
Murano Documentation

Release 15.1.0.dev5

OpenStack Foundation

Aug 15, 2023

CONTENTS

1	Introduction to Murano	3
1.1	Overview	3
1.1.1	Key features	3
	Application catalog	3
	Application catalog management	4
	Application lifecycle management	4
1.1.2	Target users	4
	Cloud administrators	5
	Cloud end users	5
1.1.3	Architecture	5
1.1.4	Use cases	6
1.1.5	Appendix	7
	High-level definitions of Murano concepts	7
	Tutorials	7
	REST API specification	7
	Murano command-line client	7
	Glossary	17
	Miscellaneous	17
2	Using Murano	75
2.1	QuickStart	75
2.1.1	Upload an application	75
2.1.2	Deploy an application	77
2.1.3	Delete an application	79
2.2	User Guide	80
2.2.1	Managing environments	80
	Create an environment	80
	Edit an environment	81
	Review an environment	81
2.2.2	Managing applications	82
	Import an application package	82
	Search for an application in the catalog	87
	Delete an application package	89
	Add an application to environment	90
	Deploy an environment	95
	Delete an application	102
2.2.3	Log in to murano-spawned instance	103
2.2.4	Using CLI	105
	Install and use the murano client	105

Manage environments	107
Manage packages	109
Manage categories	115
Manage environment templates	115
2.2.5 Deploying environments using CLI	116
Create an environment	116
Create a configuration session	117
Add applications to an environment	117
Verify your object model	118
Deploy your environment	118
2.2.6 Support for OpenStack regions	119
Deploy an app in the current region	119
Associate environments with regions	119
3 Installation	121
3.1 Application Catalog service	121
3.1.1 Application Catalog service overview	121
3.1.2 Install and configure	121
Install Murano API	122
Install Murano Dashboard	125
Install Murano from Source	126
Network Configuration	131
SSL configuration	132
3.1.3 Verify operation	134
3.1.4 Next steps	134
Import Murano Applications	134
Additional Resources	135
4 Configuration	137
4.1 Configuration Guide	137
5 CLI Guide	139
5.1 Murano CLI Documentation	139
5.1.1 murano-status	139
CLI interface for Murano status commands	139
6 Administrator Documentation	141
6.1 Deploying Murano	141
6.1.1 Deploying murano	141
System requirements	141
Integrate murano with DevStack	144
Install murano manually	145
Configure SSL	151
6.1.2 Prepare a lab for murano	152
System prerequisites	153
Test your lab host performance	154
Baseline data	155
Host optimizations	155
6.1.3 Murano Policies	156
Sample File Generation	156
Merged File Generation	156
List Redundant Configurations	156

	Policy configuration	157
	Default Murano Policies	158
6.1.4	Managing packages	158
	Managing packages on engine side	158
6.1.5	Managing images	159
	Build an image	159
	Manage images	159
6.1.6	Managing categories	159
6.1.7	Murano repository	159
	Use an existing repository	159
	Set up a custom repository	159
6.1.8	Murano agent	159
	Murano-agent on a new VM	159
	Interaction with murano-engine	159
	Execution plans and execution plan templates	160
6.1.9	Policy enforcement using Congress	161
	Setting up policy enforcement	162
	Creating policy enforcement rules	164
	Murano policy enforcement internals	166
	Using policy for the base modification of an environment	169
6.1.10	Using Glare as a storage for packages	171
	DevStack installation	171
	Set up Glare API endpoint manually	173
6.1.11	Network configuration	173
	Nova-network support	173
	Neutron support	173
6.1.12	Murano service broker for Cloud Foundry	177
	Service broker overview	177
	Configure service broker	177
	How to use service broker	179
	Known issues	180
	Useful links	180
6.1.13	Troubleshooting	180
	Log location	180
	Issues during configuration	180
	Issues during deployment	181
6.1.14	Application Developer Guide	183
	Developing Murano Packages 101	183
	Execution plan template	218
	HOT packages	220
	MuranoPL Reference	225
	Murano packages	263
	Murano bundles	278
	Migrating applications between releases	279
	Application unit tests	289
	Cinder volume support	293
	Multi-region application	295
	Examples	298
	Use-cases	298
	Application development framework	303
	Application developer's cookbook	318

Garbage collection system in MuranoPL	319
Managing Sensitive Data in Murano	320
6.1.15 Installing Murano API via WSGI	321
WSGI Application	322
Murano API behind uwsgi	322
Murano API behind mod_wsgi	322
7 First App Guide	325
7.1 My first Murano App getting started guide	325
7.1.1 Prerequisites	325
/source	325
/samples	325
Contents	326
8 Application Developer Documentation	329
8.1 FAQ	329
9 Contributor Documentation	331
9.1 So You Want to Contribute...	331
9.1.1 Communication	331
9.1.2 Contacting the Core Team	331
9.1.3 New Feature Planning	331
9.1.4 Task Tracking	331
9.1.5 Reporting a Bug	332
9.1.6 Getting Your Patch Merged	332
9.1.7 Project Team Lead Duties	332
9.2 Contributor Guide	332
9.2.1 How to contribute	332
9.2.2 Development guidelines	332
Conventions	332
High-level overview of Murano components	332
Coding guidelines	332
Debug tips	333
9.2.3 Murano plug-ins	333
MuranoPL extension plug-ins	333
MuranoPL package type plug-ins	334
Creating a Murano plug-in	337
Installing a plug-in	337
Plug-in versioning	338
Organization	338
9.2.4 Development environment	338
9.2.5 Testing	338
Testing guidelines	338
Continuous Integration service	338
UI testing	338
Tempest tests	338
Automated testing machinery	339
9.2.6 Documentation guidelines	339
9.2.7 Backporting to stable/branches	339
Upstream support phases	339
Bug nomination process	340

Murano is an open source OpenStack project that combines an application catalog with versatile tooling to simplify and accelerate packaging and deployment. It can be used with almost any application and service in OpenStack.

Murano project consists of several source code repositories:

- `murano` -- the main repository. It contains code for Murano API server, Murano engine and MuranoPL.
- `murano-agent` -- the agent that runs on guest VMs and executes the deployment plan.
- `murano-dashboard` -- Murano UI implemented as a plugin for the OpenStack Dashboard.
- `python-muranoclient` -- Client library and CLI client for Murano.

Note: *Administrator Documentation*, *Contributor Documentation*, and *Appendix* are under development at the moment.

INTRODUCTION TO MURANO

1.1 Overview

1.1.1 Key features

Murano has a number of features designed to interact with the application catalog, for instance managing what's in the catalog, and determining how apps in the catalog are deployed.

Application catalog

1. Easy browsing:
 - Icons display applications for point-and-click and drag-and-drop selection and deployment.
 - Each application provides configuration information required for deploying it to a cloud.
 - An application topology of your environment is available in a separate tab, and shows the number of instances spawned by each application.
 - The presence of the *Quick Deploy* button on the applications page saves the time.
2. Quick filtering by:
 - Tags and words included in application name and description.
 - Recent activity.
 - Predefined category.
3. Dependency tracking:
 - Automatic detection of dependent applications that minimizes the possibility of an application deployment with incorrect configuration.
 - No underlying IaaS configuration knowledge is required.

Application catalog management

1. Easy application uploading using UI or CLI from:
 - Local zip file.
 - URL.
 - Package name, using an application repository.
2. Managing applications include:
 - Application organization in categories or transfer between them.
 - Application name, description and tags update.
 - Predefined application categories list setting.
3. Deployment tracking includes the availability of:
 - Logs for deployments via UI.
 - Deployment modification history to track the recent changes.

Application lifecycle management

1. Simplified configuration and integration:
 - It is up to an application developer to decide what their application will be able to do.
 - Dependencies between applications are easily configured.
 - New applications can be connected with already existing ones.
 - Well specified application actions are available.
2. HA-mode and auto-scaling:
 - Application authors can set up any available monitoring system to track application events and call corresponding actions, such as failover, starting additional instances, and others.
3. Isolation:
 - Applications in the same environments can easily interact with each other, though applications between different projects (tenants) are isolated.

1.1.2 Target users

Cloud end users want to simply use applications as opposed to installing and managing them. Cloud administrators, in turn, would like to offer a well tested set of on demand self-service applications to dramatically reduce their support burden.

Murano solves the problems of both constituents. It enables cloud administrators to publish cloud-ready applications in an online catalog. Cloud end users can use the catalog to deploy these on demand applications, reliably and consistently, with a button click.

Cloud administrators

For cloud administrators Murano provides UI and API to easily compose, deploy, run applications, and manage their lifecycle.

Designed to be operating system independent, it can handle apps on all manner of the environments in the cloud, either Windows or Linux/Unix-based operating systems.

It can be used to pre-configure and deploy anything that can run in the cloud, from low-level networking services to end-user applications. By automating these ongoing cloud application management activities, Murano speeds up the deployment, even for complex distributed applications, without sacrificing simplicity of use.

Cloud end users

Murano catalog lets cloud end users choose from the available applications and services, and compose reliable distributed environments with an intuitive UI. Even users unfamiliar with cloud environments can easily deploy cloud-aware applications.

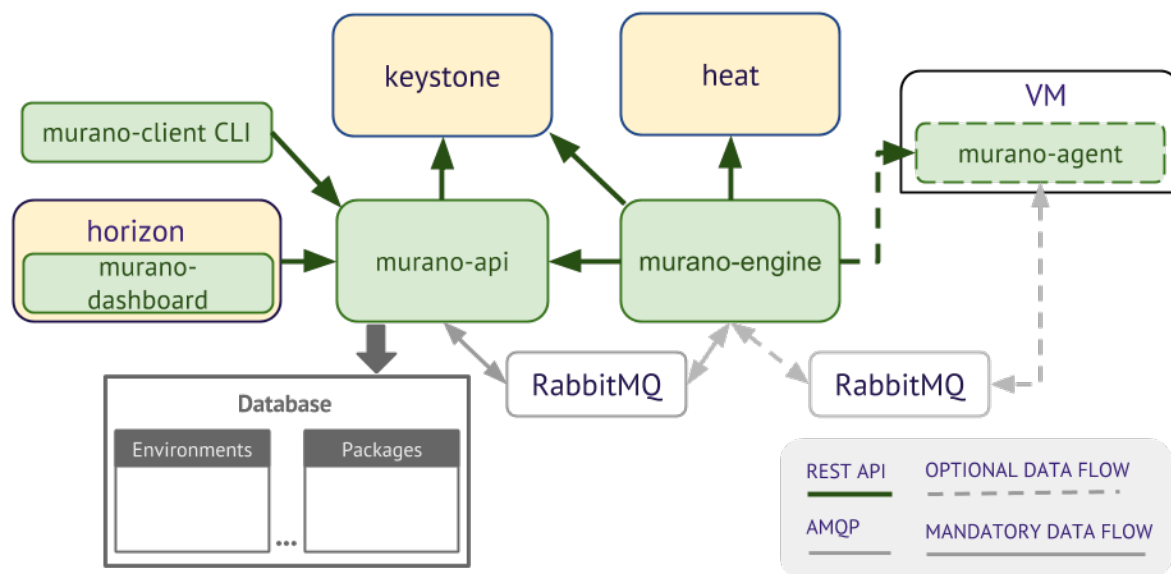
Murano masks cloud-infrastructure specifics from end users, letting them reliably compose and deploy applications in the cloud for the widest range of workloads and use cases without touching IaaS internals.

1.1.3 Architecture

Murano is composed of the following major components:

- murano command-line client
- murano-dashboard
- murano-api
- murano-engine
- murano-agent

They interact with each other as illustrated in the following diagram:



All remote operations on users' servers, such as software installation and configuration, are carried out through an AMQP queue to the murano-agent. Such communication can easily be configured on a separate instance of AMQP to ensure that the infrastructure and servers are isolated.

Besides, Murano uses other OpenStack services to prevent the reimplementing of the existing functionality. Murano interacts with these services using their REST API through their python clients.

The external services used by Murano are:

- the **Orchestration service** (Heat) to orchestrate infrastructural resources such as servers, volumes, and networks. Murano dynamically creates heat templates based on application definitions.
- the **Identity service** (Keystone) to make murano API available to all OpenStack users.

1.1.4 Use cases

IT-as-a-Service

An *IT organization* manages applications and controls the applications availability to different OpenStack cloud users in a simple and timesaving manner.

A *cloud end user* can easily find and deploy any available application from the catalog.

Self-service portal

An *application developer* and *quality assurance engineer* reduces efforts on testing an application for compatibility with other applications, databases, platforms, and other components it depends on, by configuring compound combinations of applications dynamically and deploying environments that satisfy all requirements within minutes.

Glue layer use case

A *cloud end user* is able to link an ever growing number of technologies to any application in an OpenStack cloud with a minimum cost due to the powerful Murano architecture.

Currently, Murano applications have been integrated with the following technologies: Docker, Legacy apps VMs or bare metal, apps outside of OpenStack, and others.

The following technologies are to become available in the future: Cloudify and TOSCA, Apache Brooklyn, and APS.

1.1.5 Appendix

High-level definitions of Murano concepts

Tutorials

Integration with Docker

Integration with Kubernetes

HA and autoscaling

REST API specification

Murano command-line client

The `murano` client is the command-line interface (CLI) for the Application catalog API and its extensions.

For help on a specific `murano` command, enter:

```
murano help COMMAND

murano usage
usage: murano [--version] [-d] [-v] [-k] [--os-cacert <ca-certificate>]
  [--cert-file CERT_FILE] [--key-file KEY_FILE]
  [--ca-file CA_FILE] [--api-timeout API_TIMEOUT]
  [--os-username OS_USERNAME] [--os-password OS_PASSWORD]
  [--os-tenant-id OS_TENANT_ID] [--os-tenant-name OS_TENANT_NAME]
  [--os-auth-url OS_AUTH_URL] [--os-region-name OS_REGION_NAME]
  [--os-auth-token OS_AUTH_TOKEN] [--os-no-client-auth]
  [--murano-url MURANO_URL] [--glance-url GLANCE_URL]
  [--murano-api-version MURANO_API_VERSION]
  [--os-service-type OS_SERVICE_TYPE]
  [--os-endpoint-type OS_ENDPOINT_TYPE] [--include-password]
  [--murano-repo-url MURANO_REPO_URL]
  <subcommand> ...
```

Subcommands

- *bundle-import* Import a bundle.
- *category-create* Create a category.
- *category-delete* Delete a category.
- *category-list* List all available categories.
- *category-show*
- *deployment-list* List deployments for an environment.
- *env-template-add-app* Add application to the environment template.
- *env-template-create* Create an environment template.
- *env-template-del-app* Delete application to the environment template.
- *env-template-delete* Delete an environment template.
- *env-template-list* List the environments templates.
- *env-template-show* Display environment template details.
- *env-template-update* Update an environment template.
- *environment-create* Create an environment.
- *environment-delete* Delete an environment.
- *environment-list* List the environments.
- *environment-rename* Rename an environment.
- *environment-show* Display environment details.
- *package-create* Create an application package.
- *package-delete* Delete a package.
- *package-download* Download a package to a filename or stdout.
- *package-import* Import a package.
- *package-list* List available packages.
- *package-show* Display details for a package.
- *service-show*
- *bash-completion* Prints all of the commands and options to stdout.
- *help* Display help about this program or one of its subcommands.

Murano optional arguments

--version

show program's version number and exit

-d, --debug

Defaults to env[MURANOCLIENT_DEBUG]

-v, --verbose

Print more verbose output

-k, --insecure

Explicitly allow muranoclient to perform "insecure" SSL (https) requests. The server's certificate will not be verified against any certificate authorities. This option should be used with caution.

--os-cacert <ca-certificate>

Specify a CA bundle file to use in verifying a TLS (https) server certificate. Defaults to env[OS_CACERT]

--cert-file CERT_FILE

Path of certificate file to use in SSL connection. This file can optionally be prepended with the private key.

--key-file KEY_FILE

Path of client key to use in SSL connection. This option is not necessary if your key is prepended to your cert file.

--ca-file CA_FILE

Path of CA SSL certificate(s) used to verify the remote server certificate. Without this option glance looks for the default system CA certificates.

--api-timeout API_TIMEOUT

Number of seconds to wait for an API response, defaults to system socket timeout

--os-username OS_USERNAME

Defaults to env[OS_USERNAME]

--os-password OS_PASSWORD

Defaults to env[OS_PASSWORD]

--os-project-id OS_PROJECT_ID

Defaults to env[OS_PROJECT_ID]

--os-project-name OS_PROJECT_NAME

Defaults to env[OS_PROJECT_NAME]

--os-auth-url OS_AUTH_URL

Defaults to env[OS_AUTH_URL]

--os-region-name OS_REGION_NAME

Defaults to env[OS_REGION_NAME]

--os-auth-token OS_AUTH_TOKEN

Defaults to env[OS_AUTH_TOKEN]

--os-no-client-auth

Do not contact keystone for a token. Defaults to env[OS_NO_CLIENT_AUTH].

- murano-url MURANO_URL**
Defaults to env[MURANO_URL]**
- glance-url GLANCE_URL**
Defaults to env[GLANCE_URL]
- murano-api-version MURANO_API_VERSION**
Defaults to env[MURANO_API_VERSION] or 1
- os-service-type OS_SERVICE_TYPE**
Defaults to env[OS_SERVICE_TYPE]
- os-endpoint-type OS_ENDPOINT_TYPE**
Defaults to env[OS_ENDPOINT_TYPE]
- include-password**
Send os-username and os-password to murano.
- murano-repo-url MURANO_REPO_URL**
Defaults to env[MURANO_REPO_URL] or *http://storage.apps.openstack.org_*

Application catalog API v1 commands

murano bundle-import

Import a bundle. FILE can be either a path to a zip file, URL or name from repo. if FILE is a local file does not attempt to parse requirements and treat Names of packages in a bundle as file names, relative to location of bundle file.

Positional arguments

- <FILE>**
Bundle URL, bundle name, or path to the bundle file

Optional arguments

- is-public**
Make packages available to users from other project
- exists-action {a,s,u}**
Default action when a package already exists

murano category-create

Create a category.

Positional arguments

<CATEGORY_NAME>
Category name

murano category-delete

Delete a category.

Positional arguments

<ID>
ID of a category(s) to delete

murano category-list

List all available categories.

murano category-show

Positional arguments

<ID>
ID of a category(s) to show

murano deployment-list

List deployments for an environment.

Positional arguments

<ID>
Environment ID for which to list deployments

murano env-template-add-app

Add application to the environment template.

Positional arguments

<ENV_TEMPLATE_NAME>
Environment template name

<FILE>
Path to the template.

murano env-template-create

Create an environment template.

Positional arguments

<ENV_TEMPLATE_NAME>
Environment template name

murano env-template-del-app

Delete application to the environment template.

Positional arguments

<ENV_TEMPLATE_ID>
Environment template ID

<ENV_TEMPLATE_APP_ID>
Application ID

murano env-template-delete

Delete an environment template.

Positional arguments

<ID>

ID of environment(s) template to delete

murano env-template-list

List the environments templates.

murano env-template-show

Display environment template details.

Positional arguments

<ID>

Environment template ID

murano env-template-update

Update an environment template.

Positional arguments

<ID>

Environment template ID

<ENV_TEMPLATE_NAME>

Environment template name

murano environment-create

Create an environment.

Positional arguments

<ENVIRONMENT_NAME>

Environment name

murano environment-delete

Delete an environment.

Positional arguments

<NAME or ID>

ID or name of environment(s) to delete

Optional arguments

--abandon

If set will abandon environment without deleting any of its resources

murano environment-list

List the environments.

murano environment-rename

Rename an environment.

Positional arguments

<NAME or ID>

Environment ID or name

<ENVIRONMENT_NAME>

A name to which the environment will be renamed

murano environment-show

Display environment details.

Positional arguments

<NAME or ID>

Environment ID or name

murano package-create

Create an application package.

Optional arguments

- t <HEAT_TEMPLATE>, --template <HEAT_TEMPLATE>**
Path to the Heat template to import as an Application Definition
- c <CLASSES_DIRECTORY>, --classes-dir <CLASSES_DIRECTORY>**
Path to the directory containing application classes
- r <RESOURCES_DIRECTORY>, --resources-dir <RESOURCES_DIRECTORY>**
Path to the directory containing application resources
- n <DISPLAY_NAME>, --name <DISPLAY_NAME>**
Display name of the Application in Catalog
- f <full-name>, --full-name <full-name>**
Fully-qualified name of the Application in Catalog
- a <AUTHOR>, --author <AUTHOR>**
Name of the publisher
- tags [<TAG1 TAG2> [<TAG1 TAG2> ...]]**
A list of keywords connected to the application
- d <DESCRIPTION>, --description <DESCRIPTION>**
Detailed description for the Application in Catalog
- o <PACKAGE_NAME>, --output <PACKAGE_NAME>**
The name of the output file archive to save locally
- u <UI_DEFINITION>, --ui <UI_DEFINITION>**
Dynamic UI form definition
- type TYPE**
Package type. Possible values: Application or Library
- l <LOGO>, --logo <LOGO>**
Path to the package logo

murano package-delete

Delete a package.

Positional arguments

<ID>

Package ID to delete

murano package-download

Download a package to a filename or stdout.

Positional arguments

<ID>

Package ID to download

file

Filename for download (defaults to stdout)

murano package-import

Import a package. FILE can be either a path to a zip file, URL or a FQPN. categories can be separated by a comma.

Positional arguments

<FILE>

URL of the murano zip package, FQPN, or path to zip package

Optional arguments

-c [<CAT1 CAT2 CAT3> [<CAT1 CAT2 CAT3> ...]], --categories [<CAT1 CAT2 CAT3> [<CAT1 CAT2 CAT3> ...]]

Category list to attach

--is-public

Make the package available for user from other project

--package-version VERSION

Version of the package to use from repository (ignored when importing with multiple packages)

--exists-action {a,s,u}

Default action when package already exists

murano package-list

List available packages.

Optional arguments

--include-disabled

murano package-show

Display details for a package.

Positional arguments

<ID>

Package ID to show

murano service-show

Positional arguments

<ID>

Environment ID to show applications from

Optional arguments

-p <PATH>, --path <PATH>

Level of detalization to show. Leave empty to browse all services in the environment

Glossary

Miscellaneous

Background Concepts for Murano

Murano workflow

What happens when a component is being created in an environment? This document will use the Telnet package referenced elsewhere as an example. It assumes the package has been previously uploaded to Murano.

Step 1. Begin deployment

The API sends a message that instructs `murano-engine`, the workflow component of Murano, to deploy an environment. The message consists of a JSON document containing the class types required to create the environment, as well as any parameters the user selected prior to deployment. Examples are:

- An *Class: Environment* object (`io.murano.Environment`) with a *name*
- An object (or objects) referring to networks that need to be created or that already exist
- A list of Applications (e.g. `io.murano.apps.linux.Telnet`). Each Application will contain, or will reference, anything it requires. The Telnet example, has a property called *instance* whose contract states it must be of type `io.murano.resources.Instance`. In turn the Instance has properties it requires (like a name, a flavor, a keypair name).

Each object in this *model* has an ID so that the state of each can be tracked.

The classes that are required are determined by the application's manifest. In the Telnet example only one class is explicitly required; the telnet application definition.

The Telnet class definition refers to several other classes. It extends *Class: Application* and it requires an *Class: Instance*. It also refers to the *Class: Environment* in which it will be contained, sends reports through the environment's *Class: StatusReporter* and adds security group rules to the *Class: SecurityGroupManager*.

Step 2. Load definitions

The engine makes a series of requests to the API to download packages it needs. These requests pass the class names the environment will require, and during this stage the engine will validate that all the required classes exist and are accessible, and will begin creating them. All Classes whose *workflow* sections contain an *initialize* fragment are then initialized. A typical initialization order would be (defined by the ordering in the *model* sent to the `murano-engine`):

- *Class: Network*
- *Class: Instance*
- *Class: Object*
- *Class: Environment*

Step 3. Deploy resources

The workflow defined in `Environment.deploy` is now executed. The first step typically is to initialize the messaging component that will pay attention to `murano-agent` (see later). The next stage is to deploy each application the environment knows about in turn, by running `deploy()` for each application. This happens concurrently for all the applications belonging to an instance.

In the Telnet example (under *Workflow*), the workflow dictates sending a status message (via the environment's *reporter*, and configuring some security group rules. It is at this stage that the engine first contacts Heat to request information about any pre-existing resources (and there will be none for a fresh deploy) before updating the new Heat template with the security group information.

Next it instructs the engine to deploy the *instance* it relies on. A large part of the interaction with Heat is carried out at this stage; the first thing an Instance does is add itself to the environment's network. Since

the network doesn't yet exist, murano-engine runs the neutron network workflow which pushes template fragments to Heat. These fragments can define: * Networks * Subnets * Router interfaces

Once this is done the Instance itself constructs a Heat template fragment and again pushes it to Heat. The Instance will include a *userdata* script that is run when the instance has started up, and which will configure and run murano-agent.

Step 4. Software configuration via murano-agent

If the workflow includes murano-agent components (and the telnet example does), typically the application workflow will execute them as the next step.

In the telnet example, the workflow instructs the engine to load *DeployTelnet.yaml* as YAML, and pass it to the murano-agent running on the configured instance. This causes the agent to execute the *EntryPoint* defined in the agent script (which in this case deploys some packages and sets some iptables rules).

Step 5. Done

After execution is finished, the engine sends a last message indicating that fact; the API receives it and marks the environment as deployed.

Tutorials

Building Murano Image

MS Windows image builder for OpenStack Murano

Introduction

This repository contains MS Windows templates, powershell scripts and bash scripted logic used to create qcow2 images for QEMU/KVM based virtual machines used in OpenStack.

MS Windows Versions

Supported by builder versions with en_US localization:

- Windows 2012 R2
- Windows 2012 R2 Core
- Windows 2008 R2
- Windows 2008 R2 Core

Getting Started

Trial versions of Windows 2008 R2 / 2012 R2 used by default. You could use these images for 180 days without activation. You could download evaluation versions from official Microsoft website:

- [\[Windows 2012 R2 - download\]](#)
- [\[Windows 2008 R2 - download\]](#)

System requirements

- Debian based Linux distribution, like Ubuntu, Mint and so on.
- Packages required: `qemu-kvm virt-manager virt-goodies virtinst bridge-utils libvirt-bin uuid-runtime samba samba-common cifs-utils`
- User should be able to run `sudo` without password prompt!

```
sudo echo "${USER}    ALL = NOPASSWD: ALL" > /etc/sudoers.d/${USER}
sudo chmod 440 /etc/sudoers.d/${USER}
```

- Free disk space > 50G on partition where script will spawn virtual machines because of 40G required by virtual machine HDD image.
- Internet connectivity.
- Samba shared resource.

Configuring builder

Configuration parameters to tweak:

[default]

- `workdir` - place where script would prepare all software required by build scenarios. By *default* is not set, i.e. script directory would used as root of working space.
- `vmsworkdir` - must contain valid path, this parameter tells script where it should spawn virtual machines.
- `runparallel` - *true* or *false*, **false** set by default. This parameter describes how to start virtual machines, one by one or in launch them in background.

[samba]

- `mode` - *local* or *remote*. In local mode script would try to install and configure Samba server locally. If set to remote, you should also provide information about connection.
- `host` - in local mode - is 192.168.122.1, otherwise set proper ip address.
- `user` - set to **guest** by default in case of guest rw access.
- `domain` - Samba server user domain, if not set *host* value used.
- `password` - Samba server user password.
- `image-builder-share` - Samba server remote directory.

MS Windows install preparation:

[win2k12r2] or [win2k8r2] - shortcuts for 2012 R2 and 2008 R2.

- `enabled` - *true* or *false*, include or exclude release processing by script.
- `editions` - standard, core or both(space used as delimiter).
- `iso` - local path to iso file

By default [win2k8r2] - disabled, if you need you can enable this release in *config.ini* file.

Run

Preparation

Run `chmod +x *.sh` in builder directory to make script files executable.

Command line parameters:

`runme.sh` - the main script

- `--help` - shows usage
- `--forceinstall-dependencies` - Runs dependencies install.
- `--check-smb` - Run checks or configuration of Samba server.
- `--download-requirements` - Download all required and configures software except MS Windows ISO.
- `--show-configured` - Shows configured and available to use MS Windows releases.
- `--run` - normal run

Experimental options:

- `--config-file` - Set configuration file location instead of default.

Use cases

All examples below describes changes in `config.ini` file

1. I want to build one image for specific version and edition. For example: version - **2012 R2** and edition - **standard**. Steps to reach the goal:

- Disable [win2k8r2] from script processing.

```
[win2k8r2]
enabled=false
```

- Update [win2k12r2] with desired edition(**standard**).

```
[win2k12r2]
enabled=true
editions=standard
```

- Execute `runme.sh --run`
2. I want to build two images for specific version with all supported by script editions. For example: **2012 R2** and editions - **standard** and **core**. Steps to reach the goal:
- Disable `[win2k8r2]` from script processing.

```
[win2k8r2]
enabled=false
```

- Update `[win2k12r2]` with desired editions(**standard** and **core**).

```
[win2k12r2]
enabled=true
editions=standard core
```

- Execute `runme.sh --run`
3. I want to build two images for all supported by script versions with specific editions. For example: versions - **2012 R2** and **2008 R2** and edition - **core**. Steps to reach the goal:
- Update `[win2k8r2]` with desired edition(**core**).

```
[win2k8r2]
enabled=true
editions=core
```

- Update `[win2k12r2]` with desired edition(**core**).

```
[win2k12r2]
enabled=true
editions=core
```

- Execute `runme.sh --run`

Linux Image

At the moment the best way to build a Linux image with the murano agent is to use disk image builder.

Note: Disk image builder requires sudo rights

The process is quite simple. Let's assume that you use a directory `~/git` for cloning git repositories:

```
export GITDIR=~/git
mkdir -p $GITDIR
```

Clone the components required to build an image to that directory:

```
cd $GITDIR
git clone https://opendev.org/openstack/murano
git clone https://opendev.org/openstack/murano-agent
```

Install diskimage-builder

```
sudo pip install diskimage-builder
```

Install additional packages required by disk image builder:

```
sudo apt-get install qemu-utils curl python3-tox
```

Export paths where additional dib elements are located:

```
export ELEMENTS_PATH=$GITDIR/murano/contrib/elements:$GITDIR/murano-agent/
↪contrib/elements
```

Build Ubuntu-based image with the murano agent:

```
disk-image-create vm ubuntu murano-agent -o murano-agent.qcow2
```

If you need a Fedora based image, replace 'ubuntu' to 'fedora' in the last command.

It'll take a while (up to 30 minutes if your hard drive and internet connection are slow).

When you are done upload the murano-agent.qcow2 image to glance and play :)

Upload image into glance

To deploy applications with murano, virtual machine images should be uploaded into glance in a special way - *murano_image_info* property should be set.

1. Use the OpenStack client image create command to import your disk image to glance:

```
openstack image create --public \
> --disk-format qcow2 --container-format bare \
> --file <IMAGE_FILE> --property <IMAGE_METADATA> <NAME>
```

Replace the command line arguments to openstack image create with the appropriate values for your environment and disk image:

- Replace **<IMAGE_FILE>** with the local path to the image file to upload. E.g. **ws-2012-std.qcow2**.
- Replace **<IMAGE_METADATA>** with the following property string
- Replace **<NAME>** with the name that users will refer to the disk image by. E.g. **ws-2012-std**

```
murano_image_info='{"title": "Windows 2012 Standard Edition", "type":
↪"windows.2012"}'
```

where:

- **title** - user-friendly description of the image

- **type** - murano image type, see *Murano image types*

2. To update metadata of the existing image run the command:

```
openstack image set --property <IMAGE_METADATA> <IMAGE_ID>
```

- Replace **<IMAGE_METADATA>** with `murano_image_info` property, e.g.
- Replace **<IMAGE_ID>** with image id from the previous command output.

```
murano_image_info='{"title": "Windows 2012 Standard Edition", "type":  
↪ "windows.2012"}'
```

Warning: The value of the `--property` argument (named `murano_image_info`) is a JSON string. Only double quotes are valid in JSON, so please type the string exactly as in the example above.

Note: Existing images could be marked in a simple way in the horizon UI with the murano dashboard installed. Navigate to *Applications -> Manage -> Images -> Mark Image* and fill up a form:

- **Image** - ws-2012-std
 - **Title** - My Prepared Image
 - **Type** - Windows Server 2012
-

After these steps desired image can be chosen in application creation wizard.

Murano image types

Type Name	Description
windows.2012	Windows Server 2012
linux	Generic Linux images, Ubuntu / Debian, RedHat / Centos, etc
cirros.demo	Murano demo image, based on CirrOS

Murano automated tests description

This page describes automated tests for a Murano project:

- where tests are located
- how they are run
- how to execute tests on a local machine
- how to find the root of problems with FAILED tests

Murano continuous integration service

Murano project has separate CI server, which runs tests for all commits and verifies that new code does not break anything.

Murano CI uses OpenStack QA cloud for testing infrastructure.

Murano CI url: <https://murano-ci.mirantis.com/jenkins/> Anyone can login to that server, using launchpad credentials.

There you can find each job for each repository: one for **murano** and another one for **murano-dashboard**.

- `gate-murano-dashboard-ubuntu*` verifies each commit to the murano-dashboard repository
- `gate-murano-ubuntu*` verifies each commit to the murano repository

Other jobs allow one to build and test Murano documentation and to perform other useful work to support the Murano CI infrastructure. All jobs are run following a fresh installation of the operating system and all components are installed on each run.

UI tests

The Murano project has a web user interface and all possible user scenarios should be tested. All UI tests are located at <https://opendev.org/openstack/murano-dashboard/src/branch/master/muranodashboard/tests/functional>.

Automated tests for the Murano web UI are written in Python using the special Selenium library. This library is used to automate web browser interactions with Python. See official [Selenium documentation](#) for details.

Prerequisites:

- Install the Python module called nose using either the **easy_install nose** or **pip install nose** command. This will install the nose libraries, as well as the nosetests script, which you can use to automatically discover and run tests.
- Install external Python libraries, which are required for the Murano web UI tests: `testtools` and `selenium`.
- Verify that you have one of the following web browsers installed:
 - Mozilla Firefox 46.0

Note: If you do not have Firefox package out of the box, install and remove it. Otherwise, you will need to install dependent libraries manually. To downgrade Firefox:

```
apt-get remove firefox
wget https://ftp.mozilla.org/pub/firefox/releases/46.0/linux-x86_64/
en-US/firefox-46.0.tar.bz2
tar -xjf firefox-46.0.tar.bz2
rm -rf /opt/firefox
```

(continues on next page)

(continued from previous page)

```
mv firefox /opt/firefox46
ln -s /opt/firefox46/firefox /usr/bin/firefox
```

– Google Chrome

- To run the tests on a remote server, configure the remote X server. Use VNC Software to see the test results in real-time.

1. Specify the display environment variable:

```
$DISPLAY=: <value>
```

2. Configure remote X server and VNC Software by typing:

```
apt-get install xvfb xfonts-100dpi xfonts-75dpi xfonts-cyrillic xorg_
↪ dbus-x11
"Xvfb -fp "/usr/share/fonts/X11/misc/" :$DISPLAY -screen 0
↪ "1280x1024x16" &"
apt-get install --yes x11vnc
x11vnc -bg -forever -nopw -display :$DISPLAY -ncache 10
sudo iptables -I INPUT 1 -p tcp --dport 5900 -j ACCEPT
```

Download and run tests

To download and run the tests:

1. Verify that all additional components has been installed.
2. Clone the murano-dashboard git repository:

```
git clone https://opendev.org/openstack/murano-dashboard
```

3. Change the default settings:

1. Specify the Murano Repository URL variable for Horizon local settings in `murano_dashboard/muranodashboard/local/local_settings.d/_50_murano.py`:

```
MURANO_REPO_URL = 'http://localhost:8099'
```

2. Copy `muranodashboard/tests/functional/config/config.conf.sample` to `config.conf`.
3. Set appropriate URLs and credentials for your OpenStack lab. Only Administrator user credentials are appropriate.

```
[murano]

horizon_url = http://localhost/dashboard
murano_url = http://localhost:8082
user = ***
password = ***
```

(continues on next page)

(continued from previous page)

```
tenant = ***  
keystone_url = http://localhost:5000/v3
```

All tests are kept in `sanity_check.py` and divided into 10 test suites:

- `TestSuiteSmoke` - verification of Murano panels; checks that they can be open without errors.
- `TestSuiteEnvironment` - verification of all operations with environment are finished successfully.
- `TestSuiteImage` - verification of operations with images.
- `TestSuiteFields` - verification of custom fields validators.
- `TestSuitePackages` - verification of operations with Murano packages.
- `TestSuiteApplications` - verification of Application Catalog page and of application creation process.
- `TestSuiteAppsPagination` - verification of apps pagination in case of many applications installed.
- `TestSuiteRepository` - verification of importing packages and bundles.
- `TestSuitePackageCategory` - verification of main operations with categories.
- `TestSuiteCategoriesPagination` - verification of categories pagination in case of many categories created.
- `TestSuiteMultipleEnvironments` - verification of ability to apply action to multiple environments.

To run the tests follow these instructions:

- To run all tests:

```
nosetests sanity_check.py
```

- To run a single suite:

```
nosetests sanity_check.py:<test suite name>
```

- To run a single test:

```
nosetests sanity_check.py:<test suite name>.<test name>
```

In case of successful execution, you should see something like this:

```
.....  
Ran 34 tests in 1.440s  
OK
```

In case of failure, the folder with screenshots of the last operation of tests that finished with errors would be created. It is located in `murano/dashboard/tests/functional` folder.

There are also a number of command line options that can be used to control the test execution and generated outputs. For more details about `nosetests`, type:

```
nosetests -h
```

Tempest tests

All Murano services have tempest-based automated tests, which verify API interfaces and deployment scenarios. Tempest tests for Murano are located at <https://opendev.org/openstack/murano/src/branch/master/murano/tests/functional>.

The following Python files contain basic test suites for different Murano components.

API tests

Murano API tests are run on the devstack gate located at https://opendev.org/openstack/murano-tempest-plugin/src/branch/master/murano_tempest_tests/tests/api.

- `test_murano_envs.py` contains test suite with actions on murano environments (create, delete, get, and others).
- `test_murano_sessions.py` contains test suite with actions on murano sessions (create, delete, get, and others).
- `test_murano_services.py` contains test suite with actions on murano services (create, delete, get, and others).
- `test_murano_repository.py` contains test suite with actions on murano package repository.

Engine tests

Murano Engine Tests are run on murano-ci at https://opendev.org/openstack/murano-tempest-plugin/src/branch/master/murano_tempest_tests/tests/functional:

- `base.py` contains base test class and tests with actions on deploy Murano services such as Telnet and Apache.

Command-line interface tests

Murano CLI tests are currently in the middle of creation. The current scope is read-only operations on a cloud that are hard to test through unit tests. All tests have description and execution steps in their docstrings.

Guidelines

Development Guidelines

Coding Guidelines

For all the code in Murano we have a rule - it should pass [PEP 8](#).

To check your code against PEP 8 run:

```
tox -e pep8
```

See also:

- <https://pep8.readthedocs.org/en/latest/>
- <https://flake8.readthedocs.org>
- <https://docs.openstack.org/hacking/latest/>

Blueprints and Specs

Murano team uses the [murano-specs](#) repository for its blueprint and specification (specs) review process. See [Launchpad](#) to propose or see the status of a current blueprint.

Testing Guidelines

Murano has a suite of tests that are run on all submitted code, and it is recommended that developers execute the tests themselves to catch regressions early. Developers are also expected to keep the test suite up-to-date with any submitted code changes.

Unit tests are located at `murano/tests`.

Murano's suite of unit tests can be executed in an isolated environment with [Tox](#). To execute the unit tests run the following from the root of Murano repo on Python 3.x:

```
tox -e py3.x
```

Documentation Guidelines

Murano dev-docs are written using Sphinx / RST and located in the main repo in doc directory.

The documentation in docstrings should follow the [PEP 257](#) conventions (as mentioned in the [PEP 8](#) guidelines).

More specifically:

1. Triple quotes should be used for all docstrings.
2. If the docstring is simple and fits on one line, then just use one line.
3. For docstrings that take multiple lines, there should be a newline after the opening quotes, and before the closing quotes.
4. [Sphinx](#) is used to build documentation, so use the restructured text markup to designate parameters, return values, etc. Documentation on the sphinx specific markup can be found [here](#):

Run the following command to build docs locally.

```
tox -e docs
```

Gerrit review dashboard

Murano Gerrit Dashboard

Description

If you would like to contribute to murano by reviewing patches to murano-related projects you can use this gerrit dashboard, or create your own using [Gerrit Dash Creator](#)

URL

```
https://review.opendev.org/#/dashboard/?foreach=%28project%3A%5E.%2A%2F.
↪%2Amurano.%2A+OR+project%3Aopenstack%2Fyaql%29+NOT+label%3AWorkflow%3C%3D
↪%2D1+NOT+label%3ACode%2DReview%3C%3D%2D2+status%3Aopen&title=Murano&
↪My+Patches=owner%3Aself&You+are+a+reviewer%2C+but+haven
↪%27t+voted+in+the+current+revision=NOT+label%3ACode%2DReview%3C%3D2
↪%2Cself+reviewer%3Aself+NOT+owner%3Aself&Need+Feedback=NOT+label%3ACode
↪%2DReview%3C%3D2+NOT+label%3AVerified%3C%3D%2D1+NOT+owner%3Aself&
↪Passed+Jenkins%2C+No+Negative+Feedback=label%3ACode%2DReview%3E
↪%3D1+NOT+label%3ACode%2DReview%3C%3D%2D1+AND+NOT+label%3AVerified%3C%3D
↪%2D1+NOT+owner%3Aself+NOT+reviewer%3Aself+limit%3A50&Maybe+Review
↪%3F=NOT+owner%3Aself+NOT+reviewer%3Aself+limit%3A25&My+%2B1s=label%3ACode
↪%2DReview%3D1%2Cself+limit%3A25&Need+final+%2B2=label%3ACode%2DReview%3E
↪%3D2+NOT+label%3ACode%2DReview%3C%3D%2D1+NOT+label%3AVerified%3C%3D
↪%2D1+NOT+label%3ACode%2DReview%3C%3D%2D2%2Cself+NOT+owner%3Aself+limit%3A25&My+
↪%2B2s=label%3ACode%2DReview%3D2%2Cself+limit%3A25
```

[View this dashboard](#)

Configuration

```
[dashboard]
title = Murano
description = Murano Review Inbox
foreach = (project: ^.*/*.*murano.* OR project: openstack/yaql) NOT_
↪label: Workflow<=-1 NOT label: Code-Review<=-2 status: open

[section "My Patches"]
query = owner: self

[section "You are a reviewer, but haven't voted in the current revision"]
query = NOT label: Code-Review<=2, self reviewer: self NOT owner: self

[section "Need Feedback"]
query = NOT label: Code-Review<=2 NOT label: Verified<=-1 NOT owner: self

[section "Passed Jenkins, No Negative Feedback"]
query = label: Code-Review>=1 NOT label: Code-Review<=-1 AND NOT label: Verified
↪<=-1 NOT owner: self NOT reviewer: self limit: 50
```

(continues on next page)

(continued from previous page)

```
[section "Maybe Review?"]
query = NOT owner:self NOT reviewer:self limit:25

[section "My +1s"]
query = label:Code-Review=1,self limit:25

[section "Need final +2"]
query = label:Code-Review>=2 NOT label:Code-Review<=-1 NOT label:Verified<=-1
↳NOT label:Code-Review<=2,self NOT owner:self limit:25

[section "My +2s"]
query = label:Code-Review=2,self limit:25
```

API specification

Murano API v1 specification

General information

- **Introduction**

The murano service API is a programmatic interface used for interaction with murano. Other interaction mechanisms like the murano dashboard or the murano CLI should use the API as an underlying protocol for interaction.

- **Allowed HTTPs requests**

- *POST* : To create a resource
- *GET* : Get a resource or list of resources
- *DELETE* : To delete resource
- *PATCH* : To update a resource

- **Description Of Usual Server Responses**

- 200 OK - the request was successful.
- 201 Created - the request was successful and a resource was created.
- 204 No Content - the request was successful but there is no representation to return (i.e. the response is empty).
- 400 Bad Request - the request could not be understood or required parameters were missing.
- 401 Unauthorized - authentication failed or user didn't have permissions for requested operation.
- 403 Forbidden - access denied.
- 404 Not Found - resource was not found
- 405 Method Not Allowed - requested method is not supported for resource.
- 406 Not Acceptable - the requested resource is only capable of generating content not acceptable according to the Accept headers sent in the request.

– 409 Conflict - requested method resulted in a conflict with the current state of the resource.

- **Response of POSTs and PUTs**

All POST and PUT requests by convention should return the created object (in the case of POST, with a generated ID) as if it was requested by GET.

- **Authentication**

All requests include a keystone authentication token header (X-Auth-Token). Clients must authenticate with keystone before interacting with the murano service.

Murano API

Glossary

- **Environment**

The environment is a set of applications managed by a single project (tenant). They could be related logically with each other or not. Applications within a single environment may comprise of complex configuration while applications in different environments are always independent from one another. Each environment is associated with a single OpenStack project.

- **Session**

Since murano environments are available for local modification for different users and from different locations, it's needed to store local modifications somewhere. Sessions were created to provide this opportunity. After a user adds an application to the environment - a new session is created. After a user sends an environment to deploy, a session with a set of applications changes status to *deploying* and all other open sessions for that environment become invalid. One session could be deployed only once.

- **Object Model**

Applications are defined in MuranoPL object model, which is defined as a JSON object. The murano API doesn't know anything about it.

- **Package**

A .zip archive, containing instructions for an application deployment.

- **Environment-Template**

The environment template is the specification of a set of applications managed by a single project, which are related to each other. The environment template is stored in an environment template catalog, and it can be managed by the user (creation, deletion, updating). Finally, it can be deployed on OpenStack by translating into an environment.

Environment API

Attribute	Type	Description
id	string	Unique ID
name	string	User-friendly name
created	datetime	Creation date and time in ISO format
updated	datetime	Modification date and time in ISO format
tenant_id	string	OpenStack project ID
version	int	Current version
networking	string	Network settings
acquired_by	string	Id of a session that acquired this environment (for example is deploying it)
status	string	Deployment status: ready, pending, deploying

Common response codes

Code	Description
200	Operation completed successfully
403	User is not authorized to perform the operation

List environments

Request

Method	URI	Description
GET	/environments	Get a list of existing Environments

Parameters:

- *all_tenants* - boolean, indicates whether environments from all projects are listed. *True* environments from all projects are listed. Admin user required. *False* environments only from current project are listed (default like option unspecified).
- *tenant* - indicates environments from specified tenant are listed. Admin user required.

Response

This call returns a list of environments. Only the basic properties are returned.

```
{
  "environments": [
    {
      "status": "ready",
      "updated": "2014-05-14T13:02:54",
      "networking": {},
      "name": "test1",
      "created": "2014-05-14T13:02:46",
      "tenant_id": "726ed856965f43cc8e565bc991fa76c3",

```

(continues on next page)

(continued from previous page)

```

    "version": 0,
    "id": "2fa5ab704749444bbeafe7991b412c33"
  },
  {
    "status": "ready",
    "updated": "2014-05-14T13:02:55",
    "networking": {},
    "name": "test2",
    "created": "2014-05-14T13:02:51",
    "tenant_id": "726ed856965f43cc8e565bc991fa76c3",
    "version": 0,
    "id": "744e44812da84e858946f5d817de4f72"
  }
]
}

```

Create environment

Attribute	Type	Description
name	string	Environment name; at least one non-white space symbol

Request

Method	URI	Description
POST	/environments	Create new Environment

- **Content-Type** application/json
- **Example**
{"name": "env_name"}

Response

```

{
  "id": "ce373a477f211e187a55404a662f968",
  "name": "env_name",
  "created": "2013-11-30T03:23:42Z",
  "updated": "2013-11-30T03:23:44Z",
  "tenant_id": "0849006f7ce94961b3aab4e46d6f229a",
  "version": 0
}

```

Update environment

Attribute	Type	Description
name	string	Environment name; at least one non-white space symbol

Request

Method	URI	Description
PUT	/environments/<env_id>	Update an existing Environment

- **Content-Type** application/json
- **Example** {"name": "env_name_changed"}

Response

Content-Type
application/json

```
{
  "id": "ce373a477f211e187a55404a662f968",
  "name": "env_name_changed",
  "created": "2013-11-30T03:23:42Z",
  "updated": "2013-11-30T03:45:54Z",
  "tenant_id": "0849006f7ce94961b3aab4e46d6f229a",
  "version": 0
}
```

Code	Description
200	Edited environment
400	Environment name must contain at least one non-white space symbol
403	User is not authorized to access environment
404	Environment not found
409	Environment with specified name already exists

Get environment details

Request

Return information about the environment itself and about applications, including this environment.

Metho	URI	Header	Description
GET	/environ-ments/{id}	X-Configuration-Session (optional)	Response detailed information about Environment including child entities

Response

Content-Type

application/json

```
{
  "status": "ready",
  "updated": "2014-05-14T13:12:26",
  "networking": {},
  "name": "quick-env-2",
  "created": "2014-05-14T13:09:55",
  "tenant_id": "726ed856965f43cc8e565bc991fa76c3",
  "version": 1,
  "services": [
    {
      "instance": {
        "flavor": "m1.medium",
        "image": "cloud-fedora-v3",
        "name": "exgchhv6nbika2",
        "ipAddresses": [
          "10.0.0.200"
        ],
        "?": {
          "type": "io.murano.resources.Instance",
          "id": "14cce9d9-aaa1-4f09-84a9-c4bb859edaff"
        }
      },
      "name": "rewt4w56",
      "?": {
        "status": "ready",
        "_26411a1861294160833743e45d0eaad9": {
          "name": "Telnet"
        },
        "type": "io.murano.apps.linux.Telnet",
        "id": "446373ef-03b5-4925-b095-6c56568fa518"
      }
    }
  ],
  "id": "20d4a012628e4073b48490a336a8acbfb"
}
```

Delete environment

Request

Method	URI	Description
DELETE	/environments/{id}?abandon	Remove specified Environment.

Parameters:

- abandon* - boolean, indicates how to delete environment. *False* is used if all resources used by

environment must be destroyed; *True* is used when just database must be cleaned

Response

Code	Description
200	OK. Environment deleted successfully
403	User is not allowed to delete this resource
404	Not found. Specified environment doesn't exist

Environment configuration API

Multiple *sessions* could be opened for one environment simultaneously, but only one session going to be deployed. First session that starts deploying is going to be deployed; other ones become invalid and could not be deployed at all. User could not open new session for environment that in *deploying* state (that's why we call it "almost lock free" model).

Attribute	Type	Description
id	string	Session unique ID
environment_id	string	Environment that going to be modified during this session
created	datetime	Creation date and time in ISO format
updated	datetime	Modification date and time in ISO format
user_id	string	Session owner ID
version	int	Environment version for which configuration session is opened
state	string	Session state. Could be: open, deploying, deployed

Configure environment / open session

During this call a new working session is created with its ID returned in response body. Notice that the session ID should be added to request headers with name *X-Configuration-Session* in subsequent requests when necessary.

Request

Method	URI	Description
POST	/environments/<env_id>/configure	Creating new configuration session

Response

Content-Type

application/json

```
{
  "id": "257bef44a9d848daa5b2563779714820",
  "updated": datetime.datetime(2014, 5, 14, 14, 17, 58, 949358),
  "environment_id": "744e44812da84e858946f5d817de4f72",
  "ser_id": "4e91d06270c54290b9dbdf859356d3b3",
```

(continues on next page)

(continued from previous page)

```

"created": datetime.datetime(2014, 5, 14, 14, 17, 58, 949305),
"state": "open",
"version": 0L
}

```

Code	Description
200	Session created successfully
401	User is not authorized to access this session
403	Could not open session for environment, environment has deploying status
404	Not found. Specified environment doesn't exist

Deploy session

With this request all local changes made within the environment start to deploy on OpenStack.

Request

Method	URI	Description
POST	/environments/<env_id>/sessions/<session_id>/deploy	Deploy changes made in session with specified session_id

Response

Code	Description
200	Session status changes to <i>deploying</i>
401	User is not authorized to access this session
403	Session is already deployed or deployment is in progress
404	Not found. Specified session or environment doesn't exist

Get session details

Request

Method	URI	Description
GET	/environments/<env_id>/sessions/<session_id>	Get details about session with specified session_id

Response

```
{
  "id": "4aecdc2178b9430cbbb8db44fb7ac384",
  "environment_id": "4dc8a2e8986fa8fa5bf24dc8a2e8986fa8",
  "created": "2013-11-30T03:23:42Z",
  "updated": "2013-11-30T03:23:54Z",
  "user_id": "d7b501094caf4daab08469663a9e1a2b",
  "version": 0,
  "state": "deploying"
}
```

Code	Description
200	Session details information received
401	User is not authorized to access this session
403	Session is invalid
404	Not found. Specified session or environment doesn't exist

Delete session

Request

Method	URI	Description
DELETE	/environments/<env_id>/sessions/ session_id>	<ses- Delete session with specified ses- sion_id

Response

Code	Description
200	Session is deleted successfully
401	User is not authorized to access this session
403	Session is in deploying state and could not be deleted
404	Not found. Specified session or environment doesn't exist

Environment model API

Get environment model

Method	URI	Header	Description
GET	/environ- ments/<env_id>/model/<path>	X-Configuration-Session (op- tional)	Get an Environment model

Specifying <path> allows to get a specific section of the model, for example *defaultNetworks*, *region* or *?* or any of the subsections.

*Response***Content-Type**

application/json

```

{
  "defaultNetworks": {
    "environment": {
      "internalNetworkName": "net_two",
      "?": {
        "type": "io.murano.resources.ExistingNeutronNetwork",
        "id": "594e94fcfe4c48ef8f9b55edb3b9f177"
      }
    },
    "flat": null
  },
  "region": "RegionTwo",
  "name": "new_env",
  "regions": {
    "": {
      "defaultNetworks": {
        "environment": {
          "autoUplink": true,
          "name": "new_env-network",
          "externalRouterId": null,
          "dnsNameservers": [],
          "autogenerateSubnet": true,
          "subnetCidr": null,
          "openstackId": null,
          "?": {
            "dependencies": {
              "onDestruction": [{
                "subscriber":
↪ "c80e33dd67a44f489b2f04818b72f404",
                "handler": null
              }]
            },
            "type": "io.murano.resources.NeutronNetwork/0.0.0@io.
↪ murano",
            "id": "e145b50623c04a68956e3e656a0568d3",
            "name": null
          },
          "regionName": "RegionOne"
        },
        "flat": null
      },
      "name": "RegionOne",
      "?": {
        "type": "io.murano.CloudRegion/0.0.0@io.murano",
        "id": "c80e33dd67a44f489b2f04818b72f404",
        "name": null
      }
    }
  }
}

```

(continues on next page)

(continued from previous page)

```

    }
  },
  "RegionOne": "c80e33dd67a44f489b2f04818b72f404",
  "RegionTwo": {
    "defaultNetworks": {
      "environment": {
        "autoUplink": true,
        "name": "new_env-network",
        "externalRouterId": "e449bdd5-228c-4747-a925-18cda80fbd6b
↪",
        "dnsNameservers": ["8.8.8.8"],
        "autogenerateSubnet": true,
        "subnetCidr": "10.0.198.0/24",
        "openstackId": "00a695c1-60ff-42ec-acb9-b916165413da",
        "?": {
          "dependencies": {
            "onDestruction": [{
              "subscriber":
↪"f8cb28d147914850978edb35eca156e1",
              "handler": null
            }
          ]
        },
        "type": "io.murano.resources.NeutronNetwork/0.0.0@io.
↪murano",
        "id": "72d2c13c600247c98e09e2e3c1cd9d70",
        "name": null
      },
      "regionName": "RegionTwo"
    },
    "flat": null
  },
  "name": "RegionTwo",
  "?": {
    "type": "io.murano.CloudRegion/0.0.0@io.murano",
    "id": "f8cb28d147914850978edb35eca156e1",
    "name": null
  }
}
},
services: []
"?": {
  "type": "io.murano.Environment/0.0.0@io.murano",
  "_actions": {
    "f7f22c174070455c9cafc59391402bdc_deploy": {
      "enabled": true,
      "name": "deploy",
      "title": "deploy"
    }
  }
},

```

(continues on next page)

(continued from previous page)

```

    "id": "f7f22c174070455c9cafc59391402bdc",
    "name": null
  }
}

```

Code	Description
200	Environment model received successfully
403	User is not authorized to access environment
404	Environment is not found or specified section of the model does not exist

Update environment model

Request

Method	URI	Header	Description
PATCH	/environments/<env_id>/model/	X-Configuration-Session	Update an Environment model

- **Content-Type** application/env-model-json-patch

Allowed operations for paths:

- "" (model root): no operations
- "defaultNetworks": "replace"
- "defaultNetworks/environment": "replace"
- "defaultNetworks/environment/?/id": no operations
- "defaultNetworks/flat": "replace"
- "name": "replace"
- "region": "replace"
- "?/type": "replace"
- "?/id": no operations

For other paths any operation (add, replace or remove) is allowed.

- **Example of request body with JSON-patch**

```

[ {
  "op": "replace",
  "path": "/defaultNetworks/flat",
  "value": true
} ]

```

Response

Content-Type

application/json

See *GET* request response.

Code	Description
200	Environment is edited successfully
400	Body format is invalid
403	User is not authorized to access environment or specified operation is forbidden for the given property
404	Environment is not found or specified section of the model does not exist

Environment deployments API

Environment deployment API allows to track changes of environment status, deployment events and errors. It also allows to browse deployment history.

List Deployments

Returns information about all deployments of the specified environment.

