openATTIC Documentation

Release 2.0.22-201802211529

SUSE Linux GmbH

Feb 21, 2018

Contents

The times when storage was considered a server-based resource and every system needed to have its own hard drives are long gone. In modern data centers central storage systems have become ubiquitous for obvious reasons. Centrally managed storage increases flexibility and reduces the cost for unused storage reserves. With the introduction of a cluster or virtualization solution shared storage becomes a necessity.

This mission-critical part of IT used to be dominated by proprietary offerings. Even though mature open source projects may now meet practically every requirement of a modern storage system, managing and using these tools is often quite complex and is mostly done decentrally.

openATTIC is a full-fledged central storage management system. Hardware resources can be managed, logical storage areas can be shared and distributed and data can be stored more efficiently and less expensively than ever before – and you can control everything from a central management interface. It is no longer necessary to be intimately familiar with the inner workings of the individual storage tools. Any task can be carried out by either using openATTIC's intuitive web interface or via the REST API.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License version 2 as published by the Free Software Foundation.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

CHAPTER 1

Trademarks

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

“DRBD®”, the DRBD logo, “LINBIT®”, and the LINBIT logo are trademarks or registered trademarks of LINBIT in Austria, the United States and other countries.

“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.

“VMware” is a trademark or registered trademark of VMware, Inc. in the United States and/or other jurisdictions.

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

CHAPTER 2

Installation and Getting Started

This section guides you through the necessary operating system preparation and the installation and configuration process of the openATTIC software.

The installation can be broken down into the following steps:

1. *Base Operating System Installation*
2. *Basic Storage Configuration*
3. **openATTIC installation:**
    - *Installation on Debian Linux*
    - *Installation on Ubuntu Linux*
    - *Installation on Red Hat Enterprise Linux (and Derivatives)*
    - *Installation on SUSE Linux Enterprise Server and openSUSE Leap*
4. *Post-installation Configuration*
5. *Download Preconfigured Virtual Machine*
6. *Getting started*

2.1 System requirements

openATTIC can be installed on the most popular Linux distributions. 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.

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

- Debian Linux 8 “Jessie”
- Red Hat Enterprise Linux 7 (RHEL) and derivatives (CentOS 7, Oracle Linux 7 or Scientific Linux 7)

- openSUSE Leap 42.1, SUSE Linux Enterprise Server 12 (SLES12) (via the openSUSE Build Service)
- Ubuntu 14.04 LTS "Trusty Thar"
- Ubuntu 16.04 LTS "Xenial Xerus"

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

For testing openATTIC, you should dedicate and prepare at least one additional entire hard disk to it. See *Basic Storage Configuration* for details.

When setting up a production server, there are a couple of things you should be aware of when designing the system. See *Storage Recommendations* and *Hardware Recommendations* for further details.

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).

2.2.1 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 and other things might be blocked by the default firewall configuration. For example on EL7 system, execute the following commands:

```
# firewall-cmd --permanent --zone=<your zone ie internal|public> --add-
↪service=http
# firewall-cmd --permanent --zone=<your zone ie internal|public> --add-
↪service=samba
# firewall-cmd --permanent --zone=<your zone ie internal|public> --add-service=nfs
# firewall-cmd --permanent --zone=<your zone ie internal|public> --add-
↪service=iscsi-target
# firewall-cmd --reload
```

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

2.3 Basic Storage Configuration

Note: If you only want to use openATTIC for managing and monitoring a Ceph cluster, you can skip the storage configuration. No additional local disks or storage pools are required for performing this functionality. After performing the basic openATTIC software installation, follow the steps outlined in *Enabling Ceph Support in openATTIC* to make your Ceph cluster known to openATTIC.

At a minimum, openATTIC should have one dedicated storage pool (e.g. an LVM volume group or a ZFS zpool) for creating storage volumes. In the following chapters, we'll explain how to create an LVM volume group or, alternatively, a ZFS zpool.

Configuring storage for openATTIC in a reliable and performant way depends on a number of factors. See *Storage Recommendations* and *Hardware Recommendations* for some recommendations.

Note: Currently, openATTIC requires that a storage pool (LVM or ZFS) has already been configured/prepared on the command line. This step has to be performed until the required functionality has been implemented in openATTIC itself. See OP-108 and OP-109 for details.

2.3.1 Create an LVM Volume Group for openATTIC

One way of managing storage with openATTIC is using the Linux Logical Volume Manager "LVM". The required command line tools are usually installed on a Linux distribution by default. To learn more about LVM, consult your distribution's documentation or the LVM HOWTO.

In the following steps, we'll create a logical volume group for openATTIC to use. The volume group name and device names may differ on your system. In this example, we'll use the second and third hard disk of the system, and create a volume group named `vgdata`:

```
# vgcreate vgdata /dev/sdb /dev/sdc
```

Consult the *lvm(8)* manual page and the LVM HOWTO for further information on how to create volume groups and the supported modes of redundancy and performance.

2.3.2 Tag OS Volume Groups / Logical Volumes

If you have installed your operating system's file systems on logical volumes (which is the default for many distributions), you can tag these volumes or the entire volume group with a `sys` tag to prevent openATTIC from registering them for usage when running `oaconfig install`.

For example, on CentOS, you could run the following command to mark the entire `centos` volume group as reserved for the operating system:

```
# vgchange --addtag sys centos
```

This will prevent the entire `centos` volume group from being registered for management as a storage pool by openATTIC.

Alternatively, you can tag selected logical volumes within the volume group:

```
# lvchange --addtag sys centos/root
# lvchange --addtag sys centos/swap
```

The `centos` volume group will be visible as a storage pool in openATTIC and you can create and manage volumes in there, except for the `root` and `swap` volumes.

2.3.3 Create a ZFS zpool

As an alternative to using LVM, openATTIC also supports using the OpenZFS file system for managing the underlying storage.

In order to use the ZFS file system, you need to install the required filesystem driver modules for ZFS on Linux separately. Installation packages for various Linux distributions are available from the ZFS on Linux web site. See the "Getting Started" pages on that site for details on the distribution-specific installation steps.

Once ZFS on Linux has been installed and configured, a simple zpool for testing purposes on a single disk could be created using the following command:

```
# zpool create -m /media/tank tank /dev/sdb
```

In a production environment, you should create a zpool across multiple disks (e.g. in a RAID-1 configuration), to achieve the desired level of performance and redundancy. See *Storage Recommendations* and the ZFS documentation for recommendations.

Note: The ZFS zpool needs to be mounted below `/media/<poolname>` in order for openATTIC to manage it.

