openATTIC Documentation

Release 3.7.3-201904170926

SUSE Linux GmbH

Apr 17, 2019

Contents

openATTIC is an Open Source Ceph and storage management solution for Linux, with a strong focus on storage management in a datacenter environment.

The various resources of a Ceph cluster can be managed and monitored from a central web-based management interface. It is no longer necessary to be intimately familiar with the inner workings of the individual Ceph components. Any task can be carried out by either using openATTIC's intuitive web interface or via the REST API.

CHAPTER 1

Trademarks

“Apache HTTP Server”, “Apache”, and the Apache feather logo are trademarks of The Apache Software Foundation.

“Linux” is the registered trademark of Linus Torvalds in the U.S. and other countries.

“Red Hat Linux” and “CentOS” are trademarks of Red Hat, Inc. in the U.S. and other countries.

“openSUSE”, “SUSE” and the SUSE and openSUSE logo are trademarks of SUSE IP Development Limited or its subsidiaries or affiliates.

All other names and trademarks used herein are the property of their respective owners.

CHAPTER 2

Prerequisites

openATTIC can be installed on Linux only. It is designed to run on commodity hardware, so you are not in any way bound to a specific vendor or hardware model.

You need to make sure that your Linux distribution of choice supports the hardware you intend to use. Check the respective hardware compatibility lists or consult your hardware vendor for details.

2.1 Supported distributions

Installable packages of openATTIC 3.7 are currently available for the following Linux distributions:

- openSUSE Leap 42.3 (via the openSUSE Build Service) - see *Installation on openSUSE Leap* for details.

Note: openATTIC has been designed to be installed on a 64-bit Linux operating system. Installation on 32-bit systems is not supported.

2.2 Base Operating System Installation

The basic installation of the operating system (Linux distribution) depends on your requirements and preferences and is beyond the scope of this document.

Consult the distribution's installation documentation for details on how to perform the initial deployment.

We recommend performing a minimal installation that just installs the basic operating system (no GUI, no development tools or other software not suitable on a production system).

2.3 Post-installation Operating System Configuration

After performing the base installation of your Linux distribution of choice, the following configuration changes should be performed:

1. The system must be connected to a network and should be able to establish outgoing Internet connections, so additional software and regular OS updates can be installed.
2. Make sure the output of `hostname --fqdn` is something that makes sense, e.g. `srvopenattic01.yourdomain.com` instead of `localhost.localdomain`. If this doesn't fit, edit `/etc/hostname` and `/etc/hosts` to contain the correct names.
3. Install and configure an NTP daemon on every host, so the clocks on all these nodes are in sync.
4. HTTP access might be blocked by the default firewall configuration. Make sure to update the configuration in order to enable HTTP access to the openATTIC API/Web UI.

Consult your Linux distribution's documentation for further details on how to make these changes.

CHAPTER 3

Installation

This section guides you through the necessary installation process of the openATTIC software.

3.1 Quick Start Guide

Since version 3.x, some Ceph management features of openATTIC depend on a Ceph cluster deployed and managed via DeepSea. To be able to test the full feature set of openATTIC you'll need to deploy a Ceph cluster, services and openATTIC with this framework.

This "Quick Start Guide" will show you how to achieve this as fast as possible.

3.1.1 Requirements

- Three VMs or bare metal nodes - our recommendation would be to have at least five or six nodes.
- Ideally, all node host names should follow a fixed naming convention, e.g. `ceph-nn.yourdomain.com`
- Distribution: openSUSE-Leap42.3 (x86_64)
- Firewall must be disabled on all nodes

3.1.2 Setup a Ceph cluster with DeepSea

The way how to setup a Ceph cluster with DeepSea is well described in the upstream README and the DeepSea Wiki.

In this quick walkthrough we'll highlight the most important parts of the installation.

DeepSea uses salt to deploy, setup and manage the cluster. Therefore we have to define one of our nodes as the "master" (management) node.

Note: DeepSea currently only supports Salt 2016.11.04, while openSUSE Leap ships with a newer version (2017.7.2) by default. We therefore need to add a dedicated package repository that provides the older version and make sure that the package management system does not update it to a newer version by accident.

1. Log into the "master" node and run the following commands to add the DeepSea/openATTIC repository, install DeepSea and to start the `salt-master` service:

```
# zypper addrepo http://download.opensuse.org/repositories/filesystems:/ceph:/
↪luminous/openSUSE_Leap_42.3/filesystems:ceph:luminous.repo
# zypper addrepo http://download.opensuse.org/repositories/
↪systemsmanagement:saltstack:products/openSUSE_Leap_42.3/
↪systemsmanagement:saltstack:products.repo
# zypper addrepo http://download.opensuse.org/repositories/
↪filesystems:openATTIC:3.x/openSUSE_Leap_42.3/filesystems:openATTIC:3.x.repo
# zypper refresh
# zypper install salt-2016.11.04
# zypper install deepsea
# systemctl enable salt-master.service
# systemctl start salt-master.service
```

2. Next, install and configure the `salt-minion` service on all your nodes (**including the "master" node**) with the following commands:

```
# zypper addrepo http://download.opensuse.org/repositories/
↪systemsmanagement:saltstack:products/openSUSE_Leap_42.3/
↪systemsmanagement:saltstack:products.repo
# zypper refresh
# zypper install salt-minion-2016.11.04
# zypper al 'salt*'
```

   Configure all minions to connect to the master. If your Salt master is not reachable by the host name "salt", edit the file `/etc/salt/minion` or create a new file `/etc/salt/minion.d/master.conf` with the following content:

```
master: host_name_of_salt_master
```

   After you've changed the Salt minion configuration as mentioned above, start the Salt service on all Salt minions:

```
# systemctl enable salt-minion.service
# systemctl start salt-minion.service
```

3. Connect to your "master" node again:

   Check that the file `/srv/pillar/ceph/master_minion.sls` on the Salt master points to your Salt master and enable and start the Salt minion service on the master node:

```
# systemctl enable salt-minion.service
# systemctl start salt-minion.service
```

   Now accept all Salt keys on the Salt master:

```
# salt-key --accept-all
```

   Verify that the keys have been accepted:

```
# salt-key --list accepted
```

In order to avoid conflicts with other minions managed by the Salt master, DeepSea needs to know which Salt minions should be considered part of the Ceph cluster to be deployed.

This can be configured in file `/srv/pillar/ceph/deepsea_minions.sls`, by defining a naming pattern. By default, DeepSea targets all minions that have a grain `deepsea` applied to them.

This can be accomplished by running the following Salt command on all Minions that should be part of your Ceph cluster:

```
# salt -L <list of minions> grains.append deepsea default
```

Alternatively, you can change `deepsea_minions` in `deepsea_minions.sls` to any valid Salt target definition. See *man deepsea-minions* for details.

4. We can now start the Ceph cluster deployment from the "master" node:

   Stage 0 - During this stage all required updates are applied and your systems may be rebooted:

   ```
   # deepsea stage run ceph.stage.0
   ```

   Stage 1 - The discovery stage collects all nodes and their hardware configuration in your cluster:

   ```
   # deepsea stage run ceph.stage.1
   ```

   Now you've to create a `policy.cfg` within `/srv/pillar/ceph/proposals`. This file describes the layout of your cluster and how it should be deployed.

   You can find some examples upstream as well as in the documentation included in the `deepsea` RPM package at `/usr/share/doc/packages/deepsea/examples`.

   For this deployment we've chosen the rolebased policy. Please change this file according to your environment. See `man 5 policy.cfg` for details.

   Stage 2 - The configuration stage parses the `policy.cfg` file and merges the included files into their final form:

   ```
   # deepsea stage run ceph.stage.2
   ```

   Stage 3 - The actual deployment will be done:

   ```
   # deepsea stage run ceph.stage.3
   ```

   Stage 4 - This stage will deploy all of the defined services within the `policy.cfg`:

   ```
   # deepsea stage run ceph.stage.4
   ```

Congratulations, you're done! You can now reach the openATTIC Web-UI on "http://<your-master-node>.<yourdomain>

3.2 Installation on openSUSE Leap

openATTIC is available for installation on openSUSE Leap 42.3 from the openSUSE Build Service.

The software is delivered in the form of RPM packages via a dedicated zypper repository named `filesystems:openATTIC:3.x`.

Note: Before proceeding with the openATTIC installation, make sure that you have followed the steps outlined in *Base Operating System Installation*.

3.2.1 Zypper Repository Configuration

From a web browser, the installation of openATTIC on SLES or Leap can be performed via "1 Click Install" from the openSUSE download site.

From the command line, you can run the following command to enable the openATTIC package repository.

For openSUSE Leap 42.3 run the following as root:

```
# zypper addrepo http://download.opensuse.org/repositories/filesystems:/ceph:/
↪luminous/openSUSE_Leap_42.3/filesystems:ceph:luminous.repo
# zypper addrepo http://download.opensuse.org/repositories/filesystems:openATTIC:3.x/
↪openSUSE_Leap_42.3/filesystems:openATTIC:3.x.repo
# zypper refresh
```

Note: If you're interested in testing the latest state of the master branch (which is our development branch) please add the following repositories to your system:

```
# zypper addrepo http://download.opensuse.org/repositories/filesystems:/ceph:/
↪luminous/openSUSE_Leap_42.3/filesystems:ceph:luminous.repo
# zypper addrepo http://download.openattic.org/rpm/openattic-nightly-master-openSUSE_
↪Leap_42.3-x86_64/ openattic_repo
# rpm --import http://download.openattic.org/A7D3EAFA.txt
# zypper refresh
# zypper dist-upgrade
```

3.2.2 Package Installation

To install the openATTIC base packages on SUSE Linux, run the following command:

```
# zypper install openattic
```

Proceed with the installation by following the steps outlined in *Post-installation Configuration - (Manual deployment)*.

3.3 Post-installation Configuration - (Manual deployment)

If you want to use openATTIC without DeepSea or if you've deployed oA and DeepSea independently you have to configure a few things manually. Also be aware of that only a subset of the management functionality is available without DeepSea, e.g. the NFS Ganesha or iSCSI target configuration.

3.3.1 Enabling Basic Ceph Support in openATTIC

openATTIC depends on Ceph's librados Python bindings for performing many Ceph management and monitoring tasks, e.g. the management of Ceph Pools or RADOS block devices.