Request

Method	URI	Description
GET	/environments/<env_id>/deployments	Get list of environment deployments
GET	/deployments	Get list of deployments for all environments in user's project

*Response***Content-Type**

application/json

```
{
  "deployments": [
    {
      "updated": "2014-05-15T07:24:21",
      "environment_id": "744e44812da84e858946f5d817de4f72",
      "description": {
        "services": [
          {
            "instance": {
              "flavor": "m1.medium",
              "image": "cloud-fedora-v3",
              "?": {
                "type": "io.murano.resources.Instance",
                "id": "ef729199-c71e-4a4c-a314-0340e279add8"
              }
            }
          }
        ]
      }
    }
  ]
}
```

(continues on next page)

(continued from previous page)

```

        },
        "name": "xkaduhv7qeg4m7"
    },
    "name": "teslnet1",
    "?": {
        "_26411a1861294160833743e45d0eaad9": {
            "name": "Telnet"
        },
        "type": "io.murano.apps.linux.Telnet",
        "id": "6e437be2-b5bc-4263-8814-6fd57d6ddb5"
    }
},
],
"defaultNetworks": {
    "environment": {
        "name": "test2-network",
        "?": {
            "type": "io.murano.lib.networks.neutron.NewNetwork
↔",
            "id": "b6a1d515434047d5b4678a803646d556"
        }
    },
    "flat": null
},
"name": "test2",
"?": {
    "type": "io.murano.Environment",
    "id": "744e44812da84e858946f5d817de4f72"
}
},
"created": "2014-05-15T07:24:21",
"started": "2014-05-15T07:24:21",
"finished": null,
"state": "running",
"id": "327c81e0e34a4c93ad9b9052ef42b752"
}
]
}

```

Code	Description
200	Deployments information received successfully
401	User is not authorized to access this environment

Application management API

All applications should be created within an environment and all environment modifications are held within the session. Local changes apply only after successful deployment of an environment session.

Get application details

Using GET requests to applications endpoint user works with list containing all applications for specified environment. A user can request a whole list, specific application, or specific attribute of a specific application using tree traversing. To request a specific application, the user should add to endpoint part an application id, e.g.: `/environments/<env_id>/services/<application_id>`. For selection of specific attribute on application, simply appending part with attribute name will work. For example to request application name, user should use next endpoint: `/environments/<env_id>/services/<application_id>/name`

Request

Method	URI	Header
GET	<code>/environments/<env_id>/services/<app_id></code>	X-Configuration-Session (optional)

Parameters:

- `env_id` - environment ID, required
- `app_id` - application ID, optional

Response

Content-Type

application/json

```
{
  "instance": {
    "flavor": "m1.medium",
    "image": "cloud-fedora-v3",
    "?": {
      "type": "io.murano.resources.Instance",
      "id": "060715ff-7908-4982-904b-3b2077ff55ef"
    },
    "name": "hbhmyhv6qihln3"
  },
  "name": "dfg34",
  "?": {
    "status": "pending",
    "_26411a1861294160833743e45d0eaad9": {
      "name": "Telnet"
    },
    "type": "io.murano.apps.linux.Telnet",
    "id": "6e7b8ad5-888d-4c5a-a498-076d092a7eff"
  }
}
```

Create new application

Create a new application and add it to the murano environment. Result JSON is calculated in Murano dashboard, which is based on UI definition.

Request

Content-Type

application/json

Method	URI	Header
POST	/environments/<env_id>/services	X-Configuration-Session

```
{
  "instance": {
    "flavor": "m1.medium",
    "image": "clod-fedora-v3",
    "?": {
      "type": "io.murano.resources.Instance",
      "id": "bce8308e-5938-408b-a27a-0d3f0a2c52eb"
    },
    "name": "nhekhv6r7mhd4"
  },
  "name": "sdf34sadf",
  "?": {
    "_26411a1861294160833743e45d0eaad9": {
      "name": "Telnet"
    },
    "type": "io.murano.apps.linux.Telnet",
    "id": "190c8705-5784-4782-83d7-0ab55a1449aa"
  }
}
```

Response

Created application returned

Content-Type

application/json

```
{
  "instance": {
    "flavor": "m1.medium",
    "image": "cloud-fedora-v3",
    "?": {
      "type": "io.murano.resources.Instance",
      "id": "bce8308e-5938-408b-a27a-0d3f0a2c52eb"
    },
    "name": "nhekhv6r7mhd4"
  },
  "name": "sdf34sadf",
```

(continues on next page)

(continued from previous page)

```

"?: {
  "_26411a1861294160833743e45d0eaa9": {
    "name": "Telnet"
  },
  "type": "io.murano.apps.linux.Telnet",
  "id": "190c8705-5784-4782-83d7-0ab55a1449a1"
}
}

```

Code	Description
200	Application was created successfully
401	User is not authorized to perform this action
403	Policy prevents this user from performing this action
404	Not found. Environment doesn't exist
400	Required header or body are not provided

Update applications

Applications list for environment can be updated.

Request

Content-Type

application/json

Method	URI	Header
PUT	/environments/<env_id>/services	X-Configuration-Session

```

[ {
  "instance": {
    "availabilityZone": "nova",
    "name": "apache-instance",
    "assignFloatingIp": true,
    "keyname": "",
    "flavor": "m1.small",
    "image": "146d5523-7b2d-4abc-b0d0-2041f920ce26",
    "?: {
      "type": "io.murano.resources.LinuxMuranoInstance",
      "id": "25185cb6f29b415fa2e438309827a797"
    }
  },
  "name": "ApacheHttpServer",
  "enablePHP": true,
  "?: {
    "type": "com.example.apache.ApacheHttpServer",
    "id": "6e66106d7dcb4748a5c570150a3df80f"
  }
}

```

(continues on next page)

(continued from previous page)

```
}
}]
```

Response

Updated applications list returned

Content-Type

application/json

```
[{
  "instance": {
    "availabilityZone": "nova",
    "name": "apache-instance",
    "assignFloatingIp": true,
    "keyname": "",
    "flavor": "m1.small",
    "image": "146d5523-7b2d-4abc-b0d0-2041f920ce26",
    "?": {
      "type": "io.murano.resources.LinuxMuranoInstance",
      "id": "25185cb6f29b415fa2e438309827a797"
    }
  },
  "name": "ApacheHttpServer",
  "enablePHP": true,
  "?": {
    "type": "com.example.apache.ApacheHttpServer",
    "id": "6e66106d7dcb4748a5c570150a3df80f"
  }
}]
```

Code	Description
200	Services are updated successfully
400	Required header is not provided
401	User is not authorized
403	Session is in deploying state and could not be updated or user is not allowed to update services
404	Not found. Specified environment and/or session do not exist

Delete application from environment

Delete one or all applications from the environment

Request

Method	URI	Header
DELETE	/environments/<env_id>/services/<app_id>	X-Configuration-Session(optional)

Parameters:

- *env_id* - environment ID, required
- *app_id* - application ID, optional

Statistic API

Statistic API intends to provide billing feature

Instance environment statistics

Request

Get information about all deployed instances in the specified environment

Method	URI
GET	/environments/<env_id>/instance-statistics/raw/<instance_id>

Parameters:

- *env_id* - environment ID, required
- *instance_id* - ID of the instance for which need to provide statistic information, optional

Response

Attribute	Type	Description
type	int	Code of the statistic object; 200 - instance, 100 - application
type_name	string	Class name of the statistic object
instance_id	string	Id of deployed instance
active	bool	Instance status
type_title	string	User-friendly name for browsing statistic in UI
duration	int	Seconds of instance uptime

Content-Type

application/json

```
[
  {
    "type": 200,
    "type_name": "io.murano.resources.Instance",
    "instance_id": "ef729199-c71e-4a4c-a314-0340e279add8",
    "active": true,
    "type_title": null,
    "duration": 1053,
  }
]
```

Request

Method	URI
GET	/environments/<env_id>/instance-statistics/aggregated

Response

Attribute	Type	Description
type	int	Code of the statistic object; 200 - instance, 100 - application
duration	int	Amount uptime of specified type objects
count	int	Quantity of specified type objects

Content-Type

application/json

```
[
  {
    "duration": 720,
    "count": 2,
    "type": 200
  }
]
```

General Request Statistics

Request

Method	URI
GET	/stats

Response

Attribute	Type	Description
requests_per_tenant	int	Number of incoming requests for user project
errors_per_second	int	Class name of the statistic object
errors_count	int	Class name of the statistic object
requests_per_second	float	Average number of incoming request received in one second
requests_count	int	Number of all requests sent to the server
cpu_percent	bool	Current cpu usage
cpu_count	int	Available cpu power is $cpu_count * 100\%$
host	string	Server host-name
average_response_time	float	Average time response waiting, seconds

Content-Type

application/json

```
[
  {
    "updated": "2014-05-15T08:26:17",
    "requests_per_tenant": "{\"726ed856965f43cc8e565bc991fa76c3\": 313}",
    "created": "2014-04-29T13:23:59",
    "cpu_count": 2,
    "errors_per_second": 0,
    "requests_per_second": 0.0266528,
    "cpu_percent": 21.7,
    "host": "fervent-VirtualBox",
    "error_count": 0,
    "request_count": 320,
    "id": 1,
    "average_response_time": 0.55942
  }
]
```

Actions API

Murano actions are simple MuranoPL methods, that can be called on deployed applications. Application contains a list with available actions. Actions may return a result.

Execute an action

Generate task with executing specified action. Input parameters may be provided.

Request

Content-Type

application/json

Method	URI	Header
POST	/environments/<env_id>/actions/<action_id>	

Parameters:

- *env_id* - environment ID, required
- *actions_id* - action ID to execute, required

```
"{<action_property>: value}"
```

or

```
"{}" in case action has no properties
```

Response

Task ID that executes specified action is returned

Content-Type

application/json

```
{
  "task_id": "620e883070ad40a3af566d465aa156ef"
}
```

GET action result

Request result value after action execution finish. Not all actions have return values.

Request

Method	URI	Header
GET	/environments/<env_id>/actions/<task_id>	

Parameters:

- *env_id* - environment ID, required
- *task_id* - task ID, generated on desired action execution

Response

Json, describing action result is returned. Result type and value are provided.

Content-Type

application/json

```
{
  "isException": false,
  "result": ["item1", "item2"]
}
```

Static Actions API

Static actions are MuranoPL methods that can be called on a MuranoPL class without deploying actual applications and usually return a result.

Execute a static action

Invoke public static method of the specified MuranoPL class. Input parameters may be provided if method requires them.

Request

Content-Type

application/json

Method	URI	Header
POST	/actions	

```
{
  "className": "my.class.fqn",
  "methodName": "myMethod",
  "packageName": "optional.package.fqn",
  "classVersion": "1.2.3",
  "parameters": {
    "arg1": "value1",
    "arg2": "value2"
  }
}
```

Attribute	Type	Description
className	string	Fully qualified name of MuranoPL class with static method
methodName	string	Name of the method to invoke
package-Name	string	Fully qualified name of a package with the MuranoPL class (optional)
classVersion	string	Class version specification, "=0" by default
parameters	object	Key-value pairs of method parameter names and their values, "{}" by default

Response

JSON-serialized result of the static method execution.

HTTP codes:

Code	Description
200	OK. Action was executed successfully
400	Bad request. The format of the body is invalid, method doesn't match provided arguments, mandatory arguments are not provided
403	User is not allowed to execute the action
404	Not found. Specified class, package or method doesn't exist or method is not exposed
503	Unhandled exception in the action

Application catalog API

Manage application definitions in the Application Catalog. You can browse, edit and upload new application packages (.zip.package archive with all data that required for a service deployment).

Packages

Methods for application package management

Package Properties

- **id**: guid of a package (`fully_qualified_name` can also be used for some API functions)
- **fully_qualified_name**: fully qualified domain name - domain name that specifies exact application location
- **name**: user-friendly name
- **type**: package type, "library" or "application"
- **description**: text information about application
- **author**: name of application author
- **tags**: list of short names, connected with the package, which allows to search applications easily
- **categories**: list of application categories
- **class_definition**: list of class names used by a package
- **is_public**: determines whether the package is shared for other projects
- **enabled**: determines whether the package is browsed in the Application Catalog
- **owner_id**: id of a project that owns the package

Note: It is possible to use `in` operator for properties `id`, `category` and `tag`. For example to get packages with `id1`, `id2`, `id3` use `id=in:id1,id2,id3`.

List packages

```
/v1/catalog/packages?{marker}{limit}{order_by}{type}{category}{fqn}{owned}{id}{catalog}{class_name}{name}
[GET]
```

This is the compound request to list and search through application catalog. If there are no search parameters all packages that `is_public`, `enabled` and belong to the user's project will be listed. Default order is by 'created' field. For an admin role all packages are available.

Parameters

Attribute	Type	Description
<code>catalog</code>	bool	If false (default) - search packages, that current user can edit (own for non-admin, all for admin) If true - search packages, that current user can deploy (i.e. his own + public)
<code>marker</code>	string	A package identifier marker may be specified. When present only packages which occur after the identifier ID will be listed
<code>limit</code>	string	When present the maximum number of results returned will not exceed the specified value. The typical pattern of limit and marker is to make an initial limited request and then to use the ID of the last package from the response as the marker parameter in a subsequent limited request.
<code>order_type</code>	string	Allows to sort packages by: <i>fqn, name, created</i> . Created is default value.
<code>type</code>	string	Allows to point a type of package: <i>application, library</i>
<code>category</code>	string	Allows to point a categories for a search
<code>fqn</code>	string	Allows to point a fully qualified package name for a search
<code>owned</code>	bool	Search only from packages owned by current project
<code>id</code>	string	Allows to point an id for a search
<code>include</code>	bool	Include disabled packages in a the result
<code>search</code>	string	Gives opportunity to search specified data by all the package parameters and order packages
<code>class_name</code>	string	Search only for packages, that use specified class
<code>name</code>	string	Allows to point a package name for a search

Response 200 (application/json)

```
{
  "packages": [
    {
      "id": "fed57567c9fa42c192dcbe0566f8ea33",
      "fully_qualified_name": "com.example.murano.services.linux.
      ↪telnet",
      "is_public": false,
      "name": "Telnet",
      "type": "linux",
      "description": "Installs Telnet service",
      "author": "OpenStack, Inc.",
      "created": "2014-04-02T14:31:55",
      "enabled": true,
      "tags": ["linux", "telnet"],
      "categories": ["Utility"],
      "owner_id": "fed57567c9fa42c192dcbe0566f8ea40"
    },
    {
      "id": "fed57567c9fa42c192dcbe0566f8ea31",
      "fully_qualified_name": "com.example.murano.services.windows.
      ↪WebServer",
      "is_public": true,
      "name": "Internet Information Services",
      "type": "windows",
      "description": "The Internet Information Service sets up an
      ↪IIS server and joins it into an existing domain",

```

(continues on next page)

(continued from previous page)

```

        "author": "OpenStack, Inc.",
        "created": "2014-04-02T14:31:55",
        "enabled": true,
        "tags": ["windows", "web"],
        "categories": ["Web"],
        "owner_id": "fed57567c9fa42c192dcbe0566f8ea40"
    }
}

```

Upload a new package[POST]

/v1/catalog/packages

See the example of multipart/form-data request, It should contain two parts - text (JSON string) and file object

Request (multipart/form-data)

```

Content-type: multipart/form-data, boundary=AaB03x
Content-Length: $requestlen

--AaB03x
content-disposition: form-data; name="submit-name"

--AaB03x
Content-Disposition: form-data; name="JsonString"
Content-Type: application/json

{"categories":["web"] , "tags": ["windows"], "is_public": false, "enabled":
↪false}
`categories` - array, required
`tags` - array, optional
`name` - string, optional
`description` - string, optional
`is_public` - bool, optional
`enabled` - bool, optional

--AaB03x
content-disposition: file; name="file"; filename="test.tar"
Content-Type: targz
Content-Transfer-Encoding: binary

$binarydata
--AaB03x--

```

Response 200 (application/json)

```

{
  "updated": "2014-04-03T13:00:13",

```

(continues on next page)

(continued from previous page)

```

    "description": "A domain service hosted in Windows environment by using
↪Active Directory Role",
    "tags": ["windows"],
    "is_public": true,
    "id": "8f4f09bd6bcb47fb968afd29aacc0dc9",
    "categories": ["test1"],
    "name": "Active Directory",
    "author": "Mirantis, Inc",
    "created": "2014-04-03T13:00:13",
    "enabled": true,
    "class_definition": [
        "com.mirantis.murano.windows.activeDirectory.ActiveDirectory",
        "com.mirantis.murano.windows.activeDirectory.SecondaryController",
        "com.mirantis.murano.windows.activeDirectory.Controller",
        "com.mirantis.murano.windows.activeDirectory.PrimaryController"
    ],
    "fully_qualified_name": "com.mirantis.murano.windows.activeDirectory.
↪ActiveDirectory",
    "type": "Application",
    "owner_id": "fed57567c9fa42c192dcbe0566f8ea40"
}

```

Get package details

/v1/catalog/packages/{id} [GET]

Display details for a package.

Parameters

id (required) Hexadecimal *id* (or fully qualified name) of the package

Response 200 (application/json)

```

{
    "updated": "2014-04-03T13:00:13",
    "description": "A domain service hosted in Windows environment by using
↪Active Directory Role",
    "tags": ["windows"],
    "is_public": true,
    "id": "8f4f09bd6bcb47fb968afd29aacc0dc9",
    "categories": ["test1"],
    "name": "Active Directory",
    "author": "Mirantis, Inc",
    "created": "2014-04-03T13:00:13",
    "enabled": true,
    "class_definition": [
        "com.mirantis.murano.windows.activeDirectory.ActiveDirectory",
        "com.mirantis.murano.windows.activeDirectory.SecondaryController",
        "com.mirantis.murano.windows.activeDirectory.Controller",

```

(continues on next page)

(continued from previous page)

```

    "com.mirantis.murano.windows.activeDirectory.PrimaryController"
  ],
  "fully_qualified_name": "com.mirantis.murano.windows.activeDirectory.
↔ActiveDirectory",
  "type": "Application",
  "owner_id": "fed57567c9fa42c192dcbe0566f8ea40"
}

```

Response 403

- In attempt to get a non-public package by a user whose project is not an owner of this package.

Response 404

- In case the specified package id doesn't exist.

Update a package

/v1/catalog/packages/{id} [PATCH]

Allows to edit mutable fields (categories, tags, name, description, is_public, enabled). See the full specification [here](#).

Parameters

id (required) Hexadecimal *id* (or fully qualified name) of the package

Content type

application/murano-packages-json-patch

Allowed operations:

```

[
  { "op": "add", "path": "/tags", "value": [ "foo", "bar" ] },
  { "op": "add", "path": "/categories", "value": [ "foo", "bar" ] },
  { "op": "remove", "path": "/tags", ["foo"] },
  { "op": "remove", "path": "/categories", ["foo"] },
  { "op": "replace", "path": "/tags", "value": [] },
  { "op": "replace", "path": "/categories", "value": ["bar"] },
  { "op": "replace", "path": "/is_public", "value": true },
  { "op": "replace", "path": "/enabled", "value": true },
  { "op": "replace", "path": "/description", "value": "New description" },
  { "op": "replace", "path": "/name", "value": "New name" }
]

```

Request 200 (application/murano-packages-json-patch)

```

[
  { "op": "add", "path": "/tags", "value": [ "windows", "directory" ] },
  { "op": "add", "path": "/categories", "value": [ "Directory" ] }
]

```

Response 200 (application/json)

```
{
  "updated": "2014-04-03T13:00:13",
  "description": "A domain service hosted in Windows environment by using
↳Active Directory Role",
  "tags": ["windows", "directory"],
  "is_public": true,
  "id": "8f4f09bd6bcb47fb968afd29aacc0dc9",
  "categories": ["test1"],
  "name": "Active Directory",
  "author": "Mirantis, Inc",
  "created": "2014-04-03T13:00:13",
  "enabled": true,
  "class_definition": [
    "com.mirantis.murano.windows.activeDirectory.ActiveDirectory",
    "com.mirantis.murano.windows.activeDirectory.SecondaryController",
    "com.mirantis.murano.windows.activeDirectory.Controller",
    "com.mirantis.murano.windows.activeDirectory.PrimaryController"
  ],
  "fully_qualified_name": "com.mirantis.murano.windows.activeDirectory.
↳ActiveDirectory",
  "type": "Application",
  "owner_id": "fed57567c9fa42c192dcbe0566f8ea40"
}
```

Response 403

- An attempt to update immutable fields
- An attempt to perform operation that is not allowed on the specified path
- An attempt to update non-public package by user whose project is not an owner of this package

Response 404

- An attempt to update package that doesn't exist

Delete application definition from the catalog

/v1/catalog/packages/{id} [DELETE]

Parameters

- `id` (required) Hexadecimal *id* (or fully qualified name) of the package to delete

Response 404

- An attempt to delete package that doesn't exist

Get application package

/v1/catalog/packages/{id}/download [GET]

Get application definition package

Parameters

- `id` (required) Hexadecimal *id* (or fully qualified name) of the package

Response 200 (application/octet-stream)

The sequence of bytes representing package content

Response 404

Specified package id doesn't exist

Get UI definition

/v1/catalog/packages/{id}/ui [GET]

Retrieve UI definition for an application which described in a package with provided id

Parameters

- `id` (required) Hexadecimal *id* (or fully qualified name) of the package

Response 200 (application/octet-stream)

The sequence of bytes representing UI definition

Response 404

Specified package id doesn't exist

Response 403

Specified package is not public and not owned by user project, performing the request

Response 404

- Specified package id doesn't exist

Get logo

Retrieve application logo which described in a package with provided id

/v1/catalog/packages/{id}/logo [GET]

Parameters

`id` (required) Hexadecimal *id* (or fully qualified name) of the package

Response 200 (application/octet-stream)

The sequence of bytes representing application logo

Response 403

Specified package is not public and not owned by user project, performing the request

Response 404

Specified package is not public and not owned by user project, performing the request

Categories

Provides category management. Categories are used in the Application Catalog to group application for easy browsing and search.

List categories

- */v1/catalog/packages/categories [GET]*

!DEPRECATED (Plan to remove in L release) Retrieve list of all available application categories

Response 200 (application/json)

A list, containing category names

Content-Type

application/json

```
{
  "categories": ["Web service", "Directory", "Database", "Storage"]
}
```

- */v1/catalog/categories [GET]*

Method	URI	Description
GET	/catalog/categories	Get list of existing categories

Retrieve list of all available application categories

Response 200 (application/json)

A list, containing detailed information about each category

Content-Type

application/json

```
{
  "categories": [
    {
      "id": "0420045dce7445fabae7e5e61fff9e2f",
      "updated": "2014-12-26T13:57:04",
      "name": "Web",
      "created": "2014-12-26T13:57:04",
      "package_count": 1
    },
    {
      "id": "3dd486b1e26f40ac8f35416b63f52042",
      "updated": "2014-12-26T13:57:04",

```

(continues on next page)

(continued from previous page)

```

    "name": "Databases",
    "created": "2014-12-26T13:57:04",
    "package_count": 0
  }
}

```

Get category details

`/catalog/categories/<category_id>` [GET]

Return detailed information for a provided category

Request

Method	URI	Description
GET	<code>/catalog/categories/<category_id></code>	Get category detail

Parameters

- `category_id` - required, category ID, required

Response

Content-Type

application/json

```

{
  "id": "b308f7fa8a2f4a5eb419970c827f4466",
  "updated": "2015-01-28T17:00:19",
  "packages": [
    {
      "fully_qualified_name": "io.murano.apps.ZabbixServer",
      "id": "4dfb566e69e6445fbd4aea5099fe95e9",
      "name": "Zabbix Server"
    }
  ],
  "name": "Web",
  "created": "2015-01-28T17:00:19",
  "package_count": 1
}

```

Code	Description
200	OK. Category deleted successfully
401	User is not authorized to access this session
404	Not found. Specified category doesn't exist

Add new category

/catalog/categories [POST]

Add new category to the Application Catalog

Parameters

Attribute	Type	Description
name	string	Environment name; only alphanumeric characters and '-'

Request

Method	URI	Description
POST	/catalog/categories	Create new category

Content-Type

application/json

Example

```
{"name": "category_name"}
```

Response

```
{
  "id": "ce373a477f211e187a55404a662f968",
  "name": "category_name",
  "created": "2013-11-30T03:23:42Z",
  "updated": "2013-11-30T03:23:44Z",
  "package_count": 0
}
```

Code	Description
200	OK. Category created successfully
401	User is not authorized to access this session
409	Conflict. Category with specified name already exist

Delete category

/catalog/categories [DELETE]

Request

Method	URI	Description
DELETE	/catalog/categories/<category_id>	Delete category with specified ID

Parameters:

- `category_id` - required, category ID, required

Response

Code	Description
200	OK. Category deleted successfully
401	User is not authorized to access this session
404	Not found. Specified category doesn't exist
403	Forbidden. Category with specified name is assigned to the package, presented in the catalog

Environment template API

Manage environment template definitions in murano. It is possible to create, update, delete, and deploy into OpenStack by translating it into an environment. In addition, applications can be added to or deleted from the environment template.

Environment Template Properties

Attribute	Type	Description
<code>id</code>	string	Unique ID
<code>name</code>	string	User-friendly name
<code>created</code>	datetime	Creation date and time in ISO format
<code>updated</code>	datetime	Modification date and time in ISO format
<code>tenant_id</code>	string	OpenStack project
<code>version</code>	int	Current version
<code>networking</code>	string	Network settings
<code>description</code>	string	The environment template specification

Common response codes

Code	Description
200	Operation completed successfully
401	User is not authorized to perform the operation

Methods for Environment Template API

List Environments Templates

Request

Method	URI	Description
GET	<code>/templates</code>	Get a list of existing environment templates

Parameters:

- *is_public* - boolean, indicates whether public environment templates are listed or not. *True* public environments templates from all projects are listed. *False* private environments templates from current project are listed *empty* all project templates plus public templates from all projects are listed

Response

This call returns a list of environment templates. Only the basic properties are returned.

```
{
  "templates": [
    {
      "updated": "2014-05-14T13:02:54",
      "networking": {},
      "name": "test1",
      "created": "2014-05-14T13:02:46",
      "tenant_id": "726ed856965f43cc8e565bc991fa76c3",
      "version": 0,
      "is_public": false,
      "id": "2fa5ab704749444bbeafe7991b412c33"
    },
    {
      "updated": "2014-05-14T13:02:55",
      "networking": {},
      "name": "test2",
      "created": "2014-05-14T13:02:51",
      "tenant_id": "123452452345346345634563456345346",
      "version": 0,
      "is_public": true,
      "id": "744e44812da84e858946f5d817de4f72"
    }
  ]
}
```

Create environment template

Attribute	Type	Description
name	string and '-'	Environment template name; only alphanumeric characters

Request

Method	URI	Description
POST	/templates	Create a new environment template

Content-Type

application/json

Example

```
{"name": "env_temp_name"}
```

Response

```
{
  "id": "ce373a477f211e187a55404a662f968",
  "name": "env_temp_name",
  "created": "2013-11-30T03:23:42Z",
  "updated": "2013-11-30T03:23:44Z",
  "tenant_id": "0849006f7ce94961b3aab4e46d6f229a",
}
```

Code	Description
200	Operation completed successfully
401	User is not authorized to perform the operation
409	The environment template already exists

Get environment templates details

Request

Return information about environment template itself and about applications, including to this environment template.

Method	URI	Description
GET	/templates/{env-temp-id}	Obtains the environment template information

- *env-temp-id* - environment template ID, required

Response

Content-Type

application/json

```
{
  "updated": "2015-01-26T09:12:51",
  "networking":
  {
  },
  "name": "template_name",
  "created": "2015-01-26T09:12:51",
  "tenant_id": "00000000000000000000000000000001",
  "version": 0,
  "id": "aa9033ca7ce245fca10e38e1c8c4bbf7",
}
```

Code	Description
200	OK. Get environment template details successfully
401	User is not authorized to access this session
404	The environment template does not exist

Delete environment template

Request

Method	URI	Description
DELETE	/templates/<env-temp-id>	Delete the template id

Parameters:

- *env-temp_id* - environment template ID, required

Response

Code	Description
200	OK. Environment Template deleted successfully
401	User is not authorized to access this session
404	The environment template does not exist

Adding application to environment template

Request

Method	URI	Description
POST	/templates/{env-temp-id}/services	Create a new application

Parameters:

- *env-temp-id* - The environment-template id, required
- *payload* - the service description

Content-Type

application/json

Example

```
{
  "instance": {
    "assignFloatingIp": "true",
    "keyname": "mykeyname",
    "image": "cloud-fedora-v3",
    "flavor": "m1.medium",
    "?": {
      "type": "io.murano.resources.LinuxMuranoInstance",
      "id": "ef984a74-29a4-45c0-b1dc-2ab9f075732e"
    }
  },
  "name": "orion",
```

(continues on next page)

(continued from previous page)

```

"port": "8080",
"?: {
  "type": "io.murano.apps.apache.Tomcat",
  "id": "54cea43d-5970-4c73-b9ac-fea656f3c722"
}
}

```

Response

```

{
  "instance":
  {
    "assignFloatingIp": "true",
    "keyname": "mykeyname",
    "image": "cloud-fedora-v3",
    "flavor": "m1.medium",
    "?: {
      {
        "type": "io.murano.resources.LinuxMuranoInstance",
        "id": "ef984a74-29a4-45c0-b1dc-2ab9f075732e"
      }
    },
    "name": "orion",
    "?: {
      {
        "type": "io.murano.apps.apache.Tomcat",
        "id": "54cea43d-5970-4c73-b9ac-fea656f3c722"
      }
    },
    "port": "8080"
  }
}

```

Code	Description
200	OK. Application added successfully
401	User is not authorized to access this session
404	The environment template does not exist

Delete application from an environment template*Request*

Method	URI	Description
DELETE	/templates/{env-temp-id}/services/{app-id}	Delete application with Specified id

Parameters:

- *env-temp-id* - The environment template ID, required

- *app-id* - The application ID, required

Content-Type

application/json

Response

```
{
  "updated": "2015-01-26T09:12:51",
  "services": [],
  "name": "template_name",
  "created": "2015-01-26T09:12:51",
  "tenant_id": "00000000000000000000000000000001",
  "version": 0,
  "id": "aa9033ca7ce245fca10e38e1c8c4bbf7",
}
```

Code	Description
200	OK. Application deleted successfully
401	User is not authorized to access this session
404	The application does not exist

Get applications information from an environment template**Request**

Method	URI	Description
GET	/templates/{env-temp-id}/services	It obtains the service description

Parameters:

- *env-temp-id* - The environment template ID, required

Content-Type

application/json

Response

```
[
  {
    "instance":
    {
      "assignFloatingIp": "true",
      "keyname": "mykeyname",
      "image": "cloud-fedora-v3",
      "flavor": "m1.medium",
      "?":
      {
        "type": "io.murano.resources.LinuxMuranoInstance",

```

(continues on next page)

(continued from previous page)

```

        "id": "ef984a74-29a4-45c0-b1dc-2ab9f075732e"
    },
    "name": "tomcat",
    "?":
    {
        "type": "io.murano.apps.apache.Tomcat",
        "id": "54cea43d-5970-4c73-b9ac-fea656f3c722"
    },
    "port": "8080"
},
{
    "instance": "ef984a74-29a4-45c0-b1dc-2ab9f075732e",
    "password": "XXX",
    "name": "mysql",
    "?":
    {
        "type": "io.murano.apps.database.MySQL",
        "id": "54cea43d-5970-4c73-b9ac-fea656f3c722"
    }
}
]

```

Code	Description
200	OK. Application information received successfully
401	User is not authorized to access this session
404	The environment template does not exist

Update applications information from an environment template

Request

Method	URI	Description
PUT	/templates/{env-temp-id}/services/{service-id}	It updates the service description

Parameters:

- *env-temp-id* - The environment template ID, required
- *service-id* - The service ID to be updated
- *payload* - the service description

Content-Type

application/json

Example

```
{
  "instance": {
    "assignFloatingIp": "true",
    "keyname": "mykeyname",
    "image": "cloud-fedora-v3",
    "flavor": "m1.medium",
    "?": {
      "type": "io.murano.resources.LinuxMuranoInstance",
      "id": "ef984a74-29a4-45c0-b1dc-2ab9f075732e"
    }
  },
  "name": "orion",
  "port": "8080",
  "?": {
    "type": "io.murano.apps.apache.Tomcat",
    "id": "54cea43d-5970-4c73-b9ac-fea656f3c722"
  }
}
```

Response

```
{
  "instance":
  {
    "assignFloatingIp": "true",
    "keyname": "mykeyname",
    "image": "cloud-fedora-v3",
    "flavor": "m1.medium",
    "?":
    {
      "type": "io.murano.resources.LinuxMuranoInstance",
      "id": "ef984a74-29a4-45c0-b1dc-2ab9f075732e"
    }
  },
  "name": "orion",
  "?":
  {
    "type": "io.murano.apps.apache.Tomcat",
    "id": "54cea43d-5970-4c73-b9ac-fea656f3c722"
  },
  "port": "8080"
}
```

Code	Description
200	OK. Environment Template updated successfully
401	User is not authorized to access this session
404	The environment template does not exist

Create an environment from an environment template

Request

Method	URI	Description
POST	/templates/{env-temp-id}/create-environment	Create an environment

Parameters:

- *env-temp-id* - The environment template ID, required

Payload:

- 'environment name': The environment name to be created.

Content-Type

application/json

Example

```
{
  "name": "environment_name"
}
```

Response

```
{
  "environment_id": "aa90fadfafca10e38e1c8c4bbf7",
  "name": "environment_name",
  "created": "2015-01-26T09:12:51",
  "tenant_id": "00000000000000000000000000000001",
  "version": 0,
  "session_id": "adf4dadfaa9033ca7ce245fca10e38e1c8c4bbf7",
}
```

Code	Description
200	OK. Environment created from template successfully
401	User is not authorized to access this session
404	The environment template does not exist
409	The environment already exists

POST /templates/{env-temp-id}/clone

Request

Method	URI	Description
POST	/templates/{env-temp-id}/clone	It clones a public template from one project to another

Parameters:

- *env-temp-id* - environment template ID, required

Example Payload

```
{
  'name': 'cloned_env_template_name'
}
```

Content-Type

application/json

Response

```
{
  "updated": "2015-01-26T09:12:51",
  "name": "cloned_env_template_name",
  "created": "2015-01-26T09:12:51",
  "tenant_id": "00000000000000000000000000000001",
  "version": 0,
  "is_public": False,
  "id": "aa9033ca7ce245fca10e38e1c8c4bbf7",
}
```

Code	Description
200	OK. Environment Template cloned successfully
401	User is not authorized to access this session
403	User has no access to these resources
404	The environment template does not exist
409	Conflict. The environment template name already exists

USING MURANO

Learn how to use the Application Catalog directly from the Dashboard and through the command-line interface (CLI), manage applications and environments. The screenshots provided in this guide are of the Liberty release.

2.1 QuickStart

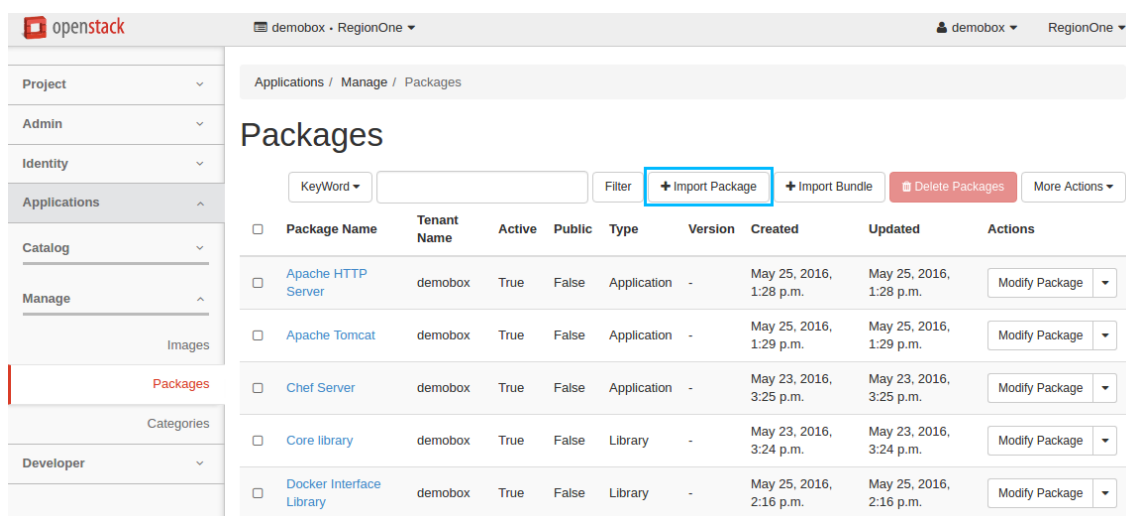
This is a brief walkthrough to quickly get you familiar with the basic operations you can perform when using the Application catalog directly from the dashboard.

For the detailed instructions on how to *manage your environments* and *applications*, please proceed with dedicated sections.

2.1.1 Upload an application

To upload an application to the catalog:

1. Log in to the OpenStack dashboard.
2. Navigate to *Applications > Manage > Packages*.
3. Click on the *Import Package* button:



4. In the *Import Package* dialog:

- Select URL from the Package Source drop-down list;

- Specify the URL in the *Package URL* field. Lets upload the Apache HTTP Server package using `http://storage.apps.openstack.org/apps/com.example.apache.ApacheHttpServer.zip`;
- Click *Next* to continue:

Import Package ✕

Package Source

URL

Package URL * ?

`http://storage.apps.openstack.org/apps/io.murano.:`

Description:

Package URL: HTTP/HTTPS URL of the package file.

Note: If the package depends upon other packages and/or requires specific glance images, those are going to be installed with it from murano repository.

Cancel Next

5. View the package details in the new dialog, click *Next* to continue:

Import Package ✕

Name

Apache HTTP Server

Tags ?

HTTP, Server, WebServer, HTML, Apache

Public

Active

Description

The Apache HTTP Server Project is an effort to develop and maintain an open-source HTTP server for modern operating systems including UNIX and Windows NT. The goal of this project is to provide a secure, efficient and extensible server that provides HTTP services in sync with the current HTTP standards.

Apache httpd has been the most popular web server on the Internet since

Description:

Name: Set up for identifying a package.

Tags: Used for identifying and filtering packages.

Public: Defines whether or not a package can be used by other tenants. (Applies to package dependencies)

Active: Allows to hide a package from the catalog. (Applies to package dependencies)

Description: Allows adding additional information about a package.

Cancel Next

6. Select the *Application Servers* from the application category list, click *Create* to import the application package:

Import Package x

Application Category

Application Servers
Key-Value Storage
SAP
Microsoft Services

Description:

Categories Select one or more categories for a package

Specifying a category helps to filter applications in the catalog

Cancel

Create

- Now your application is available from *Applications > Catalog > Browse* page.

2.1.2 Deploy an application

To add an application to an environment's component list and deploy the environment:

- Log in to the OpenStack dashboard.
- Navigate to *Applications > Catalog > Browse*.
- Click on the *Quick Deploy* button from the required application from the list. Lets deploy Apache HTTP Server, for example:

The screenshot shows the OpenStack dashboard interface. The top navigation bar includes the OpenStack logo, the user 'demobox', and the region 'RegionOne'. The left sidebar contains navigation menus for Project, Admin, Identity, Applications, Catalog, Environments, Manage, and Developer. The main content area is titled 'Browse' and shows 'Recent Activity' with a list of applications. The first application, 'Apache HTTP Server', is highlighted with a blue border, and its 'Quick Deploy' button is also circled in blue. Below the application list, there are filters for 'App Category' (set to 'All') and 'Environment' (set to 'quick-env-2'). There are also search and filter buttons.

- Check *Assign Floating IP* and click *Next* to proceed:

Configure Application: Apache HTTP Server
✕

Application Name *

Enable PHP

Assign Floating IP

Apache HTTP Server

Apache License, Version 2.0

Application Name: Enter a desired name for the application. Just A-Z, a-z, 0-9, dash and underline are allowed

Enable PHP: Add php support to the Apache WebServer

Assign Floating IP: Select to true to assign floating IP automatically

Next

5. Select the *Instance Image* from the drop-down list and click *Create*:

Configure Application: Apache HTTP Server
✕

Instance flavor

Instance image *

Key Pair

Availability zone

Instance Naming Pattern ?

Apache HTTP Server

Specify some instance parameters on which the application would be created

Instance flavor: Select registered in Openstack flavor. Consider that application performance depends on this parameter.

Instance image: Select valid image for the application. Image should already be prepared and registered in glance.

Key Pair: Select the Key Pair to control access to instances. You can login to instances using this KeyPair after the deployment of application.

Availability zone: Select availability zone where application would be installed.

Instance Naming Pattern: Specify a string, that will be used in instance hostname. Just A-Z, a-z, 0-9, dash and underline are allowed.

Back
Create

6. Now the Apache HTTP Server application is successfully added to the newly created quick-env-4 environment. Click the *Deploy This Environment* button to start the deployment:

The screenshot shows the OpenStack Murano dashboard for an environment named 'quick-env-4'. The interface includes a sidebar with navigation options like Project, Admin, Identity, Applications, and Catalog. The main content area shows the environment name 'quick-env-4' with a 'Delete Environment' button. Below this, there are tabs for Components, Topology, and Deployment History. The 'Application Components' section displays a grid of component icons including Apache HTTP Server, Apache Tomcat, Chef Server, Docker MariaDB, Docker Nginx, and Docker Redis. A 'Drop Components here' area is visible below the grid. At the bottom, a table lists the components, with 'ApacheHttpServer' highlighted in green. The table has columns for Name, Type, Status, Last operation, Time updated, and Actions. The 'ApacheHttpServer' row shows a status of 'Ready to deploy' and a 'Delete Component' button.

It may take some time for the environment to deploy. Wait until the status is changed from Deploying to Ready.

7. Navigate to *Applications > Catalog > Environments* to view the details.

2.1.3 Delete an application

To delete an application that belongs to the environment:

1. Log in to the OpenStack dashboard.
2. Navigate to *Applications > Catalog > Environments*.
3. Click on the name of the environment to view its details, which include components, topology, and deployment history.
4. In the *Component List* section, click on the *Delete Component* button next to the application to be deleted. Confirm the deletion.

Note: If an application that you are deleting has already been deployed, you should redeploy it to apply the recent changes. If the environment has not been deployed with this component, the changes are applied immediately on receiving the confirmation.

2.2 User Guide

2.2.1 Managing environments

An environment is a set of logically connected applications that are grouped together for an easy management. By default, each environment has a single network for all its applications, and the deployment of the environment is defined in a single heat stack. Applications in different environments are always independent from one another.

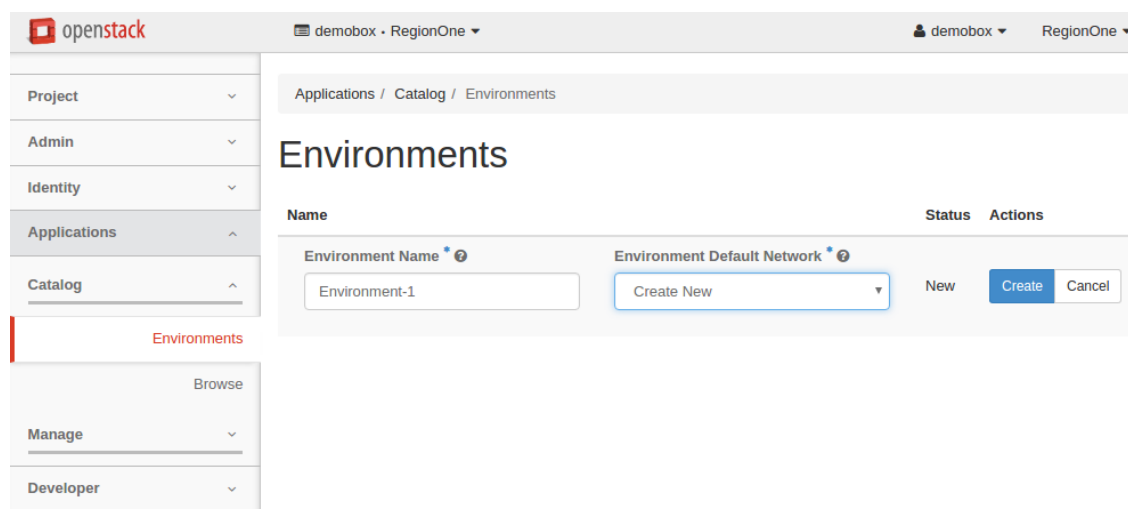
An environment is a single unit of deployment. This means that you deploy not an application but an environment that contains one or multiple applications.

Using OpenStack dashboard you can easily perform such actions with an environment as creating, editing, reviewing, deploying, and others.

Create an environment

To create an environment, perform the following steps:

1. In OpenStack dashboard, navigate to Applications > Catalog > Environments.
2. On the *Environments* page, click the *Add New* button.
3. In the *Environment Name* field, enter the name for the new environment.
4. From the *Environment Default Network* drop-down list, choose a specific network, if necessary, or leave the default *Create New* option to generate a new network.



5. Click the rightmost *Create* button. You will be redirected to the page with the environment components.

Alternatively, you can create an environment automatically using the *Quick Deploy* button below any application in the Application Catalog. For more information, see: [Quick deploy](#).

Edit an environment

You can edit the name of an environment. For this, perform the following steps:

1. In OpenStack dashboard, navigate to Applications > Catalog > Environments.
2. Position your mouse pointer over the environment name and click the appeared pencil icon.
3. Edit the name of the environment.
4. Click the tick icon to apply the change.

Review an environment

This section provides a general overview of an environment, its structure, possible statuses, and actions. An environment groups applications together. An application that is added to an environment is called a component.

To see an environment status, navigate to *Applications > Catalog > Environments*. Environments may have one of the following statuses:

- **Ready to configure.** When the environment is new and contains no components.
- **Ready to deploy.** When the environment contains a component or multiple components and is ready for deployment.
- **Ready.** When the environment has been successfully deployed.
- **Deploying.** When the deploying is in progress.
- **Deploy FAILURE.** When the deployment finished with errors.
- **Deleting.** When deleting of an environment is in progress.
- **Delete FAILURE.** You can abandon the environment in this case.

Currently, the component status corresponds to the environment status.

To review an environment and its components, or reconfigure the environment, click the name of an environment or simply click the rightmost *Manage Components* button.

- On the *Components* tab you can:
 - Add or delete a component from an environment
 - Send an environment to deploy
 - Track a component status
 - Call murano actions of a particular application in a deployed environment:
For more information on murano actions, see: *Murano actions*.
- On the *Topology*, *Deployment History*, and *Latest Deployment Log* tabs of the environment page you can view the following:
 - The application topology of an environment. For more information, see: *Application topology*.
 - The log of a particular deployment. For more information, see: *Deployment history*.
 - The information on the latest deployment of an environment. For more information, see: *Latest deployment log*.

The screenshot shows the Murano dashboard for environment 'quick-env-4'. The 'Application Components' section is active, displaying a list of components. The 'KubernetesCluster' component is highlighted in the table below. The 'Actions' dropdown menu is open, showing various management options.

Name	Type	Status	Last operation	Time updated	Actions
KubernetesCluster	Kubernetes Cluster	Ready	Kubernetes cluster is up and running	May 26, 2016, 1:02 p.m.	<ul style="list-style-type: none"> Delete Component scaleGatewaysDown scaleGatewaysUp scaleNodesDown scaleNodesUp

2.2.2 Managing applications

In murano, each application, as well as the form of application data entry, is defined by its package. The murano dashboard allows you to import and manage packages as well as search, filter, and add applications from catalog to environments.

This section provides detailed instructions on how to import application packages into murano and then add applications to an environment and deploy it. This section also shows you how to find component details, application topology, and deployment logs.

Import an application package

There are several ways of importing an application package into murano:

- *from a zip file*
- *from murano applications repository*
- *from bundles of applications*

From a zip file

Perform the following steps to import an application package from a .zip file:

1. In OpenStack dashboard, navigate to *Applications > Manage > Packages*.
2. Click the *Import Package* button on the top right of the page.

- From the *Package source* drop-down list choose *File*, then click *Browse* to select a .zip file you want to import, and then click *Next*.

- At this step, the package is already uploaded. Choose a category from the *Application Category* menu. You can select multiple categories while holding down the Ctrl key. If necessary, verify and update the information about the package, then click the *Create* button.

✕

Import Package

Name

Application Category

- Web
- Load Balancers
- Message Queue
- Databases
- Key-Value Storage

Tags ⓘ

Public

Active

Description

MySQL is a relational database management system (RDBMS), and ships with no GUI tools to administer MySQL databases or manage data contained within the databases.

Description:

Name is a human-readable name of a package.

Categories are a predefined set of values used to filter the packages.

Tags are an arbitrary comma-separated values also used to filter the packages.

Public Defines whether or not a package is available for use by other tenants. (Applies to package dependencies)

Active Allows the status of a package to be changed. (Applies to package dependencies)

Description consists of several sentences about the package's purpose.

Note: Though specifying a category is optional, we recommend that you specify at least one. It helps to filter applications in the catalog.

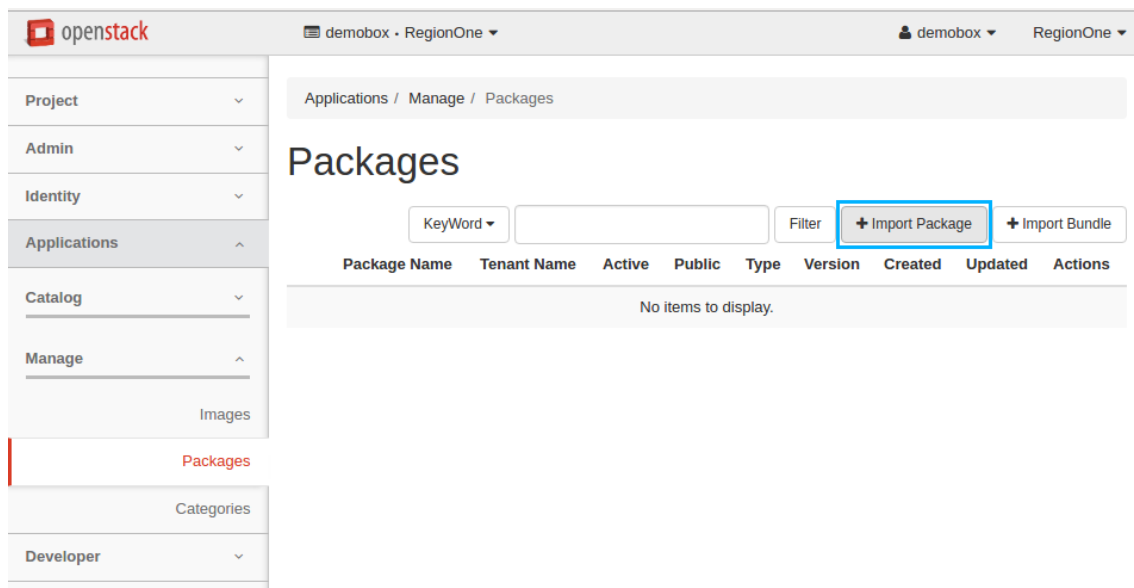
Green messages appear at the top right corner when the application is successfully uploaded. In case of a failure, you will see a red message with the problem description. For more information, please refer to the logs.

From a repository

Perform the following steps to import an application package from murano applications repository:

Note: To import an application package from a repository, you need to know the full name of the package. For the packages names, go to <http://apps.openstack.org/#tab=murano-apps> and click on the desired package to see its full name.

1. In OpenStack dashboard, navigate to *Applications > Manage > Packages*.
2. Click the *Import Package* button on the top right of the page.



3. From the *Package source* drop-down list, choose *Repository*, enter the package name, and then click *Next*. Note that you may also specify the version of the package.

 A screenshot of the 'Import Package' dialog box. The title is 'Import Package'. On the left, under 'Package Source', a dropdown menu is set to 'Repository'. Below it, the 'Package Name' field is empty and has a red asterisk with a question mark. The 'Package version' dropdown is set to 'Optional'. On the right, under 'Description:', there are instructions: 'Package Name: Fully qualified package name.', 'Package Version: Version of the package (optional).', and 'The package is going to be imported from http://storage.apps.openstack.org/ repository.' A note states: 'Note: If the package depends upon other packages and/or requires specific glance images, those are going to be installed with it from murano repository.' At the bottom right, there are 'Cancel' and 'Next' buttons.

4. At this step, the package is already uploaded. Choose a category from the *Application Category*

menu. You can select multiple categories while holding down the Ctrl key. If necessary, verify and update the information about the package, then click the *Create* button.

×

Import Package

Name

Application Category

- Web
- Load Balancers
- Message Queue
- Databases
- Key-Value Storage

Tags ⓘ

Public

Active

Description

MySQL is a relational database management system (RDBMS), and ships with no GUI tools to administer MySQL databases or manage data contained within the databases.

Description:

Name is a human-readable name of a package.

Categories are a predefined set of values used to filter the packages.

Tags are an arbitrary comma-separated values also used to filter the packages.

Public Defines whether or not a package is available for use by other tenants. (Applies to package dependencies)

Active Allows the status of a package to be changed. (Applies to package dependencies)

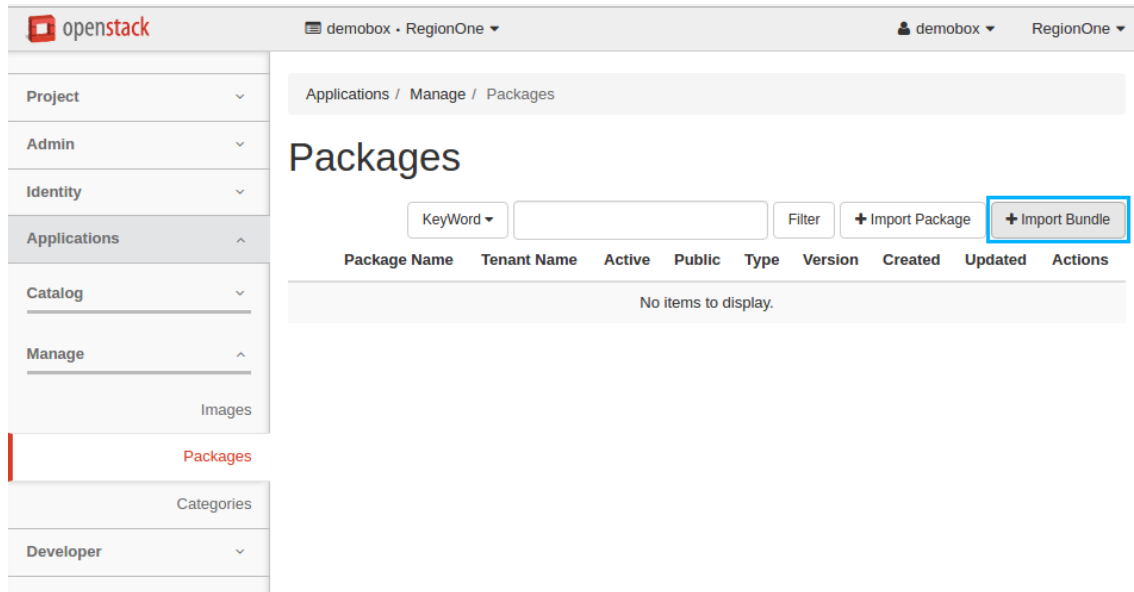
Description consists of several sentences about the package's purpose.

From a bundle of applications

Perform the following steps to import a bundle of applications:

Note: To import an application bundle from a repository, you need to know the full name of the package bundle. To find it out, go to <http://apps.openstack.org/#tab=murano-apps> and click on the desired bundle to see its full name.

1. In OpenStack dashboard, navigate to *Applications > Manage > Packages*.
2. Click the *Import Bundle* button on the top right of the page.



- From the *Package Bundle Source* drop-down list, choose *Repository*, enter the bundle name, and then click *Create*.

Import Bundle ✕

Package Bundle Source

Repository

Bundle Name * ?

Description:

Bundle Name: Bundle's full name.

The bundle is going to be installed from <http://storage.apps.openstack.org/> repository.

Note: You'll have to configure each package installed from this bundle separately.
If packages depend upon other packages and/or require specific glance images, those are going to be installed with them from murano repository.

Cancel

Create

Search for an application in the catalog

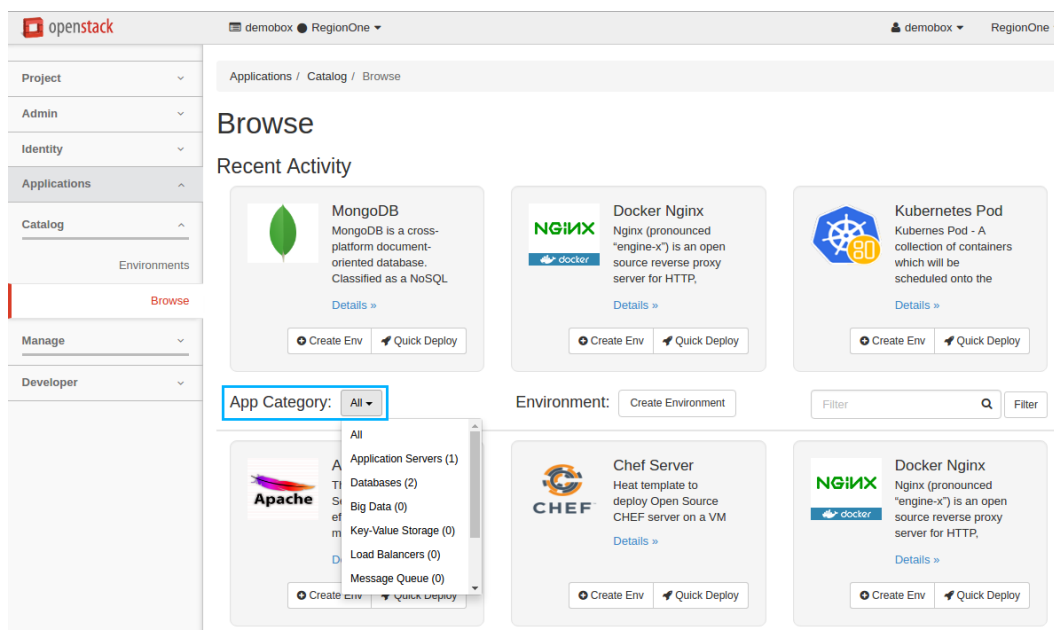
When you have imported many applications and want to quickly find a required one, you can filter them by category, tags and words that the application name or description contains:

In OpenStack dashboard, navigate to *Applications > Catalog > Browse*.