To enable ZFS support in openATTIC, you also need to install the additional `openattic-module-zfs` package and run `oaconfig install` to register the newly created zpool.

2.4 Installation on Debian Linux

We provide installable DEB packages of openATTIC via apt package repositories from http://apt.openattic.org .

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

2.4.1 Importing the openATTIC Keyfile

The openATTIC packages are signed using a cryptographic key. You can import the public GPG key from the download site using the following command:

```
# wget http://apt.openattic.org/A7D3EAFA.txt -q -O - | apt-key add -
```

The GPG key's fingerprint can be verified with `apt-key finger` and should look as follows:

```
pub   2048R/A7D3EAFA 2012-03-05
      Key fingerprint = 9A91 1EDD 45A2 4B25 9C39  E7D4 1D5C D44D A7D3 EAFA
uid                  Business Critical Computing <is-bcc@it-novum.com>
sub   2048R/A99076EE 2012-03-05
```

2.4.2 Enabling the openATTIC Apt Package Repository

In order to add the openATTIC apt repository, create a file named `/etc/apt/sources.list.d/openattic.list`, and put the following lines into it:

```
deb     http://apt.openattic.org/ jessie  main
deb-src http://apt.openattic.org/ jessie  main
```

Enabling Nightly Builds

In addition to the official releases, we also provide nightly builds, built off the current "default" branch that will eventually become the next official openATTIC release.

To enable the nightly repo, the file `/etc/apt/sources.list.d/openattic.list` needs to be expanded to look as follows:

```
deb     http://apt.openattic.org/ jessie   main
deb-src http://apt.openattic.org/ jessie   main
deb     http://apt.openattic.org/ nightly  main
deb-src http://apt.openattic.org/ nightly  main
```

2.4.3 Package Installation

After enabling the apt repository, run the following commands to install the openATTIC DEB packages:

```
# apt-get update
# apt-get install openattic
```

Note: Installation of the `openattic-gui` package will replace the distribution's default `index.html` page in the Apache web server's document root with a redirect page to the openATTIC web interface.

Proceed with the installation by following the steps outlined in *Post-installation Configuration*.

2.5 Installation on Ubuntu Linux

We provide installable DEB packages of openATTIC via apt package repositories from http://apt.openattic.org .

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

2.5.1 Importing the openATTIC Keyfile

The openATTIC packages are signed using a cryptographic key. You can import the public GPG key from the download site using the following command:

```
$ sudo apt-key adv --fetch-keys http://apt.openattic.org/A7D3EAFA.txt
```

The GPG key's fingerprint can be verified with `apt-key finger` and should look as follows:

```
pub   2048R/A7D3EAFA 2012-03-05
      Key fingerprint = 9A91 1EDD 45A2 4B25 9C39  E7D4 1D5C D44D A7D3 EAFA
uid                  Business Critical Computing <is-bcc@it-novum.com>
sub   2048R/A99076EE 2012-03-05
```

2.5.2 Enabling the openATTIC Apt Package Repository

In order to add the openATTIC apt repository, run the following command for adding the openATTIC repository.

Note: The command `lsb_release -cs` will return the correct code name of your distribution.

- `trusty` (for Ubuntu 14.04 LTS "Trusty Thar")
- `xenial` (for Ubuntu 16.04 LTS "Xenial Xerus")

```
$ sudo add-apt-repository "deb http://apt.openattic.org/ $(lsb_release -cs) main"
```

Enabling Nightly Builds

In addition to the official releases, we also provide nightly builds, built off the current "default" branch that will eventually become the next official openATTIC release.

To enable the nightly repo, run the following command:

```
$ sudo add-apt-repository "deb http://apt.openattic.org/ $(lsb_release -cs) main"
$ sudo add-apt-repository "deb http://apt.openattic.org/ nightly main"
```

2.5.3 Package Installation

After enabling the apt repository, run the following commands to install the openATTIC DEB packages.

Note: For **Ubuntu 14.04 LTS** it is necessary to install some extra package in order to get the `lio-utils` package working which is used by `openattic-module-lio` (installed by the base openATTIC package). You may need to restart the target service as well:

```
$ sudo apt-get install linux-image-extra-$(uname -r)
$ sudo service target restart
```

Now, install openATTIC:

```
$ sudo apt-get update
$ sudo apt-get install openattic
```

Note: Installation of the `openattic-gui` package will replace the distribution's default `index.html` page in the Apache web server's document root with a redirect page to the openATTIC web interface.

Note: For **Ubuntu 16.04 LTS** some required LVM services may not run after the installation of openATTIC. Please enable them by executing:

```
$ sudo systemctl enable lvm2-lvmetad.socket
$ sudo systemctl start lvm2-lvmetad.socket
```

Proceed with the installation by following the steps outlined in *Post-installation Configuration*.

2.6 Installation on Red Hat Enterprise Linux (and Derivatives)

Starting with version 2.0, openATTIC is also available for RPM-based Linux distributions, namely Red Hat Enterprise Linux 7 (RHEL) and derivatives (e.g. CentOS 7, Oracle Linux 7 or Scientific Linux 7). For the sake of simplicy, we refer to these distributions as Enterprise Linux 7 (EL7).

The software is delivered in the form of RPM packages via dedicated yum repositories.

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

2.6.1 Preliminary Preparations on RHEL 7

Note: This step is not required on CentOS and other RHEL derivatives.

To install on RHEL 7, be sure to disable the "EUS" and "RT" yum repos, and enable the "Optional" repo:

```
# subscription-manager repos --disable=rhel-7-server-eus-rpms
# subscription-manager repos --disable=rhel-7-server-rt-rpms
# subscription-manager repos --enable=rhel-7-server-optional-rpms
```

Afterwards, just continue with the following installation steps.

2.6.2 Disable SELinux

For the time being, SELinux needs to be disabled or put into "permissive" mode when running openATTIC (see OP-543 for details).

On the command line, run the following command:

```
# setenforce 0
```

To disable SELinux at system bootup, edit `/etc/sysconfig/selinux` and change the configuration option `SELINUX` to `permissive`.

Use the command `getenforce` to ensure that SELinux has been disabled correctly.

2.6.3 Yum Repository Configuration

openATTIC requires some additional packages that are not part of the official EL7 distribution, but can be obtained from the Extra Packages for Enterprise Linux (EPEL) yum repository.

To enable the EPEL repository, you need to run the following command:

```
# yum install epel-release
```

Download and install the `openattic-release` RPM package located in the following directory:

```
# yum install http://repo.openattic.org/rpm/openattic-2.x-el7-x86_64/openattic-
↪release.rpm
```

This will automatically enable package installation from the openATTIC Release repository.

