case is that the token can be valid once the middleware passed, and invalid (timed-out) during a view rendering and this generates authorization errors during the view rendering. By setting this value to a few seconds, you can avoid token expiration during a view rendering.

USER_TABLE_EXTRA_INFO

New in version 10.0.0(Newton).

See also:

PROJECT_TABLE_EXTRA_INFO for the equivalent setting on the Projects table

Default: {}

Adds additional information for users as extra attributes. Users can have extra attributes as defined by Keystone v3. This setting allows those attributes to be shown in Horizon.

For example:

```
USER_TABLE_EXTRA_INFO = {  
    'phone_num': _('Phone Number'),  
}
```

WEBSO_CHOICES

New in version 2015.1(Kilo).

Default:

```
(  
    ("credentials", _("Keystone Credentials")),  
    ("oidc", _("OpenID Connect")),  
    ("saml2", _("Security Assertion Markup Language"))  
)
```

This is the list of authentication mechanisms available to the user. It includes Keystone federation protocols such as OpenID Connect and SAML, and also keys that map to specific identity provider and federation protocol combinations (as defined in *WEBSO_IDP_MAPPING*). The list of choices is completely configurable, so as long as the id remains intact. Do not remove the credentials mechanism unless you are sure. Once removed, even admins will have no way to log into the system via the dashboard.

WEBSO_ENABLED

New in version 2015.1(Kilo).

Default: False

Enables keystone web single-sign-on if set to True. For this feature to work, make sure that you are using Keystone V3 and Django OpenStack Auth V1.2.0 or later.

WEBSSO_IDP_MAPPING

New in version 8.0.0(Liberty).

Default: {}

A dictionary of specific identity provider and federation protocol combinations. From the selected authentication mechanism, the value will be looked up as keys in the dictionary. If a match is found, it will redirect the user to a identity provider and federation protocol specific WebSSO endpoint in keystone, otherwise it will use the value as the protocol_id when redirecting to the WebSSO by protocol endpoint.

Example:

```
WEBSSO_CHOICES = (
    ("credentials", _("Keystone Credentials")),
    ("oidc", _("OpenID Connect")),
    ("saml2", _("Security Assertion Markup Language")),
    ("acme_oidc", "ACME - OpenID Connect"),
    ("acme_saml2", "ACME - SAML2")
)

WEBSSO_IDP_MAPPING = {
    "acme_oidc": ("acme", "oidc"),
    "acme_saml2": ("acme", "saml2")
}
```

Note: The value is expected to be a tuple formatted as: (<idp_id>, <protocol_id>)

WEBSSO_INITIAL_CHOICE

New in version 2015.1(Kilo).

Default: "credentials"

Specifies the default authentication mechanism. When user lands on the login page, this is the first choice they will see.

WEBSSO_DEFAULT_REDIRECT

New in version 15.0.0(Stein).

Default: False

Allows to redirect on login to the IdP provider defined on PROTOCOL and REGION In cases you have a single IdP providing websso, in order to improve user experience, you can redirect on the login page to the IdP directly by specifying WEBSSO_DEFAULT_REDIRECT_PROTOCOL and WEBSSO_DEFAULT_REDIRECT_REGION variables.

WEBSSO_DEFAULT_REDIRECT_PROTOCOL

New in version 15.0.0(Stein).

Default: None

Allows to specify the protocol for the IdP to contact if the WEBSSO_DEFAULT_REDIRECT is set to True

WEBSSO_DEFAULT_REDIRECT_REGION

New in version 15.0.0(Stein).

Default: OPENSTACK_KEYSTONE_URL

Allows to specify the region of the IdP to contact if the WEBSSO_DEFAULT_REDIRECT is set to True

WEBSSO_DEFAULT_REDIRECT_LOGOUT

New in version 15.0.0(Stein).

Default: None

Allows to specify a callback to the IdP to cleanup the SSO resources. Once the user logs out it will redirect to the IdP log out method.

WEBSSO_KEYSTONE_URL

New in version 15.0.0(Stein).

Default: None

The full auth URL for the Keystone endpoint used for web single-sign-on authentication. Use this when OPENSTACK_KEYSTONE_URL is set to an internal Keystone endpoint and is not reachable from the external network where the identity provider lives. This URL will take precedence over OPENSTACK_KEYSTONE_URL if the login choice is an external identity provider (IdP).

Neutron

ALLOWED_PRIVATE_SUBNET_CIDR

New in version 10.0.0(Newton).

Default:

```
{
  'ipv4': [],
  'ipv6': []
}
```


A dictionary used to restrict user private subnet CIDR range. An empty list means that user input will not be restricted for a corresponding IP version. By default, there is no restriction for both IPv4 and IPv6.

Example:

```
{
  'ipv4': [
    '192.168.0.0/16',
    '10.0.0.0/8'
  ],
  'ipv6': [
    'fc00::/7',
  ]
}
```

OPENSTACK_NEUTRON_NETWORK

New in version 2013.1(Grizzly).

Default:

```
{
  'default_dns_nameservers': [],
  'enable_auto_allocated_network': False,
  'enable_distributed_router': False,
  'enable_fip_topology_check': True,
  'enable_ha_router': False,
  'enable_ipv6': True,
  'enable_quotas': True,
  'enable_rbac_policy': True,
  'enable_router': True,
  'extra_provider_types': {},
  'physical_networks': [],
  'segmentation_id_range': {},
  'supported_provider_types': ["*"],
  'supported_vnic_types': ["*"],
}
```

A dictionary of settings which can be used to enable optional services provided by Neutron and configure Neutron specific features. The following options are available.

default_dns_nameservers

New in version 10.0.0(Newton).

Default: None (Empty)

Default DNS servers you would like to use when a subnet is created. This is only a default. Users can still choose a different list of dns servers.

Example: ["8.8.8.8", "8.8.4.4", "208.67.222.222"]

enable_auto_allocated_network

New in version 14.0.0(Rocky).

Default: False

Enable or disable Nova and Neutron get-me-a-network feature. This sets up a neutron network topology for a project if there is no network in the project. It simplifies the workflow when launching a server. Horizon checks if both nova and neutron support the feature and enable it only when supported. However, whether the feature works properly depends on deployments, so this setting is disabled by default. (The detail on the required preparation is described in [the Networking Guide](#).)

enable_distributed_router

New in version 2014.2(Juno).

Default: False

Enable or disable Neutron distributed virtual router (DVR) feature in the Router panel. For the DVR feature to be enabled, this option needs to be set to True and your Neutron deployment must support DVR. Even when your Neutron plugin (like ML2 plugin) supports DVR feature, DVR feature depends on l3-agent configuration, so deployers should set this option appropriately depending on your deployment.

enable_fip_topology_check

New in version 8.0.0(Liberty).

Default: True

The Default Neutron implementation needs a router with a gateway to associate a FIP. So by default a topology check will be performed by horizon to list only VM ports attached to a network which is itself attached to a router with an external gateway. This is to prevent from setting a FIP to a port which will fail with an error. Some Neutron vendors do not require it. Some can even attach a FIP to any port (e.g.: OpenContrail) owned by a tenant. Set to False if you want to be able to associate a FIP to an instance on a subnet with no router if your Neutron backend allows it.

enable_ha_router

New in version 2014.2(Juno).

Default: False

Enable or disable HA (High Availability) mode in Neutron virtual router in the Router panel. For the HA router mode to be enabled, this option needs to be set to True and your Neutron deployment must support HA router mode. Even when your Neutron plugin (like ML2 plugin) supports HA router mode, the feature depends on l3-agent configuration, so deployers should set this option appropriately depending on your deployment.

enable_ipv6

New in version 2014.2(Juno).

Default: False

Enable or disable IPv6 support in the Network panels. When disabled, Horizon will only expose IPv4 configuration for networks.

enable_quotas

Changed in version 17.0.0(Ussuri): The default value was changed to True

Default: True

Enable support for Neutron quotas feature. To make this feature work appropriately, you need to use Neutron plugins with quotas extension support and `quota_driver` should be `DbQuotaDriver` (default config).

enable_rbac_policy

New in version 15.0.0(Stein).

Default: True

Set this to True to enable RBAC Policies panel that provide the ability for users to use RBAC function. This option only affects when Neutron is enabled.

enable_router

New in version 2014.2(Juno).

Default: True

Enable (True) or disable (False) the panels and menus related to router and Floating IP features. This option only affects when Neutron is enabled. If your Neutron deployment has no support for Layer-3 features, or you do not wish to provide the Layer-3 features through the Dashboard, this should be set to False.

extra_provider_types

New in version 10.0.0(Newton).

Default: {}

For use with the provider network extension. This is a dictionary to define extra provider network definitions. Network types supported by Neutron depend on the configured plugin. Horizon has predefined provider network types but horizon cannot cover all of them. If you are using a provider network type not defined in advance, you can add a definition through this setting.

The **key** name of each item in this must be a network type used in the Neutron API. **value** should be a dictionary which contains the following items:

- `display_name`: string displayed in the network creation form.

- `require_physical_network`: a boolean parameter which indicates this network type requires a physical network.
- `require_segmentation_id`: a boolean parameter which indicates this network type requires a segmentation ID. If True, a valid segmentation ID range must be configured in `segmentation_id_range` settings above.

Example:

```
{
  'awesome': {
    'display_name': 'Awesome',
    'require_physical_network': False,
    'require_segmentation_id': True,
  },
}
```

physical_networks

New in version 12.0.0(Pike).

Default: []

Default to an empty list and the physical network field on the admin create network modal will be a regular input field where users can type in the name of the physical network to be used. If it is set to a list of available physical networks, the physical network field will be shown as a dropdown menu where users can select a physical network to be used.

Example: ['default', 'test']

segmentation_id_range

New in version 2014.2(Juno).

Default: {}

For use with the provider network extension. This is a dictionary where each key is a provider network type and each value is a list containing two numbers. The first number is the minimum segmentation ID that is valid. The second number is the maximum segmentation ID. Pertains only to the vlan, gre, and vxlan network types. By default this option is not provided and each minimum and maximum value will be the default for the provider network type.

Example:

```
{
  'vlan': [1024, 2048],
  'gre': [4094, 65536]
}
```

supported_provider_types

New in version 2014.2(Juno).

Default: ["*"]

For use with the provider network extension. Use this to explicitly set which provider network types are supported. Only the network types in this list will be available to choose from when creating a network. Network types defined in Horizon or defined in *extra_provider_types* settings can be specified in this list. As of the Newton release, the network types defined in Horizon include network types supported by Neutron ML2 plugin with Open vSwitch driver (local, flat, vlan, gre, vxlan and geneve) and supported by Midonet plugin (midonet and uplink). ["*"] means that all provider network types supported by Neutron ML2 plugin will be available to choose from.

Example: ['local', 'flat', 'gre']

supported_vnic_types

New in version 2015.1(Kilo).

Changed in version 12.0.0(Pike): Added virtio-forwarder VNIC type Clarified VNIC type availability for users and operators

Default ['*']

For use with the port binding extension. Use this to explicitly set which VNIC types are available for users to choose from, when creating or editing a port. The VNIC types actually supported are determined by resource availability and Neutron ML2 plugin support. Currently, error reporting for users selecting an incompatible or unavailable VNIC type is restricted to receiving a message from the scheduler that the instance cannot spawn because of insufficient resources. VNIC types include normal, direct, direct-physical, macvtap, baremetal and virtio-forwarder. By default all VNIC types will be available to choose from.

Example: ['normal', 'direct']

To disable VNIC type selection, set an empty list ([]) or None.

Nova

CREATE_INSTANCE_FLAVOR_SORT

New in version 2013.2(Havana).

Default:

```
{
  'key': 'ram'
}
```

When launching a new instance the default flavor is sorted by RAM usage in ascending order. You can customize the sort order by: id, name, ram, disk and vcpus. Additionally, you can insert any custom callback function. You can also provide a flag for reverse sort. See the description in local_settings.py.example for more information.

This example sorts flavors by vcpus in descending order:

```
CREATE_INSTANCE_FLAVOR_SORT = {  
    'key': 'vcpus',  
    'reverse': True,  
}
```

CONSOLE_TYPE

New in version 2013.2(Havana).

Changed in version 2014.2(Juno): Added the None option, which deactivates the in-browser console

Changed in version 2015.1(Kilo): Added the SERIAL option

Changed in version 2017.11(Queens): Added the MKS option

Default: "AUTO"

This setting specifies the type of in-browser console used to access the VMs. Valid values are "AUTO", "VNC", "SPICE", "RDP", "SERIAL", "MKS", and None.

DEFAULT_BOOT_SOURCE

New in version 18.1.0(Ussuri).

Default: image

A default instance boot source. Allowed values are:

- image - boot instance from image (default option)
- snapshot - boot instance from instance snapshot
- volume - boot instance from volume
- volume_snapshot - boot instance from volume snapshot

INSTANCE_LOG_LENGTH

New in version 2015.1(Kilo).

Default: 35

This setting enables you to change the default number of lines displayed for the log of an instance. Valid value must be a positive integer.

LAUNCH_INSTANCE_DEFAULTS

New in version 9.0.0(Mitaka).

Changed in version 10.0.0(Newton): Added the `disable_image`, `disable_instance_snapshot`, `disable_volume` and `disable_volume_snapshot` options.

Changed in version 12.0.0(Pike): Added the `create_volume` option.

Changed in version 15.0.0(Stein): Added the `hide_create_volume` option.

Changed in version 19.1.0(Wallaby): Added the `default_availability_zone` option.

Default:

```
{
  "config_drive": False,
  "create_volume": True,
  "hide_create_volume": False,
  "disable_image": False,
  "disable_instance_snapshot": False,
  "disable_volume": False,
  "disable_volume_snapshot": False,
  "enable_scheduler_hints": True,
  "default_availability_zone": "Any",
}
```

A dictionary of settings which can be used to provide the default values for properties found in the Launch Instance modal. An explanation of each setting is provided below.

`config_drive`

New in version 9.0.0(Mitaka).

Default: False

This setting specifies the default value for the Configuration Drive property.

`create_volume`

New in version 12.0.0(Pike).

Default: True

This setting allows you to specify the default value for the option of creating a new volume in the workflow for image and instance snapshot sources.

hide_create_volume

New in version 15.0.0(Stein).

Default: `False`

This setting allow your to hide the Create New Volume option and rely on the default value you select with `create_volume` to be the most suitable for your users.

disable_image

New in version 10.0.0(Newton).

Default: `False`

This setting disables Images as a valid boot source for launching instances. Image sources wont show up in the Launch Instance modal.

disable_instance_snapshot

New in version 10.0.0(Newton).

Default: `False`

This setting disables Snapshots as a valid boot source for launching instances. Snapshots sources wont show up in the Launch Instance modal.

disable_volume

New in version 10.0.0(Newton).

Default: `False`

This setting disables Volumes as a valid boot source for launching instances. Volumes sources wont show up in the Launch Instance modal.

disable_volume_snapshot

New in version 10.0.0(Newton).

Default: `False`

This setting disables Volume Snapshots as a valid boot source for launching instances. Volume Snapshots sources wont show up in the Launch Instance modal.

enable_scheduler_hints

New in version 9.0.0(Mitaka).

Default: True

This setting specifies whether or not Scheduler Hints can be provided when launching an instance.

default_availability_zone

New in version 19.1.0(Wallaby).

Default: Any

This setting allows an administrator to specify a default availability zone for a new server creation. The valid value is Any or availability zone list. If Any is specified, the default availability zone is decided by the nova scheduler. If one of availability zones is specified, the specified availability zone is used as the default availability zone. If a value specified in this setting is not found in the availability zone list, the setting will be ignored and the behavior will be same as when Any is specified.

LAUNCH_INSTANCE_LEGACY_ENABLED

New in version 8.0.0(Liberty).

Changed in version 9.0.0(Mitaka): The default value for this setting has been changed to False

Deprecated since version 19.1.0(Wallaby): The Python Launch Instance workflow is deprecated. Consider switching to the AngularJS workflow instead.

Default: False

This setting enables the Python Launch Instance workflow.

Note: It is possible to run both the AngularJS and Python workflows simultaneously, so the other may be need to be toggled with *LAUNCH_INSTANCE_NG_ENABLED*

LAUNCH_INSTANCE_NG_ENABLED

New in version 8.0.0(Liberty).

Changed in version 9.0.0(Mitaka): The default value for this setting has been changed to True

Default: True

This setting enables the AngularJS Launch Instance workflow.

Note: It is possible to run both the AngularJS and Python workflows simultaneously, so the other may be need to be toggled with *LAUNCH_INSTANCE_LEGACY_ENABLED*

OPENSTACK_ENABLE_PASSWORD_RETRIEVE

New in version 2014.1(Icehouse).

Default: "False"

When set, enables the instance action Retrieve password allowing password retrieval from metadata service.

OPENSTACK_HYPERVERSOR_FEATURES

New in version 2012.2(Folsom).

Changed in version 2014.1(Icehouse): `can_set_mount_point` and `can_set_password` now default to False

Default:

```
{
    'can_set_mount_point': False,
    'can_set_password': False,
    'requires_keypair': False,
    'enable_quotas': True
}
```

A dictionary containing settings which can be used to identify the capabilities of the hypervisor for Nova.

The Xen Hypervisor has the ability to set the mount point for volumes attached to instances (other Hypervisors currently do not). Setting `can_set_mount_point` to True will add the option to set the mount point from the UI.

Setting `can_set_password` to True will enable the option to set an administrator password when launching or rebuilding an instance.

Setting `requires_keypair` to True will require users to select a key pair when launching an instance.

Setting `enable_quotas` to False will make Horizon treat all Nova quotas as disabled, thus it wont try to modify them. By default, quotas are enabled.

OPENSTACK_INSTANCE_RETRIEVE_IP_ADDRESSES

New in version 13.0.0(Queens).

Default: True

This settings controls whether IP addresses of servers are retrieved from neutron in the project instance table. Setting this to False may mitigate a performance issue in the project instance table in large deployments.

If your deployment has no support of floating IP like provider network scenario, you can set this to False in most cases. If your deployment supports floating IP, read the detail below and understand the under-the-hood before setting this to False.

Nova has a mechanism to cache network info but it is not fast enough in some cases. For example, when a user associates a floating IP or updates an IP address of an server port, it is not reflected to the nova

network info cache immediately. This means an action which a user makes from the horizon instance table is not reflected into the table content just after the action. To avoid this, horizon retrieves IP address info from neutron when retrieving a list of servers from nova.

On the other hand, this operation requires a full list of neutron ports and can potentially lead to a performance issue in large deployments ([bug 1722417](#)). This issue can be avoided by skipping querying IP addresses to neutron and setting this to `False` achieves this. Note that when disabling the query to neutron it takes some time until associated floating IPs are visible in the project instance table and users may reload the table to check them.

OPENSTACK_USE_SIMPLE_TENANT_USAGE

New in version 19.0.0(Wallaby).

Default: `True`

This setting controls whether `SimpleTenantUsage` nova API is used in the usage overview. According to feedbacks to the horizon team, the usage of `SimpleTenantUsage` can cause performance issues in the nova API in larger deployments. Try to set this to `False` for such cases.

Swift

SWIFT_FILE_TRANSFER_CHUNK_SIZE

New in version 2015.1(Kilo).

Default: `512 * 1024`

This setting specifies the size of the chunk (in bytes) for downloading objects from Swift. Do not make it very large (higher than several dozens of Megabytes, exact number depends on your connection speed), otherwise you may encounter socket timeout. The default value is 524288 bytes (or 512 Kilobytes).

SWIFT_STORAGE_POLICY_DISPLAY_NAMES

New in version 18.3.0(Ussuri).

Default: `{}`

A dictionary mapping from the swift storage policy name to an alternate, user friendly display name which will be rendered on the dashboard. If no display is specified for a storage policy, the storage policy name will be used verbatim.

Django Settings

Note: This is not meant to be anywhere near a complete list of settings for Django. You should always consult the [upstream documentation](#), especially with regards to deployment considerations and security best-practices.

ADD_INSTALLED_APPS

New in version 2015.1(Kilo).

See also:

[Djangos INSTALLED_APPS documentation](#)

A list of Django applications to be prepended to the `INSTALLED_APPS` setting. Allows extending the list of installed applications without having to override it completely.

ALLOWED_HOSTS

New in version 2013.2(Havana).

See also:

[Djangos ALLOWED_HOSTS documentation](#)

Default: `['localhost']`

This list should contain names (or IP addresses) of the host running the dashboard; if its being accessed via name, the DNS name (and probably short-name) should be added, if its accessed via IP address, that should be added. The setting may contain more than one entry.

Note: `ALLOWED_HOSTS` is required. If Horizon is running in production (`DEBUG` is `False`), set this with the list of host/domain names that the application can serve. For more information see [Djangos Allowed Hosts documentation](#)

DEBUG

New in version 2011.2(Cactus).

See also:

[Djangos DEBUG documentation](#)

Default: `True`

Controls whether unhandled exceptions should generate a generic 500 response or present the user with a pretty-formatted debug information page.

When set, `CACHED_TEMPLATE_LOADERS` will not be cached.

This setting should **always** be set to `False` for production deployments as the debug page can display sensitive information to users and attackers alike.

SECRET_KEY

New in version 2012.1(Essex).

See also:

[Djangos SECRET_KEY documentation](#)

This should absolutely be set to a unique (and secret) value for your deployment. Unless you are running a load-balancer with multiple Horizon installations behind it, each Horizon instance should have a unique secret key.

Note: Setting a custom secret key:

You can either set it to a specific value or you can let Horizon generate a default secret key that is unique on this machine, regardless of the amount of Python WSGI workers (if used behind Apache+mod_wsgi). However, there may be situations where you would want to set this explicitly, e.g. when multiple dashboard instances are distributed on different machines (usually behind a load-balancer). Either you have to make sure that a session gets all requests routed to the same dashboard instance or you set the same SECRET_KEY for all of them.

```
from horizon.utils import secret_key

SECRET_KEY = secret_key.generate_or_read_from_file(
    os.path.join(LOCAL_PATH, '.secret_key_store'))
```

The `local_settings.py.example` file includes a quick-and-easy way to generate a secret key for a single installation.

STATIC_ROOT

New in version 8.0.0(Liberty).

See also:

[Djangos STATIC_ROOT documentation](#)

Default: `<path_to_horizon>/static`

The absolute path to the directory where static files are collected when collectstatic is run.

STATIC_URL

New in version 8.0.0(Liberty).

See also:

[Djangos STATIC_URL documentation](#)

Default: `/static/`

URL that refers to files in `STATIC_ROOT`.

By default this value is `WEBROOT/static/`.

This value can be changed from the default. When changed, the alias in your webserver configuration should be updated to match.

Note: The value for `STATIC_URL` must end in `/`.

This value is also available in the `scss` namespace with the variable name `$static_url`. Make sure you run `python manage.py collectstatic` and `python manage.py compress` after any changes to this value in `settings.py`.

TEMPLATES

New in version 10.0.0(Newton).

See also:

[Djangos TEMPLATES documentation](#)

Horizons usage of the `TEMPLATES` involves 3 further settings below; it is generally advised to use those before attempting to alter the `TEMPLATES` setting itself.

ADD_TEMPLATE_DIRS

New in version 15.0.0(Stein).

Template directories defined here will be added to `DIRS` option of Django `TEMPLATES` setting. It is useful when you would like to load deployment-specific templates.

ADD_TEMPLATE_LOADERS

New in version 10.0.0(Newton).

Template loaders defined here will be loaded at the end of `TEMPLATE_LOADERS`, after the `CACHED_TEMPLATE_LOADERS` and will never have a cached output.

CACHED_TEMPLATE_LOADERS

New in version 10.0.0(Newton).

Template loaders defined here will have their output cached if `DEBUG` is set to `False`.

TEMPLATE_LOADERS

New in version 10.0.0(Newton).

These template loaders will be the first loaders and get loaded before the CACHED_TEMPLATE_LOADERS. Use ADD_TEMPLATE_LOADERS if you want to add loaders at the end and not cache loaded templates. After the whole settings process has gone through, TEMPLATE_LOADERS will be:

```

TEMPLATE_LOADERS += (
    ('django.template.loaders.cached.Loader', CACHED_TEMPLATE_LOADERS),
) + tuple(ADD_TEMPLATE_LOADERS)

```

Other Settings

KUBECONFIG_ENABLED

New in version TBD.

Default: False

Kubernetes clusters can use Keystone as an external identity provider. Horizon can generate a kubeconfig file from the application credentials control panel which can be used for authenticating with a Kubernetes cluster. This setting enables this behavior.

See also:

[*KUBECONFIG_KUBERNETES_URL*](#) and [*KUBECONFIG_CERTIFICATE_AUTHORITY_DATA*](#) to provide parameters for the kubeconfig file.

KUBECONFIG_KUBERNETES_URL

New in version TBD.

Default: ""

A Kubernetes API endpoint URL to be included in the generated kubeconfig file.

See also:

[*KUBECONFIG_ENABLED*](#) to enable the kubeconfig file generation.

KUBECONFIG_CERTIFICATE_AUTHORITY_DATA

New in version TBD.

Default: ""

Kubernetes API endpoint certificate authority data to be included in the generated kubeconfig file.

See also:

[*KUBECONFIG_ENABLED*](#) to enable the kubeconfig file generation.

2.2.2 Pluggable Panels and Groups

Introduction

Horizon allows dashboards, panels and panel groups to be added without modifying the default settings. Pluggable settings are a mechanism to allow settings to be stored in separate files. Those files are read at startup and used to modify the default settings.

The default location for the dashboard configuration files is `openstack_dashboard/enabled`, with another directory, `openstack_dashboard/local/enabled` for local overrides. Both sets of files will be loaded, but the settings in `openstack_dashboard/local/enabled` will overwrite the default ones. The settings are applied in alphabetical order of the filenames. If the same dashboard has configuration files in `enabled` and `local/enabled`, the local name will be used. Note, that since names of python modules cant start with a digit, the files are usually named with a leading underscore and a number, so that you can control their order easily.

General Pluggable Settings

Before we describe the specific use cases, the following keys can be used in any pluggable settings file:

ADD_EXCEPTIONS

New in version 2014.1(Icehouse).

A dictionary of exception classes to be added to `HORIZON['exceptions']`.

ADD_INSTALLED_APPS

New in version 2014.1(Icehouse).

A list of applications to be prepended to `INSTALLED_APPS`. This is needed to expose static files from a plugin.

ADD_ANGULAR_MODULES

New in version 2014.2(Juno).

A list of AngularJS modules to be loaded when Angular bootstraps. These modules are added as dependencies on the root Horizon application `horizon`.

ADD_JS_FILES

New in version 2014.2(Juno).

A list of javascript source files to be included in the compressed set of files that are loaded on every page. This is needed for AngularJS modules that are referenced in `ADD_ANGULAR_MODULES` and therefore need to be included in every page.

ADD_JS_SPEC_FILES

New in version 2015.1(Kilo).

A list of javascript spec files to include for integration with the Jasmine spec runner. Jasmine is a behavior-driven development framework for testing JavaScript code.

ADD_SCSS_FILES

New in version 8.0.0(Liberty).

A list of scss files to be included in the compressed set of files that are loaded on every page. We recommend one scss file per dashboard, use `@import` if you need to include additional scss files for panels.

ADD_XSTATIC_MODULES

New in version 14.0.0(Rocky).

A list of xstatic modules containing javascript and scss files to be included in the compressed set of files that are loaded on every page. Related files specified in `ADD_XSTATIC_MODULES` do not need to be included in `ADD_JS_FILES`. This option expects a list of tuples, each consists of a xstatic module and a list of javascript files to be loaded if any. For more details, please check the comment of `BASE_XSTATIC_MODULES` in `openstack_dashboard/utils/settings.py`.

Example:

```
ADD_XSTATIC_MODULES = [  
    ('xstatic.pkg.foo', ['foo.js']),  
    ('xstatic.pkg.bar', None),  
]
```

AUTO_DISCOVER_STATIC_FILES

New in version 8.0.0(Liberty).

If set to `True`, JavaScript files and static angular html template files will be automatically discovered from the `static` folder in each apps listed in `ADD_INSTALLED_APPS`.

JavaScript source files will be ordered based on naming convention: files with extension `.module.js` listed first, followed by other JavaScript source files.

JavaScript files for testing will also be ordered based on naming convention: files with extension *.mock.js* listed first, followed by files with extension *.spec.js*.

If `ADD_JS_FILES` and/or `ADD_JS_SPEC_FILES` are also specified, files manually listed there will be appended to the auto-discovered files.

DISABLED

New in version 2014.1(Icehouse).

If set to `True`, this settings file will not be added to the settings.

EXTRA_STEPS

New in version 14.0.0(Rocky).

Extra workflow steps can be added to a workflow in horizon or other horizon plugins by using this setting. Extra steps will be shown after default steps defined in a corresponding workflow.

This is a dict setting. A key of the dict specifies a workflow which extra step(s) are added. The key must match a full class name of the target workflow.

A value of the dict is a list of full name of an extra step classes (where a module name and a class name must be delimited by a period). Steps specified via `EXTRA_STEPS` will be displayed in the order of being registered.

Example:

```
EXTRA_STEPS = {
    'openstack_dashboard.dashboards.identity.projects.workflows.UpdateQuota':
    (
        ('openstack_dashboard.dashboards.identity.projects.workflows.'
         'UpdateVolumeQuota'),
        ('openstack_dashboard.dashboards.identity.projects.workflows.'
         'UpdateNetworkQuota'),
    ),
}
```

EXTRA_TABS

New in version 14.0.0(Rocky).

Extra tabs can be added to a tab group implemented in horizon or other horizon plugins by using this setting. Extra tabs will be shown after default tabs defined in a corresponding tab group.

This is a dict setting. A key of the dict specifies a tab group which extra tab(s) are added. The key must match a full class name of the target tab group.

A value of the dict is a list of full name of an extra tab classes (where a module name and a class name must be delimited by a period). Tabs specified via `EXTRA_TABS` will be displayed in the order of being registered.

There might be cases where you would like to specify the order of the extra tabs as multiple horizon plugins can register extra tabs. You can specify a priority of each tab in `EXTRA_TABS` by using a tuple of priority and a tab class as an element of a dict value instead of a full name of an extra tab. Priority is an integer of a tab and a tab with a lower value will be displayed first. If a priority of an extra tab is omitted, `0` is assumed as a priority.

Example:

```
EXTRA_TABS = {
    'openstack_dashboard.dashboards.project.networks.tabs.NetworkDetailsTabs
↪': (
        'openstack_dashboard.dashboards.project.networks.subnets.tabs.
↪SubnetsTab',
        'openstack_dashboard.dashboards.project.networks.ports.tabs.PortsTab',
    ),
}
```

Example (with priority):

```
EXTRA_TABS = {
    'openstack_dashboard.dashboards.project.networks.tabs.NetworkDetailsTabs
↪': (
        (1, 'openstack_dashboard.dashboards.project.networks.subnets.tabs.
↪SubnetsTab'),
        (2, 'openstack_dashboard.dashboards.project.networks.ports.tabs.
↪PortsTab'),
    ),
}
```

UPDATE_HORIZON_CONFIG

New in version 2014.2(Juno).

A dictionary of values that will replace the values in `HORIZON_CONFIG`.

Pluggable Settings for Dashboards

New in version 2014.1(Icehouse).

The following keys are specific to registering a dashboard:

DASHBOARD

New in version 2014.1(Icehouse).

The slug of the dashboard to be added to `HORIZON['dashboards']`. Required.

DEFAULT

New in version 2014.1(Icehouse).

If set to `True`, this dashboard will be set as the default dashboard.

Examples

To disable a dashboard locally, create a file `openstack_dashboard/local/enabled/_40_dashboard-name.py` with the following content:

```
DASHBOARD = '<dashboard-name>'
DISABLED = True
```

To add a Tuskar-UI (Infrastructure) dashboard, you have to install it, and then create a file `openstack_dashboard/local/enabled/_50_tuskar.py` with:

```
from tuskar_ui import exceptions

DASHBOARD = 'infrastructure'
ADD_INSTALLED_APPS = [
    'tuskar_ui.infrastructure',
]
ADD_EXCEPTIONS = {
    'recoverable': exceptions.RECOVERABLE,
    'not_found': exceptions.NOT_FOUND,
    'unauthorized': exceptions.UNAUTHORIZED,
}
```

Pluggable Settings for Panels

New in version 2014.1(Icehouse).

The following keys are specific to registering or removing a panel:

PANEL

New in version 2014.1(Icehouse).

The slug of the panel to be added to `HORIZON_CONFIG`. Required.

PANEL_DASHBOARD

New in version 2014.1(Icehouse).

The slug of the dashboard the PANEL associated with. Required.

PANEL_GROUP

New in version 2014.1(Icehouse).

The slug of the panel group the PANEL is associated with. If you want the panel to show up without a panel group, use the panel group default.

DEFAULT_PANEL

New in version 2014.1(Icehouse).

If set, it will update the default panel of the PANEL_DASHBOARD.

ADD_PANEL

New in version 2014.1(Icehouse).

Python panel class of the PANEL to be added.

REMOVE_PANEL

New in version 2014.1(Icehouse).

If set to True, the PANEL will be removed from PANEL_DASHBOARD/PANEL_GROUP.

Examples

To add a new panel to the Admin panel group in Admin dashboard, create a file `openstack_dashboard/local/enabled/_60_admin_add_panel.py` with the following content:

```
PANEL = 'plugin_panel'  
PANEL_DASHBOARD = 'admin'  
PANEL_GROUP = 'admin'  
ADD_PANEL = 'test_panels.plugin_panel.panel.PluginPanel'
```

To remove Info panel from Admin panel group in Admin dashboard locally, create a file `openstack_dashboard/local/enabled/_70_admin_remove_panel.py` with the following content:

```
PANEL = 'info'  
PANEL_DASHBOARD = 'admin'  
PANEL_GROUP = 'admin'  
REMOVE_PANEL = True
```

To change the default panel of Admin dashboard to Instances panel, create a file `openstack_dashboard/local/enabled/_80_admin_default_panel.py` with the following content:

```
PANEL = 'instances'  
PANEL_DASHBOARD = 'admin'  
PANEL_GROUP = 'admin'  
DEFAULT_PANEL = 'instances'
```

Pluggable Settings for Panel Groups

New in version 2014.1(Icehouse).

The following keys are specific to registering a panel group:

PANEL_GROUP

New in version 2014.1(Icehouse).

The slug of the panel group to be added to `HORIZON_CONFIG`. Required.

PANEL_GROUP_NAME

New in version 2014.1(Icehouse).

The display name of the `PANEL_GROUP`. Required.

PANEL_GROUP_DASHBOARD

New in version 2014.1(Icehouse).

The slug of the dashboard the `PANEL_GROUP` associated with. Required.

Examples

To add a new panel group to the Admin dashboard, create a file `openstack_dashboard/local/enabled/_90_admin_add_panel_group.py` with the following content:

```
PANEL_GROUP = 'plugin_panel_group'  
PANEL_GROUP_NAME = 'Plugin Panel Group'  
PANEL_GROUP_DASHBOARD = 'admin'
```

2.2.3 Customizing Horizon

See also:

You may also be interested in *Themes* and *Branding Horizon*.

Changing the Site Title

The OpenStack Dashboard Site Title branding (i.e. **OpenStack** Dashboard) can be overwritten by adding the attribute `SITE_BRANDING` to `local_settings.py` with the value being the desired name.

The file `local_settings.py` can be found at the Horizon directory path of `openstack_dashboard/local/local_settings.py`.

Changing the Brand Link

The logo also acts as a hyperlink. The default behavior is to redirect to `horizon:user_home`. By adding the attribute `SITE_BRANDING_LINK` with the desired url target e.g., `http://sample-company.com` in `local_settings.py`, the target of the hyperlink can be changed.

Customizing the Footer

It is possible to customize the global and login footers by using Django's recursive inheritance to extend the `base.html`, `auth/login.html`, and `auth/_login_form.html` templates. You do this by naming your template the same name as the template you wish to extend and only overriding the blocks you wish to change.

Your themes `base.html`:

```
{% extends "base.html" %}

{% block footer %}
  <p>My custom footer</p>
{% endblock %}
```

Your themes `auth/login.html`:

```
{% extends "auth/login.html" %}

{% block footer %}
  <p>My custom login footer</p>
{% endblock %}
```

Your themes `auth/_login_form.html`:

```
{% extends "auth/_login_form.html" %}

{% block login_footer %}
  {% comment %}
    You MUST have block.super because that includes the login button.
  {% endcomment %}
```

(continues on next page)

(continued from previous page)

```
{{ block.super }}  
<p>My custom login form footer</p>  
{% endblock %}
```

See the [example theme](#) for a working theme that uses these blocks.

Modifying Existing Dashboards and Panels

If you wish to alter dashboards or panels which are not part of your codebase, you can specify a custom python module which will be loaded after the entire Horizon site has been initialized, but prior to the URLconf construction. This allows for common site-customization requirements such as:

- Registering or unregistering panels from an existing dashboard.
- Changing the names of dashboards and panels.
- Re-ordering panels within a dashboard or panel group.

Default Horizon panels are loaded based upon files within the `openstack_dashboard/enabled/` folder. These files are loaded based upon the filename order, with space left for more files to be added. There are some example files available within this folder, with the `.example` suffix added. Developers and deployers should strive to use this method of customization as much as possible, and support for this is given preference over more exotic methods such as monkey patching and overrides files.

Horizon customization module (overrides)

Horizon has a global overrides mechanism available to perform customizations that are not yet customizable via configuration settings. This file can perform monkey patching and other forms of customization which are not possible via the enabled folders customization method.

This method of customization is meant to be available for deployers of Horizon, and use of this should be avoided by Horizon plugins at all cost. Plugins needing this level of monkey patching and flexibility should instead look for changing their `__init__.py` file and performing customizations through other means.

To specify the python module containing your modifications, add the key `customization_module` to your `HORIZON_CONFIG` dictionary in `local_settings.py`. The value should be a string containing the path to your module in dotted python path notation. Example:

```
HORIZON_CONFIG["customization_module"] = "my_project.overrides"
```

You can do essentially anything you like in the customization module. For example, you could change the name of a panel:

```
from django.utils.translation import ugettext_lazy as _  
  
import horizon  
  
# Rename "User Settings" to "User Options"  
settings = horizon.get_dashboard("settings")  
user_panel = settings.get_panel("user")  
user_panel.name = _("User Options")
```


Or get the instances panel:

```
projects_dashboard = horizon.get_dashboard("project")
instances_panel = projects_dashboard.get_panel("instances")
```

Or just remove it entirely:

```
projects_dashboard.unregister(instances_panel.__class__)
```

You cannot unregister a `default_panel`. If you wish to remove a `default_panel`, you need to make a different panel in the dashboard as a `default_panel` and then unregister the former. For example, if you wished to remove the `overview_panel` from the Project dashboard, you could do the following:

```
project = horizon.get_dashboard('project')
project.default_panel = "instances"
overview = project.get_panel('overview')
project.unregister(overview.__class__)
```

You can also override existing methods with your own versions:

```
from openstack_dashboard.dashboards.admin.info import tabs
from openstack_dashboard.dashboards.project.instances import tables

NO = lambda *x: False

tables.AssociateIP.allowed = NO
tables.SimpleAssociateIP.allowed = NO
tables.SimpleDisassociateIP.allowed = NO
```

You could also customize what columns are displayed in an existing table, by redefining the `columns` attribute of its `Meta` class. This can be achieved in 3 steps:

1. Extend the table that you wish to modify
2. Redefine the `columns` attribute under the `Meta` class for this new table
3. Modify the `table_class` attribute for the related view so that it points to the new table

For example, if you wished to remove the Admin State column from the `NetworksTable`, you could do the following:

```
from openstack_dashboard.dashboards.project.networks import tables
from openstack_dashboard.dashboards.project.networks import views

class MyNetworksTable(tables.NetworksTable):

    class Meta(tables.NetworksTable.Meta):
        columns = ('name', 'subnets', 'shared', 'status')

views.IndexView.table_class = MyNetworksTable
```

If you want to add a column you can override the parent table in a similar way, add the new column definition and then use the `Meta columns` attribute to control the column order as needed.

Note: `my_project.overrides` needs to be importable by the python process running Horizon. If your module is not installed as a system-wide python package, you can either make it installable (e.g., with a `setup.py`) or you can adjust the python path used by your WSGI server to include its location.

Probably the easiest way is to add a `python-path` argument to the `WSGIDaemonProcess` line in Apaches Horizon config.

Assuming your `my_project` module lives in `/opt/python/my_project`, youd make it look like the following:

```
WSGIDaemonProcess [... existing options ...] python-path=/opt/python
```

Customize the project and user table columns

Keystone V3 has a place to store extra information regarding project and user. Using the override mechanism described in *Horizon customization module (overrides)*, Horizon is able to show these extra information as a custom column. For example, if a user in Keystone has an attribute `phone_num`, you could define new column:

```
from django.utils.translation import ugettext_lazy as _

from horizon import forms
from horizon import tables

from openstack_dashboard.dashboards.identity.users import tables as user_
↳ tables
from openstack_dashboard.dashboards.identity.users import views

class MyUsersTable(user_tables.UsersTable):
    phone_num = tables.Column('phone_num',
                              verbose_name=_('Phone Number'),
                              form_field=forms.CharField(),)

    class Meta(user_tables.UsersTable.Meta):
        columns = ('name', 'description', 'phone_num')

views.IndexView.table_class = MyUsersTable
```

Customize Angular dashboards

In Angular, you may write a plugin to extend certain features. Two components in the Horizon framework that make this possible are the extensibility service and the resource type registry service. The `extensibleService` allows certain Horizon elements to be extended dynamically, including add, remove, and replace. The `resourceTypeRegistry` service provides methods to set and get information pertaining to a resource type object. We use Heat type names like `OS::Glance::Image` as our reference name.

Some information you may place in the registry include:

- API to fetch data from

- Property names
- Actions (e.g. Create Volume)
- URL paths to detail view or detail drawer
- Property information like labels or formatting for property values

These properties in the registry use the extensibility service (as of Newton release):

- globalActions
- batchActions
- itemActions
- detailViews
- tableColumns
- filterFacets

Using the information from the registry, we can build out our dashboard panels. Panels use the high-level directive `hzResourceTable` that replaces common templates so we do not need to write boilerplate HTML and controller code. It gives developers a quick way to build a new table or change an existing table.

Note: You may still choose to use the HTML template for complete control of form and functionality. For example, you may want to create a custom footer. You may also use the `hzDynamicTable` directive (what `hzResourceTable` uses under the hood) directly. However, neither of these is extensible. You would need to override the panel completely.

This is a sample module file to demonstrate how to make some customizations to the Images Panel.:

```
(function() {
  'use strict';

  angular
    .module('horizon.app.core.images')
    .run(customizeImagePanel);

  customizeImagePanel.$inject = [
    'horizon.framework.conf.resource-type-registry.service',
    'horizon.app.core.images.basePath',
    'horizon.app.core.images.resourceType',
    'horizon.app.core.images.actions.surprise.service'
  ];

  function customizeImagePanel(registry, basePath, imageResourceType, ↵
↵surpriseService) {
    // get registry for ``OS::Glance::Image``
    registry = registry.getResourceType(imageResourceType);

    // replace existing Size column to make the font color red
    var column = {
```

(continues on next page)

(continued from previous page)

```
        id: 'size',
        priority: 2,
        template: '<a style="color:red;">${ item.size | bytes $}</a>'
    };
    registry.tableColumns.replace('size', column);

    // add a new detail view
    registry.detailsViews
        .append({
            id: 'anotherDetailView',
            name: gettext('Another Detail View'),
            template: basePath + 'demo/detail.html'
        });

    // set a different summary drawer template
    registry.setSummaryTemplateUrl(basePath + 'demo/drawer.html');

    // add a new global action
    registry.globalActions
        .append({
            id: 'surpriseAction',
            service: surpriseService,
            template: {
                text: gettext('Surprise')
            }
        });
    }
}());
```

Additionally, you should have content defined in `detail.html` and `drawer.html`, as well as define the `surpriseService` which is based off the actions directive and needs `allowed` and `perform` methods defined.

Icons

Horizon uses font icons from Font Awesome. Please see [Font Awesome](#) for instructions on how to use icons in the code.

To add icon to Table Action, use `icon` property. Example:

```
class CreateSnapshot(tables.LinkAction):
    name = "snapshot"
    verbose_name = _("Create Snapshot")
    icon = "camera"
```

Additionally, the site-wide default button classes can be configured by setting `ACTION_CSS_CLASSES` to a tuple of the classes you wish to appear on all action buttons in your `local_settings.py` file.

Custom Stylesheets

It is possible to define custom stylesheets for your dashboards. Horizons base template `openstack_dashboard/templates/base.html` defines multiple blocks that can be overridden.

To define custom css files that apply only to a specific dashboard, create a base template in your dashboards templates folder, which extends Horizons base template e.g. `openstack_dashboard/dashboards/my_custom_dashboard/templates/my_custom_dashboard/base.html`.

In this template, redefine block `css`. (Dont forget to include `_stylesheets.html` which includes all Horizons default stylesheets.):

```
{% extends 'base.html' %}

{% block css %}
    {% include "_stylesheets.html" %}

    {% load compress %}
    {% compress css %}
        <link href='{{ STATIC_URL }}my_custom_dashboard/scss/my_custom_dashboard.
↪scss' type='text/scss' media='screen' rel='stylesheet' />
    {% endcompress %}
{% endblock %}
```

The custom stylesheets then reside in the dashboards own static folder `openstack_dashboard/dashboards/my_custom_dashboard/static/my_custom_dashboard/scss/my_custom_dashboard.scss`.

All dashboards templates have to inherit from dashboards base.html:

```
{% extends 'my_custom_dashboard/base.html' %}
...
```

Custom Javascript

Similarly to adding custom styling (see above), it is possible to include custom javascript files.

All Horizons javascript files are listed in the `openstack_dashboard/templates/horizon/_scripts.html` partial template, which is included in Horizons base template in block `js`.

To add custom javascript files, create an `_scripts.html` partial template in your dashboard `openstack_dashboard/dashboards/my_custom_dashboard/templates/my_custom_dashboard/_scripts.html` which extends `horizon/_scripts.html`. In this template override the block `custom_js_files` including your custom javascript files:

```
{% extends 'horizon/_scripts.html' %}

{% block custom_js_files %}
    <script src='{{ STATIC_URL }}my_custom_dashboard/js/my_custom_js.js' type=
↪'text/javascript' charset='utf-8'></script>
{% endblock %}
```

In your dashboards own base template `openstack_dashboard/dashboards/my_custom_dashboard/templates/my_custom_dashboard/base.html` override block `js` with inclusion of dashboards own `_scripts.html`:

```
{% block js %}
    {% include "my_custom_dashboard/_scripts.html" %}
{% endblock %}
```

The result is a single compressed js file consisting both Horizon and dashboards custom scripts.

Custom Head js

Additionally, some scripts require you to place them within the pages `<head>` tag. To do this, recursively extend the `base.html` template in your theme to override the `custom_head_js` block.

Your themes `base.html`:

```
{% extends "base.html" %}

{% block custom_head_js %}
    <script src='{{ STATIC_URL }}/my_custom_dashboard/js/my_custom_js.js' type=
    ↪'text/javascript' charset='utf-8'></script>
{% endblock %}
```

See the example theme for a working theme that uses these blocks.

Warning: Dont use the `custom_head_js` block for analytics tracking. See below.

Custom Analytics

For analytics or tracking scripts you should avoid the `custom_head_js` block. We have a specific block instead called `custom_analytics`. Much like the `custom_head_js` block this inserts additional content into the head of the `base.html` template and it will be on all pages.

The reason for an analytics specific block is that for security purposes we want to be able to turn off tracking on certain pages that we deem sensitive. This is done for the safety of the users and the cloud admins. By using this block instead, pages using `base.html` can override it themselves when they want to avoid tracking. They cant simply override the custom js because it may be non-tracking code.

Your themes `base.html`:

```
{% extends "base.html" %}

{% block custom_analytics %}
    <script src='{{ STATIC_URL }}/my_custom_dashboard/js/my_tracking_js.js'
    ↪type='text/javascript' charset='utf-8'></script>
{% endblock %}
```

See the example theme for a working theme that uses these blocks.

Customizing Meta Attributes

To add custom metadata attributes to your projects base template use the `custom_metadata` block. To do this, recursively extend the `base.html` template in your theme to override the `custom_metadata` block. The contents of this block will be inserted into the pages `<head>` just after the default Horizon meta tags.