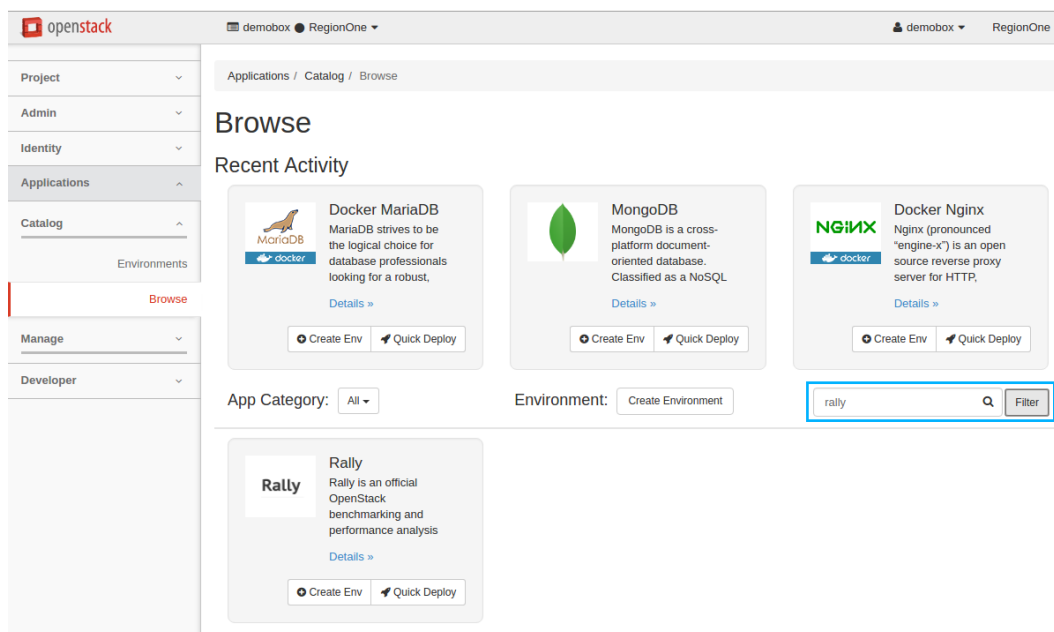
The page is divided into two sections:

- **Recent Activity** shows the most recently imported or deployed applications.
- The bottom section contains all the available applications sorted alphabetically.

To view all the applications of a specific category, select it from the *App Category* drop-down list:

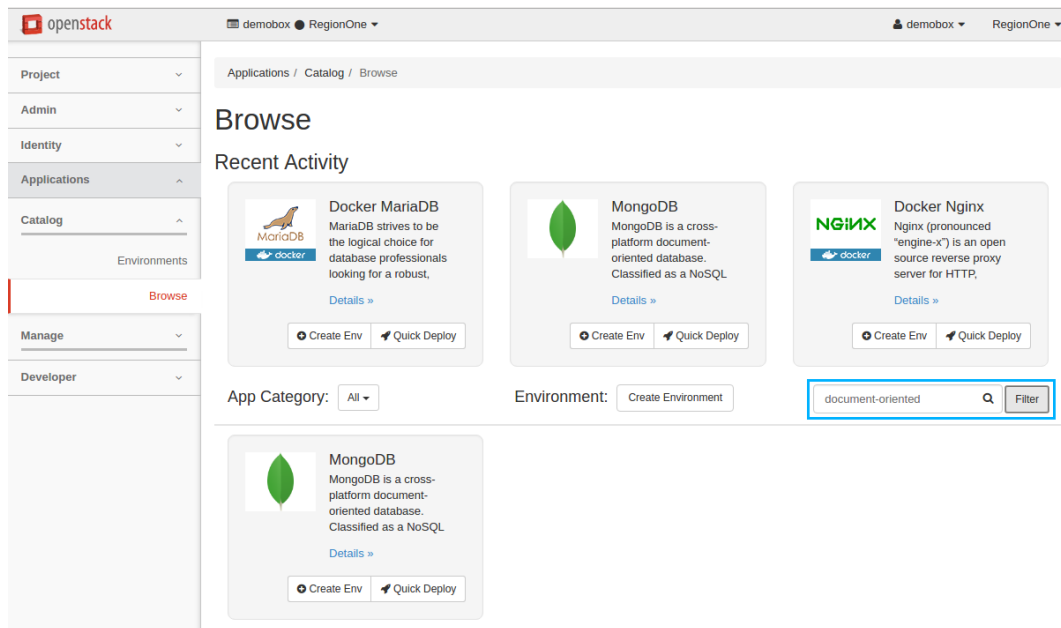


To filter applications by tags or words from the application name or description, use the rightmost filter:



Note: Tags can be specified during the import of an application package.

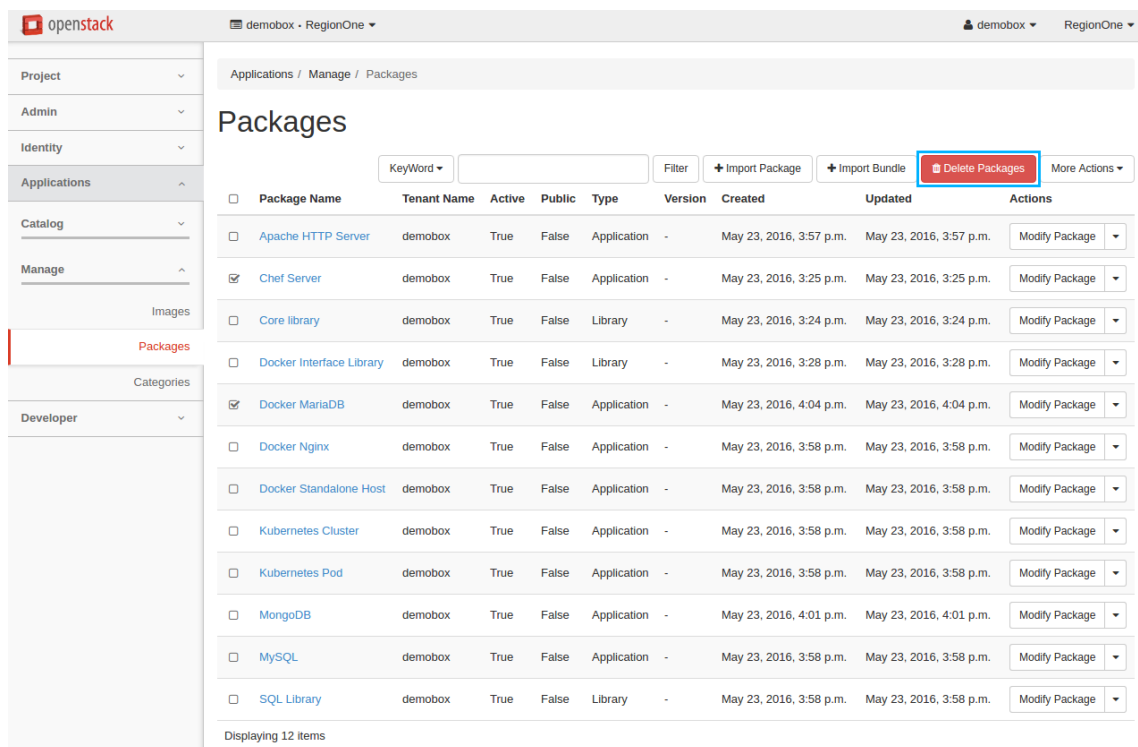
For example, there is an application that has the word *document-oriented* in description. Let's find it with the filter. The following screenshot shows you the result.



Delete an application package

To delete an application package from the catalog, please perform the following steps:

1. In OpenStack dashboard, navigate to *Applications > Manage > Packages*.
2. Select a package or multiple packages you want to delete and click *Delete Packages*.



3. Confirm the deletion.

Add an application to environment

After uploading an application, the second step is to add it to an environment. You can do this:

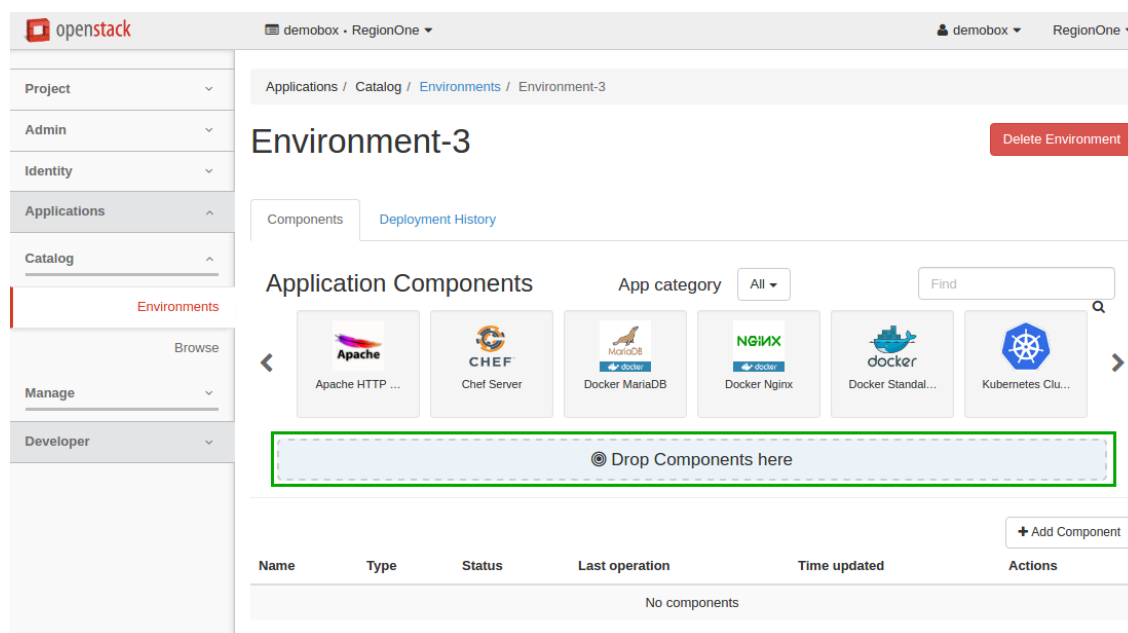
- from *environment details page*
- from *applications catalog page*

From environment details page

1. In OpenStack dashboard, navigate to *Applications > Catalog > Environments*.
2. Find the environment you want to manage and click *Manage Components*, or simply click on the environment's name.
3. Proceed with the *Drop Components here* field or the *Add Component* button.

Use of Drop Components here field

1. On the Environment Components page, drag and drop a desired application into the *Drop Components here* field under the *Application Components* section.

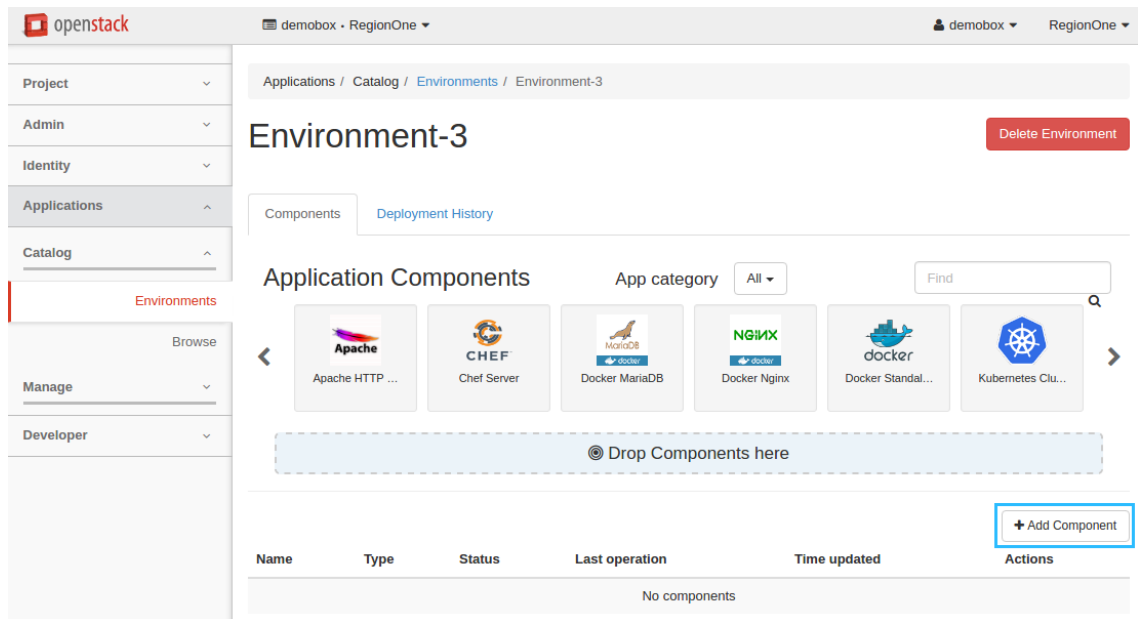


2. Configure the application. Note that the settings may vary from app to app and are predefined by the application author. When done, click *Next*, then click *Create*.

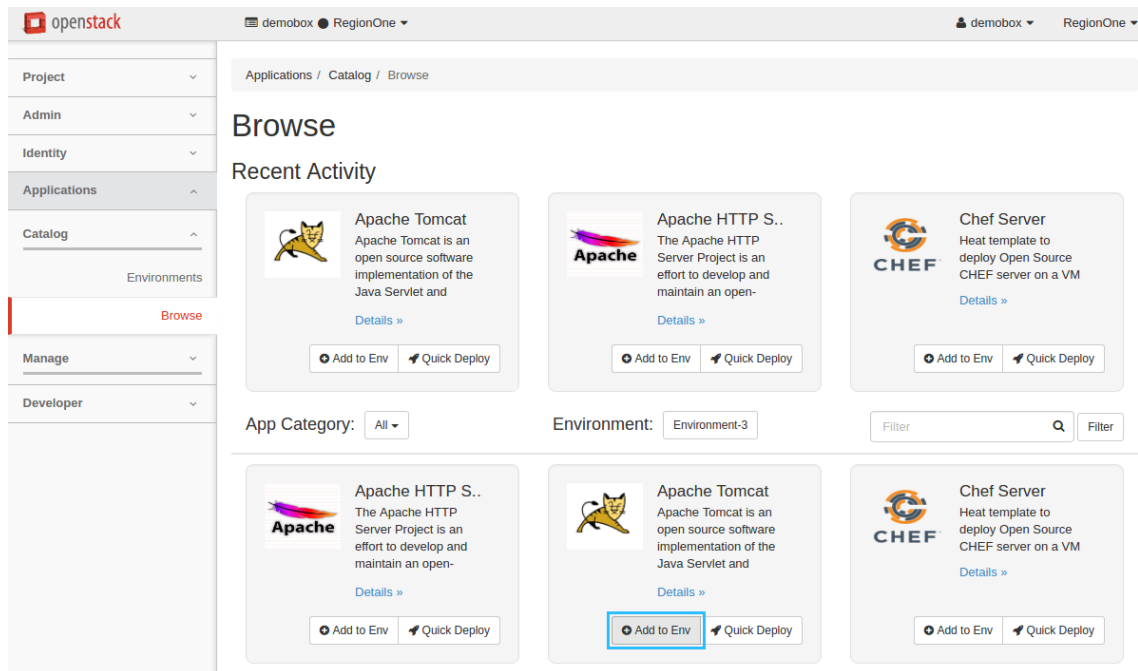
Now the application appears in the *Component List* section on the Environment Components page.

Use of Add Component button

1. On the Environment Components page, click *Add Component*.



2. Find the application you want to add and click *Add to Env*.



3. Configure the application and click *Next*. Note that the settings may vary from app to app and are predefined by the application author.
4. To add more applications, check *Continue application adding*, then click *Create* and repeat the steps above. Otherwise, just click *Create*.

Configure Application: Apache Tomcat

Application Name *

Continue application adding

Apache Tomcat
Application Name:
 Enter a desired name for the application. Just A-Z, a-z, 0-9, dash and underline are allowed

Continue application adding:
 If checked, you will be returned to the Application Catalog page. If not - to the Environment page, where you can deploy the application.

Now the application appears in the *Component List* section on the Environment Components page.

From applications catalog page

1. In OpenStack dashboard, navigate to *Applications > Catalog > Browse*.
2. On the Applications catalog page, use one of the following methods:
 - *Quick deploy*. Automatically creates an environment, adds the selected application, and redirects you to the page with the environment components.
 - *Add to Env*. Adds an application to an already existing environment.

Quick Deploy button

1. Find the application you want to add and click *Quick Deploy*. Let's add Apache Tomcat, for example.

openstack demobox RegionOne

Applications / Catalog / Browse

Browse

Recent Activity

Apache Tomcat
Apache Tomcat is an open source software implementation of the Java Servlet and

[Details >](#)

Apache HTTP S..
The Apache HTTP Server Project is an effort to develop and maintain an open-

[Details >](#)

Chef Server
Heat template to deploy Open Source CHEF server on a VM

[Details >](#)

App Category: All Environment: Environment-3 Filter

Apache HTTP S..
The Apache HTTP Server Project is an effort to develop and maintain an open-

[Details >](#)

Apache Tomcat
Apache Tomcat is an open source software implementation of the Java Servlet and

[Details >](#)

Chef Server
Heat template to deploy Open Source CHEF server on a VM

[Details >](#)

2. Configure the application. Note that the settings may vary from app to app and are predefined by the application author. When done, click *Next*, then click *Create*. In the example below we assign a floating IP address.

Configure Application: Apache Tomcat
✕

Application Name *

Assign Floating IP

Apache Tomcat

Apache License, Version 2.0

Application Name: Enter a desired name for the application. Just A-Z, a-z, 0-9, dash and underline are allowed

Assign Floating IP: Select to true to assign floating IP automatically

Next

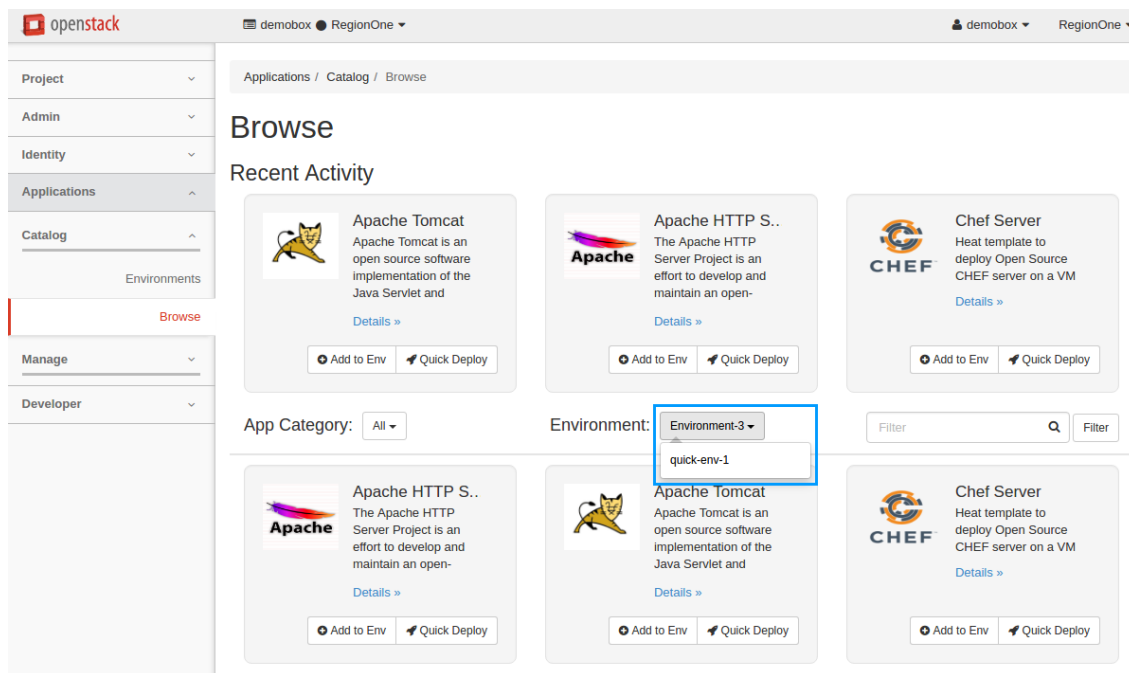
Now the Apache Tomcat application is successfully added to an automatically created quick-env-1 environment.

The screenshot shows the OpenStack Murano console interface. The breadcrumb navigation indicates the path: Applications / Catalog / Environments / quick-env-1. A green success message in the top right corner states: "Success: The 'Apache Tomcat' application successfully added to environment." The main content area is titled "quick-env-1" and includes tabs for "Components", "Topology", and "Deployment History". Under the "Application Components" section, there is a carousel of application icons: Apache HTTP Se..., Apache Tomcat, CHEF, Docker MariaDB, NGINX, and Docker Standalon... The "Apache Tomcat" component is highlighted with a green box. Below the carousel is a dashed box labeled "Drop Components here". At the bottom, there is a table with columns: Name, Type, Status, Last operation, Time updated, and Actions. The table contains one row for "Tomcat" with status "Ready to deploy" and last operation "Component draft created". A "Delete Component" button is visible in the Actions column.

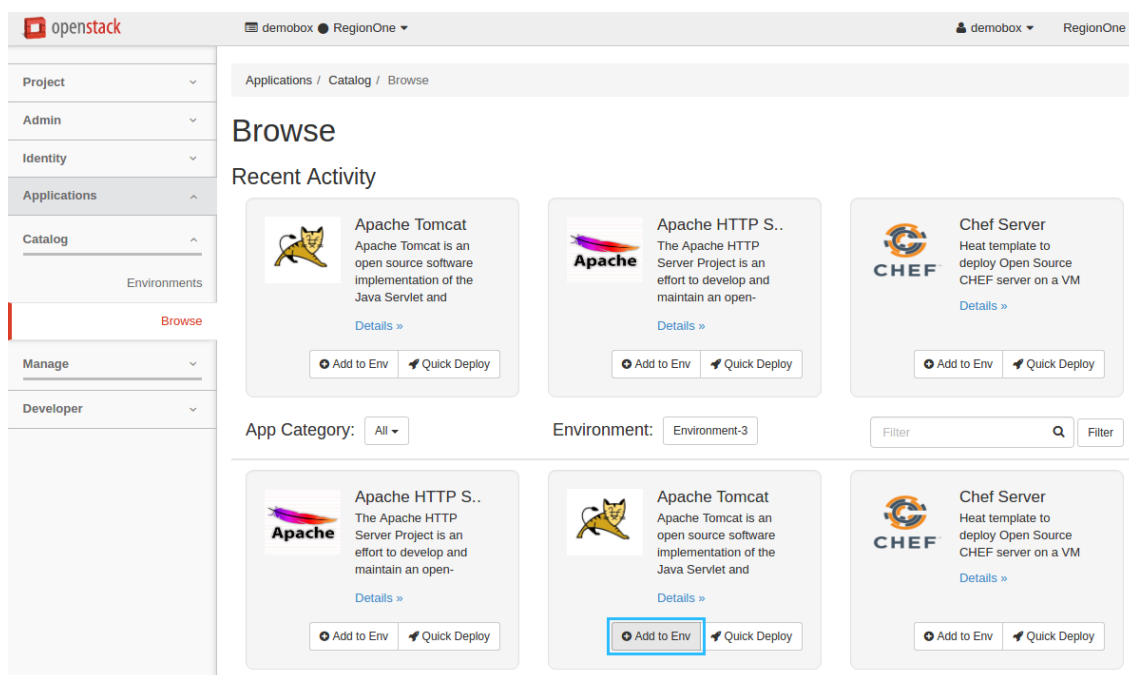
Name	Type	Status	Last operation	Time updated	Actions
Tomcat	Apache Tomcat	Ready to deploy	Component draft created		Delete Component

Add to Env button

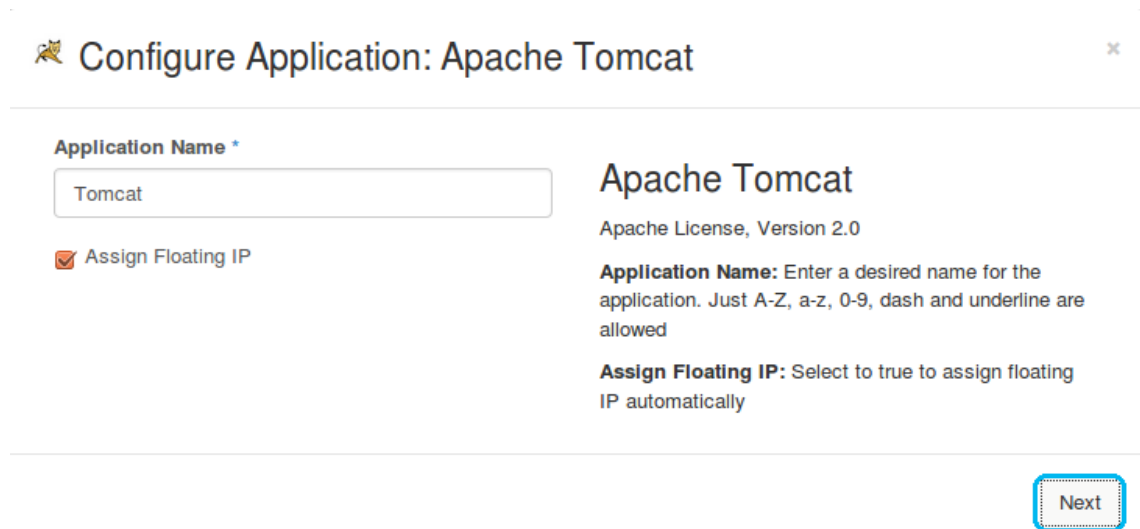
1. From the *Environment* drop-down list, select the required environment.



2. Find the application you want to add and click *Add to Env*. Let's add Apache Tomcat, for example.



3. Configure the application and click *Next*. Note that the settings may vary from app to app and are predefined by the application author. In the example below we assign a floating IP address.



Application Name *

Tomcat

Assign Floating IP

Apache Tomcat

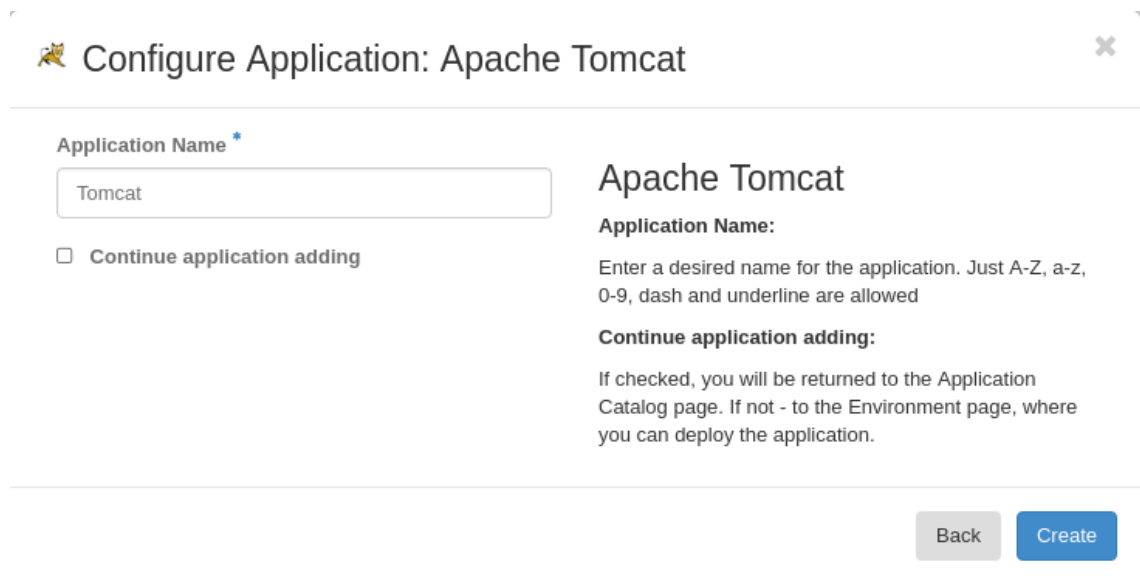
Apache License, Version 2.0

Application Name: Enter a desired name for the application. Just A-Z, a-z, 0-9, dash and underline are allowed

Assign Floating IP: Select to true to assign floating IP automatically

Next

4. To add more applications, check *Add more applications to the environment*, then click *Create* and repeat the steps above. Otherwise, just click *Create*.



Application Name *

Tomcat

Continue application adding

Apache Tomcat

Application Name:
Enter a desired name for the application. Just A-Z, a-z, 0-9, dash and underline are allowed

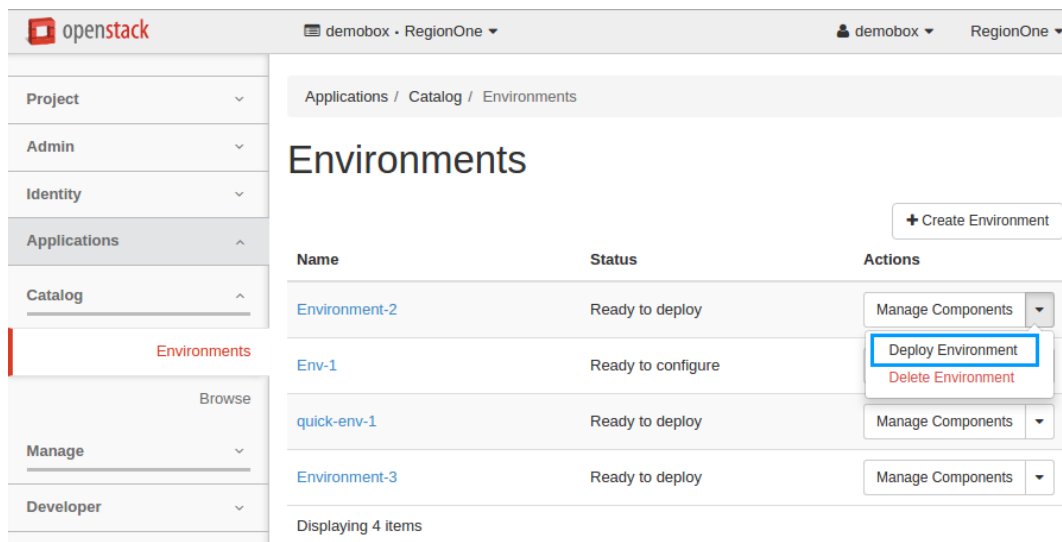
Continue application adding:
If checked, you will be returned to the Application Catalog page. If not - to the Environment page, where you can deploy the application.

Back Create

Deploy an environment

Make sure to add necessary applications to your environment, then deploy it following one of the options below:

- Deploy an environment from the Environments page
 1. In OpenStack dashboard, navigate to *Applications > Catalog > Environments*.
 2. Select *Deploy Environment* from the Actions drop-down list next to the environment you want to deploy.



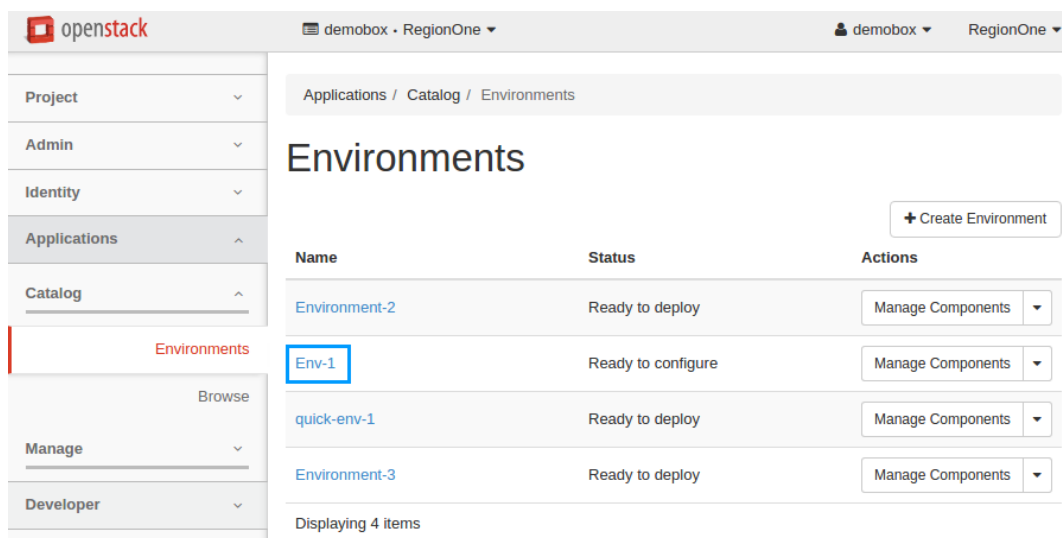
The screenshot shows the OpenStack Murano dashboard. The breadcrumb navigation is 'Applications / Catalog / Environments'. The main heading is 'Environments'. There is a '+ Create Environment' button. Below is a table with the following data:

Name	Status	Actions
Environment-2	Ready to deploy	Manage Components
Env-1	Ready to configure	Deploy Environment Delete Environment
quick-env-1	Ready to deploy	Manage Components
Environment-3	Ready to deploy	Manage Components

At the bottom of the table, it says 'Displaying 4 items'.

It may take some time for the environment to deploy. Wait for the status to change from *Deploying* to *Ready*. You cannot add applications to your environment during deployment.

- Deploy an environment from the Environment Components page
 1. In OpenStack dashboard, navigate to *Applications > Catalog > Environments*.
 2. Click the name of the environment you want to deploy.



The screenshot shows the OpenStack Murano dashboard. The breadcrumb navigation is 'Applications / Catalog / Environments'. The main heading is 'Environments'. There is a '+ Create Environment' button. Below is a table with the following data:

Name	Status	Actions
Environment-2	Ready to deploy	Manage Components
Env-1	Ready to configure	Manage Components
quick-env-1	Ready to deploy	Manage Components
Environment-3	Ready to deploy	Manage Components

At the bottom of the table, it says 'Displaying 4 items'.

3. On the Environment Components page, click *Deploy This Environment* to start the deployment.

The screenshot shows the OpenStack Murano dashboard for environment 'Env-1'. The left sidebar contains navigation menus for Project, Admin, Identity, Applications, and Catalog. The main content area shows the 'Application Components' section with a search bar and a list of components: Apache HTTP, Apache Tomcat, Chef Server, Docker MariaDB, Docker Nginx, and Docker Standalone. Below the component list is a table with the following data:

Name	Type	Status	Last operation	Time updated	Actions
Tomcat	Apache Tomcat	Ready to deploy	Component draft created		Delete Component

It may take some time for the environment to deploy. You cannot add applications to your environment during deployment. Wait for the status to change from *Deploying* to *Ready*. You can check the status either on the Environments page or on the Environment Components page.

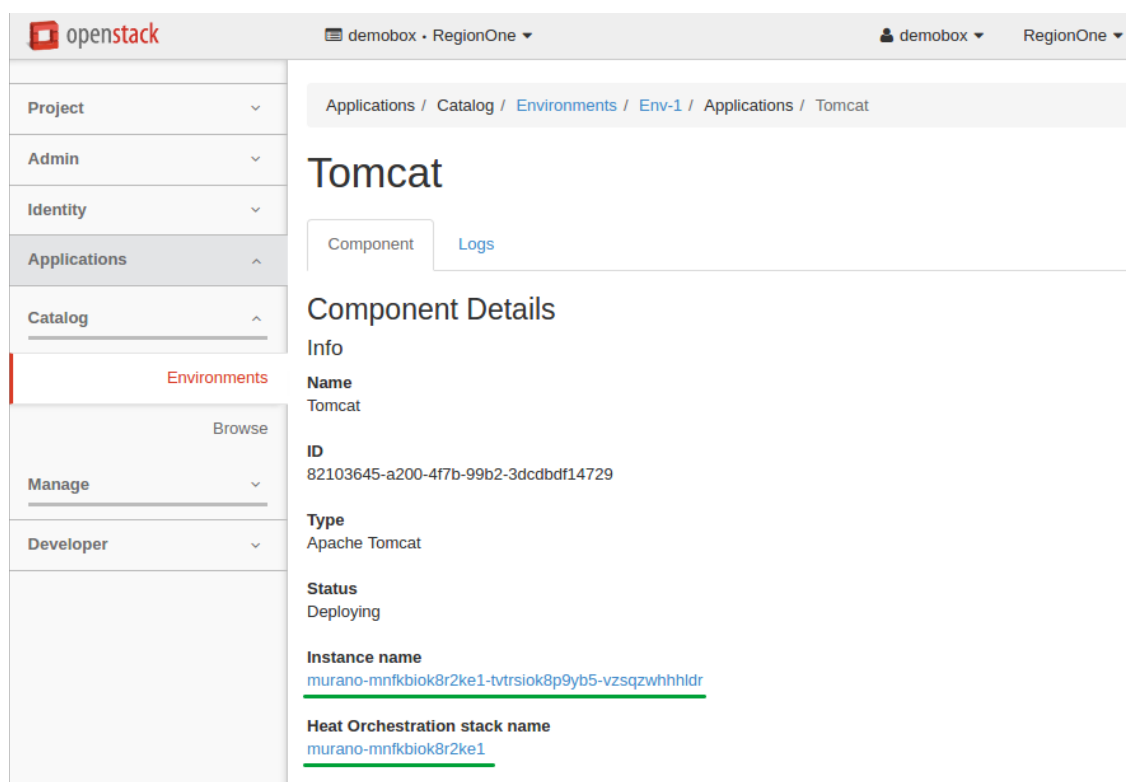
Browse component details

You can browse component details to find the following information about a component:

- Name
- ID
- Type
- Instance name (available only after deployment)
- Heat orchestration stack name (available only after deployment)

To browse a component details, perform the following steps:

1. In OpenStack dashboard, navigate to *Applications > Catalog > Environments*.
2. Click the name of the required environment.
3. In the *Component List* section, click the name of the required component.



The screenshot shows the OpenStack Murano dashboard interface. The top navigation bar includes the OpenStack logo, the user 'demobox', and the region 'RegionOne'. The breadcrumb trail is 'Applications / Catalog / Environments / Env-1 / Applications / Tomcat'. The main content area is titled 'Tomcat' and has two tabs: 'Component' (selected) and 'Logs'. Below the tabs is the 'Component Details' section, which includes an 'Info' subsection with the following details:

- Name:** Tomcat
- ID:** 82103645-a200-4f7b-99b2-3dcdcdf14729
- Type:** Apache Tomcat
- Status:** Deploying
- Instance name:** [murano-mnfkbiok8r2ke1-tvtrsiok8p9yb5-vzsqzwhhldr](#)
- Heat Orchestration stack name:** [murano-mnfkbiok8r2ke1](#)

The left sidebar contains navigation menus for 'Project', 'Admin', 'Identity', 'Applications', and 'Catalog'. The 'Applications' menu is expanded, showing 'Environments' (with a 'Browse' button) and 'Developer'.

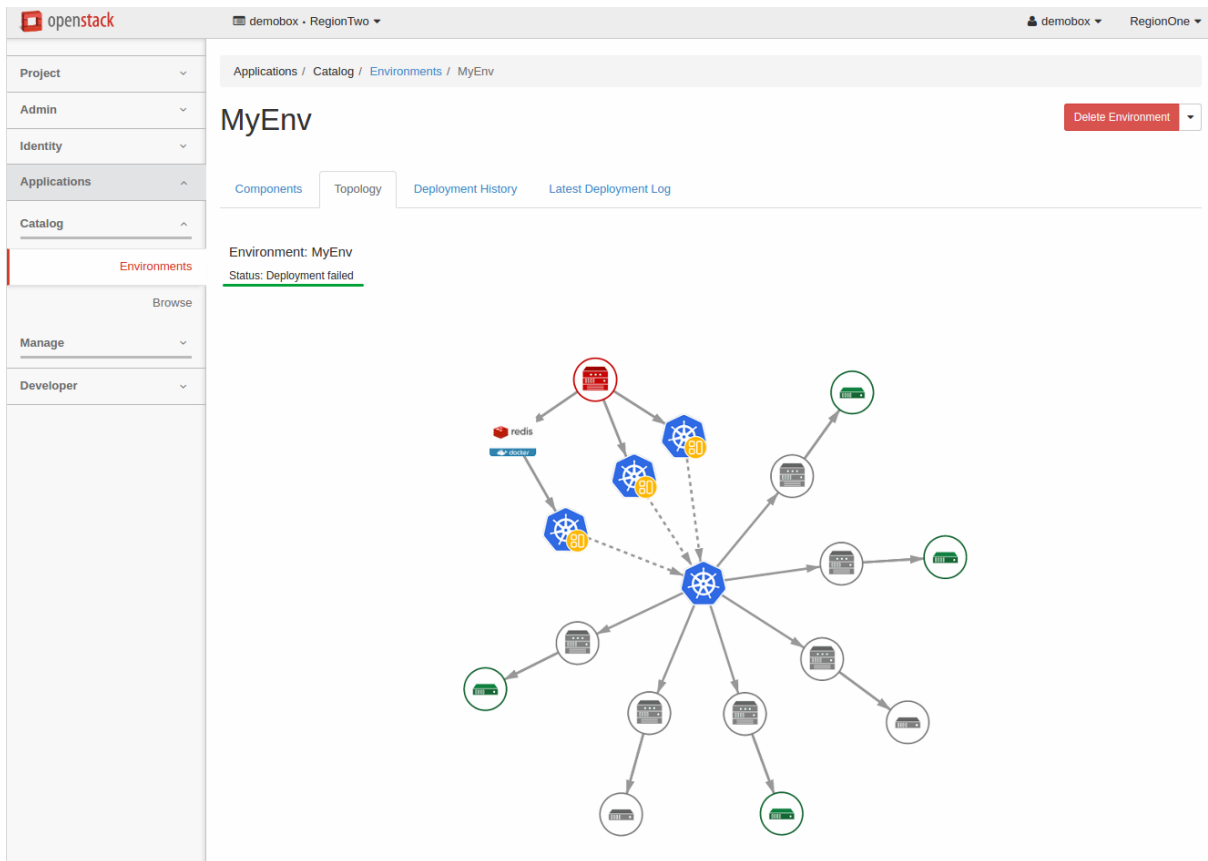
The links redirect to corresponding horizon pages with the detailed information on instance and heat stack.

Application topology



Once you add an application to your environment, the application topology of this environment becomes available in a separate tab. The topology represents an elastic diagram showing the relationship between a component and the infrastructure it runs on. To view the topology:

1. In OpenStack dashboard, navigate to *Applications > Catalog > Environments*.
2. Click the name of the necessary environment.
3. Click the *Topology* tab.

The topology is helpful to visually display complex components, for example Kubernetes. The red icons reflect errors during the deployment while the green ones show success.



The following elements of the topology are virtual machine and an instance of dependent MuranoPL class:

Element	Meaning
	Virtual machine
	Instance

Position your mouse pointer over an element to see its name, ID, and other details.

The screenshot shows the OpenStack Murano dashboard interface. At the top, the OpenStack logo and user information 'tlashchova · RegionOne' are visible. The breadcrumb navigation is 'Applications / Catalog / Environments / quick-env-5'. The main heading is 'quick-env-5' with a 'Delete Environment' button. Below the heading are tabs for 'Components', 'Topology', 'Deployment History', and 'Latest Deployment Log'. The 'Topology' tab is active, showing a diagram of the environment's components: Apache, WordPress, and MySQL. A green box highlights the instance details for 'uksykioccc5wj4'.

Environment: quick-env-5
Status: Deployed

```

Name: uksykioccc5wj4
Availabilityzone: nova
Openstackid: 252b4aa9-05b7-4dcc-be08-0cd1bc4d310b
Securitygroupname: None
Image: 213b54b5-80b6-4c19-a264-fd0ea3104002
Id: 140e53e1-6bfd-42c4-bab3-131fc4ab8742
Keyname:
Floatingipaddress: None
Flavor: m1.medium
Type: io.murano.resources.LinuxMuranoInstance
Assignfloatingip: False

```

Deployment logs

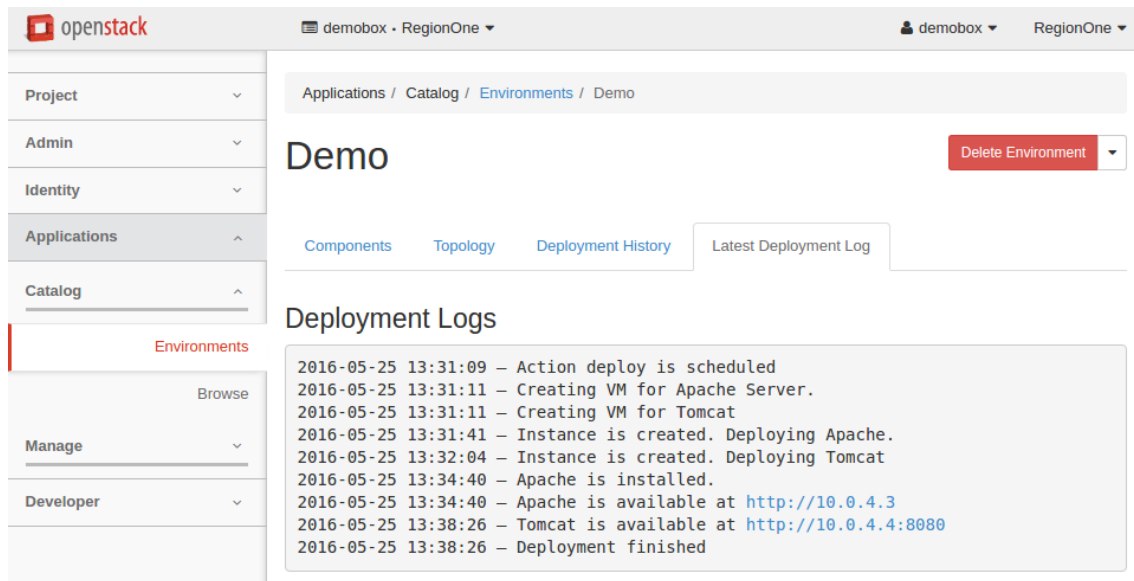
To get detailed information on a deployment, use:

- *Deployment history*, which contains logs and deployment structure of an environment.
- *Latest deployment log*, which contains information on the latest deployment of an environment.
- *Component logs*, which contain logs on a particular component in an environment.

Deployment history

To see the log of a particular deployment, proceed with the steps below:

1. In OpenStack dashboard, navigate to *Applications > Catalog > Environments*.
2. Click the name of the required environment.
3. Click the *Deployment History* tab.
4. Find the required deployment and click *Show Details*.
5. Click the *Logs* tab to see the logs.



openstack demobox · RegionOne demobox RegionOne

Applications / Catalog / Environments / Demo

Demo

Delete Environment

Components Topology Deployment History Latest Deployment Log

Deployment Logs

```

2016-05-25 13:31:09 - Action deploy is scheduled
2016-05-25 13:31:11 - Creating VM for Apache Server.
2016-05-25 13:31:11 - Creating VM for Tomcat
2016-05-25 13:31:41 - Instance is created. Deploying Apache.
2016-05-25 13:32:04 - Instance is created. Deploying Tomcat
2016-05-25 13:34:40 - Apache is installed.
2016-05-25 13:34:40 - Apache is available at http://10.0.4.3
2016-05-25 13:38:26 - Tomcat is available at http://10.0.4.4:8080
2016-05-25 13:38:26 - Deployment finished

```

Latest deployment log

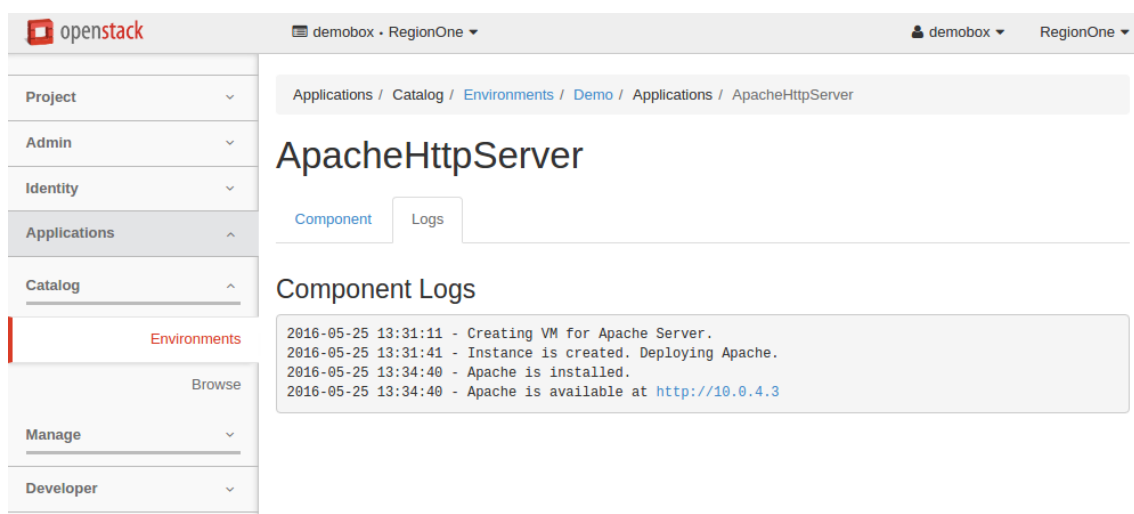
To see the latest deployment log, proceed with the steps below:

1. In OpenStack dashboard, navigate to *Applications > Catalog > Environments*.
2. Click the name of the required environment.
3. Click the *Latest Deployment Log* tab to see the logs.

Component logs

To see the logs of a particular component of an environment, proceed with the steps below:

1. In OpenStack dashboard, navigate to *Applications > Catalog > Environments*.
2. Click the name of the required environment.
3. In the *Component List* section, click the required component.
4. Click the *Logs* tab to see the component logs.



openstack demobox · RegionOne demobox RegionOne

Applications / Catalog / Environments / Demo / Applications / ApacheHttpServer

ApacheHttpServer

Component Logs

```

2016-05-25 13:31:11 - Creating VM for Apache Server.
2016-05-25 13:31:41 - Instance is created. Deploying Apache.
2016-05-25 13:34:40 - Apache is installed.
2016-05-25 13:34:40 - Apache is available at http://10.0.4.3

```

Delete an application

To delete an application that belongs to the environment:

1. In OpenStack dashboard, navigate to *Applications > Catalog > Environments*.
2. Click on the name of the environment you want to delete an application from.

Name	Status	Actions
Environment-2	Ready to deploy	Manage Components
Env-1	Ready to configure	Manage Components
quick-env-1	Ready to deploy	Manage Components
Environment-3	Ready to deploy	Manage Components

3. In the *Component List* section, click the *Delete Component* button next to the application you want to delete. Then confirm the deletion.

Name	Type	Status	Last operation	Time updated	Actions
Tomcat	Apache Tomcat	Ready to deploy	Component draft created		Delete Component

Note: If the application that you are deleting has already been deployed, you should redeploy the environment to apply the recent changes. If the environment has not been deployed with this component, the changes are applied immediately on receiving the confirmation.

2.2.3 Log in to murano-spawned instance

After the application is successfully deployed, you may need to log in to the virtual machine with the installed application.

All cloud images, including images imported from the [OpenStack Application Catalog](#), have password authentication turned off. Therefore, it is not possible to log in from the dashboard console. SSH is used to reach an instance spawned by murano.

Possible default image users are:

- *ec2-user*
- *ubuntu* or *debian* (depending on the operating system)

To log in to murano-spawned instance, perform the following steps:

1. Prepare a key pair.

To log in through SSH, provide a key pair during the application creation. If you do not have a key pair, click the plus sign to create one directly from the *Configure Application* dialog.

git Configure Application: GitChef ×

Instance flavor

Instance Image *

Key Pair

 +

Availability zone

Network

Instance Naming Pattern ?

GitChef

Specify some instance parameters on which the application would be created

Instance flavor: Select registered in Openstack flavor. Consider that application performance depends on this parameter.

Instance image: Select valid image for the application. Image should already be prepared and registered in glance.

Key Pair: Select the Key Pair to control access to instances. You can login to instances using this KeyPair after the deployment of application.

Availability zone: Select availability zone where application would be installed.

Network: Select a network to join. 'Auto' corresponds to a default environment's network.

Instance Naming Pattern: Specify a string, that will be used in instance hostname. Just A-Z, a-z, 0-9, dash and underline are allowed.

2. After the deployment is completed, find out the instance IP address. For this, see:

- Deployment logs

- Detailed instance parameters

See the *Instance name* link on the *Component Details* page.

3. To connect to the instance through SSH with the key pair, run:

```
$ ssh <username>@<IP> -i <key.location>
```


2.2.4 Using CLI

This section provides murano end users with information on how they can use the Application Catalog through the command-line interface (CLI).

Using `python-muranoclient`, the CLI client for murano, you can easily manage your environments, packages, categories, and deploy environments.

Install and use the murano client

The Application Catalog project provides a command-line client, `python-muranoclient`, which enables you to access the project API. For prerequisites, see [Install the prerequisite software](#).

To install the latest murano CLI client, run the following command in your terminal:

```
$ pip install python-muranoclient
```

Discover the client version number

To discover the version number for the `python-muranoclient`, run the following command:

```
$ murano --version
```

To check the latest version, see [Client library for Murano API](#).

Upgrade or remove the client

To upgrade or remove the `python-muranoclient`, use the corresponding commands.

To upgrade the client:

```
$ pip install --upgrade python-muranoclient
```

To remove the client:

```
$ pip uninstall python-muranoclient
```

Set environment variables

To use the murano client, you must set the environment variables. To do this, download and source the OpenStack RC file. For more information, see [Download and source the OpenStack RC file](#).

Alternatively, create the `PROJECT-openrc.sh` file from scratch. For this, perform the following steps:

1. In a text editor, create a file named `PROJECT-openrc.sh` containing the following authentication information:

```
export OS_USERNAME=user
export OS_PASSWORD=password
export OS_PROJECT_NAME=tenant
export OS_USER_DOMAIN_NAME=Default
export OS_PROJECT_DOMAIN_NAME=Default
export OS_AUTH_URL=http://auth.example.com:5000/v3
export MURANO_URL=http://murano.example.com:8082/
```

2. In the terminal, source the `PROJECT-openrc.sh` file. For example:

```
$ . admin-openrc.sh
```

Once you have configured your authentication parameters, run **murano help** to see a complete list of available commands and arguments. Use **murano help <sub_command>** to get help on a specific sub-command.

See also:

Set environment variables using the OpenStack RC file.

Bash completion

To get the latest bash completion script, download `murano.bash_completion` from the source repository and add it to your completion scripts.

If you are not aware of the completion scripts location, perform the following steps:

1. Create a new directory:

```
$ mkdir -p ~/.bash_completion/
```

2. Create a file containing the bash completion script:

```
$ curl https://opendev.org/openstack/python-muranoclient/raw/branch/
↪master/tools/murano.bash_completion > ~/.bash_completion/murano.sh
```

3. Add the following code to the `~/.profile` file:

```
for file in $HOME/.bash_completion/*.sh; do
    if [ -f "$file" ]; then
        . "$file"
    fi
done
```

4. In the current terminal, run:

```
$ . ~/.bash_completion/murano.sh
```

Manage environments

An environment is a set of logically connected applications that are grouped together for an easy management. By default, each environment has a single network for all its applications, and the deployment of the environment is defined in a single heat stack. Applications in different environments are always independent from one another.

An environment is a single unit of deployment. This means that you deploy not an application but an environment that contains one or multiple applications.

Using CLI, you can easily perform such actions with an environment as creating, renaming, editing, viewing, and others.

Create an environment

To create an environment, use the following command specifying the environment name:

```
$ murano environment-create <NAME>
```

Rename an environment

To rename an environment, use the following command specifying the old name of the environment or its ID and the new name:

```
$ murano environment-rename <OLD_NAME_OR_ID> <NEW_NAME>
```

Delete an environment

To delete an environment, use the following command specifying the environment name or ID:

```
$ murano environment-delete <NAME_OR_ID>
```

List deployments for an environment

To get a list of deployments for a particular environment, use the following command specifying the environment name or ID:

```
$ murano deployment-list <NAME_OR_ID>
```

List the environments

To get a list of all existing environments, run:

```
$ murano environment-list
```

Show environment object model

To get a complete object model of the environment, run:

```
$ murano environment-model-show <ID>
```

To get some part of the environment model, run:

```
$ murano environment-model-show <ID> --path <PATH>
```

For example:

```
$ murano environment-model-show 534bcf2f2fc244f2b94ad55ff0f24a42 --path /default-  
Networks/environment
```

To get a draft of an object model of environment in pending state, also specify id of the session:

```
$ murano environment-model-show <ID> --path <PATH> --session-id <SESSION_ID>
```

Edit environment object model

To edit an object model of the environment, run:

```
$ murano environment-model-edit <ID> <FILE> --session-id <SESSION_ID>
```

<FILE> is the path to the file with the JSON-patch to modify the object model.

JSON-patch is a valid JSON that contains a list of changes to be applied to the current object. Each change contains a dictionary with three keys: `op`, `path` and `value`. `op` (operation) can be one of the three values: `add`, `replace` or `remove`.

Allowed operations for paths:

- "" (model root): no operations
- "defaultNetworks": "replace"
- "defaultNetworks/environment": "replace"
- "defaultNetworks/environment/?/id": no operations
- "defaultNetworks/flat": "replace"
- "name": "replace"
- "region": "replace"
- "?/type": "replace"
- "?/id": no operations

For other paths any operation (add, replace or remove) is allowed.

Example of JSON-patch:

```
[{
  "op": "replace",
  "path": "/defaultNetworks/flat",
  "value": true
}]
```

The patch above changes the value of the `flat` property of the environment's `defaultNetworks` property to `true`.

Manage packages

This section describes how to manage packages using the command line interface. You can easily:

- *import a package or bundles of packages*
- *list the existing packages*
- *display details for a package*
- *download a package*
- *delete a package*
- *create a package*

Import a package

With the **package-import** command you can import packages into murano in several different ways:

- *from a local .zip file*
- *from murano app repository*
- *from an http URL*

From a local .zip file

To import a package from a local .zip file, run:

```
$ murano package-import /path/to/PACKAGE.zip
```

where `PACKAGE` is the name of the package stored on your computer.

For example:

```
$ murano package-import /home/downloads/mysql.zip
Importing package com.example.databases.MySql
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
↪-----+-----+
| ID                | Name | FQN                |
↪Author            | Is Public|
```

(continues on next page)

(continued from previous page)

```
↔-----+-----+
| 83e4038885c248e3a758f8217ff8241f| MySQL| com.example.databases.MySql|↵
↔Mirantis, Inc|           |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
↔-----+-----+↵
```

To make the package available for users from other projects (tenants), use the `--is-public` parameter. For example:

```
$ murano package-import --is-public mysql.zip
```

Note: The `package-import` command supports multiple positional arguments. This means that you can import several packages at once.

From murano app repository

To import a package from murano applications repository, specify the URL of the repository with `--murano-repo-url` and a fully qualified package name. For package names, go to [Packages](#), and click on the desired package to see its full name.

Note: You can also specify the URL of the repository with the corresponding `MURANO_REPO_URL` environment variable.

The following example shows how to import the MySQL package from the murano applications repository:

```
$ murano --murano-repo-url=http://storage.apps.openstack.org \
package-import com.example.databases.MySql
```

This command supports an optional `--package-version` parameter that instructs murano client to download a specified package version.

The `package-import` command inspects package requirements specified in the package's manifest under the *Require* section, and attempts to import them from murano repository. The `package-import` command also inspects any image prerequisites mentioned in the `images.lst` file in the package. If there are any image requirements, client would inspect images already present in the image database. Unless image with the specific name is present, client would attempt to download it.