To enable the nightly RPM builds, edit `/etc/yum.repos.d/openattic.repo` and enable the `[openattic-nightly]` yum repository by setting `enabled` to `1`.

2.6.4 Package Installation

To install the openATTIC base packages on EL7, run the following command:

```
# yum install openattic
```

The openATTIC web GUI is not installed automatically when using `yum install openattic`, as it might not be required on each node of an openATTIC cluster.

It can be installed with the following command:

```
# yum install openattic-gui
```

Note: Installation of the `openattic-gui` package will install an `index.html` page in the Apache web server's document root that will redirect requests to the openATTIC web interface.

2.6.5 Configure PNP4Nagios on EL7

openATTIC uses Nagios and the PNP4Nagios addon for analyzing performance data and generating graphs to display the performance and utilization of disks and volumes.

By default, PNP4Nagios is configured by openATTIC automatically to run in bulk mode with npcdmod to process performance data.

Unfortunately Nagios in the EPEL repository has been updated to version 4.0.x some time ago, which does no longer support this mode. See OP-820 for more details.

Instead, PNP4Nagios on EL7 needs to be configured manually for using bulk mode with NPCD, by following the steps outlined below.

Append the following to `/etc/nagios/nagios.cfg`:

```
#
# Bulk / NPCD mode
#

# *** the template definition differs from the one in the original nagios.cfg
#
service_perfdata_file=/var/log/pnp4nagios/service-perfdata
service_perfdata_file_template=DATATYPE::SERVICEPERFDATA\tTIMET::$TIMET$\tHOSTNAME::
↪$HOSTNAME$\tSERVICEDESC::$SERVICEDESC$\tSERVICEPERFDATA::$SERVICEPERFDATA
↪$\tSERVICECHECKCOMMAND::$SERVICECHECKCOMMAND$\tHOSTSTATE::$HOSTSTATE
↪$\tHOSTSTATETYPE::$HOSTSTATETYPE$\tSERVICESTATE::$SERVICESTATE$\tSERVICESTATETYPE::
↪$SERVICESTATETYPE$
```

(continues on next page)

(continued from previous page)

```
service_perfdata_file_mode=a
service_perfdata_file_processing_interval=15
service_perfdata_file_processing_command=process-service-perfdata-file

# *** the template definition differs from the one in the original nagios.cfg
#
host_perfdata_file=/var/log/pnp4nagios/host-perfdata
host_perfdata_file_template=DATATYPE::HOSTPERFDATA\tTIMET::$TIMET$\tHOSTNAME::
↪$HOSTNAME$\tHOSTPERFDATA::$HOSTPERFDATA$\tHOSTCHECKCOMMAND::$HOSTCHECKCOMMAND
↪$\tHOSTSTATE::$HOSTSTATE$\tHOSTSTATETYPE::$HOSTSTATETYPE$
host_perfdata_file_mode=a
host_perfdata_file_processing_interval=15
host_perfdata_file_processing_command=process-host-perfdata-file
```

Add the following to `/etc/nagios/objects/commands.cfg`:

```
#
# definitions for PNP processing commands
# Bulk with NPCD mode
#
define command {
 command_name process-service-perfdata-file
 command_line /bin/mv /var/log/pnp4nagios/service-perfdata /var/spool/pnp4nagios/
↪service-perfdata.$TIMET$
}

define command {
 command_name process-host-perfdata-file
 command_line /bin/mv /var/log/pnp4nagios/host-perfdata /var/spool/pnp4nagios/host-
↪perfdata.$TIMET$
}
```

To make sure that all changes have been applied correctly, please run `nagios --verify-config /etc/nagios/nagios.cfg` afterwards, to verify the configuration files for errors.

Nagios will be restarted during the openATTIC installation and should then generate the necessary RRD and XML files in `/var/lib/pnp4nagios/<hostname>`.

Proceed with the installation by following the steps outlined in *Post-installation Configuration*.

2.7 Installation on SUSE Linux Enterprise Server and openSUSE Leap

openATTIC is available for installation on SUSE Linux Enterprise Server 12 (SLES12) and openSUSE Leap 42 from the openSUSE Build Service.

The software is delivered in the form of RPM packages via dedicated yum repositories named `filesystems:openATTIC`.

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

2.7.1 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 42.1 run the following as root:

```
# zypper addrepo http://download.opensuse.org/repositories/filesystems:openATTIC/
↪openSUSE_Leap_42.1/filesystems:openATTIC.repo
# zypper refresh
```

For SLE 12 SP1 run the following as root:

```
# zypper addrepo http://download.opensuse.org/repositories/filesystems:openATTIC/SLE_
↪12_SP1/filesystems:openATTIC.repo
# zypper refresh
```

For SLE 12 run the following as root:

```
# zypper addrepo http://download.opensuse.org/repositories/filesystems:openATTIC/SLE_
↪12/filesystems:openATTIC.repo
# zypper refresh
```

2.7.2 Package Installation

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

```
# zypper install openattic
```

The openATTIC web GUI is not installed automatically when using `zypper install openattic`, as it might not be required on each node of an openATTIC cluster.

It can be installed with the following command:

```
# zypper install openattic-gui
```

Proceed with the installation by following the steps outlined in *Post-installation Configuration*.

2.8 Post-installation Configuration

2.8.1 openATTIC Base Configuration

After all the required packages have been installed and a storage pool has been created, 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 pools and volumes to include.

2.8.2 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 *Getting started* 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.

2.8.3 Installing additional openATTIC Modules

After installing openATTIC, you can install additional modules (`openattic-module-<module-name>`), by using your operating system's native package manager, i.e.:

```
# apt-get install openattic-module-drbd # Debian/Ubuntu
# yum install openattic-module-btrfs # RHEL/CentOS
```

Note: Don't forget to run `oaconfig install` after installing new modules.