Note: Ceph support in openATTIC is currently developed against Ceph 12.1.0 aka "Luminous". Older Ceph versions may not work as expected. If your Linux distribution ships an older version of Ceph (as most currently do), please either use the upstream Ceph package repositories or find an alternative package repository for your distribution that provides a version of Ceph that meets the requirements. Note that this applies to both the version of the Ceph tools installed on the openATTIC node as well as the version running on your Ceph cluster.

To set up openATTIC with Ceph you first have to copy the Ceph administrator keyring and configuration from your Ceph admin node to your local openATTIC system.

From your Ceph admin node, you can perform this step by using `ceph-deploy` (assuming that you can perform SSH logins from the admin node into the openATTIC host):

```
# ceph-deploy admin openattic.yourdomain.com
```

On the openATTIC node, you should then have the following files:

```
/etc/ceph/ceph.client.admin.keyring
/etc/ceph/ceph.conf
```

Note: Please ensure that these files are actually readable by the openATTIC system user (`openattic` by default). This could be done by making them readable by the `openattic` user group:

```
# chgrp openattic /etc/ceph/ceph.conf /etc/ceph/ceph.client.admin.keyring
# chmod g+r /etc/ceph/ceph.conf /etc/ceph/ceph.client.admin.keyring
```

Alternatively, you can copy these files manually.

Note: openATTIC partially supports managing multiple Ceph clusters, provided they have different names and FSIDs. You can add another cluster by copying the cluster's admin keyring and configuration into `/etc/ceph` using a different cluster name, e.g. `development` instead of the default name `ceph`:

```
/etc/ceph/development.client.admin.keyring
/etc/ceph/development.conf
```

It is also possible to configure a Ceph cluster's configuration and keyring file in the settings file `/etc/sysconfig/openattic`.

`CEPH_CLUSTERS` is a string setting containing paths to `ceph.conf` files. Multiple clusters can be added by seperating them with a `;` sign like so:

```
CEPH_CLUSTERS="/etc/ceph/ceph.conf;/home/user/ceph/build/ceph.conf"
```

For each Ceph cluster, one can set the path to the keyring file by adding `CEPH_KEYRING_` appended by the uppercase cluster fsid as follows:

```
CEPH_KEYRING_123ABCDE_4567_ABCD_1234_567890ABCDEF="/home/user/ceph/build/keyring"
```

It is also possible to define a specific user name for each cluster, by adding `CEPH_KEYRING_USER_` appended by the uppercase cluster fsid, like so:

```
CEPH_KEYRING_USER_123ABCDE_4567_ABCD_1234_567890ABCDEF="client.openattic"
```

The last step is to recreate your openATTIC configuration:

```
# oaconfig install
```

3.3.2 DeepSea integration in openATTIC

DeepSea is a Ceph installation and management framework developed by SUSE which is based on the Salt Open automation and orchestration software. It highly automates the deployment, configuration and management of an entire Ceph cluster and all of its components.

Some openATTIC features like iSCSI target and Ceph object gateway (RGW) management depend on communicating with DeepSea via the Salt REST API.

To enable the REST API of DeepSea you would have to issue the following command on the Salt master node:

```
salt-call state.apply ceph.salt-api
```

By default, openATTIC assumes that Salt master hostname is `salt`, API port is `8000` and API username is `admin`. If you need to change any of this default values, you should configure it in either `/etc/default/openattic` for Debian-based distributions or in `/etc/sysconfig/openattic` for RedHat-based distributions as well as SUSE Linux.

Available settings are:

```
SALT_API_HOST="salt"
SALT_API_PORT="8000"
SALT_API_USERNAME="admin"
SALT_API_PASSWORD="admin"
```

Caution: Do not use spaces before or after the equal signs

3.3.3 Ceph Object Gateway management features

If you want to enable the Ceph Object Gateway management features, and you are using DeepSea, you just have to guarantee that the Salt REST API is correctly configured (see *DeepSea integration in openATTIC*). In case you are not using DeepSea, you have to configure the Rados Gateway manually by editing either `/etc/default/openattic` for Debian-based distributions or `/etc/sysconfig/openattic` for RedHat-based distributions as well as SUSE Linux.

This is an example for the manually configured Rados Gateway credentials:

```
RGW_API_HOST="ceph-1"
RGW_API_PORT=80
RGW_API_SCHEME="http"
RGW_API_ACCESS_KEY="VFEG733GBY0DJCIV6NK0"
RGW_API_SECRET_KEY="lJzPbZYZTv8FzmJS5eiiZPHxlT2LMGOMW8ZAeOAq"
```

Note: If your Rados Gateway admin resource isn't configured to use the default value `admin` (e.g. http://host:80/admin), you will need to also set the `RGW_API_ADMIN_RESOURCE` option appropriately.

You can obtain these credentials by issuing the `radosgw-admin` command like so:

```
radosgw-admin user info --uid=admin
```

3.3.4 openATTIC Base Configuration

After all the required packages have been installed, you need to perform the actual openATTIC configuration, by running `oaconfig`:

```
# oaconfig install
```

`oaconfig install` will start and enable a number of services, initialize the openATTIC database and scan the system for.

3.3.5 Changing the Default User Password

By default, `oaconfig` creates a local adminstrative user account `openattic`, with the same password.

As a security precaution, we strongly recommend to change this password immediately:

```
# oaconfig changepassword openattic
Changing password for user 'openattic'
Password: <enter password>
Password (again): <re-enter password>
Password changed successfully for user 'openattic'
```

Now, your openATTIC storage system can be managed via the user interface.

See *Accessing the Web UI* for instructions on how to access the web user interface.

If you don't want to manage your users locally, consult the chapter *Configuring Authentication and Single Sign-On* for alternative methods for authentication and authorization.

3.4 Accessing the Web UI

After openATTIC has been installed, it can be managed using the web-based user interface.

Open a web browser and navigate to http://openattic.yourdomain.com/openattic

The default login username and password is **openattic**.

See the *User Manual* for further information.

3.5 Configuring Authentication and Single Sign-On

When logging in, each user passes through two phases: **Authentication** and **Authorization**. The authentication phase employs mechanisms to ensure the users are who they say they are. The authorization phase then checks if that user is allowed access.

"Authentication is the mechanism of associating an incoming request with a set of identifying credentials, such as the user the request came from, or the token that it was signed with (Tom Christie)."

The openATTIC authentication is based on the Django REST framework authentication methods.

Currently openATTIC supports the following authentication methods of the Django REST framework:

- BasicAuthentication
- TokenAuthentication
- SessionAuthentication

Read more about the Django REST framework authentication methods here: Django REST framework - Authentication

3.5.1 Authentication

openATTIC supports three authentication providers:

1. Its internal database. If a user is known to the database and they entered their password correctly, authentication is passed.
2. Using Pluggable Authentication Modules to delegate authentication of username and password to the Linux operating system. If PAM accepts the credentials, a database user without any permissions is created and authentication is passed.
3. Using Kerberos tickets via mod_auth_kerb. Apache will verify the Kerberos ticket and tell openATTIC the username the ticket is valid for, if any. openATTIC will then create a database user without any permissions and pass authentication.

3.5.2 Authorization

Once users have been authenticated, the authorization phase makes sure that users are only granted access to the openATTIC GUI if they posess the necessary permissions.

Authorization is always checked against the openATTIC user database. In order to pass authorization, a user account must be marked active and a staff member.

Users created by the PAM and Kerberos authentication backends will automatically be marked active, but will not be staff members. Otherwise, *every* user in your domain would automatically gain access to openATTIC, which is usually not desired.

However, usually there is a distinct group of users which are designated openATTIC administrators and therefore should be allowed to access all openATTIC systems, without needing to be configured on every single one.

In order to achieve that, openATTIC allows the name of a system group to be configured. During the authorization phase, if a user is active but not a staff member, openATTIC will then check if the user is a member of the configured user group, and if so, make them a staff member automatically.

3.5.3 Configuring Domain Authentication and Single Sign-On

To configure authentication via a domain and to use Single Sign-On via Kerberos, a few steps are required.

1. Configuring openATTIC

   As part of the domain join process, the `oaconfig` script creates a file named `/etc/openattic/domain.ini` which contains all the relevant settings in Python's ConfigParser format.

   The `[domain]` section contains the kerberos `realm` and Windows `workgroup` name.

   The `[pam]` section allows you to enable password-based domain account authentication, and allows you to change the name of the PAM service to be queried using the `service` parameter. Note that by default, the PAM backend changes user names to upper case before passing them on to PAM – change the `is_kerberos` parameter to `no` if this is not desired.

Likewise, the `[kerberos]` section allows you to enable ticket-based domain account authentication.

In order to make use of the domain group membership check, add a section named `[authz]` and set the `group` parameter to the name of your group in lower case, like so:

```
[authz]
group = yourgroup
```

To verify the group name, you can try the following on the shell:

```
$ getent group yourgroup
yourgroup:x:30174:user1,user2,user3
```

2. Configuring Apache

   Please take a look at the openATTIC configuration file (`/etc/apache2/conf.d/openattic` on Debian/Ubuntu). At the bottom, this file contains a configuration section for Kerberos. Uncomment the section, and adapt the settings to your domain.

   In order to activate the new configuration, run:

```
apt-get install libapache2-mod-auth-kerb
a2enmod auth_kerb
a2enmod authnz_ldap
service apache2 restart
```

3. Logging in with Internet Explorer should work already. Mozilla Firefox requires you to configure the name of the domain in `about:config` under `network.negotiate-auth.trusted-uris`.

3.5.4 Troubleshooting Authentication Issues

Kerberos and LDAP are complex technologies, so it's likely that some errors occur.

Before proceeding, please double-check that NTP is installed and configured and make sure that `hostname --fqdn` returns a fully qualified domain name as outlined in the installation instructions.

Below is a list of common error messages and their usual meanings.

- `Client not found in Kerberos database while getting initial credentials`

  Possible reason: The KDC doesn't know the service (i.e., your domain join failed).

- `Generic preauthentication failure while getting initial credentials`

  Possible reason: `/etc/krb5.keytab` is outdated. Update it using the following commands:

```
net ads keytab flush
net ads keytab create
net ads keytab add HTTP
```

- `gss_acquire_cred() failed: Unspecified GSS failure. Minor code may provide more information (, )`

  Possible reason: Apache is not allowed to read `/etc/krb5.keytab`, or wrong `KrbServiceName` in `/etc/apache2/conf.d/openattic`.

3.6 Troubleshooting

In case of errors, the openATTIC WebUI will display popup notifications ("Toasties") that provide a brief explanation of the error that occured. For most errors, the openATTIC backend will raise a "500 - Internal Server Error" HTTP error, writing a more detailed Python traceback message to the openATTIC logfile.