If any of the packages being installed is already registered in murano, the client asks you what to do with it. You can specify the default action with `--exists-action`, passing `s` - for skip, `u` - for update, and `a` - for abort.

From an URL

To import an application package from an URL, use the following command:

```
$ murano package-import http://example.com/path/to/PACKAGE.zip
```

The example below shows how to import a MySQL package from the murano applications repository using the package URL:

```

$ murano package-import http://storage.apps.openstack.org/apps/com.example.
↳databases.MySql.zip
Inspecting required images
Importing package com.example.databases.MySql
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
↳+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| ID                                     | Name   | FQN                                     |
↳| Author           | Active | Is Public| Type           |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
↳+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1aa62196595f411399e4e48cc2f6a512 | MySQL | com.example.databases.MySql |
↳| Mirantis, Inc | True   |           | Application|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
↳+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

Import bundles of packages

With the **bundle-import** command you can install packages in several different ways:

- *from a local bundle*
- *from an URL*
- *from murano app repository*

When importing bundles, you can set their publicity with `--is-public`.

From a local bundle

To import a bundle from the a local file system, use the following command:

```
$ murano bundle-import /path/to/bundle/BUNDLE_NAME
```

This command imports all the requirements of packages and images.

When importing a bundle from a file system, the murano client searches for packages in a directory relative to the bundle location before attempting to download a package from repository. This facilitates cases with no Internet access.

The following example shows the import of a monitoring bundle:

```

$ murano bundle-import /home/downloads/monitoring.bundle
Inspecting required images
Importing package com.example.ZabbixServer
Importing package com.example.ZabbixAgent
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
↳+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| ID                                     | Name           | FQN                                     |
↳| Author           | Active | Is Public| Type           |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
↳+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| fb0b35359e384fe18158ff3ed8f969b5 | Zabbix Agent  | com.example.ZabbixAgent |
↳| Mirantis, Inc | True   |           | Application|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
↳+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

(continues on next page)

(continued from previous page)

```
| 00a77e302a65420c8080dc97cc0f2723 | Zabbix Server | com.example.ZabbixServer.
↪| Mirantis, Inc | True | | Application|
+-----+-----+-----+-----+
↪+-----+-----+-----+-----+
```

Note: The `bundle-import` command supports multiple positional arguments. This means that you can import several bundles at once.

From an URL

To import a bundle from an URL, use the following command:

```
$ murano bundle-import http://example.com/path/to/bundle/BUNDLE_NAME
```

Where `http://example.com/path/to/bundle/BUNDLE_NAME` is any external http/https URL to load the bundle from.

For example:

```
$ murano bundle-import http://storage.apps.openstack.org/bundles/monitoring.
↪bundle
```

From murano applications repository

To import a bundle from murano applications repository, use the following command, where `bundle_name` stands for the bundle name:

```
$ murano bundle-import BUNDLE_NAME
```

For example:

```
$ murano bundle-import monitoring
```

Note: For bundle names, go to [Murano Applications Repository](#), click the **Format** tab to show bundles first, and then click on the desired bundle to see its name.

List packages

To list all the existing packages you have, use the `package-list` command. The result will show you the package ID, name, author and if it is public or not. For example:

```
$ murano package-list
+-----+-----+-----+-----+
↪-----+-----+-----+-----+
| ID | Name | FQN |
↪ | Author | Active | Is Public | Type |
+-----+-----+-----+-----+
↪-----+-----+-----+-----+
```

(continues on next page)

(continued from previous page)

```

| daa46cfd78c74c11bcbe66d3239e546e | Apache HTTP Server | com.example.apache.
↪ApacheHttpServer | Mirantis, Inc | True | | Application|
| 5252c9897e864c9f940e08500056f155 | Cloud Foundry | com.example.paas.
↪CloudFoundry | Mirantis, Inc | True | | Application|
| 1aa62196595f411399e4e48cc2f6a512 | MySQL | com.example.
↪databases.MySql | Mirantis, Inc | True | | Application|
| 11d73cfdc6d7447a910984d95090463b | SQL Library | com.example.
↪databases | Mirantis, Inc | True | | Application|
| fb0b35359e384fe18158ff3ed8f969b5 | Zabbix Agent | com.example.
↪ZabbixAgent | Mirantis, Inc | True | | Application|
| 00a77e302a65420c8080dc97cc0f2723 | Zabbix Server | com.example.
↪ZabbixServer | Mirantis, Inc | True | | Application|
+-----+-----+-----+-----+
↪-----+-----+-----+-----+

```

Show packages

To get full information about a package, use the **package-show** command. For example:

```

$ murano package-show 1aa62196595f411399e4e48cc2f6a512
+-----+-----+-----+-----+
| Property | Value |
+-----+-----+-----+-----+
| categories | | |
| class_definitions | com.example.databases.MySql |
| description | MySQL is a relational database management system |
| | | (RDBMS), and ships with no GUI tools to administer |
| | | MySQL databases or manage data contained within the |
| | | databases. |
| enabled | True |
| fully_qualified_name | com.example.databases.MySql |
| id | 1aa62196595f411399e4e48cc2f6a512 |
| is_public | False |
| name | MySQL |
| owner_id | 1ddb2c610d4e4c5dab5185e32554560a |
| tags | Database, MySql, SQL, RDBMS |
| type | Application |
+-----+-----+-----+-----+

```

Delete a package

To delete a package, use the following command:

```
$ murano package-delete PACKAGE_ID
```

Download a package

With the following command you can download a .zip archive with a specified package:

```
$ murano package-download PACKAGE_ID > FILE.zip
```

You need to specify the package ID and enter the .zip file name under which to save the package.

For example:

```
$ murano package-download e44a3f526dfb4e08b3c1018c9968d911 > Wordpress.zip
```

Create a package

With the murano client you can create application packages from package source files or directories. The **package-create** command is useful when application package files are spread across several directories. This command has the following required parameters:

```
-r RESOURCES_DIRECTORY  
-c CLASSES_DIRECTORY  
--type TYPE  
-o PACKAGE_NAME.zip  
-f FULL_NAME  
-n DISPLAY_NAME
```

Example:

```
$ murano package-create -c Downloads/Folder1/Classes -r Downloads/Folder2/  
↳Resources \  
-n mysql -f com.example.MySQL -d Package -o MySQL.zip --type Library  
Application package is available at /home/Downloads/MySQL.zip
```

After this, the package is ready to be imported to the application catalog.

The **package-create** command is also useful for autogenerating packages from heat templates. In this case you do not need to manually specify so many parameters. For more information on automatic package composition, please see [Automatic package composing](#).

Manage categories

In murano, applications can belong to a category or multiple categories. Administrative users can create and delete a category as well as list available categories and view details for a particular category.

Create a category

To create a category, use the following command specifying the category name:

```
$ murano category-create <NAME>
```

List available categories

To get a list of all existing categories, run:

```
$ murano category-list
```

Show category details

To see packages that belong to a particular category, use the following command specifying the category ID:

```
$ murano category-show <ID>
```

Delete a category

To delete a category, use the following command specifying the ID of a category or multiple categories to delete:

```
$ murano category-delete <ID> [<ID> ...]
```

Note: Verify that no packages belong to the category to be deleted, otherwise an error appears. For this, use the `murano category-show <ID>` command.

Manage environment templates

To manage environment templates, use the following commands specifying appropriate values:

```
murano env-template-create <ENV_TEMPLATE_NAME>
```

Creates an environment template.

```
murano env-template-clone <ID> <NEW_ENV_TEMPLATE_NAME>
```

Creates a new template, cloned from an existing template.

```
murano env-template-create-env <ID> <ENV_TEMPLATE_NAME>
```

Creates a new environment from template.

murano env-template-add-app <ENV_TEMPLATE_ID> <FILE>

Adds an application or multiple applications to the environment template.

murano env-template-del-app <ENV_TEMPLATE_ID> <ENV_TEMPLATE_APP_ID>

Deletes an application from the environment template.

murano env-template-list

Lists the environments templates.

murano env-template-show <ID>

Displays environment template details.

murano env-template-update <ID> <ENV_TEMPLATE_NAME>

Updates an environment template.

murano env-template-delete <ID>

Deletes an environment template.

See also:

[Application Catalog service command-line client.](#)

2.2.5 Deploying environments using CLI

The main tool for deploying murano environments is murano-dashboard. It is designed to be easy-to-use and intuitive. But it is not the only tool you can use to deploy a murano environment, murano CLI client also possesses required functionality for the task. This is an advanced scenario, however, that requires knowledge of *internal murano workflow*, *murano object model*, and *murano environment* lifecycle. This scenario is suitable for deployments without horizon or deployment automation.

Note: This is an advanced mechanism and you should use it only when you are confident in what you are doing. Otherwise, it is recommended that you use murano-dashboard.

Create an environment

The following command creates a new murano environment that is ready for configuration. For convenience, this guide refers to environment ID as \$ENV_ID.

```
$ murano environment-create deployed_from_cli

+-----+-----+-----+
↪+-----+
| ID                               | Name               | Created           |
↪| Updated                         |                    |                   |
+-----+-----+-----+
↪+-----+
| a66e5ea35e9d4da48c2abc37b5a9753a | deployed_from_cli | 2015-10-06T13:50:45 |
↪| 2015-10-06T13:50:45 |                    |                   |
+-----+-----+-----+
↪+-----+
```

Create a configuration session

Murano uses configuration sessions to allow several users to edit and configure the same environment concurrently. Most of environment-related commands require the `--session-id` parameter. For convenience, this guide refers to session ID as `$SESS_ID`.

To create a configuration session, use the `murano environment-session-create $ENV_ID` command:

```
$ murano environment-session-create $ENV_ID

+-----+-----+
| Property | Value |
+-----+-----+
| id       | 5cbe7e561ffc484ebf11aabf83f9f4c6 |
+-----+-----+
```

Add applications to an environment

To manipulate environments object model from CLI, use the `environment-apps-edit` command:

```
$ murano environment-apps-edit --session-id $SESS_ID $ENV_ID object_model_
↪patch.json
```

The `object_model_patch.json` contains the `jsonpatch` object. This object is applied to the `/services` key of the environment in question. Below is an example of the `object_model_patch.json` file content:

```
[
  { "op": "add", "path": "/-", "value":
    {
      "instance": {
        "availabilityZone": "nova",
        "name": "xwvupifdxq27t1",
        "image": "fa578106-b3c1-4c42-8562-4e2e2d2a0a0c",
        "keyname": "",
        "flavor": "m1.small",
        "assignFloatingIp": false,
        "?": {
          "type": "io.murano.resources.LinuxMuranoInstance",
          "id": "===id1==="
        }
      },
      "name": "ApacheHttpServer",
      "enablePHP": true,
      "?": {
        "type": "com.example.apache.ApacheHttpServer",
        "id": "===id2==="
      }
    }
  }
]
```

(continues on next page)

(continued from previous page)

```
}  
]
```

For convenience, the murano client replaces the "===id1===", "===id2===" (and so on) strings with UUIDs. This way you can ensure that object IDs inside your object model are unique. To learn more about jsonpatch, consult jsonpatch.com and [RFC 6902](https://tools.ietf.org/html/rfc6902). The **environment-apps-edit** command fully supports jsonpatch. This means that you can alter, add, or remove parts of your applications object model.

Verify your object model

To verify whether your object model is correct, check the environment by running the **environment-show** command with the `--session-id` parameter:

```
$ murano environment-show $ENV_ID --session-id $SESS_ID --only-apps  
  
[  
  {  
    "instance": {  
      "availabilityZone": "nova",  
      "name": "xwvupifdxq27t1",  
      "assignFloatingIp": false,  
      "keyname": "",  
      "flavor": "m1.small",  
      "image": "fa578106-b3c1-4c42-8562-4e2e2d2a0a0c",  
      "?: {  
        "type": "io.murano.resources.LinuxMuranoInstance",  
        "id": "fc4fe975f5454bab99bb0e309249e2d2"  
      }  
    },  
    "?: {  
      "status": "pending",  
      "type": "com.example.apache.ApacheHttpServer",  
      "id": "69cdf10d31e64196b4de894e7ea4f1be"  
    },  
    "enablePHP": true,  
    "name": "ApacheHttpServer"  
  }  
]
```

Deploy your environment

To deploy a session `$SESS_ID` of your environment, use the **murano environment-deploy** command:

```
$ murano environment-deploy $ENV_ID --session-id $SESS_ID
```

You can later use the **murano environment-show** command to track the deployment status.

To view the deployed applications of a particular environment, use the **murano environment-show** command with the `--only-apps` parameter and specifying the environment ID:

```
$ murano environment-show $ENV_ID --only-apps
```

2.2.6 Support for OpenStack regions

Murano supports multi-region deployment. If OpenStack setup has several regions it is possible to choose the region to deploy an application.

There is the new option in the murano configuration file:

- *home_region* - default region name used to get services endpoints. The region where murano-api resides.

Now murano has two possible ways to deploy apps in different regions:

1. Deploy an application in the current murano region.
2. Associate environments with regions.

Deploy an app in the current region

Each region has a copy of murano services and its own RabbitMQ for api to engine communication. In this case application will be deployed to the same region that murano run in.

See also:

Multi-region application

Associate environments with regions

Murano services are in one region but environments can be associated with different regions. There are two new properties in the class *io.murano.Environment*:

- *regionConfigs* - a dict with RabbitMQ settings for each region. The structure of the agentRabbitMq part of the dict is identical to [rabbitmq] section in the *murano.conf* file. For example:

```
regionConfigs:
  RegionOne:
    agentRabbitMq:
      host: 192.1.1.1
      login: admin
      password: admin
```

User can store such configs as YAML or JSON files. These config files must be stored in a special folder that is configured in [engine] section of *murano.conf* file under *class_configs* key and must be named using *%FQ class name%.json* or *%FQ class name%.yaml* pattern.

- *region* - region name to deploy an app. It can be passed when creating environment via CLI:

```
murano environment-create environment_name --region RegionOne
```

If it is not specified a value from *home_region* option of *murano.conf* file will be used.

INSTALLATION

3.1 Application Catalog service

3.1.1 Application Catalog service overview

The Application Catalog service consists of the following components:

murano command-line client

A CLI that communicates with the `murano-api` to publish various cloud-ready applications on new virtual machines.

murano-api service

An OpenStack-native REST API that processes API requests by sending them to the `murano-engine` service via AMQP.

murano-agent service

The agent that runs on guest VMs and executes the deployment plan, a combination of execution plan templates and scripts.

murano-engine service

The workflow component of Murano, responsible for the deployment of an environment.

murano-dashboard service

Murano UI implemented as a plugin for the OpenStack Dashboard.

3.1.2 Install and configure

This section describes how to install and configure the Application Catalog service, code-named `murano`, on the controller node.

This section assumes that you already have a working OpenStack environment with at least the following components installed: Identity service, Image service, Compute service, Networking service, Block Storage service and Orchestration service. See [OpenStack Install Guides](#).

Note that installation and configuration vary by distribution. Currently, this installation guide is tailored toward Ubuntu environments, but can easily be adapted to work with other types of distros.

Note: Fedora support wasn't thoroughly tested. We do not guarantee that `murano` will work on Fedora.

Install Murano API

This section describes how to install and configure the Application Catalog service for Ubuntu 16.04 (LTS).

Prerequisites

Before you install and configure the Application Catalog service, you must create a database, service credentials, and API endpoints.

1. To create the database, complete these steps:

Murano can use various database types on the back end. For development purposes, SQLite is enough in most cases. For production installations, you should use MySQL or PostgreSQL databases.

Warning: Although murano could use a PostgreSQL database on the back end, it wasn't thoroughly tested and should be used with caution.

- Use the database access client to connect to the database server as the `root` user:

```
$ mysql -u root -p
```

- Create the murano database:

```
CREATE DATABASE murano;
```

- Grant proper access to the murano database:

```
GRANT ALL PRIVILEGES ON murano.* TO 'murano'@'localhost' IDENTIFIED BY 'MURANO_DBPASS';
```

Replace `MURANO_DBPASS` with a suitable password.

- Exit the database access client.

```
exit;
```

2. Source the admin credentials to gain access to admin-only CLI commands:

```
$ . admin-openrc
```

3. To create the service credentials, complete these steps:

- Create the murano user:

```
$ openstack user create --domain default --password-prompt murano
```

- Add the `admin` role to the murano user:

```
$ openstack role add --project service --user murano admin
```

- Create the murano service entities:

```
$ openstack service create --name murano --description "Application_
↪Catalog" application-catalog
```

4. Create the Application Catalog service API endpoints:

```
$ openstack endpoint create --region RegionOne \
application-catalog public http://<murano-ip>:8082
$ openstack endpoint create --region RegionOne \
application-catalog internal http://<murano-ip>:8082
$ openstack endpoint create --region RegionOne \
application-catalog admin http://<murano-ip>:8082
```

Note: URLs (publicurl, internalurl and adminurl) may be different depending on your environment.

Install and configure components

1. Install the packages:

```
# apt-get update
# apt-get install murano-engine murano-api
```

2. Edit `murano.conf` with your favorite editor. Below is an example which contains basic settings you likely need to configure.

Note: The example below uses SQLite database. Edit **[database]** section if you want to use any other database type.

```
[DEFAULT]
debug = true
verbose = true
transport_url = rabbit://%RABBITMQ_USER%:%RABBITMQ_PASSWORD%@%RABBITMQ_
↪SERVER_IP%:5672/
...

[database]
connection = mysql+pymysql://murano:MURANO_DBPASS@controller/murano
...

[keystone]
auth_url = http://%OPENSTACK_KEYSTONE_ENDPOINT%
...
```

(continues on next page)

(continued from previous page)

```
[keystone_authtoken]
project_domain_name = Default
project_name = %OPENSTACK_ADMIN_PROJECT%
user_domain_name = Default
password = %OPENSTACK_ADMIN_PASSWORD%
username = %OPENSTACK_ADMIN_USER%
auth_url = http://%OPENSTACK_KEYSTONE_ENDPOINT%
auth_type = password

...

[murano]
url = http://%YOUR_HOST_IP%:8082

[rabbitmq]
host = %RABBITMQ_SERVER_IP%
login = %RABBITMQ_USER%
password = %RABBITMQ_PASSWORD%
virtual_host = %RABBITMQ_SERVER_VIRTUAL_HOST%

[networking]
default_dns = 8.8.8.8 # In case openstack neutron has no default
                  # DNS configured
```

3. Populate the Murano database:

```
# su -s /bin/sh -c "murano-db-manage upgrade" murano
```

Note: Ignore any deprecation messages in this output.

Finalize installation

1. Restart the Application Catalog services:

```
# service murano-api restart
# service murano-engine restart
```

Install Murano Dashboard

Murano API & Engine services provide the core of Murano. However, you need a control plane to use it. This section describes how to install and run Murano Dashboard.

1. Install OpenStack Dashboard, the steps please reference from [OpenStack Dashboard Install Guide](#).
2. Install the packages:

```
# apt install python-murano-dashboard
```

3. Edit the `/etc/openstack-dashboard/local_settings.py` file to customize local settings of your envi

```
...
OPENSTACK_HOST = '%OPENSTACK_HOST_IP%'
OPENSTACK_KEYSTONE_DEFAULT_ROLE = '%OPENSTACK_ROLE%'
...
```

Change the default session back end-from using browser cookies to using a database instead to avoid issues with forms during the creation of applications:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'murano-dashboard.sqlite',
    }
}

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

4. (Optional) If you do not plan to get the murano service from the keystone application catalog, specify where the murano-api service is running:

```
MURANO_API_URL = 'http://%MURANO_IP%:8082'
```

Finalize installation

1. Restart the Apache service:

```
# service apache2 restart
```

Install Murano from Source

This section describes how to install and configure the Application Catalog service for Ubuntu 16.04 (LTS) from source code.

Prerequisites

Before you install and configure the Application Catalog service, you must create a database, service credentials, and API endpoints.

1. To create the database, complete these steps:

Murano can use various database types on the back end. For development purposes, SQLite is enough in most cases. For production installations, you should use MySQL or PostgreSQL databases.

Warning: Although murano could use a PostgreSQL database on the back end, it wasn't thoroughly tested and should be used with caution.

- Use the database access client to connect to the database server as the `root` user:

```
$ mysql -u root -p
```

- Create the murano database:

```
CREATE DATABASE murano;
```

- Grant proper access to the murano database:

```
GRANT ALL PRIVILEGES ON murano.* TO 'murano'@'localhost' IDENTIFIED BY 'MURANO_DBPASS';
```

Replace `MURANO_DBPASS` with a suitable password.

- Exit the database access client.

```
exit;
```

2. Source the admin credentials to gain access to admin-only CLI commands:

```
$ . admin-openrc
```

3. To create the service credentials, complete these steps:

- Create the murano user:

```
$ openstack user create --domain default --password-prompt murano
```

- Add the admin role to the murano user:

```
$ openstack role add --project service --user murano admin
```

- Create the murano service entities:

```
$ openstack service create --name murano --description "Application_
↪Catalog" application-catalog
```

4. Create the Application Catalog service API endpoints:

```
$ openstack endpoint create --region RegionOne \
application-catalog public http://<murano-ip>:8082
$ openstack endpoint create --region RegionOne \
application-catalog internal http://<murano-ip>:8082
$ openstack endpoint create --region RegionOne \
application-catalog admin http://<murano-ip>:8082
```

Note: URLs (publicurl, internalurl and adminurl) may be different depending on your environment.

Install the API service and Engine

1. Create a folder which will hold all Murano components.

```
mkdir ~/murano
```

2. Clone the murano git repository to the management server.

```
cd ~/murano
git clone https://opendev.org/openstack/murano
```

3. Set up the murano config file

Murano has a common config file for API and Engine services.

First, generate a sample configuration file, using tox

```
cd ~/murano/murano
tox -e genconfig
```

And make a copy of it for further modifications

```
cd ~/murano/murano/etc/murano
ln -s murano.conf.sample murano.conf
```

4. Edit murano.conf with your favorite editor. Below is an example which contains basic settings you likely need to configure.

Note: The example below uses SQLite database. Edit **[database]** section if you want to use any other database type.

```
[DEFAULT]
debug = true
verbose = true
transport_url = rabbit://%RABBITMQ_USER%:%RABBITMQ_PASSWORD%@%RABBITMQ_
↪SERVER_IP%:5672/

...

[database]
connection = mysql+pymysql://murano:MURANO_DBPASS@controller/murano

...

[keystone]
auth_url = http://%OPENSTACK_KEYSTONE_ENDPOINT%

...

[keystone_authtoken]
project_domain_name = Default
project_name = %OPENSTACK_ADMIN_PROJECT%
user_domain_name = Default
password = %OPENSTACK_ADMIN_PASSWORD%
username = %OPENSTACK_ADMIN_USER%
auth_url = http://%OPENSTACK_KEYSTONE_ENDPOINT%
auth_type = password

...

[murano]
url = http://%YOUR_HOST_IP%:8082

[rabbitmq]
host = %RABBITMQ_SERVER_IP%
login = %RABBITMQ_USER%
password = %RABBITMQ_PASSWORD%
virtual_host = %RABBITMQ_SERVER_VIRTUAL_HOST%

[networking]
default_dns = 8.8.8.8 # In case openstack neutron has no default
                  # DNS configured
```

5. Create a virtual environment and install Murano prerequisites. We will use *tox* for that. The virtual environment will be created under *.tox* directory.

```
cd ~/murano/murano
```

(continues on next page)

(continued from previous page)

```
tox
```

6. Create database tables for Murano.

```
cd ~/murano/murano
tox -e venv -- murano-db-manage \
    --config-file ./etc/murano/murano.conf upgrade
```

7. Open a new console and launch Murano API. A separate terminal is required because the console will be locked by a running process.

```
cd ~/murano/murano
tox -e venv -- murano-api --config-file ./etc/murano/murano.conf
```

8. Import Core Murano Library.

```
cd ~/murano/murano
pushd ./meta/io.murano
zip -r ../../io.murano.zip *
popd
tox -e venv -- murano --murano-url http://localhost:8082 \
    package-import --is-public io.murano.zip
```

9. Open a new console and launch Murano Engine. A separate terminal is required because the console will be locked by a running process.

```
cd ~/murano/murano
tox -e venv -- murano-engine --config-file ./etc/murano/murano.conf
```

Install Murano Dashboard

Murano API & Engine services provide the core of Murano. However, you need a control plane to use it. This section describes how to install and run Murano Dashboard.

1. Clone the murano dashboard repository.

```
$ cd ~/murano
$ git clone https://opendev.org/openstack/murano-dashboard
```

2. Clone the horizon repository

```
$ git clone https://opendev.org/openstack/horizon
```

3. Create a virtual environment and install muranodashboard as an editable module:

```
$ cd horizon
$ tox -e venv -- pip install -e ../murano-dashboard
```

4. Prepare local settings.

```
$ cp openstack_dashboard/local/local_settings.py.example \
openstack_dashboard/local/local_settings.py
```

For more information, check out the official [horizon documentation](#).

5. Enable and configure Murano dashboard in the OpenStack Dashboard:

- For Newton (and later) OpenStack installations, copy the plugin file, local settings files, and policy files.

```
$ cp ../murano-dashboard/muranodashboard/local/enabled/*.py \
openstack_dashboard/local/enabled/

$ cp ../murano-dashboard/muranodashboard/local/local_settings.d/*.py \
↪ \
openstack_dashboard/local/local_settings.d/

$ cp ../murano-dashboard/muranodashboard/conf/* openstack_dashboard/
↪ conf/
```

- For the OpenStack installations prior to the Newton release, run:

```
$ cp ../murano-dashboard/muranodashboard/local/_50_murano.py \
openstack_dashboard/local/enabled/
```

Customize local settings of your horizon installation, by editing the `openstack_dashboard/local/local_settings.py` file:

```
...
ALLOWED_HOSTS = '*'

# Provide OpenStack Lab credentials
OPENSTACK_HOST = '%OPENSTACK_HOST_IP%'

...

DEBUG_PROPAGATE_EXCEPTIONS = DEBUG
```

Change the default session back end—from using browser cookies to using a database instead to avoid issues with forms during the creation of applications:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'murano-dashboard.sqlite',
    }
}

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

6. (Optional) If you do not plan to get the murano service from the keystone application catalog, specify where the murano-api service is running:

```
MURANO_API_URL = 'http://%MURANO_IP%:8082'
```

7. (Optional) If you have set up the database as a session back-end (this is done by default with the murano local_settings file starting with Newton), perform database migration:

```
$ tox -e venv -- python manage.py migrate --noinput
```

8. Run the Django server at 127.0.0.1:8000 or provide different IP and PORT parameters:

```
$ tox -e venv -- python manage.py runserver <IP:PORT>
```

Note: The development server restarts automatically following every code change.

Result: The murano dashboard is available at <http://IP:PORT>.

Network Configuration

Murano may work in various networking environments and is capable of detecting the current network configuration and choosing the appropriate settings automatically. However, some additional actions are required to support advanced scenarios.

Nova network support

Nova Network is the simplest networking solution, which has limited capabilities but is available on any OpenStack deployment without the need to deploy any additional components. For more information about Nova Network, see <https://docs.openstack.org/admin-guide/compute-networking-nova.html>.

When a new Murano Environment is created, Murano checks if a dedicated networking service (i.e. Neutron) exists in the current OpenStack deployment. It relies on Keystone's service catalog for that. If such a service is not present, Murano automatically falls back to Nova Network. No further configuration is needed in this case; all the VMs spawned by Murano will join the same network.

Neutron support

If Neutron is installed, Murano enables its advanced networking features that give you the ability to not care about configuring networks for your application.

By default, Murano will create an isolated network for each environment and attach all VMs needed by your application to that network. To install and configure applications in just-spawned virtual machines, Murano also requires a router connected to the external network.

Automatic Neutron network configuration

To create a router automatically, provide the following parameters in the config file:

```
[networking]
external_network = %EXTERNAL_NETWORK_NAME%
router_name = %MURANO_ROUTER_NAME%
create_router = true
```

SSL configuration

Murano components are able to work with SSL. This section will help you to configure proper settings for SSL configuration.

HTTPS for Murano API

SSL for the Murano API service can be configured in the *ssl* section in `/etc/murano/murano.conf`. Just point to a valid SSL certificate. See the example below:

```
[ssl]
cert_file = PATH
key_file = PATH
ca_file = PATH
```

- *cert_file* Path to the certificate file the server should use when binding to an SSL-wrapped socket.
- *key_file* Path to the private key file the server should use when binding to an SSL-wrapped socket.
- *ca_file* Path to the CA certificate file the server should use to validate client certificates provided during an SSL handshake. This is ignored if *cert_file* and "key_file" are not set.

Note: The use of SSL is automatically started after pointing to an HTTPS protocol instead of HTTP, during the registration of the Murano API service endpoints (Change `publicurl` argument to start with `https://`).

SSL for Murano API is implemented like in any other OpenStack component. This is because Murano uses the `ssl` python module; more information about it can be found [here](#).

SSL for RabbitMQ

All Murano components communicate with each other via RabbitMQ. This interaction can be encrypted with SSL. By default, all messages in Rabbit MQ are not encrypted. Each RabbitMQ Exchange should be configured separately.

Murano API <-> Rabbit MQ exchange <-> Murano Engine

Edit `ssl` parameters in default section of `/etc/murano/murano.conf`. Set the `rabbit_use_ssl` option to `true` and configure the `ssl kombu` parameters. Specify the path to the SSL keyfile and SSL CA certificate in a regular format: `/path/to/file` without quotes or leave it empty to allow for self-signed certificates.

```
# connect over SSL for RabbitMQ (boolean value)
#rabbit_use_ssl=false

# SSL version to use (valid only if SSL enabled). valid values
# are TLSv1, SSLv23 and SSLv3. SSLv2 may be available on some
# distributions (string value)
#kombu_ssl_version=

# SSL key file (valid only if SSL enabled) (string value)
#kombu_ssl_keyfile=

# SSL cert file (valid only if SSL enabled) (string value)
#kombu_ssl_certfile=

# SSL certification authority file (valid only if SSL enabled)
# (string value)
#kombu_ssl_ca_certs=
```

Murano Agent -> Rabbit MQ exchange

In the main murano configuration file, there is a section named *rabbitmq*, which is responsible for setting up communication between Murano Agent and Rabbit MQ. Just set the *ssl* parameter to True to enable ssl.

```
[rabbitmq]
host = localhost
port = 5672
login = guest
password = guest
virtual_host = /
ssl = True
```

If you want to configure Murano Agent in a different way, change the default template. It can be found in the Murano Core Library, located at <https://opendev.org/openstack/murano/src/branch/master/meta/io.murano/Resources/Agent-v1.template>. Take a look at the *appSettings* section:

```
<appSettings>
  <add key="rabbitmq.host" value="%RABBITMQ_HOST%" />
  <add key="rabbitmq.port" value="%RABBITMQ_PORT%" />
  <add key="rabbitmq.user" value="%RABBITMQ_USER%" />
  <add key="rabbitmq.password" value="%RABBITMQ_PASSWORD%" />
  <add key="rabbitmq.vhost" value="%RABBITMQ_VHOST%" />
  <add key="rabbitmq.inputQueue" value="%RABBITMQ_INPUT_QUEUE%" />
  <add key="rabbitmq.resultExchange" value="" />
  <add key="rabbitmq.resultRoutingKey" value="%RESULT_QUEUE%" />
  <add key="rabbitmq.durableMessages" value="true" />

  <add key="rabbitmq.ssl" value="%RABBITMQ_SSL%" />
  <add key="rabbitmq.allowInvalidCA" value="true" />
  <add key="rabbitmq.sslServerName" value="" />
```

(continues on next page)

(continued from previous page)

```
</appSettings>
```

The desired parameter should be set directly to the value of the key that you want to change. Quotes need to be kept. Thus you can change "rabbitmq.ssl" and "rabbitmq.port" values to make Rabbit MQ work with this exchange differently than the default Murano Engine way.

Note: After modification, don't forget to zip and re-upload the core library.

SSL for Murano Dashboard

If you are not going to use self-signed certificates, additional configuration does not need to be done. Just prefix https in the URL. Otherwise, set `MURANO_API_INSECURE = True` in Horizon's config file. You can find it in `/etc/openstack-dashboard/local_settings.py`.

3.1.3 Verify operation

Verify operation of the Application Catalog service.

Note: Perform these commands on the controller node.

1. Source the admin project credentials to gain access to admin-only CLI commands:

```
$ . admin-openrc
```

2. List service components to verify successful launch and registration of each process:

```
$ openstack service list | grep application-catalog  
| 7b12ef5edef848fc9200c271f71b1307 | murano          | application-catalog |
```

3.1.4 Next steps

Your OpenStack environment now includes the Murano service.

Import Murano Applications

Applications need to be imported to fill the catalog. This can be done via the dashboard or via CLI:

1. Clone the murano apps repository.

```
cd ~/murano  
git clone https://opendev.org/openstack/murano-apps
```

2. Import every package you need from this repository, using the command below.

```
cd ~/murano/murano
pushd ../murano-apps/Docker/Applications/%APP-NAME%/package
zip -r ~/murano/murano/app.zip *
popd
tox -e venv -- murano --murano-url http://<murano-ip>:8082 package-import.
↵ app.zip
```

Additional Resources

1. To add additional services, see <https://docs.openstack.org/latest/install/>.
2. If you would like to add glare as the storage service for packages, see: https://docs.openstack.org/murano/latest/admin/using_glare.html.

The Murano Project introduces an application catalog to OpenStack, enabling application developers and cloud administrators to publish various cloud-ready applications in a browsable categorized catalog. Cloud users -- including inexperienced ones -- can then use the catalog to compose reliable application environments with the push of a button.

This chapter assumes a working setup of OpenStack following the [OpenStack Installation Tutorial](#).

CONFIGURATION

4.1 Configuration Guide

5.1 Murano CLI Documentation

In this section you will find information on Murano's command line interface.

5.1.1 murano-status

CLI interface for Murano status commands

Synopsis

```
murano-status <category> <command> [<args>]
```

Description

murano-status is a tool that provides routines for checking the status of a Murano deployment.

Options

The standard pattern for executing a **murano-status** command is:

```
murano-status <category> <command> [<args>]
```

Run without arguments to see a list of available command categories:

```
murano-status
```

Categories are:

- upgrade

Detailed descriptions are below:

You can also run with a category argument such as **upgrade** to see a list of all commands in that category:

```
murano-status upgrade
```

These sections describe the available categories and arguments for **murano-status**.

Upgrade

murano-status upgrade check

Performs a release-specific readiness check before restarting services with new code. For example, missing or changed configuration options, incompatible object states, or other conditions that could lead to failures while upgrading.

Return Codes

Return code	Description
0	All upgrade readiness checks passed successfully and there is nothing to do.
1	At least one check encountered an issue and requires further investigation. This is considered a warning but the upgrade may be OK.
2	There was an upgrade status check failure that needs to be investigated. This should be considered something that stops an upgrade.
255	An unexpected error occurred.

History of Checks

7.0.0 (Stein)

- Sample check to be filled in with checks as they are added in Stein.

ADMINISTRATOR DOCUMENTATION

Learn how to manage images, categories, and repositories using the Murano client.

6.1 Deploying Murano

6.1.1 Deploying murano

System requirements

This section provides basic information about the murano environment system requirements. Additionally, it contains a description of the performance test scenario, which you may use to check if your hardware fits the requirements. To do this, run the test and compare the results with the baseline data provided.

Software prerequisites

Before you install murano, verify your system meets the following prerequisites.

Supported operating systems:

- Ubuntu Server
- RHEL/CentOS
- Debian

System packages for Ubuntu:

- gcc
- python3-pip
- python3-dev
- libxml2-dev
- libxslt-dev
- libffi-dev
- libpq-dev
- python3-openssl
- mysql-client

System packages for CentOS:

- gcc
- python3-pip
- python3-devel
- libxml2-devel
- libxslt-devel
- libffi-devel
- postgresql-devel
- pyOpenSSL
- mysql

Hardware requirements

We recommend that your system meets the following hardware requirements:

Criteria	Minimal	Recommended
CPU	4 core @ 2.4 GHz	24 core @ 2.67 GHz
RAM	8 GB	24 GB or more
HDD	2 x 500 GB (7200 rpm)	4 x 500 GB (7200 rpm)
RAID	Software RAID-1 (use mdadm as it improves the read performance almost twice)	Hardware RAID-10

Other possible storage configurations:

- 1x SSD 500+ GB
- 1x HDD (7200 rpm) 500+ GB and 1x SSD 250+ GB (install the system onto the HDD and mount the SSD drive to the directory where the virtual machines images are stored)
- 1x HDD (15000 rpm) 500+ GB

Testing the performance

We have measured the time required to boot 1 to 5 instances of the Windows operating system simultaneously. You can use this data as the baseline to check if your system is fast enough.

Note: Use *sysprepped* images for this test to simulate an instance first boot.

To reproduce the performance test, proceed with the following steps:

1. Prepare a Windows 2012 Standard (with GUI) image in the QCOW2 format. This example uses the `ws-2012-std.qcow2` image.

- Verify that there are no KVM processes running on the host:

```
ps aux | grep kvm
```

- Make 5 copies of the Windows image file:

```
for i in $(seq 5); do \  
cp ws-2012-std.qcow2 ws-2012-std-$i.qcow2; done
```

- Create the `start-vm.sh` script in the directory with the `.qcow2` files:

```
#!/bin/bash  
[ -z $1 ] || echo "VM count not provided!"; exit 1  
for i in $(seq $1); do  
echo "Starting VM $i ..."  
kvm -m 1024 -drive file=ws-2012-std-$i.qcow2,if=virtio -net user -net nic,  
↪model=virtio -nographic -usbdevice tablet -vnc :$i & done
```

- Start ONE instance using the command below (as root) and measure time between the instance launch and the moment when the Server Manager window displays.

```
sudo ./start-vm.sh 1
```

To view the instance desktop, connect with VNC viewer to your host to VNC screen :1 (port 5901).

- Turn off the instance. You may simply kill all KVM processes by running:

```
sudo killall kvm
```

- Start FIVE instances with the command below (as root) and measure time interval between ALL instances launch and the moment when the LAST Server Manager window displays.

```
sudo ./start-vm.sh 5
```

To view VM's desktops, connect with VNC viewer to your host to VNC screens :1 thru :5 (ports 5901-5905).

- Turn off the instances. You may simply kill all KVM processes by running:

```
sudo killall kvm
```

Baseline data

The table below provides the baseline data that was received in our test murano environment.

	Boot ONE instance	Boot FIVE instances
Avg. Time	3m:40s	8m
Max. Time	5m	20m

Avg. Time

Refers to the environment with the recommended hardware configuration

Max. Time

Refers to the minimal hardware configuration

Host optimizations

You can improve your default KVM installation performance with the following optimizations up to 30%:

- Change the default scheduler from **CFQ** to **Deadline**
- Use **ksm**
- Use **vhost-net**

Integrate murano with DevStack

You can install murano with DevStack. The `murano/devstack` directory in the murano repository contains the files necessary to integrate murano with DevStack.

To install the development version of an OpenStack environment with murano, proceed with the following steps:

1. Download DevStack:

```
git clone https://opendev.org/openstack/devstack
cd devstack
```

2. Edit `local.conf` to enable murano DevStack plug-in:

```
> cat local.conf
[[local|localrc]]
enable_plugin murano https://opendev.org/openstack/murano
```

3. If you want to enable Murano Cloud Foundry Broker API service, add the following line to `local.conf`:

```
enable_service murano-cfapi
```

4. If you want to use Glare Artifact Repository as a storage for packages, add the following line to `local.conf`:

```
enable_service g-glare
```

For more information on how to use Glare Artifact Repository, see *Using Glare as a storage for packages*.

5. (Optional) To import murano packages when DevStack is up, define an ordered list of FQDN packages in `local.conf`. Verify that you list all package dependencies. These packages will be imported from the `murano-apps` git repository by default. For example:

```
MURANO_APPS=com.example.apache.Tomcat,com.example.Guacamole
```

To configure the git repository that will be used as the source for the imported packages, configure the `MURANO_APPS_REPO` and `MURANO_APPS_BRANCH` variables.

6. Run DevStack:

```
./stack.sh
```

Result: Murano has installed with DevStack.

Install murano manually

Before you install Murano, verify that you completed the following tasks:

1. Install software prerequisites depending on the operating system you use as described in the System prerequisites section.
2. Install tox:

```
sudo pip install tox
```

3. Install and configure a database.

Murano can use various database types on back end. For development purposes, use SQLite. For production installations, consider using MySQL database.

Warning: Murano supports PostgreSQL as well. Though, use it with caution as it has not been thoroughly tested yet.

Before you can use MySQL database, proceed with the following:

1. Install MySQL:

```
apt-get install mysql-server
```

2. Create an empty database:

Replace `%MURANO_DB_PASSWORD%` with the actual password. For example, 'admin'.

```
mysql -u root -p

mysql> CREATE DATABASE murano;
mysql> GRANT ALL PRIVILEGES ON murano.* TO 'murano'@'localhost' \
IDENTIFIED BY %MURANO_DB_PASSWORD%;
mysql> exit;
```

Install the API service and engine

1. Create a folder to which all murano components will be stored:

```
mkdir ~/murano
```

2. Clone the murano git repository to the management server:

```
cd ~/murano
git clone https://opendev.org/openstack/murano
```

3. Create the configuration file. Murano has a common configuration file for API and engine services.

1. Generate a sample configuration file using tox:

```
cd ~/murano/murano
tox -e genconfig
```

2. Create a copy of `murano.conf` for further modifications:

```
cd ~/murano/murano/etc/murano
cp murano.conf.sample murano.conf
```

4. Edit the `murano.conf` file. An example below contains the basic configuration.

Note: The example uses MySQL database. If you want to use another database type, edit the `[database]` section correspondingly.

Replace items in "`%`" with the actual values. For example, replace `%RABBITMQ_SERVER_IP%` with `127.0.0.1`. So, the complete row with the replaced value will be `rabbit_host = 127.0.0.1`

```
[DEFAULT]
debug = true
verbose = true
rabbit_host = %RABBITMQ_SERVER_IP%
rabbit_userid = %RABBITMQ_USER%
rabbit_password = %RABBITMQ_PASSWORD%
rabbit_virtual_host = %RABBITMQ_SERVER_VIRTUAL_HOST%
...

[database]
connection = mysql+pymysql://murano:%MURANO_DB_PASSWORD%@127.0.0.1/murano
...

[keystone]
auth_url = 'http://%OPENSTACK_HOST_IP%:5000'
...

[keystone_authtoken]
www_authenticate_uri = 'http://%OPENSTACK_HOST_IP%:5000'
auth_host = '%OPENSTACK_HOST_IP%'
auth_port = 5000
auth_protocol = http
admin_tenant_name = %OPENSTACK_ADMIN_TENANT%
admin_user = %OPENSTACK_ADMIN_USER%
admin_password = %OPENSTACK_ADMIN_PASSWORD%
...
```

(continues on next page)

(continued from previous page)

```
[murano]
url = http://%YOUR_HOST_IP%:8082

[rabbitmq]
host = %RABBITMQ_SERVER_IP%
login = %RABBITMQ_USER%
password = %RABBITMQ_PASSWORD%
virtual_host = %RABBITMQ_SERVER_VIRTUAL_HOST%

[networking]
default_dns = 8.8.8.8 # In case OpenStack neutron has no default
                  # DNS configured

[oslo_messaging_notifications]
driver = messagingv2
```

5. Create a virtual environment and install murano prerequisites using **tox**. The virtual environment will be created under the **tox** directory.

1. Install MySQL driver since it is not a part of the murano requirements:

```
tox -e venv -- pip install PyMySQL
```

2. Create database tables for murano:

```
cd ~/murano/murano
tox -e venv -- murano-db-manage \
--config-file ./etc/murano/murano.conf upgrade
```

3. Launch the murano API in a separate terminal:

```
cd ~/murano/murano
tox -e venv -- murano-api --config-file ./etc/murano/murano.conf
```

Note: Run the command in a new terminal as the process will be running in the terminal until you terminate it, therefore, blocking the current terminal.

4. Leaving the API process running, return to the previous console and import murano core library and other libraries from the *meta* directory:

```
cd ~/murano/murano/meta/
for i in */; do pushd ./"$i"; zip -r ../../"${i%}/.zip" *; popd; done
cd ..
tox -e venv -- murano --os-username %OPENSTACK_ADMIN_USER% \
--os-password %OPENSTACK_ADMIN_PASSWORD% \
--os-auth-url http://%OPENSTACK_HOST_IP%:5000 \
--os-project-name %OPENSTACK_ADMIN_TENANT% \
--murano-url http://%MURANO_IP%:8082 \
package-import --is-public *.zip
rm *.zip
```

5. Launch the murano engine in a separate terminal:

```
cd ~/murano/murano
tox -e venv -- murano-engine --config-file ./etc/murano/murano.conf
```

Note: Run the command in a new terminal as the process will be running in the terminal until you terminate it, therefore, blocking the current terminal.

Register in keystone

To make the murano API available to all OpenStack users, you need to register the Application Catalog service within the Identity service.

1. Add the application-catalog service to keystone:

```
openstack service create --name murano --description \
"Application Catalog for OpenStack" application-catalog
```

2. Provide an endpoint for this service:

```
openstack endpoint create --region RegionOne --publicurl 'http://%MURANO_
↪IP%:8082/' \
--adminurl 'http://%MURANO_IP%:8082/' --internalurl 'http://%MURANO_IP
↪%:8082/' \
%MURANO_SERVICE_ID%
```

where MURANO-SERVICE-ID is the unique service number that can be found in the **openstack service create** output.

Note: URLs (--publicurl, --internalurl, and --adminurl values) may differ depending on your environment.

Install the murano dashboard

This section describes how to install and run the murano dashboard.

1. Clone the repository with the murano dashboard:

```
cd ~/murano
git clone https://opendev.org/openstack/murano-dashboard
```

2. Clone the horizon repository:

```
git clone https://opendev.org/openstack/horizon
```

3. Create a virtual environment and install muranodashboard as an editable module:

```
cd horizon
tox -e venv -- pip install -e ../murano-dashboard
```

4. Prepare local settings.

```
cp openstack_dashboard/local/local_settings.py.example \
openstack_dashboard/local/local_settings.py
```

For more information, check out the official [horizon documentation](#).

5. Enable and configure Murano dashboard in the OpenStack Dashboard:

- For the Newton (and later) OpenStack installations, copy plug-in file local settings files, and policy files:

```
cp ../murano-dashboard/muranodashboard/local/enabled/*.py \
openstack_dashboard/local/enabled/

cp ../murano-dashboard/muranodashboard/local/local_settings.d/*.py \
openstack_dashboard/local/local_settings.d/

cp ../murano-dashboard/muranodashboard/conf/* openstack_dashboard/
↪conf/
```

- For the OpenStack installations prior to the Newton release, run:

```
cp ../murano-dashboard/muranodashboard/local/_50_murano.py \
openstack_dashboard/local/enabled/
```

Customize local settings of your horizon installation, by editing the `openstack_dashboard/local/local_settings.py` file:

```
...
ALLOWED_HOSTS = '*'

# Provide your OpenStack Lab credentials
OPENSTACK_HOST = '%OPENSTACK_HOST_IP%'

...

DEBUG_PROPAGATE_EXCEPTIONS = DEBUG
```

Change the default session back end from browser cookies to database to avoid issues with forms during the applications creation:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'murano-dashboard.sqlite',
    }
}

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

- (Optional) If you do not plan to get the murano service from the keystone application catalog, specify where the murano-api service is running:

```
MURANO_API_URL = 'http://%MURANO_IP%:8082'
```

- (Optional) If you have set up the database as a session back end (this is done by default with murano local_settings file starting with Newton), perform database migration:

```
tox -e venv -- python manage.py migrate --noinput
```

Since a separate user is not required for development purpose, you can reply no.

- Run Django server at 127.0.0.1:8000 or provide a different IP and PORT parameters:

```
tox -e venv -- python manage.py runserver <IP:PORT>
```

Note: The development server restarts automatically on every code change.

Result: The murano dashboard is available at `http://IP:PORT`.

Import murano applications

To fill the application catalog, you need to import applications to your OpenStack environment. You can import applications using the murano dashboard, as well as the command-line client.

To import applications using CLI, complete the following tasks:

- Clone the murano apps repository:

```
cd ~/murano
git clone https://opendev.org/openstack/murano-apps
```

- Import every package you need from this repository by running the following command:

```
cd ~/murano/murano
pushd ../murano-apps/Docker/Applications/%APP-NAME%/package
zip -r ~/murano/murano/app.zip *
popd
tox -e venv -- murano --murano-url http://%MURANO_IP%:8082 package-import.
↵ app.zip
```

Result: The applications are imported and available from the application catalog.

Configure SSL

Murano components can work with SSL. This section provides information on how to set SSL properly.

Configure SSL for Murano API

To configure SSL for the Murano API service, modify the `[ssl]` section in `/etc/murano/murano.conf`:

```
[ssl]
cert_file = <PATH>
key_file = <PATH>
ca_file = <PATH>
```

Parameter	Description
<code>cert_file</code>	A path to the certificate file the server should use when binding to an SSL-wrapped socket.
<code>key_file</code>	A path to the private key file the server should use when binding to an SSL-wrapped socket.
<code>ca_file</code>	A path to the CA certificate file the server should use to validate client certificates provided during an SSL handshake. This parameter is ignored if the <code>cert_file</code> and <code>key_file</code> parameters are not set.

Murano API starts using SSL automatically after you point to the HTTPS protocol instead of HTTP during the registration of the Murano API service in endpoints, modifying the `publicurl` argument to start with `https://`.

SSL for Murano API is implemented the same way as in any other OpenStack component. See [ssl python module](#) for details.

Configure SSL for RabbitMQ

All murano components communicate with each other using RabbitMQ. By default, all messages in RabbitMQ are not encrypted. You can encrypt this interaction with SSL. Configure each RabbitMQ exchange separately.

Murano API <-> RabbitMQ <-> Murano engine

Modify the `[default]` section in the `/etc/murano/murano.conf` file:

1. Enable SSL for RabbitMQ:

```
# connect over SSL for RabbitMQ (boolean value)
rabbit_use_ssl = true
```

2. Set the kombu parameters.

Specify the paths to the SSL key file and SSL CA certificate in a regular `</PATH/TO/FILE>` format without quotes or leave them empty to enable self-signed certificates:

```
# SSL version to use (valid only if SSL enabled). valid values
# are TLSv1, SSLv23 and SSLv3. SSLv2 may be available on some
# distributions (string value)
kombu_ssl_version =

# SSL key file (valid only if SSL enabled) (string value)
kombu_ssl_keyfile =

# SSL cert file (valid only if SSL enabled) (string value)
kombu_ssl_certfile =

# SSL certification authority file (valid only if SSL enabled)
# (string value)
kombu_ssl_ca_certs =
```

Murano agent -> RabbitMQ

To encrypt the communication between the murano agent and RabbitMQ, set `ssl = True` in the `[rabbitmq]` section of `/etc/murano/murano.conf`:

```
[rabbitmq]
...
ssl = True
insecure = False
```

If you want to configure the murano agent differently, you need to change the [default template](#) located in the murano core library. After you finish with the template modification, verify that you zip and re-upload the murano core library.

Configure SSL for the Dashboard

If you do not plan to use self-signed certificates, no additional configurations are required. Just point your web browser to the URL starting with `https://`.

Otherwise, set the `MURANO_API_INSECURE` parameter to `True` in `/etc/openstack-dashboard/local_settings.py`.

6.1.2 Prepare a lab for murano

This section provides basic information about lab's system requirements. It also contains a description of a test which you may use to check if your hardware fits the requirements. To do this, run the test and compare the results with baseline data provided.

System prerequisites

Supported operating systems

- Ubuntu Server 16.04 LTS or higher
- RHEL/CentOS 7.4 or higher

System packages are required for Murano

Ubuntu

- gcc
- python3-pip
- python3-dev
- libxml2-dev
- libxslt-dev
- libffi-dev
- libpq-dev
- python3-openssl
- mysql-client

Install all the requirements on Ubuntu by running:

```
sudo apt-get install gcc python3-pip python3-dev \  
libxml2-dev libxslt-dev libffi-dev \  
libpq-dev python3-openssl mysql-client
```

CentOS

- gcc
- python3-pip
- python3-devel
- libxml2-devel
- libxslt-devel
- libffi-devel
- postgresql-devel
- pyOpenSSL
- mysql

Install all the requirements on CentOS by running:

```
sudo yum install gcc python3-pip python3-devel libxml2-devel \  
libxslt-devel libffi-devel postgresql-devel pyOpenSSL \  
mysql
```

Lab requirements

Criteria	Minimal	Recommended
CPU	4 core @ 2.4 GHz	24 core @ 2.67 GHz
RAM	8 GB	24 GB or more
HDD	2 x 500 GB (7200 rpm)	4 x 500 GB (7200 rpm)
RAID	Software RAID-1 (use mdadm as it will improve read performance almost two times)	Hardware RAID-10

Table: Hardware requirements

There are a few possible storage configurations except the shown above. All of them were tested and were working well.

- 1x SSD 500+ GB
- **1x HDD (7200 rpm) 500+ GB and 1x SSD 250+ GB (install the system onto the HDD and mount the SSD drive to folder where VM images are)**
- 1x HDD (15000 rpm) 500+ GB

Test your lab host performance

We have measured time required to boot 1 to 5 instances of Windows system simultaneously. You can use this data as the baseline to check if your system is fast enough.

You should use sysprepped images for this test, to simulate VM first boot.

Steps to reproduce test:

1. Prepare Windows 2012 Standard (with GUI) image in QCOW2 format. Let's assume that its name is ws-2012-std.qcow2
2. Ensure that there is NO KVM PROCESSES on the host. To do this, run command:

```
ps aux | grep kvm
```

3. Make 5 copies of Windows image file:

```
for i in $(seq 5); do \
cp ws-2012-std.qcow2 ws-2012-std-$i.qcow2; done
```

4. Create script start-vm.sh in the folder with .qcow2 files:

```
#!/bin/bash
[ -z $1 ] || echo "VM count not provided!"; exit 1
for i in $(seq $1); do
echo "Starting VM $i ..."
kvm -m 1024 -drive file=ws-2012-std-$i.qcow2,if=virtio -net user -net nic,
↪model=virtio -nographic -usbdevice tablet -vnc :$i & done
```

5. Start ONE instance with command below (as root) and measure time between VM's launch and the moment when Server Manager window appears. To view VM's desktop, connect with VNC viewer to your host to VNC screen :1 (port 5901):

```
sudo ./start-vm.sh 1
```

6. Turn VM off. You may simply kill all KVM processes by

```
sudo killall kvm
```

7. Start FIVE instances with command below (as root) and measure time interval between ALL VM's launch and the moment when LAST Server Manager window appears. To view VM's desktops, connect with VNC viewer to your host to VNC screens :1 thru :5 (ports 5901-5905):

```
sudo ./start-vm.sh 5
```

8. Turn VMs off. You may simply kill all KVM processes by

```
sudo killall kvm
```

Baseline data

The table below provides baseline data which we've got in our environment.

	Boot 1 instance	Boot 5 instances
Avg. Time	3m:40s	8m
Max. Time	5m	20m

Avg. Time refers to the lab with recommended hardware configuration, while Max. Time refers to minimal hardware configuration.

Host optimizations

Default KVM installation could be improved to provide better performance.

The following optimizations may improve host performance up to 30%:

- change default scheduler from CFQ to Deadline
- use ksm
- use vhost-net

6.1.3 Murano Policies

Murano only uses 2 roles for policy enforcement. Murano allows access by default and uses the admin role for any action that involves accessing data across multiple projects in the cloud.

role:Member

User is non-admin to all APIs.

role:admin

User is admin to all APIs.

Sample File Generation

To generate a sample policy.yaml file from the Murano defaults, run the oslo policy generation script:

```
oslopolicy-sample-generator \  
--config-file etc/oslo-policy-generator/murano-policy-generator.conf \  
--output-file policy.yaml.sample
```

or using tox:

```
tox -egenpolicy
```

Note: In previous OpenStack releases the default policy format was JSON, but now the [recommended format](#) is YAML.

Merged File Generation

This will output a policy file which includes all registered policy defaults and all policies configured with a policy file. This file shows the effective policy in use by the project:

```
oslopolicy-sample-generator \  
--config-file etc/oslo-policy-generator/murano-policy-generator.conf
```

List Redundant Configurations

This will output a list of matches for policy rules that are defined in a configuration file where the rule does not differ from a registered default rule. These are rules that can be removed from the policy file with no change in effective policy:

```
oslopolicy-list-redundant \  
--config-file etc/oslo-policy-generator/murano-policy-generator.conf
```

Policy configuration

Like each service in OpenStack, Murano has its own role-based access policies that determine who can access objects and under what circumstances. The default implementation for these policies is defined in the service's source code -- under `murano.common.policies`. The default policy definitions can be overridden using the `policy.yaml` file.

On each API call the corresponding policy check is performed. `policy.yaml` file can be changed without interrupting the API service.

For detailed information on `policy.yaml` syntax, please refer to the [OpenStack official documentation](#)

With this file you can set who may upload packages and perform other operations.

So, changing `"upload_package": "rule:default"` to `"rule:admin_api"` will forbid regular users from uploading packages.

For reference:

- `"get_package"` is checked whenever a user accesses a package from the catalog. default: anyone
- `"upload_package"` is checked whenever a user uploads a package to the catalog. default: anyone
- `"modify_package"` is checked whenever a user modifies a package in the catalog. default: anyone
- `"publicize_package"` is checked whenever a user is trying to make a murano package public (both when creating a new package or modifying an existing one). default: admin users
- `"manage_public_package"` is checked whenever a user attempts to modify parameters of a public package. default: admin users
- `"delete_package"` is checked whenever a user attempts to delete a package from the catalog. default: anyone
- `"download_package"` is checked whenever a user attempts to download a package from the catalog. default: anyone
- `"list_environments_all_tenants"` is checked whenever a request to list environments of all tenants is made. default: admin users
- `"execute_action"` is checked whenever a user attempts to execute an action on deployment environments. default: anyone

Note: The package upload wizard in Murano dashboard consists of several steps: The `"upload_package"` policy is enforced during the first step while `"modify_package"` is enforced during the second step. Package parameters are modified during package upload. So, please modify both policy definitions together. Otherwise it will not be possible to browse package details on the second step of the wizard.