2.9 Enabling Ceph Support in openATTIC

Note: Ceph support in openATTIC is currently developed against Ceph 10.2 aka "Jewel". 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 user (`openattic`) and the Nagios/Icinga user account (usually `nagios` or `icinga`) that runs the related Nagios checks. In a default installation, these users are added to the group `openattic`, so it should be sufficient to make sure these files are either world-readable or owned and readable by this 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 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
```

The next step is to install the openATTIC Ceph module `openattic-module-ceph` on your system:

```
# apt-get install openattic-module-ceph
- or -
# yum install openattic-module-ceph
```

The packages should automatically install any additionally required packages. The last step is to recreate your openATTIC configuration:

```
# oaconfig install
```

2.10 Download Preconfigured Virtual Machine

openATTIC can be downloaded as preconfigured virtual machines from http://download.openattic.org/vms/.

At the moment you can download openATTIC installed on Debian and Ubuntu. More coming soon...

You can choose between two different image files - `qcow2` for KVM and `vdi` for VirtualBox.

The default login username for the VMs is **root** and the password is **openattic**.

The default login username for the openATTIC WebUI is **openattic** and the password is **openattic**.

Note: Please run `oaconfig install` the first time you've started the virtual machine. Otherwise the WebUI will be empty.

How to use those images:

1. KVM - Libvirt XML example file - you can use this example and import it to libvirt if you want to. Revise disk path and bridge name according to your needs. Otherwise create a new vm and select the image file as your disk.

   ```
   <domain type='kvm'>
     <name>oa-vm-deb</name>
     <uuid>e8afb480-d464-ed28-c200-000000000002</uuid>
   ```

```
  <memory unit='KiB'>2097152</memory>
  <currentMemory unit='KiB'>2097152</currentMemory>
  <vcpu placement='static'>2</vcpu>
  <resource>
    <partition>/machine</partition>
  </resource>
  <os>
    <type arch='x86_64' machine='pc-1.0'>hvm</type>
    <boot dev='hd'/>
  </os>
  <features>
    <acpi/>
    <apic/>
    <pae/>
  </features>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>restart</on_crash>
  <devices>
    <emulator>/usr/bin/kvm</emulator>
    <disk type='file' device='disk'>
      <driver name='qemu' type='qcow2' cache='none' io='native'/>
      <source file='/var/lib/libvirt/images/oa-vm-debian-2.0.22.qcow2'/>
      <target dev='sda' bus='scsi'/>
    </disk>
    <disk type='file' device='cdrom'>
      <driver name='qemu' type='raw'/>
      <target dev='sdb' bus='scsi'/>
      <readonly/>
    </disk>
    <controller type='ide' index='0'>
    </controller>
    <controller type='usb' index='0'>
    </controller>
    <controller type='pci' index='0' model='pci-root'/>
    <controller type='scsi' index='0'>
    </controller>
    <interface type='bridge'>
      <mac address='52:54:00:00:00:02'/>
      <source bridge='virbr0'/>
      <model type='virtio'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x03'␣
↪function='0x0'/>
    </interface>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
    <input type='mouse' bus='ps2'/>
    <input type='keyboard' bus='ps2'/>
    <graphics type='vnc' port='5900' autoport='no' listen='0.0.0.0'␣
```

```
↪keymap='de'>
      <listen type='address' address='0.0.0.0'/>
    </graphics>
    <video>
      <model type='cirrus' vram='16384' heads='1'/>
    </video>
    <memballoon model='virtio'>
    </memballoon>
  </devices>
</domain>
```

2. VirtualBox - Create a new virtual machine and select "already existing disk" or create a virtual machine without a disk and add it afterwards.

2.11 Getting started

2.11.1 Accessing the Web UI

openATTIC can be managed using a web-based user interface, if the package `openattic-gui` has been installed and the Apache http Server has been restarted afterwards.

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

The default login username is **openattic**. Use the password you defined during the *Post-installation Configuration*.

See the *User Manual* for further information.

2.12 Installing an openATTIC Multi-node System

openATTIC can be installed in a multi-node setup, in which any node can be used to manage the whole system and commands are distributed to the appropriate node automatically. This is implemented by using a shared configuration database, connecting all openATTIC nodes to the same PostgreSQL database.

This is usually the database of the first node that you have installed and configured, but can be a database running on a dedicated node, too.

In order to use DRBD®, you will need to set up a multi-node setup consisting of **two** hosts.

Note: Note that multi-node support currently applies to the "traditional" storage management functionality of openATTIC only. For managing Ceph, you need to connect to the web interface of the openATTIC node configured to connect to the Ceph cluster directly.

2.12.1 Step 1 - Install Two openATTIC Hosts

In the following example the first host is called **openattic01.yourdomain.com** (IP address: 192.168.1.101) and the second **openattic02.yourdomain.com** (IP address: 192.168.1.102). Both hosts should be able to connect to each other using their host names, so make sure that DNS is configured correctly (or you have configured `/etc/hosts` accordingly on both nodes).

Note that these two systems don't necessarily need to have the exact same specifications (e.g. hardware, hard disks). However, the version of openATTIC and the operating system (and particularly the Django version) running on these hosts must be identical.

In the example below, Debian Linux is assumed as the operating system. The path names to configuration files and some configuration details (e.g. PostgreSQL or firewall configuration) might differ on other platforms.

As a first step, you should setup and install these two openATTIC hosts as described in *Installation and Getting Started*.

Note: You should only perform the *Post-installation Configuration* on **one** of the two hosts for now! This example assumes that the command was executed on host **openattic01**, which will result in the installation of the entire openATTIC system including the configuration database on that node.

2.12.2 Step 2 - Database Configuration on openattic01

Next, the PostgreSQL database configuration on **openattic01** needs to be adjusted so it accepts incoming remote connection attempts from **openattic02**.

Edit the `/etc/postgresql/<VERSION>/main/postgresql.conf` and `/etc/postgresql/<VERSION>/main/pg_hba.conf` configuration files on **openattic01**.

Note: The location of these files might be different on other Linux distributions.

First, set the correct listen addresses within the `postgresql.conf` file. Add **openattic01**'s external IP address to `listen_addresses` and uncomment this configuration setting:

```
#------------------------------------------------------------------------------
# CONNECTIONS AND AUTHENTICATION
#------------------------------------------------------------------------------

# - Connection Settings -

listen_addresses = 'localhost, 192.168.1.101'  # what IP address(es) to listen on;
                                  # comma-separated list of addresses;
                                  # defaults to 'localhost'; use '*' for all
```

Note: On some operating systems, the firewall configuration might prevent external communication requests to the TCP port used by PostgreSQL (5432 by default). Please consult your distribution's documentation on how to configure the firewall to accept incoming connections from **openattic02** to this port.

Next, you need to add **openattic02** to PostgreSQL's client authentication configuration file `pg_hba.conf`. Edit the file and add the following line to the IPv4 local connections section as follows:

```
# IPv4 local connections:
host    all             all             127.0.0.1/32            md5
host    openattic       openattic       192.168.1.102/32        md5
```

This ensures that PostgreSQL accepts authentication requests to the local `openattic` database from the remote host **openattic02**.

You need to restart the PostgreSQL service on **openattic01** afterwards, to apply these settings:

```
# systemctl restart postgresql
```

2.12.3 Step 3 - Remote Database Configuration on openattic02

Since **openattic02** needs to connect to the database of **openattic01** you will have to enter the database information (database name, user, password and host) from **openattic01** into the database configuration file `/etc/openattic/database.ini` on **openattic02** manually. The password can be obtained from the `database.ini` file on **openattic01**. The username and database name are `openattic` by default.

The `database.ini` file on **openattic02** should look something like this:

```
[default]
engine   = django.db.backends.postgresql_psycopg2
name     = openattic
user     = openattic
password = <password>
host     = openattic01.yourdomain.com
port     =
```

2.12.4 Step 4 - Execute `oaconfig install` on openattic02

Now that you have configured **openattic02** to connect to the database running on **openattic01**, you can conclude the *Post-installation Configuration* on **openattic02** by executing `oaconfig install` there.

If everything worked out well, you should now see both **openattic01** and **openattic02** in the **Hosts** tab of the web UI running on **openattic01** (and **openattic02** respectively), as well as the disks, pools and volumes of both hosts.

2.13 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

2.13.1 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.

2.13.2 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.

2.13.3 Joining openATTIC to a Windows Active Directory Domain Using `oaconfig`

It is possible to configure openATTIC to join an Microsoft Windows Active Directory Domain for authentication and authorization purposes.

Note: The automatic Domain join using *oaconfig* currently works on Debian/Ubuntu Linux only.

The `oaconfig` tool performs the required steps for joining an Active Directory (AD) domain.

This process requires the following packages and their dependencies to be installed: *openattic-auth-kerberos*, *openattic-module-samba*.

You need to provide your Windows domain name and administrator username and password:

```
# oaconfig domainjoin username yourdomain.com YOURDOMAIN
User:            username
Domain:          yourdomain.com
Realm:           YOURDOMAIN.COM
Workgroup:       YOURDOMAIN
Machine Account: HOSTNAME$
Updating krb5.conf...
Probing Kerberos...
Password for username@YOURDOMAIN.COM: ********
Configuring Samba...
method return sender=:1.248 -> dest=:1.251 reply_serial=2
Removing old keytab...
Joining Domain...
Enter username's password: ********
```

(continues on next page)

(continued from previous page)

```
Using short domain name -- YOURDOMAIN
Joined 'HOSTNAME' to realm 'yourdomain.com'
Processing principals to add...
Logging in as HOSTNAME$ (this may fail a couple of times)...
kinit: Preauthentication failed while getting initial credentials
kinit: Preauthentication failed while getting initial credentials
Configuring openATTIC...
[ ok ] Stopping: openATTIC systemd.
[ ok ] Starting: openATTIC systemd.
[ ok ] Reloading web server config: apache2.
Configuring libnss...
Restarting Samba and Winbind...
Initialized config from /etc/openattic/cli.conf
Could not connect to the server: [Errno 111] Connection refused
Initialized config from /etc/openattic/cli.conf
pong
method return sender=:1.252 -> dest=:1.253 reply_serial=2
[ ok ] Stopping Samba daemons: nmbd smbd.
[ ok ] Starting Samba daemons: nmbd smbd.
[ ok ] Stopping the Winbind daemon: winbind.
[ ok ] Starting the Winbind daemon: winbind.
To see if it worked, let's try 'getent passwd "username"':
username:*:20422:10513:Lastname, Firstname:/home/YOURDOMAIN/username:/bin/true
```

2.13.4 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`.

2.13.5 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).

- `Preauthentication failed while getting initial credentials`

  Possible reason: Wrong password or `/etc/krb5.keytab` is outdated (the latter should not happen because `oaconfig domainjoin` ensures that it is up to date).

- `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`.

2.14 Hardware Recommendations

1. Buy an enclosure with enough room for disks. The absolute minimum recommendation is twelve disks, but if you can, you should add two hot-spares, so make that fourteen. For larger setups, use 24 disks.

   Warning: Any other number of disks will hinder performance.

2. Are you building a storage backend for virtualization? If so, you will require SAS disks, a very clean setup and a good caching mechanism to achieve good performance.

   Note: Using SSDs instead of SAS disks does not necessarily boost performance. A clean setup on SAS disks delivers the same performance as SSDs, and an unclean SSD setup may even be slower.

3. If the enclosure has any room for hot spare disks, you should have some available. This way a disk failure can be dealt with immediately, instead of having to wait until the disk has been replaced.

   Note: A degraded RAID only delivers limited performance. Taking measures to minimize the time until it can resume normal operations is therefore highly advisable.

4. You should have some kind of hardware device for caching. If you're using a RAID controller, make sure it has a BBU installed so you can make use of the integrated cache. For ZFS setups, consider adding two SSDs.

   Note: When using SSDs for caching, the total size of the cache should be one tenth the size of the device being cached, and the cache needs to be ten times faster. So:

   - only add a cache if you have to - no guessing allowed, measure!
   - don't make it too large
   - don't add an SSD cache to a volume that is itself on SSDs

5. Do you plan on using replication in order to provide failure tolerance? If so, ...

   - you will require the same hardware for all of your nodes, because when using synchronous replication, the slowest node limits the performance of the whole system.
   - make sure the network between the nodes has a low latency and enough bandwidth to support not only the bandwidth your application needs, but also has some extra for bursts and recovery traffic.

   Note: When running VMs, a Gigabit link will get you pretty far. Money for a 10GE card would be better spent on faster disks.

6. You should have a dedicated line available for replication and cluster communication. There should be no other active components on that line, so that when the line goes down, the cluster can safely assume its peer to be dead.
7. Up to the supported maximum of 128GB per node, add as much RAM as you can (afford). The operating system will require about 1GB for itself, everything else is then used for things like caching and the ZFS deduplication table. Adding more RAM will generally speed things up and is always a good idea.

2.15 Storage Recommendations

1. Consider dedicating two disks to a RAID1 for the operating system. It doesn't matter if you use hardware or software RAID for this volume, just that you split it off from the rest.

   Note: You can also use other devices to boot from if they fit your redundancy needs.

2. When using hardware RAID:

   (a) Group the other disks into RAID5 arrays of exactly 5 disks each with a chunk size (strip size) of 256KiB. Do not create a partition table on these devices. If your RAID controller does not support 256KiB chunks, use the largest supported chunk size.

   (b) Using mdadm, create a Software-RAID0 device on exactly two or four of your hardware RAID devices. Again, do not create a partition table on the resulting MD device. Make sure the chunk size of the RAID0

array matches that of the underlying RAID5 arrays. This way, you will not be able to add more than 20 disks to one PV. This is intentional. If you need to add more disks, create multiple PVs in the same manner.

(c) Using pvcreate, create an LVM Physical Volume on the MD device and add it to a VG using vgcreate or vgextend.

(d) Do not mix PVs of different speeds in one single VG.

3. When using ZFS:

   You will need to specify the complete layout in the zpool create command, so before running it, consider all the following points.

   (a) Group exactly six disks in each raidz2. Use multiple raidz2 vdevs in order to add all disks to the zpool.

   (b) When adding SSDs, add them as mirrored log devices.

   (c) Set the mount point to /media/<poolname> instead of just /<poolname>.

   (d) Do not use /dev/sdc etc, but use /dev/disk/by-id/... paths instead.

   So, the command you're going to use will look something like this:

```
# zpool create -m /media/tank tank \
  raidz2 /dev/disk/by-id/scsi-3500000e1{1,2,3,4,5,6} \
  raidz2 /dev/disk/by-id/scsi-350000392{1,2,3,4,5,6} \
  log mirror /dev/disk/by-id/scsi-SATA_INTEL_SSD{1,2}
```

2.16 Further Operating System Configuration Hints

1. Disable swap.
2. In a two-node cluster, add a variable named `$PEER` to your environment that contains the hostname (not the FQDN) of the cluster peer node. This simplifies every command that has something to do with the peer. Exchange SSH keys.
3. In pacemaker-based clusters, define the following Shell aliases to make your life easier:

```
alias maint="crm configure property maintenance-mode=true"
alias unmaint="crm configure property maintenance-mode=false"
```

4. After setting up MD raids, make sure `mdadm.conf` is up to date. This can be ensured by running these commands:

```
# /usr/share/mdadm/mkconf > /etc/mdadm/mdadm.conf
# update-initramfs -k all -u
```

5. You may want to install the `ladvd` package, which will ensure that your switches correctly identify your system using LLDP.
6. Make sure `/etc/drbd.d/global_common.conf` contains the following variables:

```
disk {
 no-disk-barrier;
 no-disk-flushes;
 no-md-flushes;
}

net {
```

(continues on next page)

(continued from previous page)

```
 max-buffers 8000;
 max-epoch-size 8000;
}

syncer {
 al-extents 3389;
}
```

CHAPTER 3

User Manual

This section covers the openATTIC web user interface (GUI), focusing on storage tasks like adding volumes and shares, system management tasks like the configuration of users and API credentials, and the integrated monitoring system.

3.1 Administration Guide

3.1.1 Introducing the New Graphical User Interface

The new user interface is now based on Bootstrap to make it look more modern, realizing this was a great advantage when we switched from the ExtJS to the AngularJS JavaScript framework.

We restructured the openATTIC user interface in order to make it more intuitive and user-friendly. This included a clean-up of the menu tree as well. Actions like snapshots and shares are now directly available in the volumes panel - by selecting a volume those options get activated and will only display useful actions, depending on the volume type.

Also, we have integrated wizards on the dashboard so that users can be guided through the single steps based on specific use cases like **VM storage** or **Raw Block Storage**.

3.2 How to Perform Common Tasks

- Dashboard
  - overview of the system (disk load, cpu load)
  - cluster/host status (written data, network traffic)
  - wizards
- Disks
  - displays all disks
  - create pool

- Pools
  - all existing pools
  - add pool
- Volumes
  - volumes overview
  - actions
    - add
    - delete
    - set deletion protection for volume
    - clone
    - resize
  - more options (detail-view)
    - click volume and
      - make a snapshot
      - create clone from snapshot
      - create a share
      - automatically only shows available options for volume type
    - without filesystem
      - only iSCSI/FibreChannel
    - with filesystem
      - http
      - NFS
      - CIFS
      - check performance
- Hosts
  - host overview
  - actions
    - add
      - add attribute (peer, initiator for iSCSI share/FibreChannel WWN for FC share)
- System
  - Users
    - add
    - edit
    - delete
    - update: field “is superuser” was changed to “has all privileges” | “is staff” was changed to “is administrator”

  - Command Logs
    * all nagios logs
    * options
      · delete by date
      · delete
  - CRUSH Map

Removed: API-Keys

3.3 Background-Tasks

3.3.1 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.

3.3.2 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.

3.3.3 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.

3.3.4 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.

3.3.5 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.

3.3.6 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.

3.3.7 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.

CHAPTER 4

Developer Documentation

openATTIC consists of a set of components built on different frameworks, which work together to provide a comprehensive 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 openATTIC code base.

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 Core* 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 systemd System and Service Manager).
- The openATTIC *systemd* executes commands on the system and delivers the results.

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.

4.1 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.

4.2 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*.

4.2.1 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.

4.2.2 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
```

4.2.3 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.

4.2.4 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
```

4.2.5 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.

4.2.6 Debugging openATTIC with PyCharm Community

Please follow the instructions from the official documentation

4.2.7 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 pre_install
python manage.py migrate
```

(continues on next page)

(continued from previous page)

```
python manage.py loaddata */fixtures/initial_data.json
python manage.py createcachetable status_cache
python manage.py add-host
python manage.py makedefaultadmin
python manage.py post_install
```

4.2.8 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.

4.3 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.

4.3.1 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 and Getting Started*, 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
```

4. Install Bower and Grunt (to build the Web UI):

```
# npm install -g bower
# npm install -g grunt-cli
```

5. 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
```

6. 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
```

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

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

8. Now build the Web UI:

```
# cd /srv/openattic/webui
# npm install
# bower install --allow-root
# grunt build
```

   If you intend to make changes to the web interface, it may be useful to run `grunt 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 jshint output), if required.

9. 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.

4.3.2 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

4.4 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*.

4.4.1 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`.

4.4.2 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.

4.4.3 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
```

4.4.4 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.

4.5 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). Use the `flake8` tool to verify that your changes don't violate this style before comitting your modifications.
- 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 grunt-jscs and grunt-contrib-jshint to perform automated syntax and style checks of the JavaScript code. The configuration files for these WebUI tests can be found in file `webui/.jshintrc` and `webui/.jscsrc`, please consult them for more details on the coding style and conventions. By running `grunt inspect --force` on the command line, you can check the quality of your JavaScript code.
- 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 commit message and pull request.
- Pull requests must be accompanied with a corresponding `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.