Your themes `base.html`:

```
{% extends "base.html" %}

{% block custom_metadata %}
  <meta name="description" content="My custom metadata.">
{% endblock %}
```

See the [example theme](#) for a working theme that uses these blocks.

2.2.4 Themes

As of the Kilo release, styling for the OpenStack Dashboard can be altered through the use of a theme. A theme is a directory containing a `_variables.scss` file to override the color codes used throughout the SCSS and a `_styles.scss` file with additional styles to load after dashboard styles have loaded.

As of the Mitaka release, Horizon can be configured to run with multiple themes available at run time. It uses a browser cookie to allow users to toggle between the configured themes. By default, Horizon is configured with the two standard themes available: `default` and `material`.

To configure or alter the available themes, set `AVAILABLE_THEMES` in `local_settings.py` to a list of tuples, such that `('name', 'label', 'path')`

name The key by which the theme value is stored within the cookie

label The label shown in the theme toggle under the User Menu

path The directory location for the theme. The path must be relative to the `openstack_dashboard` directory or an absolute path to an accessible location on the file system

To use a custom theme, set `AVAILABLE_THEMES` in `local_settings.py` to a list of themes. If you wish to run in a mode similar to legacy Horizon, set `AVAILABLE_THEMES` with a single tuple, and the theme toggle will not be available at all through the application to allow user configuration themes.

For example, a configuration with multiple themes:

```
AVAILABLE_THEMES = [
    ('default', 'Default', 'themes/default'),
    ('material', 'Material', 'themes/material'),
]
```

A configuration with a single theme:

```
AVAILABLE_THEMES = [
    ('default', 'Default', 'themes/default'),
]
```

Both the Dashboard custom variables and Bootstrap variables can be overridden. For a full list of the Dashboard SCSS variables that can be changed, see the variables file at `openstack_dashboard/static/dashboard/scss/_variables.scss`.

In order to build a custom theme, both `_variables.scss` and `_styles.scss` are required and `_variables.scss` must provide all the default Bootstrap variables.

Inherit from an Existing Theme

Custom themes must implement all of the Bootstrap variables required by Horizon in `_variables.scss` and `_styles.scss`. To make this easier, you can inherit the variables needed in the default theme and only override those that you need to customize. To inherit from the default theme, put this in your themes `_variables.scss`:

```
@import "/themes/default/variables";
```

Once you have made your changes you must re-generate the static files with:

```
python manage.py collectstatic
```

By default, all of the themes configured by `AVAILABLE_THEMES` setting are collected by horizon during the `collectstatic` process. By default, the themes are collected into the dynamic `static/themes` directory, but this location can be customized via the `local_settings.py` variable: `THEME_COLLECTION_DIR`

Once collected, any theme configured via `AVAILABLE_THEMES` is available to inherit from by importing its variables and styles from its collection directory. The following is an example of inheriting from the material theme:

```
@import "/themes/material/variables";  
@import "/themes/material/styles";
```

All themes will need to be configured in `AVAILABLE_THEMES` to allow inheritance. If you wish to inherit from a theme, but not show that theme as a selectable option in the theme picker widget, then simply configure the `SELECTABLE_THEMES` to exclude the parent theme. `SELECTABLE_THEMES` must be of the same format as `AVAILABLE_THEMES`. It defaults to `AVAILABLE_THEMES` if it is not set explicitly.

Bootswatch

Horizon packages the Bootswatch SCSS files for use with its `material` theme. Because of this, it is simple to use an existing Bootswatch theme as a base. This is due to the fact that Bootswatch is loaded as a 3rd party static asset, and therefore is automatically collected into the `static` directory in `/horizon/lib/`. The following is an example of how to inherit from Bootswatch's `darkly` theme:

```
@import "/horizon/lib/bootswatch/darkly/variables";  
@import "/horizon/lib/bootswatch/darkly/bootswatch";
```


Organizing Your Theme Directory

A custom theme directory can be organized differently, depending on the level of customization that is desired, as it can include static files as well as Django templates. It can include special subdirectories that will be used differently: `static`, `templates` and `img`.

The `static` Folder

If the theme folder contains a sub-folder called `static`, then that sub folder will be used as the **static root of the theme**. I.e., Horizon will look in that sub-folder for the `_variables.scss` and `_styles.scss` files. The contents of this folder will also be served up at `/static/custom`.

The `templates` Folder

If the theme folder contains a sub-folder `templates`, then the path to that sub-folder will be prepended to the `TEMPLATE_DIRS` tuple to allow for theme specific template customizations.

Using the `templates` Folder

Any Django template that is used in Horizon can be overridden through a theme. This allows highly customized user experiences to exist within the scope of different themes. Any template that is overridden must adhere to the same directory structure that the extending template expects.

For example, if you wish to customize the sidebar, Horizon expects the template to live at `horizon/_sidebar.html`. You would need to duplicate that directory structure under your `templates` directory, such that your override would live at `{ theme_path }/templates/horizon/_sidebar.html`.

The `img` Folder

If the static root of the theme folder contains an `img` directory, then all images that make use of the `{% themable_asset %}` templatetag can be overridden.

These assets include `logo.svg`, `splash-logo.svg` and `favicon.ico`, however overriding the SVG/GIF assets used by Heat within the `dashboard/img` folder is not currently supported.

Customizing the Logo

Simple

If you wish to customize the logo that is used on the splash screen or in the top navigation bar, then you need to create an `img` directory under your themes static root directory and place your custom `logo.svg` or `logo-splash.svg` within it.

If you wish to override the `logo.svg` using the previous method, and if the image used is larger than the height of the top navigation, then the image will be constrained to fit within the height of nav. You can customize the height of the top navigation bar by customizing the SCSS variable: `$navbar-height`. If the images height is smaller than the navbar height, then the image will retain its original resolution and size, and simply be centered vertically in the available space.

Prior to the Kilo release the images files inside of Horizon needed to be replaced by your images files or the Horizon stylesheets needed to be altered to point to the location of your image.

Advanced

If you need to do more to customize the logo than simply replacing the existing SVG, then you can also override the `_brand.html` through a custom theme. To use this technique, simply add a `templates/header/_brand.html` to the root of your custom theme, and add markup directly to the file. For an example of how to do this, see `openstack_dashboard/themes/material/templates/header/_brand.html`.

The splash / login panel can also be customized by adding `templates/auth/_splash.html`. See `openstack_dashboard/themes/material/templates/auth/_splash.html` for an example.

2.2.5 Branding Horizon

As of the Liberty release, Horizon has begun to conform more strictly to Bootstrap standards in an effort to embrace more responsive web design as well as alleviate the future need to re-brand new functionality for every release.

Supported Components

The following components, organized by release, are the only ones that make full use of the Bootstrap theme architecture.

- 8.0.0 (Liberty)
 - *Top Navbar*
 - *Side Nav*
 - *Pie Charts*
- 9.0.0 (Mitaka)
 - *Tables*
 - *Bar Charts*
 - *Login*
 - *Tabs*
 - *Alerts*
 - *Checkboxes*

Step 1

The first step needed to create a custom branded theme for Horizon is to create a custom Bootstrap theme. There are several tools to aid in this. Some of the more useful ones include:

- [Bootstrap](#)
- [Paintstrap](#)
- [Bootstrap](#)

Note: Bootstrap uses LESS by default, but we use SCSS. All of the above tools will provide the `variables.less` file, which will need to be converted to `_variables.scss`

Top Navbar

The top navbar in Horizon now uses a native Bootstrap navbar. There are a number of variables that can be used to customize this element. Please see the **Navbar** section of your variables file for specifics on what can be set: any variables that use `navbar-default`.

It is important to also note that the navbar now uses native Bootstrap dropdowns, which are customizable with variables. Please see the **Dropdowns** section of your variables file.

The top navbar is now responsive on smaller screens. When the window size hits your `$screen-sm` value, the topbar will compress into a design that is better suited for small screens.

Side Nav

The side navigation component has been refactored to use the native Stacked Pills element from Bootstrap. See **Pills** section of your variables file for specific variables to customize.

Charts

Pie Charts

Pie Charts are SVG elements. SVG elements allow CSS customizations for only a basic elements look and feel (i.e. colors, size).

Since there is no native element in Bootstrap specifically for pie charts, the look and feel of the charts are inheriting from other elements of the theme. Please see `_pie_charts.scss` for specifics.

Bar Charts

Bar Charts can be either a Bootstrap Progress Bar or an SVG element. Either implementation will use the Bootstrap Progress Bar styles.

The SVG implementation will not make use of the customized Progress Bar height though, so it is recommended that Bootstrap Progress Bars are used whenever possible.

Please see `_bar_charts.scss` for specifics on what can be customized for SVGs. See the **Progress bars** section of your variables file for specific variables to customize.

Tables

The standard Django tables now make use of the native Bootstrap table markup. See **Tables** section of your variables file for variables to customize.

The standard Bootstrap tables will be borderless by default. If you wish to add a border, like the default theme, see `openstack_dashboard/themes/default/horizon/components/_tables.scss`

Login

Login Splash Page

The login splash page now uses a standard Bootstrap panel in its implementation. See the **Panels** section in your variables file to variables to easily customize.

Modal Login

The modal login experience, as used when switching regions, uses a standard Bootstrap dialog. See the **Modals** section of your variables file for specific variables to customize.

Tabs

The standard tabs make use of the native Bootstrap tab markup.

See **Tabs** section of your variables file for variables to customize.

Alerts

Alerts use the basic Bootstrap brand colors. See **Colors** section of your variables file for specifics.

Checkboxes

Horizon uses icon fonts to represent checkboxes. In order to customize this, you simply need to override the standard scss. For an example of this, see [themes/material/static/horizon/components/_checkboxes.scss](#)

Bootstrap and Material Design

[Bootstrap](#) is a collection of free themes for Bootstrap and is now available for use in Horizon.

In order to showcase what can be done to enhance an existing Bootstrap theme, Horizon now includes a secondary theme, roughly based on [Google's Material Design](#) called `material`. Bootstrap's **Paper** is a simple Bootstrap implementation of Material Design and is used by `material`.

Bootstrap provides a number of other themes, that once Horizon is fully theme compliant, will allow easy toggling and customizations for darker or accessibility driven experiences.

Development Tips

When developing a new theme for Horizon, it is required that the dynamically generated `static` directory be cleared after each change and the server restarted. This is not always ideal. If you wish to develop and not have to restart the server each time, it is recommended that you configure your development environment to not run in OFFLINE mode. Simply verify the following settings in your `local_settings.py`:

```
COMPRESS_OFFLINE = False
COMPRESS_ENABLED = False
```

2.3 OpenStack Dashboard User Documentation

As a cloud end user, you can use the OpenStack dashboard to provision your own resources within the limits set by administrators. You can modify the examples provided in this section to create other types and sizes of server instances.

2.3.1 Log in to the dashboard

The dashboard is generally installed on the controller node.

1. Ask the cloud operator for the host name or public IP address from which you can access the dashboard, and for your user name and password. If the cloud supports multi-domain model, you also need to ask for your domain name.
2. Open a web browser that has JavaScript and cookies enabled.

Note: To use the Virtual Network Computing (VNC) client for the dashboard, your browser must support HTML5 Canvas and HTML5 WebSockets. The VNC client is based on noVNC. For details, see [noVNC: HTML5 VNC Client](#). For a list of supported browsers, see [Browser support](#).

3. In the address bar, enter the host name or IP address for the dashboard, for example, `https://ipAddressOrHostName/`.

Note: If a certificate warning appears when you try to access the URL for the first time, a self-signed certificate is in use, which is not considered trustworthy by default. Verify the certificate or add an exception in the browser to bypass the warning.

4. On the *Log In* page, enter your user name and password, and click *Sign In*. If the cloud supports multi-domain model, you also need to enter your domain name.

The top of the window displays your user name. You can also access the *Settings* tab (*OpenStack dashboard Settings tab*) or sign out of the dashboard.

The visible tabs and functions in the dashboard depend on the access permissions, or roles, of the user you are logged in as.

- If you are logged in as an end user, the *Project* tab (*OpenStack dashboard Project tab*) and *Identity* tab (*OpenStack dashboard Identity tab*) are displayed.
- If you are logged in as an administrator, the *Project* tab (*OpenStack dashboard Project tab*) and *Admin* tab (*OpenStack dashboard Admin tab*) and *Identity* tab (*OpenStack dashboard Identity tab*) are displayed.

OpenStack dashboard Project tab

Projects are organizational units in the cloud and are also known as tenants or accounts. Each user is a member of one or more projects. Within a project, a user creates and manages instances.

From the *Project* tab, you can view and manage the resources in a selected project, including instances and images. You can select the project from the drop-down menu at the top left. If the cloud supports multi-domain model, you can also select the domain from this menu.

From the *Project* tab, you can access the following categories:

- *API Access*: View API endpoints.

Compute tab

- *Overview*: View reports for the project.
- *Instances*: View, launch, create a snapshot from, stop, pause, or reboot instances, or connect to them through VNC.
- *Images*: View images and instance snapshots created by project users, plus any images that are publicly available. Create, edit, and delete images, and launch instances from images and snapshots.
- *Key Pairs*: View, create, edit, import, and delete key pairs.

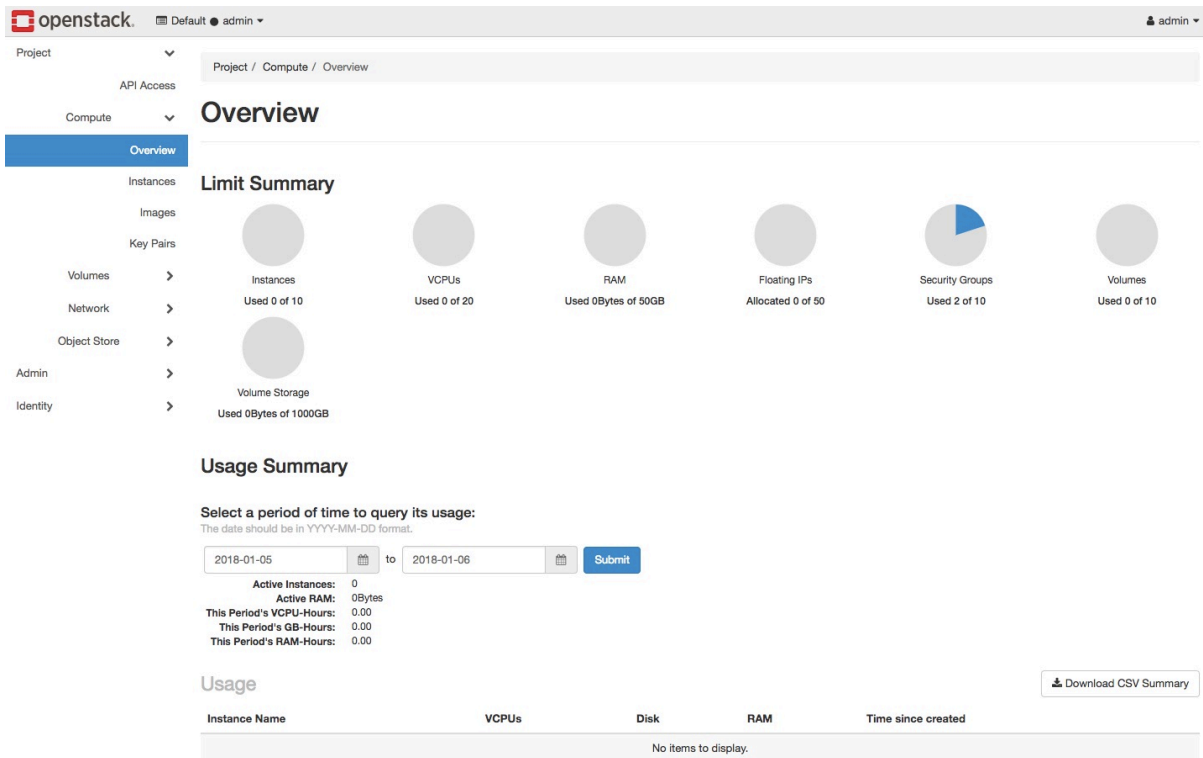


Fig. 1: Figure: Project tab

Volume tab

- *Volumes*: View, create, edit, and delete volumes.
- *Backups*: View, create, edit, and delete backups.
- *Snapshots*: View, create, edit, and delete volume snapshots.
- *Consistency Groups*: View, create, edit, and delete consistency groups.
- *Consistency Group Snapshots*: View, create, edit, and delete consistency group snapshots.

Network tab

- *Network Topology*: View the network topology.
- *Networks*: Create and manage public and private networks.
- *Routers*: Create and manage routers.
- *Security Groups*: View, create, edit, and delete security groups and security group rules..
- *Floating IPs*: Allocate an IP address to or release it from a project.

Object Store tab

- *Containers*: Create and manage containers and objects.

OpenStack dashboard Admin tab

Administrative users can use the *Admin* tab to view usage and to manage instances, volumes, flavors, images, networks, and so on.

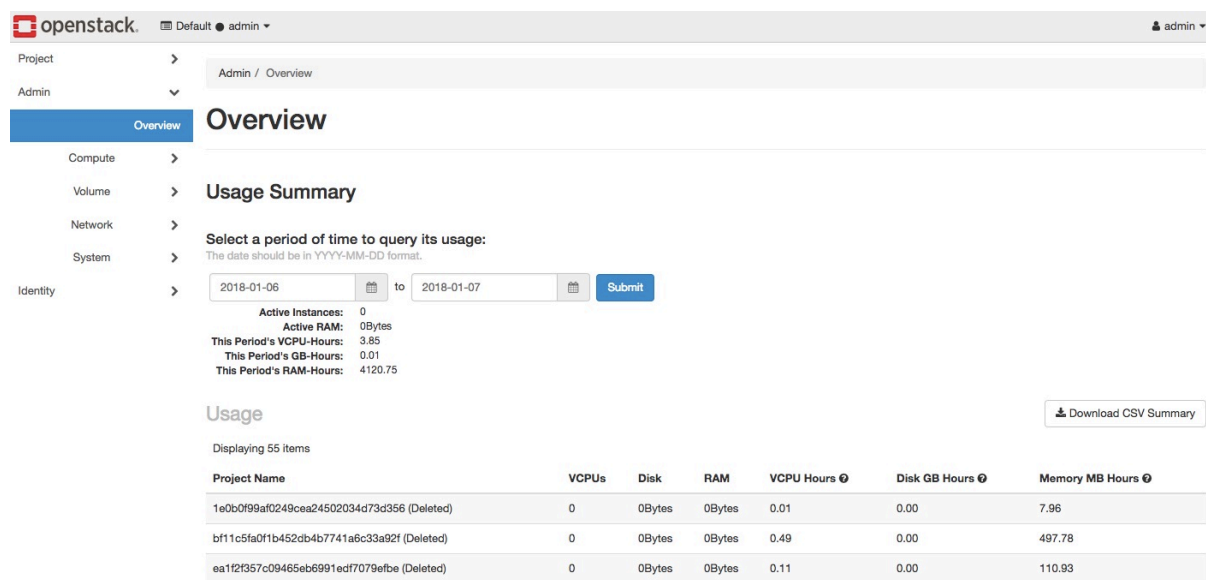


Fig. 2: Figure: Admin tab

From the *Admin* tab, you can access the following category to complete these tasks:

Overview tab

- *Overview*: View basic reports.

Compute tab

- *Hypervisors*: View the hypervisor summary.
- *Host Aggregates*: View, create, and edit host aggregates. View the list of availability zones.
- *Instances*: View, pause, resume, suspend, migrate, soft or hard reboot, and delete running instances that belong to users of some, but not all, projects. Also, view the log for an instance or access an instance through VNC.
- *Flavors*: View, create, edit, view extra specifications for, and delete flavors. A flavor is the size of an instance.
- *Images*: View, create, edit properties for, and delete custom images.

Volume tab

- *Volumes*: View, create, manage, and delete volumes.
- *Snapshots*: View, manage, and delete volume snapshots.
- *Volume Types*: View, create, manage, and delete volume types.

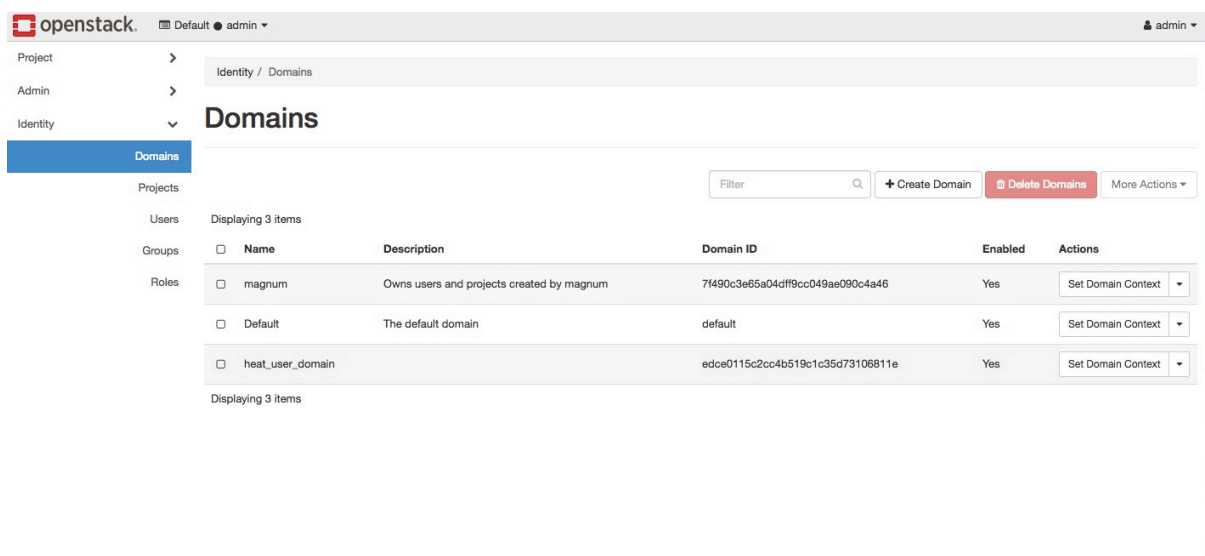
Network tab

- *Networks*: View, create, edit properties for, and delete networks.
- *Routers*: View, create, edit properties for, and delete routers.
- *Floating IPs*: Allocate an IP address for a project or release it.

System tab

- *Defaults*: View default quota values. Quotas are hard-coded in OpenStack Compute and define the maximum allowable size and number of resources.
- *Metadata Definitions*: Import namespace and view the metadata information.
- *System Information*: Use the following tabs to view the service information:
 - *Services*: View a list of the services.
 - *Compute Services*: View a list of all Compute services.
 - *Block Storage Services*: View a list of all Block Storage services.
 - *Network Agents*: View the network agents.

OpenStack dashboard Identity tab



The screenshot shows the OpenStack dashboard interface for the Identity tab, specifically the Domains section. The page title is "Domains" and it displays a table of domains. The table has columns for Name, Description, Domain ID, Enabled, and Actions. Three domains are listed: magnum, Default, and heat_user_domain. The magnum domain is selected, and its description is "Owns users and projects created by magnum". The Default domain has the description "The default domain". The heat_user_domain has the description "heat_user_domain".

Name	Description	Domain ID	Enabled	Actions
magnum	Owns users and projects created by magnum	71490c3e65a04dff9cc049ae090c4a46	Yes	Set Domain Context
Default	The default domain	default	Yes	Set Domain Context
heat_user_domain		edce0115c2cc4b519c1c35d73106811e	Yes	Set Domain Context

Fig. 3: Figure:Identity tab

- *Projects*: View, create, assign users to, remove users from, and delete projects.
- *Users*: View, create, enable, disable, and delete users.

OpenStack dashboard Settings tab

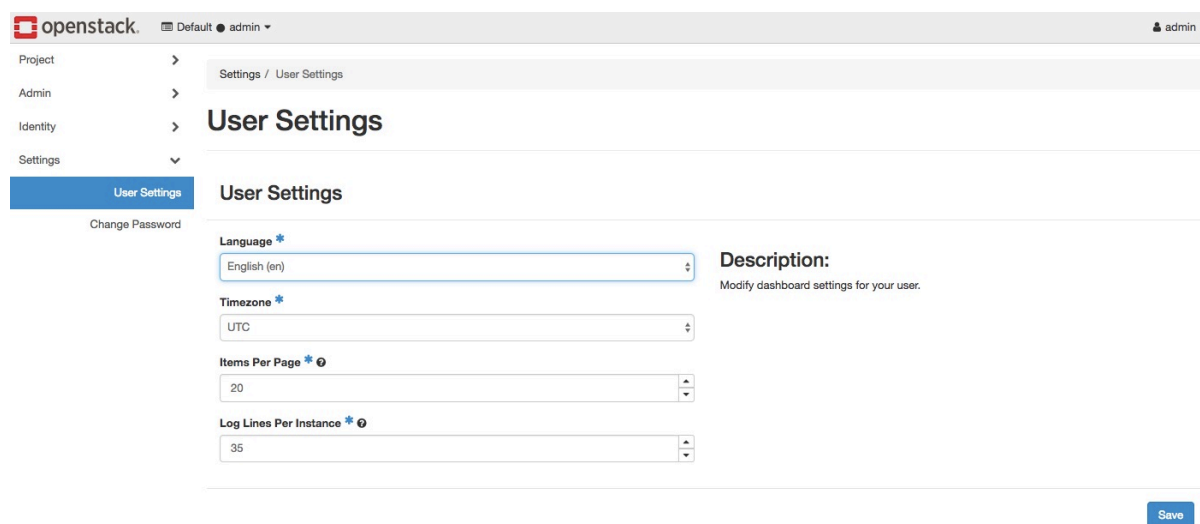


Fig. 4: Figure:Settings tab

Click the *Settings* button from the user drop down menu at the top right of any page, you will see the *Settings* tab.

- *User Settings*: View and manage dashboard settings.
- *Change Password*: Change the password of the user.

2.3.2 Upload and manage images

A virtual machine image, referred to in this document simply as an image, is a single file that contains a virtual disk that has a bootable operating system installed on it. Images are used to create virtual machine instances within the cloud. For information about creating image files, see the [OpenStack Virtual Machine Image Guide](#).

Depending on your role, you may have permission to upload and manage virtual machine images. Operators might restrict the upload and management of images to cloud administrators or operators only. If you have the appropriate privileges, you can use the dashboard to upload and manage images in the admin project.

Note: You can also use the **openstack** and **glance** command-line clients or the Image service to manage images.

Upload an image

Follow this procedure to upload an image to a project:

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Compute* tab and click *Images* category.
4. Click *Create Image*.

The *Create An Image* dialog box appears.

Fig. 5: Dashboard Create Image

5. Enter the following values:

<i>Image Name</i>	Enter a name for the image.
<i>Image Description</i>	Enter a brief description of the image.
<i>Image Source</i>	Choose the image source from the dropdown list. Your choices are <i>Image Location</i> and <i>Image File</i> .
<i>Image File or Image Location</i>	Based on your selection for <i>Image Source</i> , you either enter the location URL of the image in the <i>Image Location</i> field, or browse for the image file on your file system and add it.
<i>Format</i>	Select the image format (for example, QCOW2) for the image.
<i>Architecture</i>	Specify the architecture. For example, <code>i386</code> for a 32-bit architecture or <code>x86_64</code> for a 64-bit architecture.
<i>Minimum Disk (GB)</i>	Leave this field empty.
<i>Minimum RAM (MB)</i>	Leave this field empty.
<i>Copy Data</i>	Specify this option to copy image data to the Image service.
<i>Visibility</i>	The access permission for the image. Public or Private .
<i>Protected</i>	Select this check box to ensure that only users with permissions can delete the image. Yes or No .
<i>Image Metadata</i>	Specify this option to add resource metadata. The glance Metadata Catalog provides a list of metadata image definitions. (Note: Not all cloud providers enable this feature.)

6. Click *Create Image*.

The image is queued to be uploaded. It might take some time before the status changes from Queued to Active.

Update an image

Follow this procedure to update an existing image.

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. Select the image that you want to edit.
4. In the *Actions* column, click the menu button and then select *Edit Image* from the list.
5. In the *Edit Image* dialog box, you can perform various actions. For example:
 - Change the name of the image.
 - Change the description of the image.
 - Change the format of the image.
 - Change the minimum disk of the image.
 - Change the minimum RAM of the image.
 - Select the *Public* button to make the image public.
 - Clear the *Private* button to make the image private.

- Change the metadata of the image.

6. Click *Edit Image*.

Delete an image

Deletion of images is permanent and **cannot** be reversed. Only users with the appropriate permissions can delete images.

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Compute* tab and click *Images* category.
4. Select the images that you want to delete.
5. Click *Delete Images*.
6. In the *Confirm Delete Images* dialog box, click *Delete Images* to confirm the deletion.

2.3.3 Configure access and security for instances

Before you launch an instance, you should add security group rules to enable users to ping and use SSH to connect to the instance. Security groups are sets of IP filter rules that define networking access and are applied to all instances within a project. To do so, you either add rules to the default security group *Add a rule to the default security group* or add a new security group with rules.

Key pairs are SSH credentials that are injected into an instance when it is launched. To use key pair injection, the image that the instance is based on must contain the `cloud-init` package. Each project should have at least one key pair. For more information, see the section *Add a key pair*.

If you have generated a key pair with an external tool, you can import it into OpenStack. The key pair can be used for multiple instances that belong to a project. For more information, see the section *Import a key pair*.

Note: A key pair belongs to an individual user, not to a project. To share a key pair across multiple users, each user needs to import that key pair.

When an instance is created in OpenStack, it is automatically assigned a fixed IP address in the network to which the instance is assigned. This IP address is permanently associated with the instance until the instance is terminated. However, in addition to the fixed IP address, a floating IP address can also be attached to an instance. Unlike fixed IP addresses, floating IP addresses are able to have their associations modified at any time, regardless of the state of the instances involved.

Add a rule to the default security group

This procedure enables SSH and ICMP (ping) access to instances. The rules apply to all instances within a given project, and should be set for every project unless there is a reason to prohibit SSH or ICMP access to the instances.

This procedure can be adjusted as necessary to add additional security group rules to a project, if your cloud requires them.

Note: When adding a rule, you must specify the protocol used with the destination port or source port.

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Network* tab. The *Security Groups* tab shows the security groups that are available for this project.
4. Select the default security group and click *Manage Rules*.
5. To allow SSH access, click *Add Rule*.
6. In the *Add Rule* dialog box, enter the following values:
 - **Rule:** SSH
 - **Remote:** CIDR
 - **CIDR:** 0.0.0.0/0

Note: To accept requests from a particular range of IP addresses, specify the IP address block in the *CIDR* box.

7. Click *Add*.

Instances will now have SSH port 22 open for requests from any IP address.
8. To add an ICMP rule, click *Add Rule*.
9. In the *Add Rule* dialog box, enter the following values:
 - **Rule:** All ICMP
 - **Direction:** Ingress
 - **Remote:** CIDR
 - **CIDR:** 0.0.0.0/0

10. Click *Add*.

Instances will now accept all incoming ICMP packets.

Add a key pair

Create at least one key pair for each project.

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Compute* tab.
4. Click the *Key Pairs* tab, which shows the key pairs that are available for this project.
5. Click *Create Key Pair*.
6. In the *Create Key Pair* dialog box, enter a name for your key pair, and click *Create Key Pair*.
7. The private key will be downloaded automatically.
8. To change its permissions so that only you can read and write to the file, run the following command:

```
$ chmod 0600 yourPrivateKey.pem
```

Note: If you are using the Dashboard from a Windows computer, use PuTTYgen to load the *.pem file and convert and save it as *.ppk. For more information see the [WinSCP web page for PuTTYgen](#).

9. To make the key pair known to SSH, run the **ssh-add** command.

```
$ ssh-add yourPrivateKey.pem
```

Import a key pair

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Compute* tab.
4. Click the *Key Pairs* tab, which shows the key pairs that are available for this project.
5. Click *Import Key Pair*.
6. In the *Import Key Pair* dialog box, enter the name of your key pair, copy the public key into the *Public Key* box, and then click *Import Key Pair*.

The Compute database registers the public key of the key pair.

The Dashboard lists the key pair on the *Key Pairs* tab.

Allocate a floating IP address to an instance

When an instance is created in OpenStack, it is automatically assigned a fixed IP address in the network to which the instance is assigned. This IP address is permanently associated with the instance until the instance is terminated.

However, in addition to the fixed IP address, a floating IP address can also be attached to an instance. Unlike fixed IP addresses, floating IP addresses can have their associations modified at any time, regardless of the state of the instances involved. This procedure details the reservation of a floating IP address from an existing pool of addresses and the association of that address with a specific instance.

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Network* tab.
4. Click the *Floating IPs* tab, which shows the floating IP addresses allocated to instances.
5. Click *Allocate IP To Project*.
6. Choose the pool from which to pick the IP address.
7. Click *Allocate IP*.
8. In the *Floating IPs* list, click *Associate*.
9. In the *Manage Floating IP Associations* dialog box, choose the following options:
 - The *IP Address* field is filled automatically, but you can add a new IP address by clicking the + button.
 - In the *Port to be associated* field, select a port from the list.
The list shows all the instances with their fixed IP addresses.
10. Click *Associate*.

Note: To disassociate an IP address from an instance, click the *Disassociate* button.

To release the floating IP address back into the floating IP pool, click the *Release Floating IP* option in the *Actions* column.

2.3.4 Launch and manage instances

Instances are virtual machines that run inside the cloud. You can launch an instance from the following sources:

- Images uploaded to the Image service.
- Image that you have copied to a persistent volume. The instance launches from the volume, which is provided by the `cinder-volume` API through iSCSI.
- Instance snapshot that you took.

Launch an instance

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Compute* tab and click *Instances* category.

The dashboard shows the instances with its name, its private and floating IP addresses, size, status, task, power state, and so on.

4. Click *Launch Instance*.
5. In the *Launch Instance* dialog box, specify the following values:

Details tab

Instance Name Assign a name to the virtual machine.

Note: The name you assign here becomes the initial host name of the server. If the name is longer than 63 characters, the Compute service truncates it automatically to ensure dnsmasq works correctly.

After the server is built, if you change the server name in the API or change the host name directly, the names are not updated in the dashboard.

Server names are not guaranteed to be unique when created so you could have two instances with the same host name.

Description You can assign a brief description of the virtual machine.

Availability Zone By default, this value is set to the availability zone given by the cloud provider (for example, us-west or apac-south). For some cases, it could be nova.

Count To launch multiple instances, enter a value greater than 1. The default is 1.

Source tab

Instance Boot Source Your options are:

Boot from image If you choose this option, a new field for *Image Name* displays. You can select the image from the list.

Boot from snapshot If you choose this option, a new field for *Instance Snapshot* displays. You can select the snapshot from the list.

Boot from volume If you choose this option, a new field for *Volume* displays. You can select the volume from the list.

Boot from image (creates a new volume) With this option, you can boot from an image and create a volume by entering the *Device Size* and *Device Name* for your volume. Click the *Delete Volume on Instance Delete* option to delete the volume on deleting the instance.

Boot from volume snapshot (creates a new volume) Using this option, you can boot from a volume snapshot and create a new volume by choosing *Volume Snapshot* from a list and adding a *Device Name* for your volume. Click the *Delete Volume on Instance Delete* option to delete the volume on deleting the instance.

Image Name This field changes based on your previous selection. If you have chosen to launch an instance using an image, the *Image Name* field displays. Select the image name from the dropdown list.

Instance Snapshot This field changes based on your previous selection. If you have chosen to launch an instance using a snapshot, the *Instance Snapshot* field displays. Select the snapshot name from the dropdown list.

Volume This field changes based on your previous selection. If you have chosen to launch an instance using a volume, the *Volume* field displays. Select the volume name from the dropdown list. If you want to delete the volume on instance delete, check the *Delete Volume on Instance Delete* option.

Flavor tab

Flavor Specify the size of the instance to launch.

Note: The flavor is selected based on the size of the image selected for launching an instance. For example, while creating an image, if you have entered the value in the *Minimum RAM (MB)* field as 2048, then on selecting the image, the default flavor is `m1.small`.

Networks tab

Selected Networks To add a network to the instance, click the + in the *Available* field.

Network Ports tab

Ports Activate the ports that you want to assign to the instance.

Security Groups tab

Security Groups Activate the security groups that you want to assign to the instance.

Security groups are a kind of cloud firewall that define which incoming network traffic is forwarded to instances.

If you have not created any security groups, you can assign only the default security group to the instance.

Key Pair tab

Key Pair Specify a key pair.

If the image uses a static root password or a static key set (neither is recommended), you do not need to provide a key pair to launch the instance.

Configuration tab

Customization Script Source Specify a customization script that runs after your instance launches.

Metadata tab

Available Metadata Add Metadata items to your instance.

6. Click *Launch Instance*.

The instance starts on a compute node in the cloud.

Note: If you did not provide a key pair, security groups, or rules, users can access the instance only from inside the cloud through VNC. Even pinging the instance is not possible without an ICMP rule configured.

You can also launch an instance from the *Images* or *Volumes* category when you launch an instance from an image or a volume respectively.

When you launch an instance from an image, OpenStack creates a local copy of the image on the compute node where the instance starts.

For details on creating images, see [Creating images manually](#) in the *OpenStack Virtual Machine Image Guide*.

When you launch an instance from a volume, note the following steps:

- To select the volume from which to launch, launch an instance from an arbitrary image on the volume. The arbitrary image that you select does not boot. Instead, it is replaced by the image on the volume that you choose in the next steps.

To boot a Xen image from a volume, the image you launch in must be the same type, fully virtualized or paravirtualized, as the one on the volume.

- Select the volume or volume snapshot from which to boot. Enter a device name. Enter `vda` for KVM images or `xvda` for Xen images.

Note: When running QEMU without support for the hardware virtualization, set `cpu_mode="none"` alongside `virt_type=qemu` in `/etc/nova/nova-compute.conf` to solve the following error:

```
libvirtError: unsupported configuration: CPU mode 'host-model'  
for ``x86_64`` qemu domain on ``x86_64`` host is not supported by hypervisor
```

Connect to your instance by using SSH

To use SSH to connect to your instance, use the downloaded keypair file.

Note: The user name is `ubuntu` for the Ubuntu cloud images on TryStack.

1. Copy the IP address for your instance.
2. Use the `ssh` command to make a secure connection to the instance. For example:

```
$ ssh -i MyKey.pem ubuntu@10.0.0.2
```

3. At the prompt, type `yes`.

It is also possible to SSH into an instance without an SSH keypair, if the administrator has enabled root password injection. For more information about root password injection, see [Injecting the administrator password](#) in the *OpenStack Administrator Guide*.

Track usage for instances

You can track usage for instances for each project. You can track costs per month by showing meters like number of vCPUs, disks, RAM, and uptime for all your instances.

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Compute* tab and click *Overview* category.
4. To query the instance usage for a month, select a month and click *Submit*.
5. To download a summary, click *Download CSV Summary*.

Create an instance snapshot

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Compute* tab and click the *Instances* category.
4. Select the instance from which to create a snapshot.
5. In the actions column, click *Create Snapshot*.
6. In the *Create Snapshot* dialog box, enter a name for the snapshot, and click *Create Snapshot*.

The *Images* category shows the instance snapshot.

To launch an instance from the snapshot, select the snapshot and click *Launch*. Proceed with launching an instance.

Manage an instance

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Compute* tab and click *Instances* category.
4. Select an instance.
5. In the menu list in the actions column, select the state.

You can resize or rebuild an instance. You can also choose to view the instance console log, edit instance or the security groups. Depending on the current state of the instance, you can pause, resume, suspend, soft or hard reboot, or terminate it.

2.3.5 Create and manage networks

The OpenStack Networking service provides a scalable system for managing the network connectivity within an OpenStack cloud deployment. It can easily and quickly react to changing network needs (for example, creating and assigning new IP addresses).

Networking in OpenStack is complex. This section provides the basic instructions for creating a network and a router. For detailed information about managing networks, refer to the [OpenStack Networking Guide](#).

Create a network

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Network* tab and click *Networks* category.
4. Click *Create Network*.
5. In the *Create Network* dialog box, specify the following values.

Network tab

Network Name: Specify a name to identify the network.

Shared: Share the network with other projects. Non admin users are not allowed to set shared option.

Admin State: The state to start the network in.

Create Subnet: Select this check box to create a subnet

You do not have to specify a subnet when you create a network, but if you do not specify a subnet, the network can not be attached to an instance.

Subnet tab

Subnet Name: Specify a name for the subnet.

Network Address: Specify the IP address for the subnet.

IP Version: Select IPv4 or IPv6.

Gateway IP: Specify an IP address for a specific gateway. This parameter is optional.

Disable Gateway: Select this check box to disable a gateway IP address.

Subnet Details tab

Enable DHCP: Select this check box to enable DHCP.

Allocation Pools: Specify IP address pools.

DNS Name Servers: Specify a name for the DNS server.

Host Routes: Specify the IP address of host routes.

6. Click *Create*.

The dashboard shows the network on the *Networks* tab.

Create a router

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Network* tab and click *Routers* category.
4. Click *Create Router*.
5. In the *Create Router* dialog box, specify a name for the router and *External Network*, and click *Create Router*.

The new router is now displayed in the *Routers* tab.

6. To connect a private network to the newly created router, perform the following steps:
 - A) On the *Routers* tab, click the name of the router.
 - B) On the *Router Details* page, click the *Interfaces* tab, then click *Add Interface*.
 - C) In the *Add Interface* dialog box, select a *Subnet*.

Optionally, in the *Add Interface* dialog box, set an *IP Address* for the router interface for the selected subnet.

If you choose not to set the *IP Address* value, then by default OpenStack Networking uses the first host IP address in the subnet.

The *Router Name* and *Router ID* fields are automatically updated.

7. Click *Add Interface*.

You have successfully created the router. You can view the new topology from the *Network Topology* tab.

Create a port

1. Log in to the dashboard.
2. Select the appropriate project from the drop-down menu at the top left.
3. On the *Project* tab, click *Networks* category.
4. Click on the *Network Name* of the network in which the port has to be created.
5. Go to the *Ports* tab and click *Create Port*.
6. In the *Create Port* dialog box, specify the following values.

Name: Specify name to identify the port.

Device ID: Device ID attached to the port.

Device Owner: Device owner attached to the port.

Binding Host: The ID of the host where the port is allocated.

Binding VNIC Type: Select the VNIC type that is bound to the neutron port.

7. Click *Create Port*.

The new port is now displayed in the *Ports* list.

2.3.6 Create and manage object containers

OpenStack Object Storage (swift) is used for redundant, scalable data storage using clusters of standardized servers to store petabytes of accessible data. It is a long-term storage system for large amounts of static data which can be retrieved and updated.

OpenStack Object Storage provides a distributed, API-accessible storage platform that can be integrated directly into an application or used to store any type of file, including VM images, backups, archives, or media files. In the OpenStack dashboard, you can only manage containers and objects.

In OpenStack Object Storage, containers provide storage for objects in a manner similar to a Windows folder or Linux file directory, though they cannot be nested. An object in OpenStack consists of the file to be stored in the container and any accompanying metadata.

Create a container

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Object Store* tab and click *Containers* category.
4. Click *Container*.
5. In the *Create Container* dialog box, enter a name for the container, and then click *Create*.

You have successfully created a container.

Note: To delete a container, click the *More* button and select *Delete Container*.

Upload an object

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Object Store* tab and click *Containers* category.
4. Select the container in which you want to store your object.
5. Click the *Upload File* icon.

The *Upload File To Container: <name>* dialog box appears. <name> is the name of the container to which you are uploading the object.

6. Enter a name for the object.
7. Browse to and select the file that you want to upload.
8. Click *Upload File*.

You have successfully uploaded an object to the container.

Note: To delete an object, click the *More button* and select *Delete Object*.

Manage an object

To edit an object

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Object Store* tab and click *Containers* category.
4. Select the container in which you want to store your object.
5. Click the menu button and choose *Edit* from the dropdown list.

The *Edit Object* dialog box is displayed.

6. Browse to and select the file that you want to upload.
7. Click *Update Object*.

Note: To delete an object, click the menu button and select *Delete Object*.

To copy an object from one container to another

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Object Store* tab and click *Containers* category.
4. Select the container in which you want to store your object.
5. Click the menu button and choose *Copy* from the dropdown list.
6. In the *Copy Object* launch dialog box, enter the following values:
 - *Destination Container*: Choose the destination container from the list.
 - *Path*: Specify a path in which the new copy should be stored inside of the selected container.
 - *Destination object name*: Enter a name for the object in the new container.
7. Click *Copy Object*.

To create a metadata-only object without a file

You can create a new object in container without a file available and can upload the file later when it is ready. This temporary object acts a place-holder for a new object, and enables the user to share object metadata and URL info in advance.

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Object Store* tab and click *Containers* category.
4. Select the container in which you want to store your object.
5. Click *Upload Object*.

The *Upload Object To Container: <name>* dialog box is displayed.

<name> is the name of the container to which you are uploading the object.

6. Enter a name for the object.
7. Click *Update Object*.

To create a pseudo-folder

Pseudo-folders are similar to folders in your desktop operating system. They are virtual collections defined by a common prefix on the objects name.

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Object Store* tab and click *Containers* category.
4. Select the container in which you want to store your object.
5. Click *Create Pseudo-folder*.

The *Create Pseudo-Folder in Container <name>* dialog box is displayed. <name> is the name of the container to which you are uploading the object.

6. Enter a name for the pseudo-folder.
A slash (/) character is used as the delimiter for pseudo-folders in Object Storage.
7. Click *Create*.

2.3.7 Create and manage volumes

Volumes are block storage devices that you attach to instances to enable persistent storage. You can attach a volume to a running instance or detach a volume and attach it to another instance at any time. You can also create a snapshot from or delete a volume. Only administrative users can create volume types.

Create a volume

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Volumes* tab and click *Volumes* category.
4. Click *Create Volume*.

In the dialog box that opens, enter or select the following values.

Volume Name: Specify a name for the volume.

Description: Optionally, provide a brief description for the volume.

Volume Source: Select one of the following options:

- No source, empty volume: Creates an empty volume. An empty volume does not contain a file system or a partition table.
- Snapshot: If you choose this option, a new field for *Use snapshot as a source* displays. You can select the snapshot from the list.
- Image: If you choose this option, a new field for *Use image as a source* displays. You can select the image from the list.

- **Volume:** If you choose this option, a new field for *Use volume as a source* displays. You can select the volume from the list. Options to use a snapshot or a volume as the source for a volume are displayed only if there are existing snapshots or volumes.

Type: Leave this field blank.

Size (GB): The size of the volume in gibibytes (GiB).

Availability Zone: Select the Availability Zone from the list. By default, this value is set to the availability zone given by the cloud provider (for example, `us-west` or `apac-south`). For some cases, it could be `nova`.

5. Click *Create Volume*.

The dashboard shows the volume on the *Volumes* tab.

Attach a volume to an instance

After you create one or more volumes, you can attach them to instances. You can attach a volume to one instance at a time.

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Volumes* tab and click *Volumes* category.
4. Select the volume to add to an instance and click *Manage Attachments*.
5. In the *Manage Volume Attachments* dialog box, select an instance.
6. Enter the name of the device from which the volume is accessible by the instance.

Note: The actual device name might differ from the volume name because of hypervisor settings.

7. Click *Attach Volume*.

The dashboard shows the instance to which the volume is now attached and the device name.

You can view the status of a volume in the *Volumes* tab of the dashboard. The volume is either *Available* or *In-Use*.

Now you can log in to the instance and mount, format, and use the disk.

Detach a volume from an instance

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Volumes* tab and click the *Volumes* category.
4. Select the volume and click *Manage Attachments*.
5. Click *Detach Volume* and confirm your changes.

A message indicates whether the action was successful.

Create a snapshot from a volume

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Volumes* tab and click *Volumes* category.
4. Select a volume from which to create a snapshot.
5. In the *Actions* column, click *Create Snapshot*.
6. In the dialog box that opens, enter a snapshot name and a brief description.
7. Confirm your changes.

The dashboard shows the new volume snapshot in Volume Snapshots tab.

Edit a volume

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Volumes* tab and click *Volumes* category.
4. Select the volume that you want to edit.
5. In the *Actions* column, click *Edit Volume*.
6. In the *Edit Volume* dialog box, update the name and description of the volume.
7. Click *Edit Volume*.

Note: You can extend a volume by using the *Extend Volume* option available in the *More* dropdown list and entering the new value for volume size.

Delete a volume

When you delete an instance, the data in its attached volumes is not deleted.

1. Log in to the dashboard.
2. Select the appropriate project from the drop down menu at the top left.
3. On the *Project* tab, open the *Volumes* tab and click *Volumes* category.
4. Select the check boxes for the volumes that you want to delete.
5. Click *Delete Volumes* and confirm your choice.