Default Murano Policies

6.1.4 Managing packages

Managing packages on engine side

To get access to the contents of murano packages, `murano-engine` queries `murano-api`. However, it is also possible to specify a list of directories that may contain packages locally. This option is useful to speed up debugging and development of packages and/or to save bandwidth between the API and the engine. If local directories are specified, they are examined before querying the API.

Local package directories

To define a list of directories where the engine would look for package files, set the `load_packages_from` option in the `engine` section of the `murano.conf` configuration file. This option can be set to a comma-separated list of directory paths. Whenever an engine needs to access a package, it would inspect these directories first, before accessing `murano-api`.

API package cache

If the package was not found in any of the `load_packages_from` directories, or if none were specified, then `murano-engine` queries API for package contents. Whenever `murano-engine` downloads a package from API, it stores and unpacks it locally. The engine uses the directory defined in the `packages_cache` option in the `engine` section of the `murano.conf` configuration file. If it is not used, a temporary directory is created.

The `enable_packages_cache` option in the same section defines whether the packages would persist on disk or not. When set to `False`, each package downloaded from API is stored in a separate directory, that will be deleted after the deployment (or action) is over. This means that every deployment or action execution needs to download all the packages it requires, regardless of any packages previously downloaded by the engine. When set to `True` (default), the engine shares downloaded packages between deployments and action executions. This means that packages persist on disk and have to be eventually deleted. Therefore, whenever the engine requires a package and that package is not found locally, the engine downloads the package. Afterwards, it checks all the previously cached packages with the same FQN and same version. If the cached package is not required by any ongoing deployment, it gets deleted. Otherwise, it stays on disk until a new version is downloaded.

Note: On UNIX-based operating systems, `fcntl` for IPC locks that support both shared and exclusive locking. On Windows, `msvcrt` is used. It does not support shared file locks. Therefore, enabling package cache mechanism under Windows might result in performance decrease, since only one process would be able to use one package at the same time.

6.1.5 Managing images

Build an image

Manage images

6.1.6 Managing categories

6.1.7 Murano repository

Use an existing repository

Set up a custom repository

6.1.8 Murano agent

Murano easily installs and configures necessary software on new virtual machines. Murano agent is one of the main participants of these processes.

Usually, it is enough to execute a single script to install a simple application. A more complex installation requires a deep script result analysis. For example, we have a cross-platform application. The first script determines the operation system and the second one calls an appropriate installation script. Note, that installation script may be written in different languages (the shell for Linux and PowerShell for Windows). Murano agent can easily handle this situation and even more complicated ones.

So murano agent operates not with scripts, but with execution plans, which are minimum units of the installation workflow.

Murano-agent on a new VM

Earlier most of the application deployments were possible only on images with pre-installed murano agent. You can refer to *corresponding documentation* on building an image with murano-agent.

Currently murano-agent can be automatically installed by cloud-init. To deploy an application on an image with pre-installed cloud-init you should mark the image with Murano specific metadata. More information about preparing images can be found *here*. This type of installation has some limitations. The image has to have pre-installed python. Murano-agent is installed from PyPi so the instance should have connectivity with the Internet. Also it requires an installation of some python packages, e.g. python3-pip, python3-dev, python3-setuptools, python3-virtualenv, which are also installed by cloud-init.

Interaction with murano-engine

First of all, communication between murano-agent and murano engine should be established. The communication is performed through AMQP protocol. This type of communication is preferable (for example, compared to SSH) because it is:

- Durable
 - To establish the connection, there is no need to wait until the instance is spawned. Murano-agent, on its turn, does not need to wait for a murano-engine task.
 - Messages can be sent to RabbitMQ asynchronously.

- The connection does not depend on network issues. And moreover, there is no way to physically connect to the virtual machine if floating IP is not set.
- It is possible to reload the instance and change network parameters during the deployment.
- Reliable

If one instance of murano-engine fails in the middle of the deployment, another one picks up the messages from the queue and continue the deployment.

Right after application author calls the **deploy** method of the class, inherited from *io.murano.resources.Instance*, new murano-agent configuration file starts forming in accordance with the values specified in the [rabbitmq] murano configuration file section. A script that runs through cloud-init copies a new file to the right place during the instance booting.

Execution plans and execution plan templates

It was already mentioned that murano-agent recognizes execution plans. These instructions contain scripts with all the required parameters. The application package author provides the execution plan templates together with scripts code. During the deployment it is complemented with all required parameters (including user-input).

For more information on execution plan templates, refer to *Execution plan template*.

Take a look at the muranoPL code snippet. The “EtcAddMember” template expects *name* and *ip* parameters. The first line shows that these parameters are passed to the template, and the second one shows that the template is sent to the agent:

```
- $template: $resources.yaml('EtcAddMember.template').bind(dict(  
    name => $.instance.name,  
    ip => $.getIp()  
))  
- $clusterConfig: $_cluster.masterNode.instance.agent.call($template,  
  ↪$resources)
```

Beside the simple agent call, there is a method that enables sending an already prepared execution plan (not a template). The main difference between template and full execution plan is in the *files* section. Prepared execution plan contains file contents and name by which they are reachable. So it is not required to provide the resources argument:

```
..instance.agent.callRaw($plan)
```

Also, there are *instance.agent.call(\$template, \$resources)* and *..instance.agent.sendRaw(\$plan)* methods which have the same meaning but indicate the engine not to wait for the script execution result. The default agent call response time (with the corresponding method call) is set in murano configuration file and equals to one hour. Take a look at the engine section:

```
[engine]  
# Time for waiting for a response from murano-agent during the  
# deployment (integer value)  
agent_timeout = 3600
```

Note: Murano-agent is able to run different types of scripts, such as powershell, python, bash, chef, and

puppets. Moreover, it has a mechanism for extending supported formats and that is why murano agent is called unified

To use puppet a deployment workflow, configure an execution plan as follows:

1. Set correct version of format:

`FormatVersion >=2.1.0`. Previous formats does not support puppet execution.

2. Use corresponding type

In the script section, script item should have `Type: Puppet`

3. Provide entry-point class

Use puppet syntax `EntryPoint: mysql::server`

Note: You can use scripts directly from git or svn repositories:

Files:

```
- mysql: https://github.com/nanliu/puppet-staging.git
```

A script output is available in the murano-agent log file. This file is located on the spawned instance at `/etc/murano/agent.conf` on a Linux-based machine, or `C:\Murano\Agent\agent.conf` on a Windows-based machine. You can also refer to murano-agent log if there is no connectivity with murano-engine (check if RabbitMQ settings are updated) or to track deployment execution.

6.1.9 Policy enforcement using Congress

Policies are defined and evaluated in the [Congress](#) project. The policy language for Congress is Datalog. The congress policy consists of the Datalog rules and facts.

Examples of policies are as follows:

- Minimum 2 GB of RAM for all VM instances.
- A certified version for all Apache server instances.
- Data placement policy: all database instances must be deployed at a given geographic location enforcing some law restriction on data placement.

These policies are evaluated over data in the form of tables (Congress data structures). A deployed Murano environment must be decomposed to the Congress data structures. The decomposed environment is sent to Congress for simulation. Congress simulates whether the resulting state violates any defined policy: deployment is aborted in case of policy violation.

Murano uses two predefined policies in Congress:

- `murano_system` contains rules and facts of policies defined by the cloud administrator.
- `murano` contains only facts/records reflecting the resulting state after the deployment of an environment.

Records in the murano policy are queried by rules from the `murano_system` policy. The Congress simulation does not create any records in the murano policy, and only provides the feedback on whether the resulting state violates the policy or not.

As a part of the policy guided fulfillment, you need to enforce policies on a murano environment deployment. If the policy enforcement fails, the deployment fails as well.

This section contains the following subsections:

Setting up policy enforcement

Before you use the policy enforcement feature, configure Murano and Congress properly.

Note: This article does not cover Murano and Congress configuration options useful for Murano application deployment, for example, DNS setup, floating IPs, and so on.

To enable policy enforcement, complete the following tasks:

1. In Murano:

- Enable the `enable_model_policy_enforcer` option in the `murano.conf` file:

```
[engine]
# Enable model policy enforcer using Congress (boolean value)
enable_model_policy_enforcer = true
```

- Restart `murano-engine`.
2. Verify that Congress is installed and available in your OpenStack environment. See the details in the [Congress official documentation](#).
 3. Install the [congress command-line client](#) as any other OpenStack command-line client.
 4. For Congress, configure the following policies that policy enforcement uses during the evaluation:

- `murano` policy

It is created by the Congress' `murano` datasource driver, which is a part of Congress. Configure it for the OpenStack project (tenant) where you plan to deploy your Murano application. Datasource driver retrieves deployed Murano environments and populates Congress' `murano` policy tables. See [Murano policy enforcement internals](#) for details.

Remove the existing `murano` policy and create a new `murano` policy configured for the `demo` project, by running:

```
# remove default murano datasource configuration, because it
↪is using 'admin' project. We need 'demo' project to be used.
openstack congress datasource delete murano
openstack congress datasource create murano murano --config_
↪username="$OS_USERNAME" --config tenant_name="demo" --
↪config password="$OS_PASSWORD" --config auth_url="$OS_AUTH_
↪URL"
```

- `murano_system` policy

It holds the user-defined rules for policy enforcement. Typically, the rules use tables from other policies, for example, `murano`, `nova`, `keystone`, and others. Pol-

icy enforcement expects the `predeploy_errors` table here that is available on the `predeploy_errors` rules creation.

Create the `murano_system` rule, by running:

```
# create murano_system policy
openstack congress policy create murano_system

# resolves objects within environment
openstack congress policy rule create murano_system 'murano_
↳env_of_object(oid,eid):-murano:connected(eid,oid),↳
↳murano:objects(eid,tid,"io.murano.Environment")'
```

- `murano_action` policy with internal management rules.

These rules are used internally in the policy enforcement request and stored in a dedicated `murano_action` policy that is created here. They are important in case an environment is redeployed.

```
# create murano_action policy
openstack congress policy create murano_action --kind action

# register action deleteEnv
openstack congress policy rule create murano_action 'action(
↳"deleteEnv")'

# states
openstack congress policy rule create murano_action 'murano:states-
↳(eid, st) :- deleteEnv(eid), murano:states( eid, st)''

# parent_types
openstack congress policy rule create murano_action 'murano:parent_
↳types-(tid, type) :- deleteEnv(eid), murano:connected(eid, tid),
↳murano:parent_types(tid,type)''
openstack congress policy rule create murano_action 'murano:parent_
↳types-(eid, type) :- deleteEnv(eid), murano:parent_types(eid,type)''

# properties
openstack congress policy rule create murano_action
↳'murano:properties-(oid, pn, pv) :- deleteEnv(eid),↳
↳murano:connected(eid, oid), murano:properties(oid, pn, pv)''
openstack congress policy rule create murano_action
↳'murano:properties-(eid, pn, pv) :- deleteEnv(eid),↳
↳murano:properties(eid, pn, pv)''

# objects
openstack congress policy rule create murano_action 'murano:objects-
↳(oid, pid, ot) :- deleteEnv(eid), murano:connected(eid, oid),↳
↳murano:objects(oid, pid, ot)''
openstack congress policy rule create murano_action 'murano:objects-
↳(eid, tnid, ot) :- deleteEnv(eid), murano:objects(eid, tnid, ot)''
```

(continues on next page)

(continued from previous page)

```
# relationships
openstack congress policy rule create murano_action
↳ 'murano:relationships-(sid, tid, rt) :- deleteEnv(eid),
↳ murano:connected(eid, sid), murano:relationships( sid, tid, rt) '
openstack congress policy rule create murano_action
↳ 'murano:relationships-(eid, tid, rt) :- deleteEnv(eid),
↳ murano:relationships(eid, tid, rt) '

# connected
openstack congress policy rule create murano_action
↳ 'murano:connected-(tid, tid2) :- deleteEnv(eid),
↳ murano:connected(eid, tid), murano:connected(tid,tid2) '
openstack congress policy rule create murano_action
↳ 'murano:connected-(eid, tid) :- deleteEnv(eid),
↳ murano:connected(eid,tid) '
```

Creating policy enforcement rules

This article illustrates how you can create policy enforcement rules. For testing purposes, create rules that prohibit the creation of instances with the flavor with over 2048 MB of RAM following the procedure below.

Procedure:

1. Verify that you have configured your OpenStack environment as described in *Setting up policy enforcement*.
2. To create the `predeploy_errors` rule, run:

```
congress policy rule create murano_system "predeploy_errors(eid, obj_id,
↳ msg) :- murano:objects(obj_id, pid, type), murano:objects(eid, tid, \
↳ "io.murano.Environment\"), murano:connected(eid, pid),
↳ murano:properties(obj_id, \"flavor\", flavor_name), flavor_ram(flavor_
↳ name, ram), gt(ram, 2048), murano:properties(obj_id, \"name\", obj_
↳ name), concat(obj_name, \": instance flavor has RAM size over 2048MB\",
↳ msg) "
```

The command above contains the following information:

```
predeploy_errors(eid, obj_id, msg) :-
  murano:objects(obj_id, pid, type),
  murano:objects(eid, tid, "io.murano.Environment"),
  murano:connected(eid, pid),
  murano:properties(obj_id, "flavor", flavor_name),
  flavor_ram(flavor_name, ram),
  gt(ram, 2048),
  murano:properties(obj_id, "name", obj_name),
  concat(obj_name, ": instance flavor has RAM size over 2048MB", msg)
```

Policy validation engine checks the `predeploy_errors` rule, and rules referenced within this rule are evaluated by the Congress engine.

In this example, we create the rule that references the `flavor_ram` rule we create afterwards. It disables flavors with RAM more than 2048 MB and constructs the message returned to the user in the `msg` variable.

In this example we use data from policy `murano` which is represented by `murano:properties`. There are stored rows with decomposition of model representing murano application. We also use built-in functions of Congress:

- `gt` stands for 'greater-than'
- `concat` joins two strings into one variable

3. To create the `flavor_ram` rule, run:

```
congress policy rule create murano_system "flavor_ram(flavor_name, ram) :-
↪ nova:flavors(id, flavor_name, cpus, ram)"
```

This rule resolves parameters of flavor by flavor name and returns the `ram` parameter. It uses the `flavors` rule from nova policy. Data in this policy is filled by the nova datasource driver.

4. Check the rule usage.

1. Create an environment with a simple application:

- Select an application from the murano applications.
- Create a `m1.medium` instance, which uses 4096 MB RAM.

Add Application to "quick-env-3"

Instance flavor
m1.medium

Instance image
Select Image

Key Pair
No keypair +

Availability zone
nova

Git Application
Specify some instance parameters on which the application would be created

Instance flavor: Select registered in Openstack flavor. Consider that application performance depends on this parameter.

Instance image: Select valid image for the application. Image should have Murano agent installed and registered in Glance.

Key Pair: Select the Key Pair to control access to instances. You can login to instances using this KeyPair after application deployment

Availability zone: Select availability zone where the application would be installed.

Back Create

2. Deploy the environment.

Deployment fails as the rule is violated: environment is in the Deploy FAILURE status. Check the deployment logs for details:

The screenshot shows the OpenStack Murano web interface. On the left is a navigation sidebar with 'Environments' selected. The main content area is titled 'Deployment information' and shows the breadcrumb 'environments > environment quick-env-2 > deployment at 2015-01-20 04:24:13'. Below this are tabs for 'Configuration' and 'Logs', with 'Logs' being the active tab. The 'Deployment Logs' section contains the following text:

```

2015-01-20 04:24:13 - Action deploy is scheduled
2015-01-20 04:24:15 - Model validation failed:
ftceni54s1ywb1: instance flavor has RAM size over 2048MB
2015-01-20 04:24:15 - Deployment finished with errors

```

See also:

- [Creating base modification rules](#)

Murano policy enforcement internals

This section describes internals of the murano policy enforcement feature.

Model decomposition

The data for the policy validation comes from the models of Murano applications. These models are transformed to a set of rules that are processed by Congress.

There are several *tables* created in murano policy for different kinds of rules that are as follows:

- murano:objects(object_id, parent_id, type_name)
- murano:properties(object_id, property_name, property_value)
- murano:relationships(source, target, name)
- murano:connected(source, target)
- murano:parent_types(object_id, parent_type_name)
- murano:states(environment_id, state)

murano:objects(object_id, parent_id, type_name)

This rule is used for representation of all objects in Murano model, such as environment, application, instance, and other.

Value of the type property is used as the type_name parameter:

```

name: wordpress-env
'?: {type: io.murano.Environment, id: 83bff5ac}
applications:
- '?: {id: e7a13d3c, type: com.example.databases.MySql}

```

The model above transforms to the following rules:

- murano:objects+("83bff5ac", "tenant_id", "io.murano.Environment")

- `murano:objects+("83bff5ac", "e7a13d3c", "com.example.databases.MySql")`

Note: The owner of the environment is a project (tenant).

`murano:properties(object_id, property_name, property_value)`

Each object may have properties. In this example we have an application with one property:

```
applications:
- '?': {id: e7a13d3c, type: com.example.databases.MySql}
database: wordpress
```

The model above transforms to the following rule:

- `murano:properties+("e7a13d3c", "database", "wordpress")`

Inner properties are also supported using dot notation:

```
instance:
'?': {id: 825dc61d, type: io.murano.resources.LinuxMuranoInstance}
networks:
useFlatNetwork: false
```

The model above transforms to the following rule:

- `murano:properties+("825dc61d", "networks.useFlatNetwork", "False")`

If a model contains list of values, it is represented as a set of multiple rules:

```
instances:
- '?': {id: be3c5155, type: io.murano.resources.LinuxMuranoInstance}
networks:
customNetworks: [10.0.1.0, 10.0.2.0]
```

The model above transforms to the following rules:

- `murano:properties+("be3c5155", "networks.customNetworks", "10.0.1.0")`
- `murano:properties+("be3c5155", "networks.customNetworks", "10.0.2.0")`

`murano:relationships(source, target, name)`

Murano application models may contain references to other applications. In this example, the WordPress application references MySQL in the database property:

```
applications:
- '?':
id: 0aafd67e
type: com.example.databases.MySql
- '?':
id: 50fa68ff
```

(continues on next page)

(continued from previous page)

```

type: com.example.WordPress
database: 0aafd67e

```

The model above transforms to the following rule:

- `murano:relationships+("50fa68ff", "0aafd67e", "database")`

Note: For the database property we do not create the `murano:properties+` rule.

If we define an object within other object, they will have relationships between them:

```

applications:
- '?':
  id: 0aafd67e
  type: com.example.databases.MySql
  instance:
    '?': {id: ed8df2b0, type: io.murano.resources.
↪LinuxMuranoInstance}

```

The model above transforms to the following rule:

- `murano:relationships+("0aafd67e", "ed8df2b0", "instance")`

There are special relationships of services from the environment to its applications:
`murano:relationships+("env_id", "app_id", "services")`

murano:connected(source, target)

This table stores both direct and indirect connections between instances. It is derived from `murano:relationships`:

```

applications:
- '?':
  id: 0aafd67e
  type: com.example.databases.MySql
  instance:
    '?': {id: ed8df2b0, type: io.murano.resources.
↪LinuxMuranoInstance}
- '?':
  id: 50fa68ff
  type: com.example.WordPress
  database: 0aafd67e

```

The model above transforms to the following rules:

- `murano:connected+("50fa68ff", "0aafd67e")` # WordPress to MySql
- `murano:connected+("50fa68ff", "ed8df2b0")` # WordPress to LinuxMuranoInstance
- `murano:connected+("0aafd67e", "ed8df2b0")` # MySql to LinuxMuranoInstance

murano:parent_types(object_id, parent_name)

Each object in murano has a class type. These classes may inherit from one or more parents. For example, `LinuxMuranoInstance > LinuxInstance > Instance`:

```
instances:
- '?': {id: be3c5155, type: LinuxMuranoInstance}
```

The model above transforms to the following rules:

- `murano:objects+("...", "be3c5155", "LinuxMuranoInstance")`
- `murano:parent_types+("be3c5155", "LinuxMuranoInstance")`
- `murano:parent_types+("be3c5155", "LinuxInstance")`
- `murano:parent_types+("be3c5155", "Instance")`

Note: The type of an object is also repeated in its parent types (`LinuxMuranoInstance` in the example) for easier handling of user-created rules.

Note: If a type inherits from more than one parent, and these parents inherit from one common type, the `parent_type` rule is included only once in the common type.

murano:states(environment_id, state)

Currently only one record for environment is created:

- `murano:states+("uugi324", "pending")`

Using policy for the base modification of an environment

Congress policies enables a user to define modification of an environment prior to its deployment. This includes:

- Adding components, for example, monitoring.
- Changing and setting properties, for example enforcing a given zone, flavors, and others.
- Configuring relationships within an environment.

Use cases examples:

- Installation of the monitoring agent on each VM instance by adding a component with the agent and creating relationship between the agent and instance.
- Enabling a certified version to all Apache server instances: setting the version property to all Apache applications within an environment to a particular version.

These policies are evaluated over data in the form of tables that are Congress data structures. A deployed murano environment must be decomposed to Congress data structures. The further workflow is as follows:

- The decomposed environment is sent to Congress for simulation.
- Congress simulates whether the resulting state requires modification.

- In case the modification of a deployed environment is required, Congress returns a list of actions in the YAML format to be performed on the environment prior to the deployment.

For example:

```
set-property: {object_id: c46770dec1db483ca2322914b842e50f, prop_name:↵
↵keyname, value: production-key}
```

The example above sets the `keyname` property to the `production-key` value on the instance identified by `object_id`. An administrator can use it as an output of the Congress rules.

- The action specification is parsed in murano. The given action class is loaded, and the action instance is created.
- The parsed parameters are supplied to the action `__init__` method.
- The action is performed on a given environment (the `modify` method).

Creating base modification rules

This example illustrates how to configure the rule enforcing all VM instances to deploy with a secure key pair. This may be required in a production environment.

Warning: Before you create rules, configure your OpenStack environment as described in [Setting up policy enforcement](#).

Procedure:

1. To create the `predeploy_modify` rule, run:

```
congress policy rule create murano_system 'predeploy_modify(eid, obj_id,↵
↵action):-murano:objects(obj_id, pid, type), murano_env_of_object(obj_id,↵
↵eid), murano:properties(obj_id, "keyname", kn), concat("set-property:↵
↵{object_id: ", obj_id, first_part), concat(first_part, ", prop_name:↵
↵keyname, value: production-key}", action)'
```

The command above contains the following information:

```
predeploy_modify(eid, obj_id, action) :-
    murano:objects(obj_id, pid, type),
    murano:objects(eid, tid, "io.murano.Environment"),
    murano:connected(eid, pid),
    murano:properties(obj_id, "keyname", kn),
    concat("set-property: {object_id: ", obj_id, first_part),
    concat(first_part, ", prop_name: keyname, value: production-key}",↵
↵action)
```

Policy validation engine checks the `predeploy_modify` rule. And the Congress engine evaluates the rules referenced inside this rule.

Note: The `production-key` key pair must already exist, though you can use any other existing

key pair.

2. Deploy the environment.

Instances within the environment are deployed with the specified key pair.

See also:

- [Creating policy enforcement rules](#)

6.1.10 Using Glare as a storage for packages

DevStack installation

1. Enable Glare service in DevStack

To enable the Glare service in DevStack, edit the local .conf file:

```
$ cat local.conf
[[local|localrc]]
enable_service g-glare
```

2. Run DevStack:

```
$ ./stack.sh
```

Result Glare service is installed with DevStack. You can find logs in g-glare screen session.

3. Install the muranoartifact plug-in from murano/contrib

```
$ cd $DEST/murano/contrib/glance/
$ sudo pip install -e .
```

4. Restart Glare

5. Set Glare as packages service in murano-engine. For this, edit the [engine] section in the murano.conf file. By default, murano.conf is located in the /etc/murano directory

```
[engine]
packages_service = glare
```

6. Restart murano-engine

Note: You also can use glance as a value of the packages_service option for the same behaviour

7. Enable Glare in murano-dashboard. For this, modify the following line in the _50_murano.py file

```
MURANO_USE_GLARE = True
```

By default, the `_50_murano.py` file is located in `$HORIZON_DIR/openstack_dashboard/local/local_settings.d/`.

8. Restart the `apache2` service. Now `murano-dashboard` will retrieve packages from Glare.
9. Log in to Dashboard and navigate to *Applications > Manage > Packages* to view the empty list of packages. Alternatively, use the `murano` command.
10. Use `--murano-packages-service` option to specify backend, used by `murano` command. Set it to `glare` for using Glare

Note: You also can use `glance` as value of `--murano-packages-service` option or environment variable `MURANO_PACKAGES_SERVICE` for same behaviour

- View list of packages:

```
$ . {DEVSTACK_SOURCE_DIR}/openrc admin admin
$ murano --murano-packages-service=glare package-list
```

ID	Name	FQN	Author	Active	Is Public	Type	Version

- Importing Core library

```
$ cd $DEST/murano/meta/io.murano/
$ zip io.murano.zip -r *
$ murano --murano-packages-service=glare package-import \
  --is-public /opt/stack/murano/meta/io.murano/io.murano.zip
```

Importing package io.murano

ID	Name	FQN	Author	Active	Is Public	Type	Version
91a9c78f-f23a-4c82-aeda-14c8cbef096a	Core library	io.murano	murano.io	True		Library	0.0.0

Set up Glare API endpoint manually

If you do not plan to get Glare service from keystone application catalog, specify where g-glare service is running.

1. Specify Glare URL in `murano.conf`. It is `http://0.0.0.0:9494` by default and can be changed by setting `bind_host` and `bind_port` options in the `glance-glare.conf` file.

```
[glare]
url = http://<GLARE_API_URL>:<GLARE_API_PORT>
```

2. Specify Glare URL in the Dashboard settings file, `_50_murano.py` :

```
GLARE_API_URL = 'http://<GLARE_API>:<GLARE_API_PORT>'
```

3. Set the `GLARE_URL` environment variable for `python-muranoclient`. Alternatively, use the `--glare-url` option in CLI.

```
$ murano --murano-packages-service=glare --glare-url=http://0.0.0.0:9494 ↵
↵package-list
```

6.1.11 Network configuration

Murano may work in various networking environments and is capable of detecting the current network configuration and choosing appropriate settings automatically. However, some additional actions are required to support advanced scenarios.

Nova-network support

Nova-network is the simplest networking solution, which has limited capabilities but is available on any OpenStack deployment without the need to deploy any additional components.

When a new murano environment is created, murano checks if a dedicated networking service, for example, neutron, exists in the current OpenStack deployment. It relies on the Identity service catalog for that. If such a service is not present, murano automatically falls back to nova-network. No further configuration is needed in this case, all the VMs spawned by Murano will be joining the same network.

Neutron support

If neutron is installed, murano enables its advanced networking features that give you the ability to avoid configuring networks for your application.

By default, it creates an isolated network for each environment and joins all VMs needed by your application to that network. To install and configure the application in a newly spawned virtual machine, murano also requires a router to be connected to the external network.

Automatic neutron configuration

To create the router automatically, provide the following parameters in the configuration file:

```
[networking]
external_network = %EXTERNAL_NETWORK_NAME%
router_name = %MURANO_ROUTER_NAME%
create_router = true
```

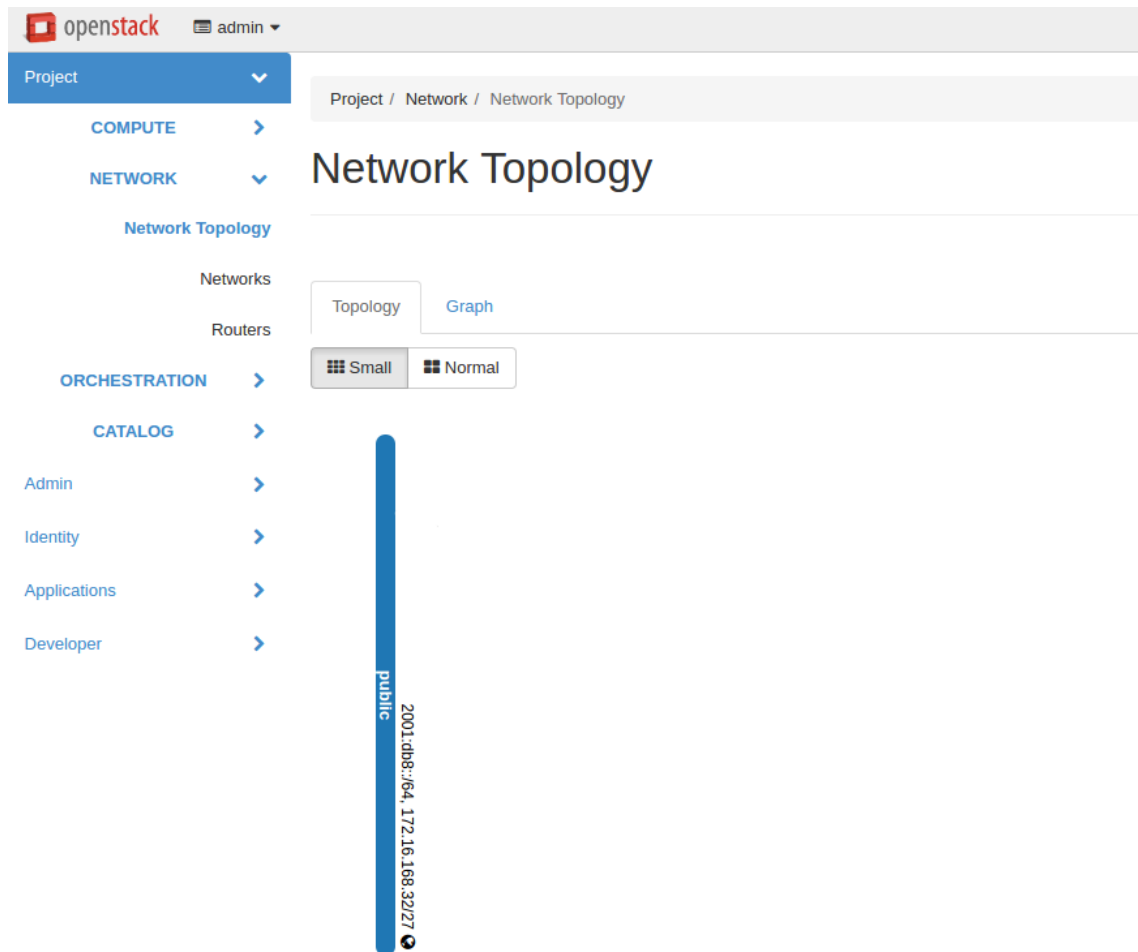
To figure out the name of the external network, run **openstack network list --external**.

During the first deployment, the required networks and router with a specified name will be created and set up.

Manual neutron configuration

To configure neutron manually, follow the steps below.

1. Create a public network.
 1. Log in to the OpenStack dashboard as an administrator.
 2. Verify the existence of external networks. For this, navigate to *Project > Network > Network Topology*.
 3. Check the network type in network details. For this, navigate to *Admin > Networks* and see the *Network name* section. Alternatively, run the **openstack network list --external** command using CLI.
 4. Create a new external network as described in the [OpenStack documentation](#).



2. Create a local network.
 1. Navigate to *Project > Network > Networks*.
 2. Click *Create Network* and fill in the form.
3. Create a router.
 1. Navigate to *Project > Network > Routers*.
 2. Click *Create Router*.
 3. In the *Router Name* field, enter *murano-default-router*. If you specify a name other than *murano-default-router*, change the following settings in the configuration file:

```
[networking]
router_name = %SPECIFIED_NAME%
create_router = false
```

4. Click *Create router*.
5. Click the newly created router name.
6. In the *Interfaces* tab, click *Add Interface*.
7. Specify the subnet and IP address.

Add Interface ✕

Subnet *

IP Address (optional) ⓘ

Router Name *

Router ID *

Description:

You can connect a specified subnet to the router.

The default IP address of the interface created is a gateway of the selected subnet. You can specify another IP address of the interface here. You must select a subnet to which the specified IP address belongs to from the above list.

8. Verify the result in *Project > Network > Network Topology*.

The screenshot shows the OpenStack web interface. The breadcrumb is *Project / Network / Network Topology*. The main heading is **Network Topology**. On the left, there is a navigation menu with categories: Project, COMPUTE, NETWORK, ORCHESTRATION, and CATALOG. Under NETWORK, there are sub-sections for Networks and Routers. The main content area shows a topology diagram with two vertical bars representing networks: a blue bar labeled 'public' with IP range '2001:db8::/64, 172.16.168.32/27' and an orange bar labeled 'local' with IP range '192.168.0.0/24'. A router icon connects the two networks. There are tabs for 'Topology' and 'Graph', and zoom controls for 'Small' and 'Normal'.

6.1.12 Murano service broker for Cloud Foundry

Service broker overview

Service broker is a new murano component which implements [Cloud Foundry Service Broker API](#).

This lets users build 'hybrid' infrastructures that are services like databases, message queues, key/value stores, and so on. This services can be uploaded and deployed with murano and made available to Cloud Foundry apps on demand. The result is lowered cost, shorter timetables, and quicker access to required tools developers can 'self serve' by building any required service, then make it instantly available in Cloud Foundry.

Configure service broker

Manual installation

If you use local murano installation, you can configure and run murano service broker in a few simple steps:

1. Change into the murano directory:

```
cd ~/murano/murano
```

2. Generate the murano service broker config file. Murano service broker has a common config file for service broker API services. Using tox, generate a sample configuration file:

```
tox -e gencfconfig
```

3. Copy the configuration file for further modifications:

```
cd ~/murano/murano/etc/murano
ln -s murano-cfapi.conf.sample murano-cfapi.conf
```

4. Edit `murano-cfapi.conf`. Below is an example of the basic settings you may need to configure.

Note: The example below uses the SQLite database. Edit the `[database]` section to use another database.

```
[DEFAULT]
debug = true
verbose = true

...

[database]
backend = sqlalchemy
connection = sqlite:///murano_cfapi.sqlite

...
```

(continues on next page)

(continued from previous page)

```
[keystone_authtoken]
www_authenticate_uri = 'http://%OPENSTACK_HOST_IP%:5000/v3'
auth_host = '%OPENSTACK_HOST_IP%'
auth_port = 5000
auth_protocol = http
admin_tenant_name = %OPENSTACK_ADMIN_TENANT%
admin_user = %OPENSTACK_ADMIN_USER%
admin_password = %OPENSTACK_ADMIN_PASSWORD%

...

[cfapi]
tenant = %TENANT_NAME%
bind_host = %HOST_IP%
bind_port = 8083
auth_url = 'http://%OPENSTACK_HOST_IP%:5000/v3'
```

Note: The bind_host IP should be in the same network as the Cloud Foundry instance.

5. Create database tables for murano service broker:

```
cd ~/murano/murano
tox -e venv -- murano-cfapi-db-manage \
    --config-file ./etc/murano/murano-cfapi.conf upgrade
```

6. Launch the murano service broker API in a separate terminal:

```
cd ~/murano/murano
tox -e venv -- murano-cfapi --config-file ./etc/murano/murano-cfapi.conf
```

Note: Run the command in a new terminal as the process will be running in the terminal until you terminate it, therefore, blocking the current terminal.

Devstack installation

It is really easy to enable service broker in your devstack installation. You need simply update your `local.conf` with the following:

```
[[local|localrc]]
enable_plugin murano https://opendev.org/openstack/murano
enable_service murano-cfapi
```

How to use service broker

After service broker is configured and started you have nothing to do with service broker from murano side - it is an adapter which is used by Cloud Foundry PaaS.

To access and use murano packages through Cloud Foundry, you need to perform following steps:

1. Log in to Cloud Foundry instance via ssh.

```
ssh -i <key_name> <username>@<hostname>
```

2. Log in to Cloud Foundry itself.

```
cf login -a https://api.<smthg>.xip.io -u <user_name> -p <password>
```

3. Add murano service broker.

```
cf create-service-broker <broker_name> <OS_USERNAME> <OS_PASSWORD> http://
↪<service_broker_ip>:8083
```

4. Enable access to murano packages.

```
cf enable-service-access <service_name>
```

Warning: By default, access to all services is prohibited.

Note: You can use `service-access` command to see human-readable list of packages.

5. Provision murano service through Cloud Foundry.

```
cf create-service 'Apache HTTP Server' default MyApacheInstance -c
↪apache.json
```

```
{
  "instance": {
    "flavor": "m1.medium",
    "?": {
      "type": "io.murano.resources.LinuxMuranoInstance"
    },
    "keyname": "nstarodubtsev",
    "assignFloatingIp": "True",
    "name": "<name_pattern>",
    "availabilityZone": "nova",
    "image": "1b9ff37e-dff3-4308-be08-9185705dad91"
  },
  "enablePHP": "True"
}
```

Known issues

- Hard to deploy complex apps

Useful links

Here is the list of the links for Cloud Foundry documentation which you might need:

1. [Cloud Foundry development version launcher](#)
2. [How to manage Cloud Foundry service brokers](#)
3. [Cloud Foundry CLI docs](#)

6.1.13 Troubleshooting

Log location

By default, logs are sent to stdout. Consider how to set up the log files.

Murano API + Engine

To define a file where to store logs, use the `log_file` option in the `murano.conf` file. You can provide an absolute or a relative path.

To enable a detailed log file configuration, set up `logging.conf`. The example is provided in `etc/murano` directory. The log configuration file location is set with the `log_config_append` option in the murano configuration file.

Murano applications

Murano applications have a separate logging handler and a separate file where all logs from application definitions should be provided. Open the `logging.conf` file and check the args: `('applications.log',)` option in the `handler_applications` section.

Verify that `log_config_append` is not empty and set to the `logging.conf` location.

Issues during configuration

If any issues occur, first of all verify the following:

- All murano components have consistent versions: `murano-dashboard` and `murano-engine` should use the same or compatible `python-muranoclient` version. Dependent component versions can be found in `requirements.txt` file.
- The database is synced with code by running:

```
murano-db-manage --config-file murano.conf upgrade
```

Failed to execute ‘murano-db-manage‘

- Make sure the `--config-file` option is provided.

- Check *connection* parameter in the provided configuration file. It should be a [connection string](#).
- Check that MySQL or PostgreSQL (depending of what you provided in the connection string) Python modules are installed on the system.

Applications panel is not seen in horizon

- Make sure that the following files are copied to the `openstack_dashboard/local/enabled` directory, and `_50_murano.py` is copied to `openstack_dashboard/local/local_settings.d` directory.

- `_50_dashboard_catalog.py`
- `_51_muranodashboard.py`
- `_60_panel_group_browse.py`
- `_63_panel_murano_catalog.py`
- `_70_panel_group_manage.py`
- `_71_panel_murano_packages.py`
- `_72_panel_murano_images.py`
- `_73_panel_murano_categories.py`
- `_80_panel_group_applications.py`
- `_81_panel_applications_environments.py`

- Check that murano data is not inserted twice in the settings file and as a plugin.

Applications panel can be browsed, but 'Unable to communicate to murano-api server.' appears

If you have murano registered in keystone, verify the endpoint URL is valid and service has *application-catalog* name. If you do not want to register the murano service in keystone, just add `MURANO_API_URL` option to the horizon local setting.

Issues during deployment

Besides identifying errors from log files, there is another and more flexible way to browse deployment errors -- directly from UI. When the *Deploy Failed* status appears, navigate to *Environment Components* and click the *Latest Deployment Log* tab. You can see steps of the deployment and the one that failed would have red color.

while scanning a simple key in "<string>", line 32, column 3: ...

There is an error in the YAML file format. Before uploading a package, validate your file in an online YAML validator like [YAMLint](#). Later [validation tool](#) to check package closely while uploading will be added.

NoPackageForClassFound: Package for class io.murano.Environment is not found

Verify that murano core package is uploaded. If not, the content of the `meta/io.murano` folder should be zipped and uploaded to Murano.

[keystoneclient.exceptions.AuthorizationFailure]: Authorization failed: You are not authorized to perform the requested action. (HTTP 403)

The token expires during the deployment. Usually the default standard token lifetime is one hour. The error occurs frequently as, in most cases, a deployment takes longer than that or does not start right after a token is generated.

Workarounds:

- Use trusts. Only possible in the v3 version. Read more in the [official documentation](#) or [here](#). Do not forget to check the corresponding heat and murano settings. Trusts are enabled by default in murano and heat since Kilo release.

In murano, the corresponding configuration option is located in the `engine` section:

```
[engine]
...
# Create resources using trust token rather than user's token (boolean
# value)
use_trusts = true
```

If your Keystone runs v2 version, see the solutions below.

- Make logout/login to compose a new token and start the deployment again. Would not help for long deployment or if the token lifetime is too small.
- Increase the token lifetime in the keystone configuration file.

The murano-agent did not respond within 3600 seconds

- Check transport access to the virtual machine: verify that the router has a gateway.
- Check the RabbitMQ settings: verify that the agent has valid RabbitMQ parameters. Go to the spawned virtual machine and open `*/etc/murano/agent.conf` on the Linux-based machine or `C:\Murano\Agent\agent.conf` on the Windows-based machine. Additionally, you can examine agent logs that by default are located at `/var/log/murano-agent.log`. The first part of the log file contains reconnection attempts to the RabbitMQ since the valid RabbitMQ address and queue have not been obtained yet.
- Verify that the `driver` option in `[oslo_messaging_notifications]` group is set to `messagingv2`.

`murano.engine.system.agent.AgentException`

The agent started the execution plan but something went wrong. Examine agent logs (see the previous paragraph for the logs placement information). Also, try to manually execute the application scripts.

[exceptions.EnvironmentError]: Unexpected stack state NOT_FOUND or UPDATE_FAILED

An issue with heat stack creation, examine the heat log file. Try to manually spawn the instance. If the reason of the stack creation fail is `no valid host was found`, there might be not enough resources or something is wrong with the nova-scheduler.

Router could not be created, no external network found

Find the `external_network` parameter in the `networking` section of the murano configuration file and verify that the specified external network does exist through Web UI or by executing the `openstack network list --external` command.

Deployment log in the UI contains incomplete reports

Sometimes logs contain only two messages after the application deployment. There are no messages provided in applications themselves:

```
2015-09-21 11:14:58 Action deploy is scheduled
2015-09-21 11:16:43 Deployment finished successfully
```

To fix the issue, set the `driver` option in the `murano.config` file to `messagingv2`.

6.1.14 Application Developer Guide

Developing Murano Packages 101

Murano provides a very powerful and flexible platform to automate the provisioning, deployment, configuration and lifecycle management of applications in OpenStack clouds. However, the flexibility comes at cost: to manage an application with Murano one has to design and develop special scenarios which will tell Murano how to handle different aspects of application lifecycle. These scenarios are usually called "Murano Applications" or "Murano Packages". It is not hard to build them, but it requires some time to get familiar with Murano's DSL to define these scenarios and to learn the common patterns and best practices. This article provides a basic introductory course of these aspects and aims to be the starting point for the developers willing to learn how to develop Murano Application packages with ease.

The course consists of the following parts:

Part 1: Creating your first Application Package

All tutorials on programming languages start with a "Hello, World" example, and since Murano provides its own programming language, this guide will start the same way. Let's do a "Hello, World" application. It will not do anything useful yet, but will provide you with an understanding of how things work in Murano. We will add more logic to the package at later stages. Now let's start with the basics:

Creating package manifest

Let's start with creating an empty Murano Package. All packages consist of multiple files (two at least) organized into a special structure. So, let's create a directory somewhere in our file system and set it as our current working directory. This directory will contain our package:

```
$ mkdir HelloWorld
$ cd HelloWorld
```

The main element of the package is its *manifest*. It is a description of the package, telling Murano how to display the package in the catalog. It is defined in a yaml file called `manifest.yaml` which should be placed right in the main package directory. Let's create this file and open it with any text editor:

```
$ vim manifest.yaml
```

This file may contain a number of sections (we will take a closer look at some of them later), but the mandatory ones are `FullName` and `Type`.

The `FullName` should be a unique identifier of the package, the name which Murano uses to distinguish it among other packages in the catalog. It is very important for this name to be globally unique: if you

publish your package and someone adds it to their catalog, there should be no chances that someone else's package has the same name. That's why it is recommended to give your packages Full Names based on the domain you (or the company your work for) own. We recommend using "reversed-domain-name" notation, similar to the one used in the world of Java development: if the *yourdomain.com* is the domain name you own, then you could name your package `com.yourdomain.HelloWorld`. This way your package name will not duplicate anybody else's, even if they also named their package "HelloWorld", because theirs will begin with a different domain-specific prefix.

Type may have either of two values: `Application` or `Library`. `Application` indicates the standard package to deploy an application with Murano, while a `Library` is bundle of reusable scenarios which may be used by other packages. For now we just need a single standalone app, so let's choose an `Application` type. The `Description` is a text attribute, providing detailed info about your package.

Enter these values and save the file. You should have something like this:

```
FullName: com.yourdomain.HelloWorld
Type: Application
Description: |
  A package which demonstrates
  development for Murano
  by greeting the user.
```

This is the minimum required to start. We'll add more manifest data later.

Adding a class

While *manifests* describe Murano packages in the catalog, the actual logic of packages is put into *classes*, which are plain YAML files placed into the `Classes` directory of the application package. So, let's create a directory to store the logic of our application, then create and edit the file to contain the first class of the package.

```
$ mkdir Classes
$ vim Classes/HelloWorld.yaml
```

Murano classes follow standard patterns of object-oriented programming: they define the types of the objects which may be instantiated by Murano. The types are composed of *properties*, defining the data structure of objects, and *methods*, containing the logic that defines the way in which Murano executes the former. The types may be *extended*: the extended class contains all the methods and properties of the class it extends, or it may override some of them.

Let's type in the following YAML to create our first class:

```
1 Name: com.yourdomain.HelloWorld
2
3 Extends: io.murano.Application
4
5 Methods:
6   deploy:
7     Body:
8       - $reporter: $this.find('io.murano.Environment').reporter
9       - $reporter.report($this, "Hello, World!")
```


Let's walk through this code line by line and see what this code does. The first line is pretty obvious: it states the name of our class, `com.yourdomain.HelloWorld`. Note that this name matches the name of the package - that's intentional. Although it is not mandatory, it is strongly recommended to give the main class of your application package the same name as the package itself.

Then, there is an `Extends` directive. It says that our class extends (or inherits) another class, called `io.murano.Application`. That is the base class for all classes which should deploy Applications in Murano. As many other classes it is shipped with Murano itself, thus its name starts with `io.murano`. prefix: `murano.io` domain is controlled by the Murano development team and no one else should create packages or classes having names in that namespace.

Note that `Extends` directive may contain not only a single value, but a list. In that case the class we create will inherit multiple base classes. Yes, Murano has multiple inheritance, yay!

Now, the `Methods` block contains all the logic encapsulated in our class. In this example there is just one method, called `deploy`. This method is defined in the base class we've just inherited - the `io.murano.Application`, so here we *override* it. `Body` block of the method contains the implementation, the actual logic of the method. It's a list of instructions (note the dash-prefixed lines - that's how YAML defines lists), each executed one by one.

There are two instruction statements here. The first one declares a *variable* named `$reporter` (note the `$` character: all the words prefixed with it are variables in Murano language) and assigns it a value. Unlike other languages Murano uses colon (`:`) as an assignment operator: this makes it convenient to express Murano statements as regular YAML mappings. The expression to the right of the colon is executed and the result value is assigned to a variable to the left of the colon.

Let's take a closer look at the right-hand side of the expression in the first statement:

```
- $reporter: $this.find('io.murano.Environment').reporter
```

It takes a value of a special variable called `$this` (which always contains a reference to the current object, i.e. the instance of our class for which the method was called; it is same as `self` in python or `this` in Java) and calls a method named `find` on it with a string parameter equal to `'io.murano.Environment'`; from the call result it takes a "reporter" attribute; this value is assigned to the variable in the left-hand side of the expression.

The meaning of this code is simple: it *finds* the object of class `io.murano.Environment` which owns the current application and returns its "reporter" object. This `io.murano.Environment` is a special object which groups multiple deployed applications. When the end-user interacts with Murano they create these *Environments* and place applications into them. So, every Application is able to get a reference to this object by calling `find` method like we just did. Meanwhile, the `io.murano.Environment` class has various methods to interact with the "outer world", for example to report various messages to the end-user via the deployment log: this is done by the "reporter" property of that class.

So, our first statement just retrieved that reporter. The second one uses it to display a message to a user: it calls a method "report", passes the reference to a reporting object and a message as the arguments of the method:

```
- $reporter.report($this, "Hello, World!")
```

Note that the second statement is not a YAML-mapping: it does not have a colon inside. That's because this statement just makes a method call, it does not need to remember the result.

That's it: we've just made a class which greets the user with a traditional "Hello, World!" message. Now we need to include this class into the package we are creating. Although it is placed within a `Classes` subdirectory of the package, it still needs to be explicitly added to the package. To do that, add a `Classes`

section to your manifest.yaml file. This should be a YAML mapping, having class names as keys and relative paths of files within the Classes directory as the values. So, for our example class it should look like this:

```
Classes:
  com.yourdomain.HelloWorld: HelloWorld.yaml
```

Paste this block anywhere in the manifest.yaml

Pack and upload your app

Our application is ready. It's very simplistic and lacks many features required for real-world applications, but it already can be deployed into Murano and run there. To do that we need to pack it first. We use good old zip for it. That's it: just zip everything inside your package directory into a zip archive, and you'll get a ready-to-use Murano package:

```
$ zip -r hello_world.zip *
```

This will add all the contents of our package directory to a zip archive called `hello_world.zip`. Do not forget the `-r` argument to include the files in subdirectories (the class file in our case).

Now, let's upload the package to murano. Ensure that your system has a `murano-client` installed and your OpenStack cloud credentials are exported as environment variables (if not, sourcing an `openrc` file, downloadable from your horizon dashboard will do the latter). Then execute the following command:

```
$ murano package-import ./hello_world.zip
Importing package com.yourdomain.HelloWorld
+-----+-----+-----+-----+-----+-----+
| ID | Name | FQN |
+-----+-----+-----+-----+-----+-----+
| 251a409645d1444aa1ead8eaac451a1d | com.yourdomain.HelloWorld | com.yourdomain.HelloWorld |
| OpenStack | True | Application |
+-----+-----+-----+-----+-----+-----+
| | | | | | |
+-----+-----+-----+-----+-----+-----+
```

As you can see from the output, the package has been uploaded to Murano catalog and is now available there. Let's now deploy it.

Deploying your application

To deploy an application with Murano one needs to create an *Environment* and add configured instances of your applications into it. It may be done either with the help of user interface (but that requires some extra effort from package developer) or by providing an explicit JSON, describing the exact application instance and its configuration. Let's do the latter option for now.

First, let's create a json snippet for our application. Since the app is very basic, the snippet is simple as well:

```
[
  {
    "op": "add",
    "path": "/-",
    "value": {
      "?": {
        "name": "Demo",
        "type": "com.yourdomain.HelloWorld",
        "id": "42"
      }
    }
  }
]
```

This json follows a standard json-patch notation, i.e. it defines a number of operations to edit a large json document. This particular one *adds* (note the value of *op* key) an object described in the value of the json to the *root* (note the *path* equal to */-* - that's root) of our environment. The object we add has the *type* of *com.yourdomain.HelloWorld* - that's the class we just created two steps ago. Other keys in this json parameterize the object we create: they add a *name* and an *id* to the object. Id is mandatory, name is optional. Note that since the id, name and type are the *system properties* of our object, they are defined in a special section of the json - the so-called *?-header*. Non-system properties, if they existed, would be defined at a top-level of the object. We'll add them in a next chapter to see how they work.

For now, save this JSON to some local file (say, *input.json*) and let's finally deploy the thing.

Execute the following sequence of commands:

```
$ murano environment-create TestHello
+-----+-----+-----+-----+
↔+-----+
| ID                               | Name       | Status | Created          |
↔+-----+-----+-----+-----+
| 34bf673a26a8439d906827dea328c99c | TestHello | ready  | 2016-10-04T13:19:12.
↔+-----+-----+-----+-----+
| 2016-10-04T13:19:12 |
+-----+-----+-----+-----+
↔+-----+

$ murano environment-session-create 34bf673a26a8439d906827dea328c99c
Created new session:
+-----+-----+-----+-----+
```

(continues on next page)

(continued from previous page)

```

| Property | Value |
+-----+-----+
| id       | 6d4a8fa2a5f4484fbc07740ef3ab60dd |
+-----+-----+

$ murano environment-apps-edit --session-id 6d4a8fa2a5f4484fbc07740ef3ab60dd_
↪ 34bf673a26a8439d906827dea328c99c ./input.json

```

This first command creates a murano environment named TestHello. Note the *id* of the created environment - we use it to reference it in subsequent operations.

The second command creates a "configuration session" for this environment. Configuration sessions allow one to edit environments in transactional isolated manner. Note the *id* of the created sessions: all subsequent calls to modify or deploy the environment use both ids of environment and session.

The third command applies the json-patch we've created before to our environment within the configuration session we created.

Now, let's deploy the changes we made:

```

$ murano environment-deploy --session-id 6d4a8fa2a5f4484fbc07740ef3ab60dd_
↪ 34bf673a26a8439d906827dea328c99c
+-----+-----+
| Property          | Value |
+-----+-----+
| acquired_by      | 7b0fe7c67ede443da9840adb2d518d5c |
| created          | 2016-10-04T13:39:34 |
| description_text | |
| id               | 34bf673a26a8439d906827dea328c99c |
| name             | TestHello |
| services         | [ |
|                 | { |
|                 |   "?": { |
|                 |     "name": "Demo", |
|                 |     "status": "deploying", |
|                 |     "type": "com.yourdomain.HelloWorld", |
|                 |     "id": "42" |
|                 |   } |
|                 | } |
|                 | ] |
| status           | deploying |
| tenant_id       | 60b7b5f7d4e64ff0b1c5f047d694d7ca |
| updated         | 2016-10-04T13:39:34 |
| version         | 0 |
+-----+-----+

```

This will deploy the environment. You may check for its status by executing the following command:

```

$ murano environment-show 34bf673a26a8439d906827dea328c99c
+-----+-----+
↪

```

(continues on next page)

(continued from previous page)

Property	Value
acquired_by	None
created	2016-10-04T13:39:34
description_text	
id	34bf673a26a8439d906827dea328c99c
name	TestHello
services	[<ul style="list-style-type: none"> { <ul style="list-style-type: none"> "?": { <ul style="list-style-type: none"> "status": "ready", "name": "Demo", "type": "com.yourdomain.HelloWorld/0.0.0@com.yourdomain.HelloWorld", "_actions": {}, "id": "42", "metadata": null
status	ready
tenant_id	60b7b5f7d4e64ff0b1c5f047d694d7ca
updated	2016-10-04T13:40:29
version	1

As you can see, the status of the Environment has changed to ready: it means that the application has

been deployed. Open Murano Dashboard, navigate to Environment list and browse the contents of the TestHello environment there. You'll see that the 'Last Operation' column near the "Demo" component says "Hello, World!" - that's the reporting made by our application:

Displaying 1 item

Name	Type	Status	Last operation
Demo	-	Ready	Hello, World!

Displaying 1 item

This concludes the first part of our course. We've created a Murano Application Package, added a manifest describing its contents, written a class which reports a "Hello, World" message, packed all of these into a package archive and uploaded it to Murano Catalog and finally deployed an Environment with this application added.

In the next part we will learn how to improve this application in various aspects, both from users' and developers' perspectives.

Part 2: Customizing your Application Package

We've built a classic "Hello, World" application during the first part of this tutorial, now let's play a little with it and customize it for better user and developer experience - while learning some more Murano features, of course.

Adding user input

Most deployment scenarios for cloud applications require user input. It may be various options which should be applied in software configuration files, passwords for default administrator's accounts, IP addresses of external services to register with and so on. Murano Application Packages may define the user inputs they expect, prompt the end-users to pass the values as these inputs, so that they may utilize these values during application lifecycle workflows.

In Murano user input is defined for each class as *input properties*. *Properties* are object-level variables of the class, they may be of different kinds, and the *input properties* are the ones which are expected to contain user input. See *Properties* for details on other kinds of them.

To define properties of the class you should add a `Properties` block somewhere in the YAML file of that class.

Note: Usually it is better to place this block after the `Name` and `Extends` blocks but before the `Methods` block. Following this suggestion will improve the overall readability of your code.

The `Properties` block should contain a YAML dictionary, mapping the names of the properties to their descriptions. These descriptions may specify the kind of properties, the restrictions on the type and value of the property (so-called *contracts*), provide default value for the property and so on.

Let's add some user input to our "Hello, World" application. Let's ask the end user to provide their name, so the application will greet the user instead of the whole world. To do that, we need to edit our `com.yourdomain.HelloWorld` class to look the following way:

```

1 Name: com.yourdomain.HelloWorld
2
3 Extends: io.murano.Application
4
5 Properties:
6   username:
7     Usage: In
8     Contract: $.string().NotNull()
9
10 Methods:
11 deploy:
12   Body:
13     - $reporter: $this.find('io.murano.Environment').reporter
14     - $reporter.report($this, "Hello, World!")

```

On line 6 we declare a property named `username`, on line 7 we specify that it is an input property, and on line 8 we provide a contract, i.e. a restriction on the value. This particular one states that the property's value should be a string and should not be null (i.e. should be provided by the user).

Note: Although there are a total of 7 different kinds of properties, it turns out that the input ones are the most common. So, for input properties you may omit the `Usage` part - all the properties without an explicit usage are considered to be input properties.

Once the property is declared within the `Properties` block, you may access it in the code of the class methods. Since the properties are object-level variables they may be accessed by calling a `$this` variable (which is a reference to a current instance of your class) followed by a dot and a property name. So, our `username` property may be accessed as `$this.username`.

Let's modify the `deploy` method of our class to make use of the property to greet the user by name:

```

Methods:
deploy:
  Body:
    - $reporter: $this.find('io.murano.Environment').reporter
    - $reporter.report($this, "Hello, " + $this.username + "!")

```

OK, let's try it. Save the file and archive your package directory again, then re-import your zip-file to the Murano Catalog as a package. You'll probably get a warning, since the package with the same name already exists in the catalog (we imported it there in the previous part of the tutorial), so `murano CLI` will ask you if you want to update it. In production it is better to make a newer version of our application and thus to have both in the catalog, but for now let's just overwrite the old package with the new one.

But you cannot deploy it with the old json input we used in the previous part: since the property's contract has that `.NotNull()` part it means that the input should contain the value for the property. If you attempt to deploy an application without this value, you'll get an error.