4.5.1 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.

4.5.2 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

(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:`.

4.5.3 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.

4.5.4 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]")

4.5.5 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.

4.5.6 Database migrations

In order to support database migrations from Django 1.6 onwards, we had to build our own database migration framework. This framework has three major requirements. First, we need to migrate the database without the Django 1.7 migration framework. Second, updates of Django should be possible. Finally, Django updates of already updated databases should work, too.

Our framework will listen to the `django_16_migrate` Django command and will then perform database migrations which are compatible to future Django versions. This allows Django's migration framework to take over existing migrations.

The idea is to execute Django 1.7+ migrations on Django 1.6 by running the same SQL command of later Django versions. You just need to generate the SQL statements by running `sqlmigrate` on a Django 1.7+ installation.

1. Adding an initial migration:

When creating a new migration, don't forget to add the "initial" migration, if it doesn't exist. Keep in mind that you need need at least Django 1.7:

```
~/openattic/backend$ ./manage.py makemigrations taskqueue

Migrations for 'taskqueue':
  0001_initial.py:
    - Create model TaskQueue
```

Then, add the `migrations` directory to your git clone:

```
~/openattic/backend$ git add taskqueue/migrations/__init__.py taskqueue/migrations/
↪0001_initial.py
```

Now, open `backend/sysutils/management/commands/django_16_migrate.py` and add the initial migration to the `_migrations` array:

```
_migrations = [
    ...
    (
        'taskqueue', u'0001_initial', None, None
    ),
]
```

2. Adding a new migration

Call `makemigrations` to create the Django migration, rename your new migration and add it to git:

```
~/openattic/backend$ ./manage.py makemigrations taskqueue

Migrations for 'taskqueue':
  0002_auto_20161216_1059.py:
    - Alter field description on taskqueue
    - Alter field result on taskqueue
~/openattic/backend/taskqueue/migrations$ mv 0002_auto_20161216_1059.py 0002_
↪taskqueue_description_textfield.py
~/openattic/backend$ git add taskqueue/migrations/0002_taskqueue_description_
↪textfield.py
```

Now, call `sqlmigrate` to generate the SQL statement needed for migrating older installations. Note that the generated SQL should be compatible to all supported Postgres versions:

```
~/openattic/backend$ ./manage.py sqlmigrate taskqueue 0002_taskqueue_description_
↪textfield
BEGIN;
ALTER TABLE "taskqueue_taskqueue" ALTER COLUMN "description" TYPE text;
ALTER TABLE "taskqueue_taskqueue" ALTER COLUMN "result" TYPE text;
COMMIT;
```

You can now append your generated migration to the `_migrations` array in `backend/sysutils/management/commands/django_16_migrate.py`:

```
_migrations = [
    ...
    (
        'taskqueue', u'0002_taskqueue_description_textfield',
```

(continues on next page)

(continued from previous page)

```
        test_taskqueue_0002_taskqueue_description_textfield,
        """
        BEGIN;
        ALTER TABLE "taskqueue_taskqueue" ALTER COLUMN "description" TYPE text;
        ALTER TABLE "taskqueue_taskqueue" ALTER COLUMN "result" TYPE text;
        COMMIT;
        """
    ),
]
```

Afterwards, make sure that already applied migrations (by executing `syncdb`) will never be applied again, as this could lead to data loss in future migrations. Thus, create a test function named `test_taskqueue_0002_taskqueue_description_textfield` which returns `True`, **if and only if** the migration should be applied. For example like this:

```
def test_taskqueue_0002_taskqueue_description_textfield(cursor):
    stmt = """SELECT data_type FROM INFORMATION_SCHEMA.COLUMNS
              WHERE table_name = 'taskqueue_taskqueue'
                  AND column_name = 'description';"""
    res = execute_and_fetch(cursor, stmt)
    return len(res) == 1 and res[0]['data_type'] != 'text'
```

Please **review** previous test functions of the same database table, as they should still work as expected.

> **Warning:** Remember to add all migrations to `django_16_migrate.py`, otherwise updating from Django 1.6 to Django 1.7 won't work anymore.

Manually migrating the database:

If you want to perform a manual migration from one database to another, please execute these Django commands:

From \ To	Django 1.6 original DB	Django 1.6 + DB Migrations	Django 1.7 original DB	Django 1.7 + DB Migrations	Django 1.8 original DB	Django 1.8 + DB Migrations
No DB Table	`syncdb`	`syncdb` + `django_16_migrate`	`migrate`	`migrate`	`migrate`	`migrate`
Django 1.6 original DB	`syncdb`	`syncdb` + `django_16_migrate`	`migrate`	`migrate`	`migrate --fake-initial`	`migrate --fake-initial`
Django 1.6 + DB Migrations	unsupported	`syncdb` + `django_16_migrate`	unsupported	`migrate`	unsupported	`migrate --fake-initial`
Django 1.7 original DB	unsupported	unsupported	•	`migrate`	`migrate`	`migrate`
Django 1.7 + DB Migrations	unsupported	unsupported	`migrate`	`migrate`	`migrate`	`migrate`
Django 1.8 original DB	unsupported	unsupported	`migrate`	`migrate`	•	`migrate`
Django 1.8 + DB Migrations	unsupported	unsupported	`migrate`	`migrate`	`migrate`	`migrate`

Notice that `syncdb` will not perform any alterations to exiting database tables, instead it only creates new database tables based on the current model information. Also, notice that `--fake-initial` is required to take over existing database tables without any exiting database migration files in Django 1.8.

In order to add a database migration, take a look at our migration framework in `backend/sysutils/management/commands/django_16_migrate.py`.

4.6 openATTIC Core

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 Framwork 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.

4.6.1 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.

4.6.2 Filesystem API

The filesystem API abstracts handling different file systems, translates actions initiated by the model into commands to be executed and calls Systemd accordingly.

4.7 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 and Getting Started* (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).

4.7.1 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).

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®.

4.7.2 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.

4.7.3 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.

4.8 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.

4.8.1 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">
```

4.8.2 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">
```

4.9 openATTIC Web UI Tests - E2E Test Suite

This section describes how our test environment is set up, as well as how you can run our existing tests on your openATTIC system and how to write your own 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.

4.9.1 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.

4.9.2 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!

4.9.3 Install Protractor

- `npm install -g protractor@4.0.10`

Note: Protractor version 4.x.x requires Node.js® version 4.x (you can check your installed version with `node -v`).