A message indicates whether the action was successful.

2.3.8 Supported Browsers

Horizon is primarily tested and supported on the latest version of Firefox and the latest version of Chrome. Issues related to IE, Safari and Opera will also be considered.

This page aims to informally document what that means for different releases, everyone is warmly encouraged to update this page based on the versions they've tested with.

Legend:

- Very good: Very well tested, should work as expected
- Good: Moderately tested, should look nice and work fine, maybe a few visual hiccups
- Poor: Doesn't look good
- Broken: Essential functionality not working (link to bug in the notes)
- No: Not supported

	Status	Notes
Firefox	Very good	
Chrome	Very good	
MS Edge	Poor	There are some bugs but most of features work
IE 11	Poor	There are some bugs but most of features work
IE 10 and below	Not supported.	
Safari	Good	
Opera	?	It should work good with Webkit

2.4 Administration Guide

The OpenStack Dashboard is a web-based interface that allows you to manage OpenStack resources and services. The Dashboard allows you to interact with the OpenStack Compute cloud controller using the OpenStack APIs. For more information about installing and configuring the Dashboard, see the *Installation Guide* for your operating system.

2.4.1 Customize and configure the Dashboard

Once you have the Dashboard installed, you can customize the way it looks and feels to suit the needs of your environment, your project, or your business.

You can also configure the Dashboard for a secure HTTPS deployment, or an HTTP deployment. The standard OpenStack installation uses a non-encrypted HTTP channel, but you can enable SSL support for the Dashboard.

For information on configuring HTTPS or HTTP, see *Configure the Dashboard*.

Customize the Dashboard

The OpenStack Dashboard on Ubuntu installs the `openstack-dashboard-ubuntu-theme` package by default. If you do not want to use this theme, remove it and its dependencies:

```
# apt-get remove --auto-remove openstack-dashboard-ubuntu-theme
```

Note: This guide focuses on the `local_settings.py` file.

The following Dashboard content can be customized to suit your needs:

- Logo
- Site colors
- HTML title
- Logo link
- Help URL

Logo and site colors

1. Create two PNG logo files with transparent backgrounds using the following sizes:
 - Login screen: 365 x 50
 - Logged in banner: 216 x 35
2. Upload your new images to `/usr/share/openstack-dashboard/openstack_dashboard/static/dashboard/img/`.
3. Create a CSS style sheet in `/usr/share/openstack-dashboard/openstack_dashboard/static/dashboard/scss/`.
4. Change the colors and image file names as appropriate. Ensure the relative directory paths are the same. The following example file shows you how to customize your CSS file:

```
/*
 * New theme colors for dashboard that override the defaults:
 * dark blue: #355796 / rgb(53, 87, 150)
 * light blue: #BAD3E1 / rgb(186, 211, 225)
 *
 * By Preston Lee <plee@tgen.org>
 */
h1.brand {
background: #355796 repeat-x top left;
border-bottom: 2px solid #BAD3E1;
}
h1.brand a {
background: url(../img/my_cloud_logo_small.png) top left no-repeat;
}
#splash .login {
```

(continues on next page)

(continued from previous page)

```

background: #355796 url(../img/my_cloud_logo_medium.png) no-repeat center;
↪35px;
}
#splash .login .modal-header {
border-top: 1px solid #BAD3E1;
}
.btn-primary {
background-image: none !important;
background-color: #355796 !important;
border: none !important;
box-shadow: none;
}
.btn-primary:hover,
.btn-primary:active {
border: none;
box-shadow: none;
background-color: #BAD3E1 !important;
text-decoration: none;
}

```

5. Open the following HTML template in an editor of your choice:

```

/usr/share/openstack-dashboard/openstack_dashboard/templates/_stylesheets.
↪html

```

6. Add a line to include your newly created style sheet. For example, `custom.css` file:

```

<link href='{{ STATIC_URL }}bootstrap/css/bootstrap.min.css' media='screen
↪' rel='stylesheet' />
<link href='{{ STATIC_URL }}dashboard/css/{% choose_css %}' media='screen
↪' rel='stylesheet' />
<link href='{{ STATIC_URL }}dashboard/css/custom.css' media='screen' rel=
↪'stylesheet' />

```

7. Restart the Apache service.
8. To view your changes, reload your Dashboard. If necessary, go back and modify your CSS file as appropriate.

HTML title

1. Set the HTML title, which appears at the top of the browser window, by adding the following line to `local_settings.py`:

```

SITE_BRANDING = "Example, Inc. Cloud"

```

2. Restart Apache for this change to take effect.

Logo link

1. The logo also acts as a hyperlink. The default behavior is to redirect to `horizon:user_home`. To change this, add the following attribute to `local_settings.py`:

```
SITE_BRANDING_LINK = "http://example.com"
```

2. Restart Apache for this change to take effect.

Help URL

1. By default, the help URL points to <https://docs.openstack.org>. To change this, edit the following attribute in `local_settings.py`:

```
HORIZON_CONFIG["help_url"] = "http://openstack.mycompany.org"
```

2. Restart Apache for this change to take effect.

Configure the Dashboard

The following section on configuring the Dashboard for a secure HTTPS deployment, or a HTTP deployment, uses concrete examples to ensure the procedure is clear. The file path varies by distribution, however. If needed, you can also configure the VNC window size in the Dashboard.

Configure the Dashboard for HTTP

You can configure the Dashboard for a simple HTTP deployment. The standard installation uses a non-encrypted HTTP channel.

1. Specify the host for your Identity service endpoint in the `local_settings.py` file with the `OPENSTACK_HOST` setting.

The following example shows this setting:

```
import os

from django.utils.translation import gettext_lazy as _

DEBUG = False
TEMPLATE_DEBUG = DEBUG
PROD = True

SITE_BRANDING = 'OpenStack Dashboard'

# Ubuntu-specific: Enables an extra panel in the 'Settings' section
# that easily generates a Juju environments.yaml for download,
# preconfigured with endpoints and credentials required for bootstrap
# and service deployment.
ENABLE_JUJU_PANEL = True
```

(continues on next page)

(continued from previous page)

```
# Note: You should change this value
SECRET_KEY = 'elj1IWiLoWHgryYxFT6j7cM5fG00xWY0'

# Specify a regular expression to validate user passwords.
# HORIZON_CONFIG = {
#     "password_validator": {
#         "regex": '.*',
#         "help_text": _("Your password does not meet the requirements.")
#     }
# }

LOCAL_PATH = os.path.dirname(os.path.abspath(__file__))

CACHES = {
    'default': {
        'BACKEND' : 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION' : '127.0.0.1:11211'
    }
}

# Send email to the console by default
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
# Or send them to /dev/null
#EMAIL_BACKEND = 'django.core.mail.backends.dummy.EmailBackend'

# Configure these for your outgoing email host
# EMAIL_HOST = 'smtp.my-company.com'
# EMAIL_PORT = 25
# EMAIL_HOST_USER = 'djangomail'
# EMAIL_HOST_PASSWORD = 'top-secret!'

# For multiple regions uncomment this configuration, and add (endpoint,
↪title).
# AVAILABLE_REGIONS = [
#     ('http://cluster1.example.com/identity/v3', 'cluster1'),
#     ('http://cluster2.example.com/identity/v3', 'cluster2'),
# ]

OPENSTACK_HOST = "127.0.0.1"
OPENSTACK_KEYSTONE_URL = "http://%s/identity/v3" % OPENSTACK_HOST
OPENSTACK_KEYSTONE_DEFAULT_ROLE = "Member"

# The OPENSTACK_KEYSTONE_BACKEND settings can be used to identify the
# capabilities of the auth backend for Keystone.
# If Keystone has been configured to use LDAP as the auth backend then
↪set
# can_edit_user to False and name to 'ldap'.
#
```

(continues on next page)

(continued from previous page)

```

# TODO(tres): Remove these once Keystone has an API to identify auth_
↳backend.
OPENSTACK_KEYSTONE_BACKEND = {
    'name': 'native',
    'can_edit_user': True
}

# OPENSTACK_ENDPOINT_TYPE specifies the endpoint type to use for the_
↳endpoints
# in the Keystone service catalog. Use this setting when Horizon is_
↳running
# external to the OpenStack environment. The default is 'internalURL'.
#OPENSTACK_ENDPOINT_TYPE = "publicURL"

# The number of Swift containers and objects to display on a single page_
↳before
# providing a paging element (a "more" link) to paginate results.
API_RESULT_LIMIT = 1000

# If you have external monitoring links, eg:
# EXTERNAL_MONITORING = [
#     ['Nagios', 'http://foo.com'],
#     ['Ganglia', 'http://bar.com'],
# ]

LOGGING = {
    'version': 1,
    # When set to True this will disable all logging except
    # for loggers specified in this configuration dictionary. Note_
↳that
    # if nothing is specified here and disable_existing_loggers is_
↳True,
    # django.db.backends will still log unless it is disabled_
↳explicitly.
    'disable_existing_loggers': False,
    'handlers': {
        'null': {
            'level': 'DEBUG',
            'class': 'logging.NullHandler',
        },
        'console': {
            # Set the level to "DEBUG" for verbose output logging.
            'level': 'INFO',
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
        # Logging from django.db.backends is VERY verbose, send to_
↳null

```

(continues on next page)

(continued from previous page)

```

# by default.
'django.db.backends': {
    'handlers': ['null'],
    'propagate': False,
},
'horizon': {
    'handlers': ['console'],
    'propagate': False,
},
'novaclient': {
    'handlers': ['console'],
    'propagate': False,
},
'keystoneclient': {
    'handlers': ['console'],
    'propagate': False,
}
}
}

```

The service catalog configuration in the Identity service determines whether a service appears in the Dashboard. For the full listing, see *Settings Reference*.

2. Restart the Apache HTTP Server.
3. Restart memcached.

Configure the Dashboard for HTTPS

You can configure the Dashboard for a secured HTTPS deployment. While the standard installation uses a non-encrypted HTTP channel, you can enable SSL support for the Dashboard.

This example uses the `http://openstack.example.com` domain. Use a domain that fits your current setup.

1. In the `local_settings.py` file, update the following options:

```

CSRF_COOKIE_SECURE = True
SESSION_COOKIE_SECURE = True
SESSION_COOKIE_HTTPONLY = True

```

The other options require that HTTPS is enabled; these options defend against cross-site scripting.

2. Edit the `openstack-dashboard.conf` file as shown in the **Example After**:

Example Before

```

WSGIScriptAlias / /usr/share/openstack-dashboard/openstack_dashboard/wsgi.
↪py
WSGIDaemonProcess horizon user=www-data group=www-data processes=3
↪threads=10
Alias /static /usr/share/openstack-dashboard/openstack_dashboard/static/

```

(continues on next page)

(continued from previous page)

```
<Location />
  <ifVersion >=2.4>
    Require all granted
  </ifVersion>
  <ifVersion <2.4>
    Order allow,deny
    Allow from all
  </ifVersion>
</Location>
```

Example After

```
<VirtualHost *:80>
  ServerName openstack.example.com
  <IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteCond %{HTTPS} off
    RewriteRule (.*) https://%{HTTP_HOST}%{REQUEST_URI}
  </IfModule>
  <IfModule !mod_rewrite.c>
    RedirectPermanent / https://openstack.example.com
  </IfModule>
</VirtualHost>

<VirtualHost *:443>
  ServerName openstack.example.com

  SSLEngine On
  # Remember to replace certificates and keys with valid paths in your
  ↪environment
  SSLCertificateFile /etc/apache2/SSL/openstack.example.com.crt
  SSLCACertificateFile /etc/apache2/SSL/openstack.example.com.crt
  SSLCertificateKeyFile /etc/apache2/SSL/openstack.example.com.key
  SetEnvIf User-Agent ".*MSIE.*" nokeepalive ssl-unclean-shutdown

  # HTTP Strict Transport Security (HSTS) enforces that all communications
  # with a server go over SSL. This mitigates the threat from attacks such
  # as SSL-Strip which replaces links on the wire, stripping away https
  ↪prefixes
  # and potentially allowing an attacker to view confidential information
  ↪on the
  ↪wire
  Header add Strict-Transport-Security "max-age=15768000"

  WSGIScriptAlias / /usr/share/openstack-dashboard/openstack_dashboard/
  ↪wsgi.py
  WSGIDaemonProcess horizon user=www-data group=www-data processes=3
  ↪threads=10
  Alias /static /usr/share/openstack-dashboard/openstack_dashboard/static/
```

(continues on next page)

(continued from previous page)

```
<Location />
  Options None
  AllowOverride None
  # For Apache http server 2.4 and later:
  <ifVersion >=2.4>
    Require all granted
  </ifVersion>
  # For Apache http server 2.2 and earlier:
  <ifVersion <2.4>
    Order allow,deny
    Allow from all
  </ifVersion>
</Location>
</VirtualHost>
```

In this configuration, the Apache HTTP Server listens on port 443 and redirects all non-secure requests to the HTTPS protocol. The secured section defines the private key, public key, and certificate to use.

3. Restart the Apache HTTP Server.
4. Restart memcached.

If you try to access the Dashboard through HTTP, the browser redirects you to the HTTPS page.

Note: Configuring the Dashboard for HTTPS also requires enabling SSL for the noVNC proxy service. On the controller node, add the following additional options to the [DEFAULT] section of the `/etc/nova/nova.conf` file:

```
[DEFAULT]
# ...
ssl_only = true
cert = /etc/apache2/SSL/openstack.example.com.crt
key = /etc/apache2/SSL/openstack.example.com.key
```

On the compute nodes, ensure the `nonvncproxy_base_url` option points to a URL with an HTTPS scheme:

```
[DEFAULT]
# ...
nonvncproxy_base_url = https://controller:6080/vnc_auto.html
```

2.4.2 Set up session storage for the Dashboard

The Dashboard uses [Django sessions framework](#) to handle user session data. However, you can use any available session back end. You customize the session back end through the `SESSION_ENGINE` setting in your `local_settings.py` file.

After architecting and implementing the core OpenStack services and other required services, combined with the Dashboard service steps below, users and administrators can use the OpenStack dashboard. Refer to the *OpenStack User Documentation* chapter of the OpenStack End User Guide for further instructions on logging in to the Dashboard.

The following sections describe the pros and cons of each option as it pertains to deploying the Dashboard.

Local memory cache

Local memory storage is the quickest and easiest session back end to set up, as it has no external dependencies whatsoever. It has the following significant drawbacks:

- No shared storage across processes or workers.
- No persistence after a process terminates.

The local memory back end is enabled as the default for Horizon solely because it has no dependencies. It is not recommended for production use, or even for serious development work.

```
SESSION_ENGINE = 'django.contrib.sessions.backends.cache'
CACHES = {
    'default' : {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache'
    }
}
```

You can use applications such as Memcached or Redis for external caching. These applications offer persistence and shared storage and are useful for small-scale deployments and development.

Memcached

Memcached is a high-performance and distributed memory object caching system providing in-memory key-value store for small chunks of arbitrary data.

Requirements:

- Memcached service running and accessible.
- Python module `python-memcached` installed.

```
SESSION_ENGINE = 'django.contrib.sessions.backends.cache'
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': 'my_memcached_host:11211',
    }
}
```

Redis

Redis is an open source, BSD licensed, advanced key-value store. It is often referred to as a data structure server.

Requirements:

- Redis service running and accessible.
- Python modules `redis` and `django-redis` installed.

```
SESSION_ENGINE = 'django.contrib.sessions.backends.cache'
CACHES = {
    "default": {
        "BACKEND": "redis_cache.cache.RedisCache",
        "LOCATION": "127.0.0.1:6379:1",
        "OPTIONS": {
            "CLIENT_CLASS": "redis_cache.client.DefaultClient",
        }
    }
}
```

Initialize and configure the database

Database-backed sessions are scalable, persistent, and can be made high-concurrency and highly available.

However, database-backed sessions are one of the slower session storages and incur a high overhead under heavy usage. Proper configuration of your database deployment can also be a substantial undertaking and is far beyond the scope of this documentation.

1. Start the MySQL command-line client.

```
# mysql
```

2. Enter the MySQL root users password when prompted.
3. To configure the MySQL database, create the dash database.

```
mysql> CREATE DATABASE dash;
```

4. Create a MySQL user for the newly created dash database that has full control of the database. Replace `DASH_DBPASS` with a password for the new user.

```
mysql> GRANT ALL PRIVILEGES ON dash.* TO 'dash'@'%' IDENTIFIED BY 'DASH_
↵DBPASS';
mysql> GRANT ALL PRIVILEGES ON dash.* TO 'dash'@'localhost' IDENTIFIED BY
↵'DASH_DBPASS';
```

5. Enter `quit` at the `mysql>` prompt to exit MySQL.
6. In the `local_settings.py` file, change these options:

```
SESSION_ENGINE = 'django.contrib.sessions.backends.db'
DATABASES = {
    'default': {
        # Database configuration here
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'dash',
        'USER': 'dash',
        'PASSWORD': 'DASH_DBPASS',
        'HOST': 'localhost',
        'default-character-set': 'utf8'
    }
}
```

7. After configuring the `local_settings.py` file as shown, you can run the `manage.py migrate` command to populate this newly created database.

```
# /usr/share/openstack-dashboard/manage.py migrate
```

8. To avoid a warning when you restart Apache on Ubuntu, create a `blackhole` directory in the Dashboard directory, as follows.

```
# mkdir -p /var/lib/dash/.blackhole
```

9. Restart the Apache service.
10. On Ubuntu, restart the `nova-api` service to ensure that the API server can connect to the Dashboard without error.

```
# service nova-api restart
```

Cached database

To mitigate the performance issues of database queries, you can use the Django `cached_db` session back end, which utilizes both your database and caching infrastructure to perform write-through caching and efficient retrieval.

Enable this hybrid setting by configuring both your database and cache, as discussed previously. Then, set the following value:

```
SESSION_ENGINE = "django.contrib.sessions.backends.cached_db"
```

Cookies

If you use Django 1.4 or later, the `signed_cookies` back end avoids server load and scaling problems.

This back end stores session data in a cookie, which is stored by the users browser. The back end uses a cryptographic signing technique to ensure session data is not tampered with during transport. This is not the same as encryption; session data is still readable by an attacker.

The pros of this engine are that it requires no additional dependencies or infrastructure overhead, and it scales indefinitely as long as the quantity of session data being stored fits into a normal cookie.

The biggest downside is that it places session data into storage on the users machine and transports it over the wire. It also limits the quantity of session data that can be stored.

See the Django [cookie-based sessions](#) documentation.

2.4.3 Create and manage images

As an administrative user, you can create and manage images for the projects to which you belong. You can also create and manage images for users in all projects to which you have access.

To create and manage images in specified projects as an end user, see the [upload and manage images with Dashboard in OpenStack End User Guide](#) and [manage images with CLI in OpenStack End User Guide](#).

To create and manage images as an administrator for other users, use the following procedures.

Create images

For details about image creation, see the [Virtual Machine Image Guide](#).

1. Log in to the Dashboard and select the *admin* project from the drop-down list.
2. On the *Admin* tab, open the *Compute* tab and click the *Images* category. The images that you can administer for cloud users appear on this page.
3. Click *Create Image*, which opens the *Create An Image* window.
4. In the *Create An Image* window, enter or select the following values:

<i>Name</i>	Enter a name for the image.
<i>Description</i>	Enter a brief description of the image.
<i>Image Source</i>	Choose the image source from the dropdown list. Your choices are <i>Image Location</i> and <i>Image File</i> .
<i>Image File</i> or <i>Image Location</i>	Based on your selection, there is an <i>Image File</i> or <i>Image Location</i> field. You can include the location URL or browse for the image file on your file system and add it.
<i>Format</i>	Select the image format.
<i>Architecture</i>	Specify the architecture. For example, <i>i386</i> for a 32-bit architecture or <i>x86_64</i> for a 64-bit architecture.
<i>Minimum Disk (GB)</i>	Leave this field empty.
<i>Minimum RAM (MB)</i>	Leave this field empty.
<i>Copy Data</i>	Specify this option to copy image data to the Image service.
<i>Public</i>	Select this option to make the image public to all users.
<i>Protected</i>	Select this option to ensure that only users with permissions can delete it.

5. Click *Create Image*.

The image is queued to be uploaded. It might take several minutes before the status changes from *Queued* to *Active*.

Create An Image ×

Name *

Description

Image Source

Image Location ?

Format *

Architecture

Minimum Disk (GB) ?

Minimum RAM (MB) ?

Copy Data ?

Public

Protected

Description:

Specify an image to upload to the Image Service.

Currently only images available via an HTTP URL are supported. The image location must be accessible to the Image Service. Compressed image binaries are supported (.zip and .tar.gz.)

Please note: The Image Location field MUST be a valid and direct URL to the image binary. URLs that redirect or serve error pages will result in unusable images.

Cancel

Create Image

Fig. 6: FigureãDashboard Create Image

Update images

1. Log in to the Dashboard and select the *admin* project from the drop-down list.
2. On the *Admin* tab, open the *Compute* tab and click the *Images* category.
3. Select the images that you want to edit. Click *Edit Image*.
4. In the *Edit Image* window, you can change the image name.

Select the *Public* check box to make the image public. Clear this check box to make the image private. You cannot change the *Kernel ID*, *Ramdisk ID*, or *Architecture* attributes for an image.

5. Click *Edit Image*.

Delete images

1. Log in to the Dashboard and select the *admin* project from the drop-down list.
2. On the *Admin* tab, open the *Compute* tab and click the *Images* category.
3. Select the images that you want to delete.
4. Click *Delete Images*.
5. In the *Confirm Delete Images* window, click *Delete Images* to confirm the deletion.

You cannot undo this action.

2.4.4 Create and manage roles

A role is a personality that a user assumes to perform a specific set of operations. A role includes a set of rights and privileges. A user assumes that role inherits those rights and privileges.

Note: OpenStack Identity service defines a users role on a project, but it is completely up to the individual service to define what that role means. This is referred to as the services policy. To get details about what the privileges for each role are, refer to the `policy.json` file available for each service in the `/etc/SERVICE/policy.json` file. For example, the policy defined for OpenStack Identity service is defined in the `/etc/keystone/policy.json` file.

Create a role

1. Log in to the dashboard and select the *admin* project from the drop-down list.
2. On the *Identity* tab, click the *Roles* category.
3. Click the *Create Role* button.
In the *Create Role* window, enter a name for the role.
4. Click the *Create Role* button to confirm your changes.

Edit a role

1. Log in to the dashboard and select the *Identity* project from the drop-down list.
2. On the *Identity* tab, click the *Roles* category.
3. Click the *Edit* button.

In the *Update Role* window, enter a new name for the role.

4. Click the *Update Role* button to confirm your changes.

Note: Using the dashboard, you can edit only the name assigned to a role.

Delete a role

1. Log in to the dashboard and select the *Identity* project from the drop-down list.
2. On the *Identity* tab, click the *Roles* category.
3. Select the role you want to delete and click the *Delete Roles* button.
4. In the *Confirm Delete Roles* window, click *Delete Roles* to confirm the deletion.

You cannot undo this action.

2.4.5 Manage projects and users

OpenStack administrators can create projects, and create accounts for new users using the OpenStack Dashboard. Projects own specific resources in your OpenStack environment. You can associate users with roles, projects, or both.

Add a new project

1. Log into the OpenStack Dashboard as the Admin user.
2. Click on the *Identity* label on the left column, and click *Projects*.
3. Select the *Create Project* push button. The *Create Project* window will open.
4. Enter the Project name and description. Leave the *Domain ID* field set at *default*.
5. Click *Create Project*.

Note: Your new project will appear in the list of projects displayed under the *Projects* page of the dashboard. Projects are listed in alphabetical order, and you can check on the **Project ID**, **Domain name**, and status of the project in this section.

Delete a project

1. Log into the OpenStack Dashboard as the Admin user.
2. Click on the *Identity* label on the left column, and click *Projects*.
3. Select the checkbox to the left of the project you would like to delete.
4. Click on the *Delete Projects* push button.

Update a project

1. Log into the OpenStack Dashboard as the Admin user.
2. Click on the *Identity* label on the left column, and click *Projects*.
3. Locate the project you wish to update, and under the *Actions* column click on the drop down arrow next to the *Manage Members* push button. The *Update Project* window will open.
4. Update the name of the project, enable the project, or disable the project as needed.

Add a new user

1. Log into the OpenStack Dashboard as the Admin user.
2. Click on the *Identity* label on the left column, and click *Users*.
3. Click *Create User*.
4. Enter a *Domain Name*, the *Username*, and a *password* for the new user. Enter an email for the new user, and specify which *Primary Project* they belong to. Leave the *Domain ID* field set at *default*. You can also enter a description for the new user.
5. Click the *Create User* push button.

Note: The new user will then appear in the list of projects displayed under the *Users* page of the dashboard. You can check on the **User Name**, **User ID**, **Domain name**, and the User status in this section.

Delete a new user

1. Log into the OpenStack Dashboard as the Admin user.
2. Click on the *Identity* label on the left column, and click *Users*.
3. Select the checkbox to the left of the user you would like to delete.
4. Click on the *Delete Users* push button.

Update a user

1. Log into the OpenStack Dashboard as the Admin user.
2. Click on the *Identity* label on the left column, and click *Users*.
3. Locate the User you would like to update, and select the *Edit* push button under the *Actions* column.
4. Adjust the *Domain Name*, *User Name*, *Description*, *Email*, and *Primary Project*.

Enable or disable a user

1. Log into the OpenStack Dashboard as the Admin user.
2. Click on the *Identity* label on the left column, and click *Users*.
3. Locate the User you would like to update, and select the arrow to the right of the *Edit* push button. This will open a drop down menu.
4. Select *Disable User*.

Note: To reactivate a disabled user, select *Enable User* under the drop down menu.

2.4.6 Manage instances

As an administrative user, you can manage instances for users in various projects. You can view, terminate, edit, perform a soft or hard reboot, create a snapshot from, and migrate instances. You can also view the logs for instances or launch a VNC console for an instance.

For information about using the Dashboard to launch instances as an end user, see the [OpenStack End User Guide](#).

Create instance snapshots

1. Log in to the Dashboard and select the *admin* project from the drop-down list.
2. On the *Admin* tab, open the *Compute* tab and click the *Instances* category.
3. Select an instance to create a snapshot from it. From the *Actions* drop-down list, select *Create Snapshot*.
4. In the *Create Snapshot* window, enter a name for the snapshot.
5. Click *Create Snapshot*. The Dashboard shows the instance snapshot in the *Images* category.
6. To launch an instance from the snapshot, select the snapshot and click *Launch*. For information about launching instances, see the [OpenStack End User Guide](#).

Control the state of an instance

1. Log in to the Dashboard and select the *admin* project from the drop-down list.
2. On the *Admin* tab, open the *Compute* tab and click the *Instances* category.
3. Select the instance for which you want to change the state.
4. From the drop-down list in the Actions column, select the state.

Depending on the current state of the instance, you can perform various actions on the instance. For example, pause, un-pause, suspend, resume, soft or hard reboot, or terminate (actions in red are dangerous).

Project	Host	Name	Image Name	IP Address	Size	Status	Task	Power State	Time since created	Actions
admin	chenqiaomin-m	vm1	cirros	10.2.1.8 Floating IPs: 172.24.4.227	m1.tiny	Active	None	Running	3 days, 11 hours	Edit Instance Console View Log Create Snapshot Pause Instance Suspend Instance Shelve Instance Migrate Instance Live Migrate Instance Soft Reboot Instance Hard Reboot Instance Delete Instance

Displaying 1 item

Fig. 7: Dashboard Instance Actions

Track usage

Use the *Overview* category to track usage of instances for each project.

You can track costs per month by showing meters like number of VCPUs, disks, RAM, and uptime of all your instances.

1. Log in to the Dashboard and select the *admin* project from the drop-down list.
2. On the *Admin* tab, click the *Overview* category.
3. Select a month and click *Submit* to query the instance usage for that month.
4. Click *Download CSV Summary* to download a CSV summary.

2.4.7 Manage flavors

In OpenStack, a flavor defines the compute, memory, and storage capacity of a virtual server, also known as an instance. As an administrative user, you can create, edit, and delete flavors.

As of Newton, there are no default flavors. The following table lists the default flavors for Mitaka and earlier.

Flavor	VCPUs	Disk (in GB)	RAM (in MB)
m1.tiny	1	1	512
m1.small	1	20	2048
m1.medium	2	40	4096
m1.large	4	80	8192
m1.xlarge	8	160	16384

Create flavors

1. Log in to the Dashboard and select the *admin* project from the drop-down list.
2. In the *Admin* tab, open the *Compute* tab and click the *Flavors* category.
3. Click *Create Flavor*.
4. In the *Create Flavor* window, enter or select the parameters for the flavor in the *Flavor Information* tab.

Name	Enter the flavor name.
ID	Unique ID (integer or UUID) for the new flavor. If specifying auto, a UUID will be automatically generated.
VC-PU s	Enter the number of virtual CPUs to use.
RAM (MB)	Enter the amount of RAM to use, in megabytes.
Root Disk (GB)	Enter the amount of disk space in gigabytes to use for the root (/) partition.
Ephemeral Disk (GB)	Enter the amount of disk space in gigabytes to use for the ephemeral partition. If unspecified, the value is 0 by default. Ephemeral disks offer machine local disk storage linked to the lifecycle of a VM instance. When a VM is terminated, all data on the ephemeral disk is lost. Ephemeral disks are not included in any snapshots.
Swap Disk (MB)	Enter the amount of swap space (in megabytes) to use. If unspecified, the default is 0.
RX/TX Factor	Optional property allows servers with a different bandwidth to be created with the RX/TX Factor. The default value is 1. That is, the new bandwidth is the same as that of the attached network.

5. In the *Flavor Access* tab, you can control access to the flavor by moving projects from the *All Projects* column to the *Selected Projects* column.

Only projects in the *Selected Projects* column can use the flavor. If there are no projects in the right column, all projects can use the flavor.

6. Click *Create Flavor*.

Create Flavor ✕

Flavor Information *Flavor Access

Name *

ID ⓘ

VCPUs *

RAM (MB) *

Root Disk (GB) *

Ephemeral Disk (GB)

Swap Disk (MB)

RX/TX Factor

Flavors define the sizes for RAM, disk, number of cores, and other resources and can be selected when users deploy instances.

Fig. 8: Dashboard Create Flavor

Update flavors

1. Log in to the Dashboard and select the *admin* project from the drop-down list.
2. In the *Admin* tab, open the *Compute* tab and click the *Flavors* category.
3. Select the flavor that you want to edit. Click *Edit Flavor*.
4. In the *Edit Flavor* window, you can change the flavor name, VCPUs, RAM, root disk, ephemeral disk, and swap disk values.
5. Click *Save*.

Update Metadata

1. Log in to the Dashboard and select the *admin* project from the drop-down list.
2. In the *Admin* tab, open the *Compute* tab and click the *Flavors* category.
3. Select the flavor that you want to update. In the drop-down list, click *Update Metadata* or click *No* or *Yes* in the *Metadata* column.
4. In the *Update Flavor Metadata* window, you can customize some metadata keys, then add it to this flavor and set them values.
5. Click *Save*.

Optional metadata keys

CPU limits	quota:cpu_shares
	quota:cpu_period
	quota:cpu_limit
	quota:cpu_reservation
	quota:cpu_quota
Disk tuning	quota:disk_read_bytes_sec
	quota:disk_read_iops_sec
	quota:disk_write_bytes_sec
	quota:disk_write_iops_sec
	quota:disk_total_bytes_sec
	quota:disk_total_iops_sec
Bandwidth I/O	quota:vif_inbound_average
	quota:vif_inbound_burst
	quota:vif_inbound_peak
	quota:vif_outbound_average
	quota:vif_outbound_peak
Watchdog behavior	hw:watchdog_action
Random-number generator	hw_rng:allowed
	hw_rng:rate_bytes
	hw_rng:rate_period

For information about supporting metadata keys, see the the Compute service documentation.

Delete flavors

1. Log in to the Dashboard and select the *admin* project from the drop-down list.
2. In the *Admin* tab, open the *Compute* tab and click the *Flavors* category.
3. Select the flavors that you want to delete.
4. Click *Delete Flavors*.
5. In the *Confirm Delete Flavors* window, click *Delete Flavors* to confirm the deletion. You cannot undo this action.

2.4.8 Manage volumes and volume types

Volumes are the Block Storage devices that you attach to instances to enable persistent storage. Users can attach a volume to a running instance or detach a volume and attach it to another instance at any time. For information about using the dashboard to create and manage volumes as an end user, see the *OpenStack End User Guide*.

As an administrative user, you can manage volumes and volume types for users in various projects. You can create and delete volume types, and you can view and delete volumes. Note that a volume can be encrypted by using the steps outlined below.

Create a volume type

1. Log in to the dashboard and select the *admin* project from the drop-down list.
2. On the *Admin* tab, open the *Volume* tab.
3. Click the *Volume Types* tab, and click *Create Volume Type* button. In the *Create Volume Type* window, enter a name for the volume type.
4. Click *Create Volume Type* button to confirm your changes.

Note: A message indicates whether the action succeeded.

Create an encrypted volume type

1. Create a volume type using the steps above for *Create a volume type*.
2. Click *Create Encryption* in the Actions column of the newly created volume type.
3. Configure the encrypted volume by setting the parameters below from available options (see table):
 - Provider** Specifies the encryption provider format.
 - Control Location** Specifies whether the encryption is from the front end (nova) or the back end (cinder).
 - Cipher** Specifies the encryption algorithm.
 - Key Size (bits)** Specifies the encryption key size.
4. Click *Create Volume Type Encryption*.

Create an Encrypted Volume Type ✕

Name	<input type="text" value="test"/>	Description: Creating encryption for a volume type causes all volumes with that volume type to be encrypted. Encryption information cannot be added to a volume type if volumes are currently in use with that volume type. The Provider is the encryption provider format (e.g. 'luks' or 'plain'). The Control Location is the notional service where encryption is performed (e.g., front-end=Nova). The default value is 'front-end.' The Cipher is the encryption algorithm/mode to use (e.g., aes-xts-plain64). If the field is left empty, the provider default will be used. The Key Size is the size of the encryption key, in bits (e.g., 256). If the field is left empty, the provider default will be used.
Provider *	<input type="text"/>	
Control Location *	<input type="text" value="front-end"/>	
Cipher	<input type="text"/>	
Key Size (bits)	<input type="text"/>	

Fig. 9: Encryption Options

The table below provides a few alternatives available for creating encrypted volumes.

En-cryption pa-rameters	Parameter options	Comments
Provider	luks (Recommended)	Allows easier import and migration of imported encrypted volumes, and allows access key to be changed without re-encrypting the volume
	plain	Less disk overhead than LUKS
Control Loca-tion	front-end (Recommended)	The encryption occurs within nova so that the data transmitted over the network is encrypted
	back-end	This could be selected if a cinder plug-in supporting an encrypted back-end block storage device becomes available in the future. TLS or other network encryption would also be needed to protect data as it traverses the network
Cipher	aes-xts-plain64 (Recommended)	See NIST reference below to see advantages*
	aes-cbc-essiv	Note: On the command line, type <code>cryptsetup benchmark</code> for additional options
Key Size (bits)	256 (Recommended for aes-xts-plain64 and aes-cbc-essiv)	Using this selection for aes-xts, the underlying key size would only be 128-bits*

* Source [NIST SP 800-38E](#)

Note: To see further information and CLI instructions, see [Create an encrypted volume type](#) in the OpenStack Block Storage Configuration Guide.

Delete volume types

When you delete a volume type, volumes of that type are not deleted.

1. Log in to the dashboard and select the *admin* project from the drop-down list.
2. On the *Admin* tab, open the *Volume* tab.
3. Click the *Volume Types* tab, select the volume type or types that you want to delete.
4. Click *Delete Volume Types* button.
5. In the *Confirm Delete Volume Types* window, click the *Delete Volume Types* button to confirm the action.

Note: A message indicates whether the action succeeded.

Delete volumes

When you delete an instance, the data of its attached volumes is not destroyed.

1. Log in to the dashboard and select the *admin* project from the drop-down list.
2. On the *Admin* tab, open the *Volume* tab.
3. Click the *Volumes* tab, Select the volume or volumes that you want to delete.
4. Click *Delete Volumes* button.
5. In the *Confirm Delete Volumes* window, click the *Delete Volumes* button to confirm the action.

Note: A message indicates whether the action succeeded.

2.4.9 View and manage quotas

To prevent system capacities from being exhausted without notification, you can set up quotas. Quotas are operational limits. For example, the number of gigabytes allowed for each project can be controlled so that cloud resources are optimized. Quotas can be enforced at both the project and the project-user level.

Typically, you change quotas when a project needs more than ten volumes or 1TB on a compute node.

Using the Dashboard, you can view default Compute and Block Storage quotas for new projects, as well as update quotas for existing projects.

Note: Using the command-line interface, you can manage quotas for the [OpenStack Compute service](#), the [OpenStack Block Storage service](#), and the OpenStack Networking service (For CLI details, see [OpenStackClient CLI reference](#)). Additionally, you can update Compute service quotas for project users.

The following table describes the Compute and Block Storage service quotas:

Quota Descriptions

Quota Name	Defines the number of	Service
Gigabytes	Volume gigabytes allowed for each project.	Block Storage
Instances	Instances allowed for each project.	Compute
Injected Files	Injected files allowed for each project.	Compute
Injected File Content Bytes	Content bytes allowed for each injected file.	Compute
Keypairs	Number of keypairs.	Compute
Metadata Items	Metadata items allowed for each instance.	Compute
RAM (MB)	RAM megabytes allowed for each instance.	Compute
Security Groups	Security groups allowed for each project.	Compute
Security Group Rules	Security group rules allowed for each project.	Compute
Snapshots	Volume snapshots allowed for each project.	Block Storage
VCPUs	Instance cores allowed for each project.	Compute
Volumes	Volumes allowed for each project.	Block Storage

View default project quotas

1. Log in to the dashboard and select the *admin* project from the drop-down list.
2. On the *Admin* tab, open the *System* tab and click the *Defaults* category.
3. The default quota values are displayed.

Note: You can sort the table by clicking on either the *Quota Name* or *Limit* column headers.

Update project quotas

1. Log in to the dashboard and select the *admin* project from the drop-down list.
2. On the *Admin* tab, open the *System* tab and click the *Defaults* category.
3. Click the *Update Defaults* button.
4. In the *Update Default Quotas* window, you can edit the default quota values.
5. Click the *Update Defaults* button.

Note: The dashboard does not show all possible project quotas. To view and update the quotas for a service, use its command-line client. See [OpenStack Administrator Guide](#).

2.4.10 View services information

As an administrative user, you can view information for OpenStack services.

1. Log in to the Dashboard and select the *admin* project from the drop-down list.
2. On the *Admin* tab, open the *System* tab and click the *System Information* category.

View the following information on these tabs:

- *Services*: Displays the internal name and the public OpenStack name for each service, the host on which the service runs, and whether or not the service is enabled.
- *Compute Services*: Displays information specific to the Compute service. Both host and zone are listed for each service, as well as its activation status.
- *Block Storage Services*: Displays information specific to the Block Storage service. Both host and zone are listed for each service, as well as its activation status.
- *Network Agents*: Displays the network agents active within the cluster, such as L3 and DHCP agents, and the status of each agent.

2.4.11 Create and manage host aggregates

Host aggregates enable administrative users to assign key-value pairs to groups of machines.

Each node can have multiple aggregates and each aggregate can have multiple key-value pairs. You can assign the same key-value pair to multiple aggregates.

The scheduler uses this information to make scheduling decisions. For information, see [Scheduling](#).

To create a host aggregate

1. Log in to the Dashboard and select the *admin* project from the drop-down list.
2. On the *Admin* tab, open the *Compute* tab and click the *Host Aggregates* category.
3. Click *Create Host Aggregate*.
4. In the *Create Host Aggregate* dialog box, enter or select the following values on the *Host Aggregate Information* tab:
 - *Name*: The host aggregate name.
 - *Availability Zone*: The cloud provider defines the default availability zone, such as *us-west*, *apac-south*, or *nova*. You can target the host aggregate, as follows:
 - When the host aggregate is exposed as an availability zone, select the availability zone when you launch an instance.
 - When the host aggregate is not exposed as an availability zone, select a flavor and its extra specs to target the host aggregate.
5. Assign hosts to the aggregate using the *Manage Hosts within Aggregate* tab in the same dialog box.

To assign a host to the aggregate, click + for the host. The host moves from the *All available hosts* list to the *Selected hosts* list.

You can add one host to one or more aggregates. To add a host to an existing aggregate, edit the aggregate.

To manage host aggregates

1. Select the *admin* project from the drop-down list at the top of the page.
2. On the *Admin* tab, open the *Compute* tab and click the *Host Aggregates* category.
 - To edit host aggregates, select the host aggregate that you want to edit. Click *Edit Host Aggregate*.

In the *Edit Host Aggregate* dialog box, you can change the name and availability zone for the aggregate.
 - To manage hosts, locate the host aggregate that you want to edit in the table. Click *More* and select *Manage Hosts*.

In the *Add/Remove Hosts to Aggregate* dialog box, click + to assign a host to an aggregate. Click - to remove a host that is assigned to an aggregate.
 - To delete host aggregates, locate the host aggregate that you want to edit in the table. Click *More* and select *Delete Host Aggregate*.

- To deploy the dashboard, see the *Installation Guide*.
- To launch instances with the dashboard as an end user, see the *Launch and manage instances* in the OpenStack End User Guide.
- To create and manage ports, see the *Create and manage networks* section of the OpenStack End User Guide.

CONTRIBUTOR DOCS

For those wishing to develop Horizon itself, or go in-depth with building your own *Dashboard* or *Panel* classes, the following documentation is provided.

3.1 Contributor Documentation

3.1.1 So You Want to Contribute

For general information on contributing to OpenStack, please check out the [contributor guide](#) to get started. It covers all the basics that are common to all OpenStack projects: the accounts you need, the basics of interacting with our Gerrit review system, how we communicate as a community, etc.

Below will cover the more project specific information you need to get started with horizon.

Project Resources

- Source code: <https://opendev.org/openstack/horizon>
- Documentation: <https://docs.openstack.org/horizon/latest/>
- Project page: <https://launchpad.net/horizon>
- Bug tracker: <https://bugs.launchpad.net/horizon>
- Code review: <https://review.opendev.org/#/q/project:openstack/horizon+status:open>

Communication

- IRC channel: `#openstack-horizon` at OFTC

Most active contributors are online at IRC while they are active, so it would be the easiest way to contact the team directly. Note that all IRC conversations are stored [here](#).

- Mailing list: [openstack-discuss](#) with `[horizon]` tag.

The mailing list would be a good place if you would like to discuss your topic with the OpenStack community more broadly. Most OpenStack users, operators and developers subscribe it and you can get useful feedbacks from various perspectives.

- Team meeting:

The horizon team has a weekly meeting which covers all things related to the horizon project like announcements, project priorities, community goals, bugs and so on.

There is the On Demand Agenda section at the end of the meeting, where anyone can add a topic to discuss with the team. It is suggested to add such topic to the On-Demand agenda in the Weekly meeting in [horizon release priority etherpad](#).

- Time: http://eavesdrop.openstack.org/#Horizon_Team_Meeting
- Agenda: <https://wiki.openstack.org/wiki/Meetings/Horizon>

Contacting the Core Team

The list of the current core reviewers is found at [gerrit](#).

Most core reviewers are online in the IRC channel and you can contact them there.

New Feature Planning

If you would like to add a new feature to horizon, file a blueprint to <https://blueprints.launchpad.net/horizon>. You can find a template for a blueprint at <https://blueprints.launchpad.net/horizon/+spec/template>. The template is not a strict requirement but it would be nice to cover a motivation and an approach of your blueprint. From the nature of GUI, a discussion on UI design during a patch review could be more productive, so there is no need to explain the detail of UI design in your blueprint proposal.

We don't have a specific deadline during a development cycle. You can propose a feature any time. Only thing you keep in your mind is that we do not merge features during the feature freeze period after the milestone 3 in each cycle.

There are a number of unsupported OpenStack features in horizon. Implementing such would be appreciated even if it is small.

Task Tracking

We track our tasks in [Launchpad](#).

If you're looking for some smaller, please look through the list of bugs and find what you think you can work on. If you are not sure the status of a bug feel free to ask to the horizon team. We can help you. Note that we recently do not maintain low-hanging-fruit tag and some of them with this tag are not simple enough.

Reporting a Bug

You found an issue and want to make sure we are aware of it? You can do so on [Launchpad](#).

Please file a bug first even if you already have a fix for it. If you can reproduce the bug reliably and identify its cause then it's usually safe to start working on it. However, getting independent confirmation (and verifying that it's not a duplicate) is always a good idea if you can be patient.

Getting Your Patch Merged

All changes proposed to horizon require two +2 votes from the horizon core reviewers before one of the core reviewers can approve a change by giving Workflow +1 vote.

In general, all changes should be proposed along with at least unit test coverage (python or JavaScript). Integration test support would be appreciated.

More detailed guidelines for reviewers of patches are available at [OpenDev Developers Guide](#).

Project Team Lead Duties

All common PTL duties are enumerated in the [PTL guide](#).

The horizon PTL is expected to coordinate and encourage the core reviewer team and contributors for the success. The expectations for the core reviewer team is documented at [Core Reviewer Team](#) and the PTL would play an important role in this.

Etiquette

The community's guidelines for etiquette are fairly simple:

- Treat everyone respectfully and professionally.
- If a bug is in progress in the bug tracker, don't start working on it without contacting the author. Try on IRC, or via the launchpad email contact link. If you don't get a response after a reasonable time, then go ahead. Checking first avoids duplicate work and makes sure nobody's toes get stepped on.
- If a blueprint is assigned, even if it hasn't been started, be sure you contact the assignee before taking it on. These larger issues often have a history of discussion or specific implementation details that the assignee may be aware of that you are not.
- Please don't re-open tickets closed by a core developer. If you disagree with the decision on the ticket, the appropriate solution is to take it up on IRC or the mailing list.
- Give credit where credit is due; if someone helps you substantially with a piece of code, it's polite (though not required) to thank them in your commit message.

3.1.2 Horizon Basics

Values

Think simple as my old master used to say - meaning reduce the whole of its parts into the simplest terms, getting back to first principles.

—Frank Lloyd Wright

Horizon holds several key values at the core of its design and architecture:

- Core Support: Out-of-the-box support for all core OpenStack projects.
- Extensible: Anyone can add a new component as a first-class citizen.
- Manageable: The core codebase should be simple and easy-to-navigate.
- Consistent: Visual and interaction paradigms are maintained throughout.

- **Stable:** A reliable API with an emphasis on backwards-compatibility.
- **Usable:** Providing an *awesome* interface that people *want* to use.

The only way to attain and uphold those ideals is to make it *easy* for developers to implement those values.

History

Horizon started life as a single app to manage OpenStacks compute project. As such, all it needed was a set of views, templates, and API calls.

From there it grew to support multiple OpenStack projects and APIs gradually, arranged rigidly into dash and syspanel groupings.

During the Diablo release cycle an initial plugin system was added using signals to hook in additional URL patterns and add links into the dash and syspanel navigation.

This incremental growth served the goal of Core Support phenomenally, but left Extensible and Manageable behind. And while the other key values took shape of their own accord, it was time to re-architect for an extensible, modular future.

The Current Architecture & How It Meets Our Values

At its core, **Horizon should be a registration pattern for applications to hook into.** Heres what that means and how it is implemented in terms of our values:

Core Support

Horizon ships with three central dashboards, a User Dashboard, a System Dashboard, and a Settings dashboard. Between these three they cover the core OpenStack applications and deliver on Core Support.

The Horizon application also ships with a set of API abstractions for the core OpenStack projects in order to provide a consistent, stable set of reusable methods for developers. Using these abstractions, developers working on Horizon dont need to be intimately familiar with the APIs of each OpenStack project.

Extensible

A Horizon dashboard application is based around the *Dashboard* class that provides a consistent API and set of capabilities for both core OpenStack dashboard apps shipped with Horizon and equally for third-party apps. The *Dashboard* class is treated as a top-level navigation item.

Should a developer wish to provide functionality within an existing dashboard (e.g. adding a monitoring panel to the user dashboard) the simple registration pattern makes it possible to write an app which hooks into other dashboards just as easily as creating a new dashboard. All you have to do is import the dashboard you wish to modify.

Manageable

Within the application, there is a simple method for registering a *Panel* (sub-navigation items). Each panel contains the necessary logic (views, forms, tests, etc.) for that interface. This granular breakdown prevents files (such as `api.py`) from becoming thousands of lines long and makes code easy to find by correlating it directly to the navigation.