So, let's edit the `input.json` file we created in the previous part and add the value of the property to the input:

```

1  [
2    {
3      "op": "add",
4      "path": "/-",
5      "value": {
6        "?": {
7          "name": "Demo",
8          "type": "com.yourdomain.HelloWorld",
9          "id": "42"
10       },
11      "username": "Alice"
12     }
13   }
14  ]

```

Save the json file and repeat the steps from the previous part to create an environment, open a configuration session, add an application and deploy it. Now in the 'Last Operation' of Murano Dashboard you will see the updated reporting message, containing the username:

Displaying 1 item

Name	Type	Status	Last operation
HelloWorld	com.yourdomain.HelloWorld	Ready	Hello, Alice!

Displaying 1 item

Adding user interface

As you can see in all the examples above, deploying applications via Murano CLI is quite a cumbersome process: the user has to create environments and sessions and provide the appropriate json-based input for the application.

This is inconvenient for a real user, of course. The CLI is intended to be used by various external automation systems which interact with Murano via scripts, but the human users will use Murano Dashboard which simplifies all those actions and provides a nice interface for them.

Murano Dashboard provides a nice interface to create and deploy environments and manages sessions transparently for the end users, but when it comes to the generation of input JSON it can't do it out of the box: it needs some hints from the package developer. By having hints, Murano Dashboard will be able to generate nicely looking wizard-like dialogs to configure applications and add them to an environment. In this section we'll learn how to create these UI hints.

The UI hints (also called *UI definitions*) should be defined in a separate YAML file (yeah, YAML again) in your application package. The file should be named `ui.yaml` and placed in a special directory of your package called `UI`.

The main section which is mandatory for all the UI definitions is called `Application`: it defines the object structure which should be passed as the input to Murano. That's it: it is equivalent to the JSON `input.json` we were creating before. The data structure remains the same: `?`-header is for system properties and all other properties belong inside the top level of the object.

The `Application` section for our modified "Hello, World" application should look like this:


```

1 Application:
2   ?:
3     type: com.yourdomain.HelloWorld
4     username: Alice

```

This input is almost the same as the `input.json` we used last time, except that the data is expressed in a different format. However, there are several important differences: there are not JSON-Patch related keywords ("op", "path" and "value") - that's because Murano Dashboard will generate them automatically.

Same is true for the missing `id` and `name` from the `?`-header of the object: the dashboard will generate the `id` on its own and ask the end-user for the name, and then will insert both into the structure it sends to Murano.

However, there is one problem in the example above: it has the `username` hardcoded to be `Alice`. Of course we do not want the user input to be hardcoded: it won't be an input then. So, let's define a user interface which will ask the end user for the actual value of this parameter.

Since Murano Dashboard works like a step-by-step wizard, we need to define at least one wizard step (so-called *form*) and place a single text-box control into it, so the end-user will be able to enter his/her name there.

These steps are defined in the `Forms` section of our ui definition file. This section should contain a list of key-value pairs. Keys are the identifiers of the forms, while values should define a list of *field* objects. Each field may define a name, a type, a description, a requirement indicator and some other attributes intended for advanced usage.

For our example we need a single step with a single text field. The `Forms` section should look like this:

```

1 Forms:
2   - step1:
3     fields:
4       - name: username
5         type: string
6         description: Username of the user to say 'hello' to
7         required: true

```

This defines the needed textbox control in the ui. Finally, we need to bind the value user puts into that textbox to the appropriate position in our `Application` section. To do that we replace the hardcoded value with an expression of form `$.<formId>.<fieldName>`. In our case this will be `$step1.username`.

So, our final UI definition will look like this:

```

1 Application:
2   ?:
3     type: com.yourdomain.HelloWorld
4     username: $.step1.username
5
6 Forms:
7   - step1:
8     fields:
9       - name: username
10      type: string

```

(continues on next page)

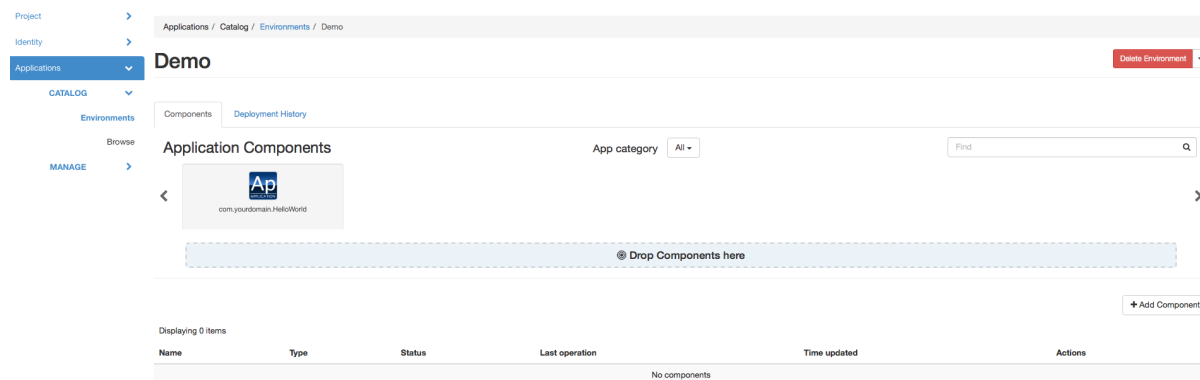
(continued from previous page)

```
11 description: Username of the user to say 'hello' to
12 required: true
```

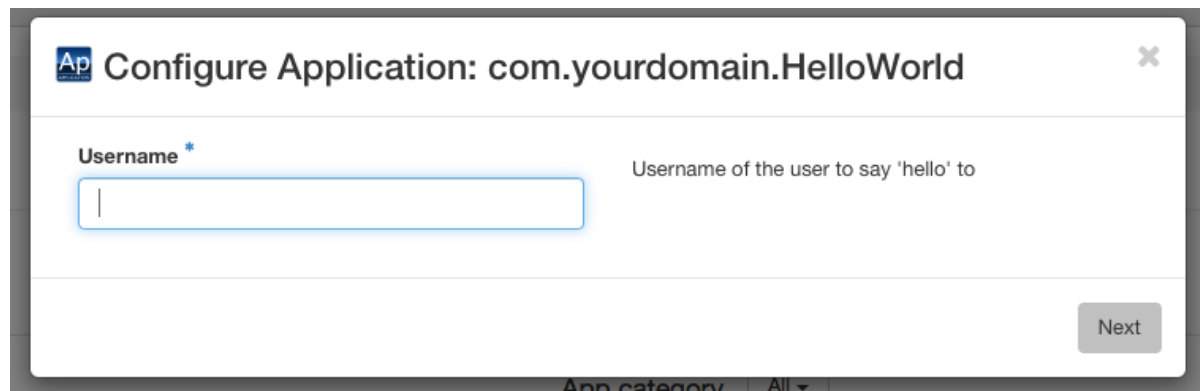
Save this code into your UI/ui .yaml file and then re-zip your package directory and import the resulting archive to Murano Catalog again.

Now, let's deploy this application using Murano Dashboard.

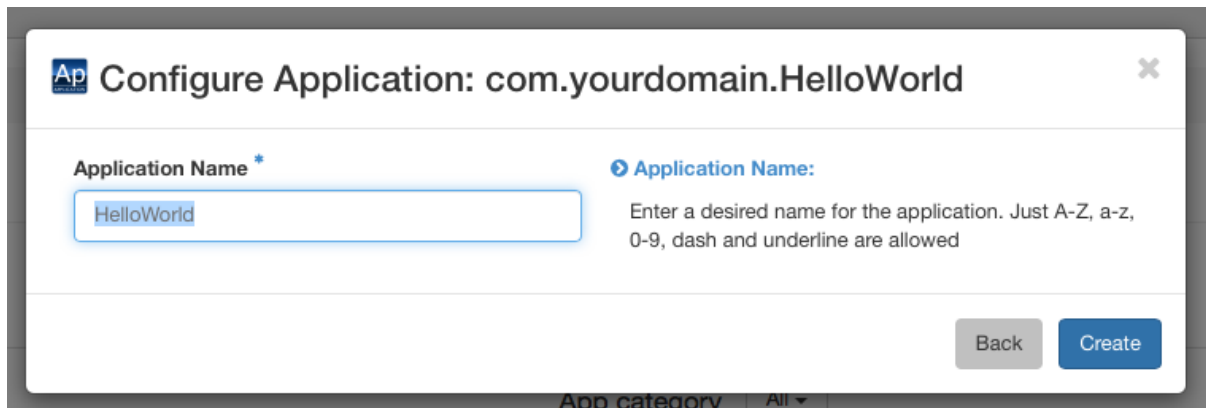
Open Murano Dashboard with your browser, navigate to "Applications/Catalog/Environments" panel, click the "Create Environment" button, enter the name for your environment and click "Create". You'll be taken to the contents of your environment: you'll see that it is empty, but on top of the screen there is a list of components you may add to it. If your Murano Catalog was empty when you started this tutorial, this list will contain just one item: your "Hello, World" application. The screen should look like this:



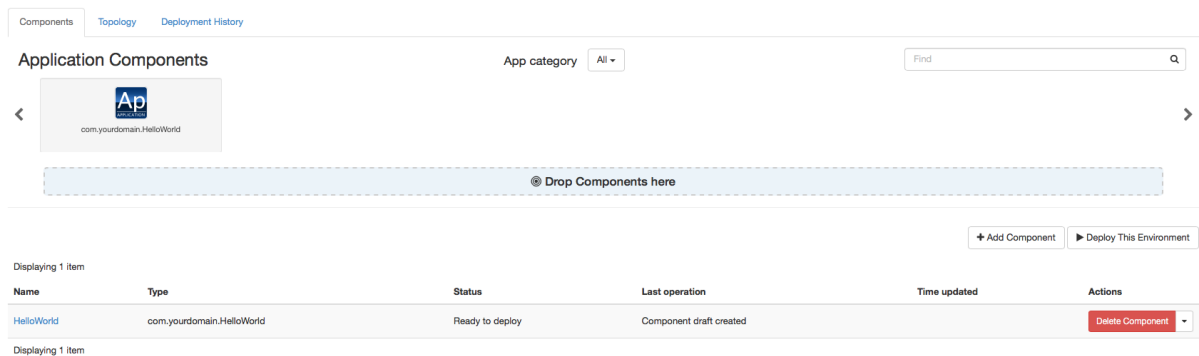
Drag-n-drop your "com.yourdomain.HelloWorld" application from the list on top of the screen to the "Drop components here" panel beneath it. You'll see a dialog, prompting you to enter a username:



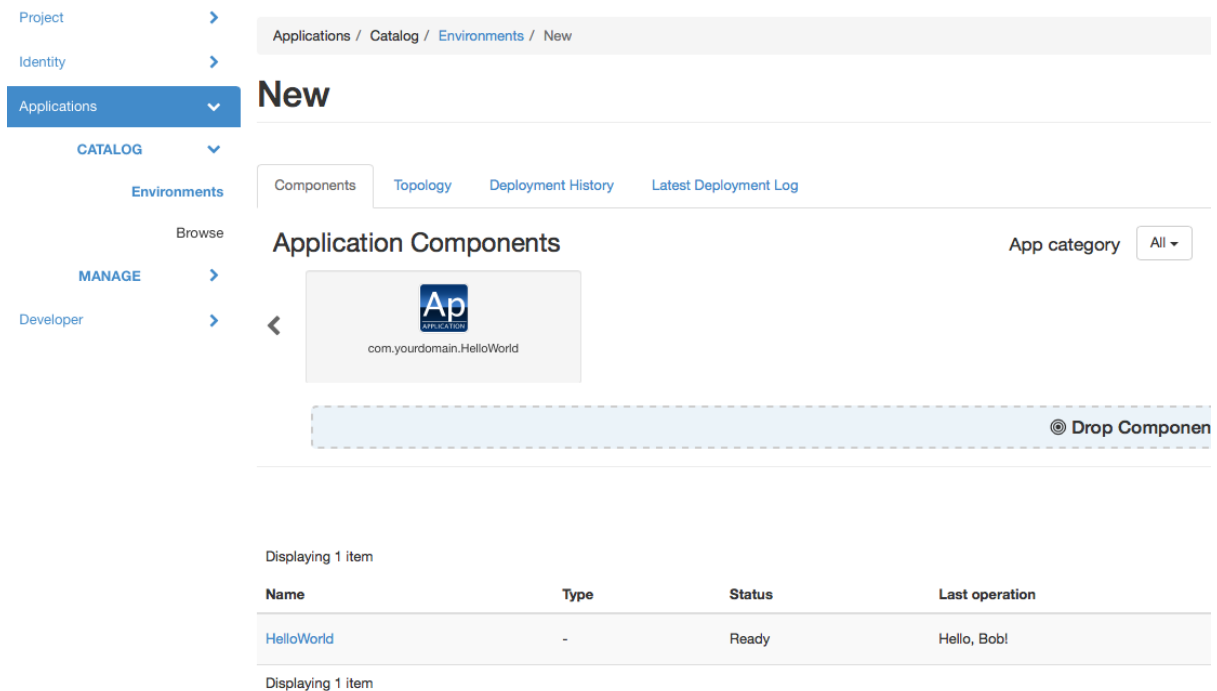
Enter the name and click "Next". Although you've configured just one step of the wizard, the actual interface will consist of two: the dashboard always adds a final step to prompt the user to enter the name of the application instance within the environment:



When you click "Create" button an instance of your application will be added to the environment, you'll see it in the list of components:



So, now you may click the "Deploy this Environment" button and the application will greet the user with the name you've entered.



Simplifying code: namespaces

Now that we've learned how to simplify the user's life by adding a UI definition, let's simplify the developer's life a bit.

When you were working with Murano classes in the previous part you probably noticed that the long class names with all those domain-name-based segments were hard to write and that it was easy to make a mistake:

```
1 Name: com.yourdomain.HelloWorld
2
3 Extends: io.murano.Application
4
5 Methods:
6   deploy:
7     Body:
8       - $reporter: $this.find('io.murano.Environment').reporter
9       - $reporter.report($this, "Hello, World!")
```

To simplify the code we may use the concept of *namespaces* and *short names*. All but last segments of a long class name are namespaces, while the last segment is a short name of a class. In our example `com.yourdomain` is a namespace while the `HelloWorld` is a short name.

Short names have to be unique only within their namespace, so they tend to be expressive, short and human readable, while the namespaces are globally unique and thus are usually long and too detailed.

Murano provides a capability to abbreviate long namespaces with a short alias. Unlike namespaces, aliases don't need to be globally unique: they have to be unique only within a single file which uses them. So, they may be very short. So, in your file you may abbreviate your `com.yourdomain` namespace as `my`, and standard Murano's `io.murano` as `std`. Then instead of a long class name you may write a namespace alias followed by a colon character and then a short name, e.g. `my:HelloWorld` or `std:Application`. This becomes very helpful when you have lots of class names in your code.

To use this feature, declare a special section called `Namespaces` in your class file. Inside that section provide a mapping of namespace aliases to full namespaces, like this:

```
Namespaces:
  my: com.yourdomain
  std: io.murano
```

Note: Since namespaces are often used in all other sections of files it is considered good practice to declare this section at a very top of your class file.

Quite often there is a namespace which is used much more often than others in a given file. In this case it would be beneficial to declare this namespace as a *default namespace*. Default namespace does not need a prefix at all: you just type short name of the class and Murano will interpret it as being in your default namespace. Use '=' character to declare the default namespace in your namespaces block:

```
1 Namespaces:
2   =: com.yourdomain
3   std: io.murano
```

(continues on next page)

(continued from previous page)

```

4
5 Name: HelloWorld
6
7 Extends: std:Application
8
9 Methods:
10   deploy:
11     Body:
12       - $reporter: $this.find(std:Environment).reporter
13       - $reporter.report($this, "Hello, World!")

```

Notice that Name definition at line 5 uses the default namespace: the HelloWorld is not prefixed with any namespaces, but is properly resolved to `com.yourdomain.HelloWorld` because of the default namespace declaration at line 2. Also, because Murano recognizes the `ns:Class` syntax there is no need to enclose `std:Environment` in quote marks, though it will also work.

Adding more info for the catalog

As you could see while browsing Murano Catalog your application entry in it is not particularly informative: the user can't get any description about your app, and the long domain-based name is not very user-friendly either.

This can easily be improved. The `manifest.yaml` which we wrote in the first part contained only mandatory fields. This is how it should look by now:

```

1 FullName: com.yourdomain.HelloWorld
2 Type: Application
3 Description: |
4   A package which demonstrates
5   development for Murano
6   by greeting the user.
7 Classes:
8   com.yourdomain.HelloWorld: HelloWorld.yaml

```

Let's add more fields here.

First, you can add a Name attribute. Unlike FullName, it is not a unique identifier of the package. But, if specified, it overrides the name of the package that is displayed in the catalog.

Then an Author field: here you can put your name or the name of your company, so it will be displayed in catalog as the name of the package developer. If this field is omitted, the catalog will consider the package to be made by "OpenStack", so don't forget this field if you care about your copyright.

When you add these fields your manifest may look like this:

```

1 FullName: com.yourdomain.HelloWorld
2 Type: Application
3 Name: 'Hello, World'
4 Description: |
5   A package which demonstrates
6   development for Murano

```

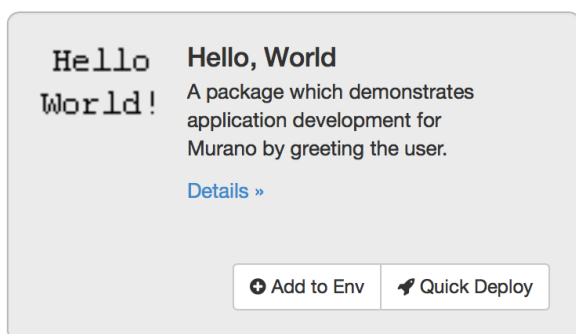
(continues on next page)

(continued from previous page)

```
7   by greeting the user.  
8   Author: John Doe  
9   Classes:  
10  com.yourdomain.HelloWorld: HelloWorld.yaml
```

You may also add an icon to be displayed for your application. To do that just place a `logo.png` file with an appropriate image into the root folder of your package.

Zip the package directory and re-upload the file to the catalog. Then use Murano Dashboard and navigate to Applications/Catalog/Browse panel. You'll see that your app gets a logo, a more appropriate name and a description:



So, here we've learned how to improve both the user's and developer's experience with developing Murano application packages. That was all we could do with the oversimplistic "Hello World" app. Let's move forward and touch some real-life applications.

Part 3: Creating a Plone CMS application package

If you've completed "Hello, World" scenarios in the previous parts and are ready for some serious tasks, we've got a good example here.

Let's automate the deployment of some real application. We've chosen a "Plone CMS" for this purpose. Plone is a simple, but powerful and flexible Content Management System which can efficiently run on cloud. Its deployment scenario can be very simple for demo cases and can become really complicated for production-grade usage. So it's a good playground: in this part we'll create a Murano application to address the simplest scenario, then we will gradually add more features of production-grade deployments.

Note: To learn more about Plone, its features, capabilities and deployment scenarios you may visit the [Official website of Plone Foundation](#).

The goal

Simplest deployment of Plone CMS requires a single server, or, in the case of OpenStack, a Virtual Machine, to run on. Then a software should be downloaded and configured to run on that server.

So, as a bare minimum our Plone application package for Murano should automate the following steps:

1. Provision a virtual machine in OpenStack (VM);
2. Configure ths VM's network connectivity and security;
3. Download a distribution of Plone from Internet to the virtual machine;
4. Install the distribution and configure some of its parameters with user input.

Preparation

First let's revisit what we've learned in previous parts and create a new application package with its manifest and create a class file to contain the logic of your app.

Create a new directory for a package, call it `PloneApp`. Create a `manifest.yaml` file as described in part 1 of this tutorial in the root of the package and fill it with data: name your package `com.yourdomain.Plone`, set its type to `Application`, give it a display name of "Plone CMS" and put your name as the author of the package:

```

1 FullName: com.yourdomain.Plone
2 Name: Plone CMS
3 Description: Simple Plone Deployment
4 Type: Application
5 Author: John Doe

```

Then create a `Classes` sub directory inside your package directory and create a `plone.yaml` there. This will be your application class.

At the top of this file declare a `Namespace` section: this will simplify the code and save time on typing long class names. Make your namespace (`com.yourdomain`) a default namespace of the file, also include the standard namespace for Murano applications - `io.murano`, alias it as `std`.

Don't forget to include the `Name` of your class. Since you've declared a default namespace for a file you can name your class without a need to type its long part, just using the shortname.

Also include the `Extends` section: same as in our "Hello, World" example this application will inherit the `io.murano.Application` class, but since we've aliased this namespace as well, it may be shortened to `std:Application`

By now your class file should look like this:

```

Namespaces:
  :=: com.yourdomain
  std: io.murano

Name: Plone

Extends: std:Application

```

We'll add the actual logic in the next section. Now, save the file and include it into the `Classes` section of your `manifest.yaml`, which should now look like this:

```
1 FullName: com.yourdomain.Plone
2 Name: Plone CMS
3 Description: Simple Plone Deployment
4 Type: Application
5 Author: John Doe
6 Classes:
7   com.yourdomain.Plone: plone.yaml
```

You are all set and ready to go. Let's add the actual deployment logic.

Library classes

Murano comes bundled with a so-called "Murano Core Library" - a Murano Package containing the classes to automate different scenarios of interaction with other entities such as OpenStack services or virtual machines. They follow object-oriented design: for example, there is a Murano class called `Instance` which represents an OpenStack virtual machine: if you create an object of this class and execute a method called `deploy` for it Murano will do all the needed system calls to OpenStack Services to orchestrate the provisioning of a virtual machine and its networking configuration. Then this object will contain information about the state and configuration of the VM, such as its hostname, ip addresses etc. After the VM is provisioned you can use its object to send the configuration scripts to the VM to install and configure software for your application.

Other OpenStack resources such as Volumes, Networks, Ports, Routers etc also have their corresponding classes in the core library.

Provisioning a VM

When creating your application package you can *compose* your application out of the components of core library. For example for an application which should run on a VM you can define an input property called `instance` and restrict the value type of this property to the aforementioned `Instance` class with a contract.

Let's do that in the `plone.yaml` class file you've created. First, add a new namespace alias to your `Namespaces` section: shorten `io.murano.resources` as `res`. This namespace of the core library contains all the resource classes, including the `io.murano.resources.Instance` which we need to define the virtual machine:

```
Namespaces:
  :=: com.yourdomain
  std: io.murano
  res: io.murano.resources
```

Now, let's add an input property to your class:

```
1 Properties:
2   instance:
3     Usage: In
4     Contract: $.class(res:Instance)
```


Notice the contract at line 4: it limits the values of this property to the objects of class `io.murano.resources.Instance` or its subclasses.

This defines that your application needs a virtual machine. Now let's ensure that it is provisioned - or provision it otherwise. Add a `deploy` method to your application class and call instance's `deploy` method from it:

```
1 Methods:
2   deploy:
3     Body:
4     - $this.instance.deploy()
```

That's very simple: you just access the `instance` property of your current object and run a method `deploy` for it. The core library defines this method of the `Instance` class in an *idempotent* manner: you may call it as many times as you want: the first call will actually provision the virtual machine in the cloud, while all the subsequent calls will do nothing, thus you may always call this method to ensure that the VM was properly provisioned. It's important since we define it as an input property: theoretically a user can pass an already-provisioned VM object as input, but you need to be sure. Always calling the `deploy` method is the best practice to follow.

Running a command on the VM

Once the VM has been provisioned you may execute various kinds of software configuration scenarios on it to install and configure the actual application on the VM. Murano supports different types of software configuration tools to be run on a VM, but the simplest and the most common type is just a shell script.

To run a shell script on a virtual machine you may use a *static method* `runCommand` of class `io.murano.configuration.Linux`. Since this method is static you do not need to create any objects of its class: you can just do something like:

```
- type('io.murano.configuration.Linux').runCommand($server.agent, 'sudo apt-
↳get update')
```

or, if we declare another namespace prefix

```
Namespaces:
...
conf: io.murano.configuration
```

this may be shortened to

```
- conf:Linux.runCommand($server.agent, 'sudo apt-get update')
```

In this case `$server` should be a variable containing an object of `io.murano.resources.Instance` class, everything you pass as a second argument (`apt get update` in the example above) is the shell command to be executed on a VM. You may pass not just a single line, but a multi-line text: it will be treated as a shell script.

Note: The shell scripts and commands you send to a VM are executed by a special software component running on the VM - a *murano agent*. For the most popular distributions of Linux (Debian, Ubuntu, Centos, Fedora, etc.) it automatically gets installed on the VM once it is provisioned, but for other

distribution and non-Linux OSes it has to be manually pre-installed in the image. See *Building Murano Image* for details.

Loading a script from a resource file

Passing strings as a second argument of a `runCommand` method is convenient for short commands like the `apt-get update` shown in an example above. However for larger scripts it is not that useful. Instead it is preferable to load a script text from a file and run it. You can do that in Murano.

For example, let's make a script which downloads, unpacks, installs and configures Plone CMS on our VM. First, create a directory called `Resources` inside your package directory. Then, create a file named `install-plone.sh` and put the following script there:

```
#!/bin/bash

#input parameters

PL_PATH="$1"
PL_PASS="$2"
PL_PORT="$3"

# Write log. Redirect stdout & stderr into log file:
exec &> /var/log/runPloneDeploy.log
# echo "Update all packages."
sudo apt-get update

# Install the operating system software and libraries needed to run Plone:
sudo apt-get install python-setuptools python-dev build-essential libssl-dev
↳libxml2-dev libxslt1-dev libbz2-dev libjpeg62-dev

# Install optional system packages for the handling of PDF and Office files.
↳Can be omitted:
sudo apt-get install libreadline-dev wv poppler-utils

# Download the latest Plone unified installer:
wget --no-check-certificate https://launchpad.net/plone/5.0/5.0.4/+download/
↳Plone-5.0.4-UnifiedInstaller.tgz

# Unzip the latest Plone unified installer:
tar -xvf Plone-5.0.4-UnifiedInstaller.tgz
cd Plone-5.0.4-UnifiedInstaller

# Set the port that Plone will listen to on available network interfaces.
↳Editing "http-address" param in buildout.cfg file:
sed -i "s/^http-address = [0-9]*$/http-address = ${PL_PORT}/" buildout_
↳templates/buildout.cfg

# Run the Plone installer in standalone mode
```

(continues on next page)

(continued from previous page)

```
./install.sh --password="${PL_PASS}" --target="${PL_PATH}" standalone

# Start Plone
cd "${PL_PATH}/zinstance"
bin/plonectl start
```

Note: As you can see, this script uses apt to install the prerequisite software packages, so it expects a Debian-compatible Linux distro as the VM operating system. This particular script was tested on Ubuntu 14.04. Other distros may have a different set of preinstalled software and thus require different additional prerequisites.

The comments in the script give the needed explanation: the script installs all the prerequisites, downloads a tar.gz archive with a distribution of Plone, unpacks it, edits the `buildout.cfg` file to specify the port Plone will listen at, then runs the installation script which is included in the distribution. When that script is finished, the Plone daemon is started.

Save the file as `Resources/install-plone.sh`. Now you may load its contents into a string variable in your class file. To do that, you need to use another static method: a `string()` method of a `io.murano.system.Resources` class:

```
- $script: type('io.murano.system.Resources').string('install-plone.sh')
```

or, with the introduction of another namespace prefix

```
- $script: sys:Resources.string('install-plone.sh')
```

But before sending this script to a VM, it needs to be parametrized: as you can see in the script snippet above, it declares three variables which are used to set the installation path in the VM's filesystem, a default administrator's password and a listening port. In the script these values are initialized with stubs \$1, \$2 and \$3, now we need to replace these stubs with the actual user input. To do that our class needs to define the appropriate input properties and then do string replacement.

First, let's define the appropriate input properties in the `Properties` block of the class, right after the `instance` property:

```
1 Properties:
2   instance:
3     Usage: In
4     Contract: $.class(res:Instance)
5
6   installationPath:
7     Usage: In
8     Contract: $.string().NotNull()
9     Default: '/opt/plone'
10
11  defaultPassword:
12    Usage: In
13    Contract: $.string().NotNull()
14
15  listeningPort:
```

(continues on next page)

(continued from previous page)

```

16 Usage: In
17 Contract: $.int().NotNull()
18 Default: 8080

```

Now, let's replace the stub values in that script value we've loaded into the `$script` variable. This may be done using a `replace` function:

```

- $script: $script.replace({"$1" => $this.installationPath,
    "$2" => $this.defaultPassword,
    "$3" => $this.listeningPort})

```

Finally, the resulting `$script` variable may be passed as a second argument of a `runCommand` method, while the first one should be the `instance` property, containing our VM-object:

```

- conf:Linux.runCommand($this.instance.agent, $script)

```

Configuring OpenStack Security

By now we've got code which provisions a VM and a script which deploys and configures Plone on it. However, in most OpenStack clouds this is not enough: usually all incoming traffic to all the VMs is blocked by default, so we need to configure security group of OpenStack to allow the incoming http calls to our VM on the port our Plone server listens at.

To do that we need to use a `securityGroupManager` property of the `Environment` class which owns our application. That property contains an object of type `io.murano.system.SecurityGroupManager`, which defines a `addGroupIngress` method. This method allows us to add a security group rule to allow incoming traffic of some type through a specific port within a port range. It accepts a list of YAML objects, each having four keys: `FromPort` and `ToPort` to define the boundaries of the port range, `IpProtocol` to define the type of the protocol and `External` boolean flag to indicate if the incoming traffic should be allowed to originate from outside of the environment (if this flag is false, the traffic will be accepted only from the VMs deployed by the application in the same Murano environment).

Let's do this in code:

```

1 - $environment: $this.find(std:Environment)
2 - $manager: $environment.securityGroupManager
3 - $rules:
4   - FromPort: $this.listeningPort
5     ToPort: $this.listeningPort
6     IpProtocol: tcp
7     External: true
8 - $manager.addGroupIngress($rules)
9 - $environment.stack.push()

```

It's quite straightforward, just notice the last line. It is required, because current implementation of `SecurityGroupManager` relies on Heat underneath - it modifies the *Heat Stack* associated with our environment, but does not apply the changes to the actual cloud. To apply them the stack needs to be *pushed*, i.e. submitted to Heat Orchestration service. The last line does exactly that.

Notifying end-user on Plone location

When the deployment is completed and our instance of Plone server starts listening on a provisioned virtual machine, the end user has one last question to solve: to find out where it is. Of course, the user may use OpenStack Dashboard to list all the provisioned VMs, find the one which has just been created and look for its IP address. But that's inconvenient. It would be much better if Murano notified the end-user on where to find Plone once it is ready.

We may utilize the same approach we used in the previous parts to say "Hello, World" - call a `report` method of `reporter` attribute of the `Environment` class. The tricky part is getting the IP address.

Class `io.murano.resources.Instance` has an *output property* called `ipAddresses`. Unlike input properties the output ones are not provided by users but are set by objects themselves while their methods are executed. The `ipAddresses` is assigned during the execution of `deploy` method of the VM. The value is the list of ip addresses assigned to different interfaces of the machine. Also, if the `assignFloatingIp` input property is set to `true`, another output property will be set during the execution of `deploy` - a `floatingIpAddress` will contain the floating ip attached to the VM.

Let's use this knowledge and build a proper report message:

```

1 - $message: 'Plone is up and running at '
2 - If: $this.instance.assignFloatingIp
3   Then:
4     - $message: $message + $this.instance.floatingIpAddress
5   Else:
6     - $message: $message + $this.instance.ipAddresses.first()
7 - $message: $message + ":" + str($this.listeningPort)
8 - $environment.reporter.report($this, $message)

```

Note the usage of `If` expression: it is similar to other programming languages, just uses YAML keys to define the "if" and "else" blocks.

This code creates a string variable called `$message`, initializes it with the beginning of the message string, then appends either a floating ip address of the VM (if it's set) or the first of the regular ips otherwise. Then it appends a listening port after a colon character - and reports the resulting message to the user.

Completing the Plone class

We've got all the pieces to deploy our Plone application, now let's combine them together. Our final class file should look like this:

```

Namespaces:
  =: com.yourdomain
  std: io.murano
  res: io.murano.resources
  sys: io.murano.system

Name: Plone

Extends: std:Application

Properties:

```

(continues on next page)

```

instance:
  Usage: In
  Contract: $.class(res:Instance)

installationPath:
  Usage: In
  Contract: $.string().notNull()
  Default: '/opt/plone'

defaultPassword:
  Usage: In
  Contract: $.string().notNull()

listeningPort:
  Usage: In
  Contract: $.int().notNull()
  Default: 8080

Methods:
deploy:
  Body:
  - $this.instance.deploy()
  - $script: sys:Resources.string('install-plone.sh')
  - $script: $script.replace({
    "$1" => $this.installationPath,
    "$2" => $this.defaultPassword,
    "$3" => $this.listeningPort
  })
  - type('io.murano.configuration.Linux').runCommand($this.instance.agent,
↪ $script)
  - $environment: $this.find(std:Environment)
  - $manager: $environment.securityGroupManager
  - $rules:
    - FromPort: $this.listeningPort
      ToPort: $this.listeningPort
      IpProtocol: tcp
      External: true
    - $manager.addGroupIngress($rules)
  - $environment.stack.push()
  - $formatString: 'Plone is up and running at {0}:{1}'
  - If: $this.instance.assignFloatingIp
    Then:
      - $address: $this.instance.floatingIpAddress
    Else:
      - $address: $this.instance.ipAddresses.first()
  - $message: format($formatString, $address, $this.listeningPort)
  - $environment.reporter.report($this, $message)

```

That's all, our class is ready.

Providing a UI definition

Last but not least, we need to add a UI definition file to define a template for the user input and create wizard steps.

This time both are a bit more complicated than they were for the "Hello, World" app.

First, let's create the wizard steps. It's better to decompose the UI into two steps: the first one will define the properties of a Virtual Machine, and the second one the configuration properties of the Plone application itself.

```

1 Forms:
2   - instanceConfiguration:
3     fields:
4       - name: hostname
5         type: string
6         required: true
7       - name: image
8         type: image
9         imageType: linux
10      - name: flavor
11        type: flavor
12      - name: assignFloatingIp
13        type: boolean
14   - ploneConfiguration:
15     fields:
16       - name: installationPath
17         type: string
18       - name: defaultPassword
19         type: password
20         required: true
21       - name: listeningPort
22         type: integer

```

This is familiar to what we had on the previous step, however there are several new types of fields: while the types `integer` and `boolean` are quite obvious - they will render a numeric up-and-down textbox and checkbox controls respectively - other field types are more specific.

Field of type `image` will render a drop-down list allowing you to choose an image for your VM, and the list of images will contain only the ones having appropriate metadata associated (the type of metadata is defined by the `imageType` attribute: this particular example requires it to be tagged as "Generic Linux").

Field of type `flavor` will render a drop-down list allowing you to choose a flavor for your VM among the ones registered in Nova.

Field of type `password` will render a pair of text-boxes in a password input mode (i.e. hiding all the input with '*'-characters). The rendered field will have appropriate validation: it will ensure that the values entered in both fields are identical (thus providing a "repeat password" functionality) and will also enforce password complexity check.

This defines the basic UI, but it is not particularly user friendly: when MuranoDashboard renders the wizard it will label appropriate controls with the names of the fields, but they usually don't look informative and pretty.

So, to improve the user experience you may add additional attributes to field descriptors here. `label`

attribute allows you to define a custom label to be rendered next to appropriate control, `description` allows you to provide a longer text to be displayed on the form as a description of the control, and, finally, an `initial` attribute allows you define the default value to be entered into the control when it is shown to the end-user.

Modify the Forms section to use these attributes:

```

1 Forms:
2   - instanceConfiguration:
3     fields:
4       - name: hostname
5         type: string
6         label: Host Name
7         description: >-
8           Enter a hostname for a virtual machine to be created
9         initial: plone-vm
10        required: true
11       - name: image
12         type: image
13         imageType: linux
14         label: Instance image
15         description: >-
16           Select valid image for the application. Image should already be
17           ↪prepared and
18           registered in glance.
19       - name: flavor
20         type: flavor
21         label: Instance flavor
22         description: >-
23           Select registered in Openstack flavor. Consider that application
24           ↪performance
25           depends on this parameter.
26       - name: assignFloatingIp
27         type: boolean
28         label: Assign Floating IP
29         description: >-
30           Check to assign floating IP automatically
31   - ploneConfiguration:
32     fields:
33       - name: installationPath
34         type: string
35         label: Installation Path
36         initial: '/opt/plone'
37         description: >-
38           Enter the path on the VM filesystem to deploy Plone into
39       - name: defaultPassword
40         label: Admin password
41         description: Default administrator's password
42         type: password
43         required: true
44       - name: listeningPort

```

(continues on next page)

(continued from previous page)

```

43     type: integer
44     label: Listening Port
45     description: Port to listen at
46     initial: 8080

```

Now, let's add an Application section to provide templated input for our app:

```

1  Application:
2  ?:
3     type: com.yourdomain.Plone
4  instance:
5     ?:
6         type: io.murano.resources.LinuxMuranoInstance
7         name: $.instanceConfiguration.hostname
8         image: $.instanceConfiguration.image
9         flavor: $.instanceConfiguration.flavor
10        assignFloatingIp: $.instanceConfiguration.assignFloatingIp
11        installationPath: $.ploneConfiguration.installationPath
12        defaultPassword: $.ploneConfiguration.defaultPassword
13        listeningPort: $.ploneConfiguration.listeningPort

```

Note the instance part here: since our instance input property is not a scalar value but rather an object, we are placing another object template inside the appropriate section. Note that the type of this object is not `io.murano.resources.Instance` as you could expect based on the property contract, but a more specific class: `LinuxMuranoInstance` in the same namespace. Since this class inherits the former, it matches the contract, but it provides a more appropriate implementation than the base one.

Let's combine the two snippets together, we'll get the final UI definition of our app:

```

1  Application:
2  ?:
3     type: com.yourdomain.Plone
4  instance:
5     ?:
6         type: io.murano.resources.LinuxMuranoInstance
7         name: $.instanceConfiguration.hostname
8         image: $.instanceConfiguration.image
9         flavor: $.instanceConfiguration.flavor
10        assignFloatingIp: $.instanceConfiguration.assignFloatingIp
11        installationPath: $.ploneConfiguration.installationPath
12        defaultPassword: $.ploneConfiguration.defaultPassword
13        listeningPort: $.ploneConfiguration.listeningPort
14  Forms:
15    - instanceConfiguration:
16      fields:
17        - name: hostname
18          type: string
19          label: Host Name
20          description: >-
21            Enter a hostname for a virtual machine to be created

```

(continues on next page)

```
22     initial: 'plone-vm'
23     required: true
24     - name: image
25       type: image
26       imageType: linux
27       label: Instance image
28       description: >-
29         Select valid image for the application. Image should already be
↳prepared and
30         registered in glance.
31     - name: flavor
32       type: flavor
33       label: Instance flavor
34       description: >-
35         Select registered in Openstack flavor. Consider that application
↳performance
36         depends on this parameter.
37     - name: assignFloatingIp
38       type: boolean
39       label: Assign Floating IP
40       description: >-
41         Check to assign floating IP automatically
42     - ploneConfiguration:
43       fields:
44         - name: installationPath
45           type: string
46           label: Installation Path
47           initial: '/opt/plone'
48           description: >-
49             Enter the path on the VM filesystem to deploy Plone into
50         - name: defaultPassword
51           label: Admin password
52           description: Default administrator's password
53           type: password
54           required: true
55         - name: listeningPort
56           type: integer
57           label: Listening Port
58           description: Port to listen at
59           initial: 8080
```

Save this file as a `ui.yaml` in a `UI` folder of your package. As a final touch add a logo to the package - save the image below to the root directory of your package as `logo.png`:



The package is ready. Zip it and import to Murano catalog. We are ready to try it.

Deploying the package

Go to Murano Dashboard, create an environment and add a "Plone CMS" application to it. You'll see the nice wizard with all the field labels and descriptions you've added to the ui definition file:

Configure Application: Plone CMS

Host Name *
plone-vm
Host Name: Enter a hostname for a virtual machine to be created

Instance image *
Debian 8 x64 (pre-installed murano-agent)
Instance image: Select valid image for the application. Image should already be prepared and registered in glance.

Instance flavor *
m1.medium
Instance flavor: Select registered in Openstack flavor. Consider that application performance depends on this parameter.

Assign Floating IP
Assign Floating IP: Check to assign floating IP automatically

Configure Application: Plone CMS

Installation Path *
/opt/plone
Installation Path: Enter the path on the VM filesystem to deploy Plone into

Admin password *
Admin password: Default administrator's password

Confirm password *
Confirm password: Confirm administrator's password

Listening Port *
8080
Listening Port: Port to listen at

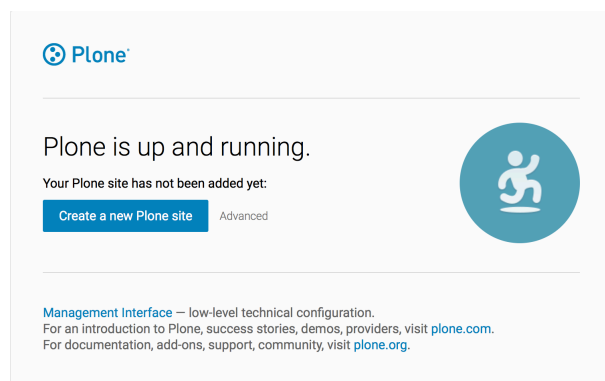
After the app is added to the environment, click the "Deploy this environment" button. The deployment will take about 10 minutes, depending on the speed of the VM's internet connection and the amount of packages to be updated. When it is over, check the "Last operation" column in the environment's list of components near the Plone component. It should contain a message "Plone is up and running at ..." followed by ip address and port:

Displaying 1 item

Name	Type	Status	Last operation
Plone	Plone CMS	Ready	Plone is up and running at 172.16.40.175:8080

Displaying 1 item

Enter this address to the address bar of your browser. You'll see the default management interface of Plone:



If you click a "Create a new Plone site" button you'll be prompted for username and password. Use `admin` username and the password which you entered in the Wizard. See [Plone Documentation](#) for details on how to operate Plone.

This concludes this part of the course. The application package we created demonstrates the basic capabilities of Murano for the deployments of real-world applications. However, the deployed configuration of Plone is not of production-grade service: it is just a single VM with all-in-one service topology, which is not a scalable or fault-tolerant solution. In the next part we will learn some advanced features which may help to bring more production-grade capabilities to our package.

Part 4: Refactoring code to use the Application Framework

Up until this point we wrote the Plone application in a manner that was common to all applications that were written before the application framework was introduced.

In this last tutorial step we are going to refactor the Plone code in order to take advantage of the framework.

Application framework was written in order to simplify the application development and encapsulate common deployment workflows. This gives things primitives for application scaling and high availability without the need to develop them over and over again for each application.

When using the frameworks, an application developer only has to inherit the class that best suits him and provide it only with the code that is specific to the application, while leaving the rest to the framework. This typically includes:

- instructions on how to provision the software on each node (server)
- instructions on how to configure the provisioned software
- server group onto which the software should be installed. This may be a fixed server list, a shared server pool, or a scalable server group that creates servers using the given instance template, or one of the several other implementations provided by the framework

The framework is located in a separate library package `io.murano.applications` that is shipped with Murano. We are going to use the `apps` namespace prefix to refer to this namespace through the code.

Step 1: Add dependency on the App Framework

In order to use one Murano Package from another, the former must be explicitly specified as a requirement for the latter. This is done by filling the `Require` section in the package's manifest file.

Open the Plone's `manifest.yaml` file and append the following lines:

```
Require:
  io.murano.applications:
```

Requirements are specified as a mapping from package name to the desired version of that package (or version range). The missing value indicates the dependency on the latest `0.*.*` version of the package which is exactly what we need since the current version of the app framework library is 0.

Step 2: Get rid of the instance

Since we are going to have a multi-server Plone application there won't be a single instance belonging to the application. Instead, we are going to provide it with the server group that abstracts the server management from the application.

So instead of

```
Properties:
  instance:
    Contract: $.class(res:Instance)
```

we are going to have

```
Properties:
  servers:
    Contract: $.class(apps:ServerGroup).NotNull()
```

Step 3: Change the base classes

Another change that we are going to make to the main application class is to change its base classes. Regular applications inherit from the `std:Application` which only has the method `deploy` that does all the work.

Application framework provides us with its own implementation of that class and method. Instead of one monolithic method that does everything, with the framework, the application provides only the code needed to provision and configure the software on each server.

So instead of `std:Application` class we are going to inherit two of the framework classes:

```
Extends:
- apps:MultiServerApplicationWithScaling
- apps:OpenStackSecurityConfigurable
```

The first class tells us that we are going to have an application that runs on multiple servers. In the following section we are going to split out `deploy` method into two smaller methods that are going to be invoked by the framework to install the software on each of the servers. By inheriting the

`apps:MultiServerApplicationWithScaling`, the application automatically gets all the UI buttons to scale it out and in.

The second class is a mix-in class that tells the framework that we are going to provide the OpenStack-specific security group configuration for the application.

Step 4: Split the deployment logic

In this step we are going to split the installation into two phases: provisioning and configuration.

Provisioning is implemented by overriding the `onInstallServer` method, which is called every time a new server is added to the server group. In this method we are going to install the Plone software bits onto the server (which is provided as a method parameter).

Configuration is done through the `onConfigureServer`, which is called upon the first installation on the server, and every time any of the application settings change, and `onCompleteConfiguration` which is executed on each server after everything was configured so that we can perform post-configuration steps like starting application daemons and reporting messages to the user.

Thus we are going to split the `install-plone.sh` script into two scripts: `installPlone.sh` and `configureServer.sh` and execute each one in their corresponding methods:

```
onInstallServer:
Arguments:
- server:
    Contract: $.class(res:Instance).NotNull()
- serverGroup:
    Contract: $.class(apps:ServerGroup).NotNull()
Body:
- $file: sys:Resources.string('installPlone.sh').replace({
    "$1" => $this.deploymentPath,
    "$2" => $this.adminPassword
  })
- conf:Linux.runCommand($server.agent, $file)

onConfigureServer:
Arguments:
- server:
    Contract: $.class(res:Instance).NotNull()
- serverGroup:
    Contract: $.class(apps:ServerGroup).NotNull()
Body:
- $primaryServer: $serverGroup.getServers().first()
- If: $server = $primaryServer
  Then:
  - $file: sys:Resources.string('configureServer.sh').replace({
    "$1" => $this.deploymentPath,
    "$2" => $primaryServer.ipAddresses[0]
  })
  Else:
  - $file: sys:Resources.string('configureClient.sh').replace({
    "$1" => $this.deploymentPath,
```

(continues on next page)

(continued from previous page)

```

"$2" => $this.servers.primaryServer.ipAddresses[0],
"$3" => $this.listeningPort})
- conf:Linux.runCommand($server.agent, $file)

onCompleteConfiguration:
  Arguments:
    - servers:
      Contract:
        - $.class(res:Instance).notNull()
    - serverGroup:
      Contract: $.class(apps:ServerGroup).notNull()
    - failedServers:
      Contract:
        - $.class(res:Instance).notNull()
  Body:
    - $startCommand: format('{0}/zeocluster/bin/plonectl start', $this.
↪deploymentPath)
    - $primaryServer: $serverGroup.getServers().first()
    - If: $primaryServer in $servers
      Then:
        - $this.report('Starting DB node')
        - conf:Linux.runCommand($primaryServer.agent, $startCommand)
        - conf:Linux.runCommand($primaryServer.agent, 'sleep 10')

    - $otherServers: $servers.where($ != $primaryServer)
    - If: $otherServers.any()
      Then:
        - $this.report('Starting Client nodes')
        # run command on all other nodes in parallel with pselect
        - $otherServers.pselect(conf:Linux.runCommand($.agent,
↪$startCommand))

    # build an address string with IPs of all our servers
    - $addresses: $serverGroup.getServers().
      select(
        switch($.assignFloatingIp => $.floatingIpAddress,
          true => $.ipAddresses[0])
        + ':' + str($this.listeningPort)
      ).join(', ')
    - $this.report('Plone listeners are running at ' + str($addresses))

```

During configuration phase we distinguish the first server in the server group from the rest of the servers. The first server is going to be the primary node and treated differently from the others.

Step 5: Configuring OpenStack security group

The last change to the main class is to set up the security group rules. We are going to do this by overriding the `getSecurityRules` method that we inherited from the `apps:OpenStackSecurityConfigurable` class:

```
getSecurityRules:  
  Body:  
    - Return:  
      - FromPort: $this.listeningPort  
        ToPort: $this.listeningPort  
        IpProtocol: tcp  
        External: true  
      - FromPort: 8100  
        ToPort: 8100  
        IpProtocol: tcp  
        External: false
```

The code is very similar to that of the old `deploy` method with the only difference being that it returns the rules rather than sets them on its own.

Step 6: Provide the server group instance

Do you remember, that previously we replaced the `instance` property with `servers` of type `apps:ServerGroup`? Since the object is coming from the UI definition, we must change the latter in order to provide the class with the `apps:ServerReplicationGroup` instance rather than `resources:Instance`.

To do this we are going to replace the `instance` property in the Application template with the following snippet:

```
servers:  
  ?:  
    type: io.murano.applications.ServerReplicationGroup  
    numItems: $.ploneConfiguration.numNodes  
    provider:  
      ?:  
        type: io.murano.applications.TemplateServerProvider  
    template:  
      ?:  
        type: io.murano.resources.LinuxMuranoInstance  
        flavor: $.instanceConfiguration.flavor  
        image: $.instanceConfiguration.osImage  
        assignFloatingIp: $.instanceConfiguration.assignFloatingIP  
        serverNamePattern: $.instanceConfiguration.unitNamingPattern
```

If you take a closer look at the code above you will find out that the new declaration is very similar to the old one. But now instead of providing the `Instance` property values directly, we are providing them as a template for the `TemplateServerProvider` server provider. `ServerReplicationGroup` is going to use the provider each time it requires another server. In turn, the provider is going to use the familiar template for the new instances.

Besides the instance template we also specify the initial number of Plone nodes using the `numItems` property and the name pattern for the servers. Thus we must also add it to the list of our controls:

Forms:

- **instanceConfiguration:**
 - fields:**
 - ...
 - name: `unitNamingPattern`
 - type: `string`
 - label: `Instance Naming Pattern`
 - required: `false`
 - maxLength: `64`
 - initial: `'plone-{0}'`
 - description:** >-
 - Specify a string, that will be used in instance hostname.
 - Just A-Z, a-z, 0-9, dash and underline are allowed.
- **ploneConfiguration:**
 - fields:**
 - ...
 - name: `numNodes`
 - type: `integer`
 - label: `Initial number of Client Nodes`
 - initial: `1`
 - minValue: `1`
 - required: `true`
 - description:** >-
 - Select the initial number of Plone Client Nodes

Step 6: Using server group composition

By this step we should already have a working Plone application. But let's go one step further and enhance our sample application.

Since we are running the database on the first server group server only, we might want it to have different properties. For example we might want to give it a bigger flavor or just a special name. This is a perfect opportunity for us to demonstrate how to construct complex server groups. All we need to do is to just use another implementation of `apps:ServerGroup`. Instead of `apps:ServerReplicationGroup` we are going to use the `apps:CompositeServerGroup` class, which allows us to compose several server groups together. One of them is going to be a single-server server group consisting of our primary server, and the second is going to be the scalable server group that we used to create in the previous step.

So again, we change the `Application` section of our UI definition file with even a more advanced servers property definition:

```

servers:
  ?:
    type: io.murano.applications.CompositeServerGroup
    serverGroups:
      - ?:
        type: io.murano.applications.SingleServerGroup

```

(continues on next page)

(continued from previous page)

```

server:
  ?:
    type: io.murano.resources.LinuxMuranoInstance
    name: format($.instanceConfiguration.unitNamingPattern, 'db')
    image: $.instanceConfiguration.image
    flavor: $.instanceConfiguration.flavor
    assignFloatingIp: $.instanceConfiguration.assignFloatingIp
- ?:
  type: io.murano.applications.ServerReplicationGroup
  numItems: $.ploneConfiguration.numNodes
  provider:
    ?:
      type: io.murano.applications.TemplateServerProvider
    template:
      ?:
        type: io.murano.resources.LinuxMuranoInstance
        flavor: $.instanceConfiguration.flavor
        image: $.instanceConfiguration.osImage
        assignFloatingIp: $.instanceConfiguration.assignFloatingIP
        serverNamePattern: $.instanceConfiguration.unitNamingPattern

```

Here the instance definition for the `SingleServerGroup` (our primary server) differs from the servers in the `ServerReplicationGroup` by its name only. However the same technique might be used to customize other properties as well as to create even more sophisticated server group topologies. For example, we could implement region bursting by composing several scalable server groups that allocate servers in different regions. And all of that without making any changes to the application code itself!

Before you proceed, please ensure that you have an OpenStack cloud (devstack-based will work just fine) and the latest version of Murano deployed. This guide assumes that the reader has a basic knowledge of some programming languages and object-oriented design and is a bit familiar with the scripting languages used to configure Linux servers. Also it would be beneficial to be familiar with YAML format: lots of software configuration tools nowadays use YAML, and Murano is no different.

Execution plan template

An execution plan template is a set of metadata that describes the installation process of an application on a virtual machine. It is a minimal executable unit that can be triggered in Murano workflows and is understandable to the Murano agent, which is responsible for receiving, correctness verification and execution of the statements included in the template.

The execution plan template is able to trigger any type of script that executes commands and installs application components as the result. Each script included in the execution plan template may consist of a single file or a set of interrelated files. A single script can be reused across several execution plans.

This section is devoted to the structure and syntax of an execution plan template. For different configurations of templates, please refer to the *Examples* section.

Template sections

The table below contains the list of the sections that can be included in the execution plan template with the description of their meaning and the default attributes which are used by the agent if any of the listed parameters is not specified.

Section name	Meaning and default value
FormatVersion	a version of the execution plan template syntax format. Default is 1.0.0. Optional
Name	a human-readable name for the execution plan to be used for logging. Optional
Version	a version of the execution plan itself, is used for logging and tracing. Each time the content of the template content changes (main script, attached scripts, properties, etc.), the version value should be incremented. This is in contrast with <code>FormatVersion</code> , which is used to distinguish the execution plan format. The default value is 0.0.0. Optional
Body	string that represents the Python statement and is executed by the murano-agent. Scripts defined in the Scripts section are invoked from here. Required
Parameters	a dictionary of the <code>String->JsonObject</code> type that maps parameter names to their values. Optional .
Scripts	a dictionary that maps script names to their script definitions. Required

FormatVersion property

`FormatVersion` is a property that all other depend on. That is why it is very important to specify it correctly.

`FormatVersion` 1.0.0 (default) is still used by Windows murano-agent. Almost all the applications in murano-apps repository work with `FormatVersion` 2.0.0. New features that are introduced in Kilo, such as Chef or Puppet, and downloadable files require version 2.1.0 or greater. Since `FormatVersion` 2.2.0 it is possible to enable Berkshelf. It requires Mitaka version of agent. If you omit the `FormatVersion` property or put something like `<2.0.0`, it will lead to the incorrect behaviour. The same happens if, for example, `FormatVersion=2.1.0`, and a VM has the pre-Kilo agent.

Scripts section

Scripts are the building blocks of execution plan templates. As the name implies those are the scripts for different deployment platforms.

Each script may consists of one or more files. Those files are script's program modules, resource files, configs, certificates etc.

Scripts may be executed as a whole (like a single piece of code), expose some functions that can be independently called in an execution plan script or both. This depends on deployment platform and executor capabilities.

Scripts are specified using `Scripts` attribute of execution plan. This attribute maps script name to a structure (document) that describes the script. It has the following properties:

Type

the name of a deployment platform the script is targeted to. The available alternative options for `version` $\geq 2.1.0$ are `Application`, `Chef`, `Puppet`, and for `version` $< 2.1.0$ is `Application` only. String, required.

Version

the minimum version of the deployment platform/executor required by the script. String, optional.

EntryPoint

the name of the script file that is an entry point for this execution plan template. String, required.

Files

the filenames of the additional files required for the script. Thus, if the script specified in the `EntryPoint` section imports other scripts, they should be provided in this section.

The filenames may include slashes that the agent preserve on VM. If a filename is enclosed in the angle brackets (`<...>`) it will be base64-encoded. Otherwise, it will be treated as a plain-text that may affect line endings.

In Kilo, entries for this property may be not just strings but also dictionaries (for example, `filename: URL`) to specify downloadable files or git repositories.

The default value is `[]` that means that no extra files are used. Array, optional.

Options

an optional dictionary of type `String->JsonObject` that contains additional options for the script executor. If not provided, an empty dictionary is assumed.

Available alternatives are: `captureStdout`, `captureStderr`, `verifyExitcode` (raise an exception if result is not positive). As `Options` are executor-dependent, these three alternatives are available for the `Application` executor, but may have no sense for other types. `captureStdout`, `captureStderr` and `verifyExitcode` require boolean values, and have `True` as their default values.

Dictionary, optional.

Please make sure the files specified in `EntryPoint` and `Files` sections exist.

HOT packages

Compose a package

Murano is an Application catalog which intends to support applications defined in different formats. As a first step to universality, support of a heat orchestration template was added. It means that any heat template could be added as a separate application into the Application Catalog. This could be done in two ways: manual and automatic.

Automatic package composing

Before uploading an application into the catalog, it should be prepared and archived. A Murano command line will do all preparation for you. Just choose the desired Heat Orchestration Template and perform the following command:

```
murano package-create --template wordpress/template.yaml
```

Note, that optional parameters could be specified:

- name**
an application name, copied from a template by default
- logo**
an application square logo, by default the heat logo will be used
- description**
text information about an application, by default copied from a template
- author**
a name of an application author
- output**
a name of an output file archive to save locally
- full-name**
a fully qualified domain name that specifies exact application location
- resources-dir**
a path to the directory containing application resources

Note: To performing this command `python-muranoclient` should be installed in the system

As the result, an application definition archive will be ready for uploading.

Manual package composing

Application package could be composed manually. Follow the 5 steps below.