3.6.1 openATTIC log files

The main log file to consult in case of errors is the file `/var/log/openattic/openattic.log`. This log file is written by the openATTIC Django application and should provide a detailed error message including the traceback that explains which Python module the error originated in.

Other logfiles worth checking in case of errors include the Apache HTTP server's error log. Its location depends on the Linux distribution that openATTIC has been installed on - please consult the distribution's documentation for details.

CHAPTER 4

User Manual

4.1 Introduction to the openATTIC Graphical User Interface

The openATTIC web interface is based on the AngularJS JavaScript framework combined with the Bootstrap HTML/CSS framework, to provide a modern look and feel.

This manual describes how to use the openATTIC web user interface (GUI) for performing storage management tasks like creating Ceph pools and RBD images, configuring iSCSI targets, NFS shares, as well as system management tasks like the configuration of users and API credentials, and the integrated monitoring system.

4.1.1 Administration Guide

Introducing the openATTIC Graphical User Interface

The openATTIC web interface is based on the AngularJS JavaScript framework combined with the Bootstrap HTML/CSS framework, to provide a modern look and feel.

4.1.2 How to Perform Common Tasks

Note: This part of the openATTIC documentation has not been created yet.

CHAPTER 5

Developer Documentation

openATTIC consists of a set of components built on different frameworks, which work together to provide a comprehensive Ceph storage management platform.

This document describes the architecture and components of openATTIC and provides instructions on how to set up a development environment and work on the various components included in the openATTIC code base.

If you would like to contribute to the openATTIC project, you need to prepare a development environment first.

Follow the outlined steps to *Create Your own openATTIC git Fork on BitBucket*.

Next, follow the instructions on *Setting up a Development System with Vagrant* or *Setting up a Development System*. Then code away, implementing whatever changes you want to make.

If you're looking for inspiration or some easy development tasks to get started with, we've created a list of low hanging fruit tasks that are limited in scope and should be fairly easy to tackle.

See *Contributing Code to openATTIC* for details on how to submit your changes to the upstream developers. Follow the *openATTIC Contributing Guidelines* to make sure your patches will be accepted.

If your changes modify documented behaviour or implement new functionality, the documentation should be updated as well. See *Working on the openATTIC documentation* for instructions on how to update the documentation.

5.1 Create Your own openATTIC git Fork on BitBucket

The openATTIC source code is managed using the git distributed version control system.

Git offers you a full-fledged version control, where you can commit and manage your source code locally and also exchange your modifications with other developers by pushing and pulling change sets across repositories.

If you're new to git, take a look at the git documentation web site. This will teach you the basics of how to get started.

The openATTIC source code repository is publicly hosted in a git Repository on BitBucket.

A "fork" is a remote git clone of a repository. Every openATTIC developer makes code modifications on a local copy (clone) of his fork before they are merged into the main repository via pull requests. See *Contributing Code to openATTIC* for instructions on how to get your code contributions included in the openATTIC main repository.

The quickest way to create a local clone of the main openATTIC git repository is to simply run the following command:

```
$ git clone https://bitbucket.org/openattic/openattic
```

However, if you would like to collaborate with the openATTIC developers, you should consider creating a user account on BitBucket and create a "Fork". We require real user names over pseudonyms when working with contributors.

Once you are logged into BitBucket, go to the openATTIC main repository and click **Fork** on the left side under **ACTIONS**. Now you should have your own openATTIC fork on BitBucket, which will be used to create a local copy (clone). You can find your repository's SSH or HTTPS URL in the top right corner of the repository overview page. Use this URL with `git clone` to create your local development clone.

Take a look at the BitBucket Documentation for further instructions on how to use BitBucket and how to work with repositories.

If you would like to contribute code to openATTIC, please make sure to read *Contributing Code to openATTIC* for instructions specfic to our project.

5.2 Setting up a Development System with Vagrant

Setting up a development system using Vagrant is by far the easiest way to start developing on openATTIC. However, we also provide instructions for setting up a classical development environment in *Setting up a Development System*.

5.2.1 WebUI preparation

To be able to run the WebUI with this setup you will have to change the default value of the API URL from "/openattic/api/" to "/api/".

More information on how to do this can be found at *WebUI local configuration*.

5.2.2 Vagrant Installation

Our Vagrant setup uses either a VirtualBox or a KVM/libvirt VM as base image. You will need to install at least one of them.

For example, KVM/libvirt can be installed on Ubuntu by running:

```
sudo apt-get install qemu-kvm
```

Please follow the official documentation for installing Vagrant.

After installing Vagrant, install the `vagrant-cachier` plugin for caching packages that are downloaded while setting up the development environment:

```
vagrant plugin install vagrant-cachier
```

The `vagrant-libvirt` plugin is required when using KVM on Linux:

```
vagrant plugin install vagrant-libvirt
```

If you're using VirtualBox on your host operating system, the `vagrant-vbguest` plugin enables guest support for some VirtualBox features like shared folders:

```
vagrant plugin install vagrant-vbguest
```

Note: If you experience an error while trying to install `vagrant-libvirt`, you might need to install the `libvirt-dev` and `gcc` package.

5.2.3 Network preparation

In order to enable internet access for your Vagrant box you need to enable IP forwarding and NAT on your host system:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
iptables -t nat -A POSTROUTING -s 192.168.10.0/24 \! -d 192.168.10.0/24 -j MASQUERADE
```

5.2.4 Starting the Virtual Machine

The openATTIC source code repository contains a Vagrant configuration file that performs the necessary steps to get you started. Follow the instructions outlined in *Create Your own openATTIC git Fork on BitBucket* on how to create your own fork and local git repository.

Navigate to the `vagrant` subdirectory of your local git clone and run this command to start your VM:

```
vagrant up
```

or, in case you are using KVM/libvirt, you need to specify the libvirt provider:

```
vagrant up --provider libvirt
```

This command will perform all steps to provide a running VM for developing openATTIC. After the completion of `vagrant up`, ssh into the VM:

```
vagrant ssh
```

In your VM, start openATTIC by running these commands. Notice, your local repository is available in the virtual machine at `~/openattic`:

```
. env/bin/activate
python openattic/backend/manage.py runserver 0.0.0.0:8000
```

Then, start your browser an open the URL as shown in the last lines of the log output of `vagrant up`.

Note: If you experience an error while trying to run `vagrant up --provider libvirt`, you might need to restart `libvirtd` service.

5.2.5 Choosing a different Linux distribution

Per default, the VM is based on OpenSUSE, but developing openATTIC based on an other Vagrant box is also possible by setting the environment variable `DISTRO`. These distributions are available:

- `DISTRO=jessie` (for Debian 8 "Jessie")
- `DISTRO=trusty` (for Ubuntu 14.04 LTS "Trusty Thar")
- `DISTRO=xenial` (for Ubuntu 16.04 LTS "Xenial Xerus")

- `DISTRO=malachite` (for openSUSE 42.1 "Malachite")

For example, to run a Xenial VM, run:

```
DISTRO=xenial vagrant up
```

or using KVM/libvirt:

```
DISTRO=xenial vagrant up --provider libvirt
```

Note: On a Windows host system using Windows Powershell, the environment variable can be defined as follows:

```
$env:DISTRO="xenial"
vagrant up
```

5.2.6 Debugging openATTIC with PyCharm Professional

With a running Vagrant VM, you can now debug the openATTIC Python backend using PyCharm.

First, configure a Vagrant Remote Interpreter pointing to `/home/vagrant/env/bin/python` on your VM. Then, add `/home/vagrant/openattic/backend` to the Python interpreter paths. You will be asked to activate a few PyCharm extensions, like a Django support or the remote interpreter tools.

Finally, add the openATTIC Django Server as a Pycharm *Django server* in the *Run Configurations* using your configured remote interpreter and host 0.0.0.0.

5.2.7 Debugging openATTIC with PyCharm Community

Please follow the instructions from the official documentation

5.2.8 Perform an openATTIC Base Configuration

It is not possible to execute `oaconfig install` in a Vagrant VM, you have to execute the following commands instead.

```
. env/bin/activate
cd openattic/backend
which systemctl && sudo systemctl reload dbus || sudo service dbus reload
sudo /home/vagrant/env/bin/python /home/vagrant/openattic/backend/manage.py␣
↪runsystemd &
python manage.py install --pre-install
python manage.py install --post-install
```

5.2.9 Troubleshooting

openATTIC systemd

If the openATTIC *systemd* is not running on your VM, you can start it by executing:

```
sudo env/bin/python openattic/backend/manage.py runsystemd
```

in your VM.

'vagrant destroy' fails due to a permission problem

To fix this error:

```
/home/<user>/.vagrant.d/gems/gems/fog-libvirt-0.0.3/lib/fog/libvirt/requests/compute/
↪volume_action.rb:6:in `delete': Call to virStorageVolDelete failed: Cannot delete '/
↪var/lib/libvirt/images/vagrant_default.img': Insufficient permissions␣
↪(Libvirt::Error)
```

Run this command or change the owner of `/var/lib/libvirt/images`:

```
chmod 777 /var/lib/libvirt/images
```

'vagrant destroy' fails due to wrong provider

You may also encounter the error that Vagrant tells you to *vagrant destroy*, but it doesn't seem to work. In that case you may be experiencing this issue.

A workaround for this is to specify your provider as default provider in the Vagrantfile like so:

```
ENV['VAGRANT_DEFAULT_PROVIDER'] = 'libvirt'
```

'vagrant up' fails on "Waiting for domain to get an IP address…"

It looks like this problem has something to do with the libvirt library and specific mainboards. We haven't found the cause of this problem, but using a different libvirt driver at least works around it.

Using `qemu` instead of `kvm` as driver does the trick. But kvm is and will be enabled by default, because qemu runs slower than kvm. You have to adapt the driver yourself in the `Vagrantfile` like so:

```
Vagrant.configure(2) do |config|
    config.vm.provider :libvirt do |lv|
        lv.driver = 'qemu'
    end
end
```

If you want to know more about this problem or even want to contribute to it, visit our bug tracker on issue OP-1455.

5.3 Setting up a Development System

In order to begin coding on openATTIC, you need to set up a development system, by performing the following steps. The instructions below assume a Debian "Jessie" or Ubuntu "Trusty" Linux environment. The package names and path names likely differ on other Linux distributions.

If you don't want to bother with performing the following steps manually, take a look at *Setting up a Development System with Vagrant*, which automates the process of setting up a development environment in a virtual machine to keep it separated from your local system.

5.3.1 Installing the Development Tools

openATTIC requires a bunch of tools and software to be installed and configured, which is handled automatically by the Debian packages. While you could of course configure these things manually, doing so would involve a lot of manual work which isn't really necessary. Set up the system just as described in *Installation*, but **do not yet execute** `oaconfig install`.

We recommend installing a nightly build for development systems, which is based on the latest commit in the `default` branch.

1. Set the installed packages on `hold` to prevent Apt from updating them:

```
# apt-mark hold 'openattic-.*'
```

2. Install Git:

```
# apt-get install git
```

3. Install Node.js and the Node Package Manager `npm`:

```
# apt-get install nodejs npm
# ln -s /usr/bin/nodejs /usr/bin/node
```

Note: We currently support Node.js v6.11.x

4. Go to the `/srv` directory, and create a local clone of your openATTIC fork there, using the current `master` branch as the basis:

```
# cd /srv
# git clone https://bitbucket.org/<Your user name>/openattic.git
# git checkout master
```

5. Customize the Apache configuration by editing `/etc/apache2/conf-available/openattic.conf`:
   - Replace the path `/usr/share/openattic` with `/srv/openattic/backend`
   - Add the following directive:

```
<Directory /srv/openattic>
    Require all granted
</Directory>
```

   - Adapt the `WSGIScriptAlias` paths to your local clone:

```
WSGIScriptAlias /openattic/serverstats /srv/openattic/backend/serverstats.wsgi
WSGIScriptAlias /openattic             /srv/openattic/backend/openattic.wsgi
```

6. In file `/etc/default/openattic`, change the `OADIR` variable to point to the local git clone:

```
OADIR="/srv/openattic/backend"
```

7. Now build the Web UI:

```
# cd /srv/openattic/webui
# npm run build
```

   If you intend to make changes to the web interface, it may be useful to run `npm run dev` as a background task, which watches the project directory for any changed files and triggers an automatic rebuild of the web interface code (including the `eslint` output), if required.

Note: Webpack will not include the `eslint` output for angular, because the provided configuration is there to help **you** develop the UI.

Note: If you modify any npm package or webpack configuration you will have to stop the background task and run it again. After doing so, the modification will be reflected.

8. Run `oaconfig install` and start openATTIC by running `oaconfig start`.

The openATTIC web interface should now be accessible from a local web browser via <http://localhost/openattic/>_ . The default username and password is "openattic".

You can now start coding by making modifications to the files in `/srv/openattic`. The openATTIC daemons, GUI and the `oaconfig` tool will automatically adapt to the new directory and use the code located therein.

See chapters *Contributing Code to openATTIC* and *openATTIC Contributing Guidelines* for further details on how to prepare your code contributions for upstream inclusion.

5.3.2 How to run the frontend in a local server

If you wish to run a frontend server on your local machine, while keeping the backend on a different machine, you can do it by simply creating an extra configuration file and replacing the build command shown in *Installing the Development Tools* with `npm start`.

We already provide you with a sample configuration file, that can be found at ''webui/webpack.config.json.sample'', so you just need to copy the contents of that file to a new one named ''webpack.config.json" and update the value of the target property so it reflects the hostname of your backend.

After creating the config file you can run the command and an instance of webpack dev server will start running on the background.

One big difference of this server is that it will keep all the compiled code in memory, allowing for a faster development process. It also comes with a live reload functionallity that will reload your browser automatically each time it detects a change in your code. And last, but not least, it will provide you with a proxy to your remote backend so you will not have CORS problems.

You can access openATTIC at http://localhost:8080/openattic/.

5.3.3 How to use eslint in your developer environment

Our `eslint` configuration gives you useful hints and tips how to provide `angularJS` code that is highly maintainable and forward-thinking. To see the hints and tips you have to install `eslint` and the plugin for `angularJS` globally:

```
# npm install -g eslint eslint-config-angular eslint-plugin-angular
```

To simply use it from the command line you can do the following:

```
% cd <clone>/webui
% eslint <path>
```

Or with vim without Syntastic:

```
:!eslint %
```

For all IDEs `eslint` can be installed as a plugin, if not already enabled.

5.3.4 How to get the authentication token for your own user

If you like to use the openATTIC TokenAuthentication (*Configuring Authentication and Single Sign-On*) in your own scripts in order to achieve automatization for example, you need to find out your own authentication token at first.

Here are two examples how you can get your authentication token via the REST API:

Curl:

```
curl --data "username=username&password=password"
http://<openattic-host>/openattic/api/api-token-auth/
```

Python requests:

```
import requests

requests.post("http://<openattic-host>/openattic/api/api-token-auth/",
data={"username": "<username>", "password": "<password>"})
```

Examples for additional scripts can be found here:

- Snapshot Python script with authtoken
- Cronjob Snapshot Script for openATTIC

5.3.5 WebUI local configuration

Our frontend application reads most of its default values from a global configuration file found in `webui/app/config.js`.

If you ever need to permanently change one of those values you can just open the file, change it and save the modification. This way everyone will have access to that same value.

But in situations where the changes you intent to apply only makes sense to your development environment, e.g. when using our vagrant setup (*Setting up a Development System with Vagrant*), you will have to take an extra step. You will have to create a local configuration file that will overwrite all the values of the preexisting file. To do that, simply create a new file, `webui/app/config.local.js`, with the content of `webui/app/config.local.js.sample`. Finally you have to *rebuild the frontend*. After that you, and only you, will see your custom configuration applied.

5.3.6 Backend local configuration

Same as to the frontend application, the backend part reads most of its default values from a global configuration file found in `backend/settings.py`.

If you want to customize those settings equal to the frontend application, then simply create the file `backend/settings_local.conf` and put the key/value pairs you want to override into this file.:

```
SALT_API_HOST='deepsea-1.xyz.net'
SALT_API_EAUTH='sharedsecret'
SALT_API_SHARED_SECRET='173a59b3-5abf-4a78-808a-253fe9ae3d94'

RGW_API_HOST="deepsea-1.xyz.net"
RGW_API_ADMIN_RESOURCE="admin"
RGW_API_USER_ID="admin"
RGW_API_ACCESS_KEY="PK258BAY1G1KEM7UH2Y3"
RGW_API_SECRET_KEY="rsOV874KLsaUBKLQzJ1oYdzyo7OXV4OAWoGDOdvE"
```

(continues on next page)

(continued from previous page)

```
GRAFANA_API_HOST="deepsea-1.xyz.net"
GRAFANA_API_PORT="3000"
GRAFANA_API_USERNAME="admin"
GRAFANA_API_PASSWORD="admin"
```

The local configuration will be applied when you restart the webserver and openATTIC systemd daemon.

5.4 Contributing Code to openATTIC

This is an introduction on how to contribute code or patches to the openATTIC project. If you intend to submit your code upstream, please also review and consider the guidelines outlined in chapter *openATTIC Contributing Guidelines*.

5.4.1 Keeping Your Local Repository in Sync

If you have followed the instructions in *Create Your own openATTIC git Fork on BitBucket*, you should already have a local openATTIC instance that is based on the current master branch.

You should update your repository configuration so that you will always pull from the upstream openATTIC repository and push to your openATTIC fork by default. This ensures that your fork is always up to date, by tracking the upstream development.

It is pretty common to name the upstream remote repository `upstream` and your personal fork `origin`.

If you've cloned your local repo from your personal fork already, it should already be named `origin` - you can verify this with the following command:

```
$ git remote -v
origin      git@bitbucket.org:<username>/openattic.git (fetch)
origin      git@bitbucket.org:<username>/openattic.git (push)
```

If the name differs, you can use `git remote rename <old> <new>`.

Now add the upstream repository by running the following command:

```
$ git remote add upstream ssh://git@bitbucket.org/openattic/openattic.git
```

Now you can keep your local repository in sync with the upstream repository by running `git fetch upstream`.

5.4.2 Using git+ssh behind a Proxy Server

If you want to use SSH behind a proxy you may use corkscrew. After the installation, append the following two lines to your `$HOME/.ssh/config` file:

```
Host bitbucket.org
    ProxyCommand corkscrew <proxy name or ip> <port number> %h %p
```

Now you can use SSH behind the proxy, because corkscrew now tunnels your SSH connections through the proxy to bitbucket.org.

5.4.3 Working With Branches

It is strongly recommended to separate changes required for a new feature or for fixing a bug in a separate git branch. Please refer to the git documentation for a detailed introduction into working with branches.

If you intend to submit a patch to the upstream openATTIC repository via a pull request, please make sure to follow the *openATTIC Contributing Guidelines*.

To create a new feature branch, update your repository, change to the `master` branch and create your new branch on top of it, in which you commit your feature changes:

```
$ git fetch upstream
$ git checkout master
$ git pull upstream master
$ git checkout -b <branchname>
< Your code changes >
$ git commit -a
```

To list your branches type the following (the current branch will be marked with an asterisk):

```
$ git branch --list
```

To just see the current branch you are working with type:

```
$ git rev-parse --abbrev-ref HEAD
```

After you are done with your changes, you can push them to your fork:

```
$ git push origin
```

5.4.4 Submitting Pull Requests

Now that your fork on BitBucket contains your changes in a separate branch, you can create a pull-request on Bitbucket to request an inclusion of the changes you have made into the `master` branch of the upstream openATTIC repository.

To do this, go to your fork on Bitbucket and click `Create pull request` in the left panel. On the next page, choose the branch with your changes as source and the openATTIC `master` branch as target.

Below the **Create pull request** button, first check out the **Diff** part if there are any merge conflicts. If you have some, you have go back into your branch and update it:

```
$ git fetch upstream
$ git rebase upstream/master
<resolve conflicts, mark them as resolved using "git add">
<test and review changes>
$ git rebase --continue
$ git push -f origin
```

After you have resolved the merge conflicts and pushed them into your fork, retry submitting the pull-request. If you already created a pull request, BitBucket will update it automatically.

After the pull-request was reviewed and accepted, your feature branch will be merged into the main repository. You may delete your feature branch on your local repository and BitBucket fork once it has been merged.

5.5 openATTIC Contributing Guidelines

Please see *Contributing Code to openATTIC* for details on the process of how to contribute code changes.

The following recommendations should be considered when working on the openATTIC code base.

While adhering to these guidelines may sound more work in the first place, following them has multiple benefits:

- It supports the collaboration with other developers and others involved in the product development life cycle (e.g. documentation, QA, release engineering).
- It makes the product development life cycle more reliable and reproducible.
- It makes it more transparent to the user what changes went into a build or release.

Some general recommendations for making changes and for documenting and tracking them:

- New Python code should adhere to the Style Guide for Python Code (PEP 8) with the exception of using 100 instead of 80 characters per line. Use the `flake8` tool to verify that your changes don't violate this style before committing your modifications. Also, please have a look at the Django coding style guides as a reference.
- Existing code can be refactored to adhere to PEP 8, if you feel like it. However, such style changes must be committed separately from other code modifications, to ease the reviewing of such pull requests.
- For JavaScript code, we use ESLint using the eslint-loader to perform automated syntax and style checks of the JavaScript code. ESLint is configured to adhere to the official AngularJS style guide. Please consult this style guide for more details on the coding style and conventions. The configuration files for these WebUI rules can be found in `webui/.eslintrc.json` and `webui/webpack.eslintrc.json`.
- Every bug fix or notable change made to a release branch must be accompanied by a JIRA issue. The issue ID must be mentioned in the summary of the commit message and pull request in the following format: `<summary> (OP-xxxx)`.
- Pull requests must be accompanied by a suggested `CHANGELOG` entry that documents the change.
- New features and other larger changes also require a related JIRA issue that provides detailed background information about the change.
- Code and the related changes to the documentation should be committed in the same change set, if possible. This way, both the code and documentation are changed at the same time.
- Write meaningful commit messages and pull request descriptions. Commit messages should include a detailed description of the change, including a reference to the related JIRA issue, if appropriate. "Fixed OP-xxx" is not a valid or useful commit message! For details on why this matters, see The Science (or Art?) of Commit Messages and How to Write a Git Commit Message
- When resolving a JIRA issue as fixed, include the resulting git revision ID or add a link to the ChangeSet or related pull request on BitBucket for reference. This makes it easier to review the code changes that resulted from a bug report or feature request.

5.5.1 Documenting Your Changes

Depending on what you have changed, your modifications should be clearly described and documented. Basically, you have two different audiences that have different expectations on how and where you document your changes:

- **Developers** that need to review and comment on your changes from an architectural and code quality point of view. They are primarily interested in the descriptions you put into the git commit messages and the description of your pull request, but will also review and comment on any other documentation you provide.

- **End users or administrators** that use openATTIC and need to be aware of potential changes in behaviour, new features or important bug and security fixes. They primarily consult the official documentation, release notes and the `CHANGELOG`.

Changes that should be user-visibly documented in the `CHANGELOG`, release notes or documentation include:

- Bug/security fixes on a release branch.
- User-visible changes or changes in behavior on a release branch. Make sure to review and update the documentation, if required.
- Major changes / new features. In addition to the `CHANGELOG`, these must be described in the documentation as well. See *Working on the openATTIC documentation* for details on how to update the openATTIC documentation.

Minor or "behind the scene" changes that have no user-visible impact or do not cause changes in behavior/functionality (e.g. improvements to build scripts, typo fixes, internal code refactoring) usually don't have to be documented in the `CHANGELOG` or the release notes.

Trust your judgment or ask other developers if you're unsure if something should be user-visibly documented or not.

Don't worry too much about the wording or formatting, the `CHANGELOG` and Release Notes will be reviewed and improved before a final release build anyway. It's much more important that we keep track of all notable changes without someone having to trawl JIRA or the commit messages prior to a release.

5.5.2 Signing Your Patch Contribution

To improve tracking of who did what, we use the "sign-off" procedure introduced by the Linux kernel. The sign-off is a simple line at the end of the explanation for the patch, which certifies that you wrote it or otherwise have the right to pass it on as an open-source patch.

The rules are pretty simple: if you can certify the following:

```
Developer Certificate of Origin
Version 1.1

Copyright (C) 2004, 2006 The Linux Foundation and its contributors.
660 York Street, Suite 102,
San Francisco, CA 94110 USA

Everyone is permitted to copy and distribute verbatim copies of this
license document, but changing it is not allowed.

Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the best
    of my knowledge, is covered under an appropriate open source
    license and I have the right under that license to submit that
    work with modifications, whether created in whole or in part
    by me, under the same open source license (unless I am
    permitted to submit under a different license), as indicated
    in the file; or
```

(continues on next page)

(continued from previous page)

```
(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including all
    personal information I submit with it, including my sign-off) is
    maintained indefinitely and may be redistributed consistent with
    this project or the open source license(s) involved.
```

then you just add the following line below your commit message and pull request saying:

```
Signed-off-by: Random J Developer <random@developer.example.org>
```

using your **real name and email address** (sorry, no pseudonyms or anonymous contributions).

Using git, this can be performed by adding the option `--signoff` to your commit command.

If you like, you can put extra tags at the end:

1. `Reported-by:` is used to credit someone who found the bug that the patch attempts to fix.
2. `Acked-by:` says that the person who is more familiar with the area the patch attempts to modify liked the patch.
3. `Reviewed-by:`, unlike the other tags, can only be offered by the reviewer and means that she is completely satisfied that the patch is ready for application. It is usually offered only after a detailed review.
4. `Tested-by:` is used to indicate that the person applied the patch and found it to have the desired effect.

You can also create your own tag or use one that's in common usage such as `Thanks-to:`, `Based-on-patch-by:`, or `Mentored-by:`.

5.5.3 Merging Pull Requests

The following steps should be performed when you're reviewing and processing a pull request on BitBucket:

1. A developer fixes a bug or implements a new feature in a dedicated feature branch. If required, he documents the changes in the documentation (for end-users) and the git commit messages (including the related Jira issue ID and a `Signed-off by:` line as outlined in chapter *Signing Your Patch Contribution*)
2. The developer creates a new Pull Request on BitBucket as described in chapter *Submitting Pull Requests*. The Pull Request description should include a detailed description of the change in a form suitable for performing a code review, summarizing the necessary changes. The description should also include a text suitable for inclusion into the `CHANGELOG`, describing the change from an end-user perspective.
3. After the pull request has been reviewed and approved, you perform the merge into the `master` branch using the BitBucket merge functionality.
4. Use a "merge" commit, not a "squash" commit for merging pull requests via BitBucket.

5.5.4 Backport commits

The following steps should be performed when you want to backport a fix to a stable release branch:

1. Ensure that the commits you want to backport exists on master (original pull request is merged to master)
2. Update your upstream repo: `git fetch upstream`

3. Create a branch from the stable release branch: `git checkout -b OP-<issue_id>-backport upstream/2.x`
4. Cherry pick the commits, using -x option: `git cherry-pick -x <sha-1>`
5. Adapt the CHANGELOG
6. Run all tests
7. Submit a pull request to the `2.x` stable release branch (title should be prefixed with "[2.x]")

5.5.5 Error Handling in Python

A few notes about error handling in openATTIC.

Good error handling is a key requirement in creating a good user experience and providing a good API. In our opinion, providing good errors to users is a blocker for releasing any non-beta releases of openATTIC.

Assume all user input is bad. As we are using Django, we can make use of Django's user input validation. For example, Django will validate model objects when deserializing from JSON and before saving them into the database. One way to achieve this is to add constraints to Django's model field definitions, like `unique=True` to catch duplicate inserts.

In general, input validation is the best place to catch errors and generate the best error messages. If feasible, generate errors as soon as possible.

Django REST framework has a default way of serializing errors. We should use this standard when creating own exceptions. For example, we should attach an error to a specific model field, if possible.

Our WebUI should show errors generated by the API to the user. Especially field-related errors in wizards and dialogs or show non-intrusive notifications.

Handling exceptions in Python should be an exception. In general, we should have few exception handlers in our project. Per default, propagate errors to the API, as it will take care of all exceptions anyway. In general, log the exception by adding `logger.exception()` with a description to the handler.

In Python it is easier to ask for forgiveness than permission (EAFP). This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the LBYL style common to many other languages such as C.

When calling system utilities or call external libraries, raise exceptions if appropriate to inform the user of problems. For example, if mounting a volume fails, raise an exception. From the Zen Of Python:

> Errors should never pass silently. Unless explicitly silenced.

Distinguish user errors from internal errors and programming errors. Using different exception types will ease the task for the API layer and for the user interface:

- Use `NotImplementedError` in abstract methods when they **require** derived classes to override the method. Have a look at the official documentation.
- Use `ValidationError` in an input validation step. For example. Django is using `ValidationErrors` when deserializing input.
- In case a `NotImplementedError` is not appropriate, because it is intentional not to implement a method and a `ValidationError` is not appropriate, because you're not validating any input, you can use a `NotSupportedError`. For example, if a file system does not support shrinking, you can use this exception here. They will be converted to a 400 status code in the API.
- Standard Python errors, like `SystemError`, `ValueError` or `KeyError` will end up as internal server errors in the API.
- Assert statements can help, if you want to protect functions from having bad side effects or return bad results.

In general, do not return error responses in the REST API. They will be returned by the openATTIC error handler `exception.custom_handler`. Instead, raise the appropriate exception.

In a Python function, in case of an error, try to raise an exception instead of returning `None`. Returning `None` in this case forces your caller to always check for `None` values.

5.5.6 Database migrations

Please follow the standard guidelines for adding database migration for Django 1.8+. Even, if you changed a *NoDB* model that don't write any data into the database. This ensures a consistent table schema for future Django upgrades.

5.6 openATTIC Architecture

The openATTIC core makes heavy use of the Django framework and is implemented as a Django project, consisting of several apps, one for each supported functionality or backend system.

Each app bundles a set of submodules. Models are used to represent the structure of the objects an app is supposed to be able to manage. The REST API (based on the Django REST Framework is used for interaction with the models. And lastly, the System API can be used in order to run other programs on the system in a controlled way.

When an application (e.g. the openATTIC Web UI, a command line tool or an external application), wants to perform an action, the following happens:

- The REST API receives a request in form of a function call, decides which host is responsible for answering the request, and forwards it to the core on that host.
- The *openATTIC Architecture* consists of two layers:
  - Django Models, the brains. They keep an eye on the whole system and decide what needs to be done.
  - File system layer: Decides which programs need to be called in order to implement the actions requested by the models, and calls those programs via the openATTIC `systemd` background process (not to be confused with the Linux operating system's systemd System and Service Manager).
- The openATTIC `systemd` executes commands on the system and delivers the results.

5.6.1 Models

Models are used to provide an abstraction for the real-world objects that your app has to cope with. They are responsible for database communication and for keeping an eye on the state of the whole system, being able to access any other piece of information necessary.

Please check out Django at a glance for more information.

5.6.2 openATTIC systemd background process

Some tasks require root privileges for being performed. In openATTIC, this is done via a separate background process `openattic-systemd`, written in Python. The Django web application uses `DBUS` as a bidirectional communication channel to this background service.

5.7 Working on the openATTIC documentation

The documentation for openATTIC consists of several documents, which are managed in the subdirectory `documentation` of the source code repository:

- *Installation* (subdirectory `install_guides`)
- *User Manual* (subdirectory `gui_docs`)
- *Developer Documentation* (subdirectory `developer_docs`)

The documentation is written in the reStructuredText markup language. We use the Sphinx documentation generator to build the documentation in HTML and PDF format, which is available online from http://docs.openattic.org/.

If you would like to work on the documentation, you first need to checkout a copy of the openATTIC source code repository as outlined in chapter *Setting up a Development System* (you can skip the steps of installing the development tools, if you intend to only work on the documentation).

5.7.1 Requirements

The documentation can be edited using your favorite text editor. Many editors provide built-in support for reStructuredText to ease the task of formatting.

To setup the Sphinx document generator, consult your Linux distribution's documentation. Most distributions ship with Sphinx included in the base distribution, so installing the package `python-sphinx` using your distribution's package management tool usually gets you up and running quickly, at least for creating the HTML documentation. Creating the PDF documentation is somewhat more involved, as it requires a LateX environment (e.g. the `texlive` distribution) and the `latexpdf` utility (usually included in the `pdftex` package).

Note: `python-sphinx` version should be >=1.3.6.

For previewing the HTML documentation, you need a local web browser, e.g. Mozilla Firefox or Google Chrome/Chromium. The PDF document can be previewed using any PDF viewer, e.g. Evince or Adobe Acrobat Reader®.

5.7.2 Documentation Guidelines

In order to maintain a common document structure and formatting, please keep the following recommendation in mind when working on the documentation:

- Use 2 spaces for indendation, not tabs.
- Wrap long lines at 78 characters, if possible.
- Overlong command line examples should be wrapped in a way that still supports cutting and pasting them into a command line, e.g. by using a backslash ("\") for breaking up shell commands.

5.7.3 Building the documentation

After you have made your changes to the respective reST text files, you can perform a build of the HTML documentation by running the following command from within the `documentation` directory:

```
$ make html
```

Take a close look at the build output for any errors or warnings that might occur. The resulting HTML files can be found in the directory `_build/html`. To open the start page of the documentation, open the index page in a web browser, e.g. as follows:

```
$ firefox _build/html/index.html
```

To build the PDF document, run the following command:

```
$ make latexpdf
```

This build process will take some more time, again make sure to check for any warnings or errors that might occur. The resulting PDF can be found in the directory `_build/latex`. Open it in a PDF viewer, e.g. as follows:

```
$ evince _build/latex/openATTIC.pdf
```

If you are satisfied with the outcome, commit and push your changes.

If you would like to contribute your changes, please make sure to read *Contributing Code to openATTIC* for instructions.

5.8 Customizing the openATTIC WebUI

The openATTIC user interface is a web application based on the AngularJS 1 JavaScript MVW framework the Bootstrap framework. Using Cascading Style Sheets (CSS), it is possible to customize the look to some degree, e.g. by replacing the logo or adapting the color scheme.

These modifications can be performed by adding your changes to the `vendor.css` CSS file. It is located in the directory `webui/app/styles/vendor.css` in the Mercurial source code repository and the source tar archive, or in `/usr/share/openattic-gui/styles/vendor.css` in the RPM and DEB packages.

Take a look at the file webui/app/styles/openattic-theme.css to get an overview about the existing class names and their attributes.

Alternatively, you can use Mozilla Firebug or similar web development tools to obtain this information.

5.8.1 Changing the favicon

An alternative favicon image (PNG format, 32x32 pixels) must be copied to the `images/` directory (`webui/app/images` in the source, `/usr/share/openattic-gui/images` for the installation packages).

If you choose a different name for the image file, the file name in `index.html` must be adapted. As of the time of writing, this information is located in lines 27-29:

```
<!-- favicons -->
<link rel="shortcut icon" href="images/openattic_favicon_32x32.png" type="image/x-icon
↪">
<link rel="icon" href="images/openattic_favicon_32x32.png" type="image/x-icon">
```

5.8.2 Changing the logo

It is possible to customize the application logo displayed in the top left corner of the application window. The format should be PNG, the size should not exceed 250x25 pixel (to ensure it is displayed properly on mobile devices).

The logo file should be copied into the `images/` directory. If you choose a different name than the default, update the file name in file `components/navigation/templates/navigation.html` (currently located in line 5).

If you comment out line 5 and enable line 6, the graphical logo can be replaced with regular text:

```
<a class="navbar-brand" href="#"><img src="images/openattic_brand_bright.png" alt=
↪"openATTIC"></a>
<!--<a class="navbar-brand" href="#">openATTIC storage management framework</a>-->
```

In addition to that, the logo on the login screen should be replaced to match your desired logo. It should be in PNG format and should not exceed 256x256 px. This can be achieved by changing the image file name in file `components/auth/templates/login.html`, line 4:

```
<img src="images/openattic_favicon_256x256.png" alt="openATTIC" class="pull-right">
```

5.9 Background-Tasks

5.9.1 What is a background task?

A background task is a task of a process that takes time, while you would normally be waiting on the frontend, for it to already finish. Instead of waiting in the UI you will be redirected as soon as the task is created. The task will finish in the background fulfilling it's duty.

5.9.2 Where can I find the running background tasks?

You can watch your current background tasks by one click on "Background-Tasks" at the top right, to the left of your login name. A dialogue will open and list the current pending tasks if any.

5.9.3 Are there different types of tasks?

There are three categories of tasks - pending, failed and finished tasks. You can choose them through the tabs, named after the category, in the background task dialog. The pending tab is always opened when you open up the dialog. * Pending task are queued and waiting to run or running. * Failed tasks are tasks that failed due to there process or because a user deleted a pending task. * Finished tasks are task that have successfully processed what they should do.

5.9.4 How can I test it?

You can. The openATTIC API needed to implement the creation of test task which are doing nothing than counting numbers, in order to test the functionality with tasks of a predefined running time.

Open up your Javascript console of your browser after your have logged in and paste the following function in it:

```
var createTestTask = function (time) {
  var xhr = new XMLHttpRequest();
  var url = "/openattic/api/taskqueue/test_task";
  xhr.open("post", url, true);
  xhr.setRequestHeader("Content-Type", "application/json");
  var data = JSON.stringify({times: time});
  xhr.send(data);
}
```

Now you can create a test task like this:

```
createTestTask(<time in seconds>)
```

The time a task runs is not really the value you pass, the value determines the calculation rounds the task will do. One round estimates to around one second at low numbers.

5.9.5 Can I delete them?

Yes, even pending tasks, but you will be warned if you want that, because the running process will not be stopped immediately instead all follow up executions will be canceled and the action taken will not be revoked. But if you do so, the task will be handled as a failed task. Failed and finished task can be deleted with out the risk of data corruption.

5.9.6 Do I have to wait for the task to finish?

No, you see the changes more rapidly. For example if you create a ceph pool the pool will be created and be available in the pool listing, while it's still building up, so you should't modify it right away.

5.9.7 Which processes create a background task?

At the moment the following operations are running as background tasks:

- Setting the number of PGs in a ceph pool.
- Getting RBD performance data of a cluster.

5.10 Task Queue Module

This module is intended to track long running tasks in the background.

Please read this mailing first, before continuing.

Tasks are functions which are scheduled from our Django backend and are executed in our openATTIC-systemD daemon.

5.10.1 Hello World

Let's build our first task:

```
from taskqueue.models import task

@task
def hello(name):
    print 'hello {}!'.format(name)
```

To schedule this task, restart our openattic-systemd and run

```
hello.delay("world")
```

Now, our daemon should print *hello world!* into the console. Keep in mind that our daemon needs to import our hello task, thus if you see `AttributeError: 'module' object has no attribute 'hello'` in your log file, make the task importable by putting it into a python file.

Note: There is no guarantee that a task will not be executed multiple times. Although, multiple executions of one task will not happen concurrently.

Note: A task is expected to be a top-level function of a module. Class methods or inner functions may not work as expected.

5.10.2 Recursion

Let's wait for a long running task:

```
@task
def wait(times):
    if times:
        return wait(times - 1)
    else:
        print 'finished'
```

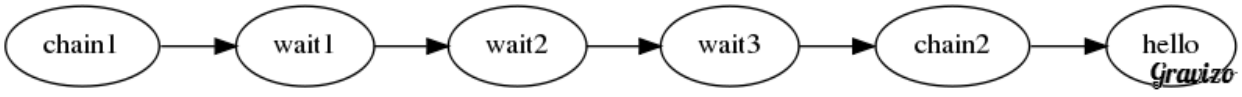
This tasks schedules itself, similar to a recursive function. As we're using a trampoline, this will not grow the stack. This also means that you cannot synchronously wait for a task to finish, thus if you want to run multiple iterations, you can only use end-recursion. This is similar to continuation passing style, where the next task is continuation of our current iteration.

We're JSON serializing the parameters into the database, so you are limited to basic data types, like int, str, float, list, dict, tuple. As an extension to JSON, you can also use a task as a parameter. For example, you can use this to chain tasks into one task:

```
from taskqueue.models import deserialize_task
@task
def chain(values):
    tasks = map(deserialize_task, values) # need to manually deserialize the tasks
    first, *rest = tasks
    res = first.run_once()
    if isinstance(res, Task):
        return chain([res] + rest, total_count)
    elif not rest:
        return res
    else:
        # Ignoring res here.
        return chain(rest, total_count)

@task
def wait_and_print(times, name):
    return chain([wait(times), hello(name)])
```

When calling `wait_and_print.delay(3, 'Sebastian')`, Task Queue will run a state machine like this:



A ready-to-use chain task is available by importing `taskqueue.models.chain`.

5.10.3 Progress Percentage

A task also has an attached progress percentage value. In case you have a long running task where a progress may be useful to a user, you can provide a `percent` argument to `@task` like so:

```
@task(percent=lambda total, remaining: 100 * remaining / total)
def wait(total, remaining):
    if remaining:
        return wait(total, remaining - 1)
    else:
        print 'finished'
```

The percent parameter will be called with the same parameters as your task.

Note: The function is expected not to have any side effects, as it may be called multiple times or never.

Note: Always use keyword arguments for the task decorator, as positional arguments may not work as expected.

5.10.4 Revision Upgrades

Warning: Keep in mind, that we're serializing the tasks into the database.

If you modify code, keep these restrictions in mind:

1. A task, including all parameters are serialized into the database,
2. thus be prepared to be called with a **outdated and ancient** function arguments.
3. Deleting the Python source of a task will eventually throw an exception.
4. Rule of thumb, **only** add optional parameters at the end to existing tasks.
5. If something goes wrong, a task may be aborted between function calls.
6. Try not to run important modifying commands later on.
7. Validate your function parameters.
8. As long as you only modify the implementation, everything is fine.

5.10.5 Referencing a newly created TaskQueue object

The `taskqueue` module provides a Python mixin for referencing a `TaskQueue` object in a HTTP header from another REST API. First, add the `TaskQueueMixin` to your ViewSet class like so:

```
from taskqueue.restapi import TaskQueueLocationMixin

class MyModelViewSet(TaskQueueLocationMixin, ViewSet):
   pass
```

Second, create a `_task_queue` attribute of your saved model instance in your `save` method:

```
class MyModel(Model):
   def save(self, *args, **kwargs):
       # ...
       self._task_queue = app.tasks.my_task.delay()
```

Now, if a `MyModel` instance is saved, a `Taskqueue-Location` HTTP header pointing to the `TaskQueue` object is added to your response.

5.10.6 Integration with openATTIC-systemD

Tasks are executed in our openATTIC-systemD process, thus they are independent of Apache worker processes and can run without being interrupted.

On the other hand, openATTIC-systemD runs in glibs MainLoop. In order to integrate with it, we need to create a GObject with a periodic timer event. Here is the code to start the timer of `TaskQueueManager`:

```
try:
    import taskqueue.manager
    taskqueue_manager = taskqueue.manager.TaskQueueManager()
except ImportError:
    pass
```

5.10.7 Background

As the architecture is similar to other task queues, I've tried to make a task definition similar to the API of Celery, which also uses a task decorator.

Task Queue is also similar to a Haskell package called Workflow, quote:

> Transparent support for interruptable computations. A workflow can be seen as a persistent thread that executes a monadic computation. Therefore, it can be used in very time consuming computations such are CPU intensive calculations or procedures that are most of the time waiting for the action of a process or an user, that are prone to communication failures, timeouts or shutdowns. It also can be used if you like to restart your program at the point where the user left it last time. The computation can be restarted at the interrupted point thanks to its logged state in permanent storage.

Task Queue stores the computation context between each trampoline call. Workflow uses some kind of continuation monad to hide interruptions between restarts. Task queue use a similar idea, although in a greatly reduced variant, as the syntax of Python is not as expressive as other Languages, like C#.

You can even think of a task as being a green thread, because you can schedule multiple tasks at once. Each of them will be executed interleaved.

5.11 openATTIC Web UI Tests - Unit Tests

openATTIC Web UI has both **Unit** and **E2E** tests. This section is related to **Unit tests**, if you are looking for **E2E tests** documentation please refer to *openATTIC Web UI Tests - E2E Test Suite*.

Unit tests are implemented in `*.spec.js` files, which must be in the same directory as the file being tested:

```
+-- oa-sample
|   '-- oa-sample.service.js
|   '-- oa-sample.service.specjs
```

5.11.1 Writing Unit Tests

Use "angular.mock.module" instead of "module" seen in most examples. The reason behind that is the usage of "webpack". It has a assigned module variable which is a constant this way it can't be overwritten by the import of "angular-mocks". To be consistent call other mock methods via "angular.mock".

5.11.2 Run Unit Tests

To run openATTIC Web UI unit tests, you should guarantee that your dependencies are updated (`$ npm install`), and perform the following command:

```
$ npm test
```

If you are a developer and want to run the tests on each change automatically use this command:

```
$ npm run devTest
```

5.11.3 Coverage Report

After running the unit tests, an HTML coverage report will be generated into `coverage` directory. To open this coverage report you should use a web browser, e.g.:

```
$ firefox coverage/index.html
```

5.11.4 References

For more information on the tools that are used by the Web UI unit tests, see:

- Karma
- Jarmine

5.12 openATTIC Web UI Tests - E2E Test Suite

This section describes how our **E2E test** environment is set up, as well as how you can run our existing **E2E tests** on your openATTIC system and how to write your own tests. If you are looking for Web UI **Unit tests** documentation please refer to *openATTIC Web UI Tests - Unit Tests*.

By continuously writing E2E-tests, we want to make sure that our graphical user interface is stable and acts the way it is supposed to be - that offered functionalities really do what we expect them to do.

We want to deliver a well-tested application, so that you - as users and community members - do not get bothered with a buggy user interface. Instead, you should be able to get started with the real deal - MANAGING storage with openATTIC.

5.12.1 About Protractor

Protractor is a end-to-end test framework, which is especially made for AngularJS applications and is based on WebDriverJS. Protractor will run tests against the application in a real browser and interacts with it in the same way a user would.

For more information, please refer to the protractor documentation.

5.12.2 System Requirements

Testing VM:

- Based on our experience,the system on which you want to run the tests needs at least 4GB RAM to prevent it from being laggy or very slow!

5.12.3 Install Protractor

Note: Protractor and most of its dependencies will be installed locally when you execute `npm install` on `webui/` .

- *(optional)* `npm install -g protractor@5.1.2`
- `apt-get install openjdk-8-jre-headless oracle-java8-installer`
- Choose/Install your preferred browser (Protractor supports the two latest major versions of Chrome, Firefox, Safari, and IE)
- Please adapt the `protractor.conf.js` file which can be found in `/openattic/webui/` to your system setup - see instructions below

5.12.4 Protractor Configuration

Before starting the tests, you need to configure and adapt some files.

Use Multiple Browsers

When using Chrome and Firefox for the tests, you could append the following to your `protractor.conf.js` so the test will run in both browsers:

```
exports.config.multiCapabilities = [
    {'browserName': 'chrome'},
    {'browserName': 'firefox'}
];
```

To prevent running both browsers at the same time you can add:

```
exports.config.maxSessions = 1;
```

Set up `configs.js`

Create a `configs.js` file in folder `e2e` and add the URL to you openATTIC system as well as login data - see below:

```
(function() {
  module.exports = {
    urls: {
      base: '<proto://addr:port>',
      ui: '/openattic/#/',
      api: '/openattic/api/'
    },
```

(continues on next page)

(continued from previous page)

```
    //leave this if you want to use openATTIC's default user for login
    username: 'openattic',
    password: 'openattic',
  };
}());
```

If you are using a Vagrant box, then you have to set urls.ui to `/#/` and urls.api to `/api/`.

In order to run our graphical user interface tests, please make sure that your openATTIC system at least has:

- One LVM volume group
- One ZFS zpool

and add them to `e2e/configs.js`.

Note: For more information have a look at `e2e/configs.js.sample`.

It is important that the first element in this config file is your volume group.

If you do not have a ZFS zpool configured and you do not want to create one, you can of course skip those tests by removing the suite from `protractor.conf.js` or putting them in to the comment section.

5.12.5 Start webdriver manager Environment

Go to `webui/` and type the following command:

`$ webdriver-manager start` *or:* `$ npm run webdriver`

5.12.6 Make Protractor Execute the Tests

Use a separate tab/window, go to `webui/` and type:

```
$ npm run protractor (-- --suite <suiteName>)
```

Important: Without a given suite protractor will execute all tests (and this will probably take a while!)

5.12.7 Starting Only a Specific Test Suite

If you only want to test a specific action, you can run i.e. `$ npm run protractor -- --suite general`.

Available test cases can be looked up in `protractor.conf.js`, i.e.:

```
suites: {
  //suite name       : '/path/to/e2e-test/file.e2e.js'
  general            : '../e2e/base/general/**/general.e2e.js',
}
```

Note: When running protractor.conf and the browser window directly closes and you can see something like "user-data error" (i.e. when using Chrome) in your console just create a dir (i.e. in your home directory) and run `google-chrome --user-data-dir=/path/to/created/dir`

5.12.8 How to Cancel the Tests

When running the tests and you want to cancel them, rather press `CTRL+C` on the commandline (in same window in which you've started `protractor`) than closing the browser. Just closing the browser window causes every single test to fail because protractor now tries to execute the tests and can not find the browser window anymore.

5.12.9 E2E-Test Directory and File Structure

In directory `e2e/` the following directories can be found:

```
+-- base
|   '-- auth
|   '-- datatable
|   '-- general
|   '-- pagination
|   '-- pools
|   '-- settings
|   '-- taskqueue
|   '-- users
+-- ceph
|   `-- iscsi
|   `-- nfs
|   `-- pools
|   `-- rbds
|   `-- rgw
```

Most of the directories contain a `*form.e2e.js` in which we only test things like validation, the number of input fields, the title of the form etc. Actions like `add`, `clone` etc. are always in a separate file. This makes it better to get an overview and prevents the files from getting very huge and confusing.

5.12.10 Writing Your Own Tests

Please include `common.js` in every `.e2e.js` file by adding `var helpers = require('../common.js');`. In some cases (depending on how you've structured your tests) you may need to adapt the path.

By including it as `var helpers` you can now make use of helper functions from `common.js`, i.e. the `setLocation` function, you just have to add `helpers.` to the function: `helpers.setLocation( location [, dialogIsShown ] )`.

The following helper functions are implemented:

- `setLocation`
- `leaveForm`
- `checkForUnsavedChanges`
- `get_list_element`
- `get_list_element_cells`

- delete_selection
- search_for
- search_for_element
- login
- hasClass

When using more than one helper function in one file, please make sure that you use the right order of creating and deleting functions in `beforeAll` and `afterAll`.

If you need to navigate to a specific menu entry (every time!) where your tests should take place, you can make use of:

```
beforeEach(function(){

  //always navigates to menu entry "ISCSI" before executing the actions
  //defined in 'it('', function(){});'
  element(by.css('.tc_menuitem_ceph_iscsi')).click();