Consistent

By providing the necessary core classes to build from, as well as a solid set of reusable templates and additional tools (base form classes, base widget classes, template tags, and perhaps even class-based views) we can maintain consistency across applications.

Stable

By architecting around these core classes and reusable components we create an implicit contract that changes to these components will be made in the most backwards-compatible ways whenever possible.

Usable

Ultimately that's up to each and every developer that touches the code, but if we get all the other goals out of the way then we are free to focus on the best possible experience.

See also:

- *Quickstart* A short guide to getting started with using Horizon.
- *Frequently Asked Questions* Common questions and answers.
- *Glossary* Common terms and their definitions.

3.1.3 Project Policies

This page collects basic policies on horizon development.

Supported Software

Back-end service support

- N release of horizon supports N and N-1 releases of back-end OpenStack services (like nova, cinder, neutron and so on). This allows operators to upgrade horizon separately from other OpenStack services.
- Horizon should check features in back-end services through APIs as much as possible by using micro-versioning for nova, cinder and so on and API extensions for neutron (and others if any).
- Related to the previous item, features available in N-4 releases (which means the recent four releases including the development version) are assumed without checking the availability of features to simplify the implementation.

- Removals and deprecations of back-end feature supports basically follows [the standard deprecation policy](#) defined by the technical committee, but there are some notes. Deprecations in back-end services are applied to corresponding horizon features automatically and it is allowed to drop some feature from horizon without an explicit deprecation.

Django support

- Horizon usually syncs with [Djangos Roadmap](#) and supports LTS (long term support) versions of Django as of the feature freeze of each OpenStack release. Supports for other maintained Django versions are optional and best-effort.

Horizon Teams

Horizon project defines several teams to maintain the project.

Gerrit

Core Reviewer Team

The core reviewer team is responsible for the horizon development in the master branch from various perspective. See [Core Reviewer Team](#) for more detail.

Stable Maintenance Team

Members of this gerrit group are responsible for maintaining stable branches. The members are expected to understand [the stable branch policy](#). Most members overlaps with the core reviewer team but being a core reviewer is not a requirement for being a member of the stable maintenance team. Folks who would like to be a member of this team is recommended to express how they understand the stable branch policy in reviews.

The member list is found at <https://review.opendev.org/#/admin/groups/537,members>.

Launchpad

Bug Supervisor Team

Members of this launchpad group are responsible for bug management. They have privileges to set status, priority and milestone of bugs. Most members overlaps with the core reviewer team but it is not a requirement for being a member of this team.

The member list is found at <https://launchpad.net/~horizon-bugs>.

Horizon Drivers Team

Members of this launchpad group can do all things in the horizon launchpad such as defining series and milestones. This group is included to the bug supervisor team.

The member list is found at <https://launchpad.net/~horizon-drivers>.

Security Contact Team

Members of this launchpad group are responsible for security issues. Members are expected to be familiar with [Vulnerability Management Process](#) in OpenStack. Private security issues are handled differently from usual public reports. All steps including patch development and review are done in a launchpad bug report.

The member list is found at <https://launchpad.net/~horizon-coresec>.

Note that the access permission to private information of this team is configured at <https://launchpad.net/horizon/+sharing>. (You can find Sharing menu at the top-right corner of the [launchpad top page](#).)

Core Reviewer Team

The horizon core reviewer team is responsible for many aspects of the horizon project. These include, but are not limited to:

- Mentor community contributors in solution design, testing, and the review process
- Actively reviewing patch submissions, considering whether the patch: - is functional - fits use cases and vision of the project - is complete in terms of testing, documentation, and release notes - takes into consideration upgrade concerns from previous versions
- Assist in bug triage and delivery of bug fixes
- Curating the gate and triaging failures
- Maintaining accurate, complete, and relevant documentation
- Ensuring the level of testing is adequate and remains relevant as features are added
- Answering questions and participating in mailing list discussions
- Interfacing with other OpenStack teams
- Helping horizon plugin maintenances

In essence, core reviewers share the following common ideals:

- They share responsibility in the projects success in its mission.
- They value a healthy, vibrant, and active developer and user community.
- They have made a long-term, recurring time investment to improve the project.
- They spend their time doing what needs to be done to ensure the projects success, not necessarily what is the most interesting or fun.
- A core reviewers responsibility doesnt end with merging code.

Core Reviewer Expectations

Members of the core reviewer team are expected to:

- Attend and participate in the weekly IRC meetings (if your timezone allows)
- Monitor and participate in-channel at #openstack-horizon
- Monitor and participate in [horizon] discussions on the mailing list
- Participate in related design sessions at Project Team Gatherings (PTGs)
- Review patch submissions actively and consistently

Please note in-person attendance at PTGs, mid-cycles, and other code sprints is not a requirement to be a core reviewer. Participation can also include contributing to the design documents discussed at the design sessions.

Active and consistent review of review activity, bug triage and other activity will be performed periodically and fed back to the core reviewer team so everyone knows how things are progressing.

Code Merge Responsibilities

While everyone is encouraged to review changes, members of the core reviewer team have the ability to +2/-2 and +A changes to these repositories. This is an extra level of responsibility not to be taken lightly. Correctly merging code requires not only understanding the code itself, but also how the code affects things like documentation, testing, upgrade impacts and interactions with other projects. It also means you pay attention to release milestones and understand if a patch you are merging is marked for the release, especially critical during the feature freeze.

Horizon Plugin Maintenance

GUI supports for most OpenStack projects are achieved via horizon plugins. The horizon core reviewer team has responsibility to help horizon plugin teams from the perspective of horizon related changes as the framework, for example, Django version bump, testing improvements, plugin interface changes in horizon and so on. A member of the team is expected to send and review patches related to such changes.

Note that involvements in more works in horizon plugins are up to individuals but it would be nice to be involved if you have time :)

Horizon Bugs

Horizon project maintains all bugs in [Launchpad horizon](#).

Bug Tags

Tags are used to classify bugs in a meaningful way. Popular tags are found at [OpenStack Wiki](#) (See [Horizon](#) and [All projects](#) sections).

Triaging Bugs

One of the important things in the bug management process is to triage incoming bugs appropriately. To keep you up-to-date to incoming bugs, see [Receiving incoming bugs](#).

The bug triaging process would be:

- Check if a bug is filed for a correct project. Otherwise, change the project or mark it as Invalid
- Check if enough information like below is provided:
 - High level description
 - Step-by-step instruction to reproduce the bug
 - Expected output and actual output
 - Version(s) of related components (at least horizon version is required)
- Check if a similar bug was reported before. If found, mark it as duplicate (using Mark as duplicate button in the right-top menu).
- Add or update proper bug tags
- Verify if a bug can be reproduced.
 - If the bug cannot be reproduced, there would be some pre-conditions. It is recommended to request more information from the bug reporter. Setting the status to Incomplete usually makes sense in this case.
- Assign the importance. If it breaks horizon basic functionality, the importance should be marked as Critical or High.

Receiving incoming bugs

To check incoming bugs, you can check Launchpad bug page directly, but at the moment the easiest way is to subscribe Launchpad bug mails. The steps to subscribe to the Launchpad bugs are as follows:

- Go to the [horizon bugs page](#).
- On the right hand side, click on Subscribe to bug mail.
- In the pop-up that is displayed, keep the recipient as Yourself, and set the subscription name to something useful like horizon-bugs. You can choose either option for how much mail you get, but keep in mind that getting mail for all changes - while informative - will result in more emails.

You will now receive bug mails from Launchpad when a new bug is filed. Note that you can classify emails based on the subscription name above.

3.1.4 Quickstart

Note: This section has been tested for Horizon on Ubuntu (18.04-amd64) and RPM-based (RHEL 8.x) distributions. Feel free to add notes and any changes according to your experiences or operating system.

Linux Systems

Install the prerequisite packages.

On Ubuntu

```
$ sudo apt install git python3-dev python3-pip gettext
```

On RPM-based distributions (e.g., Fedora/RHEL/CentOS)

```
$ sudo yum install gcc git-core python3-devel python3-virtualenv gettext
```

Note: Some tests rely on the Chrome web browser being installed. While the above requirements will allow you to run and manually test Horizon, you will need to install Chrome to run the full test suite.

Setup

To begin setting up a Horizon development environment simply clone the Horizon git repository from <https://opendev.org/openstack/horizon>

```
$ git clone https://opendev.org/openstack/horizon
```

Next you will need to configure Horizon by adding a `local_settings.py` file. A good starting point is to use the example config with the following command, from within the `horizon` directory.

```
$ cp openstack_dashboard/local/local_settings.py.example openstack_dashboard/  
↪ local/local_settings.py
```

Horizon connects to the rest of OpenStack via a Keystone service catalog. By default Horizon looks for an endpoint at `http://localhost/identity/v3`; this can be customised by modifying the `OPENSTACK_HOST` and `OPENSTACK_KEYSTONE_URL` values in `openstack_dashboard/local/local_settings.py`

Note: The DevStack project (<http://devstack.org/>) can be used to install an OpenStack development environment from scratch. For a `local.conf` that enables most services that Horizon supports managing, see *DevStack for Horizon*

Horizon uses `tox` to manage virtual environments for testing and other development tasks. You can install it with

```
$ pip3 install tox
```

The `tox` environments provide wrappers around `manage.py`. For more information on `manage.py`, which is a Django command, see <https://docs.djangoproject.com/en/dev/ref/django-admin/>

To start the Horizon development server use the command below

```
$ tox -e runserver
```

Note: The default port for `runserver` is 8000 which might be already consumed by `heat-api-cfn` in `DevStack`. If running in `DevStack` `tox -e runserver -- localhost:9000` will start the test server at `http://localhost:9000`. If you use `tox -e runserver` for developments, then configure `SESSION_ENGINE` to `django.contrib.sessions.backends.signed_cookies` in `openstack_dashboard/local/local_settings.py` file.

Once the Horizon server is running, point a web browser to `http://localhost` or to the IP and port the server is listening for. Enter your Keystone credentials, log in and you'll be presented with the Horizon dashboard. Congratulations!

Managing Settings

You can save changes you made to `openstack_dashboard/local/local_settings.py` with the following command:

```
$ python manage.py migrate_settings --gendiff
```

Note: This creates a `local_settings.diff` file which is a diff between `local_settings.py` and `local_settings.py.example`

If you upgrade Horizon, you might need to update your `openstack_dashboard/local/local_settings.py` file with new parameters from `openstack_dashboard/local/local_settings.py.example` to do so, first update Horizon

```
$ git remote update && git pull --ff-only origin master
```

Then update your `openstack_dashboard/local/local_settings.py` file

```
$ mv openstack_dashboard/local/local_settings.py openstack_dashboard/local/  
→local_settings.py.old  
$ python manage.py migrate_settings
```

Note: This applies `openstack_dashboard/local/local_settings.diff` on `openstack_dashboard/local/local_settings.py.example` to regenerate an `openstack_dashboard/local/local_settings.py` file. The migration can sometimes have difficulties to migrate some settings, if this happens you will be warned with a conflict message pointing to an `openstack_dashboard/local/local_settings.py_Some_DateTime.rej` file. In this file, you will see the lines which could not be automatically changed and you will have to redo only these few

changes manually instead of modifying the full `openstack_dashboard/local/local_settings.py.example` file.

When all settings have been migrated, it is safe to regenerate a clean diff in order to prevent Conflicts for future migrations

```
$ mv openstack_dashboard/local/local_settings.diff openstack_dashboard/local/  
↪local_settings.diff.old  
$ python manage.py migrate_settings --gendiff
```

Editing Horizons Source

Although DevStack installs and configures an instance of Horizon when running `stack.sh`, the preferred development setup follows the instructions above on the server/VM running DevStack. There are several advantages to maintaining a separate copy of the Horizon repo, rather than editing the DevStack installed copy.

- Source code changes aren't as easily lost when running `unstack.sh / stack.sh`
- The development server picks up source code changes while still running.
- Log messages and print statements go directly to the console.
- Debugging with `pdb` becomes much simpler to interact with.

Note: To ensure that JS and CSS changes are picked up without a server restart, you can disable compression with `COMPRESS_ENABLED = False` in your local settings file.

Horizons Structure

This project is a bit different from other OpenStack projects in that it has two very distinct components underneath it: `horizon`, and `openstack_dashboard`.

The `horizon` directory holds the generic libraries and components that can be used in any Django project.

The `openstack_dashboard` directory contains a reference Django project that uses `horizon`.

If dependencies are added to either `horizon` or `openstack_dashboard`, they should be added to `requirements.txt`.

Project Structure

Dashboard configuration

To add a new dashboard to your project, you need to add a configuration file to `openstack_dashboard/local/enabled` directory. For more information on this, see *Pluggable Panels and Groups*.

URLs

Then you add a single line to your projects `urls.py`

```
url(r'', include(horizon.urls)),
```

Those urls are automatically constructed based on the registered Horizon apps. If a different URL structure is desired it can be constructed by hand.

Templates

Pre-built template tags generate navigation. In your `nav.html` template you might have the following

```
{% load horizon %}

<div class='nav'>
  {% horizon_main_nav %}
</div>
```

And in your `sidebar.html` you might have

```
{% load horizon %}

<div class='sidebar'>
  {% horizon_dashboard_nav %}
</div>
```

These template tags are aware of the current active dashboard and panel via template context variables and will render accordingly.

Application Design

Structure

An application would have the following structure (well use project as an example)

```
project/
|---__init__.py
|---dashboard.py <-----Registers the app with Horizon and sets dashboard.
↳properties
|---overview/
|---images/
  |-- images
  |-- __init__.py
  |--panel.py <-----Registers the panel in the app and defines panel.
↳properties
  |-- snapshots/
  |-- templates/
  |-- tests.py
```

(continues on next page)

(continued from previous page)

```
|-- urls.py
|-- views.py
...
...
```

Dashboard Classes

Inside of `dashboard.py` you would have a class definition and the registration process

```
import horizon

....
# ObjectStorePanels is an example for a PanelGroup
# for panel classes in general, see below
class ObjectStorePanels(horizon.PanelGroup):
    slug = "object_store"
    name = _("Object Store")
    panels = ('containers',)

class Project(horizon.Dashboard):
    name = _("Project") # Appears in navigation
    slug = "project" # Appears in URL
    # panels may be strings or refer to classes, such as
    # ObjectStorePanels
    panels = (BasePanels, NetworkPanels, ObjectStorePanels)
    default_panel = 'overview'
    ...

horizon.register(Project)
```

Panel Classes

To connect a *Panel* with a *Dashboard* class you register it in a `panel.py` file

```
import horizon

from openstack_dashboard.dashboards.project import dashboard

class Images(horizon.Panel):
    name = "Images"
    slug = 'images'
    permissions = ('openstack.roles.admin', 'openstack.service.image')
    policy_rules = (('endpoint', 'endpoint:rule'),)
```

(continues on next page)

(continued from previous page)

```
# You could also register your panel with another application's dashboard
dashboard.Project.register(Images)
```

By default a *Panel* class looks for a `urls.py` file in the same directory as `panel.py` to include in the rollup of url patterns from panels to dashboards to Horizon, resulting in a wholly extensible, configurable URL structure.

Policy rules are defined in `horizon/openstack_dashboard/conf/`. Permissions are inherited from Keystone and take either the form `openstack.roles.role_name` or `openstack.services.service_name` for the users roles in keystone and the services in their service catalog.

3.1.5 Horizons tests and you

How to run the tests

Because Horizon is composed of both the `horizon` app and the `openstack_dashboard` reference project, there are in fact two sets of unit tests. While they can be run individually without problem, there is an easier way:

Included at the root of the repository is the `tox.ini` config which invokes both sets of tests, and optionally generates analyses on both components in the process. `tox` is what Jenkins uses to verify the stability of the project, so you should make sure you run it and it passes before you submit any pull requests/patches.

To run all tests:

```
$ tox
```

Its also possible to run a subset of the tests. Open `tox.ini` in the Horizon root directory to see a list of test environments. You can read more about `tox` in general at <https://tox.readthedocs.io/en/latest/>.

By default running the Selenium tests will open your Firefox browser (you have to install it first, else an error is raised), and you will be able to see the tests actions:

```
$ tox -e selenium
```

If you want to run the suite headless, without being able to see them (as they are ran on Jenkins), you can run the tests:

```
$ tox -e selenium-headless
```

Selenium will use a virtual display in this case, instead of your own. In order to run the tests this way you have to install the dependency `xvfb`, like this:

```
$ sudo apt install xvfb
```

for a Debian OS flavour, or for Fedora/Red Hat flavours:

```
$ sudo yum install xorg-x11-server-Xvfb
```

If you cant run a virtual display, or would prefer not to, you can use the PhantomJS web driver instead:

```
$ tox -e selenium-phantomjs
```

If you need to install PhantomJS, you may do so with *npm* like this:

```
$ npm -g install phantomjs
```

Alternatively, many distributions have system packages for PhantomJS, or it can be downloaded from <http://phantomjs.org/download.html>.

To run integration tests you should use *integration* tox environment:

```
$ tox -e integration
```

These tests requires *geckodriver* installed. It could be downloaded from <https://github.com/mozilla/geckodriver/releases>.

tox Test Environments

This is a list of test environments available to be executed by `tox -e <name>`.

pep8

Runs pep8, which is a tool that checks Python code style. You can read more about pep8 at <https://www.python.org/dev/peps/pep-0008/>

py37

Runs the Python unit tests against the current default version of Django with Python 3.7 environment. Check `requirements.txt` in horizon repository to know which version of Django is actually used.

All other dependencies are as defined by the upper-constraints file at <https://opendev.org/openstack/requirements/raw/branch/master/upper-constraints.txt>

You can run a subset of the tests by passing the test path as an argument to tox:

```
$ tox -e py37 -- openstack_dashboard/dashboards/identity/users/tests.py
```

The following is more example to run a specific test class and a specific test:

```
$ tox -e py37 -- openstack_dashboard/dashboards/identity/users/tests.  
↪py::UsersViewTests  
$ tox -e py37 -- openstack_dashboard/dashboards/identity/users/tests.  
↪py::UsersViewTests::test_index
```

The detail way to specify tests is found at [pytest documentation](#).

You can also pass other arguments. For example, to drop into a live debugger when a test fails you can use:

```
$ tox -e py37 -- --pdb
```


py3-dj111, py3-dj21, py3-dj22

Runs the Python unit tests against Django 1.11, Django 2.1 and Django 2.2 respectively

py36

Runs the Python unit tests with a Python 3.6 environment.

releasenotes

Outputs Horizons release notes as HTML to `releasenotes/build/html`.

Also takes an alternative builder as an optional argument, such as `tox -e docs -- <builder>`, which will output to `releasenotes/build/<builder>`. Available builders are listed at <http://www.sphinx-doc.org/en/latest/builders.html>

This environment also runs the documentation style checker `doc8` against RST and YAML files under `releasenotes/source` to keep the documentation style consistent. If you would like to run `doc8` manually, see **docs** environment below.

npm

Installs the npm dependencies listed in `package.json` and runs the JavaScript tests. Can also take optional arguments, which will be executed as an npm script following the dependency install, instead of `test`.

Example:

```
$ tox -e npm -- lintq
```

docs

Outputs Horizons documentation as HTML to `doc/build/html`.

Also takes an alternative builder as an optional argument, such as `tox -e docs -- <builder>`, which will output to `doc/build/<builder>`. Available builders are listed at <http://www.sphinx-doc.org/en/latest/builders.html>

Example:

```
$ tox -e docs -- latexpdf
```

This environment also runs the documentation style checker `doc8` against RST files under `doc/source` to keep the documentation style consistent. If you would like to run `doc8` manually, run:

```
# Activate virtualenv
$ . .tox/docs/bin/activate
$ doc8 doc/source
```

Writing tests

Horizon uses Django's unit test machinery (which extends Python's `unittest2` library) as the core of its test suite. As such, all tests for the Python code should be written as unit tests. No doctests please.

In general new code without unit tests will not be accepted, and every bugfix *must* include a regression test.

For a much more in-depth discussion of testing, see the *testing topic guide*.

3.1.6 Tutorials

Detailed tutorials to help you get started.

Tutorial: Creating an Horizon Plugin

Why should I package my code as a plugin?

We highly encourage that you write and maintain your code using our plugin architecture. A plugin by definition means the ability to be connected. In practical terms, plugins are a way to extend and add to the functionality that already exists. You can control its content and progress at a rate independent of Horizon. If you write and package your code as a plugin, it will continue to work in future releases.

Writing your code as a plugin also modularizes your code making it easier to translate and test. This also makes it easier for deployers to consume your code allowing selective enablement of features. We are currently using this pattern internally for our dashboards.

Creating the Plugin

This tutorial assumes you have a basic understanding of Python, HTML, JavaScript. Knowledge of AngularJS is optional but recommended if you are attempting to create an Angular plugin.

Name of your repository

Needless to say, it is important to choose a meaningful repository name.

In addition, if you plan to support translation on your dashboard plugin, it is recommended to choose a name like `xxxx-dashboard` (or `xxxx-ui`, `xxxx-horizon`). The OpenStack CI infra script considers a repository with these suffixes as Django project.

Types of Plugins that add content

The file structure for your plugin type will be different depending on your needs. Your plugin can be categorized into two types:

- Plugins that create new panels or dashboards
- Plugins that modify existing workflows, actions, etc (Angular only)

We will cover the basics of working with panels for both Python and Angular. If you are interested in creating a new panel, follow the steps below.

Note: This tutorial shows you how to create a new panel. If you are interested in creating a new dashboard plugin, use the file structure from *Tutorial: Building a Dashboard using Horizon* instead.

File Structure

Below is a skeleton of what your plugin should look like.:

```
myplugin
├── myplugin
│   ├── __init__.py
│   ├── enabled
│   │   └── _31000_myplugin.py
│   ├── api
│   │   ├── __init__.py
│   │   ├── my_rest_api.py
│   │   └── myservice.py
│   ├── content
│   │   ├── __init__.py
│   │   ├── mypanel
│   │   │   ├── __init__.py
│   │   │   ├── panel.py
│   │   │   ├── tests.py
│   │   │   ├── urls.py
│   │   │   ├── views.py
│   │   │   └── templates
│   │   │       └── mypanel
│   │   │           └── index.html
│   └── static
│       ├── dashboard
│       ├── identity
│       ├── myplugin
│       └── mypanel
```

(continues on next page)

(continued from previous page)

```
|           mypanel.html
|           mypanel.js
|           mypanel.scss

locale
  <lang>
    LC_MESSAGES
      django.po
      djangojs.po

setup.py
setup.cfg
LICENSE
MANIFEST.in
README.rst
babel-django.cfg
babel-djangojs.cfg
```

If you are creating a Python plugin, you may ignore the `static` folder. Most of the classes you need are provided for in Python. If you intend on adding custom front-end logic, you will need to include additional JavaScript here.

An AngularJS plugin is a collection of JavaScript files or static resources. Because it runs entirely in your browser, we need to place all of our static resources inside the `static` folder. This ensures that the Django static collector picks it up and distributes it to the browser correctly.

The Enabled File

The enabled folder contains the configuration file(s) that registers your plugin with Horizon. The file is prefixed with an alpha-numeric string that determines the load order of your plugin. For more information on what you can include in this file, see pluggable settings in *Settings Reference*.

`_31000_myplugin.py`:

```
# The name of the panel to be added to HORIZON_CONFIG. Required.
PANEL = 'mypanel'

# The name of the dashboard the PANEL associated with. Required.
PANEL_DASHBOARD = 'identity'

# Python panel class of the PANEL to be added.
ADD_PANEL = 'myplugin.content.mypanel.panel.MyPanel'

# A list of applications to be prepended to INSTALLED_APPS
ADD_INSTALLED_APPS = ['myplugin']

# A list of AngularJS modules to be loaded when Angular bootstraps.
ADD_ANGULAR_MODULES = ['horizon.dashboard.identity.myplugin.mypanel']

# Automatically discover static resources in installed apps
```

(continues on next page)

(continued from previous page)

```
AUTO_DISCOVER_STATIC_FILES = True

# A list of js files to be included in the compressed set of files
ADD_JS_FILES = []

# A list of scss files to be included in the compressed set of files
ADD_SCSS_FILES = ['dashboard/identity/myplugin/mypanel/mypanel.scss']

# A list of template-based views to be added to the header
ADD_HEADER_SECTIONS = ['myplugin.content.mypanel.views.HeaderView',]
```

Note: Currently, `AUTO_DISCOVER_STATIC_FILES = True` will only discover JavaScript files, not SCSS files.

my_rest_api.py

This file will likely be necessary if creating a plugin using Angular. Your plugin will need to communicate with a new service or require new interactions with a service already supported by Horizon. In this particular example, the plugin will augment the support for the already supported Identity service, Keystone. This file serves to define new REST interfaces for the plugins client-side to communicate with Horizon. Typically, the REST interfaces here make calls into `myservice.py`.

This file is unnecessary in a purely Django based plugin, or if your Angular based plugin is relying on CORS support in the desired service. For more information on CORS, see <https://docs.openstack.org/oslo.middleware/latest/admin/cross-project-cors.html>

myservice.py

This file will likely be necessary if creating a Django or Angular driven plugin. This file is intended to act as a convenient location for interacting with the new service this plugin is supporting. While interactions with the service can be handled in the `views.py`, isolating the logic is an established pattern in Horizon.

panel.py

We define a panel where our plugins content will reside in. This is currently a necessity even for Angular plugins. The slug is the panels unique identifier and is often use as part of the URL. Make sure that it matches what you have in your enabled file.:

```
from django.utils.translation import ugettext_lazy as _
import horizon

class MyPanel(horizon.Panel):
    name = _("My Panel")
    slug = "mypanel"
```

tests.py

Write some tests for the Django portion of your plugin and place them here.

urls.py

Now that we have a panel, we need to provide a URL so that users can visit our new panel! This URL generally will point to a view.:

```
from django.conf.urls import url

from myplugin.content.mypanel import views

urlpatterns = [
    url(r'^$', views.IndexView.as_view(), name='index'),
]
```

views.py

Because rendering is done client-side, all our view needs is to reference some HTML page. If you are writing a Python plugin, this view can be much more complex. Refer to the topic guides for more details.:

```
from django.views import generic

class IndexView(generic.TemplateView):
    template_name = 'identity/mypanel/index.html'
```

index.html

The index HTML is where rendering occurs. In this example, we are only using Django. If you are interested in using Angular directives instead, read the AngularJS section below.:

```
{% extends 'base.html' %}
{% load i18n %}
{% block title %}{% trans "My plugin" %}{% endblock %}

{% block page_header %}
    {% include "horizon/common/_domain_page_header.html" with title=_("My Panel
↪") %}
{% endblock page_header %}

{% block main %}
    Hello world!
{% endblock %}
```

At this point, you have a very basic plugin. Note that new templates are required to extend base.html. Including base.html is important for a number of reasons. It is the template that contains all of your static

resources along with any functionality external to your panel (things like navigation, context selection, etc). As of this moment, this is also true for Angular plugins.

MANIFEST.in

This file is responsible for listing the paths you want included in your tar.:

```
include setup.py

recursive-include myplugin *.js *.html *.scss
```

setup.py

```
# THIS FILE IS MANAGED BY THE GLOBAL REQUIREMENTS REPO - DO NOT EDIT
import setuptools

# In python < 2.7.4, a lazy loading of package `pbr` will break
# setuptools if some other modules registered functions in `atexit`.
# solution from: http://bugs.python.org/issue15881#msg170215
try:
    import multiprocessing # noqa
except ImportError:
    pass

setuptools.setup(
    setup_requires=['pbr>=1.8'],
    pbr=True)
```

setup.cfg

```
[metadata]
name = myplugin
summary = A panel plugin for OpenStack Dashboard
description-file =
    README.rst
author = myname
author_email = myemail
home-page = https://docs.openstack.org/horizon/latest/
classifier =
    Environment :: OpenStack
    Framework :: Django
    Intended Audience :: Developers
    Intended Audience :: System Administrators
    License :: OSI Approved :: Apache Software License
    Operating System :: POSIX :: Linux
    Programming Language :: Python
```

(continues on next page)

(continued from previous page)

```

Programming Language :: Python :: 2
Programming Language :: Python :: 2.7
Programming Language :: Python :: 3
Programming Language :: Python :: 3.5

[files]
packages =
    myplugin

```

AngularJS Plugin

If you have no plans to add AngularJS to your plugin, you may skip this section. In the tutorial below, we will show you how to customize your panel using Angular.

index.html

The index HTML is where rendering occurs and serves as an entry point for Angular. This is where we start to diverge from the traditional Python plugin. In this example, we use a Django template as the glue to our Angular template. Why are we going through a Django template for an Angular plugin? Long story short, `base.html` contains the navigation piece that we still need for each panel.

```

{% extends 'base.html' %}
{% load i18n %}
{% block title %}{% trans "My panel" %}{% endblock %}

{% block page_header %}
<hz-page-header
  header="{% 'My panel' | translate %}"
  description="{% 'My custom panel!' | translate %}">
</hz-page-header>
{% endblock page_header %}

{% block main %}
<ng-include src="{% STATIC_URL %}dashboard/identity/myplugin/mypanel/
↪mypanel.html">
</ng-include>
{% endblock %}

```

This template contains both Django and AngularJS code. Angular is denoted by `{{ . }}` while Django is denoted by `{% . %}` and `{% . %}`. This template gets processed twice, once by Django on the server-side and once more by Angular on the client-side. This means that the expressions in `{{ . }}` and `{% . %}` are substituted with values by the time it reaches your Angular template.

What you chose to include in `block main` is entirely up to you. Since you are creating an Angular plugin, we recommend that you keep everything in this section Angular. Do not mix Python code in here! If you find yourself passing in Python data, do it via our REST services instead.

Remember to always use `STATIC_URL` when referencing your static resources. This ensures that changes to the static path in settings will continue to serve your static resources properly.

Note: Angulars directives are prefixed with ng. Similarly, Horizons directives are prefixed with hz. You can think of them as namespaces.

mypanel.js

Your controller is the glue between the model and the view. In this example, we are going to give it some fake data to render. To load more complex data, consider using the `$http` service.

```
(function() {
  'use strict';

  angular
    .module('horizon.dashboard.identity.myplugin.mypanel', [])
    .controller('horizon.dashboard.identity.myPluginController',
      myPluginController);

  myPluginController.$inject = [ '$http' ];

  function myPluginController($http) {
    var ctrl = this;
    ctrl.items = [
      { name: 'abc', id: 123 },
      { name: 'efg', id: 345 },
      { name: 'hij', id: 678 }
    ];
  }
})();
```

This is a basic example where we mocked the data. For exercise, load your data using the `$http` service.

mypanel.html

This is our view. In this example, we are looping through the list of items provided by the controller and displaying the name and id. The important thing to note is the reference to our controller using the `ng-controller` directive.

```
<div ng-controller="horizon.dashboard.identity.myPluginController as ctrl">
  <div>Loading data from your controller:</div>
  <ul>
    <li ng-repeat="item in ctrl.items">
      <span class="c1">{{ item.name }}</span>
      <span class="c2">{{ item.id }}</span>
    </li>
  </ul>
</div>
```

mypanel.scss

You can choose to customize your panel by providing your own scss. Be sure to include it in your enabled file via the `ADD_SCSS_FILES` setting.

Translation Support

A general instruction on how to enable translation support is described in the Infrastructure User Manual¹.

This section describes topics specific to Horizon plugins.

ADD_INSTALLED_APPS

Be sure to include `<modulename>` (`myplugin` in this example) in `ADD_INSTALLED_APPS` in the corresponding enabled file.

- If you are preparing a new plugin, you will use `<modulename>` as `INSTALLED_APPS` in most cases as suggested in this tutorial. This is good and there is nothing more to do.
- If for some reason your plugin needs to register other python modules to `ADD_INSTALLED_APPS`, ensure that you include its `<modulename>` additionally.

This comes from the combination of the following two reasons.

- Django looks for translation message catalogs from each path specified in `INSTALLED_APPS`².
- OpenStack infra scripts assumes translation message catalogs are placed under `<modulename>/locale` (for example `myplugin/locale`).

myplugin/locale

Translated message catalog files (PO files) are placed under this directory.

babel-django.cfg, babel-djangojs.cfg

These files are used to extract messages by `pybabel`: `babel-django.cfg` for python code and template files, and `babel-djangojs.cfg` for JavaScript files.

They are required to enable translation support by OpenStack CI infra. If they do not exist, the translation jobs will skip processing for your project.

¹ <https://docs.openstack.org/infra/manual/creators.html#enabling-translation-infrastructure>

² <https://docs.djangoproject.com/es/1.9/topics/i18n/translation/#how-django-discovers-translations>

Installing Your Plugin

Now that you have a complete plugin, it is time to install and test it. The instructions below assume that you have a working plugin.

- `plugin` is the location of your plugin
 - `horizon` is the location of horizon
 - `package` is the complete name of your packaged plugin
1. Run `cd plugin & python setup.py sdist`
 2. Run `cp -rv enabled horizon/openstack_dashboard/local/`
 3. Run `horizon/tools/with_venv.sh pip install dist/package.tar.gz`
 4. Restart Apache or your Django test server

Note: Step 3 installs your package into the Horizons virtual environment. You can install your plugin without using `with_venv.sh` and `pip`. The package would simply be installed in the `PYTHON_PATH` of the system instead.

If you are able to hit the URL pattern in `urls.py` in your browser, you have successfully deployed your plugin! For plugins that do not have a URL, check that your static resources are loaded using the browser inspector.

Assuming you implemented `my_rest_api.py`, you can use a REST client to hit the url directly and test it. There should be many REST clients available on your web browser.

Note that you may need to rebuild your virtual environment if your plugin is not showing up properly. If your plugin does not show up properly, check your `.tox` folder to make sure the plugins content is as you expect.

Note: To uninstall, use `pip uninstall`. You will also need to remove the enabled file from the `local/enabled` folder.

Tutorial: Building a Dashboard using Horizon

This tutorial covers how to use the various components in horizon to build an example dashboard and a panel with a tab which has a table containing data from the back end.

As an example, we'll create a new `My Dashboard` dashboard with a `My Panel` panel that has an `Instances Tab` tab. The tab has a table which contains the data pulled by the Nova instances API.

Note: This tutorial assumes you have either a `devstack` or `openstack` environment up and running. There are a variety of other resources which may be helpful to read first. For example, you may want to start with the [Quickstart](#) or the [Django tutorial](#).

Creating a dashboard

The quick version

Horizon provides a custom management command to create a typical base dashboard structure for you. Run the following commands in your Horizon root directory. It generates most of the boilerplate code you need:

```
$ mkdir openstack_dashboard/dashboards/mydashboard

$ tox -e manage -- startdash mydashboard \
    --target openstack_dashboard/dashboards/mydashboard

$ mkdir openstack_dashboard/dashboards/mydashboard/mypanel

$ tox -e manage -- startpanel mypanel \
    --dashboard=openstack_dashboard.dashboards.mydashboard \
    --target=openstack_dashboard/dashboards/mydashboard/mypanel
```

You will notice that the directory `mydashboard` gets automatically populated with the files related to a dashboard and the `mypanel` directory gets automatically populated with the files related to a panel.

Structure

If you use the `tree mydashboard` command to list the `mydashboard` directory in `openstack_dashboard/dashboards`, you will see a directory structure that looks like the following:

```
mydashboard
  dashboard.py
  __init__.py
  mypanel
  __init__.py
  panel.py
  templates
  __init__.py
  __init__.py
  index.html
  tests.py
  urls.py
  views.py
  static
  __init__.py
  scss
  __init__.py
  mydashboard.scss
  js
  mydashboard.js
  templates
  mydashboard
  base.html
```

For this tutorial, we will not deal with the static directory, or the `tests.py` file. Leave them as they are. With the rest of the files and directories in place, we can move on to add our own dashboard.

Defining a dashboard

Open the `dashboard.py` file. You will notice the following code has been automatically generated:

```
from django.utils.translation import ugettext_lazy as _

import horizon

class Mydashboard(horizon.Dashboard):
    name = _("Mydashboard")
    slug = "mydashboard"
    panels = ()          # Add your panels here.
    default_panel = ''  # Specify the slug of the dashboard's default panel.

horizon.register(Mydashboard)
```

If you want the dashboard name to be something else, you can change the name attribute in the `dashboard.py` file. For example, you can change it to be `My Dashboard`

```
name = _("My Dashboard")
```

A dashboard class will usually contain a name attribute (the display name of the dashboard), a slug attribute (the internal name that could be referenced by other components), a list of panels, default panel, etc. We will cover how to add a panel in the next section.

Creating a panel

We'll create a panel and call it `My Panel`.

Structure

As described above, the `mypanel` directory under `openstack_dashboard/dashboards/mydashboard` should look like the following:

```
mypanel
  __init__.py
  panel.py
  templates
  ãã mypanel
  ãã ã index.html
  tests.py
  urls.py
  views.py
```

Defining a panel

The `panel.py` file referenced above has a special meaning. Within a dashboard, any module name listed in the `panels` attribute on the dashboard class will be auto-discovered by looking for the `panel.py` file in a corresponding directory (the details are a bit magical, but have been thoroughly vetted in Django's admin codebase).

Open the `panel.py` file, you will have the following auto-generated code:

```
from django.utils.translation import ugettext_lazy as _

import horizon

from openstack_dashboard.dashboards.mydashboard import dashboard

class Mypanel(horizon.Panel):
    name = _("Mypanel")
    slug = "mypanel"

dashboard.Mydashboard.register(Mypanel)
```

If you want the panel name to be something else, you can change the `name` attribute in the `panel.py` file. For example, you can change it to be `My Panel`:

```
name = _("My Panel")
```

Open the `dashboard.py` file again, insert the following code above the `Mydashboard` class. This code defines the `Mygroup` class and adds a panel called `mypanel`:

```
class Mygroup(horizon.PanelGroup):
    slug = "mygroup"
    name = _("My Group")
    panels = ('mypanel',)
```

Modify the `Mydashboard` class to include `Mygroup` and add `mypanel` as the default panel:

```
class Mydashboard(horizon.Dashboard):
    name = _("My Dashboard")
    slug = "mydashboard"
    panels = (Mygroup,) # Add your panels here.
    default_panel = 'mypanel' # Specify the slug of the default panel.
```

The completed `dashboard.py` file should look like the following:

```
from django.utils.translation import ugettext_lazy as _

import horizon

class Mygroup(horizon.PanelGroup):
```

(continues on next page)

(continued from previous page)

```

slug = "mygroup"
name = _("My Group")
panels = ('mypanel',)

class Mydashboard(horizon.Dashboard):
    name = _("My Dashboard")
    slug = "mydashboard"
    panels = (Mygroup,) # Add your panels here.
    default_panel = 'mypanel' # Specify the slug of the default panel.

horizon.register(Mydashboard)

```

Tables, Tabs, and Views

We'll start with the table, combine that with the tabs, and then build our view from the pieces.

Defining a table

Horizon provides a `SelfHandlingForm` `DataTable` class which simplifies the vast majority of displaying data to an end-user. We're just going to skim the surface here, but it has a tremendous number of capabilities. Create a `tables.py` file under the `mypanel` directory and add the following code:

```

from django.utils.translation import ugettext_lazy as _

from horizon import tables

class InstancesTable(tables.DataTable):
    name = tables.Column("name", verbose_name=_("Name"))
    status = tables.Column("status", verbose_name=_("Status"))
    zone = tables.Column('availability_zone',
                          verbose_name=_("Availability Zone"))
    image_name = tables.Column('image_name', verbose_name=_("Image Name"))

    class Meta(object):
        name = "instances"
        verbose_name = _("Instances")

```

There are several things going on here we created a table subclass, and defined four columns that we want to retrieve data and display. Each of those columns defines what attribute it accesses on the instance object as the first argument, and since we like to make everything translatable, we give each column a `verbose_name` that's marked for translation.

Lastly, we added a `Meta` class which indicates the meta object that describes the instances table.

Note: This is a slight simplification from the reality of how the instance object is actually structured. In

reality, accessing other attributes requires an additional step.

Adding actions to a table

Horizon provides three types of basic action classes which can be taken on a tables data:

- *Action*
- *LinkAction*
- *FilterAction*

There are also additional actions which are extensions of the basic Action classes:

- *BatchAction*
- *DeleteAction*
- *FixedFilterAction*

Now lets create and add a filter action to the table. To do so, we will need to edit the `tables.py` file used above. To add a filter action which will only show rows which contain the string entered in the filter field, we must first define the action:

```
class MyFilterAction(tables.FilterAction):
    name = "myfilter"
```

Note: The action specified above will default the `filter_type` to be "query". This means that the filter will use the client side table sorter.

Then, we add that action to the table actions for our table.:

```
class InstancesTable:
    class Meta(object):
        table_actions = (MyFilterAction,)
```

The completed `tables.py` file should look like the following:

```
from django.utils.translation import ugettext_lazy as _

from horizon import tables

class MyFilterAction(tables.FilterAction):
    name = "myfilter"

class InstancesTable(tables.DataTable):
    name = tables.Column('name', \
                        verbose_name=_("Name"))
    status = tables.Column('status', \
                          verbose_name=_("Status"))
```

(continues on next page)

(continued from previous page)

```

zone = tables.Column('availability_zone', \
                      verbose_name=_("Availability Zone"))
image_name = tables.Column('image_name', \
                            verbose_name=_("Image Name"))

class Meta(object):
    name = "instances"
    verbose_name = _("Instances")
    table_actions = (MyFilterAction,)

```

Defining tabs

So we have a table, ready to receive our data. We could go straight to a view from here, but in this case we were also going to use horizons `TabGroup` class.

Create a `tabs.py` file under the `mypanel` directory. Lets make a tab group which has one tab. The completed code should look like the following:

```

from django.utils.translation import ugettext_lazy as _

from horizon import exceptions
from horizon import tabs

from openstack_dashboard import api
from openstack_dashboard.dashboards.mydashboard.mypanel import tables

class InstanceTab(tabs.TableTab):
    name = _("Instances Tab")
    slug = "instances_tab"
    table_classes = (tables.InstancesTable,)
    template_name = ("horizon/common/_detail_table.html")
    preload = False

    def has_more_data(self, table):
        return self._has_more

    def get_instances_data(self):
        try:
            marker = self.request.GET.get(
                tables.InstancesTable._meta.pagination_param, None)

            instances, self._has_more = api.nova.server_list(
                self.request,
                search_opts={'marker': marker, 'paginate': True})

            return instances
        except Exception:
            self._has_more = False

```

(continues on next page)

(continued from previous page)

```
        error_message = _('Unable to get instances')
        exceptions.handle(self.request, error_message)

        return []

class MypanelTabs(tabs.TabGroup):
    slug = "mypanel_tabs"
    tabs = (InstanceTab,)
    sticky = True
```

This tab gets a little more complicated. The tab handles data tables (and all their associated features), and it also uses the `preload` attribute to specify that this tab shouldn't be loaded by default. It will instead be loaded via AJAX when someone clicks on it, saving us on API calls in the vast majority of cases.

Additionally, the displaying of the table is handled by a reusable template, `horizon/common/_detail_table.html`. Some simple pagination code was added to handle large instance lists.

Lastly, this code introduces the concept of error handling in horizon. The `horizon.exceptions.handle()` function is a centralized error handling mechanism that takes all the guess-work and inconsistency out of dealing with exceptions from the API. Use it everywhere.

Tying it together in a view

There are lots of pre-built class-based views in horizon. We try to provide the starting points for all the common combinations of components.

Open the `views.py` file, the auto-generated code is like the following:

```
from horizon import views

class IndexView(views.APIView):
    # A very simple class-based view...
    template_name = 'mydashboard/mypanel/index.html'

    def get_data(self, request, context, *args, **kwargs):
        # Add data to the context here...
        return context
```

In this case we want a starting view type that works with both tabs and tables that'd be the `TabbedTableView` class. It takes the best of the dynamic delayed-loading capabilities tab groups provide and mixes in the actions and AJAX-updating that tables are capable of with almost no work on the users end. Change `views.APIView` to be `tabs.TabbedTableView` and add `MypanelTabs` as the tab group class in the `IndexView` class:

```
class IndexView(tabs.TabbedTableView):
    tab_group_class = mydashboard_tabs.MypanelTabs
```

After importing the proper package, the completed `views.py` file now looks like the following:

```

from horizon import tabs

from openstack_dashboard.dashboards.mydashboard.mypanel \
    import tabs as mydashboard_tabs

class IndexView(tabs.TabbedTableView):
    tab_group_class = mydashboard_tabs.MypanelTabs
    template_name = 'mydashboard/mypanel/index.html'

    def get_data(self, request, context, *args, **kwargs):
        # Add data to the context here...
        return context

```

URLs

The auto-generated `urls.py` file is like:

```

from django.conf.urls import url

from openstack_dashboard.dashboards.mydashboard.mypanel import views

urlpatterns = [
    url(r'^$', views.IndexView.as_view(), name='index'),
]

```

The template

Open the `index.html` file in the `mydashboard/mypanel/templates/mypanel` directory, the auto-generated code is like the following:

```

{% extends 'base.html' %}
{% load i18n %}
{% block title %}{% trans "Mypanel" %}{% endblock %}

{% block page_header %}
    {% include "horizon/common/_page_header.html" with title=_("Mypanel") %}
{% endblock page_header %}

{% block main %}
{% endblock %}

```

The main block must be modified to insert the following code:

```

<div class="row">
  <div class="col-sm-12">
    {{ tab_group.render }}
  </div>
</div>

```

(continues on next page)

(continued from previous page)

```
</div>
</div>
```

If you want to change the title of the `index.html` file to be something else, you can change it. For example, change it to be `My Panel` in the `block title` section. If you want the `title` in the `block page_header` section to be something else, you can change it. For example, change it to be `My Panel`. The updated code could be like:

```
{% extends 'base.html' %}
{% load i18n %}
{% block title %}{% trans "My Panel" %}{% endblock %}

{% block page_header %}
    {% include "horizon/common/_page_header.html" with title=_("My Panel") %}
{% endblock page_header %}

{% block main %}
<div class="row">
    <div class="col-sm-12">
        {{ tab_group.render }}
    </div>
</div>
{% endblock %}
```

This gives us a custom page title, a header, and renders our tab group provided by the view.

With all our code in place, the only thing left to do is to integrate it into our OpenStack Dashboard site.

Note: For more information about Django views, URLs and templates, please refer to the [Django documentation](#).

Enable and show the dashboard

In order to make `My Dashboard` show up along with the existing dashboards like `Project` or `Admin` on `horizon`, you need to create a file called `_50_mydashboard.py` under `openstack_dashboard/enabled` and add the following:

```
# The name of the dashboard to be added to HORIZON['dashboards']. Required.
DASHBOARD = 'mydashboard'

# If set to True, this dashboard will not be added to the settings.
DISABLED = False

# A list of applications to be added to INSTALLED_APPS.
ADD_INSTALLED_APPS = [
    'openstack_dashboard.dashboards.mydashboard',
]
```

Run and check the dashboard

Everything is in place, now run Horizon on the different port:

```
$ tox -e runserver -- 0:9000
```

Go to `http://<your server>:9000` using a browser. After login as an admin you should be able see My Dashboard shows up at the left side on horizon. Click it, My Group will expand with My Panel. Click on My Panel, the right side panel will display an Instances Tab which has an Instances table.

If you dont see any instance data, you havent created any instances yet. Go to dashboard Project -> Images, select a small image, for example, `cirros-0.3.1-x86_64-uec`, click Launch and enter an Instance Name, click the button Launch. It should create an instance if the OpenStack or devstack is correctly set up. Once the creation of an instance is successful, go to My Dashboard again to check the data.

Adding a complex action to a table

For a more detailed look into adding a table action, one that requires forms for gathering data, you can walk through *Tutorial: Adding a complex action to a table* tutorial.

Conclusion

What youve learned here is the fundamentals of how to write interfaces for your own project based on the components horizon provides.

If you have feedback on how this tutorial could be improved, please feel free to submit a bug against Horizon in [launchpad](#).

Tutorial: Adding a complex action to a table

This tutorial covers how to add a more complex action to a table, one that requires an action and form definitions, as well as changes to the view, urls, and table.

This tutorial assumes you have already completed *Tutorial: Building a Dashboard using Horizon*. If not, please do so now as we will be modifying the files created there.

This action will create a snapshot of the instance. When the action is taken, it will display a form that will allow the user to enter a snapshot name, and will create that snapshot when the form is closed using the Create snapshot button.

Defining the view

To define the view, we must create a view class, along with the template (HTML) file and the form class for that view.