- *Step 1. Choose the desired heat orchestration template*
For this example `chef-server.yaml` template will be used.
- *Step 2. Rename it to `template.yaml`*
- *Step 3. Prepare an application logo (optional step)*
It could be any picture associated with the application.
- *Step 4. Create `manifest.yaml` file*

All service information about the application is contained here. Specify the following parameters:

Format

defines an application definition format; should be set to `Heat.HOT/1.0`

Type

defines a manifest type, should be set to `Application`

FullName

a unique name which will be used to identify the application in Murano Catalog

Description

text information about an application

Author

a name of an application author or a company

Tags

keywords associated with the application

Logo

a name of a logo file for an application

Take a look at the example:

```
Format: Heat.HOT/1.0
Type: Application
FullName: com.example.Chef-Server
Name: Chef Server
Description: "Heat template to deploy Open Source CHEF
↪server on a VM"
Author: Kate
Tags:
- hot-based
Logo: logo.png
```

- *Step 5. Create a zip archive, containing the specified files: `template.yaml`, `manifest.yaml`, `logo.png`*

Browse page looks like:

Project ▾

Admin ▾

Identity ▾

Applications ▲

Catalog ▲

Environments


Browse

Manage ▾

Applications / Catalog / Browse

Browse

Recent Activity



Chef Server


Heat template to deploy
Open Source CHEF
server on a VM

[Details »](#)

+ Create Env

▶ Quick Deploy

The configuration form, where you can enter template parameters, will be generated automatically and looks as follows:



Configure Application: Chef Server

✕

Ssh Key Name *

Chef Flavor Name *

Chef Port

Chef Server Name

Chef Image Name *

Rabbit Password

Chef Server

Ssh Key Name:
Name of a Key Pair to enable SSH access to the instance

Chef Flavor Name:
Name Flavor to use for server

Chef Port:
Port Number

Chef Server Name:
The Instance Name

Chef Image Name:
Name of image to use for server

Rabbit Password:
Password for RabbitMQ

Back

Create

After filling the form the application is ready to be deployed.

Hot packages with nested Heat templates

In Murano HOT packages it is possible to allow Heat nested templates to be saved and deployed as part of a Murano Heat applications. Such templates should be placed in package under `’/Resources/HotFiles’`. Adding additional templates to a package is optional. When a Heat generated package is being deployed, if there are any Heat nested templates located in the package under `’/Resources/HotFiles’`, they are sent to Heat together with the main template and params during stack creation.

These nested templates can be referenced by putting the template name into the `type` attribute of resource definition, in the main template. This mechanism then compose one logical stack with these multiple templates. The following examples illustrate how you can use a custom template to define new types of resources. These examples use a custom template stored in a `sub_template.yaml` file

```
heat_template_version: 2015-04-30

parameters:
  key_name:
    type: string
    description: Name of a KeyPair

resources:
  server:
    type: OS::Nova::Server
    properties:
      key_name: {get_param: key_name}
      flavor: m1.small
      image: ubuntu-trusty
```

Use the template filename as type

The following main template defines the `sub_template.yaml` file as value for the `type` property of a resource

```
heat_template_version: 2015-04-30

resources:
  my_server:
    type: sub_template.yaml
    properties:
      key_name: my_key
```

Note: This feature is supported Liberty onwards.

MuranoPL Reference

To develop applications, murano project refers to Murano Programming Language (MuranoPL). It is represented by easily readable YAML and YAQL languages. The sections below describe these languages.

YAML

YAML is an easily readable data serialization format that is a superset of JSON. Unlike JSON, YAML is designed to be read and written by humans and relies on visual indentation to denote nesting of data structures. This is similar to how Python uses indentation for block structures instead of curly brackets in most C-like languages. Also YAML may contain more data types as compared to JSON. See <http://yaml.org/> for a detailed description of YAML.

MuranoPL is designed to be representable in YAML so that MuranoPL code could remain readable and structured. Usually MuranoPL files are YAML encoded documents. But MuranoPL engine itself does not deal directly with YAML documents, and it is up to the hosting application to locate and deserialize the definitions of particular classes. This gives the hosting application the ability to control where those definitions can be found (a file system, a database, a remote repository, etc.) and possibly use some other serialization formats instead of YAML.

MuranoPL engine relies on a host deserialization code when detecting YAQL expressions in a source definition. It provides them as instances of the `YaqlExpression` class rather than plain strings. Usually, YAQL expressions can be distinguished by the presence of `$` (the dollar sign) and operators, but in YAML, a developer can always state the type by using YAML tags explicitly. For example:

```

1 Some text - a string
2 $.something() - a YAQL expression
3 "$.something()" - a string because quotes are used
4 !!str $ - a string because a YAML tag is used
5 !yaql "text" - a YAQL expression because a YAML tag is used

```

YAQL

YAQL (Yet Another Query Language) is a query language that was also designed as a part of the murano project. MuranoPL makes an extensive use of YAQL. A description of YAQL can be found [here](#).

Simply speaking, YAQL is the language for expression evaluation. The following examples are all valid YAQL expressions: `2 + 2`, `foo() > bar()`, `true != false`.

The interesting thing in YAQL is that it has no built in list of functions. Everything YAQL can access is customizable. YAQL cannot call any function that was not explicitly registered to be accessible by YAQL. The same is true for operators. So the result of the expression `2 * foo(3, 4)` completely depends on explicitly provided implementations of "foo" and "operator_*".

YAQL uses a dollar sign (`$`) to access external variables, which are also explicitly provided by the host application, and function arguments. `$variable` is a syntax to get a value of the variable "`$variable`", `$1`, `$2`, etc. are the names for function arguments. "`$`" is a name for current object: data on which an expression is evaluated, or a name of a single argument. Thus, "`$`" in the beginning of an expression and "`$`" in the middle of it can refer to different things.

By default, YAQL has a lot of functions that can be registered in a YAQL context. This is very similar to how SQL works but uses more Python-like syntax. For example: `$.where($.myObj)`.

`myScalar > 5, $.myObj.myArray.len() > 0, and $.myObj.myArray.any($ = 4)).select($.myObj.myArray[0])` can be executed on `$ = array of objects`, and result in another array that is a filtration and projection of a source data.

Note: There is no assignment operator in YAQL, and `=` means comparison, the same what `==` means in Python.

As YAQL has no access to underlying operating system resources and is fully controllable by the host, it is secure to execute YAQL expressions without establishing a trust to the executed code. Also, because functions are not predefined, different methods can be accessible in different context. So, YAQL expressions that are used to specify property contracts are not necessarily valid in workflow definitions.

Common class structure

Here is a common template for class declarations. Note, that it is in the YAML format.

```
1 Name: class name
2 Namespaces: namespaces specification
3 Extends: [list of parent classes]
4 Properties: properties declaration
5 Methods:
6     methodName:
7         Arguments:
8             - list
9             - of
10            - arguments
11        Body:
12            - list
13            - of
14            - instructions
```

Thus MuranoPL class is a YAML dictionary with predefined key names, all keys except for `Name` are optional and can be omitted (but must be valid if specified).

Class name

Class names are alphanumeric names of the classes. Traditionally, all class names begin with an uppercase letter symbol and are written in PascalCasing.

In MuranoPL all class names are unique. At the same time, MuranoPL supports namespaces. So, in different namespaces you can have classes with the same name. You can specify a namespace explicitly, like `ns:MyName`. If you omit the namespace specification, `MyName` is expanded using the default namespace `=:`. Therefore, `MyName` equals `=:MyName` if `=` is a valid namespace.

Namespaces

Namespaces declaration specifies prefixes that can be used in the class body to make long class names shorter.

Namespaces:

```
=: io.murano.services.windows  
srv: io.murano.services  
std: io.murano
```

In the example above, the `srv: Something` class name is automatically translated to `io.murano.services.Something`.

`=` means the current namespace, so that `MyClass` means `io.murano.services.windows.MyClass`.

If the class name contains the period (`.`) in its name, then it is assumed to be already fully namespace qualified and is not expanded. Thus `ns.MyClass` remains as is.

Note: To make class names globally unique, we recommend specifying a developer's domain name as a part of the namespace.

Extends

MuranoPL supports multiple inheritance. If present, the `Extends` section shows base classes that are extended. If the list consists of a single entry, then you can write it as a scalar string instead of an array. If you do not specify any parents or omit the key, then the class extends `io.murano.Object`. Thus, `io.murano.Object` is the root class for all class hierarchies.

Properties

Properties are class attributes that together with methods create public class interface. Usually, but not always, properties are the values, and reference other objects that have to be entered in an environment designer prior to a workflow invocation.

Properties have the following declaration format:

propertyName:

```
Contract: property contract  
Usage: property usage  
Default: property default
```

Contract

Contract is a YAQL expression that says what type of the value is expected for the property as well as additional constraints imposed on a property. Using contracts you can define what value can be assigned to a property or argument. In case of invalid input data it may be automatically transformed to confirm to the contract. For example, if bool value is expected and user passes any not null value it will be converted to True. If converting is impossible exception `ContractViolationException` will be raised.

The following contracts are available:

Operation	Definition
<code>\$.int()</code>	an integer value (may be null). String values consisting of digits are converted to integers
<code>\$.int().notNull()</code>	a mandatory integer
<code>\$.string()</code> <code>\$.string().notNull()</code>	a string. If the value is not a string, it is converted to a string
<code>\$.bool()</code> <code>\$.bool().notNull()</code>	bools are true and false. <code>0</code> is converted to false, other integers to true
<code>\$.class(ns:ClassName)</code> <code>\$.class(ns:ClassName).notNull()</code>	value must be a reference to an instance of specified class name
<code>\$.template(ns:ClassName)</code> <code>\$.template(ns:ClassName).notNull()</code>	value must be a dictionary with object-model representation of specified class name
<code>\$.class(ns:ClassName, ns:DefaultClassName)</code>	create instance of the <code>ns:DefaultClassName</code> class if no instance provided
<code>\$.class(ns:Name).check(\$.p = 12)</code>	the value must be of the <code>ns:Name</code> type and have the <code>p</code> property equal to 12
<code>\$.class(ns:Name).owned()</code>	a current object must be direct or indirect owner of the value
<code>\$.class(ns:Name).notOwned()</code>	the value must be owned by any object except current one
<code>[\$.int()]</code> <code>[\$.int().notNull()]</code>	an array of integers. Similar to other types.

6.1. Deploying Murano

229

`[$.int()].check($ > 0)`

an array of the positive integers (thus not null)

In the example below property `port` must be int value greater than 0 and less than 65536; `scope` must be a string value and one of 'public', 'cloud', 'host' or 'internal', and `protocol` must be a string value and either 'TCP' or 'UDP'. When user passes some values to these properties it will be checked that values confirm to the contracts.

```
Namespaces:
  =: io.murano.apps.docker
  std: io.murano

Name: ApplicationPort

Properties:
  port:
    Contract: $.int().notNull().check($ > 0 and $ < 65536)

  scope:
    Contract: $.string().notNull().check($ in list(public, cloud, host, ↵
↵internal))
    Default: private

  protocol:
    Contract: $.string().notNull().check($ in list(TCP, UDP))
    Default: TCP

Methods:
  getRepresentation:
    Body:
      Return:
        port: $.port
        scope: $.scope
        protocol: $.protocol
```

The `template` contract does the same validation as the `class` contract, but does not require the actual object to be passed as a property or argument. Instead it allows to create an object from the given template later. Also you can exclude some of the properties from validation and provide them later in the body of the method.

Consider the following example:

```
Namespaces:
  =: io.murano.applications
  res: io.murano.resources
  std: io.murano

Name: TemplateServerProvider

Properties:
  template:
    Contract: $.template(res:Instance, excludeProperties => [name]).notNull()
  serverNamePattern:
    Contract: $.string().notNull()
```

(continues on next page)

(continued from previous page)

```
threshold:
  Contract: $.int().check($ > 0)

Methods:
createReplica:
  Arguments:
  - index:
    Contract: $.int().notNull()
  - owner:
    Contract: $.class(std:Object)
  Body:
  - If: $index < $this.threshold
    Then:
    - $template: $this.template
    - $template.name: $this.serverNamePattern.format($index)
    - $template['?'].name: format('Server {0}', $index)
    - Return: new($template, $owner)
  - Else:
    - Return: null
```

In the example above the class has the `template` property that is validated by the `template` contract. It holds the template of the object of the `Instance` class or its inheritor. In the `createReplica` method `template` is used to dynamically create instances in runtime considering some conditions and customizing the `name` property of an instance, as it was excluded from validation.

You still can pass an actual object to the property or argument with the `template` contract, but it will be automatically converted to its object model representation.

Property usage

Usage states the purpose of the property. This implies who and how can access it. The following usages are available:

Value	Explanation
In	Input property. Values of such properties are obtained from a user and cannot be modified in MuranoPL workflows. This is the default value for the Usage key.
Out	A value is obtained from executing MuranoPL workflow and cannot be modified by a user.
InOut	A value can be modified both by user and by workflow.
Const	The same as In but once workflow is executed a property cannot be changed neither by a user nor by a workflow.
Runtime	A property is visible only from within workflows. It is neither read from input nor serialized to a workflow output.
Static	Property is defined on a class rather than on an instance. See <i>Static methods and properties</i> for details.
Config	A property allows to have per-class configuration. A value is obtained from the config file rather than from the object model. These config files are stored in a special folder that is configured in the [engine] section of the Murano config file under the class_configs key.

The usage attribute is optional and can be omitted (which implies In).

If the workflow tries to write to a property that is not declared with one of the types above, it is considered to be private and accessible only to that class (and not serialized to output and thus would be lost upon the next deployment). An attempt to read the property that was not initialized results in an exception.

Default

Default is a value that is used if the property value is not mentioned in the input object model, but not when it is set to null. Default, if specified, must conform to a declared property contract. If Default is not specified, then null is the default.

For properties that are references to other classes, Default can modify a default value of the referenced objects. For example:

```
p:
  Contract: $.class(MyClass)
  Default: {a: 12}
```

This overrides default for the a property of MyClass for instance of MyClass that is created for this property.

Workflow

Workflows are the methods that describe how the entities that are represented by MuranoPL classes are deployed.

In a typical scenario, the root object in an input data model is of the `io.murano.Environment` type, and has the `deploy` method. This method invocation causes a series of infrastructure activities (typically, a Heat stack modification) and the deployment scripts execution initiated by VM agents commands. The role of the workflow is to map data from the input object model, or a result of previously executed actions, to the parameters of these activities and to initiate these activities in a correct order.

Methods

Methods have input parameters, and can return a value to a caller. Methods are defined in the Workflow section of the class using the following template:

```
methodName:
  Scope: Public
  Arguments:
    - list
    - of
    - arguments
  Body:
    - list
    - of
    - instructions
```

Public is an optional parameter that specifies methods to be executed by direct triggering after deployment.

Method arguments

Arguments are optional too, and are declared using the same syntax as class properties. Same as properties, arguments also have contracts and optional defaults.

Unlike class properties Arguments may have a different set of Usages:

Value	Explanation
Standard	Regular method argument. Holds a single value based on its contract. This is the default value for the Usage key.
VarArgs	A variable length argument. Method body sees it as a list of values, each matching a contract of the argument.
KwArgs	A keyword-based argument, Method body sees it as a dict of values, with keys being valid keyword strings and values matching a contract of the argument.

Arguments example:

```
scaleRc:
  Arguments:
  - rcName:
    Contract: $.string().notNull()
  - newSize:
    Contract: $.int().notNull()
  - rest:
    Contract: $.int()
    Usage: VarArgs
  - others:
    Contract: $.int()
    Usage: KwArgs
```

Method body

The Method body is an array of instructions that get executed sequentially. There are 3 types of instructions that can be found in a workflow body:

- Expressions,
- Assignments,
- Block constructs.

Method usage

Usage states the purpose of the method. This implies who and how can access it. The following usages are available:

Value	Explanation
Runtime	Normal instance method.
Static	Static method that does not require class instance. See <i>Static methods and properties</i> for details.
Extension	Extension static method that extends some other type. See <i>Extension methods</i> for details.
Action	Method can be invoked from outside (using Murano API). This option is deprecated for the package format versions > 1.3 in favor of <code>Scope: Public</code> and occasionally will be no longer supported. See <i>Murano actions</i> for details.

The Usage attribute is optional and can be omitted (which implies `Runtime`).

Method scope

The Scope attribute declares method visibility. It can have two possible values:

- *Session* - regular method that is accessible from anywhere in the current execution session. This is the default if the attribute is omitted;
- *Public* - accessible anywhere, both within the session and from outside through the API call.

The Scope attribute is optional and can be omitted (which implies `Session`).

Expressions

Expressions are YAQL expressions that are executed for their side effect. All accessible object methods can be called in the expression using the `$obj.methodName(arguments)` syntax.

Expression	Explanation
<code>\$.methodName()</code> <code>\$this.methodName()</code>	invoke method 'methodName' on this (self) object
<code>\$.property.methodName()</code> <code>\$this.property.methodName()</code>	invocation of method on object that is in property
<code>\$.method(1, 2, 3)</code>	methods can have arguments
<code>\$.method(1, 2, thirdParameter => 3)</code>	named parameters also supported
<code>list(\$.foo().bar(\$this.property), \$p)</code>	complex expressions can be constructed

Assignment

Assignments are single key dictionaries with a YAQL expression as a key and arbitrary structure as a value. Such a construct is evaluated as an assignment.

Assignment	Explanation
<code>\$x: value</code>	assigns <code>value</code> to the local variable <code>\$x</code>
<code>\$.x: value</code> <code>\$this.x: value</code>	assign <code>value</code> to the object's property
<code>\$.x: \$.y</code>	copies the value of the property <code>y</code> to the property <code>x</code>
<code>\$x: [\$a, \$b]</code>	sets <code>\$x</code> to the array of two values: <code>\$a</code> and <code>\$b</code>
<code>\$x:</code> <code>SomeKey:</code> <code>NestedKey: \$variable</code>	structures of any level of complexity can be evaluated
<code>\$.x[0]: value</code>	assigns <code>value</code> to the first array entry of the <code>x</code> property
<code>\$.x: \$.x.append(value)</code>	appends <code>value</code> to the array in the <code>x</code> property
<code>\$.x: \$.x.insert(1, value)</code>	inserts <code>value</code> into position 1 of the array in the <code>x</code> property
<code>\$x: list(\$a, \$b).delete(0)</code>	sets <code>\$x</code> to the list without the item at index 0
<code>\$.x.key.subKey: value</code> <code>\$x[key][subKey]: value</code>	deep dictionary modification

Block constructs

Block constructs control a program flow. They are dictionaries that have strings as all their keys.

The following block constructs are available:

Assignment	Explanation
Return: value	Returns value from a method
If: predicate() Then: - code - block Else: - code - block	<p>predicate() is a YAQL expression that must be evaluated to True or False</p> <p>The Else section is optional</p> <p>One-line code blocks can be written as scalars rather than an array.</p>
While: predicate() Do: - code - block	<p>predicate() must be evaluated to True or False</p>
For: variableName In: collection Do: - code - block	<p>collection must be a YAQL expression returning iterable collection or evaluatable array as in assignment instructions, for example, [1, 2, \$x]</p> <p>Inside a code block loop, a variable is accessible as \$variableName</p>
Repeat: Do: - code - block	<p>Repeats the code block specified number of times</p>
Break:	Breaks from loop
Match: case1: - code - block case2: - code - block Default: - code	<p>Matches the result of \$valExpression() against a set of possible values (cases). The code block of first matched case is executed.</p> <p>If no case matched and the default key is present than the Default code block get executed.</p> <p>The case values are constant values (not expressions).</p>

Notice, that if you have more than one block construct in your workflow, you need to insert dashes before each construct. For example:

Body:

- If: predicate1()
 - Then:
 - code
 - block
- While: predicate2()
 - Do:
 - code
 - block

Object model

Object model is a JSON serialized representation of objects and their properties. Everything you do in the OpenStack dashboard is reflected in an object model. The object model is sent to the Application catalog engine when the user decides to deploy the built environment. On the engine side, MuranoPL objects are constructed and initialized from the received Object model, and a predefined method is executed on the root object.

Objects are serialized to JSON using the following template:

```
1 {
2   "?": {
3     "id": "globally unique object ID (UUID)",
4     "type": "fully namespace-qualified class name",
5
6     "optional designer-related entries can be placed here": {
7       "key": "value"
8     }
9   },
10
11   "classProperty1": "propertyValue",
12   "classProperty2": 123,
13   "classProperty3": ["value1", "value2"],
14
15   "reference1": {
16     "?": {
17       "id": "object id",
18       "type": "object type"
19     },
20
21     "property": "value"
22   },
23
24   "reference2": "referenced object id"
25 }
```

Objects can be identified as dictionaries that contain the ? entry. All system fields are hidden in that entry.

There are two ways to specify references:

1. `reference1` as in the example above. This method allows inline definition of an object. When the instance of the referenced object is created, an outer object becomes its parent/owner that is responsible for the object. The object itself may require that its parent (direct or indirect) be of a specified type, like all applications require to have `Environment` somewhere in a parent chain.
2. Referring to an object by specifying other object ID. That object must be defined elsewhere in an object tree. Object references distinguished from strings having the same value by evaluating property contracts. The former case would have `$.class(Name)` while the later - the `$.string()` contract.

MuranoPL Core Library

Some objects and actions can be used in several application deployments. All common parts are grouped into MuranoPL libraries. Murano core library is a set of classes needed in each deployment. Class names from core library can be used in the application definitions. This library is located under the `meta` directory.

Classes included in the Murano core library are as follows:

io.murano

- *Class: Object*
- *Class: Application*
- *Class: SecurityGroupManager*
- *Class: Environment*
- *Class: CloudRegion*

io.murano.resources

- *Class: Instance*
- *Class: Network*

io.murano.system

- *Class: Logger*
- *Class: StatusReporter*

Class: Object

A parent class for all MuranoPL classes. It implements the `initialize`, `setAttr`, and `getAttr` methods defined in the pythonic part of the Object class. All MuranoPL classes are implicitly inherited from this class.

See also:

Source `Object.yaml` file.

Class: Application

Defines an application itself. All custom applications must be derived from this class.

See also:

Source [Application.yaml](#) file.

Class: SecurityGroupManager

Manages security groups during an application deployment.

See also:

Source [SecurityGroupManager.yaml](#) file.

Class: CloudRegion

Defines a CloudRegion and groups region-local properties

Table 1: **CloudRegion** class properties

Property	Description	Default usage
name	A region name.	In
agentListener	A property containing the <code>io.murano.system.AgentListener</code> object that can be used to interact with Murano Agent.	Runtime
stack	A property containing a HeatStack object that can be used to interact with Heat.	Runtime
defaultNetwork	A property containing user-defined Networks (<code>io.murano.resources.Network</code>) that can be used as default networks for the instances in this environment.	In
securityGroupM	A property containing the <code>SecurityGroupManager</code> object that can be used to construct a security group associated with this environment.	Runtime

See also:

Source [CloudRegion.yaml](#) file.

Class: Environment

Defines an environment in terms of the deployment process and groups all Applications and their related infrastructures. It also able to deploy them at once.

Environments is intent to group applications to manage them easily.

Table 2: Environment class properties

Property	Description	Default usage
<code>name</code>	An environment name.	In
<code>applications</code>	A list of applications belonging to an environment.	In
<code>agentListener</code>	A property containing the <code>io.murano.system.AgentListener</code> object that can be used to interact with Murano Agent.	Runtime
<code>stack</code>	A property containing a <code>HeatStack</code> object in default region that can be used to interact with Heat.	Runtime
<code>instanceNotifi</code>	A property containing the <code>io.murano.system.InstanceNotifier</code> object that can be used to keep track of the amount of deployed instances.	Runtime
<code>defaultNetwork</code>	A property containing templates for user-defined Networks in regions (<code>io.murano.resources.Network</code>).	In
<code>securityGroupM</code>	A property containing the <code>SecurityGroupManager</code> object from default region that can be used to construct a security group associated with this environment.	Runtime
<code>homeRegionName</code>	A property containing the name of home region from <i>murano</i> config	Runtime
<code>regions</code>	A property containing the map <code>regionName -> CloudRegion</code> instance.	InOut
<code>regionConfigs</code>	A property containing the map <code>regionName -> CloudRegion</code> config	Config

See also:

Source [Environment.yaml](#) file.

Class: Instance

Defines virtual machine parameters and manages an instance lifecycle: spawning, deploying, joining to the network, applying security group, and deleting.

Table 3: Instance class properties

Property	Description	Default usage
regionName	Inherited from CloudResource. Describe region for instance deployment	In
name	An instance name.	In
flavor	An instance flavor defining virtual machine hardware parameters.	In
image	An instance image defining operation system.	In
keyname	Optional. A key pair name used to connect easily to the instance.	In
agent	Configures interaction with the Murano agent using <code>io.murano.system.Agent</code> .	Runtime
ipAddresses	A list of all IP addresses assigned to an instance. Floating ip address is placed in the list tail if present.	Out
networks	Specifies the networks that an instance will be joined to. Custom networks that extend <i>Network class</i> can be specified. An instance will be connected to them and for the default environment network or flat network if corresponding values are set to True. Without additional configuration, instance will be joined to the default network that is set in the current environment.	In
volumes	Specifies the mapping of a mounting path to volume implementations that must be attached to the instance. Custom volumes that extend <i>Volume class</i> can be specified.	In
blockDevices	Specifies the list of block device mappings that an instance will use to boot from. Each mapping defines a volume that must be an instance of <i>Volume class</i> , device name, device type, and boot order. Either the <code>blockDevices</code> property or <code>image</code> property must be specified in order to boot an instance	In
assignFloating	Determines if floating IP is required. Default is False.	In
floatingIpAddr	IP addresses assigned to an instance after an application deployment.	Out
securityGroupN	Optional. A security group that an instance will be joined to.	In

See also:

Source [Instance.yaml](#) file.

Resources

Instance class uses the following resources:

Agent-v2.template

Python Murano Agent template.

Note: This agent is supposed to be unified. Currently, only Linux-based machines are supported. Windows support will be added later.

linux-init.sh

Python Murano Agent initialization script that sets up an agent with valid information containing

an updated agent template.

Agent-v1.template

Windows Murano Agent template.

windows-init.sh

Windows Murano Agent initialization script.

Class: Network

The basic abstract class for all MuranoPL classes representing networks.

See also:

Source [Network.yaml](#) file.

Class: Logger

Logging API is the part of core library since Liberty release. It was introduced to improve debuggability of MuranoPL programs.

You can get a logger instance by calling a `logger` function which is located in `io.murano.system` namespace. The `logger` function takes a logger name as the only parameter. It is a common recommendation to use full class name as a logger name within that class. This convention avoids names conflicts in logs and ensures a better logging subsystem configurability.

Logger class instantiation:

```
$log: logger('io.murano.apps.activeDirectory.ActiveDirectory')
```

Table 4: Log levels prioritized in order of severity

Level	Description
CRITICAL	Very severe error events that will presumably lead the application to abort.
ERROR	Error events that might not prevent the application from running.
WARNING	Events that are potentially harmful but will allow the application to continue running.
INFO	Informational messages highlighting the progress of the application at the coarse-grained level.
DEBUG	Detailed informational events that are useful when debugging an application.
TRACE	Even more detailed informational events comparing to the DEBUG level.

There are several methods that fully correspond to the log levels you can use for logging events. They are `debug`, `trace`, `info`, `warning`, `error`, and `critical`.

Logging example:

```
$log.info('print my info message {message}', message=>message)
```

Logging methods use the same format rules as the YAQL `format` function. Thus the line above is equal to the:

```
$log.info('print my info message {message}'.format(message=>message))
```

To print an exception stacktrace, use the **exception** method. This method uses the ERROR level:

```
Try:  
- Throw: exceptionName  
  Message: exception message  
Catch:  
With: exceptionName  
As: e  
Do:  
- $log.exception($e, 'something bad happen "{message}"', message=>message)
```

Note: You can configure the logging subsystem through the `logging.conf` file of the Murano Engine.

See also:

- Source [Logger.yaml](#) file.
- [OpenStack networking logging configuration](#).

Class: StatusReporter

Provides feedback feature. To follow the deployment process in the UI, all status changes should be included in the application configuration.

See also:

Source [StatusReporter.yaml](#) file.

Reflection capabilities in MuranoPL.

Reflection provides objects that describes MuranoPL classes and packages.

The first important function is `typeinfo`. Usage:

```
$typeInfo: typeinfo($someObject)
```

Now `$typeInfo` variable contains instance of type of `$someObject` (MuranoClass instance).

MuranoPL provide following abilities to reflection:

Types

Property	Description
name	name of MuranoPL class
version	version (<i>SemVer</i>) of MuranoPL class.
ancestors	list of class ancestors
properties	list of class properties. See <i>Properties</i>
package	package information. See <i>Packages</i>
methods	list of methods. See <i>Methods</i>
type	reference to type, which can be used as argument in engine functions

Example

```
- $TypeInfo: typeinfo($)  
...  
# log name, version and package name of this class  
- $log.info("This is \"{class_name}/{version} from {package}\",  
  class_name => $TypeInfo.name,  
  version => str($TypeInfo.version),  
  package => $TypeInfo.package.name))  
- $log.info("Ancestors:")  
- For: ancestor  
  In: $TypeInfo.ancestors  
  Do:  
    #log all ancestors names  
    - $log.info("{ancestor_name}", ancestor_name => $ancestor.name)  
# log full class version  
- $log.info("{version}", version => str($TypeInfo.version))  
# create object with same class  
- $newObject = new($TypeInfo.type)
```

Properties

Property introspection

Property	Description
name	name of property
hasDefault	boolean value. <i>True</i> , if property has default value, <i>False</i> otherwise
usage	<i>Usage</i> property's field. See <i>Property usage</i> for details
declaringType	type - owner of declared property

Property access

Methods	Description
\$property. setValue(\$target \$value)	set value of \$property for object \$target to \$value
\$property. getValue(\$target)	get value of \$property for object \$target

Example

```
- $TypeInfo: typeinfo($)  
...  
# select first property  
- $selectedProperty: $TypeInfo.properties.first()  
# log property name  
- $log.info("Hi, my name is {p_name}, p_name => $selectedProperty.name)  
# set new property value  
- $selectedProperty.setValue($, "new_value")  
# log new property value using reflection  
- $log.info("My new value is {value}", value => $selectedProperty.getValue($))  
# also, if property static, $target can be null  
- $log.info("Static property value is {value},  
  value => $staticProperty.getValue(null))
```

Packages

Property	Description
types	list of types, declared in package
name	package name
version	package version

Example

```
- $TypeInfo: typeinfo($)  
...  
- $packageRef: $TypeInfo.package  
- $log.info("This is package {p_name}/{p_version}",  
  p_name => $packageRef.name,  
  p_version => str($packageRef.version))  
- $log.info("Types in package:")  
- For: type_  
  In: $packageRef.types  
  Do:  
  - $log.info("{typename}", typename => type_.name)
```


Methods

Methods properties

Property	Description
name	method's name
declaringType	type - owner of declared method
arguments	list of method's arguments. See <i>Method arguments</i>

Method invoking

Methods	Description
<code>\$method.invoke(\$target, karg1, ..., kargN named arguments \$arg1, ... \$argN, karg1 => value1, ..., kargN => valueN)</code>	call <code>\$target</code> 's method <code>\$method</code> with <code>\$arg1, ..., \$argN</code> positional arguments and <code>kwarg1, ..., kwargN</code> named arguments

Example

```
- $TypeInfo: typeinfo($)  
...  
# select single method by name  
- $selectedMethod: $TypeInfo.methods.where($.name = sampleMethodName).single()  
# log method name  
- $log.info("Method name: {m_name}", m_name => $selectedMethod.name)  
# log method arguments names  
- For: argument  
  In: $selectedMethod.arguments  
  Do:  
    - $log.info("{name}", name => $argument.name)  
# call method with positional argument 'bar' and named `baz` == 'baz'  
- $selectedMethod.invoke($, 'bar', baz => baz)
```

Method arguments

Property	Description
name	argument's name
hasDefault	<i>True</i> if argument has default value, <i>False</i> otherwise
declaringMethod	method - owner of argument
usage	argument's usage type. See <i>Method arguments</i> for details

```
- $firstArgument: $selectedMethod.arguments.first()  
# store argument's name  
- $argName: $firstArgument.name
```

(continues on next page)

(continued from previous page)

```
# store owner's name
- $methodName: $firstArgument.declaringMethod.name
- $log.info("Hi, my name is {a_name} ! My owner is {m_name}",
  a_name => $argName,
  m_name => $methodName)
```

Static methods and properties

In MuranoPL, static denotes class methods and class properties (as opposed to instance methods and instance properties). These methods and properties can be accessed without an instance present.

Static methods are often used for helper methods that are not bound to any object (that is, do not maintain a state) or as a convenient way to write a class factory.

Type objects

Usually static methods and properties are accessed using *type object*. That is, an object that represents the class rather than class instance.

For any given class *foo.Bar* its type object may be retrieved using any of the following ways:

- Using `ns:Bar` notation considering that *ns* is declared in *Namespaces* section (and it is *foo* in this case),
- Using `:Bar` syntax if *Bar* is in the current namespace (that is, what `=:Bar` would mean if `=` was a valid namespace prefix),
- Using `type()` function with a fully qualified class name: `type('foo.Bar')`,
- By obtaining a type of class instance: `type($object)` (available for packages with format version starting from *1.3*),
- Through reflection: `typeinfo($object).type`.

No matter what method was used to get type object, the returned object will be the same because there can be only one type object per class.

All functions that accept type name, for example `new()` function, also accept type objects.

Accessing static methods and properties

Static methods can be invoked using one of the two ways:

- Using *type object*: `ns:Bar.foo(arg)`, `:Bar.foo(arg)`, and so on,
- On a class instance similar to normal methods: `$obj.foo(arg)`.

Access to properties is similar to that:

- Using *type object*: `ns:Bar.property`, `:Bar.property`, and so on,
- On a class instance: `$obj.property`.

Static properties are defined on a class rather than on an instance. Therefore, their values will be the same for all class instances (for particular version of the class).

Declaration of static methods and properties

Methods and properties are declared to be static by specifying `Usage: Static` on them.

For example:

```

Properties:
  property:
    Contract: $.string()
    Usage: Static

Methods:
  foo:
    Usage: Static
    Body:
    - Return: $.property

```

Static properties are never initialized from object model but can be modified from within MuranoPL code (i.e. they are not immutable). Static methods also can be executed as an action from outside using `Scope: Public`. Within static method *Body* `$this` (and `$` if not set to something else in expression) are set to type object rather than to instance, as it is for regular methods.

Static methods written in Python

For MuranoPL classes entirely or partially written in Python, all methods that have either `@staticmethod` or `@classmethod` decorators are automatically imported as static methods and work as they normally do in Python.

Extension methods

Extension methods are a special kind of static methods that can act as if they were regular instance methods of some other type.

Extension methods enable you to "add" methods to existing types without modifying the original type.

Defining extension methods

Extension methods are declared with the `Usage: Extension modifier`.

For example:

```

Name: SampleClass
Methods:
  mul:
    Usage: Extension
    Arguments:

```

(continues on next page)

(continued from previous page)

```

- self:
  Contract: $.int().NotNull()
- arg:
  Contract: $.int().NotNull()
Body:
Return: $self * $arg

```

Extension method are said to extend some other type and that type is deducted from the first method argument contract. Thus extension methods must have at least one argument.

Extension methods can also be written in Python just the same way as static methods. However one should be careful in method declaration and use precise YAQL specification of the type of first method argument otherwise the method will become an extension of any type.

To turn Python static method into extension method it must be decorated with `@yaql.language.specs.meta('Usage', 'Extension')` decorator.

Using extension methods

The example above defines a method that extends integer type. Therefore, with the method above it becomes possible to say `2.mul(3)`. However, the most often usage is to extend some existing MuranoPL class using `class()` contract.

If the first argument contract does not have `NotNull()`, then the method can be invoked on the `null` object as well (like `null.foo()`).

Extension methods are static methods and, therefore, can be invoked in a usual way on type object: `:SampleClass.mul(2, 3)`. However, unlike regular static methods extensions cannot be invoked on a class instance because this can result in ambiguity.

Using extension lookup order

When somewhere in the code the `$foo.bar()` expression is encountered, MuranoPL uses the following order to locate `bar()` implementation:

- If there is an instance or static method in `$foo`'s class, it will be used.
- Otherwise if the current class (where this expression was encountered) has an extension method called `bar` and `$foo` satisfies the contract of its first argument, then this method will be called.

Normally, if no method was found an exception will be raised. However, additional extension methods can be imported into the current context. This is done using the `Import` keyword on a class level. The `Import` section specifies either a list or a single type name (or type object) which extension methods will be available anywhere within the class code:

```

Name: MyClass
Import:
- ns:SomeOtherType
- :ClassFomCurrentContext
- 'io.murano.foo.Bar'

```

If no method was found with the algorithm above, the search continues on extension methods of all classes listed in the `Import` section in the order types are listed.

MuranoPL Metadata

MuranoPL metadata is a way to attach additional information to various MuranoPL entities such as classes, packages, properties, methods, and method arguments. That information can be used by both applications (to implement dynamic programming techniques) or by the external callers (API consumers like UI or even by the Murano Engine itself to impose some runtime behavior based on well known meta values). Thus, metadata is a flexible alternative to adding new keyword for every new feature.

Work with metadata includes the following cases:

- Defining your own metadata classes
- Attaching metadata to various parts of MuranoPL code
- Obtaining metadata and its usage

Define metadata classes

Define MuranoPL class with the description of arbitrary metadata. The class that can be used as metadata differs from the regular class:

- The `Usage` attribute of the former equals to `Meta`, while the `Usage` attribute of the latter equals to `Class`. The default value of the `Usage` attribute is `Class`.
- Metadata class has additional attributes (`Cardinality`, `Applies` and `Inherited`) to control how and where instances of that class can be attached.

Cardinality

The `Cardinality` attribute can be set to either `One` or `Many` and indicates the possibility to attach two or more instances of metadata to a single language entity. The default value is `One`.

Applies

The `Applies` attribute can be set to one of `Package`, `Type`, `Method`, `Property`, `Argument` or `All` and controls the possible language entities which instances of metadata class can be attached to. It is possible to specify several values using YAML list notation. The default value is `All`.

Inherited

The `Inherited` attribute can be set to `true` or `false` and specifies if there is metadata retained for child classes, overridden methods and properties. The default value is `false`.

Using of `Inherited`: `true` has the following consequences.

If some class inherits from two classes with the same metadata attached and this metadata has `Cardinality: One`, it will lead to emerging of two metadata objects with `Cardinality: One` within a single entity and will throw an exception. However, if the child class has this metadata attached explicitly, it will override the inherited metas and there is no conflict.

If the child class has the same meta as its parent (attached explicitly), then in case of `Cardinality: One` the meta of the child overrides the meta of the parent as it is mentioned above. And in case of `Cardinality: Many` meta of the parent is added to the list of the child's metas.

Example

The following example shows a simple meta-class implementation:

```
Name: MetaClassOne
Usage: Meta
Cardinality: One
Applies: All

Properties:
  description:
    Contract: $.string()
    Default: null

  count:
    Contract: $.int().check($ >= 0)
    Default: 0
```

`MetaClassOne` is defined as a metadata class by setting the `Usage` attribute to `Meta`. The `Cardinality` and `Applies` attributes determine that only one instance of `MetaClassOne` can be attached to object of any type. The `Inherited` attribute is omitted so there is no metadata retained for child classes, overridden methods and properties. In the example above, `Cardinality` and `Applies` can be omitted as well, as their values are set to default but in this case the author wants to be explicit.

The following example shows metadata class with different values of attributes:

```
Name: MetaClassMany
Usage: Meta
Cardinality: Many
Applies: [Property, Method]
Inherited: true

Properties:
  description:
    Contract: $.string()
    Default: null
```

(continues on next page)

(continued from previous page)

```

count:
  Contract: $.int().check($ >= 0)
  Default: 0

```

An instance (or several instances) of `MetaClassMany` can be attached to either property or method. Overridden methods and properties inherit metadata from its parents.

Attach metadata to a MuranoPL entity

To attach metadata to MuranoPL class, package, property, method or method argument, add the `Meta` keyword to its description. Under the description, specify a list of metadata class instances which you want to attach to the entity. To attach only one metadata class instance, use a single scalar instead of a list.

Consider the example of attaching previously defined metadata to different entities in a class definition:

```

Namespaces:
  =: io.murano.bar
  std: io.murano
  res: io.murano.resources
  sys: io.murano.system

Name: Bar

Extends: std:Application

Meta:
  MetaClassOne:
    description: "Just an empty application class with some metadata"
    count: 1

Properties:
  name:
    Contract: $.string().notNull()
    Meta:
      - MetaClassOne:
        description: "Name of the app"
        count: 1
      - MetaClassMany:
        count: 2
      - MetaClassMany:
        count: 3

Methods:
  initialize:
    Body:
      - $_environment: $.find(std:Environment).require()

```

(continues on next page)

(continued from previous page)

```

Meta:
  MetaClassOne:
    description: "Method for initializing app"
    count: 1

deploy:
  Body:
    - If: not $.getAttr(deployed, false)
      Then:
        - $_environment.reporter.report($this, 'Deploy started')
        - $_environment.reporter.report($this, 'Deploy finished')
        - $.setAttr(deployed, true)

```

The Bar class has an instance of metadata class `MetaClassOne` attached. For this, the `Meta` keyword is added to the Bar class description and the instance of the `MetaClassOne` class is specified under it. This instance's properties are `description` and `count`.

There are three meta-objects attached to the name property of the Bar class. One of it is a `MetaClassOne` object and the other two are `MetaClassMany` objects. There can be more than one instance of `MetaClassMany` attached to a single entity since the `Cardinality` attribute of `MetaClassMany` is set to `Many`.

The `initialize` method of Bar also has its metadata.

To attach metadata to the package, add the `Meta` keyword to `manifest.yaml` file.

Example:

```

Format: 1.0
Type: Application
FullName: io.murano.bar.Bar
Name: Bar
Description: |
  Empty Description
Author: author
Tags: [bar]
Classes:
  io.murano.bar.Bar: Bar.yaml
  io.murano.bar.MetaClassOne: MetaClassOne.yaml
  io.murano.bar.MetaClassMany: MetaClassMany.yaml
Supplier:
  Name: Name
  Description: Description
  Summary: Summary
Meta:
  io.murano.bar.MetaClassOne:
    description: "Just an empty application with some metadata"
    count: 1

```


Obtain metadata in runtime

Metadata can be accessed from MuranoPL using reflection capabilities and from Python code using existing YAQL mechanism.

The following example shows how applications can access attached metadata:

```

Namespaces:
  =: io.murano.bar
  std: io.murano
  res: io.murano.resources
  sys: io.murano.system

Name: Bar

Extends: std:Application

Meta:
  MetaClassOne:
    description: "Just an empty application class with some metadata"

Methods:
  sampleAction:
    Scope: Public
    Body:
      - $_environment.reporter.report($this, typeinfo($).meta.
        where($ is MetaClassOne).single().description)

```

The `sampleAction` method is added to the `Bar` class definition. This makes use of metadata attached to the `Bar` class.

The information about the `Bar` class is received by calling the `typeinfo` function. Then metadata is accessed through the `meta` property which returns the collection of all meta attached to the property. Then it is checked that the meta is a `MetaClassOne` object to ensure that it has `description`. While executing the action, the phrase "Just an empty application class with some metadata" is reported to a log. Some advanced usages of MuranoPL reflection capabilities can be found in the corresponding section of this reference.

By using metadata, an application can get information of any type attached to any object and use this information to change its own behavior. The most valuable use-cases of metadata can be:

- Providing information about capabilities of application and its parts
- Setting application requirements

Capabilities can include version of software, information for use in UI or CLI, permissions, and any other. Metadata can also be used in requirements as a part of the contract.

The following example demonstrates the possible use cases for the metadata:

```

Name: BlogApp

Meta:
  m:SomeFeatureSupport:

```

(continues on next page)

(continued from previous page)

```

    support: true

Properties:
  volumeName:
    Contract: $.string().NotNull()
    Meta:
      m:Deprecated:
        text: "volumeName property is deprecated"
  server:
    Contract: $.class(srv:CoolServer).NotNull().check(typeinfo($).meta.
      where($ is m:SomeFeatureSupport and $.support = true).any())

Methods:
  importantAction:
    Scope: Public
    Meta:
      m:CallerMustBeAdmin

```

Note, that the classes in the example do not exist as of Murano Mitaka, and therefore the example is not a real working code.

The `SomeFeatureSupport` metadata with `support: true` says that the `BlogApp` application supports some feature. The `Deprecated` metadata attached to the `volumeName` property informs that this property has a better alternative and it will not be used in the future versions anymore. The `CallerMustBeAdmin` metadata attached to the `importantAction` method sets permission to execute this method to the admin users only.

In the contract of the `server` property it is specified that the server application must be of the `srv:CoolServer` class and must have the attached meta-object of the `m:SomeFeatureSupport` class with the `support` property set to `true`.

Versioning

Versioning is an ability to assign a version number to some particular package (and, in turn, to a class) and then distinguish packages with different versions.

Package version

It is possible to specify a version for packages. You can import several versions of the same package simultaneously and even deploy them inside a single environment. To do this, you should use `Glare` as a storage for packages. But if you're going to keep only the latest version API is still good enough and both `FormatVersion` and `Version` rules will still be there. For more information about using `Glare`, refer to *Using Glare as a storage for packages*.

To specify the version of your package, add a new section to the manifest file:

```
Version: 0.1.0
```

It should be standard SemVer format version string consisting of 3 parts: `Major.Minor.Patch` and optional SemVer suffixes `[-dev-build.label[+metadata.label]]`. All MuranoPL classes have the

version of the package they are contained in. If no version is specified, the package version is *0.0.0*.

Note: It is impossible to show multiple versions of the same application in murano dashboard: only the last one is shown if the multiple versions are present.

Package requirements

In some cases, packages may require other packages for their work. You need to list such packages in the *Require* section of the manifest file:

```
Require:
  package1_FQN: version_spec_1
  ...
  packageN_FQN: version_spec_N
```

version_spec here denotes the allowed version range. It can be either in semantic_version specification pip-like format or as a partial version string. If you do not want to specify the package version, leave this value empty:

```
Require:
  package1_FQN: '>=0.0.3'
  package2_FQN:
```

In this case, version specification is equal to *0*.

Note: All packages depend on the *io.murano* package (Core Library). If you do not specify this requirement in the list (or the list is empty, or there is no *Require* key in the package manifest), then dependency *io.murano: 0* will be automatically added.

Object version

You can specify the version of the objects in UI definition when your application requires a specific version of some class. To do this, add a new key *classVersion* to section *?* describing the object:

```
?:
  type: io.test.apps.TestApp
  classVersion: version_spec
```

Side-by-side versioning of packages

In some cases it might happen that several different versions of the same class are simultaneously present in a single environment:

- There are different versions of the same MuranoPL class inside a single object model (environment).
- Several class versions encounter within class parents. For example, class A extends B and C and class C inherits B2, where B and B2 are two different versions of the same class.

The first case, when two different versions of the same class need to communicate with each other, is handled by the fact that in order to do that there is a `class()` contract for that value. `class()` contract validates object version against package requirements. If class A has a property with contract `$.class(B)`, then an object passed in this property when upcasted to B must have a version compatible with requirement specification in A's package (requesting B's package).

For the second case, where a single class attempts to inherit from two different versions of the same class engine (DSL), it attempts to find a version of this class which satisfies all parties and use it instead. However, if it is impossible, all remained different versions of the same class are treated as if they are unrelated classes.

For example: classA inherits classB from packageX and classC from packageY. Both classB and classC inherit from classD from packageZ; however, packageX depends on the version 1.2.0 of packageZ, while packageY depends on the version 1.3.0. This leads to a situation when classA transitively inherits classD of both versions 1.2 and 1.3. Therefore, an exception is thrown. However, if packageY's dependency would be just "1" (which means any of the 1.x.x family), the conflict would be resolved and the 1.2 would be used as it satisfies both inheritance chains.

Murano engine is free to use any package version that is valid for the spec. For example, one application requires packageX with version spec `< 0.3` and another package with the spec `> 0`. If both packages are get used in the same environment and the engine already loaded version 0.3 it can still use it for the second requirement even if there is a package with version 0.4 in the catalog and the classes from both classes are never interfere. In other words, engine always tries to minimize the number of versions in use for the single package to avoid conflicts and unnecessary package downloads. However, it also means that packages not always get the latest requirements.

Manifest format versioning

The manifests of packages are versioned using *Format* attribute. Currently, available versions are: *1.0*, *1.1*, *1.2* and *1.3*. The versioning of manifest format is directly connected with YAQL and version of murano itself.

The short description of versions:

Format version	Description
1.0	supported by all versions of murano. Use this version if you are planning to use <i>yaql 0.2</i> in your application
1.1	supported since Liberty. <i>yaql 0.2</i> is supported in legacy mode. Specify it, if you want to use features from <i>yaql 0.2</i> and <i>yaql 1.0.0</i> at the same time in your application.
1.2	supported since Liberty. Do not use <i>yaql 0.2</i> in applications with this format.
1.3	supported since Mitaka. <i>yaql 1.1</i> is available. It's recommended specifying this format in new applications, where compatibility with older versions of murano is not required.
1.4	supported since Newton. Keyword <code>Scope</code> is introduced for class methods to declare method's accessibility from outside through the API call.

UI forms versioning

UI forms are versioned using `Format` attribute inside YAML definition. For more information, refer to *corresponding documentation*.

Execution plan format versioning

Format of an execution plan can be specified using property `FormatVersion`. More information can be found *here*.

Murano actions

Murano action is a type of MuranoPL method. The differences from a regular MuranoPL method are:

- Action is executed on deployed objects.
- Action execution is initiated by API request, you do not have to call the method manually.

So murano action allows performing any operations on objects:

- Getting information from the VM, like a config that is generated during the deployment
- VM rebooting
- Scaling

A list of available actions is formed during the environment deployment. Right after the deployment is finished, you can call action asynchronously. Murano engine generates a task for every action. Therefore, the action status can be tracked.

Note: Actions may be called against any MuranoPL object, including `Environment`, `Application`, and any other objects.

Note: Now murano doesn't support big files download during action execution. This is because action results are stored in murano database and are limited by approximately 10kb size.

To mark a method as an action, use `Scope: Public` or `Usage: Action`. The latter option is deprecated for the package format versions > 1.3 and occasionally will be no longer supported. Also, you cannot use both `Usage: Action` and `Scope: Session` in one method.

The following example shows an action that returns an archive with a configuration file:

```
exportConfig:
  Scope: Public
  Body:
    - $_environment.reporter.report($this, 'Action exportConfig called')
    - $resources: new(sys:Resources)
    - $template: $resources.yaml('ExportConfig.template')
    - $result: $.masterNode.instance.agent.call($template, $resources)
    - $_environment.reporter.report($this, 'Got archive from Kubernetes')
    - Return: new(std:File, base64Content => $result.content,
                  filename => 'application.tar.gz')
```

List of available actions can be found with environment details or application details API calls. It's located in object model special data. Take a look at the following example:

Request: `http://localhost:8082/v1/environments/<id>/services/<id>`

Response:

```
{
  "name": "SimpleVM",
  "?": {
    "_26411a1861294160833743e45d0eaad9": {
      "name": "SimpleApp"
    },
    "type": "com.example.Simple",
    "id": "e34c317a-f5ee-4f3d-ad2f-d07421b13d67",
    "_actions": {
      "e34c317a-f5ee-4f3d-ad2f-d07421b13d67_exportConfig": {
        "enabled": true,
        "name": "exportConfig"
      }
    }
  }
}
```

Static actions

Static methods (*Static methods and properties*) can also be called through the API if they are exposed by specifying `Scope: Public`, and the result of its execution will be returned.

Consider the following example of the static action that makes use both of static class property and user's input as an argument:

```
Name: Bar
```

```
Properties:
```

(continues on next page)

(continued from previous page)

```

greeting:
  Usage: Static
  Contract: $.string()
  Default: 'Hello, '

Methods:
  staticAction:
    Scope: Public
    Usage: Static
    Arguments:
      - myName:
        Contract: $.string().NotNull()
    Body:
      - Return: concat($.greeting, $myName)

```

Request: `http://localhost:8082/v1/actions`

Request body:

```

{
  "className": "ns.Bar",
  "methodName": "staticAction",
  "parameters": {"myName": "John"}
}

```

Response:

```
"Hello, John"
```

Murano packages

Package structure

The structure of the Murano application package is predefined. An application could be successfully uploaded to an application catalog.

The application package root folder should contain the following:

manifest.yaml file

is an application entry point.

Note: the filename is fixed, do not use any custom names.

Classes folder

contains MuranoPL class definitions.

Resources folder

contains execution plan templates and the **scripts** folder with all the files required for an application deployment located in it.

UI folder

contains the dynamic UI YAML definitions.

logo.png file (optional)

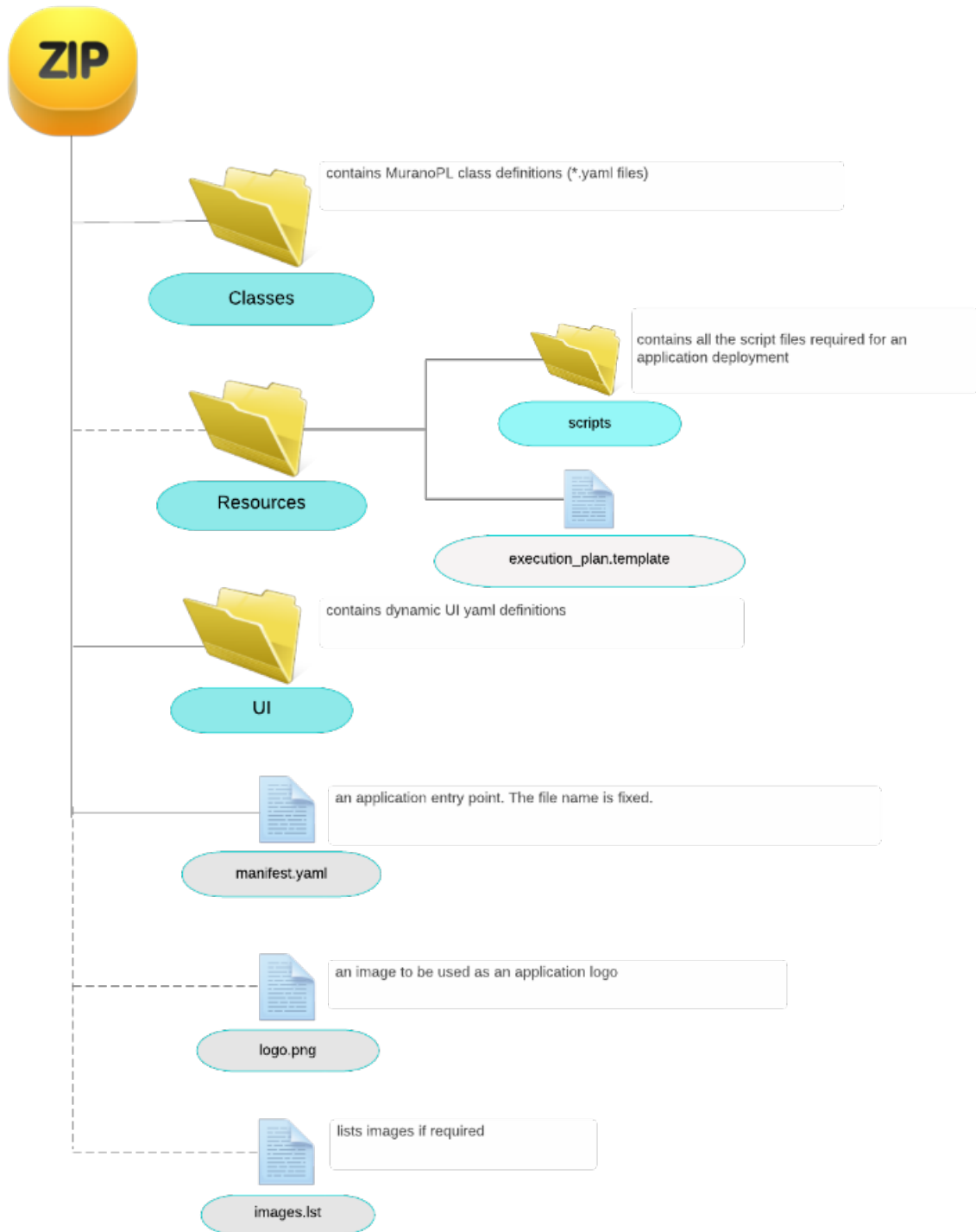
is an image file associated to your application.

Note: There are no any special limitations regarding an image filename. Though, if it differs from the default `logo.png`, specify it in an application manifest file.

images.lst file (optional)

contains a list of images required by an application.

Here is the visual representation of the Murano application package structure:



Dynamic UI definition specification

The main purpose of Dynamic UI is to generate application creation forms "on-the-fly". The Murano dashboard does not know anything about applications that will be presented in the catalog and which web forms are required to create an application instance. So all application definitions should contain an instruction, which tells the dashboard how to create an application and what validations need to be applied. This document will help you to compose a valid UI definition for your application.

The UI definition should be a valid YAML file and may contain the following sections (for version 2.x):

- **Version**
Points out the syntax version in use. *Optional*
- **Templates**
An auxiliary section, used together with an Application section to help with object model composing. *Optional*
- **Parameters**
An auxiliary section for evaluated once parameters. *Optional*
- **ParametersSource**
A static action name (ClassName.methodName) to call for additional parameters. *Optional*
- **Application**
Object model description passed to murano engine and used for application deployment.
Required
- **Forms**
Web form definitions. *Required*

Version

The syntax and format of dynamic UI definitions may change over time, so the concept of *format versions* is introduced. Each UI definition file may contain a top-level section called *Version* to indicate the minimum version of Murano Dynamic UI platform which is capable to process it. If the section is missing, the format version is assumed to be latest supported.

The version consists of two non-negative integer segments, separated by a dot, i.e. has a form of *MAJOR.MINOR*. Dynamic UI platforms having the same MAJOR version component are compatible: i.e. the platform having the higher version may process UI definitions with lower versions if their MAJOR segments are the same. For example, Murano Dynamic UI platform of version 2.2 is able to process UI definitions of versions 2.0, 2.1 and 2.2, but is unable to process 3.0 or 1.9.

Currently, the latest version of Dynamic UI platform is 2.3. It is incompatible with UI definitions of Version 1.0, which were used in Murano releases before Juno.

Note: Although the **Version** field is considered to be optional, its default value is the latest supported version. So if you intent to use applications with the previous stable murano version, verify that the version is set correctly.