- `apt-get install openjdk-7-jre-headless`
- `webdriver-manager` version should be 10.3.x
- `npm install -g jasmine-beforeAll` (in case this package is not available, try `npm install -g jasmine-before-all`)
- 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

4.9.4 Protractor Configuration

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

Here's what you have to do in `protractor.conf.js`:

4.9.5 Enable `BeforeAll` / `AfterAll`

In order to use `beforeAll` and `afterAll` you need to tell protractor to use `jasmine2` as framework (protractor uses an older version by default, which does not support beforeAll/afterAll).

Add the following line to your protractor.conf:

```
exports.config = {

  seleniumAddress: ...

  jasmineNodeOpts: {
   ....
```

(continues on next page)

(continued from previous page)

```
  },

``framework: 'jasmine2',``

  suites: {
    ...
    ...

  },
  ....
}
```

4.9.6 Maximize Browser Window

If the browser window in which the tests will be executed is too small, it occurs that protractor can't click an element and tests will fail. To prevent this, you can maximize your browser window by default by adding the following line to `webui/protractor.conf.js`:

```
exports.config = {

  seleniumAddress: ...
  jasmineNodeOpts: {
    ....
  },

  framework: 'jasmine2',

  suites: {
    ...
    ...
    ..

  },

``onPrepare: function(){``
  ``browser.driver.manage().window().maximize();``
``},``
}
```

4.9.7 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;
```

4.9.8 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 = {
    url      : 'http://IP-to-your-oA-test-sys/openattic/#/login',
    //leave this if you want to use openATTIC's default user for login
    username: 'openattic',
    password: 'openattic',
  };
}());
```

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.

4.9.9 Start webdriver manager Environment

Use a separate tab/window to run the following command:

```
$ webdriver-manager start
```

4.9.10 Make Protractor Execute the Tests

Go to `/srv/openattic/webui/` and type `protractor protractor.conf.js` in order to run the tests:

```
$ protractor protractor.conf.js (--suite <suiteName>)
```

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

4.9.11 Starting Only a Specific Test Suite

If you only want to test a specific action, you can run i.e. `protractor protractor.conf.js --suite snapshot_add`.

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

```
suites: {
  //suite name       : '/path/to/e2e-test/file.e2e.js'
  snapshot_add       : '../e2e/snapshots/add/**/*.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`

4.9.12 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.conf.js`) 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.

4.9.13 E2E-Test Directory and File Structure

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

```
+-- auth
+-- commandLogs
+-- ceph
+-- dashboard
|   `-- dashboard
+-- disks
+-- general
+-- hosts
+-- pools
+-- pagination
+-- shares
|   +-- cifs
|   +-- http
|   +-- lun
|   `-- nfs
+-- snapshots
|   +-- add
|   `-- clone
+-- users
+-- volumes
|   +-- add
|   +-- protection
|   +-- resize
|   `-- zvol
`-- wizards
    +-- block
    +-- file
    `-- vm
```

Most of the directories contain a `.._workflow.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.

4.9.14 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 `create_volume` function, you just have to add `helpers.` to the function: `helpers.create_volume( name , type [, size ] )`.

The following helper functions are implemented:

- `create_volume`
- `delete_volume`
- `create_snapshot`
- `delete_snapshot`
- `create_snap_clone`
- `delete_snap_clone`
- `create_host`
- `delete_host`

So if you want to write a test and you need a volume to test an action which is based on a volume (i.e. creating a share), you can use the following lines to create a new volume:

```
beforeAll(function(){
  helpers.login();

  //create an xfs volume before executing any test
  helpers.create_volume("volumename_here","xfs");

});
```

You can also specify the size as a string as third argument, otherwise the volume will always be initiated with 100MB by default.

Depending on which volume type you need, you can set the parameter to:

- `xfs`
- `btrfs`
- `zfs` (if `openattic-module-zfs` is installed)
- `lun`

Every helper function which is based on a volume needs to get the volume object passed.:

```
//var volumename = 'demo_volume';
//volume: var volume = element(by.cssContainingText('tr', volumename));

* ``create_snap_clone(volume)``
* ``helpers.delete_volume(volume, volumename);``
* ``helpers.create_snapshot(volume);``
* ``helpers.delete_snapshot(volume);``
```

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`.

Example:

If you put `helpers.delete_volume();` before `helpers.delete_snapshot();` the snapshot will be deleted with the volume and the second one (`delete_snapshot();`) will search for an element which does not longer exist. A second option is to only use `helpes.delete_volume();` so everything which relates to this volumes (like snapshots, shares) will be deleted with the deletion of the volume automatically.

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 "Volumes" before executing the actions defined in
↪'it('', function(){});'
  element.all(by.css('ul .tc_menuitem')).get(3);

});
```

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

- `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)
- Put required files at the top of the file
- Do not make tests complex by using a lot of for loops, if statements or even nested functions
- If something has to be done frequently 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 -> `create_volume`-/ `delete_volume`-function)
- Make use of the beforeAll/afterAll-functions if possible (see the `Install Protractor` instructions). Those functions allow you to do some steps (which are only required once) before 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 beforeEach `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.
- If possible use protractor locators like `by.model` or `by.binding` (those are performant locators). Example:

```
<ul class="example">
   <li>{{volume.name}}</li>
</ul>
```

-> Avoid doing: `var elementName = element.all(by.css('.example li')).get(0);` -> Recommended: `var elementName = element(by.binding('volume.name'));`

- If `by.model` or `by.binding` is not available, try using locators like `by.id` or `by.css` (those are also performant locators)
- Avoid using text locators like `by.linkText`, `by.buttonText` or `by.cssContainingText` at least for text which tend to change over time / often (like buttons, links and labels)
- Try to avoid using `xpath` - it is a very slow locator. Xpath expressions are hard to read and to debug
- 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
- Make test spec files independent from each other because it's not guaranteed that test files will be executed in a specific order
- 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.
- Locators and specs should apply to the Jasmine2 and Protractor version 3.x.x functionalities
- Make sure that written tests do work in Chrome (v. 49.x.x) and Firefox (v. 45.x)
- The name of folders/files should tell what the test is about (i.e. folder "user" contains "user_add.e2e.js")
- "Workflow"-files contain tests which do not place value on functionalities itself (i.e. add, delete, edit something) but check validation and user feedback in forms or dialogs (like error messages)

4.9.16 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();
```

4.9.17 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`.

4.10 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.

4.10.1 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 and Getting Started*) 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.

4.10.2 Dependencies

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

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

4.10.3 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.

4.10.4 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.

4.10.5 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.

4.10.6 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.

CHAPTER 5

Indices and Tables

- genindex
- modindex
- search