The template file

The template file contains the HTML that will be used to show the view.

Create a `create_snapshot.html` file under the `mypanel/templates/mypanel` directory and add the following code:

```
{% extends 'base.html' %}
{% load i18n %}
{% block title %}{% trans "Create Snapshot" %}{% endblock %}

{% block page_header %}
    {% include "horizon/common/_page_header.html" with title=_("Create a
    ↳ Snapshot") %}
{% endblock page_header %}

{% block main %}
    {% include 'mydashboard/mypanel/_create_snapshot.html' %}
{% endblock %}
```

As you can see, the main body will be defined in `_create_snapshot.html`, so we must also create that file under the `mypanel/templates/mypanel` directory. It should contain the following code:

```
{% extends "horizon/common/_modal_form.html" %}
{% load i18n %}

{% block modal-body-right %}
    <h3>{% trans "Description:" %}</h3>
    <p>{% trans "Snapshots preserve the disk state of a running instance." %}
    ↳</p>
{% endblock %}
```

The form

Horizon provides a `SelfHandlingForm` class which simplifies some of the details involved in creating a form. Our form will derive from this class, adding a character field to allow the user to specify a name for the snapshot, and handling the successful closure of the form by calling the nova api to create the snapshot.

Create the `forms.py` file under the `mypanel` directory and add the following:

```
from django.urls import reverse
from django.utils.translation import gettext_lazy as _

from horizon import exceptions
```

(continues on next page)

(continued from previous page)

```

from horizon import forms

from openstack_dashboard import api

class CreateSnapshot(forms.SelfHandlingForm):
    instance_id = forms.CharField(label=_("Instance ID"),
                                  widget=forms.HiddenInput(),
                                  required=False)
    name = forms.CharField(max_length=255, label=_("Snapshot Name"))

    def handle(self, request, data):
        try:
            snapshot = api.nova.snapshot_create(request,
                                                data['instance_id'],
                                                data['name'])

            return snapshot
        except Exception:
            exceptions.handle(request,
                              _('Unable to create snapshot.'))

```

The view

Now, the view will tie together the template and the form. Horizon provides a *ModalFormView* class which simplifies the creation of a view that will contain a modal form.

Open the `views.py` file under the `mypanel` directory and add the code for the `CreateSnapshotView` and the necessary imports. The complete file should now look something like this:

```

from django.urls import reverse
from django.urls import reverse_lazy
from django.utils.translation import ugettext_lazy as _

from horizon import tabs
from horizon import exceptions
from horizon import forms

from horizon.utils import memoized

from openstack_dashboard import api

from openstack_dashboard.dashboards.mydashboard.mypanel \
    import forms as project_forms

from openstack_dashboard.dashboards.mydashboard.mypanel \
    import tabs as mydashboard_tabs

class IndexView(tabs.TabbedTableView):

```

(continues on next page)

(continued from previous page)

```

tab_group_class = mydashboard_tabs.MypanelTabs
# A very simple class-based view...
template_name = 'mydashboard/mypanel/index.html'

def get_data(self, request, context, *args, **kwargs):
    # Add data to the context here...
    return context

class CreateSnapshotView(forms.ModalFormView):
    form_class = project_forms.CreateSnapshot
    template_name = 'mydashboard/mypanel/create_snapshot.html'
    success_url = reverse_lazy("horizon:project:images:index")
    modal_id = "create_snapshot_modal"
    modal_header = _("Create Snapshot")
    submit_label = _("Create Snapshot")
    submit_url = "horizon:mydashboard:mypanel:create_snapshot"

    @memoized.memoized_method
    def get_object(self):
        try:
            return api.nova.server_get(self.request,
                                       self.kwargs["instance_id"])
        except Exception:
            exceptions.handle(self.request,
                             _("Unable to retrieve instance.))

    def get_initial(self):
        return {"instance_id": self.kwargs["instance_id"]}

    def get_context_data(self, **kwargs):
        context = super(CreateSnapshotView, self).get_context_data(**kwargs)
        instance_id = self.kwargs['instance_id']
        context['instance_id'] = instance_id
        context['instance'] = self.get_object()
        context['submit_url'] = reverse(self.submit_url, args=[instance_id])
        return context

```

Adding the url

We must add the url for our new view. Open the `urls.py` file under the `mypanel` directory and add the following as a new url pattern:

```

url(r'^(?P<instance_id>[^/]+)/create_snapshot/$',
    views.CreateSnapshotView.as_view(),
    name='create_snapshot'),

```

The complete `urls.py` file should look like this:


```

from django.conf.urls import url

from openstack_dashboard.dashboards.mydashboard.mypanel import views

urlpatterns = [
    url(r'^$',
        views.IndexView.as_view(), name='index'),
    url(r'^(?P<instance_id>[^/]+)/create_snapshot/$',
        views.CreateSnapshotView.as_view(),
        name='create_snapshot'),
]

```

Define the action

Horizon provides a *LinkAction* class which simplifies adding an action which can be used to display another view.

We will add a link action to the table that will be accessible from each row in the table. The action will use the view defined above to create a snapshot of the instance represented by the row in the table.

To do this, we must edit the `tables.py` file under the `mypanel` directory and add the following:

```

def is_deleting(instance):
    task_state = getattr(instance, "OS-EXT-STS:task_state", None)
    if not task_state:
        return False
    return task_state.lower() == "deleting"

class CreateSnapshotAction(tables.LinkAction):
    name = "snapshot"
    verbose_name = _("Create Snapshot")
    url = "horizon:mydashboard:mypanel:create_snapshot"
    classes = ("ajax-modal",)
    icon = "camera"

    # This action should be disabled if the instance
    # is not active, or the instance is being deleted
    def allowed(self, request, instance=None):
        return instance.status in ("ACTIVE") \
            and not is_deleting(instance)

```

We must also add our new action as a row action for the table:

```
row_actions = (CreateSnapshotAction,)
```

The complete `tables.py` file should look like this:

```
from django.utils.translation import ugettext_lazy as _

from horizon import tables

def is_deleting(instance):
    task_state = getattr(instance, "OS-EXT-STS:task_state", None)
    if not task_state:
        return False
    return task_state.lower() == "deleting"

class CreateSnapshotAction(tables.LinkAction):
    name = "snapshot"
    verbose_name = _("Create Snapshot")
    url = "horizon:mydashboard:mypanel:create_snapshot"
    classes = ("ajax-modal",)
    icon = "camera"

    def allowed(self, request, instance=None):
        return instance.status in ("ACTIVE") \
            and not is_deleting(instance)

class MyFilterAction(tables.FilterAction):
    name = "myfilter"

class InstancesTable(tables.DataTable):
    name = tables.Column("name", verbose_name=_("Name"))
    status = tables.Column("status", verbose_name=_("Status"))
    zone = tables.Column('availability_zone', verbose_name=_("Availability_↵
↵Zone"))
    image_name = tables.Column('image_name', verbose_name=_("Image Name"))

    class Meta(object):
        name = "instances"
        verbose_name = _("Instances")
        table_actions = (MyFilterAction,)
        row_actions = (CreateSnapshotAction,)
```

Run and check the dashboard

We must once again run horizon to verify our dashboard is working:

```
$ tox -e runserver -- 0:9000
```

Go to `http://<your server>:9000` using a browser. After login as an admin, display My Panel to see the Instances table. For every ACTIVE instance in the table, there will be a Create Snapshot action on the row. Click on Create Snapshot, enter a snapshot name in the form that is shown, then click to close the form. The Project Images view should be shown with the new snapshot added to the table.

Conclusion

What youve learned here is the fundamentals of how to add a table action that requires a form for data entry. This can easily be expanded from creating a snapshot to other API calls that require more complex forms to gather the necessary information.

If you have feedback on how this tutorial could be improved, please feel free to submit a bug against Horizon in [launchpad](#).

Extending an AngularJS Workflow

A workflow extends the `extensibleService`. This means that all workflows inherit properties and methods provided by the `extensibleService`. Extending a workflow allows you to add your own steps, remove existing steps, and inject custom data handling logic. Refer to inline documentation on what those properties and methods are.

We highly recommend that you complete the *Tutorial: Creating an Horizon Plugin* if you have not done so already. If you do not know how to package and install a plugin, the rest of this tutorial will not make sense! In this tutorial, we will examine an existing workflow and how we can extend it as a plugin.

Note: Although this tutorial focuses on extending a workflow, the steps here can easily be adapted to extend any service that inherited the `extensibleService`. Examples of other extensible points include table columns and table actions.

File Structure

Remember that the goal of this tutorial is to inject our custom step into an **existing** workflow. All of the files we are interested in reside in the `static` folder.

```
myplugin
  enabled
    _31000_myplugin.py
  static
    horizon
```

(continues on next page)

```
    app
      core
        images
          plugins
            myplugin.module.js

          steps
            mystep
              mystep.controller.js
              mystep.help.html
              mystep.html
```

myplugin.module.js

This is the entry point into our plugin. We hook into an existing module via the run block which is executed after the module has been initialized. All we need to do is inject it as a dependency and then use the methods provided in the extensible service to override or modify steps. In this example, we are going to prepend our custom step so that it will show up as the first step in the wizard.

```
(function () {
  'use strict';

  angular
    .module('horizon.app.core.images')
    .run(myPlugin);

  myPlugin.$inject = [
    'horizon.app.core.images.basePath',
    'horizon.app.core.images.workflows.create-volume.service'
  ];

  function myPlugin(basePath, workflow) {
    var customStep = {
      id: 'mypluginstep',
      title: gettext('My Step'),
      templateUrl: basePath + 'steps/mystep/mystep.html',
      helpUrl: basePath + 'steps/mystep/mystep.help.html',
      formName: 'myStepForm'
    };
    workflow.prepend(customStep);
  }
})();
```

Note: Replace `horizon.app.core.images.workflows.create-volume.service` with the workflow you intend to augment.

mystep.controller.js

It is important to note that the scope is the glue between our controllers, this is how we are propagating events from one controller to another. We can propagate events upward using the `$emit` method and propagate events downward using the `$broadcast` method.

Using the `$on` method, we can listen to events generated within the scope. In this manner, actions we completed in the wizard are visually reflected in the table even though they are two completely different widgets. Similarly, you can share data between steps in your workflow as long as they share the same parent scope.

In this example, we are listening for events generated by the wizard and the user panel. We also emit a custom event that other controllers can register to when favorite color changes.

```
(function() {
  'use strict';

  angular
    .module('horizon.app.core.images')
    .controller('horizon.app.core.images.steps.myStepController',
      myStepController);

  myStepController.$inject = [
    '$scope',
    'horizon.framework.widgets.wizard.events',
    'horizon.app.core.images.events'
  ];

  function myStepController($scope, wizardEvents, imageEvents) {

    var ctrl = this;
    ctrl.favoriteColor = 'red';

    //////////////////////////////////////////////////

    $scope.$on(wizardEvents.ON_SWITCH, function(e, args) {
      console.info('Wizard is switching step!');
      console.info(args);
    });

    $scope.$on(wizardEvents.BEFORE_SUBMIT, function() {
      console.info('About to submit!');
    });

    $scope.$on(imageEvents.VOLUME_CHANGED, function(event, newVolume) {
      console.info(newVolume);
    });

    //////////////////////////////////////////////////

    $scope.$watchCollection(getFavoriteColor, watchFavoriteColor);
  }
})
```

(continues on next page)

(continued from previous page)

```

function getFavoriteColor() {
    return ctrl.favoriteColor;
}

function watchFavoriteColor(newColor, oldColor) {
    if (newColor !== oldColor) {
        $scope.$emit('mystep.favoriteColor', newColor);
    }
}
}
}

})();

```

mystep.help.html

In this tutorial, we will leave this file blank. Include additional information here if your step requires it. Otherwise, remove the file and the `helpUrl` property from your step.

mystep.html

This file contains contents you want to display to the user. We will provide a simple example of a step that asks for your favorite color. The most important thing to note here is the reference to our controller via the `ng-controller` directive. This is essentially the link to our controller.

```

<div ng-controller="horizon.app.core.images.steps.myStepController as ctrl">
  <h1 translate>Blue Plugin</h1>
  <div class="content">
    <div class="subtitle" translate>My custom step</div>
    <div translate style="margin-bottom:1em;">
      Place your custom content here!
    </div>
    <div class="selected-source clearfix">
      <div class="row">
        <div class="col-xs-12 col-sm-8">
          <div class="form-group required">
            <label class="control-label" translate>Favorite color</label>
            <input type="text" class="form-control"
              ng-model="ctrl.favoriteColor"
              placeholder="{ $ 'Enter your favorite color'|translate $ }">
          </div>
        </div>
      </div>
    </div>
    <div><!-- row -->
  </div><!-- clearfix -->
</div><!-- content -->
</div><!-- controller -->

```

Testing

Now that we have completed our plugin, lets package it and test that it works. If you need a refresher, take a look at the installation section in *Tutorial: Creating an Horizon Plugin*.

3.1.7 Topic Guides

Information on how to work with specific areas of Horizon can be found in the following topic guides.

Code Style

As a project, Horizon adheres to code quality standards.

Python

We follow [PEP8](#) for all our Python code, and use `pep8.py` (available via the shortcut `tox -e pep8`) to validate that our code meets proper Python style guidelines.

Django

Additionally, we follow [Djangos style guide](#) for templates, views, and other miscellany.

JavaScript

The following standards are divided into required and recommended sections. Our main goal in establishing these best practices is to have code that is reliable, readable, and maintainable.

Required

Reliable

- The code has to work on the stable and latest versions of Firefox, Chrome, Safari, and Opera web browsers, and on Microsoft Internet Explorer 11 and later.
- If you turned compression off during development via `COMPRESS_ENABLED = False` in `local_settings.py`, re-enable compression and test your code before submitting.
- Use `===` as opposed to `==` for equality checks. The `==` will do a type cast before comparing, which can lead to unwanted results.

Note: If typecasting is desired, explicit casting is preferred to keep the meaning of your code clear.

- Keep document reflows to a minimum. DOM manipulation is expensive, and can become a performance issue. If you are accessing the DOM, make sure that you are doing it in the most optimized way. One example is to build up a document fragment and then append the fragment to the DOM in one pass instead of doing multiple smaller DOM updates.

- Use strict, enclosing each JavaScript file inside a self-executing function. The self-executing function keeps the strict scoped to the file, so its variables and methods are not exposed to other JavaScript files in the product.

Note: Using strict will throw exceptions for common coding errors, like accessing global vars, that normally are not flagged.

Example:

```
(function(){
  'use strict';
  // code...
})();
```

- Use `forEach` | `each` when looping whenever possible. AngularJS and jQuery both provide for each loops that provide both iteration and scope.

AngularJS:

```
angular.forEach(objectToIterateOver, function(value, key) {
  // loop logic
});
```

jQuery:

```
$.each(objectToIterateOver, function(key, value) {
  // loop logic
});
```

- Do not put variables or functions in the global namespace. There are several reasons why globals are bad, one being that all JavaScript included in an application runs in the same scope. The issue with that is if another script has the same method or variable names they overwrite each other.
- Always put `var` in front of your variables. Not putting `var` in front of a variable puts that variable into the global space, see above.
- Do not use `eval()`. The `eval` (expression) evaluates the expression passed to it. This can open up your code to security vulnerabilities and other issues.
- Do not use `with` object {code}. The `with` statement is used to access properties of an object. The issue with `with` is that its execution is not consistent, so by reading the statement in the code it is not always clear how it is being used.

Readable & Maintainable

- Give meaningful names to methods and variables.
- Avoid excessive nesting.
- Avoid HTML and CSS in JS code. HTML and CSS belong in templates and stylesheets respectively. For example:
 - In our HTML files, we should focus on layout.
 1. Reduce the small/random `<script>` and `<style>` elements in HTML.

2. Avoid in-lining styles into element in HTML. Use attributes and classes instead.
- In our JS files, we should focus on logic rather than attempting to manipulate/style elements.
1. Avoid statements such as `element.css({property1,property2...})` they belong in a CSS class.
 2. Avoid statements such as `$("<div>abc</div>")` they belong in a HTML template file. Use `show | hide | clone` elements if dynamic content is required.
 3. Avoid using classes for detection purposes only, instead, defer to attributes. For example to find a div:

```
<div class="something"></div>
$(".something").html("Don't find me this way!");
```

is better found like:

```
<div data-something></div>
$("div[data-something]").html("You found me correctly!");
```

- Avoid commented-out code.
- Avoid dead code.

Performance

- Avoid creating instances of the same object repeatedly within the same scope. Instead, assign the object to a variable and re-use the existing object. For example:

```
$(document).on('click', function() { /* do something. */ });
$(document).on('mouseover', function() { /* do something. */ });
```

A better approach:

```
var $document = $(document);
$document.on('click', function() { /* do something. */ });
$document.on('mouseover', function() { /* do something. */ });
```

In the first approach a jQuery object for `document` is created each time. The second approach creates only one jQuery object and reuses it. Each object needs to be created, uses memory, and needs to be garbage collected.

Recommended

Readable & Maintainable

- Put a comment at the top of every file explaining what the purpose of this file is when the naming is not obvious. This guideline also applies to methods and variables.
- Source-code formatting (or beautification) is recommended but should be used with caution. Keep in mind that if you reformat an entire file that was not previously formatted the same way, it will mess up the diff during the code review. It is best to use a formatter when you are working on a new file by yourself, or with others who are using the same formatter. You can also choose to format a selected portion of a file only. Instructions for setting up ESLint for Eclipse, Sublime Text, Notepad++ and WebStorm/PyCharm are [provided](#).

- Use 2 spaces for code indentation.
- Use { } for if, for, while statements, and dont combine them on one line.

```
// Do this           //Not this           // Not this
if(x) {              if(x)                  if(x) y =x;
  y=x;
}
```

- Use ESLint in your development environment.

AngularJS

Note: This section is intended as a quick intro to contributing with AngularJS. For more detailed information, check the *AngularJS Topic Guide*.

John Papa Style Guide

The John Papa Style Guide is the primary point of reference for Angular code style. This style guide has been endorsed by the AngularJS team:

```
"The most current and detailed Angular Style Guide is the
community-driven effort led by John Papa and Todd Motto."
```

```
- http://angularjs.blogspot.com/2014/02/an-angularjs-style-guide-and-best.html
```

The style guide is found at the below location:

```
https://github.com/johnpapa/angular-styleguide
```

When reviewing / writing, please refer to the sections of this guide. If an issue is encountered, note it with a comment and provide a link back to the specific issue. For example, code should use named functions. A review noting this should provide the following link in the comments:

```
https://github.com/johnpapa/angular-styleguide#style-y024
```

In addition to John Papa, the following guidelines are divided into required and recommended sections.

Required

- Scope is not the model (model is your JavaScript Objects). The scope references the model. Use isolate scopes wherever possible.
 - <https://github.com/angular/angular.js/wiki/Understanding-Scopes>
 - Read-only in templates.
 - Write-only in controllers.
- Since Django already uses {{ }}, use {\$ \$} or {% verbatim %} instead.

ESLint

ESLint is a great tool to be used during your code editing to improve JavaScript quality by checking your code against a configurable list of checks. Therefore, JavaScript developers should configure their editors to use ESLint to warn them of any such errors so they can be addressed. Since ESLint has a ton of configuration options to choose from, links are provided below to the options Horizon wants enforced along with the instructions for setting up ESLint for Eclipse, Sublime Text, Notepad++ and WebStorm/PyCharm.

Instructions for setting up ESLint: [ESLint setup instructions](#)

Note: ESLint is part of the automated unit tests performed by Jenkins. The automated test use the default configurations, which are less strict than the configurations we recommended to run in your local development environment.

CSS

Style guidelines for CSS are currently quite minimal. Do your best to make the code readable and well-organized. Two spaces are preferred for indentation so as to match both the JavaScript and HTML files.

JavaScript and CSS libraries using xstatic

We do not bundle third-party code in Horizons source tree. Instead, we package the required files as xstatic Python packages and add them as dependencies to Horizon.

To create a new xstatic package:

1. Check if the library is already packaged as xstatic on PyPi, by searching for the library name. If it already is, go to step 5. If it is, but not in the right version, contact the original packager to have them update it.
2. Package the library as an xstatic package by following the instructions in [xstatic documentation](#). Install the `xstatic-release` script and follow the instructions that come with it.
3. [Create a new repository under OpenStack](#). Use `xstatic-core` and `xstatic-ptl` groups for the ACLs. Make sure to include the `-pypi-wheel-upload` job in the project config.
4. [Set up PyPi](#) to allow OpenStack (the `openstackci` user) to publish your package.
5. Add the new package to [global-requirements](#).

To make a new release of the package, you need to:

1. Ensure the version information in the `xstatic/pkg/<package name>/__init__.py` file is up to date, especially the `BUILD`.
2. Push your updated package up for review in [gerrit](#).
3. Once the review is approved and the change merged, [request a release](#) by updating or creating the appropriate file for the xstatic package in the [releases repository](#) under `deliverables/_independent`. That will cause it to be automatically packaged and released to PyPi.

Warning: Note that once a package is released, you can not un-release it. You should never attempt to modify, delete or rename a released package without a lot of careful planning and feedback from all projects that use it.

For the purpose of fixing packaging mistakes, xstatic has the build number mechanism. Simply fix the error, increment the build number and release the newer package.

Integrating a new xstatic package into Horizon

Having done a release of an xstatic package:

1. Look for the `upper-constraints.txt` edit related to the xstatic release that was just performed. One will be created automatically by the release process in the `openstack/requirements` project with the topic `new-release`. You should -1 that patch until you are confident Horizon does not break (or you have generated a patch to fix Horizon for that release.) If no `upper-constraints.txt` patch is automatically generated, ensure the `releases` yml file created in the `releases repository` has the `include-pypi-link: yes` setting.
2. Pull that patch down so you have the edited `upper-constraints.txt` file locally.
3. Set the environment variable `TOX_CONSTRAINTS_FILE` to the edited `upper-constraints.txt` file name and run tests or local development server through `tox`. This will pull in the precise version of the xstatic package that you need.
4. Move on to releasing once youre happy the Horizon changes are stable.

Releasing a new compatible version of Horizon to address issues in the new xstatic release:

1. Continue to -1 the `upper-constraints.txt` patch above until this process is complete. A +1 from a Horizon developer will indicate to the requirements team that the `upper-constraints.txt` patch is OK to merge.
2. When submitting your changes to Horizon to address issues around the new xstatic release, use a `Depends-On:` referencing the `upper-constraints.txt` review. This will cause the OpenStack testing infrastructure to pull in your updated xstatic package as well.
3. Merge the `upper-constraints.txt` patch and the Horizon patch noting that Horizons gate may be broken in the interim between these steps, so try to minimise any delay there. With the `Depends-On` its actually safe to +W the Horizon patch, which will be held up until the related `upper-constraints.txt` patch merges.
4. Once the `upper-constraints.txt` patch merges, you should propose a patch to `global-requirements` which bumps the minimum version of the package up to the `upper-constraints` version so that deployers / packagers who dont honor `upper-constraints` still get compatible versions of the packages.

HTML

Again, readability is paramount; however be conscientious of how the browser will handle whitespace when rendering the output. Two spaces is the preferred indentation style to match all front-end code.

Exception Handling

Avoid propogating direct exception messages thrown by OpenStack APIs to the UI. It is a precaution against giving obscure or possibly sensitive data to a user. These error messages from the API are also not translatable. Until there is a standard error handling framework implemented by the services which presents clean and translated messages, horizon catches all the exceptions thrown by the API and normalizes them in `horizon.exceptions.handle()`.

Documentation

Horizons documentation is written in reStructuredText (reST) and uses Sphinx for additional parsing and functionality, and should follow standard practices for writing reST. This includes:

- Flow paragraphs such that lines wrap at 80 characters or less.
- Use proper grammar, spelling, capitalization and punctuation at all times.
- Make use of Sphinxs autodoc feature to document modules, classes and functions. This keeps the docs close to the source.
- Where possible, use Sphinxs cross-reference syntax (e.g. `:class:`~horizon.foo.Bar`) when referring to other Horizon components. The better-linked our docs are, the easier they are to use.

Be sure to generate the documentation before submitting a patch for review. Unexpected warnings often appear when building the documentation, and slight reST syntax errors frequently cause links or cross-references not to work correctly.

Documentation is generated with Sphinx using the tox command. To create HTML docs and man pages:

```
$ tox -e docs
```

The results are in the `doc/build/html` and `doc/build/man` directories respectively.

Conventions

Simply by convention, we have a few rules about naming:

- The term project is used in place of Keystones tenant terminology in all user-facing text. The term tenant is still used in API code to make things more obvious for developers.
- The term dashboard refers to a top-level dashboard class, and panel to the sub-items within a dashboard. Referring to a panel as a dashboard is both confusing and incorrect.

Workflows Topic Guide

One of the most challenging aspects of building a compelling user experience is crafting complex multi-part workflows. Horizons workflows module aims to bring that capability within everyday reach.

See also:

For detailed API information refer to the *Horizon Workflows*.

Workflows

Workflows are complex forms with tabs, each workflow must consist of classes extending the *Workflow*, *Step* and *Action*

Complex example of a workflow

The following is a complex example of how data is exchanged between urls, views, workflows and templates:

1. In `urls.py`, we have the named parameter. E.g. `resource_class_id`.

```
RESOURCE_CLASS = r'^(?P<resource_class_id>[^/]+)/%s$'

urlpatterns = [
    url(RESOURCE_CLASS % 'update', UpdateView.as_view(), name='update')
]
```

2. In `views.py`, we pass data to the template and to the action(form) (action can also pass data to the `get_context_data` method and to the template).

```
class UpdateView(workflows.WorkflowView):
    workflow_class = UpdateResourceClass

    def get_context_data(self, **kwargs):
        context = super(UpdateView, self).get_context_data(**kwargs)
        # Data from URL are always in self.kwargs, here we pass the data
        # to the template.
        context["resource_class_id"] = self.kwargs['resource_class_id']
        # Data contributed by Workflow's Steps are in the
        # context['workflow'].context list. We can use that in the
        # template too.
        return context

    def _get_object(self, *args, **kwargs):
        # Data from URL are always in self.kwargs, we can use them here
        # to load our object of interest.
        resource_class_id = self.kwargs['resource_class_id']
        # Code omitted, this method should return some object obtained
        # from API.

    def get_initial(self):
```

(continues on next page)

(continued from previous page)

```

resource_class = self._get_object()
# This data will be available in the Action's methods and
# Workflow's handle method.
# But only if the steps will depend on them.
return {'resource_class_id': resource_class.id,
        'name': resource_class.name,
        'service_type': resource_class.service_type}

```

3. In `workflows.py` we process the data, it is just more complex django form.

```

class ResourcesAction(workflows.Action):
    # The name field will be automatically available in all action's
    # methods.
    # If we want this field to be used in the another Step or Workflow,
    # it has to be contributed by this step, then depend on in another
    # step.
    name = forms.CharField(max_length=255,
                           label=_("Testing Name"),
                           help_text="")

    def handle(self, request, data):
        pass
        # If we want to use some data from the URL, the Action's step
        # has to depend on them. It's then available in
        # self.initial['resource_class_id'] or data['resource_class_id'].
        # In other words, resource_class_id has to be passed by view's
        # get_initial and listed in step's depends_on list.

        # We can also use here the data from the other steps. If we want
        # the data from the other step, the step needs to contribute the
        # data and the steps needs to be ordered properly.

class UpdateResources(workflows.Step):
    action_class = ResourcesAction
    # This passes data from Workflow context to action methods
    # (handle, clean). Workflow context consists of URL data and data
    # contributed by other steps.
    depends_on = ("resource_class_id",)

    # By contributing, the data on these indexes will become available to
    # Workflow and to other Steps (if they will depend on them). Notice,
    # that the resources_object_ids key has to be manually added in
    # contribute method first.
    contributes = ("resources_object_ids", "name")

    def contribute(self, data, context):
        # We can obtain the http request from workflow.
        request = self.workflow.request
        if data:

```

(continues on next page)

(continued from previous page)

```
# Only fields defined in Action are automatically
# available for contribution. If we want to contribute
# something else, We need to override the contribute method
# and manually add it to the dictionary.
context["resources_object_ids"] =\
    request.POST.getlist("resources_object_ids")

# We have to merge new context with the passed data or let
# the superclass do this.
context.update(data)
return context

class UpdateResourceClass(workflows.Workflow):
    default_steps = (UpdateResources,)

    def handle(self, request, data):
        pass
        # This method is called as last (after all Action's handle
# methods). All data that are listed in step's 'contributes='
# and 'depends_on=' are available here.
# It can be easier to have the saving logic only here if steps
# are heavily connected or complex.

        # data["resources_object_ids"], data["name"] and
# data["resources_class_id"] are available here.
```

DataTables Topic Guide

Horizon provides the *horizon.tables* module to provide a convenient, reusable API for building data-driven displays and interfaces. The core components of this API fall into three categories: DataTables, Actions, and Class-based Views.

See also:

For a detailed API information check out the *Horizon DataTables*.

Tables

The majority of interface in a dashboard-style interface ends up being tabular displays of the various resources the dashboard interacts with. The *DataTable* class exists so you don't have to reinvent the wheel each time.

Creating your own tables

Creating a table is fairly simple:

1. Create a subclass of *DataTable*.
2. Define columns on it using *Column*.
3. Create an inner Meta class to contain the special options for this table.
4. Define any actions for the table, and add them to `table_actions` or `row_actions`.

Examples of this can be found in any of the `tables.py` modules included in the reference modules under `horizon.dashboards`.

Connecting a table to a view

Once you've got your table set up the way you like it, the next step is to wire it up to a view. To make this as easy as possible Horizon provides the *DataTableView* class-based view which can be subclassed to display your table with just a couple lines of code. At its simplest, it looks like this:

```
from horizon import tables
from .tables import MyTable

class MyTableView(tables.DataTableView):
    table_class = MyTable
    template_name = "my_app/my_table_view.html"

    def get_data(self):
        return my_api.objects.list()
```

In the template you would just need to include the following to render the table:

```
{{ table.render }}
```

That's it! Easy, right?

Actions

Actions comprise any manipulations that might happen on the data in the table or the table itself. For example, this may be the standard object CRUD, linking to related views based on the objects id, filtering the data in the table, or fetching updated data when appropriate.

When actions get run

There are two points in the request-response cycle in which actions can take place; prior to data being loaded into the table, and after the data is loaded. When you're using one of the pre-built class-based views for working with your tables the pseudo-workflow looks like this:

1. The request enters view.
2. The table class is instantiated without data.
3. Any preemptive actions are checked to see if they should run.
4. Data is fetched and loaded into the table.
5. All other actions are checked to see if they should run.
6. If none of the actions have caused an early exit from the view, the standard response from the view is returned (usually the rendered table).

The benefit of the multi-step table instantiation is that you can use preemptive actions which don't need access to the entire collection of data to save yourself on processing overhead, API calls, etc.

Basic actions

At their simplest, there are three types of actions: actions which act on the data in the table, actions which link to related resources, and actions that alter which data is displayed. These correspond to *Action*, *LinkAction*, and *FilterAction*.

Writing your own actions generally starts with subclassing one of those action classes and customizing the designated attributes and methods.

Shortcut actions

There are several common tasks for which Horizon provides pre-built shortcut classes. These include *BatchAction*, and *DeleteAction*. Each of these abstracts away nearly all of the boilerplate associated with writing these types of actions and provides consistent error handling, logging, and user-facing interaction.

It is worth noting that *BatchAction* and *DeleteAction* are extensions of the standard *Action* class. Some *BatchAction* or *DeleteAction* classes may cause some unrecoverable results, like deleted images or unrecoverable instances. It may be helpful to specify specific `help_text` to explain the concern to the user, such as Deleted images are not recoverable.

Preemptive actions

Action classes which have their `preempt` attribute set to True will be evaluated before any data is loaded into the table. As such, you must be careful not to rely on any table methods that require data, such as `get_object_display()` or `get_object_by_id()`. The advantage of preemptive actions is that you can avoid having to do all the processing, API calls, etc. associated with loading data into the table for actions which don't require access to that information.

Policy checks on actions

The `policy_rules` attribute, when set, will validate access to the action using the policy rules specified. The attribute is a list of scope/rule pairs. Where the scope is the service type defining the rule and the rule is a rule from the corresponding service policy.json file. The format of `horizon.tables.Action.policy_rules` looks like:

```
((("identity", "identity:get_user"),)
```

Multiple checks can be made for the same action by merely adding more tuples to the list. The policy check will use information stored in the session about the user and the result of `get_policy_target()` (which can be overridden in the derived action class) to determine if the user can execute the action. If the user does not have access to the action, the action is not added to the table.

If `policy_rules` is not set, no policy checks will be made to determine if the action should be visible and will be displayed solely based on the result of `allowed()`.

For more information on policy based Role Based Access Control see *Horizon Policy Enforcement (RBAC: Role Based Access Control)*.

Table Cell filters (decorators)

DataTable displays lists of objects in rows and object attributes in cell. How should we proceed, if we want to decorate some column, e.g. if we have column memory which returns a number e.g. 1024, and we want to show something like 1024.00 GB inside table?

Decorator pattern

The clear anti-pattern is defining the new attributes on object like `ram_float_format_2_gb` or to tweak a DataTable in any way for displaying purposes.

The cleanest way is to use `filters`. Filters are decorators, following GOF Decorator pattern. This way DataTable logic and displayed object logic are correctly separated from presentation logic of the object inside of the various tables. And therefore the filters are reusable in all tables.

Filter function

Horizon DataTablesTable takes a tuple of pointers to filter functions or anonymous lambda functions. When displaying a Cell, DataTable takes Column filter functions from left to right, using the returned value of the previous function as a parameter of the following function. Then displaying the returned value of the last filter function.

A valid filter function takes one parameter and returns the decorated value. So e.g. these are valid filter functions

```
# Filter function.
def add_unit(v):
    return str(v) + " GB"

# Or filter lambda function.
lambda v: str(v) + " GB"

# This is also a valid definition of course, although for the change of the
# unit parameter, function has to be wrapped by lambda
# (e.g. floatformat function example below).
def add_unit(v, unit="GB"):
    return str(v) + " " + unit
```

Using filters in DataTable column

DataTable takes tuple of filter functions, so e.g. this is valid decorating of a value with float format and with unit

```
ram = tables.Column(
    "ram",
    verbose_name=_('Memory'),
    filters=(lambda v: floatformat(v, 2),
            add_unit))
```

It always takes tuple, so using only one filter would look like this

```
filters=(lambda v: floatformat(v, 2),)
```

The decorated parameter doesn't have to be only a string or number, it can be anything e.g. list or an object. So decorating of object, that has attributes value and unit would look like this

```
ram = tables.Column(
    "ram",
    verbose_name=_('Memory'),
    filters=(lambda x: getattr(x, 'value', '') +
            " " + getattr(x, 'unit', '')),))
```

Available filters

There are a load of filters, that can be used, defined in django already: <https://github.com/django/django/blob/master/django/template/defaultfilters.py>

So its enough to just import and use them, e.g.

```
from django.template import defaultfilters as filters

# code omitted
filters=(filters.yesno, filters.capfirst)

from django.template.defaultfilters import timesince
from django.template.defaultfilters import title

# code omitted
filters=(parse_isotime, timesince)
```

Inline editing

Table cells can be easily upgraded with in-line editing. With use of `django.form.Field`, we are able to run validations of the field and correctly parse the data. The updating process is fully encapsulated into table functionality, communication with the server goes through AJAX in JSON format. The javascript wrapper for inline editing allows each table cell that has in-line editing available to:

1. Refresh itself with new data from the server.
2. Display in edit mode.
3. Send changed data to server.
4. Display validation errors.

There are basically 3 things that need to be defined in the table in order to enable in-line editing.

Fetching the row data

Defining an `get_data` method in a class inherited from `tables.Row`. This method takes care of fetching the row data. This class has to be then defined in the table Meta class as `row_class = UpdateRow`.

Example:

```
class UpdateRow(tables.Row):
    # this method is also used for automatic update of the row
    ajax = True

    def get_data(self, request, project_id):
        # getting all data of all row cells
        project_info = api.keystone.tenant_get(request, project_id,
                                              admin=True)

        return project_info
```

Defining a form_field for each Column that we want to be in-line edited.

Form field should be `django.form.Field` instance, so we can use django validations and parsing of the values sent by POST (in example validation `required=True` and correct parsing of the checkbox value from the POST data).

Form field can be also `django.form.Widget` class, if we need to just display the form widget in the table and we dont need Field functionality.

Then connecting `UpdateRow` and `UpdateCell` classes to the table.

Example:

```
class TenantsTable(tables.DataTable):
    # Adding html text input for inline editing, with required validation.
    # HTML form input will have a class attribute tenant-name-input, we
    # can define here any HTML attribute we need.
    name = tables.Column('name', verbose_name=_('Name'),
                        form_field=forms.CharField(),
                        form_field_attributes={'class': 'tenant-name-input'},
                        update_action=UpdateCell)

    # Adding html textarea without required validation.
    description = tables.Column(lambda obj: getattr(obj, 'description', None),
                               verbose_name=_('Description'),
                               form_field=forms.CharField(
                                   widget=forms.Textarea(),
                                   required=False),
                               update_action=UpdateCell)

    # Id will not be inline edited.
    id = tables.Column('id', verbose_name=_('Project ID'))

    # Adding html checkbox, that will be shown inside of the table cell with
    # label
    enabled = tables.Column('enabled', verbose_name=_('Enabled'), status=True,
                           form_field=forms.BooleanField(
                               label=_('Enabled'),
                               required=False),
                           update_action=UpdateCell)

class Meta(object):
    name = "tenants"
    verbose_name = _("Projects")
    # Connection to UpdateRow, so table can fetch row data based on
    # their primary key.
    row_class = UpdateRow
```

Horizon Policy Enforcement (RBAC: Role Based Access Control)

Introduction

Horizon's policy enforcement builds on the `oslo_policy` engine. The basis of which is `openstack_auth/policy.py`. Services in OpenStack use the oslo policy engine to define policy rules to limit access to APIs based primarily on role grants and resource ownership.

The implementation in Horizon is based on copies of policy files found in the services source code.

The service rules files are loaded into the policy engine to determine access rights to actions and service APIs.

Horizon Settings

There are a few settings that must be in place for the Horizon policy engine to work.

- `POLICY_CHECK_FUNCTION`
- `POLICY_DIRS`
- `POLICY_FILES_PATH`
- `POLICY_FILES`
- `DEFAULT_POLICY_FILES`

For more detail, see *Settings Reference*.

How users roles are determined

Each policy check uses information about the user stored on the request to determine the user's roles. This information was extracted from the scoped token received from Keystone when authenticating.

Entity ownership is also a valid role. To verify access to specific entities like a project, the target must be specified. See the section *rule targets* later in this document.

How to Utilize RBAC

Django: Table action

The primary way to add role based access control checks to panels is in the definition of table actions. When implementing a derived action class, setting the `policy_rules` attribute to valid policy rules will force a policy check before the `horizon.tables.Action.allowed()` method is called on the action. These rules are defined in the policy files pointed to by `POLICY_PATH` and `POLICY_FILES`. The rules are role based, where entity owner is also a role. The format for the `policy_rules` is a list of two item tuples. The first component of the tuple is the scope of the policy rule, this is the service type. This informs the policy engine which policy file to reference. The second component is the rule to enforce from the policy file specified by the scope. An example tuple is:

```
("identity", "identity:get_user")
```

x tuples can be added to enforce x rules.

Note: If a rule specified is not found in the policy file, the policy check will return `False` and the action will not be allowed.

Django: policy check function

The secondary way to add a role based check is to directly use the `check()` method. The method takes a list of actions, same format as the `policy_rules` attribute detailed above; the current request object; and a dictionary of action targets. This is the method that `horizon.tables.Action` class utilizes. Examples look like:

```
from openstack_dashboard import policy

allowed = policy.check(("identity", "identity:get_user"),
                      ("identity", "identity:get_project"), request)

can_see = policy.check(("identity", "identity:get_user"), request,
                      target={"domain_id": domainId})
```

Note: Any time multiple rules are specified in a single `policy.check` method call, the result is the logical *and* of each rule check. So, if any rule fails verification, the result is *False*.

Angular: ifAllowed method

The third way to add a role based check is in javascript files. Use the method `ifAllowed()` in file `openstack_dashboard.static.app.core.policy.service.js`. The method takes a list of actions, similar format with the `policy_rules` attribute detailed above. An Example looks like:

```
angular
.module('horizon.dashboard.identity.users')
.controller('identityUsersTableController', identityUsersTableController);

identityUsersTableController.$inject = [
  'horizon.app.core.openstack-service-api.policy',
];

function identityUsersTableController(toast, gettext, policy, keystone) {
  var rules = [['identity', 'identity:list_users']];
  policy.ifAllowed({ rules: rules }).then(policySuccess, policyFailed);
}
```


Angular: hz-if-policies

The fourth way to add a role based check is in html files. Use angular directive `hz-if-policies` in file `openstack_dashboard/static/app/core/cloud-services/hz-if-policies.directive.js`. Assume you have the following policy defined in your angular controller:

```
ctrl.policy = { rules: [{"identity", "identity:update_user"}] }
```

Then in your HTML, use it like so:

```
<div hz-if-policies='ctrl.policy'>
  <span>I am visible if the policy is allowed!</span>
</div>
```

Rule Targets

Some rules allow access if the user owns the entity. Policy check targets specify particular entities to check for user ownership. The target parameter to the `check()` method is a simple dictionary. For instance, the target for checking access a project looks like:

```
{"project_id": "0905760626534a74979afd3f4a9d67f1"}
```

If the value matches the `project_id` to which the users token is scoped, then access is allowed.

When deriving the `horizon.tables.Action` class for use in a table, if a policy check is desired for a particular target, the implementer should override the `horizon.tables.Action.get_policy_target()` method. This allows a programmatic way to specify the target based on the current datum. The value returned should be the target dictionary.

Policy-in-Code and deprecated rules

As the effort of [policy-in-code](#), most OpenStack projects define their default policies in their codes. All projects (except swift) covered by horizon supports policy-in-code. (Note that swift is an exception as it has its own mechanism to control RBAC.)

`oslo.policy` provides a way to deprecate existing policy rules like renaming rule definitions (`check_str`) and renaming rule names. They are defined as part of python codes in back-end services. horizon cannot import python codes of back-end services, so we need a way to restore policies defined by policy-in-code including deprecated rules.

To address the above issue, horizon adopts the following two-step approach:

- The first step scans policy-in-code of back-end services and dump the loaded default policies into YAML files per service including information of deprecated rules. This step is executed as part of the development process per release cycle and these YAML files are shipped per release.

Note that `oslopolicy-sample-generator` does not output deprecated rules in a structured way, so we prepare a dedicated script for this purpose in the horizon repo.

- The horizon policy implementation loads the above YAML file into a list of `RuleDefault` and registers the list as the default rules to the policy enforcer. The default rules and operator-defined rules are maintained separately, so operators still can edit the policy files as `oslo.policy` does in back-end services.

This approach has the following merits:

- All features supported by `oslo.policy` can be supported in horizon as default rules in back-end services are restored as-is. Horizon can evaluate deprecated rules.
- The default rules and operator defined rules are maintained separately. Operators can use the same way to maintain policy files of back-end services.

The related files in the horizon codebase are:

- `openstack_dashboard/conf/<service>_policy.yaml`: operator-defined policies. These files are generated by `oslopolicy-sample-generator`.
- `openstack_dashboard/conf/default_policies/<service>.yaml` YAML files contain default policies.
- `openstack_dashboard/management/commands/dump_default_policies.py`: This script scans policy-in-code of a specified namespace under `oslo.policy.policies` entrypoints and dump them into the YAML file under `openstack_dashboard/conf/default_policies`.
- `openstack_auth/policy.py`: `_load_default_rules` function loads the YAML files with default rules and call `register_defaults` method of the policy enforcer per service.

Policy file maintenance

- YAML files for default policies

Run the following command after installing a corresponding project. You need to run it for key-stone, nova, cinder, neutron, glance.

```
python3 manage.py dump_default_policies \
  --namespace $PROJECT \
  --output-file openstack_dashboard/conf/default_policies/${PROJECT}.yaml
```

- Sample policy files

Run the following commands after installing a corresponding project. You need to run it for key-stone, nova, cinder, neutron, glance.

```
oslopolicy-sample-generator --namespace $PROJECT \
  --output-file openstack_dashboard/conf/${PROJECT}_policy.yaml
sed -i 's/^"/#"/' openstack_dashboard/conf/${PROJECT}_policy.yaml
```

Note: We now use YAML format for sample policy files now. `oslo.policy` can accept both YAML and JSON files. We now support default policies so there is no need to define all policies using JSON files. YAML files also allows us to use comments, so we can provide good sample policy files. This is the same motivation as the Wallaby community goal [Migrate RBAC Policy Format from JSON to YAML](#).

Note: The second `sed` command is to comment out rules for rule renames. `oslopolicy-sample-generator` does not comment out them, but they are unnecessary in horizon usage. A single renaming rule can map to multiple rules, so it does not work as-is. In addition, they trigger deprecation

warnings in horizon log if these sample files are used in horizon as-is. Thus, we comment them out by default.

After syncing policies from back-end services, you need to check what are changed. If a policy referred by horizon has been changed, you need to check and modify the horizon code base accordingly.

Warning: After the support of default policies, the following tool does not work. It is a future work to make it work again or evaluate the need itself.

To summarize which policies are removed or added, a convenient tool is provided:

```
$ cd openstack_dashboard/conf/
$ python ../../tools/policy-diff.py --help
usage: policy-diff.py [-h] --old OLD --new NEW [--mode {add,remove}]

optional arguments:
-h, --help            show this help message and exit
--old OLD             Current policy file
--new NEW            New policy file
--mode {add,remove}  Diffs to be shown

# Show removed policies
# The default is "--mode remove". You can omit --mode option.
$ python ../../tools/policy-diff.py \
    --old keystone_policy.json --new keystone_policy.json.new --mode remove
default
identity:change_password
identity:get_identity_provider
```

Horizon Microversion Support

Introduction

Several services use API microversions, which allows consumers of that API to specify an exact version when making a request. This can be useful in ensuring a feature continues to work as expected across many service releases.

Adding a feature that was introduced in a microversion

1. Add the feature to the MICROVERSION_FEATURES dict in `openstack_dashboard/api/microversions.py` under the appropriate service name. The feature should have at least two versions listed; the minimum version (i.e. the version that introduced the feature) and the current working version. Providing multiple versions reduces project maintenance overheads and helps Horizon work with older service deployments.
2. Use the `is_feature_available` function for your service to show or hide the function.:

```
from openstack_dashboard.api import service

...

def allowed(self, request):
    return service.is_feature_available('feature')
```

3. Send the correct microversion with `get_microversion` function in the API layer.:

```
def resource_list(request):
    try:
        microversion = get_microversion(request, 'feature')
        client = serviceclient(request, microversion)
        return client.resource_list()
```

Microversion references

Nova <https://docs.openstack.org/nova/latest/reference/api-microversion-history.html>

Cinder https://docs.openstack.org/cinder/latest/contributor/api_microversion_history.html

API-WG https://specs.openstack.org/openstack/api-wg/guidelines/microversion_specification.html

AngularJS Topic Guide

Note: This guide is a work in progress. It has been uploaded to encourage faster reviewing and code development in Angular, and to help the community standardize on a set of guidelines. There are notes inline on sections that are likely to change soon, and the docs will be updated promptly after any changes.

Getting Started

The tooling for AngularJS testing and code linting relies on npm, the node package manager, and thus relies on Node.js. While it is not a prerequisite to developing with Horizon, it is advisable to install Node.js, either through [downloading](#) or via a [package manager](#).

Once you have npm available on your system, run `npm install` from the horizon root directory.

Code Style

We currently use the [Angular Style Guide](#) by John Papa as reference material. When reviewing AngularJS code, it is helpful to link directly to the style guide to reinforce a point, e.g. <https://github.com/johnpapa/angular-styleguide#style-y024>

ESLint

ESLint is a tool for identifying and reporting on patterns in your JS code, and is part of the automated tests run by Jenkins. You can run ESLint from the horizon root directory with `tox -e npm -- lint`, or alternatively on a specific directory or file with `eslint file.js`.

Horizon includes a `.eslintrc` in its root directory, that is used by the local tests. An explanation of the options, and details of others you may want to use, can be found in the [ESLint user guide](#).