});
```

5.12.11 Style Guide - General e2e.js File Structure / Architecture

You should follow the official Protractor style guide.

Here are a few extra recommendations:

- `describe` should contain a general description of what is going to be tested (functionality) in this spec file i.e. the site, menu entry (and its content), panel, wizard etc. example: "should test the user panel and its functionalities"
- `it` should describe, what exactly is going to be tested in this specific it-case i.e. (based on the described example above): "should test validation of form field "Name""
- Elements which are going to be used more than once should be defined in a variable on top of the file (under described)
- If something has to be done frequently and across multiple spec files one can define those steps in a function defined in above mentioned `common.js` and use this function in specific spec files i.e. if you always/often need a user before you can start the actual testing you can define a function `create_user` which contains the steps of creating a user and use the `create_user` function in the tests where it's required. Therefore you just have to require the `common.js` file in the spec file and call the `create_user` function in the *beforeAll* function. This procedure is a good way to prevent duplicated code. (for examples see `common.js` -> `login` function)
- Make use of the `beforeAll`/`afterAll` functions if possible. Those functions allow you to do some steps (which are only required once) before/after anything else in the spec file is going to be executed. For example, if you need to login first before testing anything, you can put this step in a `beforeAll` function. Also, using a `beforeAll` instead of a `beforeEach` saves a lot of time when executing tests. Furthermore, it's not always necessary to repeat a specific step before each `it` section. The `afterAll` function is a good way to "clean up" things which are no longer needed after the test. If you already have a function (i.e. `create_user`) which creates something, you probably want to delete it after the tests have been executed. So it makes sense having another function, which deletes the object (in this case a `delete_user`-function) that can simply be called in `afterAll`. In addition we decided to put an `afterAll` at the end of each test file which contains a `console.log("<protractor suite name> -> <filename>.e2e.js")`. By doing so it is possible to track which test in which file is currently executed when running all tests.

- In a bunch of openATTIC HTML files (see `openattic/webui/app/templates`) you'll find css classes which are especially set for tests (those test classes are recognizable by the `tc_`-term which stands for "test class"). This is very useful when protractor finds more than one element of something (i.e. "Add"-button) and you can specify the element by adding or just using this tc_class of the element you're looking for to the locator. This makes the needed element unique (i.e.: `element(by.css('oadatatable .tc_add_btn')).click();`)
- Tests should be readable and understandable for someone who is not familiar in detail with tests in order to make it easy to see what exactly the test does and to make it simple writing tests for contributors. Also, for someone who does not know what the software is capable of, having a look at the tests should help understanding the behavior of the application
- Always navigate to the page which should be tested before each test to make sure that the page is in a "clean state". This can be done by putting the navigation part in a `beforeEach` function - which ensures that `it` sections do not depend on each other as well.
- Make sure that written tests do work in the latest version of Chrome and Firefox
- The name of folders/files should tell what the test is about (i.e. folder "user" contains "user_add.e2e.js")

5.12.12 Tips on how to write tests that also support Firefox

Let protractor only click on clickable elements, like `a`, `button` or `input`.

If you want to select an option element use the following command to make sure that the item is selected (issue #480):

```
browser.actions().sendKeys( protractor.Key.ENTER ).perform();
```

5.12.13 Debugging your tests

To set a breakpoint use `browser.pause()` in your code.

After your test pauses, go to the terminal window where you started the test.

You can type `c` and hit enter to continue to the next command or you can type `repl` to enter the interactive mode, here you can type commands that will be executed in the test browser.

To continue the test execution press `ctrl + c`.

5.13 openATTIC REST API Tests - Gatling Test Suite

Gatling is the openATTIC integration test suite. It's based on the Python unit testing framework and contains a bunch of tests to be run against a live openATTIC installation.

Gatling sends requests to openATTIC's REST API and checks if the responses are correct. For example Gatling tries to create a volume via openATTIC's REST API and checks if it's getable and deletable afterwards. If an error should be included in a response, Gatling checks if it is really included.

Afterwards Gatling checks the openATTIC internal command log if errors occurred during execution time.

5.13.1 Quick start

To run Gatling, you need to have an openATTIC host set up that has all the features installed (have a look at *Installation*) which you intend to test. Then create a configuration file in the `conf` subdirectory (i.e., `conf/<yourhost>.conf`) as explained in section *Configuration* and run Gatling with the following command:

```
$ python gatling.py -t yourhost
```

Gatling will adapt to your environment, automatically skipping tests that cannot be run on your installation, and run all tests that can run in your environment.

5.13.2 Dependencies

Gatling depends on the `testtools` and `xmlrunner` packages. To install them, type:

```
# apt-get install python-testtools python-xmlrunner
```

5.13.3 Configuration

In order to get Gatling work well with your openATTIC environment it needs some information about the system configuration. These information are organized in configuration files. For an example configuration, have a look at the `gatling.conf` file included in the distribution. These settings are suitable in most of the cases. However all the settings which do not match your openATTIC installation need to be overridden in a separate configuration file.

The first section of the configuration file is the `options` section. It holds general settings about how to connect to your openATTIC host. Enter the complete name of your openATTIC host at the `host_name` setting. If the username or the password of the admin account does not match the default values you will need to configure them too.

If you don't want to test a specific feature - for example you don't have the openATTIC DRBD module installed, so you don't want to test it by Gatling, you just need to disable the related tests by:

```
[drbd]
enabled = no
```

For a complete overview of the configuration section and options please have a look at the `gatling.conf` file.

All available tests of Gatling are **enabled** by default.

5.13.4 CI integration

Gatling supports integration in Continuous Integration systems like Jenkins. To use this functionality, pass the `--xml` option to Gatling, which will instruct Gatling to write JUnit-compatible test reports in XML format into an output directory of your choice. You can then instruct your build server to generate reports from these documents.

5.13.5 Advanced options

Gatling uses the following command line structure:

```
python gatling.py [options] -- [unittest.TestProgram options]
```

Gatling supports all the options that the standard Python *unittest* module supports when run using `python -m unittest`. However, in order to separate Gatling's own options from those passed on to `unittest`, you need to add `--` in front of *unittest* options, like such:

```
python gatling.py --xml -- --failfast
```

If the Gatling command line does not include `--`, Gatling will by default activate test discovery and verbosity. If you want to run Gatling without *any unittest* arguments, pass `--` at the end of the command line.

5.13.6 Source code layout

Test cases are laid out in a way that ensures maximum flexibility while keeping the amount of duplicate code to an absolute minimum.

The openATTIC API is flexible enough to allow lots of different combinations of storage technologies, and testing all those different combinations is somewhat of a challenge. To mediate this without having to duplicate test cases, Gatling uses a system of combining test scenarios and tests to test cases that are then added to the test suite and run by Gatling.

Scenarios

A scenario defines the environment in which tests are supposed to be run, for instance:

- Test sharing an XFS-formatted LV using NFS.
- Test sharing a ZFS subvolume using NFS.
- Test sharing an Ext4-formatted ZVol using NFS.
- Test sharing an unformatted ZVol using iSCSI.

Scenario classes use the `setUpClass` and `tearDownClass` classmethods to prepare the openATTIC system for the tests that are to be run, creating any necessary Volume pools or other objects to be used by the tests, and provide a `_get_pool` method that returns the Volume pool on which the tests are to be run.

When implementing a Scenario, make sure that its `setUpClass` method

- raises `SkipTest` if the test scenario cannot be run on this system due to missing openATTIC modules or other errors,
- properly calls its superclass so that inheriting multiple scenarios works the way it should, like so:

```
class LvTestScenario(GatlingTestCase):
    @classmethod
    def setUpClass(cls):
        super(LvTestScenario, cls).setUpClass()
```

Generally lay out your class in a way that it can be combined with as many other scenarios as possible.

Tests

Tests are collected in classes that inherit from `object` and only define `test_<something>` methods. These classes **must not** inherit `unittest.TestCase` so they can be imported into other modules without causing the tests to be discovered and run twice.

Although this class does not inherit `unittest.TestCase` directly, their code can make use of everything the `TestCase` class provides. This is because the `*Tests` classes are abstract classes meant to be combined with a test scenario in order to be run, which then makes it a full `TestCase` subclass.

TestCases

In order to create a TestCase subclass that can be discovered and run, create a third class that inherits both the Scenario and the Tests, like so:

```
class LioTestCase(LvTestScenario, LunTestScenario, LvLioTests):
    pass
```

Be sure to inherit all the test scenarios you need for your test functions to run, so that the environment is set up and torn down correctly and tests can be skipped if necessary modules are missing.

5.14 Python Unit Tests

This chapter describes the use of unit tests for Python. Python unit tests are much simpler than our E2E or Gatling tests, as they only test the Python code without having external dependencies, thus they are purely focused on the Python code.

5.14.1 Executing Tests

Keep in mind that in order to execute the tests, openATTIC needs an execution environment with a database connection.

Please navigate to the backend root folder and execute:

```
$ ./manage.py test -t . -v 2
```

Note: Running the tests might not work due to permission issues for the *openattic* user to create an empty database. The following commands will fix that:

```
oaconfig dbshell
ALTER USER openattic CREATEDB;
```

You can exit the *dbshell* by typing "\q" or pressing "Ctrl+D".

To generate a coverage analysis, you will have to install coverage:

```
$ pip install coverage
```

Then, run the analysis:

```
$ coverage run --source='.' manage.py test -t . -v 2
```

and generate a report with:

```
$ coverage report
```

It's also possible to generate detailed HTML reports to identify uncovered code lines by running:

```
$ coverage html
```

It will create the directory `htmlcov` containing `index.html` as a starting point.

5.14.2 Starting Only a Specific Test Suite

In order to run just a subset of the tests, append the backend module name like so:

```
$ manage.py test <module-name> -t . -v 2
```

where `<module-name>` needs to be replaced with a module name, like `ceph`.

5.14.3 Writing Tests

Every backend module contains a `tests.py` containing unit tests. Please add your new unit test to this file.

We also make use of doctest. If your test would be an enhancement to the documentation and your testing subject is easy to test, i.e. without mocking and side effects, consider to add this test to the docstring and add or extend the "doctest boiler plate":

```
import doctest
def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(<module containing the doctest>))
    return tests
```

5.14.4 Style Guide

Here are a few recommendations:

- Focus purely on Python. Hesitate to test anything else.
- Aim for keeping the testing coverage in percent for every Pull Request.
- Keep the tests fast: Don't add any sleep statements or long running tests, except if really necessary. Speed is important, as it keeps the code-test loop short.
- Use mock to reduce external dependencies, but try to reduce the amount of mock statements.
- Make sure your new tests work with all supported Django versions.
- Don't execute network requests, as they introduce dependencies to other services.

CHAPTER 6

Indices and Tables

- genindex
- modindex
- search