Application Structure

OpenStack Dashboard is an example of a Horizon-based Angular application. Other applications built on the Horizon framework can follow a similar structure. It is composed of two key Angular modules:

app.module.js - The root of the application. Defines the modules required by the application, and includes modules from its pluggable dashboards.

framework.module.js - Reusable Horizon components. It is one of the application dependencies.

File Structure

Horizon has three kinds of angular code:

1. Specific to one dashboard in the OpenStack Dashboard application
2. Specific to the OpenStack Dashboard application, but reusable by multiple dashboards
3. Reusable by any application based on the Horizon framework

When adding code to horizon, consider whether it is dashboard-specific or should be broken out as a reusable utility or widget.

Code specific to one dashboard

Code that isn't shared beyond a single dashboard is placed in `openstack_dashboard/dashboards/mydashboard/static`. Entire dashboards may be enabled or disabled using Horizons plugin mechanism. Therefore no dashboards other than `mydashboard` can safely use this code.

The `openstack_dashboard/dashboards/mydashboard/static` directory structure determines how the code is deployed and matches the module structure. For example:

```
openstack_dashboard/dashboards/identity/static/dashboard/identity/  
identity.module.js  
identity.module.spec.js  
identity.scss
```

Because the code is in `openstack_dashboard/dashboards/identity` we know it is specific to just the `identity` dashboard and not used by any others.

Code shared by multiple dashboards

Views or utilities needed by multiple dashboards are placed in `openstack_dashboard/static/app`. For example:

```
openstack_dashboard/static/app/core/cloud-services/  
cloud-services.module.js  
cloud-services.spec.js  
hz-if-settings.directive.js  
hz-if-settings.directive.spec.js
```

The `cloud-services` module is used by panels in multiple dashboards. It cannot be placed within `openstack_dashboard/dashboards/mydashboard` because disabling that one dashboard would break others. Therefore, it is included as part of the application core module. Code in `app/` is guaranteed to always be present, even if all other dashboards are disabled.

Reusable components

Finally, components that are easily reused by any application are placed in `horizon/static/framework/`. These do not contain URLs or business logic that is specific to any application (even the OpenStack Dashboard application).

The modal directive `horizon/static/framework/widgets/modal/` is a good example of a reusable component.

One folder per component

Each component should have its own folder, with the code broken up into one JS component per file. (See [Single Responsibility](#) in the style guide). Each folder may include styling (`*.scss`), as well as templates (`*.html`) and tests (`*.spec.js`). You may also include examples, by appending `.example`.

For larger components, such as workflows with multiple steps, consider breaking the code down further. For example, the Launch Instance workflow, has one directory per step. See `openstack_dashboard/dashboards/project/static/dashboard/project/workflow/launch-instance/`

SCSS files

The top-level SCSS file in `openstack_dashboard/static/app/_app.scss`. It includes any styling that is part of the application core and may be reused by multiple dashboards. SCSS files that are specific to a particular dashboard are linked to the application by adding them in that dashboard's enabled file. For example, `_1920_project_containers_panel.py` is the enabled file for the Project dashboard's Container panel and includes:

```
ADD_SCSS_FILES = [  
    'dashboard/project/containers/_containers.scss',  
]
```

Styling files are hierarchical, and include any direct child SCSS files. For example, `project.scss` would include the `workflow` SCSS file, which in turn includes any launch instance styling:

```
@import "workflow/workflow";
```

This allows the application to easily include all needed styling, simply by including a dashboard's top-level SCSS file.

Module Structure

Horizon Angular modules use names that map to the source code directory structure. This provides namespace isolation for modules and services, which makes dependency injection clearer. It also reduces code conflicts where two different modules define a module, service or constant of the same name. For example:

```
openstack_dashboard/dashboards/identity/static/dashboard/identity/  
identity.module.js
```

The preferred Angular module name in this example is `horizon.dashboard.identity`. The `horizon` part of the module name maps to the `static` directory and indicates this is a `horizon` based application. `dashboard.identity` maps to folders that are created within `static`. This allows a direct mapping between the angular module name of `horizon.dashboard.identity` and the source code directory of `static\dashboard\identity`.

Services and constants within these modules should all start with their module name to avoid dependency injection collisions. For example:

```
$provide.constant('horizon.dashboard.identity.basePath', path);
```

Directives do not require the module name but are encouraged to begin with the `hz` prefix. For example:

```
.directive('hzMagicSearchBar', hzMagicSearchBar);
```

Finally, each module lists its child modules as a dependency. This allows the root module to be included by an application, which will automatically define all child modules. For example:

```
.module('horizon.framework', [  
  'horizon.framework.conf',  
  'horizon.framework.util',  
  'horizon.framework.widgets'  
)
```

`horizon.framework` declares a dependency on `horizon.framework.widgets`, which declares dependencies on each individual widget. This allows the application to access any widget, simply by depending on the top-level `horizon.framework` module.

Testing

1. Open `<dev_server_ip:port>/jasmine` in a browser. The development server can be run with `tox -e runserver` from the horizon root directory; by default, this will run the development server at `http://localhost:8000`.
2. `tox -e npm` from the horizon root directory.

The code linting job can be run with `tox -e npm -- lint`. If there are many warnings, you can also use `tox -e npm -- lintq` to see only errors and ignore warnings.

For more detailed information, see *JavaScript Testing*.

Translation (Internationalization and Localization)

See *Making strings translatable* for information on the translation architecture and how to ensure your code is translatable.

Creating your own panel

Note: This section will be extended as standard practices are adopted upstream. Currently, it may be useful to look at the Project Images Panel as a complete reference. Since Newton, it is Angular by default (set to True in the `ANGULAR_FEATURES` dict in `settings.py`). You may track all the changes made to the Image Panel [here](#)

Note: Currently, Angular module names must still be manually declared with `ADD_ANGULAR_MODULES`, even when using automatic file discovery.

This section serves as a basic introduction to writing your own panel for horizon, using AngularJS. A panel may be included with the plugin system, or it may be part of the upstream horizon project.

Upstream

JavaScript files can be discovered automatically, handled manually, or a mix of the two. Where possible, use the automated mechanism. To use the automatic functionality, add:

```
AUTO_DISCOVER_STATIC_FILES = True
```

to your enabled file (`enabled/<plugin_name>.py`). To make this possible, you need to follow some structural conventions:

- Static files should be put in a `static/` folder, which should be found directly under the folder for the dashboard/panel/panel groups Python package.
- JS code that defines an Angular module should be in a file with extension of `.module.js`.
- JS code for testing should be named with extension of `.mock.js` and of `.spec.js`.
- Angular templates should have extension of `.html`.

You can read more about the functionality in the [AUTO_DISCOVER_STATIC_FILES](#) section of the settings documentation.

To manually add files, add the following arrays and file paths to the enabled file:

```
ADD_JS_FILES = [
    ...
    'path-to/my-angular-code.js',
    ...
]

ADD_JS_SPEC_FILES = [
    ...
    'path-to/my-angular-code.spec.js',
    ...
]

ADD_ANGULAR_MODULES = [
    ...
    'my.angular.code',
    ...
]
```

Plugins

Add a new panel/ panel group/ dashboard (See [Tutorial: Building a Dashboard using Horizon](#)). JavaScript file inclusion is the same as the Upstream process.

To include external stylesheets, you must ensure that `ADD_SCSS_FILES` is defined in your enabled file, and add the relevant filepath, as below:

```
ADD_SCSS_FILES = [
    ...
    'path-to/my-styles.scss',
    ...
]
```

Note: We highly recommend using a single SCSS file for your plugin. SCSS supports nesting with `@import`, so if you have multiple files (i.e. per panel styling) it is best to import them all into one, and include that single file. You can read more in the [SASS documentation](#).

Schema Forms

JSON schemas are used to define model layout and then `angular-schema-form` is used to create forms from that schema. Horizon adds some functionality on top of that to make things even easier through `ModalFormService` which will open a modal with the form inside.

A very simple example:

```
var schema = {
  type: "object",
  properties: {
    name: { type: "string", minLength: 2, title: "Name", description: "Name_
→or alias" },
    title: {
      type: "string",
      enum: ['dr', 'jr', 'sir', 'mrs', 'mr', 'NaN', 'dj']
    }
  }
};
var model = {name: '', title: ''};
var config = {
  title: gettext('Create Container'),
  schema: schema,
  form: ['*'],
  model: model
};
ModalFormService.open(config).then(submit); // returns a promise

function submit() {
  // do something with model.name and model.title
}
```

Testing Overview

Having good tests in place is absolutely critical for ensuring a stable, maintainable codebase. Hopefully that doesn't need any more explanation.

However, what defines a good test is not always obvious, and there are a lot of common pitfalls that can easily shoot your test suite in the foot.

If you already know everything about testing but are fed up with trying to debug why a specific test failed, you can skip the intro and jump straight to *Debugging Unit Tests*.

JavaScript Testing

There are multiple components in our JavaScript testing framework:

- **Jasmine** is our testing framework, so this defines the syntax and file structure we use to test our JavaScript.
- **Karma** is our test runner. Amongst other things, this lets us run the tests against multiple browsers and generate test coverage reports. Alternatively, tests can be run inside the browser with the Jasmine spec runner.
- **PhantomJS** provides a headless WebKit (the browser engine). This gives us native support for many web features without relying on specific browsers being installed.
- **ESLint** is a pluggable code linting utility. This will catch small errors and inconsistencies in your JS, which may lead to bigger issues later on. See *Code Style* for more detail.

Jasmine uses specs (`.spec.js`) which are kept with the JavaScript files that they are testing. See the *File Structure* section or the *Examples* below for more detail on this.

Running Tests

Tests can be run in two ways:

1. Open `<dev_server_ip:port>/jasmine` in a browser. The development server can be run with `tox -e runserver` from the horizon root directory.
2. `tox -e npm` from the horizon root directory. This runs Karma, so it will run all the tests against PhantomJS and generate coverage reports.

The code linting job can be run with `tox -e npm -- lint`, or `tox -e npm -- lintq` to show errors, but not warnings.

To decipher where tests are failing it may be useful to use Jasmine in the browser to run individual tests to see where the tests are specifically breaking. To do this, navigate to your local horizon in the browser and add `/jasmine` to the end of the url. e.g: `http://localhost:8000/jasmine`. Once you have the jasmine report you may click on the title of an individual test to re-run just that test. From here, you can also use chrome dev tools or similar to set breakpoints in the code by accessing the Sources tab and clicking on lines of code where you wish to break the code. This will then show you the exact places where the code breaks.

Coverage Reports

Our Karma setup includes a plugin to generate test coverage reports. When developing, be sure to check the coverage reports on the master branch and compare your development branch; this will help identify missing tests.

To generate coverage reports, run `tox -e npm`. The coverage reports can be found at `cover/horizon/` (framework tests) and `cover/openstack_dashboard/` (dashboard tests). Load `<browser>/index.html` in a browser to view the reports.

Writing Tests

Jasmine uses suites and specs:

- Suites begin with a call to `describe`, which takes two parameters; a string and a function. The string is a name or title for the spec suite, whilst the function is a block that implements the suite.
- Specs begin with a call to `it`, which also takes a string and a function as parameters. The string is a name or title, whilst the function is a block with one or more expectations (`expect`) that test the state of the code. An expectation in Jasmine is an assertion that is either true or false; every expectation in a spec must be true for the spec to pass.

`.spec.js` files can be handled manually or automatically. To use the automatic file discovery add:

```
AUTO_DISCOVER_STATIC_FILES = True
```

to your enabled file. JS code for testing should use the extensions `.mock.js` and `.spec.js`.

You can read more about the functionality in the [AUTO_DISCOVER_STATIC_FILES](#) section of the settings documentation.

To manually add specs, add the following array and relevant file paths to your enabled file:

```
ADD_JS_SPEC_FILES = [  
    ...  
    'path_to/my_angular_code.spec.js',  
    ...  
]
```

Examples

Note: The code below is just for example purposes, and may not be current in horizon. Ellipses () are used to represent code that has been removed for the sake of brevity.

Example 1 - A reusable component in the horizon directory

File tree:

```
horizon/static/framework/widgets/modal  
modal.controller.js  
modal.module.js  
modal.service.js  
modal.spec.js
```

Lines added to `horizon/test/jasmine/jasmine_tests.py`:

```
class ServicesTests(test.JasmineTests):  
    sources = [  
        ...
```

(continues on next page)

(continued from previous page)

```
'framework/widgets/modal/modal.module.js',
'framework/widgets/modal/modal.controller.js',
'framework/widgets/modal/modal.service.js',
...
]

specs = [
...
'framework/widgets/modal/modal.spec.js',
...
]
```

modal.spec.js:

```
...

(function() {
  "use strict";

  describe('horizon.framework.widgets.modal module', function() {

    beforeEach(module('horizon.framework'));

    describe('simpleModalCtrl', function() {
      var scope;
      var modalInstance;
      var context;
      var ctrl;

      beforeEach(inject(function($controller) {
        scope = {};
        modalInstance = {
          close: function() {},
          dismiss: function() {}
        };
        context = { what: 'is it' };
        ctrl = $controller('simpleModalCtrl', {
          $scope: scope,
          $modalInstance: modalInstance,
          context: context
        });
      }));

      it('establishes a controller', function() {
        expect(ctrl).toBeDefined();
      });

      it('sets context on the scope', function() {
        expect(scope.context).toBeDefined();
      });
    });
  });
});
```

(continues on next page)

(continued from previous page)

```
    expect(scope.context).toEqual({ what: 'is it' });
  });

  it('sets action functions', function() {
    expect(scope.submit).toBeDefined();
    expect(scope.cancel).toBeDefined();
  });

  it('makes submit close the modal instance', function() {
    expect(scope.submit).toBeDefined();
    spyOn(modalInstance, 'close');
    scope.submit();
    expect(modalInstance.close.calls.count()).toBe(1);
  });

  it('makes cancel close the modal instance', function() {
    expect(scope.cancel).toBeDefined();
    spyOn(modalInstance, 'dismiss');
    scope.cancel();
    expect(modalInstance.dismiss).toHaveBeenCalled('cancel');
  });
});
...
});
})();
```

Example 2 - Panel-specific code in the openstack_dashboard directory

File tree:

```
openstack_dashboard/static/dashboard/launch-instance/network/
network.help.html
network.html
network.js
network.scss
network.spec.js
```

Lines added to openstack_dashboard/enabled/_10_project.py:

```
LAUNCH_INST = 'dashboard/launch-instance/'

ADD_JS_FILES = [
    ...
    LAUNCH_INST + 'network/network.js',
    ...
]
```

(continues on next page)

(continued from previous page)

```

ADD_JS_SPEC_FILES = [
    ...
    LAUNCH_INST + 'network/network.spec.js',
    ...
]

```

network.spec.js:

```

...

(function(){
    'use strict';

    describe('Launch Instance Network Step', function() {

        describe('LaunchInstanceNetworkCtrl', function() {
            var scope;
            var ctrl;

            beforeEach(module('horizon.dashboard.project.workflow.launch-instance
↪'));

            beforeEach(inject(function($controller) {
                scope = {
                    model: {
                        newInstanceSpec: {networks: ['net-a']},
                        networks: ['net-a', 'net-b']
                    }
                };
                ctrl = $controller('LaunchInstanceNetworkCtrl', {$scope:scope});
            }));

            it('has correct network statuses', function() {
                expect(ctrl.networkStatuses).toBeDefined();
                expect(ctrl.networkStatuses.ACTIVE).toBeDefined();
                expect(ctrl.networkStatuses.DOWN).toBeDefined();
                expect(Object.keys(ctrl.networkStatuses).length).toBe(2);
            });

            it('has correct network admin states', function() {
                expect(ctrl.networkAdminStates).toBeDefined();
                expect(ctrl.networkAdminStates.UP).toBeDefined();
                expect(ctrl.networkAdminStates.DOWN).toBeDefined();
                expect(Object.keys(ctrl.networkAdminStates).length).toBe(2);
            });

            it('defines a multiple-allocation table', function() {
                expect(ctrl.tableLimits).toBeDefined();
            });
        });
    });
}());

```

(continues on next page)

(continued from previous page)

```
    expect(ctrl.tableLimits.maxAllocation).toBe(-1);
  });

  it('contains its own labels', function() {
    expect(ctrl.label).toBeDefined();
    expect(Object.keys(ctrl.label).length).toBeGreaterThan(0);
  });

  it('contains help text for the table', function() {
    expect(ctrl.tableHelpText).toBeDefined();
    expect(ctrl.tableHelpText.allocHelpText).toBeDefined();
    expect(ctrl.tableHelpText.availHelpText).toBeDefined();
  });

  it('uses scope to set table data', function() {
    expect(ctrl.tableDataMulti).toBeDefined();
    expect(ctrl.tableDataMulti.available).toEqual(['net-a', 'net-b']);
    expect(ctrl.tableDataMulti.allocated).toEqual(['net-a']);
    expect(ctrl.tableDataMulti.displayedAllocated).toEqual([]);
    expect(ctrl.tableDataMulti.displayedAvailable).toEqual([]);
  });
});
...
});
});
```

An overview of testing

There are three main types of tests, each with their associated pros and cons:

Unit tests

These are isolated, stand-alone tests with no external dependencies. They are written from the perspective of knowing the code, and test the assumptions of the codebase and the developer.

Pros:

- Generally lightweight and fast.
- Can be run anywhere, anytime since they have no external dependencies.

Cons:

- Easy to be lax in writing them, or lazy in constructing them.
- Cant test interactions with live external services.

Functional tests

These are generally also isolated tests, though sometimes they may interact with other services running locally. The key difference between functional tests and unit tests, however, is that functional tests are written from the perspective of the user (who knows nothing about the code) and only knows what they put in and what they get back. Essentially this is a higher-level testing of does the result match the spec?.

Pros:

- Ensures that your code *always* meets the stated functional requirements.
- Verifies things from an end user perspective, which helps to ensure a high-quality experience.
- Designing your code with a functional testing perspective in mind helps keep a higher-level viewpoint in mind.

Cons:

- Requires an additional layer of thinking to define functional requirements in terms of inputs and outputs.
- Often requires writing a separate set of tests and/or using a different testing framework from your unit tests.
- Doesn't offer any insight into the quality or status of the underlying code, only verifies that it works or it doesn't.

Integration Tests

This layer of testing involves testing all of the components that your codebase interacts with or relies on in conjunction. This is equivalent to live testing, but in a repeatable manner.

Pros:

- Catches *many* bugs that unit and functional tests will not.
- Doesn't rely on assumptions about the inputs and outputs.
- Will warn you when changes in external components break your code.
- Will take screenshot of the current page on test fail for easy debug

Cons:

- Difficult and time-consuming to create a repeatable test environment.
- Did I mention that setting it up is a pain?

Screenshot directory could be set through horizon.conf file, default value: `./integration_tests_screenshots`

So what should I write?

A few simple guidelines:

1. Every bug fix should have a regression test. Period.
2. When writing a new feature, think about writing unit tests to verify the behavior step-by-step as you write the feature. Every time you'd go to run your code by hand and verify it manually, think could I write a test to do this instead?. That way when the feature is done and you're ready to commit it you've already got a whole set of tests that are more thorough than anything you'd write after the fact.
3. Write tests that hit every view in your application. Even if they don't assert a single thing about the code, it tells you that your users aren't getting fatal errors just by interacting with your code.

What makes a good unit test?

Limiting our focus just to unit tests, there are a number of things you can do to make your unit tests as useful, maintainable, and unburdensome as possible.

Test data

Use a single, consistent set of test data. Grow it over time, but do everything you can not to fragment it. It quickly becomes unmaintainable and perniciously out-of-sync with reality.

Make your test data as accurate to reality as possible. Supply *all* the attributes of an object, provide objects in all the various states you may want to test.

If you do the first suggestion above *first* it makes the second one far less painful. Write once, use everywhere.

To make your life even easier, if your codebase doesn't have a built-in ORM-like function to manage your test data you can consider building (or borrowing) one yourself. Being able to do simple retrieval queries on your test data is incredibly valuable.

Mocking

Mocking is the practice of providing stand-ins for objects or pieces of code you don't need to test. While convenient, they should be used with *extreme* caution.

Why? Because overuse of mocks can rapidly land you in a situation where you're not testing any real code. All you've done is verified that your mocking framework returns what you tell it to. This problem can be very tricky to recognize, since you may be mocking things in `setUp` methods, other modules, etc.

A good rule of thumb is to mock as close to the source as possible. If you have a function call that calls an external API in a view, mock out the external API, not the whole function. If you mock the whole function you've suddenly lost test coverage for an entire chunk of code *inside* your codebase. Cut the ties cleanly right where your system ends and the external world begins.

Similarly, don't mock return values when you could construct a real return value of the correct type with the correct attributes. You're just adding another point of potential failure by exercising your mocking framework instead of real code. Following the suggestions for testing above will make this a lot less burdensome.

Assertions and verification

Think long and hard about what you really want to verify in your unit test. In particular, think about what custom logic your code executes.

A common pitfall is to take a known test object, pass it through your code, and then verify the properties of that object on the output. This is all well and good, except if you're verifying properties that were untouched by your code. What you want to check are the pieces that were *changed*, *added*, or *removed*. Don't check the object's id attribute unless you have reason to suspect it's not the object you started with. But if you added a new attribute to it, be damn sure you verify that came out right.

It's also very common to avoid testing things you really care about because it's more difficult. Verifying that the proper messages were displayed to the user after an action, testing for form errors, making sure exception handling is tested these types of things aren't always easy, but they're extremely necessary.

To that end, Horizon includes several custom assertions to make these tasks easier. `assertNoFormErrors()`, `assertMessageCount()`, and `assertNoMessages()` all exist for exactly these purposes. Moreover, they provide useful output when things go wrong so you're not left scratching your head wondering why your view test didn't redirect as expected when you posted a form.

Debugging Unit Tests

Tips and tricks

1. Use `assertNoFormErrors()` immediately after your `client.post` call for tests that handle form views. This will immediately fail if your form POST failed due to a validation error and tell you what the error was.
2. Use `assertMessageCount()` and `assertNoMessages()` when a piece of code is failing inexplicably. Since the core error handlers attach user-facing error messages (and since the core logging is silenced during test runs) these methods give you the dual benefit of verifying the output you expect while clearly showing you the problematic error message if they fail.
3. Use Python's `pdb` module liberally. Many people don't realize it works just as well in a test case as it does in a live view. Simply inserting `import pdb; pdb.set_trace()` anywhere in your codebase will drop the interpreter into an interactive shell so you can explore your test environment and see which of your assumptions about the code isn't, in fact, flawlessly correct.
4. If the error is in the Selenium test suite, you're likely getting very little information about the error. To increase the information provided to you, edit `horizon/test/settings.py` to set `DEBUG = True` and set the logging level to `DEBUG` for the default test logger. Also, add a logger config for Django:

```
    },
    'loggers': {
+     'django': {
+         'handlers': ['test'],
+         'propagate': False,
+     },
    'django.db.backends': {
```

Testing with different Django versions

Horizon supports multiple Django versions and our CI tests proposed patches with various supported Django versions. The corresponding job names are like `horizon-tox-python3-django111`.

You can know which tox env and django version are used by checking `tox_envlist` and `django_version` of the corresponding job definition in `.zuul.yaml`.

To test it locally, you need some extra steps. Here is an example where `tox_envlist` is `py36` and `django_version` is `>=1.11,<2.0`.

```
$ tox -e py36 --notest -r
$ .tox/py36/bin/python -m pip install 'django>=1.11,<2.0'
$ tox -e py36
```

Note:

- `-r` in the first command recreates the tox environment. Omit it if you know what happens.
 - We usually need to quote the django version in the pip command-line in most shells to escape interpretations by the shell.
-

To check the django version installed in your tox env, run:

```
$ .tox/py36/bin/python -m pip freeze | grep Django
Django==1.11.27
```

To reset the tox env used for testing with different Django version to the regular tox env, run `tox` command with `-r` to recreate it.

```
$ tox -e py36 -r
```

Coverage reports

It is possible for tests to fail on your patch due to the `npm-run-test` not passing the minimum threshold. This is not necessarily related directly to the functions in the patch that have failed, but more that there are not enough tests across horizon that are related to your patch.

The coverage reports may be found in the `cover` directory. There's a subdirectory for horizon and `openstack_dashboard`, and then under a directory for the browser used to run the tests you should find an `index.html`. This can then be viewed to see the coverage details.

In this scenario you may need to submit a secondary patch to address test coverage for another function within horizon to ensure tests rise above the coverage threshold and your original patch can pass the necessary tests.

Common pitfalls

There are a number of typical (and non-obvious) ways to break the unit tests. Some common things to look for:

1. Make sure you stub out the method exactly as its called in the code being tested. For example, if your real code calls `api.keystone.tenant_get`, stubbing out `api.tenant_get` (available for legacy reasons) will fail.
2. When defining the expected input to a stubbed call, make sure the arguments are *identical*, this includes `str` vs. `int` differences.
3. Make sure your test data are completely in line with the expected inputs. Again, `str` vs. `int` or missing properties on test objects will kill your tests.
4. Make sure theres nothing amiss in your templates (particularly the `{% url %}` tag and its arguments). This often comes up when refactoring views or renaming context variables. It can easily result in errors that you might not stumble across while clicking around the development server.
5. Make sure youre not redirecting to views that no longer exist, e.g. the `index` view for a panel that got combined (such as instances & volumes).
6. Make sure you repeat any stubbed out method calls that happen more than once. They dont automatically repeat, you have to explicitly define them. While this is a nuisance, it makes you acutely aware of how many API calls are involved in a particular function.

Styling in Horizon (SCSS)

Horizon uses [SCSS](#) (not to be confused with [Sass](#)) to style its HTML. This guide is targeted at developers adding code to upstream Horizon. For information on creating your own branding/theming, see [Customizing Horizon](#).

Code Layout

The base SCSS can be found at `openstack_dashboard/static/dashboard/scss/`. This directory should **only** contain the minimal styling for functionality code that isnt configurable by themes. `horizon.scss` is a top level file that imports from the `components/` directory, as well as other base styling files; potentially some basic page layout rules that Horizon relies on to function.

Note: Currently, a great deal of theming is also kept in the `horizon.scss` file in this directory, but that will be reduced as we proceed with the new code design.

Horizons default theme stylesheets can be found at `openstack_dashboard/themes/default/`.

```
_styles.scss
_variables.scss
bootstrap/
...
horizon/
...
```

The top level `_styles.scss` and `_variables.scss` contain imports from the `bootstrap` and `horizon` directories.

The bootstrap directory

This directory contains overrides and customization of Bootstrap variables, and markup used by Bootstrap components. This should **only** override existing Bootstrap content. For examples of these components, see the *Theme Preview Panel*.

```
bootstrap/  
_styles.scss  
_variables.scss  
components/  
  _component_0.scss  
  _component_1.scss  
  ...
```

- `_styles.scss` imports the SCSS defined for each component.
- `_variables.scss` contains the definitions for every Bootstrap variable. These variables can be altered to affect the look and feel of Horizons default theme.
- The `components` directory contains overrides for Bootstrap components, such as tables or navbars.

The horizon directory

This directory contains SCSS that is absolutely specific to Horizon; code here should **not** override existing Bootstrap content, such as variables and rules.

```
horizon/  
_styles.scss  
_variables.scss  
components/  
  _component_0.scss  
  _component_1.scss  
  ...
```

- `_styles.scss` imports the SCSS defined for each component. It may also contain some minor styling overrides.
- `_variables.scss` contains variable definitions that are specific to the horizon theme. This should **not** override any bootstrap variables, only define new ones. You can however, inherit bootstrap variables for reuse (and are encouraged to do so where possible).
- The `components` directory contains styling for each individual component defined by Horizon, such as the sidebar or pie charts.

Adding new SCSS

To keep Horizon easily themable, there are several code design guidelines that should be adhered to:

- Reuse Bootstrap variables where possible. This allows themes to influence styling by simply overriding a few existing variables, instead of rewriting large chunks of the SCSS files.
- If you are unable to use existing variables - such as for very specific functionality - keep the new rules as specific as possible to your component so they do not cause issues in unexpected places.
- Check if existing components suit your use case. There may be existing components defined by Bootstrap or Horizon that can be reused, rather than writing new ones.

Theme Preview Panel

The Bootstrap Theme Preview panel contains examples of all stock Bootstrap markup with the currently applied theme, as well as source code for replicating them; click the `</>` symbol when hovering over a component.

To enable the Developer dashboard with the Theme Preview panel:

1. Set `DEBUG` setting to True.
2. Copy `_9001_developer.py` and `_9010_preview.py` from `openstack_dashboard/contrib/developer/enabled/` to `openstack_dashboard/local/enabled/`.
3. Restart the web server.

Alternate Theme

A second theme is provided by default at `openstack_dashboard/themes/material/`. When adding new SCSS to horizon, you should check that it does not interfere with the Material theme. Images of how the Material theme should look can be found at <https://bootswatch.com/3/paper/>. This theme is now configured to run as the alternate theme within Horizon.

Release Notes

Release notes for a patch should be included in the patch with the associated changes whenever possible. This allow for simpler tracking. It also enables a single cherry pick to be done if the change is backported to a previous release. In some cases, such as a feature that is provided via multiple patches, release notes can be done in a follow-on review.

If the following applies to the patch, a release note is required:

- The deployer needs to take an action when upgrading
- A new feature is implemented
- Function was removed (hopefully it was deprecated)
- Current behavior is changed
- A new config option is added that the deployer should consider changing from the default
- A security bug is fixed

Note:

- A release note is suggested if a long-standing or important bug is fixed. Otherwise, a release note is not required.
 - It is not recommended that individual release notes use **prelude** section as it is for release highlights.
-

Warning: Avoid modifying an existing release note file even though it is related to your change. If you modify a release note file of a past release, the whole content will be shown in a latest release. The only allowed case is to update a release note in a same release.

If you need to update a release note of a past release, edit a corresponding release note file in a stable branch directly.

Horizon uses [reno](#) to generate release notes. Please read the docs for details. In summary, use

```
$ tox -e venv -- reno new <bug-,bp-,whatever>
```

Then edit the sample file that was created and push it with your change.

To see the results:

```
$ git commit # Commit the change because reno scans git log.  
$ tox -e releasenotes
```

Then look at the generated release notes files in `releasenotes/build/html` in your favorite browser.

Translation in Horizon

What is the point of translating my code?

You introduced an awesome piece of code and revel in your glorious accomplishment. Suddenly your world comes crashing down when a core hands you a -1 because your code is not translated. What gives?

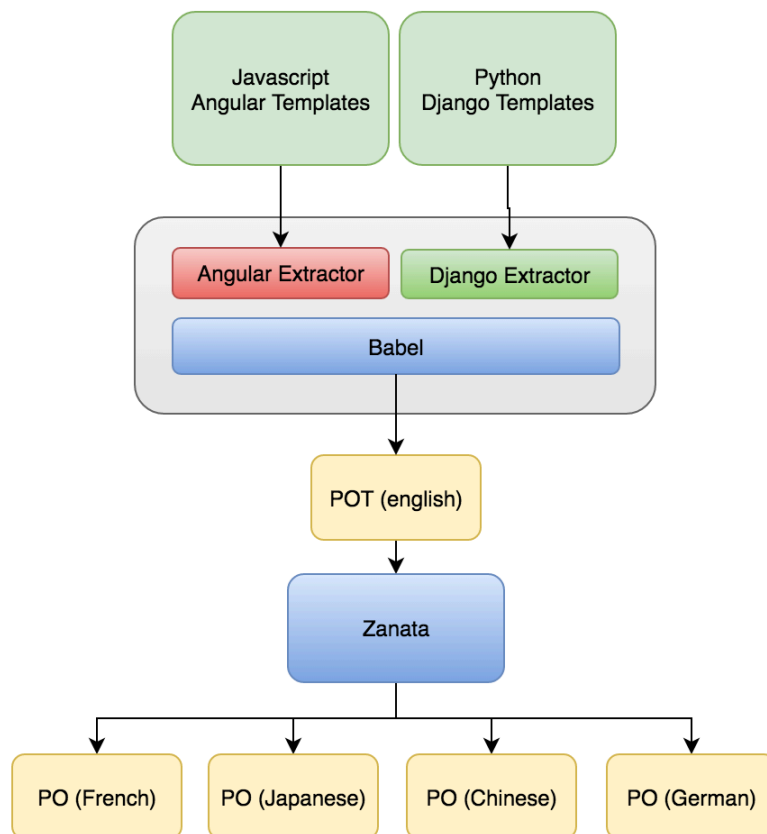
If you are writing software for a global audience, you must ensure that it is translated so that other people around the world are able to use it. Adding translation to your code is not that hard and a requirement for horizon.

If you are interested in contributing translations, you may want to investigate [Zanata](#) and the [upstream translations](#). You can visit the internationalization project IRC channel [#openstack-i18n](#), if you need further assistance.

Overview and Architecture

You can skip this section if you are only interested in learning how to use translation. This section explains the two main components to translation: message extraction and message substitution. We will briefly go over what each one does for translation as a whole.

Message Extraction



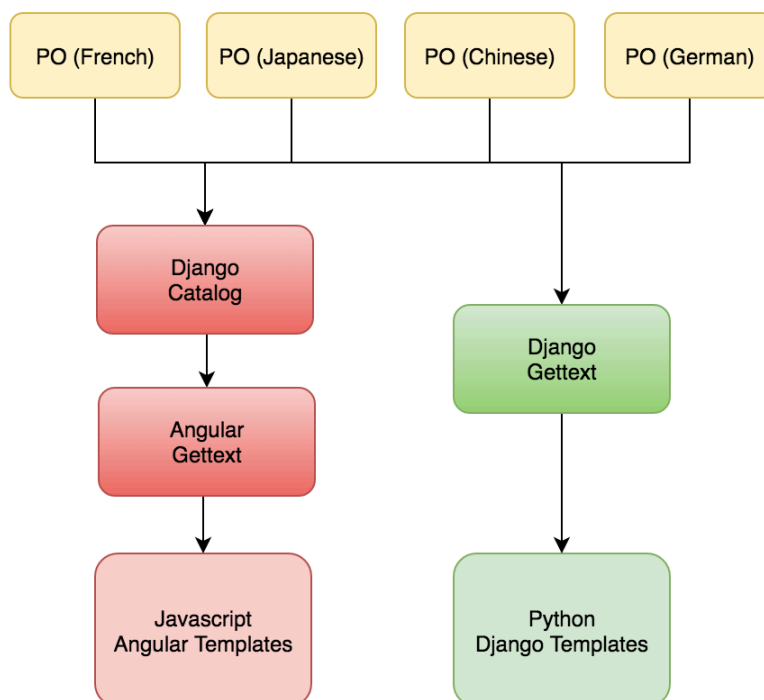
Message extraction is the process of collecting translatable strings from the code. The diagram above shows the flow of how messages are extracted and then translated. Lets break this up into steps we can follow:

1. The first step is to mark untranslated strings so that the extractor is able to locate them. Refer to the guide below on how to use translation and what these markers look like.
2. Once marked, we can then run `tox -e manage -- extract_messages`, which searches the codebase for these markers and extracts them into a Portable Object Template (POT) file. In horizon, we extract from both the `horizon` folder and the `openstack_dashboard` folder. We use the AngularJS extractor for JavaScript and HTML files and the Django extractor for Python and Django templates; both extractors are Babel plugins.
3. To update the `.po` files, you can run `tox -e manage -- update_catalog` to update the `.po` file for every language, or you can specify a specific language to update like this: `tox -e manage`

-- update_catalog de. This is useful if you want to add a few extra translatable strings for a downstream customisation.

Note: When pushing code upstream, the only requirement is to mark the strings correctly. All creation of POT and PO files is handled by a daily upstream job. Further information can be found in the [translation infrastructure documentation](#).

Message Substitution



Message substitution is not the reverse process of message extraction. The process is entirely different. Lets walk through this process.

- Remember those markers we talked about earlier? Most of them are functions like gettext or one of its variants. This allows the function to serve a dual purpose - acting as a marker and also as a replacer.
- In order for translation to work properly, we need to know the users locale. In horizon, the user can specify the locale using the Settings panel. Once we know the locale, we know which Portable Object (PO) file to use. The PO file is the file we received from translators in the message extraction process. The gettext functions that we wrapped our code around are then able to replace the untranslated strings with the translated one by using the untranslated string as the message id.

- For client-side translation, Django embeds a corresponding Django message catalog. Javascript code on the client can use this catalog to do string replacement similar to how server-side translation works.

If you are setting up a project and need to know how to make it translatable, please refer to [this guide](#).

Making strings translatable

To make your strings translatable, you need to mark it so that horizon can locate and extract it into a POT file. When a user from another locale visits your page, your string is replaced with the correct translated version.

In Django

To translate a string, simply wrap one of the gettext variants around the string. The examples below show you how to do translation for various scenarios, such as interpolation, contextual markers and translation comments.

```
from django.utils.translation import pgettext
from django.utils.translation import ugettext as _
from django.utils.translation import ungettext

class IndexView(request):

    # Single example
    _("Images")

    # Plural example
    ungettext(
        "there is %(count)d object",
        "there are %(count)d objects",
        count) % { "count": count }

    # Interpolated example
    mood = "wonderful"
    output = _("Today is %(mood)s.") % mood

    # Contextual markers
    pgettext("the month name", "May")

    # Translators: This message appears as a comment for translators!
    ugettext("Welcome translators.")
```

Note: In the example above, we imported `ugettext` as `_`. This is a common alias for `gettext` or any of its variants.

In Django templates

To use translation in your template, make sure you load the `i18n` module. To translate a line of text, use the `trans` template tag. If you need to translate a block of text, use the `blocktrans` template tag.

Sometimes, it is helpful to provide some context via the `comment` template tag. There a number of other tags and filters at your disposal should you need to use them. For more information, see the [Django docs](#)

```
{% extends 'base.html' %}
{% load i18n %}
{% block title %}
    {% trans "Images" %}
{% endblock %}

{% block main %}
    {% comment %}Translators: Images is an OpenStack resource{% endcomment %}
    {% blocktrans with amount=images.length %}
        There are {{ amount }} images available for display.
    {% endblocktrans %}
{% endblock %}
```

In JavaScript

The Django message catalog is injected into the front-end. The `gettext` function is available as a global function so you can just use it directly. If you are writing AngularJS code, we prefer that you use the `gettext` service, which is essentially a wrapper around the `gettext` function.

```
Angular
.module(myModule)
.controller(myCtrl);

myCtrl.$inject = [
    "horizon.framework.util.i18n.gettext"
];

function myCtrl(gettext) {
    var translated = gettext("Images");
}
```

Warning: For localization in AngularJS files, use the AngularJS service `horizon.framework.util.i18n.gettext`. Ensure that the injected dependency is named `gettext` or `nggettext`. If you do not do this, message extraction will not work properly!

In AngularJS templates

To use translation in your AngularJS template, use the `translate` tag or the `translate` filter. Note that we are using `angular-gettext` for message substitution but not for message extraction.

```
<translate>Directive example</translate>
<div translate>Attribute example</div>
<div translate>Interpolated {{example}}</div>
<span>{$ 'Filter example'| translate $}</span>

<span translate>
  This <em>is</em> a <strong>bad</strong> example
  because it contains HTML and makes it harder to translate.
  However, it will still translate.
</span>
```

Note: The annotations in the example above are guaranteed to work. However, not all of the `angular-gettext` annotations are supported because we wrote our own custom babel extractor. If you need support for the annotations, ask on IRC in the `#openstack-horizon` room or report a bug. Also note that you should avoid embedding HTML fragments in your texts because it makes it harder to translate. Use your best judgement if you absolutely need to include HTML.

Pseudo translation tool

The pseudo translation tool can be used to verify that code is ready to be translated. The pseudo tool replaces a languages translation with a complete, fake translation. Then you can verify that your code properly displays fake translations to validate that your code is ready for translation.

Running the pseudo translation tool

1. Make sure your `.pot` files are up to date

```
$ tox -e manage -- extract_messages
```

2. Run the pseudo tool to create pseudo translations. This example replaces the German translation with a pseudo translation

```
$ tox -e manage -- update_catalog de --pseudo
```

3. Compile the catalog

```
$ tox -e manage -- compilemessages
```

4. Run your development server.

```
$ tox -e runserver
```

5. Log in and change to the language you pseudo translated.

It should look weird. More specifically, the translatable segments are going to start and end with a bracket and they are going to have some added characters. For example, Log In will become [~Log In~ç] This is useful because you can inspect for the following, and consider if your code is working like it should:

- If you see a string in English its not translatable. Should it be?
- If you see brackets next to each other that might be concatenation. Concatenation can make quality translations difficult or impossible. See [Use string formatting variables, never perform string concatenation](#) for additional information.
- If there is unexpected wrapping/truncation there might not be enough space for translations.
- If you see a string in the proper translated language, it comes from an external source. (Thats not bad, just sometimes useful to know)
- If you get new crashes, there is probably a bug.

Dont forget to remove any pseudo translated .pot or .po files. Those should **not** be submitted for review.

Profiling Pages

In the Ocata release of Horizon a new OpenStack Profiler panel was introduced. Once it is enabled and all prerequisites are set up, you can see which API calls Horizon actually makes when rendering a specific page. To re-render the page while profiling it, youll need to use the Profile dropdown menu located in the top right corner of the screen. In order to be able to use Profile menu, the following steps need to be completed:

1. Enable the Developer dashboard by copying `_9001_developer.py` from `openstack_dashboard/contrib/developer/enabled/` to `openstack_dashboard/local/enabled/`.
2. Copy `openstack_dashboard/local/local_settings.d/_9030_profiler_settings.py.example` to `openstack_dashboard/local/local_settings.d/_9030_profiler_settings.py`
3. Copy `openstack_dashboard/contrib/developer/enabled/_9030_profiler.py` to `openstack_dashboard/local/enabled/_9030_profiler.py`.
4. To support storing profiler data on server-side, MongoDB cluster needs to be installed on your Devstack host (default configuration), see [Installing MongoDB](#). Then, change the `bindIp` key in `/etc/mongod.conf` to `0.0.0.0` and invoke `sudo service mongod restart`.
5. Collect and compress static assets with `python manage.py collectstatic -c` and `python manage.py compress`.
6. Restart the web server.
7. The Profile drop-down menu should appear in the top-right corner, you are ready to profile your pages!

Defining default settings in code

Note: This page tries to explain the plan to define default values of horizon/openstack_dashboard settings in code. This includes a blueprint [ini-based-configuration](#). This page will be updated once the effort is completed.

Planned Steps

1. Define the default values of existing settings
2. Revisit HORIZON_CONFIG
3. Introduce `oslo.config`

Define default values of existing settings

Currently all default values are defined in codes where they are consumed. This leads to the situation that it is not easy to know what are the default values and in a worse case default values defined in our codebase can have different default values.

As the first step toward ini-based-configuration, I propose to define all default values of existing settings in a single place per module. More specifically, the following modules are used:

- `openstack_dashboard.defaults` for `openstack_dashboard`
- `horizon.defaults` for `horizon`
- `openstack_auth.defaults` for `openstack_auth`

`horizon.defaults` load `openstack_auth.defaults` and overrides `openstack_auth` settings if necessary. Similarly, `openstack_dashboard.defaults` loads `horizon.defaults` and overrides `horizon` (and `openstack_auth`) settings if necessary.

The current style of `getattr(settings, <foo>, <default value>)` will be removed at the same time.

Note that `HORIZON_CONFIG` is not touched in this step. It will be covered in the next step.

Handling Django settings

Django provides a lot of settings and it is not practical to cover all in horizon. Only Django settings which horizon explicitly set will be defined in a dedicated python module.

The open question is how to maintain Django related settings in `openstack_dashboard` and `horizon`. How can we make them common? The following files are related:

- `openstack_dashboard.settings` (and `local_settings.py`)
- `openstack_dashboard.test.settings`
- `horizon.test.settings`

This will be considered as the final step of the ini-based-configuration effort after `horizon` and `openstack_dashboard` settings succeed to be migrated to `oslo.config` explained below.

Revisit HORIZON_CONFIG

HORIZON_CONFIG is an internal interface now and most/some(?) of them should not be exposed as config options. For example, the horizon plugin mechanism touches HORIZON_CONFIG to register horizon plugins.

It is better to expose only HORIZON_CONFIG settings which can be really exposed to operators. For such settings, we should define new settings in `openstack_dashboard` and can populate them into HORIZON_CONFIG in `settings.py`.

For example, `ajax_poll_interval` in HORIZON_CONFIG can be exposed to operators. In such case, we can define a new settings `AJAX_POLL_INTERVAL` in `openstack_dashboard/defaults.py` (or `horizon/defaults.py`).

Investigation is being summarized in an [etherpad page](#).

Introduce oslo.config

`local_settings.py` will have a priority over `oslo.config`. This means settings values from `oslo.config` will be loaded first and then `local_settings.py` and `local_settings.d` will be loaded in `settings.py`.

Basic strategy of mapping

- The current naming convention is random, so it sounds less reasonable to use the same name for `oslo.config`. `oslo.config` and python ini-based configuration mechanism provide a concept of category and there is no reason to use it. As category name, the categories of *Settings Reference* (like `keystone`, `glance`) will be honored.

For example, some `keystone` settings have a prefix `OPENSTACK_KEYSTONE_` like `OPENSTACK_KEYSTONE_DEFAULT_ROLE`. Some use `KEYSTONE_` like `KEYSTONE_IDP_PROVIDER_ID`. Some do not (like `ENFORCE_PASSWORD_CHECK`). In the `oslo.config` options, all prefixes will be dropped. The mapping will be:

- `OPENSTACK_KEYSTONE_DEFAULT_ROLE` <-> `[keystone] default_role`
- `KEYSTONE_IDP_PROVIDER_ID` <-> `[keystone] idp_provider_id`
- `ENFORCE_PASSWORD_CHECK` <-> `[keystone] enforce_password_check`

- `[default]` section is not used as much as possible. It will be used only for limited number of well-known options. Perhaps some common Django settings like `DEBUG`, `LOGGING` will match this category.

- Opt classes defined in `oslo.config` are used as much as possible.

- `StrOpt`, `IntOpt`
- `ListOpt`
- `MultiStrOpt`
- `DictOpt`

- A dictionary settings will be broken down into separate options. Good examples are `OPENSTACK_KEYSTONE_BACKEND` and `OPENSTACK_NEUTRON_NETWORK`.

- `OPENSTACK_KEYSTONE_BACKEND['name']` <-> `[keystone] backend_name`

- OPENSTACK_KEYSTONE_BACKEND['can_edit_user'] <-> [keystone] backend_can_edit_user
- OPENSTACK_KEYSTONE_BACKEND['can_edit_group'] <-> [keystone] backend_can_edit_group
- OPENSTACK_NEUTRON_NETWORK['enable_router'] <-> [neutron] enable_router
- OPENSTACK_NEUTRON_NETWORK['enable_ipv6'] <-> [neutron] enable_ipv6

Automatic Mapping

The straight-forward approach is to have a dictionary from setting names to oslo.config options like:

```
{
  'OPENSTACK_KEYSTONE_DEFAULT_ROLE': ('keystone', 'default_role'),
  'OPENSTACK_NEUTRON_NETWORK': {
    'enable_router': ('neutron', 'enable_router'),
    'enable_ipv6': ('neutron', 'enable_ipv6'),
    ...
  }
}
```

A key of the top-level dict is a name of Django settings. A corresponding value specifies oslo.config name by a list or a tuple where the first and second elements specify a section and a option name respectively.

When a value is a dict, this means a corresponding Django dict setting is broken down into several oslo.config options. In the above example, OPENSTACK_NEUTRON_NETWORK['enable_router'] is mapped to [neutron] enable_router.

Another idea is to introduce a new field to oslo.config classes. oslo-sample-generator might need to be updated. If this approach is really attractive, we can try this approach in future. The above dictionary-based approach will be used in the initial effort.

```
cfg.StrOpt(
    'default_role',
    default='_member_',
    django_setting='OPENSTACK_KEYSTONE_DEFAULT_ROLE',
    help=...
)

cfg.BoolOpt(
    'enable_router',
    default=True,
    django_setting=('OPENSTACK_NEUTRON_NETWORK', 'enable_router'),
    help=....)
)
```

Special Considerations

LOGGING

LOGGING setting is long enough. Python now recommend to configure logging using python dict directly, but from operator/packager perspective the legacy style of using the ini format sounds reasonable. The ini format is also used in other OpenStack projects too. In this effort, I propose to use the logging configuration via the ini format file and specify the logging conf file in a oslo.config option

Adopting oslo.log might be a good candidate, but it is not covered by this effort. It can be explored as future possible improvement.

SECURITY_GROUP_RULES

SECURITY_GROUP_RULES will be defined by YAML file. The YAML file can be validated by JSON schema in future (out of the scope of this effort)

all_tcp, all_udp and all_icmp are the reserved keyword, so it looks better to split the first three rules (all_tcp to all_icmp) and other remaining rules. The remaining rules will be loaded from a YAML file. For the first three rules, a boolean option to control their visibility in the security group rule form will be introduces in oslo.config. I am not sure this option is required or not, but as the first step of the migration it is reasonable to provide all compatibilities.

Handling Django settings

Django (and django related packages) provide many settings. It is not a good idea to expose all of them via oslo.config. What should we expose?

The proposal here is to expose only settings which openstack_dashboard expects to expose to deployers. Most Django settings are internally used in openstack_dashboard/settings.py. Settings required for horizon plugins are already exposed via the plugin settings, so there is no need to expose them. If deployers would like to customize Django basic settings, they can still configure them via local_settings.py or local_settings.d.

Packaging Software

Software packages

This section describes some general things that a developer should know about packaging software. This content is mostly derived from best practices.

A developer building a package is comparable to an engineer building a car with only a manual and very few tools. If the engineer needs a specific tool to build the car, he must create the tool, too.

As a developer, if you are going to add a library named foo, the package must adhere to the following standards:

- Be a free package created with free software.
- Include all tools that are required to build the package.
- Have an active and responsive upstream to maintain the package.

- Adhere to Filesystem Hierarchy Standards (FHS). A specific file system layout is not required.

Embedded copies not allowed

Imagine if all packages had a local copy of jQuery. If a security hole is discovered in jQuery, we must write more than 90 patches in Debian, one for each package that includes a copy. This is simply not practical. Therefore, it is unacceptable for Horizon to copy code from other repositories when creating a package. Copying code from another repository tends to create a fork, diverging from the upstream code. The fork includes code that is not being maintained, so if a bug is discovered in the original upstream, it cannot easily be fixed by updating a single package.

Another reason to avoid copying a library into Horizon source code is that it might create conflicting licenses. Distributing sources with conflicting licenses in one tarball revokes rights in best case. In the worst case, you could be held legally responsible.

Free software

Red Hat, Debian, and SUSE distributions are made only of free software (free as in Libre, or free speech). The software that we include in our repository is free. The tools are also free, and available in the distribution.

Because package maintainers care about the quality of the packages we upload, we run tests that are available from upstream repositories. This also qualifies test requirements as build requirements. The same rules apply for building the software as for the software itself. Special build requirements that are not included in the overall distribution are not allowed.

An example of historically limiting, non-free software is Selenium. For a long time, Selenium was only available from the non-free repositories of Debian. The reason was that upstream included some .xpi binaries. These .xpi included some Windows .dll and Linux .so files. Because they could not be rebuilt from the source, all of python-selenium was declared non-free. If we made Horizon build-depends on python-selenium, this would mean Horizon wouldnt be in Debian main anymore (contrib and non-free are *not* considered part of Debian). Recently, the package maintainer of python-selenium decided to remove the .xpi files from python-selenium, and upload it to Debian Experimental (this time, in main, not in non-free). If at some point it is possible for Horizon to use python-selenium (without the non-free .xpi files), then we could run Selenium tests at package build time.

Running unit tests at build time

The build environment inside a distribution is not exactly the same as the one in the OpenStack gate. For example, versions of a given library can be slightly different from the one in the gate. We want to detect when problematic differences exist so that we can fix them. Whenever possible, try to make the lives of the package maintainer easier, and allow them (or help them) to run unit tests.

Minified JavaScript policy

In free software distributions that actively maintain OpenStack packages (such as RDO, Debian, and Ubuntu), minified JavaScript is considered non-free. This means that minified JavaScript should *not* be present in upstream source code. At the very least, a non-minified version should be present next to the minified version. Also, be aware of potential security issues with minifiers. This [blog post](#) explains it very well.

Component version

Be careful about the version of all the components you use in your application. Since it is not acceptable to embed a given component within Horizon, we must use what is in the distribution, including all fonts, JavaScript, etc. This is where it becomes a bit tricky.

In most distributions, it is not acceptable to have multiple versions of the same piece of software. In Red Hat systems, it is technically possible to install 2 versions of one library at the same time, but a few restrictions apply, especially for usage. However, package maintainers try to avoid multiple versions as much as possible. For package dependency resolution, it might be necessary to provide packages for depending packages as well. For example, if you had Django-1.4 and Django-1.8 in the same release, you must provide Horizon built for Django-1.4 and another package providing Horizon built for Django-1.8. This is a large effort and needs to be evaluated carefully.

In Debian, it is generally forbidden to have multiple versions of the same library in the same Debian release. Very few exceptions exist.

Component versioning has consequences for an upstream author willing to integrate their software in a downstream distribution. The best situation is when it is possible to support whatever version is currently available in the target distributions, up to the latest version upstream. Declaring lower and upper bounds within your requirements.txt does not solve the issue. It allows all the tests to pass on gate because they are run against a narrow set of versions in requirements.txt. The downstream distribution might still have some dependencies with versions outside of the range that is specified in requirements.txt. These dependencies may lead to failures that are not caught in the OpenStack gate.

At times it might not be possible to support all versions of a library. It might be too much work, or it might be very hard to test in the gate. In this case, it is best to use whatever is available inside the target distributions. For example, Horizon currently supports jQuery $\geq 1.7.2$, as this is what is currently available in Debian Jessie and Ubuntu Trusty (the last LTS).

You can search in a distribution for a piece of software foo using a command like `dnf search foo`, or `zypper se -s foo`. `dnf info foo` returns more detailed information about the package.

Filesystem Hierarchy Standards

Every distribution must comply with the Filesystem Hierarchy Standards (FHS). The FHS defines a set of rules that we *must* follow as package maintainers. Some of the most important ones are:

- /usr is considered read only. Software must not write in /usr at runtime. However, it is fine for a package post-installation script to write in /usr. When this rule was not followed, distributions had to write many tricks to convince Horizon to write in /var/lib only. For example, distributions wrote symlinks to /var/lib/openstack-dashboard, or patched the default local_settings.py to write the SECRET_KEY in /var.

- Configuration must always be in `/etc`, no matter what. When this rule was not followed, package maintainers had to place symlinks to `/etc/openstack-dashboard/local_settings` in Red Hat based distributions instead of using directly `/usr/share/openstack-dashboard/openstack_dashboard/local/local_settings.py` which Horizon expects. In Debian, the configuration file is named `/etc/openstack-dashboard/local_settings.py`.

Packaging Horizon

Why we use XStatic

XStatic provides the following features that are not currently available by default with systems like NPM and Grunt:

- **Dependency checks:** XStatic checks that dependencies, such as fonts and JavaScript libs, are available in downstream distributions.
- **Reusable components across projects:** The XStatic system ensures components are reusable by other packages, like Fuel.
- **System-wide registry of static content:** XStatic brings a system-wide registry of components, so that it is easy to check if one is missing. For example, it can detect if there is no egg-info, or a broken package dependency exists.
- **No embedded content:** The XStatic system helps us avoid embedding files that are already available in the distribution, for example, `libjs-*` or `fonts-*` packages. It even provides a compatibility layer for distributions. Not every distribution places static files in the same position in the file system. If you are packaging an XStatic package for your distribution, make sure that you are using the static files provided by that specific distribution. Having put together an XStatic package is *no* guarantee to get it into a distribution. XStatic provides only the abstraction layer to use distribution provided static files.
- **Package build systems are disconnected from the outside network** (for several reasons). Other packaging systems download dependencies directly from the internet without verifying that the downloaded file is intact, matches a provided checksum, etc. With these other systems, there is no way to provide a mirror, a proxy or a cache, making builds even more unstable when minor networking issues are encountered.

The previous features are critical requirements of the Horizon packaging system. Any new system *must* keep these features. Although XStatic may mean a few additional steps from individual developers, those steps help maintain consistency and prevent errors across the project.

Packaging Horizon for distributions

Horizon is a Python module. Preferably, it is installed at the default location for python. In Fedora and openSUSE, this is `/usr/lib/python3.7/site-packages/horizon`, and in Debian/Ubuntu it is `/usr/lib/python3.7/dist-packages/horizon`.

Configuration files should reside under `/etc/openstack-dashboard`. Policy files should be created and modified there as well.

It is expected that `manage.py collectstatic` will be run during package build. This is the [recommended way](#) for Django applications. Depending on configuration, it might be required to `manage.py compress` during package build, too.

DevStack for Horizon

Place the following content into `devstack/local.conf` to start the services that Horizon supports in DevStack when `stack.sh` is run. If you need to use this with a stable branch you need to add `stable/<branch name>` to the end of each `enable_plugin` line (e.g. `stable/mitaka`). You can also check out DevStack using a stable branch tag. For more information on DevStack, see <https://docs.openstack.org/devstack/latest/>

```
[[local|localrc]]

ADMIN_PASSWORD="secretadmin"
DATABASE_PASSWORD="secretdatabase"
RABBIT_PASSWORD="secretrabbit"
SERVICE_PASSWORD="secretservice"

# For DevStack configuration options, see:
# https://docs.openstack.org/devstack/latest/configuration.html

# Note: there are several network setting changes that may be
# required to get networking properly configured in your environment.
# This file is just using the defaults set up by devstack.
# For a more detailed treatment of devstack network configuration
# options, please see:
# https://docs.openstack.org/devstack/latest/guides.html

### Supported Services
# The following panels and plugins are part of the Horizon tree
# and currently supported by the Horizon maintainers

# Enable Swift (Object Store) without replication
enable_service s-proxy s-object s-container s-account
SWIFT_HASH=66a3d6b56c1f479c8b4e70ab5c2000f5
SWIFT_REPLICAS=1
SWIFT_DATA_DIR=$DEST/data/swift

# Enable Neutron
enable_plugin neutron https://opendev.org/openstack/neutron

# Enable the Trunks extension for Neutron
enable_service q-trunk

# Enable the QoS extension for Neutron
enable_service q-qos

### Plugins
# Horizon has a large number of plugins, documented at
# https://docs.openstack.org/horizon/latest/install/plugin-registry.html
# See the individual repos for information on installing them.

[[post-config|$GLANCE_API_CONF]]
[DEFAULT]
```

(continues on next page)

(continued from previous page)

```
default_store=file
```

3.1.8 Module Reference

Horizon Framework

The Horizon Module

Horizon ships with a single point of contact for hooking into your project if you aren't developing your own *Dashboard* or *Panel*:

```
import horizon
```

From there you can access all the key methods you need.

Horizon

horizon.urls

The auto-generated URLconf for horizon. Usage:

```
url(r'', include(horizon.urls)),
```

horizon.register(*dashboard*)

Registers a *Dashboard* with Horizon.

horizon.unregister(*dashboard*)

Unregisters a *Dashboard* from Horizon.

horizon.get_absolute_url()

Returns the default URL for Horizons URLconf.

The default URL is determined by calling *get_absolute_url()* on the *Dashboard* instance returned by *get_default_dashboard()*.

horizon.get_user_home(*user*)

Returns the default URL for a particular user.

This method can be used to customize where a user is sent when they log in, etc. By default it returns the value of *get_absolute_url()*.

An alternative function can be supplied to customize this behavior by specifying either a URL or a function which returns a URL via the "user_home" key in HORIZON_CONFIG. Each of these would be valid:

```
{ "user_home": "/home", } # A URL
{ "user_home": "my_module.get_user_home", } # Path to a function
{ "user_home": lambda user: "/" + user.name, } # A function
{ "user_home": None, } # Will always return the default dashboard
```

This can be useful if the default dashboard may not be accessible to all users. When `user_home` is missing from `HORIZON_CONFIG`, it will default to the `settings.LOGIN_REDIRECT_URL` value.

`horizon.get_dashboard(dashboard)`

Returns the specified *Dashboard* instance.

`horizon.get_default_dashboard()`

Returns the default *Dashboard* instance.

If `"default_dashboard"` is specified in `HORIZON_CONFIG` then that dashboard will be returned. If not, the first dashboard returned by `get_dashboards()` will be returned.

`horizon.get_dashboards()`

Returns an ordered tuple of *Dashboard* modules.

Orders dashboards according to the `"dashboards"` key in `HORIZON_CONFIG` or else returns all registered dashboards in alphabetical order.

Any remaining *Dashboard* classes registered with Horizon but not listed in `HORIZON_CONFIG['dashboards']` will be appended to the end of the list alphabetically.

Dashboard

class `horizon.Dashboard(*args, **kwargs)`

A base class for defining Horizon dashboards.

All Horizon dashboards should extend from this base class. It provides the appropriate hooks for automatic discovery of *Panel* modules, automatically constructing URLconfs, and providing permission-based access control.

name

The name of the dashboard. This will be displayed in the auto-generated navigation and various other places. Default: `''`.

slug

A unique short name for the dashboard. The slug is used as a component of the URL path for the dashboard. Default: `''`.

panels

The `panels` attribute can be either a flat list containing the name of each panel **module** which should be loaded as part of this dashboard, or a list of *PanelGroup* classes which define groups of panels as in the following example:

```
class SystemPanels(horizon.PanelGroup):
    slug = "syspanel"
    name = _("System")
    panels = ('overview', 'instances', ...)

class Syspanel(horizon.Dashboard):
    panels = (SystemPanels,)
```

Automatically generated navigation will use the order of the modules in this attribute.

Default: `[]`.

Warning: The values for this attribute should not correspond to the *name* attributes of the `Panel` classes. They should be the names of the Python modules in which the `panel.py` files live. This is used for the automatic loading and registration of `Panel` classes much like Django's `ModelAdmin` machinery.

Panel modules must be listed in `panels` in order to be discovered by the automatic registration mechanism.

default_panel

The name of the panel which should be treated as the default panel for the dashboard, i.e. when you visit the root URL for this dashboard, that's the panel that is displayed. Default: `None`.

permissions

A list of permission names, all of which a user must possess in order to access any panel registered with this dashboard. This attribute is combined cumulatively with any permissions required on individual `Panel` classes.

urls

Optional path to a URLconf of additional views for this dashboard which are not connected to specific panels. Default: `None`.

nav

The `nav` attribute can be either a boolean value or a callable which accepts a `RequestContext` object as a single argument to control whether or not this dashboard should appear in automatically-generated navigation. Default: `True`.

public

Boolean value to determine whether this dashboard can be viewed without being logged in. Defaults to `False`.

allowed(*context*)

Checks for role based access for this dashboard.

Checks for access to any panels in the dashboard and of the dashboard itself.

This method should be overridden to return the result of any policy checks required for the user to access this dashboard when more complex checks are required.

get_absolute_url()

Returns the default URL for this dashboard.

The default URL is defined as the URL pattern with `name="index"` in the URLconf for the `Panel` specified by `default_panel`.

get_panel(*panel*)

Returns the `Panel` instance registered with this dashboard.

get_panel_group(*slug*)

Returns the specified `:class:~horizon.PanelGroup`.

Returns `None` if not registered.

get_panels()

Returns the `Panel` instances registered with this dashboard in order.

Panel grouping information is not included.

classmethod register(*panel*)
Registers a *Panel* with this dashboard.

classmethod unregister(*panel*)
Unregisters a *Panel* from this dashboard.

Panel

class horizon.Panel

A base class for defining Horizon dashboard panels.

All Horizon dashboard panels should extend from this class. It provides the appropriate hooks for automatically constructing URLconfs, and providing permission-based access control.

name

The name of the panel. This will be displayed in the auto-generated navigation and various other places. Default: ''.

slug

A unique short name for the panel. The slug is used as a component of the URL path for the panel. Default: ''.

permissions

A list of permission names, all of which a user must possess in order to access any view associated with this panel. This attribute is combined cumulatively with any permissions required on the Dashboard class with which it is registered.

urls

Path to a URLconf of views for this panel using dotted Python notation. If no value is specified, a file called `urls.py` living in the same package as the `panel.py` file is used. Default: `None`.

nav

The `nav` attribute can be either a boolean value or a callable which accepts a `RequestContext` object as a single argument to control whether or not this panel should appear in automatically-generated navigation. Default: `True`.

index_url_name

The name argument for the URL pattern which corresponds to the index view for this `Panel`. This is the view that `Panel.get_absolute_url()` will attempt to reverse.

static can_register()

This optional static method can be used to specify conditions that need to be satisfied to load this panel. Unlike `permissions` and `allowed` this method is intended to handle settings based conditions rather than user based permission and policy checks. The return value is boolean. If the method returns `True`, then the panel will be registered and available to user (if `permissions` and `allowed` runtime checks are also satisfied). If the method returns `False`, then the panel will not be registered and will not be available via normal navigation or direct URL access.

get_absolute_url()

Returns the default URL for this panel.

The default URL is defined as the URL pattern with `name="index"` in the URLconf for this panel.

Panel Group

class horizon.**PanelGroup**(*dashboard, slug=None, name=None, panels=None*)

A container for a set of *Panel* classes.

When iterated, it will yield each of the *Panel* instances it contains.

slug

A unique string to identify this panel group. Required.

name

A user-friendly name which will be used as the group heading in places such as the navigation.
Default: None.

panels

A list of panel module names which should be contained within this grouping.

Horizon Workflows

One of the most challenging aspects of building a compelling user experience is crafting complex multi-part workflows. Horizons `workflows` module aims to bring that capability within everyday reach.

See also:

For usage information, tips & tricks and more examples check out the *Workflows Topic Guide*.

Workflows

class horizon.workflows.**Workflow**(*request=None, context_seed=None, entry_point=None, *args, **kwargs*)

A Workflow is a collection of Steps.

Its interface is very straightforward, but it is responsible for handling some very important tasks such as:

- Handling the injection, removal, and ordering of arbitrary steps.
- Determining if the workflow can be completed by a given user at runtime based on all available information.
- Dispatching connections between steps to ensure that when context data changes all the applicable callback functions are executed.
- Verifying/validating the overall data integrity and subsequently triggering the final method to complete the workflow.

The *Workflow* class has the following attributes:

name

The verbose name for this workflow which will be displayed to the user. Defaults to the class name.

slug

The unique slug for this workflow. Required.

steps

Read-only access to the final ordered set of step instances for this workflow.

default_steps

A list of [Step](#) classes which serve as the starting point for this workflows ordered steps. Defaults to an empty list ([]).

finalize_button_name

The name which will appear on the submit button for the workflows form. Defaults to "Save".

success_message

A string which will be displayed to the user upon successful completion of the workflow. Defaults to "{ workflow.name } completed successfully."

failure_message

A string which will be displayed to the user upon failure to complete the workflow. Defaults to "{ workflow.name } did not complete."

depends_on

A roll-up list of all the depends_on values compiled from the workflows steps.

contributions

A roll-up list of all the contributes values compiled from the workflows steps.

template_name

Path to the template which should be used to render this workflow. In general the default common template should be used. Default: "horizon/common/_workflow.html".

entry_point

The slug of the step which should initially be active when the workflow is rendered. This can be passed in upon initialization of the workflow, or set anytime after initialization but before calling either `get_entry_point` or `render`.

redirect_param_name

The name of a parameter used for tracking the URL to redirect to upon completion of the workflow. Defaults to "next".

object

The object (if any) which this workflow relates to. In the case of a workflow which creates a new resource the object would be the created resource after the relevant creation steps have been undertaken. In the case of a workflow which updates a resource it would be the resource being updated after it has been retrieved.

wizard

Whether to present the workflow as a wizard, with prev and next buttons and validation after every step.

add_error_to_step(*message, slug*)

Adds an error message to the workflows Step.

This is useful when you wish for API errors to appear as errors on the form rather than using the messages framework.

The workflows Step is specified by its slug.

finalize()

Finalizes a workflow by running through all the actions.

It runs all the actions in order and calling their `handle` methods. Returns `True` on full success, or `False` for a partial success, e.g. there were non-critical errors. (If it failed completely the function wouldnt return.)

format_status_message(*message*)

Hook to allow customization of the message returned to the user.

This is called upon both successful or unsuccessful completion of the workflow.

By default it simply inserts the workflows name into the message string.

get_absolute_url()

Returns the canonical URL for this workflow.

This is used for the POST action attribute on the form element wrapping the workflow.

For convenience it defaults to the value of `request.get_full_path()` with any query string stripped off, e.g. the path at which the workflow was requested.

get_entry_point()

Returns the slug of the step which the workflow should begin on.

This method takes into account both already-available data and errors within the steps.

get_step(*slug*)

Returns the instantiated step matching the given slug.

get_success_url()

Returns a URL to redirect the user to upon completion.

By default it will attempt to parse a `success_url` attribute on the workflow, which can take the form of a reversible URL pattern name, or a standard HTTP URL.

handle(*request, context*)

Handles any final processing for this workflow.

Should return a boolean value indicating success.

is_valid()

Verifies that all required data is present in the context.

It also calls the `validate` method to allow for finer-grained checks on the context data.

classmethod register(*step_class*)

Registers a *Step* with the workflow.

render()

Renders the workflow.

classmethod unregister(*step_class*)

Unregisters a *Step* from the workflow.

validate(*context*)

Hook for custom context data validation.

Should return a boolean value or raise *WorkflowValidationError*.

Steps

class `horizon.workflows.Step(workflow)`

A wrapper around an action which defines its context in a workflow.

It knows about details such as:

- The workflows context data (data passed from step to step).
- The data which must be present in the context to begin this step (the steps dependencies).
- The keys which will be added to the context data upon completion of the step.
- The connections between this steps fields and changes in the context data (e.g. if that piece of data changes, what needs to be updated in this step).

A Step class has the following attributes:

action_class

The *Action* class which this step wraps.

depends_on

A list of context data keys which this step requires in order to begin interaction.

contributes

A list of keys which this step will contribute to the workflows context data. Optional keys should still be listed, even if their values may be set to `None`.

connections

A dictionary which maps context data key names to lists of callbacks. The callbacks may be functions, dotted python paths to functions which may be imported, or dotted strings beginning with "self" to indicate methods on the current Step instance.

before

Another Step class. This optional attribute is used to provide control over workflow ordering when steps are dynamically added to workflows. The workflow mechanism will attempt to place the current step before the step specified in the attribute.

after

Another Step class. This attribute has the same purpose as *before()* except that it will instead attempt to place the current step after the given step.

help_text

A string of simple help text which will be prepended to the Action class help text if desired.

template_name

A path to a template which will be used to render this step. In general the default common template should be used. Default: "horizon/common/_workflow_step.html".

has_errors

A boolean value which indicates whether or not this step has any errors on the action within it or in the scope of the workflow. This attribute will only accurately reflect this status after validation has occurred.

slug

Inherited from the Action class.

name

Inherited from the Action class.

permissions

Inherited from the `Action` class.

add_step_error(*message*)

Adds an error to the Step based on API issues.

allowed(*request*)

Determines whether or not the step is displayed.

Step instances can override this method to specify conditions under which this tab should not be shown at all by returning `False`.

The default behavior is to return `True` for all cases.

contribute(*data, context*)

Adds the data listed in `contributes` to the workflows context.

By default, the context is simply updated with all the data returned by the action.

Note that even if the value of one of the `contributes` keys is not present (e.g. optional) the key should still be added to the context with a value of `None`.

get_help_text()

Returns the help text for this step.

get_id()

Returns the ID for this step. Suitable for use in HTML markup.

has_required_fields()

Returns `True` if action contains any required fields.

prepare_action_context(*request, context*)

Hook to customize how the workflow context is passed to the action.

This is the reverse of what `contribute` does to make the action outputs sane for the workflow. Changes to the context are not saved globally here. They are localized to the action.

Simply returns the unaltered context by default.

render()

Renders the step.

Actions

class horizon.workflows.Action(*request, context, *args, **kwargs*)

An `Action` represents an atomic logical interaction with the system.

This is easier to understand with a conceptual example: in the context of a launch instance workflow, actions would include naming the instance, selecting an image, and ultimately launching the instance.

Because `Actions` are always interactive, they always provide form controls, and thus inherit from Django's `Form` class. However, they have some additional intelligence added to them:

- `Actions` are aware of the permissions required to complete them.
- `Actions` have a meta-level concept of help text which is meant to be displayed in such a way as to give context to the action regardless of where the action is presented in a site or workflow.

- Actions understand how to handle their inputs and produce outputs, much like `SelfHandlingForm` does now.

Action classes may define the following attributes in a `Meta` class within them:

name

The verbose name for this action. Defaults to the name of the class.

slug

A semi-unique slug for this action. Defaults to the slugified name of the class.

permissions

A list of permission names which this action requires in order to be completed. Defaults to an empty list (`[]`).

policy_rules

list of scope and rule tuples to do policy checks on, the composition of which is (scope, rule)

- scope: service type managing the policy for action
- rule: string representing the action to be checked

for a policy that requires a single rule check:

```
policy_rules should look like
    (("compute", "compute:create_instance"),)
```

for a policy that requires multiple rule checks:

```
rules should look like
    (("identity", "identity:list_users"),
     ("identity", "identity:list_roles"))
```

where two service-rule clauses are OR-ed.

help_text

A string of simple help text to be displayed alongside the Actions fields.

help_text_template

A path to a template which contains more complex help text to be displayed alongside the Actions fields. In conjunction with `get_help_text()` method you can customize your help text template to display practically anything.

add_action_error(*message*)

Adds an error to the Actions Step based on API issues.

get_help_text(*extra_context=None*)

Returns the help text for this step.

handle(*request, context*)

Handles any requisite processing for this action.

The method should return either `None` or a dictionary of data to be passed to `contribute()`.

Returns `None` by default, effectively making it a no-op.

WorkflowView

class horizon.workflows.WorkflowView

A generic view which handles the intricacies of workflow processing.

workflow_class

The *Workflow* class which this view handles. Required.

template_name

The template to use when rendering this view via standard HTTP requests. Required.

ajax_template_name

The template to use when rendering the workflow for AJAX requests. In general the default common template should be used. Defaults to "horizon/common/_workflow.html".

context_object_name

The key which should be used for the workflow object in the template context. Defaults to "workflow".

get(request, *args, **kwargs)

Handler for HTTP GET requests.

get_context_data(**kwargs)

Returns the template context, including the workflow class.

This method should be overridden in subclasses to provide additional context data to the template.

get_initial()

Returns initial data for the workflow.

Defaults to using the GET parameters to allow pre-seeding of the workflow context values.

get_layout()

Returns classes for the workflow element in template.

The returned classes are determined based on the workflow characteristics.

get_template_names()

Returns the template name to use for this request.

get_workflow()

Returns the instantiated workflow class.

post(request, *args, **kwargs)

Handler for HTTP POST requests.

validate_steps(request, workflow, start, end)

Validates the workflow steps from *start* to *end*, inclusive.

Returns a dict describing the validation state of the workflow.

Horizon DataTables

Horizon includes a componentized API for programmatically creating tables in the UI. Why would you want this? It means that every table renders correctly and consistently, table-level and row-level actions all have a consistent API and appearance, and generally you don't have to reinvent the wheel or copy-and-paste every time you need a new table!

See also:

For usage information, tips & tricks and more examples check out the [DataTables Topic Guide](#).

DataTable

The core class which defines the high-level structure of the table being represented. Example:

```
class MyTable(DataTable):
    name = Column('name')
    email = Column('email')

    class Meta(object):
        name = "my_table"
        table_actions = (MyAction, MyOtherAction)
        row_actions = (MyAction)
```

A full reference is included below:

class horizon.tables.DataTable(*request*, *data=None*, *needs_form_wrapper=None*, ***kwargs*)
A class which defines a table with all data and associated actions.

name

String. Read-only access to the name specified in the tables Meta options.

multi_select

Boolean. Read-only access to whether or not this table should display a column for multi-select checkboxes.

data

Read-only access to the data this table represents.

filtered_data

Read-only access to the data this table represents, filtered by the *filter()* method of the tables *FilterAction* class (if one is provided) using the current requests query parameters.

calculate_row_status(*statuses*)

Returns a boolean value determining the overall row status.

It is determined based on the dictionary of column name to status mappings passed in.

By default, it uses the following logic:

1. If any statuses are False, return False.
2. If no statuses are False but any or None, return None.
3. If all statuses are True, return True.

This provides the greatest protection against false positives without weighting any particular columns.

The `statuses` parameter is passed in as a dictionary mapping column names to their statuses in order to allow this function to be overridden in such a way as to weight one columns status over another should that behavior be desired.

classmethod `check_handler(request)`

Determine whether the request should be handled by this table.

`css_classes()`

Returns the additional CSS class to be added to `<table>` tag.

`get_absolute_url()`

Returns the canonical URL for this table.

This is used for the POST action attribute on the form element wrapping the table. In many cases it is also useful for redirecting after a successful action on the table.

For convenience it defaults to the value of `request.get_full_path()` with any query string stripped off, e.g. the path at which the table was requested.

`get_columns()`

Returns this tables columns including auto-generated ones.

`get_empty_message()`

Returns the message to be displayed when there is no data.

`get_filter_field()`

Get the filter field value used for server type filters.

This is the value from the filter actions list of filter choices.

`get_filter_first_message()`

Return the message to be displayed first in the filter.

when the user needs to provide a search criteria first before loading any data.

`get_filter_string()`

Get the filter string value.

For server type filters this is saved in the session so that it gets persisted across table loads. For other filter types this is obtained from the POST dict.

`get_full_url()`

Returns the full URL path for this table.

This is used for the POST action attribute on the form element wrapping the table. We use this method to persist the pagination marker.

`get_marker()`

Returns the identifier for the last object in the current data set.

The return value will be used as marker/limit-based paging in the API.

`get_object_by_id(lookup)`

Returns the data object whose ID matches `lookup` parameter.

The data object is looked up from the tables dataset and the data which matches the `lookup` parameter specified. An error will be raised if the match is not a single data object.

We will convert the object id and `lookup` to unicode before comparison.

Uses `get_object_id()` internally.

get_object_display(*datum*)

Returns a display name that identifies this object.

By default, this returns a name attribute from the given object, but this can be overridden to return other values.

get_object_id(*datum*)

Returns the identifier for the object this row will represent.

By default this returns an id attribute on the given object, but this can be overridden to return other values.

<p>Warning: Make sure that the value returned is a unique value for the id otherwise rendering issues can occur.</p>

get_pagination_string()

Returns the query parameter string to paginate to the next page.

get_prev_marker()

Returns the identifier for the first object in the current data set.

The return value will be used as marker/limit-based paging in the API.

get_prev_pagination_string()

Returns the query parameter string to paginate to the prev page.

get_row_actions(*datum*)

Returns a list of the action instances for a specific row.

get_row_status_class(*status*)

Returns a css class name determined by the status value.

This class name is used to indicate the status of the rows in the table if any `status_columns` have been specified.

get_rows()

Return the row data for this table broken out by columns.

get_table_actions()

Returns a list of the action instances for this table.

property has_actions

Indicates whether there are any available actions on this table.

Returns a boolean value.

has_more_data()

Returns a boolean value indicating whether there is more data.

Returns True if there is more data available to this table from the source (generally an API).

The method is largely meant for internal use, but if you want to override it to provide custom behavior you can do so at your own risk.

has_prev_data()

Returns a boolean value indicating whether there is previous data.

Returns True if there is previous data available to this table from the source (generally an API).

The method is largely meant for internal use, but if you want to override it to provide custom behavior you can do so at your own risk.

inline_edit_handle(*request, table_name, action_name, obj_id, new_row*)

Inline edit handler.

Showing form or handling update by POST of the cell.

inline_update_action(*request, datum, cell, obj_id, cell_name*)

Handling update by POST of the cell.

maybe_handle()

Handles table actions if needed.

It determines whether the request should be handled by any action on this table after data has been loaded.

maybe_preempt()

Determine whether the request should be handled in earlier phase.

It determines the request should be handled by a preemptive action on this table or by an AJAX row update before loading any data.

property_needs_form_wrapper

Returns if this table should be rendered wrapped in a <form> tag.

Returns a boolean value.

static parse_action(*action_string*)

Parses the `action_string` parameter sent back with the POST data.

By default this parses a string formatted as `{{ table_name }}__{{ action_name }}__{{ row_id }}` and returns each of the pieces. The `row_id` is optional.

render()

Renders the table using the template from the table options.

render_row_actions(*datum, row=False*)

Renders the actions specified in `Meta.row_actions`.

The actions are rendered using the current row data. If `row` is True, the actions are rendered in a row of buttons. Otherwise they are rendered in a dropdown box.

render_table_actions()

Renders the actions specified in `Meta.table_actions`.

sanitize_id(*obj_id*)

Override to modify an incoming `obj_id` to match existing API.

It is used to modify an incoming `obj_id` (used in Horizon) to the data type or format expected by the API.

set_multiselect_column_visibility(*visible=True*)

hide checkbox column if no current table action is allowed.

take_action(*action_name, obj_id=None, obj_ids=None*)

Locates the appropriate action and routes the object data to it.

The action should return an HTTP redirect if successful, or a value which evaluates to `False` if unsuccessful.

DataTable Options

The following options can be defined in a `Meta` class inside a `DataTable` class. Example:

```
class MyTable(DataTable):
    class Meta(object):
        name = "my_table"
        verbose_name = "My Table"
```

`class horizon.tables.base.DataTableOptions(options)`

Contains options for `DataTable` objects.

name

A short name or slug for the table.

verbose_name

A more verbose name for the table meant for display purposes.

columns

A list of column objects or column names. Controls ordering/display of the columns in the table.

table_actions

A list of action classes derived from the `Action` class. These actions will handle tasks such as bulk deletion, etc. for multiple objects at once.

table_actions_menu

A list of action classes similar to `table_actions` except these will be displayed in a menu instead of as individual buttons. Actions from this list will take precedence over actions from the `table_actions` list.

table_actions_menu_label

A label of a menu button for `table_actions_menu`. The default is `Actions` or `More Actions` depending on `table_actions`.

row_actions

A list similar to `table_actions` except tailored to appear for each row. These actions act on a single object at a time.

actions_column

Boolean value to control rendering of an additional column containing the various actions for each row. Defaults to `True` if any actions are specified in the `row_actions` option.

multi_select

Boolean value to control rendering of an extra column with checkboxes for selecting multiple objects in the table. Defaults to `True` if any actions are specified in the `table_actions` option.

filter

Boolean value to control the display of the filter search box in the table actions. By default it checks whether or not an instance of `FilterAction` is in `table_actions`.

template

String containing the template which should be used to render the table. Defaults to "horizon/common/_data_table.html".

row_actions_dropdown_template

String containing the template which should be used to render the row actions dropdown. Defaults to "horizon/common/_data_table_row_actions_dropdown.html".

row_actions_row_template

String containing the template which should be used to render the row actions. Defaults to "horizon/common/_data_table_row_actions_row.html".

table_actions_template

String containing the template which should be used to render the table actions. Defaults to "horizon/common/_data_table_table_actions.html".

context_var_name

The name of the context variable which will contain the table when it is rendered. Defaults to "table".

prev_pagination_param

The name of the query string parameter which will be used when paginating backward in this table. When using multiple tables in a single view this will need to be changed to differentiate between the tables. Default: "prev_marker".

pagination_param

The name of the query string parameter which will be used when paginating forward in this table. When using multiple tables in a single view this will need to be changed to differentiate between the tables. Default: "marker".

status_columns

A list or tuple of column names which represents the state of the data object being represented.

If `status_columns` is set, when the rows are rendered the value of this column will be used to add an extra class to the row in the form of "status_up" or "status_down" for that rows data.

The row status is used by other Horizon components to trigger tasks such as dynamic AJAX updating.

cell_class

The class which should be used for rendering the cells of this table. Optional. Default: `Cell`.

row_class

The class which should be used for rendering the rows of this table. Optional. Default: `Row`.

column_class

The class which should be used for handling the columns of this table. Optional. Default: `Column`.

css_classes

A custom CSS class or classes to add to the `<table>` tag of the rendered table, for when the particular table requires special styling. Default: "".

mixed_data_type

A toggle to indicate if the table accepts two or more types of data. Optional. Default: `False`

data_types

A list of data types that this table would accept. Default to be an empty list, but if the attribute `mixed_data_type` is set to `True`, then this list must have at least one element.

data_type_name

The name of an attribute to assign to data passed to the table when it accepts mix data. Default: `"_table_data_type"`

footer

Boolean to control whether or not to show the tables footer. Default: `True`.

hidden_title

Boolean to control whether or not to show the tables title. Default: `True`.

permissions

A list of permission names which this table requires in order to be displayed. Defaults to an empty list (`[]`).

FormsetDataTable

You can integrate the `DataTable` with a Django Formset using one of following classes:

class `horizon.tables.formset.FormsetDataTableMixin(*args, **kwargs)`

A mixin for `DataTable` to support Django Formsets.

This works the same as the `FormsetDataTable` below, but can be used to add to existing `DataTable` subclasses.

get_empty_row()

Return a row with no data, for adding at the end of the table.

get_formset()

Provide the formset corresponding to this `DataTable`.

Use this to validate the formset and to get the submitted data back.

get_required_columns()

Lists names of columns that have required fields.

get_rows()

Return the row data for this table broken out by columns.

The row objects get an additional `form` parameter, with the formset form corresponding to that row.

class `horizon.tables.formset.FormsetDataTable(*args, **kwargs)`

A `DataTable` with support for Django Formsets.

Note that `horizon.tables.DataTableOptions.row_class` and `horizon.tables.DataTableOptions.cell_class` are overwritten in this class, so setting them in `Meta` has no effect.

formset_class

A class made with `django.forms.formsets.formset_factory` containing the definition of the formset to use with this data table.

The columns that are named the same as the formset fields will be replaced with form widgets in the table. Any hidden fields from the formset will also be included. The fields that are not hidden and dont correspond to any column will not be included in the form.

Table Components

```
class horizon.tables.Column(transform, verbose_name=None, sortable=True, link=None,
                             allowed_data_types=None, hidden=False, attrs=None,
                             status=False, status_choices=None, display_choices=None,
                             empty_value=None, filters=None, classes=None,
                             summation=None, auto=None, truncate=None,
                             link_classes=None, wrap_list=False, form_field=None,
                             form_field_attributes=None, update_action=None,
                             link_attrs=None, policy_rules=None,
                             cell_attributes_getter=None, help_text=None)
```

A class which represents a single column in a [DataTable](#).

transform

A string or callable. If `transform` is a string, it should be the name of the attribute on the underlying data class which should be displayed in this column. If it is a callable, it will be passed the current rows data at render-time and should return the contents of the cell. Required.

verbose_name

The name for this column which should be used for display purposes. Defaults to the value of `transform` with the first letter of each word capitalized if the `transform` is not callable, otherwise it defaults to an empty string ("").

sortable

Boolean to determine whether this column should be sortable or not. Defaults to `True`.

hidden

Boolean to determine whether or not this column should be displayed when rendering the table. Default: `False`.

link

A string or callable which returns a URL which will be wrapped around this columns text as a link.

allowed_data_types

A list of data types for which the link should be created. Default is an empty list ([]).

When the list is empty and the `link` attribute is not `None`, all the rows under this column will be links.

status

Boolean designating whether or not this column represents a status (i.e. enabled/disabled, up/down, active/inactive). Default: `False`.

status_choices

A tuple of tuples representing the possible data values for the status column and their associated boolean equivalent. Positive states should equate to `True`, negative states should equate to `False`, and indeterminate states should be `None`.

Values are compared in a case-insensitive manner.

Example (these are also the default values):

```
status_choices = (
    ('enabled', True),
    ('true', True),
    ('up', True),
    ('active', True),
    ('yes', True),
    ('on', True),
    ('none', None),
    ('unknown', None),
    ('', None),
    ('disabled', False),
    ('down', False),
    ('false', False),
    ('inactive', False),
    ('no', False),
    ('off', False),
)
```

display_choices

A tuple of tuples representing the possible values to substitute the data when displayed in the column cell.

empty_value

A string or callable to be used for cells which have no data. Defaults to the string "-".

summation

A string containing the name of a summation method to be used in the generation of a summary row for this column. By default the options are "sum" or "average", which behave as expected. Optional.

filters

A list of functions (often template filters) to be applied to the value of the data for this column prior to output. This is effectively a shortcut for writing a custom `transform` function in simple cases.

classes

An iterable of CSS classes which should be added to this column. Example: `classes=('foo', 'bar')`.

attrs

A dict of HTML attribute strings which should be added to this column. Example: `attrs={"data-foo": "bar"}`.

cell_attributes_getter

A callable to get the HTML attributes of a column cell depending on the data. For example, to add additional description or help information for data in a column cell (e.g. in Images panel, for the column format):

```
helpText = {
    'ARI': 'Amazon Ramdisk Image',
    'QCOW2': 'QEMU Emulator'
}
```

(continues on next page)

(continued from previous page)

```

getHoverHelp(data):
    text = helpText.get(data, None)
    if text:
        return {'title': text}
    else:
        return {}
...
...
cell_attributes_getter = getHoverHelp

```

truncate

An integer for the maximum length of the string in this column. If the length of the data in this column is larger than the supplied number, the data for this column will be truncated and an ellipsis will be appended to the truncated data. Defaults to None.

link_classes

An iterable of CSS classes which will be added when the columns text is displayed as a link. This is left for backward compatibility. Deprecated in favor of the `link_attributes` attribute. Example: `link_classes=('link-foo', 'link-bar')`. Defaults to None.

wrap_list

Boolean value indicating whether the contents of this cell should be wrapped in a `` tag. Useful in conjunction with Django's `unordered_list` template filter. Defaults to False.

form_field

A form field used for inline editing of the column. A django `forms.Field` can be used or django `form.Widget` can be used.

Example: `form_field=forms.CharField()`. Defaults to None.

form_field_attributes

The additional html attributes that will be rendered to `form_field`. Example: `form_field_attributes={'class': 'bold_input_field'}`. Defaults to None.

update_action

The class that inherits from `tables.actions.UpdateAction`, `update_cell` method takes care of saving inline edited data. The `tables.base.Row.get_data` method needs to be connected to table for obtaining the data. Example: `update_action=UpdateCell`. Defaults to None.

link_attrs

A dict of HTML attribute strings which should be added when the columns text is displayed as a link. Examples: `link_attrs={"data-foo": "bar"}`. `link_attrs={"target": "_blank", "class": "link-foo link-bar"}`. Defaults to None.

policy_rules

List of scope and rule tuples to do policy checks on, the composition of which is (scope, rule)

- scope: service type managing the policy for action
- rule: string representing the action to be checked

for a policy that requires a single rule check, `policy_rules` should look like:

```
"(("compute", "compute:create_instance"),)"
```

for a policy that requires multiple rule checks, rules should look like:

```
"(("identity", "identity:list_users"),  
 ("identity", "identity:list_roles"))"
```

help_text

A string of simple help text displayed in a tooltip when you hover over the help icon beside the Column name. Defaults to None.

allowed(*request*)

Determine whether processing/displaying the column is allowed.

It is determined based on the current request.

get_data(*datum*)

Returns the final display data for this column from the given inputs.

The return value will be either the attribute specified for this column or the return value of the attr:~*horizon.tables.Column.transform* method for this column.

get_link_url(*datum*)

Returns the final value for the columns `link` property.

If `allowed_data_types` of this column is not empty and the datum has an assigned type, check if the datum's type is in the `allowed_data_types` list. If not, the datum won't be displayed as a link.

If `link` is a callable, it will be passed the current data object and should return a URL. Otherwise `get_link_url` will attempt to call `reverse` on `link` with the object's id as a parameter. Failing that, it will simply return the value of `link`.

get_raw_data(*datum*)

Returns the raw data for this column.

No filters or formatting are applied to the returned data. This is useful when doing calculations on data in the table.

get_summation()

Returns the summary value for the data in this column.

It returns the summary value if a valid summation method is specified for it. Otherwise returns None.

class `horizon.tables.Row`(*table*, *datum=None*)

Represents a row in the table.

When iterated, the Row instance will yield each of its cells.

Rows are capable of AJAX updating, with a little added work:

The `ajax` property needs to be set to `True`, and subclasses need to define a `get_data` method which returns a data object appropriate for consumption by the table (effectively the `get` lookup versus the table's list lookup).

The automatic update interval is configurable by setting the key `ajax_poll_interval` in the `HORIZON_CONFIG` dictionary. Default: 2500 (measured in milliseconds).

table

The table which this row belongs to.

datum

The data object which this row represents.

id

A string uniquely representing this row composed of the table name and the row data objects identifier.

cells

The cells belonging to this row stored in a `OrderedDict` object. This attribute is populated during instantiation.

status

Boolean value representing the status of this row calculated from the values of the tables `status_columns` if they are set.

status_class

Returns a css class for the status of the row based on `status`.

ajax

Boolean value to determine whether ajax updating for this row is enabled.

ajax_action_name

String that is used for the query parameter key to request AJAX updates. Generally you wont need to change this value. Default: "row_update".

ajax_cell_action_name

String that is used for the query parameter key to request AJAX updates of cell. Generally you wont need to change this value. It is also used for inline edit of the cell. Default: "cell_update".

can_be_selected(*datum*)

Determines whether the row can be selected.

By default if multiselect enabled return True. You can remove the checkbox after an ajax update here if required.

get_cells()

Returns the bound cells for this row in order.

get_data(*request, obj_id*)

Fetches the updated data for the row based on the given object ID.

Must be implemented by a subclass to allow AJAX updating.

load_cells(*datum=None*)

Load the rows data and initialize all the cells in the row.

It also set the appropriate row properties which require the rows data to be determined.

The rows data is provided either at initialization or as an argument to this function.

This function is called automatically by `__init__()` if the `datum` argument is provided. However, by not providing the data during initialization this function allows for the possibility of a two-step loading pattern when you need a row instance but dont yet have the data available.

Actions

class horizon.tables.**Action**(*args, **kwargs)

Represents an action which can be taken on this tables data.

name

Required. The short name or slug representing this action. This name should not be changed at runtime.

verbose_name

A descriptive name used for display purposes. Defaults to the value of name with the first letter of each word capitalized.

verbose_name_plural

Used like verbose_name in cases where handles_multiple is True. Defaults to verbose_name with the letter s appended.

method

The HTTP method for this action. Defaults to POST. Other methods may or may not succeed currently.

requires_input

Boolean value indicating whether or not this action can be taken without any additional input (e.g. an object id). Defaults to True.

preempt

Boolean value indicating whether this action should be evaluated in the period after the table is instantiated but before the data has been loaded.

This can allow actions which dont need access to the full table data to bypass any API calls and processing which would otherwise be required to load the table.

allowed_data_types

A list that contains the allowed data types of the action. If the datums type is in this list, the action will be shown on the row for the datum.

Default to be an empty list ([]). When set to empty, the action will accept any kind of data.

policy_rules

list of scope and rule tuples to do policy checks on, the composition of which is (scope, rule)

- scope: service type managing the policy for action
- rule: string representing the action to be checked

```
for a policy that requires a single rule check:
    policy_rules should look like
        (("compute", "compute:create_instance"),)
for a policy that requires multiple rule checks:
    rules should look like
        (("identity", "identity:list_users"),
         ("identity", "identity:list_roles"))
```

At least one of the following methods must be defined:

single(self, data_table, request, object_id)

Handler for a single-object action.

multiple(*self, data_table, request, object_ids*)

Handler for multi-object actions.

handle(*self, data_table, request, object_ids*)

If a single function can work for both single-object and multi-object cases then simply providing a `handle` function will internally route both `single` and `multiple` requests to `handle` with the calls from `single` being transformed into a list containing only the single object id.

get_param_name()

Returns the full POST parameter name for this action.

Defaults to `{{ table.name }}__{{ action.name }}`.

class `horizon.tables.LinkAction(*args, **kwargs)`

A table action which is simply a link rather than a form POST.

name

Required. The short name or slug representing this action. This name should not be changed at runtime.

verbose_name

A string which will be rendered as the link text. (Required)

url

A string or a callable which resolves to a url to be used as the link target. You must either define the `url` attribute or override the `get_link_url` method on the class.

allowed_data_types

A list that contains the allowed data types of the action. If the datums type is in this list, the action will be shown on the row for the datum.

Defaults to be an empty list (`[]`). When set to empty, the action will accept any kind of data.

get_link_url(*datum=None*)

Returns the final URL based on the value of `url`.

If `url` is callable it will call the function. If not, it will then try to call `reverse` on `url`. Failing that, it will simply return the value of `url` as-is.

When called for a row action, the current row data object will be passed as the first parameter.

class `horizon.tables.FilterAction(*args, **kwargs)`

A base class representing a filter action for a table.

name

The short name or slug representing this action. Defaults to `"filter"`.

verbose_name

A descriptive name used for display purposes. Defaults to the value of `name` with the first letter of each word capitalized.

param_name

A string representing the name of the request parameter used for the search term. Default: `"q"`.

filter_type

A string representing the type of this filter. If this is set to `"server"` then `filter_choices` must also be provided. Default: `"query"`.

filter_choices

Required for server type filters. A tuple of tuples representing the filter options. Tuple composition should evaluate to (string, string, boolean, string, boolean), representing the following:

- The first value is the filter parameter.
- The second value represents display value.
- The third optional value indicates whether or not it should be applied to the API request as an API query attribute. API type filters do not need to be accounted for in the filter method since the API will do the filtering. However, server type filters in general will need to be performed in the filter method. By default this attribute is not provided (False).
- The fourth optional value is used as help text if provided. The default is None which means no help text.
- The fifth optional value determines whether or not the choice is displayed to users. It defaults to True. This is useful when the choice needs to be displayed conditionally.

needs_preloading

If True, the filter function will be called for the initial GET request with an empty `filter_string`, regardless of the value of `method`.

filter(*table, data, filter_string*)

Provides the actual filtering logic.

This method must be overridden by subclasses and return the filtered data.

get_param_name()

Returns the full query parameter name for this action.

Defaults to `{{ table.name }}_{{ action.name }}_{{ action.param_name }}`.

get_select_options()

Provide the value, string, and `help_text` for the template to render.

`help_text` is returned if applicable.

is_api_filter(*filter_field*)

Determine if agiven filter field should be used as an API filter.

class horizon.tables.FixedFilterAction(*args, **kwargs)

A filter action with fixed buttons.

categorize(*table, rows*)

Override to separate rows into categories.

To have filtering working properly on the client, each row will need CSS class(es) beginning with `category-`, followed by the value of the fixed button.

Return a dict with a key for the value of each fixed button, and a value that is a list of rows in that category.

filter(*table, images, filter_string*)

Provides the actual filtering logic.

This method must be overridden by subclasses and return the filtered data.

get_fixed_buttons()

Returns a list of dict describing fixed buttons used for filtering.

Each list item should be a dict with the following keys:

- **text**: Text to display on the button
- **icon**: Icon class for icon element (inserted before text).
- **value**: Value returned when the button is clicked. This value is passed to `filter()` as `filter_string`.

class `horizon.tables.BatchAction(*args, **kwargs)`

A table action which takes batch action on one or more objects.

This action should not require user input on a per-object basis.

name

A short name or slug representing this action. Should be one word such as delete, add, disable, etc.

action_present()

Method returning a present action name. This is used as an action label.

Method must accept an integer/long parameter and return the display forms of the name properly pluralised (depending on the integer) and translated in a string or tuple/list.

The returned display form is highly recommended to be a complete action name with a form of a transitive verb and an object noun. Each word is capitalized and the string should be marked as translatable.

If tuple or list - then setting `self.current_present_action = n` will set the current active item from the `list(action_present[n])`

action_past()

Method returning a past action name. This is usually used to display a message when the action is completed.

Method must accept an integer/long parameter and return the display forms of the name properly pluralised (depending on the integer) and translated in a string or tuple/list.

The detail is same as that of `action_present`.

success_url

Optional location to redirect after completion of the delete action. Defaults to the current page.

help_text

Optional message for providing an appropriate help text for the horizon user.

action(request, datum_id)

Accepts a single object id and performs the specific action.

This method is required.

Return values are discarded, errors raised are caught and logged.

get_default_attrs()

Returns a list of the default HTML attributes for the action.

get_success_url(request=None)

Returns the URL to redirect to after a successful action.

update(request, datum)

Switches the action verbose name, if needed.

class horizon.tables.DeleteAction(*args, **kwargs)

A table action used to perform delete operations on table data.

name

A short name or slug representing this action. Defaults to delete

action_present()

Method returning a present action name. This is used as an action label.

Method must accept an integer/long parameter and return the display forms of the name properly pluralised (depending on the integer) and translated in a string or tuple/list.

The returned display form is highly recommended to be a complete action name with a form of a transitive verb and an object noun. Each word is capitalized and the string should be marked as translatable.

If tuple or list - then setting `self.current_present_action = n` will set the current active item from the `list(action_present[n])`

action_past()

Method returning a past action name. This is usually used to display a message when the action is completed.

Method must accept an integer/long parameter and return the display forms of the name properly pluralised (depending on the integer) and translated in a string or tuple/list.

The detail is same as that of `action_present`.

success_url

Optional location to redirect after completion of the delete action. Defaults to the current page.

help_text

Optional message for providing an appropriate help text for the horizon user.

action(request, obj_id)

Action entry point. Overrides base class action method.

Accepts a single object id passing it over to the delete method responsible for the objects destruction.

delete(request, obj_id)

Required. Deletes an object referenced by `obj_id`.

Override to provide delete functionality specific to your data.

Class-Based Views

Several class-based views are provided to make working with DataTables easier in your UI.

class horizon.tables.DataTableView(*args, **kwargs)

A class-based generic view to handle basic DataTable processing.

Three steps are required to use this view: set the `table_class` attribute with the desired `DataTable` class; define a `get_data` method which returns a set of data for the table; and specify a template for the `template_name` attribute.

Optionally, you can override the `has_more_data` method to trigger pagination handling for APIs that support it.

class `horizon.tables.MultiTableView(*args, **kwargs)`
Generic view to handle multiple `DataTable` classes in a single view.

Each `DataTable` class must be a `DataTable` class or its subclass.

Three steps are required to use this view: set the `table_classes` attribute with a tuple of the desired `DataTable` classes; define a `get_{table_name}_data` method for each table class which returns a set of data for that table; and specify a template for the `template_name` attribute.

Horizon Tabs and TabGroups

Horizon includes a set of reusable components for programmatically building tabbed interfaces with fancy features like dynamic AJAX loading and nearly effortless templating and styling.

Tab Groups

For any tabbed interface, your fundamental element is the tab group which contains all your tabs. This class provides a dead-simple API for building tab groups and encapsulates all the necessary logic behind the scenes.

class `horizon.tabs.TabGroup(request, **kwargs)`
A container class which knows how to manage and render Tab objects.

slug
The URL slug and pseudo-unique identifier for this tab group.

tabs
A list of `Tab` classes. Tabs specified here are displayed in the order of the list.

template_name
The name of the template which will be used to render this tab group. Default: "horizon/common/_tab_group.html"

sticky
Boolean to control whether the active tab state should be stored across requests for a given user. (State storage is all done client-side.)

show_single_tab
Boolean to control whether the tab bar is shown when the tab group has only one tab. Default: False

param_name
The name of the GET request parameter which will be used when requesting specific tab data. Default: `tab`.

classes
A list of CSS classes which should be displayed on this tab group.

attrs
A dictionary of HTML attributes which should be rendered into the markup for this tab group.

selected
Read-only property which is set to the instance of the currently-selected tab if there is one, otherwise None.

active

Read-only property which is set to the value of the current active tab. This may not be the same as the value of `selected` if no specific tab was requested via the GET parameter.

get_default_classes()

Returns a list of the default classes for the tab group.

Defaults to `["nav", "nav-tabs", "ajax-tabs"]`.

get_id()

Returns the id for this tab group.

Defaults to the value of the tab groups `horizon.tabs.Tab.slug`.

get_selected_tab()

Returns the tab specific by the GET request parameter.

In the event that there is no GET request parameter, the value of the query parameter is invalid, or the tab is not allowed/enabled, the return value of this function is `None`.

get_tab(tab_name, allow_disabled=False)

Returns a specific tab from this tab group.

If the tab is not allowed or not enabled this method returns `None`.

If the tab is disabled but you wish to return it anyway, you can pass `True` to the `allow_disabled` argument.

get_tabs()

Returns a list of the allowed tabs for this tab group.

load_tab_data()

Preload all data that for the tabs that will be displayed.

render()

Renders the HTML output for this tab group.

tabs_not_available()

The fallback handler if no tabs are either allowed or enabled.

In the event that no tabs are either allowed or enabled, this method is the fallback handler. By default its a no-op, but it exists to make redirecting or raising exceptions possible for subclasses.

Tabs

The tab itself is the discrete unit for a tab group, representing one view of data.

class `horizon.tabs.Tab(tab_group, request=None, policy_rules=None)`

A reusable interface for constructing a tab within a `TabGroup`.

name

The display name for the tab which will be rendered as the text for the tab element in the HTML. Required.

slug

The URL slug and id attribute for the tab. This should be unique for a given tab group. Required.

preload

Determines whether the contents of the tab should be rendered into the pages HTML when the tab group is rendered, or whether it should be loaded dynamically when the tab is selected.
Default: `True`.

classes

A list of CSS classes which should be displayed on this tab.

attrs

A dictionary of HTML attributes which should be rendered into the markup for this tab.

load

Read-only access to determine whether or not this tabs data should be loaded immediately.

permissions

A list of permission names which this tab requires in order to be displayed. Defaults to an empty list (`[]`).

allowed(*request*)

Determines whether or not the tab is displayed.

Tab instances can override this method to specify conditions under which this tab should not be shown at all by returning `False`.

enabled(*request*)

Determines whether or not the tab should be accessible.

For example, the tab should be rendered into the HTML on load and respond to a click event.

If a tab returns `False` from `enabled` it will ignore the value of `preload` and only render the HTML of the tab after being clicked.

The default behavior is to return `True` for all cases.

get_context_data(*request*, *kwargs*)**

Return a dictionary of context data used to render the tab.

Required.

get_default_classes()

Returns a list of the default classes for the tab.

Defaults to an empty list (`[]`), however additional classes may be added depending on the state of the tab as follows:

If the tab is the active tab for the tab group, in which the class "active" will be added.

If the tab is not enabled, the classes the class "disabled" will be added.

get_id()

Returns the id for this tab.

Defaults to "`{{ tab_group.slug }}_{{ tab.slug }}`".

get_template_name(*request*)

Returns the name of the template to be used for rendering this tab.

By default it returns the value of the `template_name` attribute on the `Tab` class.

is_active()

Method to access whether or not this tab is the active tab.

post(*request*, **args*, ***kwargs*)

Handles POST data sent to a tab.

Tab instances can override this method to have tab-specific POST logic without polluting the TabView code.

The default behavior is to ignore POST data.

render()

Renders the tab to HTML.

`get_context_data()` method and the `get_template_name()` method are called.

If `preload` is `False` and `force_load` is not `True`, or either `allowed()` or `enabled()` returns `False` this method will return an empty string.

class `horizon.tabs.TableTab`(*tab_group*, *request*)

A Tab class which knows how to deal with `DataTable` classes inside of it.

This distinct class is required due to the complexity involved in handling both dynamic tab loading, dynamic table updating and table actions all within one view.

table_classes

An iterable containing the `DataTable` classes which this tab will contain. Equivalent to the `table_classes` attribute on `MultiTableView`. For each table class you need to define a corresponding `get_{{ table_name }}_data` method as with `MultiTableView`.

get_context_data(*request*, ***kwargs*)

Adds a `{{ table_name }}_table` item to the context for each table.

The target tables are specified by the `table_classes` attribute.

If only one table class is provided, a shortcut `table` context variable is also added containing the single table.

load_table_data()

Calls the `get_{{ table_name }}_data` methods for each table class.

When returning, the loaded data is set on the tables.

TabView

There is also a useful and simple generic class-based view for handling the display of a `TabGroup` class.

class `horizon.tabs.TabView`

A generic view for displaying a `horizon.tabs.TabGroup`.

This view handles selecting specific tabs and deals with AJAX requests gracefully.

tab_group_class

The only required attribute for `TabView`. It should be a class which inherits from `horizon.tabs.TabGroup`.

get_context_data(***kwargs*)

Adds the `tab_group` variable to the context data.

get_tabs(*request*, ***kwargs*)

Returns the initialized tab group for this view.

handle_tabbed_response(*tab_group, context*)

Sends back an AJAX-appropriate response for the tab group if needed.

Otherwise renders the response as normal.

class `horizon.tabs.TabbedTableView`(*args, **kwargs)

get_tables()

A no-op on this class. Tables are handled at the tab level.

handle_table(*table_dict*)

Loads the table data based on a given `table_dict` and handles them.

For the given dict containing a `DataTable` and a `TableTab` instance, it loads the table data for that tab and calls the `maybe_handle()` method. The return value will be the result of `maybe_handle`.

load_tabs()

Loads the tab group.

It compiles the table instances for each table attached to any `horizon.tabs.TableTab` instances on the tab group. This step is necessary before processing any tab or table actions.

Horizon Forms

Horizon ships with some very useful base form classes, form fields, class-based views, and javascript helpers which streamline most of the common tasks related to form handling.

Form Classes

class `horizon.forms.base.DateField`(*args, **kwargs)

A simple form for selecting a range of time.

class `horizon.forms.base.SelfHandlingForm`(*request, *args, **kwargs*)

A base Form class which includes processing logic in its subclasses.

api_error(*message*)

Adds an error to the forms error dictionary.

It can be used after validation based on problems reported via the API. This is useful when you wish for API errors to appear as errors on the form rather than using the messages framework.

set_warning(*message*)

Sets a warning on the form.

Unlike `NON_FIELD_ERRORS`, this doesn't fail form validation.

Form Fields

class `horizon.forms.fields.ChoiceInput`(*name, value, attrs, choice, index*)

ChoiceInput class from django 1.10.7 codebase

An object used by ChoiceFieldRenderer that represents a single `<input type=$input_type>`.

class `horizon.forms.fields.DynamicChoiceField`(*add_item_link=None, add_item_link_args=None, *args, **kwargs*)

ChoiceField that make dynamically updating its elements easier.

Notably, the field declaration takes an extra argument, `add_item_link` which may be a string or callable defining the URL that should be used for the add link associated with the field.

widget

alias of `horizon.forms.fields.DynamicSelectWidget`

class `horizon.forms.fields.DynamicSelectWidget`(*attrs=None, choices=(), data_attrs=(), transform=None, transform_html_attrs=None*)

Select widget to handle dynamic changes to the available choices.

A subclass of the `Select` widget which renders extra attributes for use in callbacks to handle dynamic changes to the available choices.

render(**args, **kwargs*)

Render the widget as an HTML string.

class `horizon.forms.fields.DynamicTypedChoiceField`(*add_item_link=None, add_item_link_args=None, *args, **kwargs*)

Simple mix of `DynamicChoiceField` and `TypedChoiceField`.

class `horizon.forms.fields.ExternalFileField`(**args, **kwargs*)

Special `FileField` to upload file to some external location.

This is a special flavor of `FileField` which is meant to be used in cases when instead of uploading file to Django it should be uploaded to some external location, while the form validation is done as usual. It should be paired with `ExternalUploadMeta` metaclass embedded into the Form class.

class `horizon.forms.fields.ExternalUploadMeta`(*name, bases, attrs*)

Metaclass to process `ExternalFileField` fields in a specific way.

Set this class as the metaclass of a form that contains `ExternalFileField` in order to process `ExternalFileField` fields in a specific way. A hidden `CharField` twin of `FieldField` is created which contains just the filename (if any file was selected on browser side) and a special `clean` method for `FileField` is defined which extracts just file name. This allows to avoid actual file upload to Django server, yet process form `clean()` phase as usual. Actual file upload happens entirely on client-side.

class `horizon.forms.fields.IPField`(**args, **kwargs*)

Form field for entering IP/range values, with validation.

Supports IPv4/IPv6 in the format: `.. xxx.xxx.xxx.xxx .. xxx.xxx.xxx.xxx/zz ..`
`fff:fff:fff:fff:fff:fff:fff:fff .. fff:fff:fff:fff:fff:fff:fff/zz` and all compressed forms.
Also the short forms are supported: `xxx/yy xxx.xxx/yy`

version

Specifies which IP version to validate, valid values are 1 (fields.IPv4), 2 (fields.IPv6) or both - 3 (fields.IPv4 | fields.IPv6). Defaults to IPv4 (1)

mask

Boolean flag to validate subnet masks along with IP address. E.g: 10.0.0.1/32

mask_range_from

Subnet range limitation, e.g. 16

That means the input mask will be checked to be in the range 16:max_value. Useful to limit the subnet ranges to A/B/C-class networks.

clean(value)

Validate the given value and return its cleaned value as an appropriate Python object. Raise ValidationError for any errors.

```
class horizon.forms.fields.MACAddressField(*, required=True, widget=None, label=None,
initial=None, help_text="",
error_messages=None,
show_hidden_initial=False, validators=(),
localize=False, disabled=False,
label_suffix=None)
```

Form field for entering a MAC address with validation.

Supports all formats known by netaddr.EUI(), for example: .. xx:xx:xx:xx:xx:xx .. xx-xx-xx-xx-xx-xx .. xxxx.xxxx.xxxx

clean(value)

Validate the given value and return its cleaned value as an appropriate Python object. Raise ValidationError for any errors.

```
class horizon.forms.fields.MultiIPField(*args, **kwargs)
```

Extends IPField to allow comma-separated lists of addresses.

clean(value)

Validate the given value and return its cleaned value as an appropriate Python object. Raise ValidationError for any errors.

```
class horizon.forms.fields.SelectWidget(attrs=None, choices=(), data_attrs=(),
transform=None, transform_html_attrs=None)
```

Custom select widget.

It allows to render data-xxx attributes from choices. This widget also allows user to specify additional html attributes for choices.

data_attrs

Specifies object properties to serialize as data-xxx attribute. If passed (id,), this will be rendered as: <option data-id=123>option_value</option> where 123 is the value of choice_value.id

transform

A callable used to render the display value from the option object.

transform_html_attrs

A callable used to render additional HTML attributes for the option object. It returns a dictionary containing the html attributes and their values. For example, to define a title attribute for the choices:

```

helpText = { 'Apple': 'This is a fruit',
             'Carrot': 'This is a vegetable' }

def get_title(data):
    text = helpText.get(data, None)
    if text:
        return {'title': text}
    else:
        return {}

....
....

widget=forms.ThemableSelect( attrs={'class': 'switchable',
                                   'data-slug': 'source'},
                             transform_html_attrs=get_title )

self.fields[<field name>].choices =
    ([
        ('apple', 'Apple'),
        ('carrot', 'Carrot')
    ])

```

build_attrs(*extra_attrs=None, **kwargs*)

Helper function for building an attribute dictionary.

render(*name, value, attrs=None, renderer=None*)

Render the widget as an HTML string.

class horizon.forms.fields.**SubWidget**(*parent_widget, name, value, attrs, choices*)

SubWidget class from django 1.10.7 codebase

Some widgets are made of multiple HTML elements namely, RadioSelect. This is a class that represents the inner HTML element of a widget.

class horizon.forms.fields.**ThemableCheckboxChoiceInput**(**args, **kwargs*)

class horizon.forms.fields.**ThemableCheckboxInput**(*attrs=None, check_test=None*)

Checkbox widget which renders extra markup.

It is used to allow a custom checkbox experience.

render(*name, value, attrs=None, renderer=None*)

Render the widget as an HTML string.

class horizon.forms.fields.**ThemableChoiceField**(**, choices=(), **kwargs*)

Bootstrap based select field.

widget

alias of *horizon.forms.fields.ThemableSelectWidget*

class horizon.forms.fields.**ThemableDynamicChoiceField**(*add_item_link=None,
add_item_link_args=None,
*args, **kwargs*)

widget

alias of `horizon.forms.fields.ThemableDynamicSelectWidget`

```
class horizon.forms.fields.ThemableDynamicSelectWidget(attrs=None, choices=(),
                                                       data_attrs=(),
                                                       transform=None,
                                                       transform_html_attrs=None)
```

```
class horizon.forms.fields.ThemableDynamicTypedChoiceField(add_item_link=None,
                                                            add_item_link_args=None,
                                                            *args, **kwargs)
```

Simple mix of ThemableDynamicChoiceField & TypedChoiceField.

```
class horizon.forms.fields.ThemableSelectWidget(attrs=None, choices=(), data_attrs=(),
                                                transform=None,
                                                transform_html_attrs=None)
```

Bootstrap base select field widget.

```
render(name, value, attrs=None, renderer=None, choices=())
    Render the widget as an HTML string.
```

Form Views

```
class horizon.forms.views.ModalBackdropMixin(*args, **kwargs)
    Mixin class to allow ModalFormView and WorkflowView together.
```

This mixin class is to be used for together with ModalFormView and WorkflowView classes to augment them with modal_backdrop context data.

```
class horizon.forms.views.ModalFormMixin(*args, **kwargs)
```

```
class horizon.forms.views.ModalFormView(*args, **kwargs)
    The main view class for all views which handle forms in Horizon.
```

All view which handles forms in Horizon should inherit this class. It takes care of all details with processing `SelfHandlingForm` classes, and modal concerns when the associated template inherits from `horizon/common/_modal_form.html`.

Subclasses must define a `form_class` and `template_name` attribute at minimum.

See Django's documentation on the `FormView` class for more details.

```
form_invalid(form)
    If the form is invalid, render the invalid form.
```

```
form_valid(form)
    If the form is valid, redirect to the supplied URL.
```

```
get_context_data(**kwargs)
    Insert the form into the context dict.
```

```
get_form(form_class=None)
    Returns an instance of the form to be used in this view.
```

```
get_object_display(obj)
    Returns the display name of the created object.
```

For dynamic insertion of resources created in modals, this method returns the display name of the created object. Defaults to returning the `name` attribute.

`get_object_id(obj)`

Returns the ID of the created object.

For dynamic insertion of resources created in modals, this method returns the id of the created object. Defaults to returning the `id` attribute.

Forms Javascript

Switchable Fields

By marking fields with the "switchable" and "switched" classes along with defining a few data attributes you can programmatically hide, show, and rename fields in a form.

The triggers are fields using a `select` input widget, marked with the `switchable` class, and defining a `data-slug` attribute. When they are changed, any input with the "switched" class and defining a "data-switch-on" attribute which matches the select inputs "data-slug" attribute will be evaluated for necessary changes. In simpler terms, if the "switched" target inputs "switch-on" matches the "slug" of the "switchable" trigger input, it gets switched. Simple, right?

The "switched" inputs also need to define states. For each state in which the input should be shown, it should define a data attribute like the following: `data-<slug>-<value>=<desired label>`". When the switch event happens the value of the "switchable" field will be compared to the data attributes and the correct label will be applied to the field. If a corresponding label for that value is *not* found, the field will be hidden instead.

A simplified example is as follows:

```
source = forms.ChoiceField(
    label=_('Source'),
    choices=[
        ('cidr', _('CIDR')),
        ('sg', _('Security Group'))
    ],
    widget=forms.ThemableSelectWidget(attrs={
        'class': 'switchable',
        'data-slug': 'source'
    })
)

cidr = fields.IPField(
    label=_('CIDR'),
    required=False,
    widget=forms.TextInput(attrs={
        'class': 'switched',
        'data-switch-on': 'source',
        'data-source-cidr': _('CIDR')
    })
)
```

(continues on next page)

(continued from previous page)

```

security_group = forms.ChoiceField(
    label=_('Security Group'),
    required=False,
    widget=forms.ThemableSelectWidget(attrs={
        'class': 'switched',
        'data-switch-on': 'source',
        'data-source-sg': _('Security Group')
    })
)

```

That code would create the "switchable" control field source, and the two "switched" fields cidr and security_group which are hidden or shown depending on the value of source.

Note: A field can only safely define one slug in its "switch-on" attribute. While switching on multiple fields is possible, the behavior is very hard to predict due to the events being fired from the various switchable fields in order. You generally end up just having it hidden most of the time by accident, so its not recommended. Instead just add a second field to the form and control the two independently, then merge their results in the forms clean or handle methods at the end.

Horizon Middleware

HorizonMiddleware

class horizon.middleware.**HorizonMiddleware**(*get_response*)

The main Horizon middleware class. Required for use of Horizon.

process_exception(*request, exception*)

Catches internal Horizon exception classes.

Exception classes such as NotAuthorized, NotFound and Http302 are caught and handles them gracefully.

OperationLogMiddleware

class horizon.middleware.**OperationLogMiddleware**(*get_response*)

Middleware to output operation log.

This log can includes information below:

- domain name
- domain id
- project name
- project id
- user name
- user id

- request scheme
- referer url
- request url
- message
- method
- http status
- request parameters

and log format is defined in OPERATION_LOG_OPTIONS.

process_exception(*request, exception*)
Log error info when exception occurred.

Horizon Context Processors

Context processors used by Horizon.

`horizon.context_processors.horizon`(*request*)

The main Horizon context processor. Required for Horizon to function.

It adds the Horizon config to the context as well as setting the names `True` and `False` in the context to their boolean equivalents for convenience.

Warning: Dont put API calls in context processors; they will be called once for each template/template fragment which takes context that is used to render the complete output.

Horizon Decorators

General-purpose decorators for use with Horizon.

`horizon.decorators.require_auth`(*view_func*)

Performs user authentication check.

Similar to Django's `login_required` decorator, except that this throws `NotAuthenticated` exception if the user is not signed-in.

`horizon.decorators.require_component_access`(*view_func, component*)

Perform component `can_access` check to access the view.

:param component containing the view (panel or dashboard).

Raises a `NotAuthorized` exception if the user cannot access the component containing the view. By example the check of component policy rules will be applied to its views.

`horizon.decorators.require_perms`(*view_func, required*)

Enforces permission-based access controls.

Parameters required (*list*) A tuple of permission names, all of which the request user must possess in order access the decorated view.

Example usage:

```

from horizon.decorators import require_perms

@require_perms(['foo.admin', 'foo.member'])
def my_view(request):
    ...

```

Raises a *NotAuthorized* exception if the requirements are not met.

Horizon Exceptions

Exceptions raised by the Horizon code and the machinery for handling them.

exception `horizon.exceptions.AlreadyExists`(*name*, *resource_type*)
API resources tried to create already exists.

exception `horizon.exceptions.BadRequest`
Generic error to replace all BadRequest-type API errors.

exception `horizon.exceptions.ConfigurationError`
Exception to be raised when invalid settings have been provided.

exception `horizon.exceptions.Conflict`
Generic error to replace all Conflict-type API errors.

exception `horizon.exceptions.GetFileError`(*name*, *resource_type*)
Exception to be raised when the value of `get_file` is not expected.

The expected values start with `https://` or `http://`. Otherwise this exception will be raised.

exception `horizon.exceptions.HandledException`(*wrapped*)
Used internally to track exceptions that are already handled.

It is used to track exceptions that have gone through `horizon.exceptions.handle()` more than once.

exception `horizon.exceptions.HorizonException`
Base exception class for distinguishing our own exception classes.

class `horizon.exceptions.HorizonReporterFilter`
Error report filter that's always active, even in DEBUG mode.

is_active(*request*)

This filter is to add safety in production environments (i.e. `DEBUG` is `False`). If `DEBUG` is `True` then your site is not safe anyway. This hook is provided as a convenience to easily activate or deactivate the filter on a per request basis.

exception `horizon.exceptions.Http302`(*location*, *message=None*)
Exception used to redirect at the middleware level.

This error class which can be raised from within a handler to cause an early bailout and redirect at the middleware level.

exception `horizon.exceptions.MessageFailure`
Exception raised during message notification.

exception `horizon.exceptions.NotAuthenticated`

Raised when a user is trying to make requests and they are not logged in.

The included `HorizonMiddleware` catches `NotAuthenticated` and handles it gracefully by displaying an error message and redirecting the user to a login page.

exception `horizon.exceptions.NotAuthorized`

User tries to access a resource without sufficient permissions.

Raised whenever a user attempts to access a resource which they do not have permission-based access to (such as when failing the `require_perms()` decorator).

The included `HorizonMiddleware` catches `NotAuthorized` and handles it gracefully by displaying an error message and redirecting the user to a login page.

exception `horizon.exceptions.NotAvailable`

Exception to be raised when something is not available.

exception `horizon.exceptions.NotFound`

Generic error to replace all Not Found-type API errors.

exception `horizon.exceptions.RecoverableError`

Generic error to replace any Recoverable-type API errors.

exception `horizon.exceptions.ServiceCatalogException`(*service_name*)

A requested service is not available in the ServiceCatalog.

ServiceCatalog is fetched from Keystone.

exception `horizon.exceptions.WorkflowError`

Exception to be raised when something goes wrong in a workflow.

exception `horizon.exceptions.WorkflowValidationError`

Exception raised during workflow validation.

It is raised if required data is missing, or existing data is not valid.

`horizon.exceptions.check_message`(*keywords, message*)

Checks an exception for given keywords and raises an error if found.

It raises a new `ActionError` with the desired message if the keywords are found. This allows selective control over API error messages.

`horizon.exceptions.handle`(*request, message=None, redirect=None, ignore=False, escalate=False, log_level=None, force_log=None, details=None*)

Centralized error handling for Horizon.

Because Horizon consumes so many different APIs with completely different `Exception` types, its necessary to have a centralized place for handling exceptions which may be raised.

Exceptions are roughly divided into 3 types:

1. `UNAUTHORIZED`: Errors resulting from authentication or authorization problems. These result in being logged out and sent to the login screen.
2. `NOT_FOUND`: Errors resulting from objects which could not be located via the API. These generally result in a user-facing error message, but are otherwise returned to the normal code flow. Optionally a `redirect` value may be passed to the error handler so users are returned to a different view than the one requested in addition to the error message.

3. RECOVERABLE: Generic API errors which generate a user-facing message but drop directly back to the regular code flow.

All other exceptions bubble the stack as normal unless the `ignore` argument is passed in as `True`, in which case only unrecognized errors are bubbled.

If the exception is not re-raised, an appropriate wrapper exception class indicating the type of exception that was encountered will be returned. If `details` is `None` (default), take it from `exception.sys.exc_info`. If `details` is other string, then use that string explicitly or if `details` is empty then suppress it.

Horizon TestCase Classes

Horizon provides a base test case class which provides several useful pre-prepared attributes for testing Horizon components.

class `horizon.test.helpers.TestCase`(*methodName='runTest'*)

Base test case class for Horizon with numerous additional features.

- A `RequestFactory` class which supports Django's `contrib.messages` framework via `self.factory`.
- A ready-to-go request object via `self.request`.

assertMessageCount(*response=None, **kwargs*)

Asserts that the expected number of messages have been attached.

The expected number of messages can be specified per message type. Usage would look like `self.assertMessageCount(success=1)`.

assertNoMessages(*response=None*)

Asserts no messages have been attached by the messages framework.

The expected messages framework is `django.contrib.messages`.

setUp()

Hook method for setting up the test fixture before exercising it.

tearDown()

Hook method for deconstructing the test fixture after testing it.

The OpenStack Dashboard also provides test case classes for greater ease-of-use when testing APIs and OpenStack-specific auth scenarios.

class `openstack_dashboard.test.helpers.TestCase`(*methodName='runTest'*)

Specialized base test case class for Horizon.

It gives access to numerous additional features:

- A full suite of test data through various attached objects and managers (e.g. `self.servers`, `self.user`, etc.). See the docs for `TestData` for more information.
- A set of request context data via `self.context`.
- A `RequestFactory` class which supports Django's `contrib.messages` framework via `self.factory`.
- A ready-to-go request object via `self.request`.
- The ability to override specific time data controls for easier testing.

- Several handy additional assertion methods.

assertFormErrors(*response*, *count=0*, *message=None*, *context_name='form'*)

Check for form errors.

Asserts that the response does contain a form in its context, and that form has errors, if count were given, it must match the exact numbers of errors

assertNoFormErrors(*response*, *context_name='form'*)

Checks for no form errors.

Asserts that the response either does not contain a form in its context, or that if it does, that form has no errors.

assertNoWorkflowErrors(*response*, *context_name='workflow'*)

Checks for no workflow errors.

Asserts that the response either does not contain a workflow in its context, or that if it does, that workflow has no errors.

assertRedirectsNoFollow(*response*, *expected_url*)

Check for redirect.

Asserts that the given response issued a 302 redirect without processing the view which is redirected to.

assertStatusCode(*response*, *expected_code*)

Validates an expected status code.

Matches camel case of other assert functions

assertWorkflowErrors(*response*, *count=0*, *message=None*, *context_name='workflow'*)

Check for workflow errors.

Asserts that the response does contain a workflow in its context, and that workflow has errors, if count were given, it must match the exact numbers of errors

setUp()

Hook method for setting up the test fixture before exercising it.

tearDown()

Hook method for deconstructing the test fixture after testing it.

class `openstack_dashboard.test.helpers.APITestCase`(*methodName='runTest'*)

setUp()

Hook method for setting up the test fixture before exercising it.

class `openstack_dashboard.test.helpers.BaseAdminViewTests`(*methodName='runTest'*)

Sets an active user with the admin role.

For testing admin-only views and functionality.

openstack_auth Module

The Backend Module

Module defining the Django auth backend class for the Keystone API.

class openstack_auth.backend.KeystoneBackend

Django authentication backend for use with `django.contrib.auth`.

authenticate(*request*, *auth_url=None*, ***kwargs*)

Authenticates a user via the Keystone Identity API.

get_all_permissions(*user*, *obj=None*)

Returns a set of permission strings that the user has.

This permission available to the user is derived from the users Keystone roles.

The permissions are returned as "`openstack.{{ role.name }}`".

get_group_permissions(*user*, *obj=None*)

Returns an empty set since Keystone doesnt support groups.

get_user(*user_id*)

Returns the current user from the session data.

If authenticated, this return the user object based on the user ID and session data.

Note: This required monkey-patching the `contrib.auth` middleware to make the `request` object available to the auth backend class.

has_module_perms(*user*, *app_label*)

Returns True if user has any permissions in the given `app_label`.

Currently this matches for the `app_label` "openstack".

has_perm(*user*, *perm*, *obj=None*)

Returns True if the given user has the specified permission.

The Forms Module

class openstack_auth.forms.DummyAuth(*user_id*)

A dummy Auth object

It is needed for `_KeystoneAdapter` to get the `user_id` from, but otherwise behaves as if it doesnt exist (is falsy).

get_headers(*session*, ***kwargs*)

Fetch authentication headers for message.

This is a more generalized replacement of the older `get_token` to allow plugins to specify different or additional authentication headers to the OpenStack standard X-Auth-Token header.

How the authentication headers are obtained is up to the plugin. If the headers are still valid they may be re-used, retrieved from cache or the plugin may invoke an authentication request against a server.

The default implementation of `get_headers` calls the `get_token` method to enable older style plugins to continue functioning unchanged. Subclasses should feel free to completely override this function to provide the headers that they want.

There are no required kwargs. They are passed directly to the auth plugin and they are implementation specific.

Returning `None` will indicate that no token was able to be retrieved and that authorization was a failure. Adding no authentication data can be achieved by returning an empty dictionary.

Parameters `session` (`keystoneauth1.session.Session`) The session object that the `auth_plugin` belongs to.

Returns Headers that are set to authenticate a message or `None` for failure. Note that when checking this value that the empty dict is a valid, non-failure response.

Return type dict

class `openstack_auth.forms.Login(*args, **kwargs)`

Form used for logging in a user.

Handles authentication with Keystone by providing the domain name, username and password. A scoped token is fetched after successful authentication.

A domain name is required if authenticating with Keystone V3 running multi-domain configuration.

If the user authenticated has a default project set, the token will be automatically scoped to their default project.

If the user authenticated has no default project set, the authentication backend will try to scope to the projects returned from the users assigned projects. The first successful project scoped will be returned.

Inherits from the base `django.contrib.auth.forms.AuthenticationForm` class for added security features.

clean()

Hook for doing any extra form-wide cleaning after `Field.clean()` has been called on every field. Any `ValidationError` raised by this method will not be associated with a particular field; it will have a special-case association with the field named `__all__`.

class `openstack_auth.forms.Password(*args, **kwargs)`

Form used for changing users password without having to log in.

clean()

Hook for doing any extra form-wide cleaning after `Field.clean()` has been called on every field. Any `ValidationError` raised by this method will not be associated with a particular field; it will have a special-case association with the field named `__all__`.

The User Module

class `openstack_auth.user.Token`(*auth_ref*, *unscoped_token=None*)

Encapsulates the `AccessInfo` object from `keystoneclient`.

Token object provides a consistent interface for accessing the keystone token information and service catalog.

Added for maintaining backward compatibility with horizon that expects `Token` object in the user object.

class `openstack_auth.user.User`(*id=None*, *token=None*, *user=None*, *tenant_id=None*, *service_catalog=None*, *tenant_name=None*, *roles=None*, *authorized_tenants=None*, *endpoint=None*, *enabled=False*, *services_region=None*, *user_domain_id=None*, *user_domain_name=None*, *domain_id=None*, *domain_name=None*, *project_id=None*, *project_name=None*, *is_federated=False*, *unscoped_token=None*, *password=None*, *password_expires_at=None*)

A `User` class with some extra special sauce for Keystone.

In addition to the standard Django user attributes, this class also has the following:

token

The Keystone token object associated with the current user/tenant.

The token object is deprecated, use `auth_ref` instead.

tenant_id

The id of the Keystone tenant for the current user/token.

The `tenant_id` keyword argument is deprecated, use `project_id` instead.

tenant_name

The name of the Keystone tenant for the current user/token.

The `tenant_name` keyword argument is deprecated, use `project_name` instead.

project_id

The id of the Keystone project for the current user/token.

project_name

The name of the Keystone project for the current user/token.

service_catalog

The `ServiceCatalog` data returned by Keystone.

roles

A list of dictionaries containing role names and ids as returned by Keystone.

services_region

A list of non-identity service endpoint regions extracted from the service catalog.

user_domain_id

The domain id of the current user.

user_domain_name

The domain name of the current user.

domain_id

The id of the Keystone domain scoped for the current user/token.

is_federated

Whether user is federated Keystone user. (Boolean)

unscoped_token

Unscoped Keystone token.

password_expires_at

Password expiration date.

exception DoesNotExist

exception MultipleObjectsReturned

property authorized_tenants

Returns a memoized list of tenants this user may access.

property available_services_regions

Returns list of unique region name values in service catalog.

has_a_matching_perm(*perm_list*, *obj=None*)

Returns True if the user has one of the specified permissions.

If object is passed, it checks if the user has any of the required perms for this object.

has_perms(*perm_list*, *obj=None*)

Returns True if the user has all of the specified permissions.

Tuples in the list will possess the required permissions if the user has a permissions matching one of the elements of that tuple

property is_active

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

property is_anonymous

Return if the user is not authenticated.

Returns True if not authenticated, "False" otherwise.

property is_authenticated

Checks for a valid authentication.

property is_superuser

Evaluates whether this user has admin privileges.

Returns True or False.

is_token_expired(*margin=None*)

Determine if the token is expired.

Returns True if the token is expired, False if not, and None if there is no token set.

Parameters margin A security time margin in seconds before real expiration.

Will return True if the token expires in less than margin seconds of time. A

default margin can be set by the `TOKEN_TIMEOUT_MARGIN` in the django settings.

save(*args, **kwargs)

Save the current instance. Override this in a subclass if you want to control the saving process.

The `force_insert` and `force_update` parameters can be used to insist that the save must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

time_until_expiration()

Returns the number of remaining days until users password expires.

Calculates the number days until the user must change their password, once the password expires the user will not able to log in until an admin changes its password.

The Utils Module

`openstack_auth.utils.LOG = <Logger openstack_auth.utils (WARNING)>`

We need the request object to get the user, so well slightly modify the existing `django.contrib.auth.get_user` method. To do so we update the auth middleware to point to our overridden method.

Calling `patch_middleware_get_user` is done in our custom middleware at `openstack_auth.middleware` to monkeypatch the code in before it is needed.

`openstack_auth.utils.allow_expired_passowrd_change()`

Checks if users should be able to change their expired passwords.

`openstack_auth.utils.build_absolute_uri(request, relative_url)`

Ensure absolute_uri are relative to WEBROOT.

`openstack_auth.utils.clean_up_auth_url(auth_url)`

Clean up the auth url to extract the exact Keystone URL

`openstack_auth.utils.default_services_region(service_catalog, request=None, ks_endpoint=None)`

Return the default service region.

Order of precedence: 1. `services_region` cookie value 2. Matching endpoint in `DEFAULT_SERVICE_REGIONS` 3. * key in `DEFAULT_SERVICE_REGIONS` 4. First valid region from catalog

In each case the value must also be present in `available_regions` or we move to the next level of precedence.

`openstack_auth.utils.fix_auth_url_version_prefix(auth_url)`

Fix up the auth url if an invalid or no version prefix was given.

Fix the URL to say v3 in this case and add version if it is missing entirely. This should be smarter and use discovery. Until version discovery is implemented we need this method to get everything working.

`openstack_auth.utils.get_admin_permissions()`

Common function for getting the admin permissions from settings

This format is `openstack.roles.xxx` and xxx is a real role name.

Returns

Set object including all admin permission. If there is no permission, this will return empty:

```
{
  "openstack.roles.foo",
  "openstack.roles.bar",
  "openstack.roles.admin"
}
```

`openstack_auth.utils.get_admin_roles()`

Common function for getting the admin roles from settings

Returns

Set object including all admin roles. If there is no role, this will return empty:

```
{
  "foo", "bar", "admin"
}
```

`openstack_auth.utils.get_client_ip(request)`

Return client ip address using `SECURE_PROXY_ADDR_HEADER` variable.

If not present or not defined on settings then `REMOTE_ADDR` is used.

Parameters `request` (*django.http.HttpRequest*) Django http request object.

Returns Possible client ip address

Return type string

`openstack_auth.utils.get_endpoint_region(endpoint)`

Common function for getting the region from endpoint.

In Keystone V3, region has been deprecated in favor of `region_id`.

This method provides a way to get region that works for both Keystone V2 and V3.

`openstack_auth.utils.get_role_permission(role)`

Common function for getting the permission froms arg

This format is `openstack.roles.xxx` and `xxx` is a real role name.

Returns String like `openstack.roles.admin` If role is None, this will return None.

`openstack_auth.utils.get_websso_url(request, auth_url, websso_auth)`

Return the keystone endpoint for initiating WebSSO.

Generate the keystone WebSSO endpoint that will redirect the user to the login page of the federated identity provider.

Based on the authentication type selected by the user in the login form, it will construct the keystone WebSSO endpoint.

Parameters

- **request** (*django.http.HttpRequest*) Django http request object.

- **auth_url** (*string*) Keystone endpoint configured in the horizon setting. If `WEBSSO_KEYSTONE_URL` is defined, its value will be used. Otherwise, the value is derived from: `- OPENSTACK_KEYSTONE_URL - AVAILABLE_REGIONS`
- **websso_auth** (*string*) Authentication type selected by the user from the login form. The value is derived from the horizon setting `WEBSSO_CHOICES`.

Example of horizon WebSSO setting:

```
WEBSSO_CHOICES = (
    ("credentials", "Keystone Credentials"),
    ("oidc", "OpenID Connect"),
    ("saml2", "Security Assertion Markup Language"),
    ("acme_oidc", "ACME - OpenID Connect"),
    ("acme_saml2", "ACME - SAML2")
)

WEBSSO_IDP_MAPPING = {
    "acme_oidc": ("acme", "oidc"),
    "acme_saml2": ("acme", "saml2")
}
```

The value of `websso_auth` will be looked up in the `WEBSSO_IDP_MAPPING` dictionary, if a match is found it will return a IdP specific WebSSO endpoint using the values found in the mapping.

The value in `WEBSSO_IDP_MAPPING` is expected to be a tuple formatted as (`<idp_id>`, `<protocol_id>`). Using the values found, a IdP/protocol specific URL will be constructed:

```
/auth/OS-FEDERATION/identity_providers/<idp_id>
/protocols/<protocol_id>/websso
```

If no value is found from the `WEBSSO_IDP_MAPPING` dictionary, it will treat the value as the global WebSSO protocol `<protocol_id>` and construct the WebSSO URL by:

```
/auth/OS-FEDERATION/websso/<protocol_id>
```

Returns Keystone WebSSO endpoint.

Return type string

`openstack_auth.utils.has_in_url_path(url, subs)`

Test if any of `subs` strings is present in the `url` path.

`openstack_auth.utils.is_token_valid(token, margin=None)`

Timezone-aware checking of the auth tokens expiration timestamp.

Returns True if the token has not yet expired, otherwise False.

Parameters

- **token** The `openstack_auth.user.Token` instance to check
- **margin** A time margin in seconds to subtract from the real tokens validity. An example usage is that the token can be valid once the middleware passed,

and invalid (timed-out) during a view rendering and this generates authorization errors during the view rendering. A default margin can be set by the `TOKEN_TIMEOUT_MARGIN` in the django settings.

`openstack_auth.utils.set_response_cookie(response, cookie_name, cookie_value)`
Common function for setting the cookie in the response.

Provides a common policy of setting cookies for last used project and region, can be reused in other locations.

This method will set the cookie to expire in 365 days.

`openstack_auth.utils.store_initial_k2k_session(auth_url, request, scoped_auth_ref, unscoped_auth_ref)`

Stores session variables if there are k2k service providers

This stores variables related to Keystone2Keystone federation. This function gets skipped if there are no Keystone service providers. An unscoped token to the identity provider keystone gets stored so that it can be used to do federated login into the service providers when switching keystone providers. The settings file can be configured to set the display name of the local (identity provider) keystone by setting `KEYSTONE_PROVIDER_IDP_NAME`. The `KEYSTONE_PROVIDER_IDP_ID` settings variable is used for comparison against the service providers. It should not conflict with any of the service provider ids.

Parameters

- **auth_url** base token auth url
- **request** Django http request object
- **scoped_auth_ref** Scoped Keystone access info object
- **unscoped_auth_ref** Unscoped Keystone access info object

`openstack_auth.utils.url_path_replace(url, old, new, count=None)`
Return a copy of url with replaced path.

Return a copy of url with all occurrences of old replaced by new in the url path. If the optional argument count is given, only the first count occurrences are replaced.

The Views Module

class `openstack_auth.views.PasswordView(**kwargs)`
Changes users password when its expired or otherwise inaccessible.

form_class
alias of `openstack_auth.forms.Password`

form_valid(form)
If the form is valid, redirect to the supplied URL.

get_initial()
Return the initial data to use for forms on this view.

`openstack_auth.views.login(request)`
Logs a user in using the `Login` form.

`openstack_auth.views.logout(request, login_url=None, **kwargs)`
Logs out the user if he is logged in. Then redirects to the log-in page.

Parameters

- **login_url** Once logged out, defines the URL where to redirect after login
- **kwargs** see `django.contrib.auth.views.logout_then_login` extra parameters.

`openstack_auth.views.switch(request, tenant_id, redirect_field_name='next')`

Switches an authenticated user from one project to another.

`openstack_auth.views.switch_keystone_provider(request, keystone_provider=None, redirect_field_name='next')`

Switches the users keystone provider using K2K Federation

If `keystone_provider` is given then we switch the user to the keystone provider using K2K federation. Otherwise if `keystone_provider` is `None` then we switch the user back to the Identity Provider Keystone which a non federated token auth will be used.

`openstack_auth.views.switch_region(request, region_name, redirect_field_name='next')`

Switches the users region for all services except Identity service.

The region will be switched if the given region is one of the regions available for the scoped project. Otherwise the region is not switched.

`openstack_auth.views.websso(request)`

Logs a user in using a token from Keystones POST.

3.1.9 Frequently Asked Questions

What is the relationship between Dashboards, Panels, and navigation? The navigational structure is strongly encouraged to flow from Dashboard objects as top-level navigation items to Panel objects as sub-navigation items as in the current implementation. Template tags are provided to automatically generate this structure.

That said, you are not required to use the provided tools and can write templates and URLconfs by hand to create any desired structure.

Does a panel have to be an app in INSTALLED_APPS? A panel can live in any Python module. It can be a standalone which ties into an existing dashboard, or it can be contained alongside others within a larger dashboard app. There is no strict enforcement here. Python is a language for consenting adults. A module containing a Panel does not need to be added to `INSTALLED_APPS`, but this is a common and convenient way to load a standalone panel.

Could I hook an external service into a panel using, for example, an iFrame? Panels are just entry-points to hook views into the larger dashboard navigational structure and enforce common attributes like RBAC. The views and corresponding templates can contain anything you would like, including iFrames.

What does this mean for visual design? The ability to add an arbitrary number of top-level navigational items (Dashboard objects) poses a new design challenge. Horizons lead designer has taken on the challenge of providing a reference design for Horizon which supports this possibility.

RELEASE NOTES

See <https://docs.openstack.org/releasenotes/horizon/>.

INFORMATION

5.1 Glossary

Horizon The OpenStack dashboard project. Also the name of the top-level Python object which handles registration for the app.

Dashboard A Python class representing a top-level navigation item (e.g. project) which provides a consistent API for Horizon-compatible applications.

Panel A Python class representing a sub-navigation item (e.g. instances) which contains all the necessary logic (views, forms, tests, etc.) for that interface.

Project Used in user-facing text in place of the term Tenant which is Keystones word.