Version history

Version	Changes	OpenStack Version
1.0	<ul style="list-style-type: none">Initial Dynamic UI implementation	Icehouse
2.0	<ul style="list-style-type: none"><i>instance</i> field support is droppedNew <i>Application</i> section that describes engine object modelNew <i>Templates</i> section for keeping reusable pieces of Object	Juno, Kilo
2.1	<ul style="list-style-type: none">New <i>network</i> field provides a selection of networks and their sub-networks as a dropdown populated with those which are available to the current tenant.	Liberty
2.2	<ul style="list-style-type: none">Now <i>application name</i> is added automatically to the last service form. It is needed for a user to recognize one created application from another in the UI. Previously all application definitions contained the <i>name</i> property. So to support backward compatibility, you need to manually remove <i>name</i> field from class properties.	Liberty
2.3	<ul style="list-style-type: none">Now <i>password</i> field supports <code>confirmInput</code> flag and validator overloading with single <code>regexpValidator</code> or multiple <i>validators</i> attribute.	Mitaka
2.4	<ul style="list-style-type: none">Parameters and ParametersSource sections were added	Ocata
268	<ul style="list-style-type: none"><code>ref()</code> YAQL function were added to Application DSL	Chapter 6. Administrator Documentation

Application

The Application section describes an *application object model*. The model is a dictionary (document) of application property values (inputs). Property value might be of any JSON-serializable type (including lists and maps). In addition the value can be of an object type (another application, application component, list of components etc.). Object properties are represented either by the object model of the component (i.e. dictionary) or by an object ID (string) if the object was already defined elsewhere. Each object definition (including the one in Application itself) must have a special ? key called object header. This key holds object metadata most important of which is the object type name. Thus the Application might look like this:

```
Application:
?:
  type: "com.myCompany.myNamespace.MyClass"
property1: "string property value"
property2: 123
property3:
  key1: value1
  key2: [1, false, null]
property4:
?:
  type: "com.myCompany.myNamespace.MyComponent"
property: value
```

However in most cases the values in object model should come from input fields rather than being static as in example above. To achieve this, object model values can also be of a *YAQL* <<https://opendev.org/openstack/yaql/src/branch/master/README.rst>> expression type. With expressions language it becomes possible to retrieve input control values, do some calculations and data transformations (queries). Any YAML value that is not enclosed in quote marks and conforms to the YAQL syntax is considered to be a YAQL expression. There is also an explicit YAML tag for the YAQL expressions: `!yaql`.

So with the YAQL addition Application section might look like this:

```
Application:
?:
  type: "com.myCompany.myNamespace.MyClass"
property1: $.formName.controlName
property2: 100 + 20 + 3
property3:
  !yaql "'KEY1'.toLowerCase()': !yaql "value1 + '1'"
  key2: [$parameter, not true]
property4: null
```

When evaluating YAQL expressions \$ is set to the forms data (list of dictionaries with cleaned validated forms' data) and templates and parameters are available using \$templateName (\$parameterName) syntax. See below on templates and parameters.

YAQL comes with hundreds of functions bundled. In addition to that there are another four functions provided by murano dashboard:

- **generateHostname(pattern, index)** is used for a machine hostname template generation. It accepts two arguments: name pattern (string) and index (integer). If '#' symbol is present in name

pattern, it will be replaced with the index provided. If pattern is an empty string, a random name will be generated.

- **repeat(template, times)** is used to produce a list of data snippets, given the template snippet (first argument) and number of times it should be reproduced (second argument). Inside that template snippet current step can be referenced as *\$index*.
- **name()** returns current application name.
- **ref(templateName [, parameterName] [, idOnly])** is used to generate object definition from the template and then reference it several times in the object model. This function evaluates `templateName` and fixes the result in parameters under `parameterName` key (or `templateName` if the second parameter was omitted). Then it generates object ID and places it into `?/id` field. On the first use of `parameterName` or if `idOnly` is `false` the function will return the whole object structure. On subsequent calls or if `idOnly` is `true` it will return the ID that was generated upon the first call.

Templates

It is often that application object model contains number of similar instances of the same component/class. For example it might be list of servers for multi-server application or list of nodes or list of components. For such cases UI definition markup allow to give the repeated object model snippet a name and then refer to it by the name in the application object model. Such snippets are placed into Templates section:

```

Templates:
primaryController:
  ?:
    type: "io.murano.windows.activeDirectory.PrimaryController"
  host:
    ?:
      type: "io.murano.windows.Host"
    adminPassword: $.appConfiguration.adminPassword
    name: generateHostname($.appConfiguration.unitNamingPattern, 1)
    flavor: $.instanceConfiguration.flavor
    image: $.instanceConfiguration.osImage

secondaryController:
  ?:
    type: "io.murano.windows.activeDirectory.SecondaryController"
  host:
    ?:
      type: "io.murano.windows.Host"
    adminPassword: $.appConfiguration.adminPassword
    name: generateHostname($.appConfiguration.unitNamingPattern, $index +
↪1)
    flavor: $.instanceConfiguration.flavor
    image: $.instanceConfiguration.osImage

```

Then the template can be inserted into application object model or to another template using `$templateName` syntax. It is often case that it is used together with `repeat` function to put several

instances of template. In this case templates may use of `$index` variable which will hold current iteration number:

```

Application:
  ?:
    type: io.murano.windows.activeDirectory.ActiveDirectory
    primaryController: $primaryController
    secondaryControllers: repeat($secondaryController, $.appConfiguration.
↪dcInstances - 1)

```

It is important to remember that templates are evaluated upon each access or `repeat()` iteration. Thus if the template has some properties set to a random or generated values they are going to be different for each instance of the template.

Another use case for templates is when single object is referenced several times within application object model:

```

Templates:
  instance:
    ?:
      type: "io.murano.resources.LinuxMuranoInstance"
      image: myImage
      flavor: "m1.small"

Application:
  ?:
    type: "com.example.MyApp"
    components:
      - ?:
        type: "com.example.MyComponentType1"
        instance: ref(instance)
      - ?:
        type: "com.example.MyComponentType2"
        instance: ref(instance)

```

In example above there are two components that uses the same server instance. If this example had `$instance` instead of `ref(instance)` that would be two unrelated servers based on the same template i.e. with the same image and flavor, but not the same VM.

Parameters and ParametersSource

Parameters are values that are used to parametrize the UI form and/or application object model. Parameters are put into `Parameters` section and accessed using `$parameterName` syntax:

```

Parameters:
  param1: "Hello!"

Application:
  ?:
    type: "com.example.MyApp"
    stringProperty: $param1

```

Parameters are very similar to Templates with two differences:

1. Parameter values are evaluated only once per application instance at the very beginning whereas templates are evaluated on each access.
2. Parameter values can be used to initialize UI control attributes (e.g. initial text box value, list of choices for a drop down etc.)

However the most powerful feature about parameters is that their values might be obtained from the application class. Here is how to do it:

1. In one of the classes in the MuranoPL package (usually the main application class define a static action method without arguments that returns a dictionary of variables:

```
Name: "com.example.MyApp"
Methods:
  myMethod:
    Usage: Static
    Scope: Public
    Body:
      # arbitrary MuranoPL code can be used here
    Return:
      var1: value1
      var2: 123
```

2. In UI definition file add

```
ParametersSource: "com.example.MyApp.myMethod"
```

The class name may be omitted. In this case the dashboard will try to use the type of Application object or package FQN for that purpose.

The values returned by the method are going to be merged into Parameters section like if they were defined statically.

Forms

This section describes markup elements for defining forms, which are currently rendered and validated with Django. Each form has a name, field definitions (mandatory), and validator definitions (optionally).

Note that each form is split into 2 parts:

- **input area** - left side, where all the controls are located
- **description area** - right side, where descriptions of the controls are located

Each field should contain:

- **name** - system field name, could be any
- **type** - system field type

Currently supported options for **type** attribute are:

- *string* - text field (no inherent validations) with one-line text input
- *boolean* - boolean field, rendered as a checkbox

- *text* - same as string, but with a multi-line input
- *integer* - integer field with an appropriate validation, one-line text input
- *choice* - drop-down list of variants. Each variant has a display string that is going to be displayed to the user and associated key that is going to be a control value
- *password* - text field with validation for strong password, rendered as two masked text inputs (second one is for password confirmation)
- *clusterip* - specific text field, used for entering cluster IP address (validation for valid IP address syntax)
- *databaselist* - specific field, a list of databases (comma-separated list of databases' names, where each name has the following syntax first symbol should be latin letter or underscore; subsequent symbols can be latin letter, numeric, underscore, at the sign, number sign or dollar sign), rendered as one-line text input
- *image* - specific field, used for filtering suitable images by image type provided in murano metadata in glance properties.
- *flavor* - specific field, used for selection instance flavor from a list
- *keypair* - specific field, used for selecting a keypair from a list
- *azone* - specific field, used for selecting instance availability zone from a list
- *network* - specific field, used to select a network and subnet from a list of the ones available to the current user
- *securitygroup* - specific field, used for selecting a custom security group to assign to the instance
- *volume* - specific field, used for selecting a volume or a volume snapshot from a list of available volumes (and volume snapshots)
- any other value is considered to be a fully qualified name for some Application package and is rendered as a pair of controls: one for selecting already existing Applications of that type in an Environment, second - for creating a new Application of that type and selecting it

Other arguments (and whether they are required or not) depends on a field's type and other attributes values. Most of them are standard Django field attributes. The most common attributes are the following:

- **label** - name, that will be displayed in the form; defaults to **name** being capitalized.
- **description** - description, that will be displayed in the description area. Use YAML line folding character >- to keep the correct formatting during data transferring.
- **descriptionTitle** - title of the description, defaults to **label**; displayed in the description area
- **hidden** whether field should be visible or not in the input area. Note that hidden field's description will still be visible in the descriptions area (if given). Hidden fields are used storing some data to be used by other, visible fields.
- **minLength**, **maxLength** (for string fields) and **minValue**, **maxValue** (for integer fields) are transparently translated into django validation properties.
- **choices** - a choices for the choice control type. The format is `[["key1", "display value1"], ["key2", "display value2"]]`. Starting from version 2.4 this can also be passed as a `{key1: "display value1", key2: "display value2"}`
- **regexpValidator** - regular expression to validate user input. Used with *string* or *password* field.

- **errorMessages** - dictionary with optional 'invalid' and 'required' keys that set up what message to show to the user in case of errors.
- **validators** is a list of dictionaries, each dictionary should at least have *expr* key, under that key either some YAQL expression is stored, either one-element dictionary with *regexValidator* key (and some regexp string as value). Another possible key of a validator dictionary is *message*, and although it is not required, it is highly desirable to specify it - otherwise, when validator fails (i.e. regexp doesn't match or YAQL expression evaluates to false) no message will be shown. Note that field-level validators use YAQL context different from all other attributes and section: here \$ root object is set to the value of field being validated (to make expressions shorter).

```
- name: someField
  type: string
  label: Domain Name
  validators:
    - expr:
      regexValidator: '([^\.]+\|^\.[^\.]{1,15}\.)*$'
      message: >-
        NetBIOS name cannot be shorter than 1 symbol and
        longer than 15 symbols.
    - expr:
      regexValidator: '([^\.]+\|^\.[^\.]*\.[^\.]{2,63}\.)*$'
      message: >-
        DNS host name cannot be shorter than 2 symbols and
        longer than 63 symbols.
  helpText: >-
    Just letters, numbers and dashes are allowed.
    A dot can be used to create subdomains
```

Using of *regexValidator* and *validators* attributes with *password* field was introduced in version 2.3. By default, password should have at least 7 characters, 1 capital letter, 1 non-capital letter, 1 digit, and 1 special character. If you do not want password validation to be so strong, you can override it by setting a custom validator or multiple validators for password. For that add *regexValidator* or *validators* to the *password* field and specify custom regexp string as value, just like with any *string* field.

Example

```
- name: password
  type: password
  label: Password
  descriptionTitle: Password
  description: >-
    Please, provide password for the application. Password should be
    5-50 characters long and consist of alphanumeric characters
  regexValidator: '^[a-zA-Z0-9]{5,50}?$'
```

- **confirmInput** is a flag used only with password field and defaults to `true`. If you decided to turn off automatic password field cloning, you should set it to `false`. In this case password confirmation is not required from a user.
- **widgetMedia** sets some custom *CSS* and *JavaScript* used for the field's widget rendering. Note, that files should be placed to Django static folder in advance. Mostly they are used to do some

client-side field enabling/disabling, hiding/unhiding etc.

- **requirements** is used only with flavor field and prevents user to pick unstable for a deployment flavor. It allows to set minimum ram (in MBs), disk space (in GBs) or virtual CPU quantity.

Example that shows how to hide items smaller than regular *small* flavor in a flavor select field:

```
- name: flavor
  type: flavor
  label: Instance flavor
  requirements:
    min_disk: 20
    min_vcpus: 2
    min_memory_mb: 2048
```

- **include_snapshots** is used only with the volume field. True by default. If True, the field list includes available volumes and volume snapshots. If set to False, only available volumes are shown.
- **include_subnets** is used only with network field. True by default. If True, the field list includes all the possible combinations of network and subnet. E.g. if there are two available networks X and Y, and X has two subnets A and B, while Y has a single subnet C, then the list will include 3 items: (X, A), (X, B), (Y, C). If set to False only network names will be listed, without their subnets.
- **filter** is used only with network field. None by default. If set to a regexp string, will be used to display only the networks with names matching the given regexp.
- **murano_networks** is used only with network field. None by default. May have values None, exclude or translate. Defines the handling of networks which are created by murano. Such networks usually have very long randomly generated names, and thus look ugly when displayed in the list. If this value is set to exclude then these networks are not shown in the list at all. If set to translate the names of such networks are replaced by a string Network of %env_name%.

Note: This functionality is based on the simple string matching of the network name prefix and the names of all the accessible murano environments. If the environment is renamed after the initial deployment this feature will not be able to properly translate or exclude its network name.

- **allow_auto** is used only with network field. True by default. Defines if the default value of the dropdown (labeled "Auto") should be present in the list. The default value is a tuple consisting of two None values. The logic on how to treat this value is up to application developer. It is suggested to use this field to indicate that the instance should join default environment network. For use-cases where such behavior is not desired, this parameter should be set to False.

Network field and its specific attributes (*include_subnets*, *filter*, *murano_networks*, *allow_auto*) are available since version 2.1. Before that, there was no way for the end user to select existing network in the UI. The only way to change the default networking behavior was the usage of *networking.yaml* file. It allows to override the networking setting at the environment level, for all the murano environments of all the tenants. Now you can simply add a *network* field to your form definition and provide the ability to select the desired network for the specific application.

Example

```

- instanceConfiguration:
  fields:
    - name: network
      type: network
      label: Network
      description: Select a network to join. 'Auto' corresponds to a
↳default environment's network.
      murano_networks: translate

```

Besides field-level validators, form-level validators also exist. They use **standard context** for YAQL evaluation and are required when there is a need to validate some form's constraint across several fields.

Example

```

Forms:
- appConfiguration:
  fields:
    - name: dcInstances
      type: integer
      hidden: true
      initial: 1
      required: false
      maxLength: 15
      helpText: Optional field for a machine hostname template
    - name: unitNamingPattern
      type: string
      label: Instance Naming Pattern
      required: false
      maxLength: 64
      regexValidator: '^[a-zA-Z][-_\w]*$'
      errorMessages:
        invalid: Just letters, numbers, underscores and hyphens are
↳allowed.
      helpText: Just letters, numbers, underscores and hyphens are allowed.
      description: >-
        Specify a string that will be used in a hostname instance.
        Just A-Z, a-z, 0-9, dash, and underline are allowed.

- instanceConfiguration:
  fields:
    - name: title
      type: string
      required: false
      hidden: true
      descriptionTitle: Instance Configuration
      description: Specify some instance parameters based on which
↳service will be created.
    - name: flavor
      type: flavor
      label: Instance flavor

```

(continues on next page)

(continued from previous page)

```

description: >-
    Select a flavor registered in OpenStack. Consider that service_
↪performance
    depends on this parameter.
required: false
- name: osImage
type: image
imageType: windows
label: Instance image
description: >-
    Select valid image for a service. Image should already be_
↪prepared and
    registered in glance.
- name: availabilityZone
type: azone
label: Availability zone
description: Select an availability zone, where service will be_
↪installed.
required: false
validators:
    # if unitNamingPattern is given and dcInstances > 1, then '#' should_
↪occur in unitNamingPattern
- expr: $.appConfiguration.dcInstances < 2 or not $.appConfiguration.
↪unitNamingPattern.bool()
    or '#' in $.appConfiguration.unitNamingPattern
message: Incrementation symbol "#" is required in the Instance_
↪Naming Pattern

```

Control attributes might be initialized with a YAQL expression. However prior to version 2.4 it only worked for forms other than the first. It was designed to initialize controls with values input on the previous step. Starting with version 2.4 this limitation was removed and it become possible to use arbitrary YAQL expressions for any of control fields on any forms and use parameter values as part of these expressions.

Murano package repository

Murano client and dashboard can install both packages and bundles of packages from murano repository. To do so you should set MURANO_REPO_URL settings in murano dashboard or MURANO_REPO_URL env variable for the CLI client, and use a respective command to import the package. These commands automatically import all the prerequisites required to install the application along with any images mentioned in the applications.

Setting up your own repository

It is fairly easy to set up your own murano package repository. To do so you need a web server that would serve 3 directories:

- /apps/
- /bundles/
- /images/

When importing an application by name, the client appends any version info, if present to the application name, .zip file extension and searches for that file in the apps directory.

When importing a bundle by name, the client appends .bundle file extension to the bundle name and searches it in the bundles directory. A bundle file is a JSON or a YAML file with the following structure:

```
{  
  "Packages": [  
    {  
      "Name": "com.example.ApacheHttpServer",  
      "Version": "", "Name": "com.example.Nginx",  
      "Version": "0.0.1", "Name": "com.example.Lighttpd"}  
    ]  
}
```

Glance images can be auto-imported by the client, when mentioned in `images.lst` inside the package. Please see *Developing Murano Packages 101* for more information about package composition. When importing images from the `image.lst` file, the client simply searches for a file with the same name as the name attribute of the image in the images directory of the repository.

Murano bundles

A bundle is a collection of packages. In the Community App Catalog, you can find such bundles as `container-based-apps`, `app-servers`, and so on. The packages in the Application Catalog are sorted by usage. You can import bundles from the catalog using Dashboard or CLI. You can read about this in *Managing applications* and *Using CLI*. Specific information about `bundle-import` command can be found at *Murano command-line client*.

Bundle structure

Bundle description is a JSON structure, that contains list of packages in the bundle and bundle version. Here is the example:

```
{  
  "Packages": [  
    {  
      "Name": "com.example.apache.ApacheHttpServer",  
      "Version": ""  
    },  
    {  
      "Name": "com.example.apache.Tomcat",  
      "Version": ""  
    }  
  ]  
}
```

(continues on next page)

(continued from previous page)

```

    }
  ],
  "Version": 1
}

```

`Name` is a required parameter and should contain package fully qualified name. `Version` is not a mandatory parameter. `Version` for package entry specifies the version of the package to look into *Murano package repository*. If it is specified, murano client would look for a file with that version specification in murano repository (for example `com.example.MyApp.0.0.1.zip` for `com.example.MyApp` of version 0.0.1). If the version is omitted or left blank client would search for `com.example.MyApp.zip`.

Create local bundle

However, you may need to create a local bundle. You may need it if you want to setup your own *Murano package repository*. To create a new bundle, perform the following steps:

1. Navigate to the directory with the target packages.
2. Create a `.bundle` file. List all the required packages in `Packages` section. If needed, specify the bundle version in the `Version` section.

Migrating applications between releases

This document describes how a developer of murano application can update existing packages to make them synchronized with all implemented features and requirements.

Migrate applications from Murano v0.5 to Stable/Juno

Applications created for murano v0.5, unfortunately, are not supported in Murano stable/juno. This document provides the application code changes required for compatibility with the stable/juno murano version.

Rename *'Workflow'* to *'Methods'*

In stable/juno the name of section containing class methods is renamed to *Methods*, as the latter is more OOP and doesn't cause confusion with Mistral. So, you need to change it in `app.name/Classes` in all classes describing workflow of your app.

For example:

```

Workflow:
  deploy:
    Body:
      - $_environment.reporter.report($this, 'Creating VM')

```

Should be changed to:

Methods:

deploy:

Body:

```
- $_environment.reporter.report($this, 'Creating VM')
```

Change the Instance type in the UI definition 'Application' section

The Instance class was too generic and contained some dirty workarounds to differently handle Windows and Linux images, to bootstrap an instance in a number of ways, etc. To solve these problems more classes were added to the *Instance* inheritance hierarchy.

Now, base *Instance* class is abstract and agnostic of the desired OS and agent type. It is inherited by two classes: *LinuxInstance* and *WindowsInstance*.

- *LinuxInstance* adds a default security rule for Linux, opening a standard SSH port;
- *WindowsInstance* adds a default security rule for Windows, opening an RDP port. At the same time *WindowsInstance* prepares a user-data allowing to use Murano v1 agent.

LinuxInstance is inherited by two other classes, having different software config method:

- *LinuxMuranoInstance* adds a user-data preparation to configure Murano v2 agent;
- *LinuxUDInstance* adds a custom user-data field allowing the services to supply their own user data.

You need to specify the instance type which is required by your app. It specifies a field in UI, where user can select an image matched to the instance type. This change must be added to UI form definition in *app.name/UI/ui.yaml*.

For example, if you are going to install your application on Ubuntu, you need to change:

```
Application:  
?:  
  instance:  
    ?:  
      type: io.murano.resources.Instance
```

to:

```
Application:  
?:  
  instance:  
    ?:  
      type: io.murano.resources.LinuxMuranoInstance
```


Migrate applications to Stable/Kilo

In Kilo, there are no breaking changes that affect backward compatibility. But there are two new features which you can use since Kilo.

1. Pluggable Pythonic classes for murano

Now you can create plug-ins for MuranoPL. A plug-in (extension) is an independent Python package implementing functionality which you want to add to the workflow of your application.

For a demo application demonstrating the usage of plug-ins, see the `murano/contrib/plugins/murano_exampleplugin` folder.

The application consist of the following components:

- An `ImageValidatorMixin` class that inherits the generic instance class (`io.murano.resources.Instance`) and adds a method capable of validating the instance image for having an appropriate murano metadata type. This class may be used as a mixin when added to inheritance hierarchy of concrete instance classes.
- A concrete class called `DemoInstance` that inherits from `io.murano.resources.LinuxMuranoInstance` and `ImageValidatorMixin` to add the image validation logic to a standard, murano-enabled and Linux-based instance.
- An application that deploys a single VM using the `DemoInstance` class if the tag on the user-supplied image matches the user-supplied constant.

The `ImageValidatorMixin` demonstrates the instantiation of plug-in provided class and its usage, as well as handling of exception which may be thrown if the plug-in is not installed in the environment.

2. Murano mistral integration

The core library has a new system class for mistral client that allows to call Mistral APIs from the murano application model.

The system class allows you to:

- Upload a mistral workflow to mistral.
- Trigger the mistral workflow that is already deployed, wait for completion and return the execution output.

To use this feature, add some mistral workflow to `Resources` folder of your package. For example, create file `TestEcho_MistralWorkflow.yaml`:

```
version: '2.0'

test_echo:
  type: direct
  input:
    - input_1
  output:
    out_1: <% $.task1_output_1 %>
    out_2: <% $.task2_output_2 %>
```

(continues on next page)

(continued from previous page)

```

    out_3: <% $.input_1 %>
tasks:
  my_echo_test:
    action: std.echo output='just a string'
    publish:
      task1_output_1: 'task1_output_1_value'
      task1_output_2: 'task1_output_2_value'
    on-success:
      - my_echo_test_2

  my_echo_test_2:
    action: std.echo output='just a string'
    publish:
      task2_output_1: 'task2_output_1_value'
      task2_output_2: 'task2_output_2_value'

```

And provide workflow to use the mistral client:

```

Namespaces:
=: io.murano.apps.test
std: io.murano
sys: io.murano.system

Name: MistralShowcaseApp

Extends: std:Application

Properties:
  name:
    Contract: $.string().notNull()

  mistralClient:
    Contract: $.class(sys:MistralClient)
    Usage: Runtime

Methods:
  initialize:
    Body:
      - $this.mistralClient: new(sys:MistralClient)

  deploy:
    Body:
      - $resources: new('io.murano.system.Resources')
      - $workflow: $resources.string('TestEcho_MistralWorkflow.yaml')
      - $.mistralClient.upload(definition => $workflow)
      - $output: $.mistralClient.run(name => 'test_echo', inputs =>
dict(input_1 => input_1_value))
      - $this.find(std:Environment).reporter.report($this, $output.

```

(continues on next page)

(continued from previous page)

```
↪get('out_3'))
```

Migrate applications to Stable/Liberty

In Liberty a number of useful features that can be used by developers creating their murano applications were implemented. This document describes these features and steps required to include them to new apps.

1. Versioning

Package version

Now murano packages have a new optional attribute in their manifest called *Version* - a standard SemVer format version string. All MuranoPL classes have the version of the package they contained in. To specify the version of your package, add a new section to the manifest file:

```
Version: 0.1.0
```

If no version specified, the package version will be equal to *0.0.0*.

Package requirements

There are cases when packages may require other packages for their work. Now you need to list such packages in the *Require* section of the manifest file:

```
Require:
  package1_FQN: version_spec_1
  ...
  packageN_FQN: version_spec_N
```

version_spec here denotes the allowed version range. It can be either in semantic_version specification pip-like format or as partial version string. If you do not want to specify the package version, leave this value empty:

```
Require:
  package1_FQN: '>=0.0.3'
  package2_FQN:
```

In this case, the last dependency *0.x.y* is used.

Note: All packages depend on the *io.murano* package (core library). If you do not specify this requirement in the list (or the list is empty or even there is no *Require* key in package manifest), then dependency *io.murano: 0* will be automatically added.

Object version

Now you can specify the version of objects in UI definition when your application requires specific version of some class. To do this, add new key `classVersion` to section `?` describing object:

```
?:  
  type: io.test.apps.TestApp  
  classVersion: 0.0.1
```

`classVersion` of all classes included to package equals `Version` of this package.

2. YAQL

In Liberty, murano was updated to use `yaql 1.0.0`. The new version of YAQL allows you to use a number of new functions and features that help to increase the speed of developing new applications.

Note: Usage of these features makes your applications incompatible with older versions of murano.

Also, in Liberty you can change `Format` in the manifest of package from `1.0` to `1.1` or `1.2`.

- **1.0** - supported by all versions of murano.
- **1.1** - supported by Liberty+. Specify it, if you want to use features from `yaql 0.2` and `yaql 1.0.0` at the same time in your application.
- **1.2** - supported by Liberty+. A number of features from `yaql 0.2` do not work with this format (see the list below). We recommend you to use it for new applications where compatibility with Kilo is not required.

Some examples of `yaql 0.2` features that are not compatible with the `1.2` format

- Several functions now cannot be called as MuranoObject methods: `id()`, `cast()`, `super()`, `psuper()`, `type()`.
- Now you do not have the ability to compare non-comparable types. For example `"string != false"`
- Dicts are not iterable now, so you cannot do this: `If: $key in $dict. Use $key in $dict.keys() or $v in $dict.values()`
- Tuples are not available. `=>` always means keyword argument.

3. Simple software configuration

Previously, you always had to create execution plans even when some short scripts had to be executed on a VM. This process included creating a template file, creating a script, and describing the sending of the execution plan to the murano agent.

Now you can use a new class `io.murano.configuration.Linux` from murano `core-library`. This allows sending short commands to the VM and putting files from the `Resources` folder of packages to some path on the VM without the need of creating execution plans.

To use this feature you need to:

- Declare a namespace (for convenience)

```
Namespaces:
  conf: io.murano.configuration
  ...
```

- Create object of `io.murano.configuration.Linux` class in workflow of your application:

```
$linux: new(conf:Linux)
```

- Run one of the two feature methods: `runCommand` or `putFile`:

```
# first argument is agent of instance, second - your command
$linux.runCommand($.instance.agent, 'service apache2 restart')
```

or:

```
# getting content of file from 'Resources' folder
- $resources: new(sys:Resources)
- $fileContent: $resources.string('your_file.name')
# put this content to some directory on VM
- $linux.putFile($.instance.agent, $fileContent, '/tmp/your_file.name')
```

Note: At the moment, you can use this feature only if your app requires an instance of `LinuxMuranoInstance` type.

4. UI network selection element

Since Liberty, you can provide users with the ability to choose where to join their VM: to a new network created during the deployment, or to an already existing network. Dynamic UI now has a new type of field - `NetworkChoiseField`. This field provides a selection of networks and their subnetworks as a dropdown populated with those which are available to the current project (tenant).

To use this feature, you should make the following updates in the Dynamic UI of an application:

- Add network field:

```
fields:
  - name: network
    type: network
    label: Network
    description: Select a network to join. 'Auto' corresponds to a ↵
    ↵default environment's network.
    required: false
    murano_networks: translate
```

To see the full list of the network field arguments, refer to the UI forms [specification](#).

- Add template:

```
Templates:
  customJoinNet:
    - ?:
      type: io.murano.resources.ExistingNeutronNetwork
      internalNetworkName: $.instanceConfiguration.network[0]
      internalSubnetworkName: $.instanceConfiguration.network[1]
```

- Add declaration of *networks* instance property:

```
Application:
  ?:
    type: com.example.exampleApp
  instance:
    ?:
      type: io.murano.resources.LinuxMuranoInstance
  networks:
    useEnvironmentNetwork: $.instanceConfiguration.network[0]=null
    useFlatNetwork: false
    customNetworks: switch($.instanceConfiguration.network[0], $=null=>
↵list(), $!=null=>$customJoinNet)
```

For more details about this feature, see *use-cases*

Note: To use this feature, the version of UI definition must be **2.1+**

5. Remove name field from fields and object model in dynamic UI

Previously, each class of an application had a name property. It had no built-in predefined meaning for MuranoPL classes and mostly used for dynamic UI purposes.

Now you can create your applications without this property in classes and without a corresponding field in UI definitions. The field for app name will be automatically generated on the last management form before start of deployment. Bonus of deleting this - to remove unused property from muranopl class that is needed for dashboard only.

So, to update existing application developer should make 3 steps:

1. remove name field and property declaration from UI definition;
2. remove name property from class of application and make sure that it is not used anywhere in workflow
3. set version of UI definition to **2.2 or higher**

Migrate applications to Stable/Newton

In Newton a number of useful features that can be used by developers creating their murano applications were implemented. Also some changes are not backward compatible. This document describes these features, how they may be included into the new apps and what benefits the apps may gain.

1. New syntax for the action declaration

Previously, for declaring action in MuranoPL application, following syntax was used:

```
methodName:  
  Usage: Action
```

This syntax is deprecated now for packages with FormatVersion starting from 1.4, and you should use the *Scope* attribute:

```
methodName:  
  Scope: Public
```

For more information about actions in MuranoPL, see *Murano actions*.

2. Usage of static methods as Action

Now you can declare static method as action with *Scope* and *Usage* attributes

```
methodName:  
  Scope: Public  
  Usage: Static
```

For more information about static methods in MuranoPL, see *Static methods and properties*.

3. Template contract support

New contract function `template` was introduced. `template` works similar to the `class` in regards to the data validation but does not instantiate objects. The template is just a dictionary with object model representation of the object.

It is useful when you do not necessarily need to pass the actual object as a property or as a method argument and use it right away, but rather to create new objects of this type in runtime from the given template. It is especially beneficial for resources replication or situations when object creation depends on some conditions.

Objects that are assigned to the property or argument with `template` contract will be automatically converted to their object model representation.

4. Multi-region support

Starting from Newton release cloud resource classes (instances, networks, volumes) can be explicitly put into OpenStack regions other than environment default. Thus it becomes possible to have applications that make use of more than one region including stretching/bursting to other regions.

Each resource class has got new `regionName` property which controls its placement. If no value is provided, default region for environment is used. Applications wanting to take advantage of multi-region support should access security manager and Heat stacks from regions of their resources rather than from the environment.

Regions need to be configured before they can be used. Please refer to documentation on how to do this: [Multi-region application](#).

Changes in the core library

`io.murano.Environment` class contains `regions` property with list of `io.murano.CloudRegion` objects. Heat stack, networks and agent listener are now owned by `io.murano.CloudRegion` instances rather than by `Environment`.

You can not get `io.murano.resources.Network` objects from `Environment::defaultNetworks` now. This property only contains templates for `io.murano.CloudRegion` default networks.

The proper way to retrieve `io.murano.resources.Network` object is now the following:

```
$region: $instance.getRegion()
$networks: $region.defaultNetworks
```

5. Changes to property validation

`string()` contract no longer converts to string anything but scalar values.

6. Garbage collection

New approach to resource deallocation was introduced.

Previously murano used to load `Objects` and `ObjectsCopy` sections of the JSON object model independently which cause for objects that were not deleted between deployments to instantiate twice. If deleted objects were to cause any changes to such alive objects they were made to the objects loaded from `ObjectsCopy` and immediately discarded before the deployment. Now this behaviour is changed and there are no more duplicates of the same object.

Applications can also make use of the new features. Now it is possible to perform on-demand destruction of the unreferenced MuranoPL objects during the deployment from the application code. The `io.murano.system.GC.GarbageCollector.collect()` static method may be used for that.

Also objects obtained ability to set up destruction dependencies to the other objects. Destruction dependencies allow to define the preferable order of objects destruction and let objects be aware of other objects destruction, react to this event, including the ability to prevent other objects from being destroyed.

Please refer to the documentation on how to use the [Garbage Collector](#).

Application unit tests

Murano applications are written in *MuranoPL*. To make the development of applications easier and enable application testing, a special framework was created. So it is possible to add unit tests to an application package and check if the application is in actual state. Also, application deployment can be simulated with unit tests, so you do not need to run the murano engine.

A separate service that is called *murano-test-runner* is used to run MuranoPL unit tests.

All application test cases should be:

- Specified in the MuranoPL class, inherited from `io.murano.test.testFixture`

This class supports loading object model with the corresponding `load(json)` function. Also it contains a minimal set of assertions such as `assertEqual` and etc.

Note, that test class has the following reserved methods are:

- *initialize* is executed once, like in any other murano application
- *setUp* is executed before each test case
- *tearDown* is executed after each test case

- Named with *test* prefix

```
usage: murano-test-runner [-h] [--config-file CONFIG_FILE]
                        [--os-auth-url OS_AUTH_URL]
                        [--os-username OS_USERNAME]
                        [--os-password OS_PASSWORD]
                        [--os-project-name OS_PROJECT_NAME]
                        [-l [</path1, /path2> [</path1, /path2> ...]]] [-v]
                        <PACKAGE_FQN>
                        [<testMethod1, className.testMethod2> [<testMethod1,
→ className.testMethod2> ...]]
```

positional arguments:

<PACKAGE_FQN>

Full name of application package that is going to be tested

<testMethod1, className.testMethod2>

List of method names to be tested

optional arguments:

-h, --help show this help message and exit

--config-file CONFIG_FILE
Path to the murano config

--os-auth-url OS_AUTH_URL
Defaults to env[OS_AUTH_URL]

--os-username OS_USERNAME
Defaults to env[OS_USERNAME]

--os-password OS_PASSWORD
Defaults to env[OS_PASSWORD]

--os-project-name OS_PROJECT_NAME

(continues on next page)

(continued from previous page)

```

        Defaults to env[OS_PROJECT_NAME]
    -l [</path1 /path2> [</path1 /path2> ...]], --load_packages_from [</path1 /
↳path2> [</path1 /path2> ...]]
        Directory to search packages from. Will be used.
↳instead of
        directories, provided in the same option in murano.
↳configuration file.
    -v, --verbose          increase output verbosity
    --version             show program's version number and exit

```

The fully qualified name of a package is required to specify the test location. It can be an application package that contains one or several classes with all the test cases, or a separate package. You can specify a class name to execute all the tests located in it, or specify a particular test case name.

Authorization parameters can be provided in the murano configuration file, or with higher priority `-os-parameters`.

Consider the following example of test execution for the Tomcat application. Tests are located in the same package with application, but in a separate class called `io.murano.test.TomcatTest`. It contains `testDeploy1` and `testDeploy2` test cases. The application package is located in the `/package/location/directory` (murano-apps repository e.g). As the result of the following command, both test cases from the specified package and class will be executed.

```

murano-test-runner io.murano.apps.apache.Tomcat io.murano.test.TomcatTest -l /
↳package/location/directory /io.murano/location -v

```

The following command runs a single `testDeploy1` test case from the application package.

```

murano-test-runner io.murano.apps.apache.Tomcat io.murano.test.TomcatTest.
↳testDeploy1

```

The main purpose of MuranoPL unit test framework is to enable mocking. Special [YAQL](#) functions are registered for that:

def inject(target, target_method, mock_object, mock_name)

inject to set up mock for *class* or *object*, where mock definition is a *name of the test class method*

def inject(target, target_method, yaql_expr)

inject to set up mock for *a class* or *object*, where mock definition is a *YAQL expression*

Parameters description:

target

MuranoPL class name (namespaces can be used or full class name in quotes) or MuranoPL object

target_method

Method name to mock in target

mock_object

Object, where mock definition is contained

mock_name

Name of method, where mock definition is contained

yaql_expr

YAQL expression, parameters are allowed

So the user is allowed to specify mock functions in the following ways:

- Specify a particular method name
- Provide a YAQL expression

Consider how the following functions may be used in the MuranoPL class with unit tests:

```

Namespaces:
  =: io.murano.test
  sys: io.murano.system

Extends: TestFixture

Name: TomcatTest

Methods:
  initialize:
    Body:
      # Object model can be loaded from JSON file, or provided
      # directly in MuranoPL code as a YAML insertion.
      - $.appJson: new(sys:Resources).json('tomcat-for-mock.json')
      - $.heatOutput: new(sys:Resources).json('output.json')
      - $.log: logger('test')
      - $.agentCallCount: 0

      # Mock method to replace the original one
      agentMock:
        Arguments:
          - template:
              Contract: $
          - resources:
              Contract: $
          - timeout:
              Contract: $
              Default: null
        Body:
          - $.log.info('Mocking murano agent')
          - $.assertEqual('Deploy Tomcat', $template.Name)
          - $.agentCallCount: $.agentCallCount + 1

      # Mock method, that returns predefined heat stack output
      getStackOut:
        Body:
          - $.log.info('Mocking heat stack')
          - Return: $.heatOutput

      testDeploy1:
        Body:
          # Loading object model
          - $.env: $this.load($.appJson)

```

(continues on next page)

(continued from previous page)

```

# Set up mock for the push method of *io.murano.system.HeatStack*
↪class
- inject(sys:HeatStack, push, $.heatOutput)

# Set up mock with YAQL function
- inject($.env.stack, output, $.heatOutput)

# Set up mock for the concrete object with mock method name
- inject('io.murano.system.Agent', call, $this, agentMock)

# Mocks will be called instead of original function during the
↪deployment
- $.env.deploy()

# Check, that mock worked correctly
- $.assertEqual(1, $.agentCallCount)

testDeploy2:
  Body:
    - inject(sys:HeatStack, push, $this, getStackOut)
    - inject(sys:HeatStack, output, $this, getStackOut)

    # Mock is defined with YAQL function and it will print the original
↪variable (agent template)
    - inject(sys:Agent, call, withOriginal(t => $template) -> $.log.info('
↪{0}', $t))

    - $.env: $this.load($.appJson)
    - $.env.deploy()

    - $.isDeployed: $.env.applications[0].getattr(deployed, false, 'com.
↪example.apache.Tomcat')
    - $.assertEqual(true, $.isDeployed)

```

Provided methods are test cases for the Tomcat application. Object model and heat stack output are predefined and located in the package Resources directory. By changing some object model or heat stack parameters, different cases may be tested without a real deployment. Note, that some asserts are used in those example. The first one is checked, that agent call function was called only once as needed. And assert from the second test case checks for a variable value at the end of the application deployment.

Test cases examples can be found in `TomcatTest.yaml` class of the Apache Tomcat application located at [murano-apps repository](#). You can run test cases with the commands provided above.

Cinder volume support

Cinder volume is a block storage service for OpenStack, which represents a detachable device, similar to a USB hard drive. You can attach a volume to only one instance. In murano, it is possible to work with Cinder volumes in several ways:

- Attaching Cinder volumes to murano instance
- Booting from Cinder volume

Below both ways are considered with ApacheHttpServer application as an example.

For more information about Cinder volumes, see [Manage Cinder volumes](#).

Attaching Cinder volumes

Several volumes can be attached to the murano instance. Consider an example that shows how to attach a created volume to the instance (next, in the *Booting from Cinder volume* section, we are going to boot from a volume created by us).

Example

1. In the OpenStack dashboard, go to *Volumes* to create a volume.
2. Modify the `ui.yaml` file:

```

....
Application:
  ....
  instance:
    ....
    volumes:
      $.volumeConfiguration.volumePath:
        ?:
          type: io.murano.resources.ExistingCinderVolume
          openstackId: $.volumeConfiguration.volumeID
....

```

An existing Cinder volume can be initialized with its `openstackId` and can be attached with its `volumePath`. These parameters come here from modified `Forms` section of the `ui.yaml` file:

```

....
Forms:
- appConfiguration:
  ....
- instanceConfiguration:
  ....
- volumeConfiguration:
  fields:
    - name: volumeID
      type: string

```

(continues on next page)

(continued from previous page)

```

label: Existing volume ID
description: Put in existing volume openstackID
required: true
- name: volumePath
  type: string
  label: Path
  description: Put in volume path to be mounted
  required: true

```

Therefore, create a ZIP archive of the built package and upload it to murano. Attach created application to the environment. Enter its openstackId (which can be found in OpenStack dashboard) and path for mounting. For example, you can fill the latter with /dev/vdb value.

After the application is deployed, verify that the volume is attached to the instance in the OpenStack dashboard *Volumes* tab. Alternatively, see the topology of the Heat Stack.

Booting from Cinder volume

You can create a volume from an existing image. The example below shows how to create a volume from an image and use the volume to boot an instance.

Example

It is possible to create a volume through the Heat template, instead of the OpenStack dashboard. For this, modify the ui.yaml file:

```

....

Templates:
  customJoinNet:
    ....
  bootVolumes:
    - volume:
      ?:
        type: io.murano.resources.CinderVolume
        size: $.instanceConfiguration.volSize
        sourceImage: $.instanceConfiguration.osImage
        bootIndex: 0
        deviceName: vda
        deviceType: disk
    ....

Application:
  ....
  instance:
    ....
    blockDevices: $bootVolumes
  ....

```

The example above shows that the Templates section now has a bootVolumes field, which is stored

in the changed Application section. Pay attention that image property should be deleted from Application to avoid defining both image and volume to boot. The size and sourceImage properties come in Templates from the changed Forms section of the ui.yaml file:

```

....
Forms:
- appConfiguration:
  ....
- instanceConfiguration:
  fields:
  ....
  - name: volSize
    type: integer
    label: Size of volume
    required: true
    description: >-
      Specify volume size which is going to be created from image
  ....

```

After sending this package to murano you can boot your instance from the volume by chosen image.

Multi-region application

Since Newton release, Murano supports multi-region application deployment. All MuranoPL resource classes are inherited from the `io.murano.CloudResource` class. An application developer can set a custom region for `CloudResource` subclasses deployment.

Set a region for resources

To set a region for resources:

1. Specify a region for `CloudResource` subclasses deployment through the `regionName` property. For example:

```

Application:
?:
  type: com.example.apache.ApacheHttpServer
  enablePHP: $.appConfiguration.enablePHP
  ...
instance:
?:
  type: io.murano.resources.LinuxMuranoInstance
  regionName: 'CustomRegion'
  ...

```

2. Retrieve `io.murano.CloudRegion` objects:

```

$region: $.instance.getRegion()
$regionName: $region.name
$regionLocalStack: $region.stack
$regionDefaultNetworks: $region.defaultNetworks

```

As a result, all region-local properties are moved from the `io.murano.Environment` class to the new *Class: `CloudRegion`* class. For backward compatibility, the `io.murano.Environment` class stores region-specific properties of default region, except the `defaultNetworks` in its own properties. The `Environment::defaultNetworks` property contains templates for the `CloudRegion::defaultNetworks` property.

Through current UI, you cannot select networks, flavor, images and availability zone from a non-default region. We suggest using regular text fields to specify region-local resources.

Networking and multi-region applications

By default, each region has its own separate network. To ensure connectivity between the networks, create and configure networks in regions before deploying the application and use `io.murano.resources.ExistingNeutronNetwork` to connect the instance to an existing network. Example:

```

Application:
?:
  type: application.fully.qualified.Name
  ...

instance_in_region1:
?:
  type: io.murano.resources.LinuxMuranoInstance
  regionName: 'CustomRegion1'
  networks:
    useEnvironmentNetwork: false
    useFlatNetwork: false
    customNetworks:
      - ?:
        type: io.murano.resources.ExistingNeutronNetwork
        regionName: 'CustomRegion1'
        internalNetworkName: 'internalNetworkNameInCustomRegion1'
        internalSubnetworkName: 'internalSubNetNameInCustomRegion1'

instance_in_region2:
?:
  type: io.murano.resources.LinuxMuranoInstance
  regionName: 'CustomRegion2'
  networks:
    useEnvironmentNetwork: false
    useFlatNetwork: false
    customNetworks:
      - ?:
        type: io.murano.resources.ExistingNeutronNetwork

```

(continues on next page)

(continued from previous page)

```

regionName: 'CustomRegion2'
internalNetworkName: 'internalNetworkNameInCustomRegion2'
internalSubnetworkName: 'internalSubNetNameInCustomRegion2'

```

```
...
```

Also, you can configure networks with the same name and use a template for the region networks. That is, describe `io.murano.resources.ExistingNeutronNetwork` only once and assign it to the `Environment::defaultNetworks::environment` property. The environment will create `Network` objects for regions from the `ExistingNeutronNetwork` template. Example:

```

OS_REGION_NAME="RegionOne" openstack network create <NETWORK-NAME>
OS_REGION_NAME="RegionTwo" openstack network create <NETWORK-NAME>

# configure subnets
#...

# add ExistingNeutronNetwork to environment object model
murano environment-create --join-net-id <NETWORK-NAME> <ENV_NAME>

# also it is possible to specify subnet from <NETWORK-NAME>
murano environment-create --join-net-id <NETWORK-NAME> --join-subnet-id
↳<SUBNET_NAME> <ENV_NAME>

```

Additionally, consider the `[networking]` section in the configuration file. Currently, `[networking]` settings are common for all regions.

[networking]

```

external_network = %EXTERNAL_NETWORK_NAME%
router_name = %MURANO_ROUTER_NAME%
create_router = true

```

If you choose an automatic neutron configuration, configure the external network with identical names in all regions. If you disable the automatic router creation, create routers with identical names in all regions. Also, the `default_dns` address must be reachable from all created networks.

Note: To use regions, first configure them as described in *Support for OpenStack regions*.

Examples

Application name	Description
Zabbix Agent	<p>Zabbix Agent is a simple application. It doesn't deploy a VM by itself, but is installed on a specific VM that may contain any other applications. This VM is tracked by Zabbix and by its configuration. So Murano performs the Zabbix agent configuration based on the user input. The user chooses the way of instance tracking - HTTP or ICMP that may perform some modifications in the application package. It is worth noting that application scripts are written in Python, not in Bash as usual. This application does not work without Zabbix server application since it's a required property, determined in the application definition.</p>
Zabbix Server	<p>Zabbix Server application interacts with Zabbix Agent by calling its setUpAgent method and providing information about itself: IP and host-name of VM on which the server is installed. Server installs MySQL database and requests database name, password and some other parameters from the user.</p>
Docker Crate	<p>This is a good example on how difficult logic may be simplified with the inheritance that is supported by MuranoPL. Definition of this app is simple, but the opportunity it provides is fantastic. Crate is a distributed database, in the Murano Application catalog it looks like a regular application. It may be deployed on Google Kubernetes or regular Docker server. The user picks the desired option while filling in the form since these options are set in the UI definition. The form field has a list of possible options:</p> <pre> ... type: - com.mirantis.docker.kubernetes.KubernetesPod - com.mirantis.docker.DockerStandaloneHost </pre> <p>Information about the application itself (docker image and port that is needed to be opened) is contained in the getContainer method. All other actions for the application configuration are located at the DockerStandaloneHost definition and its dependencies. Note that this application doesn't have a filename:Resources folder at all since the installation is made by Docker itself.</p>

Use-cases

Performing application interconnections

Murano can handle application interconnections installed on virtual machines. The decision of how to combine applications is made by the author of an application.

To illustrate the way such interconnection can be configured, let's analyze the mechanisms applied in WordPress application, which uses MySQL.

MySQL is a very popular database and can be used in quite a number of various applications. Instead of the creation of a database inside definition of the WordPress application, it calls the methods from the MySQL class. At the same time MySQL remains an independent application.

MySQL has a number of methods:

- `deploy`
- `createDatabase`
- `createUser`
- `assignUser`
- `getConnectionString`

In the `com.example.WordPress` class definition the database property is a contact for the `com.example.databases.MySql` class. So, the database configuration methods can be called with the parameters passed by the user in the main method:

```
- $.database.createDatabase($.dbName)
- $.database.createUser($.dbUser, $.dbPassword)
- $.database.assignUser($.dbUser, $.dbName)
```

Any other methods of any other class can be invoked the same way to make the proposal application installation algorithm clear and constructive. Also, it allows not to duplicate the code in new applications.

Abstract dependencies between applications

In the example above it is also possible to specify a generic class in the contract `com.example.databases.SqlDatabase` instead of `com.example.databases.MySql`. It means that an object of any class inherited from `com.example.databases.SqlDatabase` can be passed to a parameter. In this case you should also use this generic class as a type for a field in the file `ui.yaml`:

Forms:

```
- appConfiguration:
  fields:
    - name: database
      type: com.example.databases.SqlDatabase
      label: Database Server
      description: >-
        Select a database server to host the application`s database
```

After that you can choose any database package in a drop-down box. The last place, which should be changed in the WordPress package to enable this feature, is manifest file. It should contain the full name of SQL Library package and optionally packages inherited from SQL library if you want them to be downloaded as dependencies. For example:

Require:

```
com.example.databases:
com.example.databases.MySql:
com.example.databases.PostgreSql:
```

Note: To use this feature you have to enable Glare as a storage for your packages and a version of your `murano-dashboard` should be not older than `newton`.

Using application already installed on the image

Suppose you have everything already prepared on image. And you want to share this image with others. This problem can be solved in several ways.

Let's use the [HDPSandbox](#) application to illustrate how this can be done with Murano.

Note: An image may not contain murano-agent at all.

Prepare an application package of the structure:

```
|_ Classes
|   |_ HDPSandbox.yaml
|
|_ UI
|   |_ ui.yaml
|
|_ logo.png
```

Note: The Resources folder is not included in the package since the image contains everything that user expects. So no extra instructions are needed to be executed on murano-agent.

UI is provided for specifying the application name, which is used for the application recognition in logging. And what is more, it contains the image name as a deployment instruction template (object model) in the `Application` section:

```
1 Application:
2 ?:
3   type: com.example.HDPSandbox
4 name: $.appConfiguration.name
5 instance:
6   ?:
7     type: io.murano.resources.LinuxMuranoInstance
8     name: generateHostname($.instanceConfiguration.unitNamingPattern, 1)
9     flavor: $.instanceConfiguration.flavor
10    image: 'hdp-sandbox'
11    assignFloatingIp: true
```

Moreover, the unsupported flavors can be specified here, so that the user can select only from the valid ones. Provide the requirements in the corresponding section to do this:

```
requirements:
  min_disk: 50          (Gb)
  min_memory_mb: 4096  (Mb)
  min_vcpus: 1
```

After the UI form creation, and the HDPSandbox application deployment, the VM with the predefined image is spawned. Such type of applications may interact with regular applications. Thus, if you have an image with Puppet, you can call the `deploy` method of the Puppet application and then puppet manifests or any shell scripts on the freshly spawned VM.

The presence of the `logo.png` should never be underestimated, since it helps to make your application recognizable among other applications included in the catalog.

Interacting with non-OpenStack services

This section tells about the interaction between an application and any non-OpenStack services, that have an API.

External load-balancer

Suppose, you have powerful load-balancer on a real server. And you want to run the application on an OpenStack VM. Murano can set up new applications to be managed by that external load-balancer (LB). Let's go into more details.

To implement this case the following apps are used:

- **LbApp**: its class methods call LB API
- **WebApp**: runs on the real LB

Several instances of **WebApp** are deployed with each of them calling two methods:

```
- $.loadBalancer.createPool()
- $.loadBalancer.addMember($instance)
# where $.loadBalancer is an instance of the LbApp class
```

The first method creates a pool and associates it with a virtual server. This happens once only. The second one registers a member in the newly created pool.

It is also possible to perform other modifications to the LB configuration, which are only restricted by the LB API functionality.

So, you need to specify the maximum instance number in the UI form related to the **WebApp** application. All of them are subsequently added to the LB pool. After the deployment, the LB virtual IP, by which an application is accessible, is displayed.

Configuring Network Access for VMs

By default, each VM instance deployed by `io.murano.resources.Instance` class or its descendants joins an environment's default network. This network gets created when the Environment is deployed for the first time, a subnet is created in it and is uplinked to a router which is detected automatically based on its name.

This behavior may be overridden in two different ways.

Using existing network as environment's default

This option is available for users when they create a new environment in the Dashboard. A dropdown control is displayed next to the input field prompting for the name of environment. By default this control provides to create a new network, but the user may opt to choose some already existing network to be the default for the environment being created. If the network has more than one subnet, the list will include all the available options with their CIDRs shown. The selected network will be used as environment's default, so no new network will be created.

Note: Murano does not check the configuration or topology of the network selected this way. It is up to the user to ensure that the network is uplinked to some external network via a router - otherwise the murano engine will not be able to communicate with the agents on the deployed VMs. If the Applications being deployed require internet connectivity it is up to the user to ensure that this net provides it, than DNS nameservers are set and accessible etc.

Modifying the App UI to prompt user for network

The application package may be designed to ask user about the network they want to use for the VMs deployed by this particular application. This allows to override the default environment's network setting regardless of its value.

To do this, application developer has to include a `network` field into the Dynamic UI definition of the app. The value returned by this field is a tuple of `network_id` and a `subnet_id`. This values may be passed as the input properties for `io.murano.resources.ExistingNeutronNetwork` object which may be in its turn passed to an instance of `io.murano.resources.Instance` as its network configuration.

The UI definition may look like this:

```
Templates:
  customJoinNet:
    - ?:
      type: io.murano.resources.ExistingNeutronNetwork
      internalNetworkName: $.instanceConfiguration.network[0]
      internalSubnetworkName: $.instanceConfiguration.network[1]
Application:
  ?:
    type: com.example.someApplicationName
    instance:
      ?:
        type: io.murano.resources.LinuxMuranoInstance
        networks:
          useEnvironmentNetwork: $.instanceConfiguration.network[0]=null
          useFlatNetwork: false
          customNetworks: switch($.instanceConfiguration.network[0], $=null=>
->list(), $!=null=>$customJoinNet)
Forms:
  - instanceConfiguration:
    fields:
      - name: network
```

(continues on next page)

(continued from previous page)

```
type: network
label: Network
description: Select a network to join. 'Auto' corresponds to a ↵
↵default environment's network.
required: false
murano_networks: translate
```

For more details on the Dynamic UI its controls and templates please refer to its *specification*.

Application development framework

Application development framework is a library that helps application developers to create applications that can be scalable, highly available, (self)healable and do not contain boilerplate code for common application workflow operations. This library is placed into the Murano repository under the `meta/io.murano.applications` folder.

To allow your applications to use the code of the library, zip it and upload to the Murano application catalog.

Framework objectives

The library allows application developers to focus on their application-specific tasks without the real need to dive into resource orchestration, server farm configuration, and so on. For example, on how to install the software on the VMs, how to configure it to interact with other applications. Application developers are able to focus more on the software configuration tools (scripts, puppets, and others) and care less about the MuranoPL if they do not need to define any custom workflow logic.

The main capabilities the library provides and its main use-cases are as follows:

- Standard operations are implemented in the framework and can be left as is
- The capability to create multi-server applications and scale them
- The capability to create composite multi-component applications
- The capability to track application failures and recover from them
- The capability to define event handlers for various events

Quickstart

To use the framework in your application, include the following lines to the `manifest.yaml` file:

```
Require:
io.murano.applications:
```

Create a one-component single-server application

To create a simple application deployed on a single server:

1. Include the following lines to the code of the application class:

```
Namespaces:  
  =: my.new.ns  
  apps: io.murano.applications  
  
Name: AppName  
Extends: apps:SingleServerApplication
```

2. Provide an input for the application server property in your `ui.yaml` file:

```
Application:  
  ?:  
    type: my.new.ns.AppName  
  server:  
    ?:  
      type: io.murano.resources.LinuxMuranoInstance  
      name: generateHostname($.instanceConfiguration.unitNamingPattern, 1)  
      flavor: $.instanceConfiguration.flavor  
      ... <other instance properties>
```

Now you already have the app that creates a server ready for installing software on it.

3. To create a fully functional app, add an installation script to the body of the `onInstallServer` method:

```
Methods:  
  onInstallServer:  
    Arguments:  
      - server:  
        Contract: $.class(res:Instance).NotNull()  
      - serverGroup:  
        Contract: $.class(apps:ServerGroup).NotNull()  
    Body:  
      - $file: sys:Resources.string('installScript.sh')  
      - conf:Linux.runCommand($server.agent, $file)
```

4. Optional. Add other methods that handle certain stages of the application workflow, such as `onBeforeInstall`, `onCompleteInstallation`, `onConfigureServer`, `onCompleteConfiguration`, and others. For details about these methods, see the *Software components* section.

Create a one-component multi-server application

To create an application that is intended to be installed on several servers:

1. Make it inherit the `MultiServerApplication` class:

```
Namespaces:
  =: my.new.ns
  apps: io.murano.applications

Name: AppName
Extends: apps:MultiServerApplication
```

2. Instead of the `server` property in `SingleServerApplication`, provide an input for the `servers` property that accepts the instance of one of the inheritors of the `ServerGroup` class. The `ui.yaml` file in this case may look as follows:

```
Application:
  ?:
    type: my.new.ns.AppName
  servers:
    ?:
      type: io.murano.applications.ServerList
    servers:
      - ?:
          type: io.murano.resources.LinuxMuranoInstance
          name: "Server-1"
          flavor: $.instanceConfiguration.flavor
          ... <other instance properties>
      - ?:
          type: io.murano.resources.LinuxMuranoInstance
          name: "Server-2"
          flavor: $.instanceConfiguration.flavor
          ... <other instance properties>
```

3. Define the custom logic of the application in the handler methods, and it will be applied to the whole app, exactly like with `SingleServerApplication`.

Create a scalable multi-server application

To provide the application with the ability to scale:

1. Make the app extend the `MultiServerApplicationWithScaling` class:

```
Namespaces:
  =: my.new.ns
  apps: io.murano.applications

Name: AppName
Extends: apps:MultiServerApplicationWithScaling
```

2. Provide the `ui.yaml` file:

```
Application:
?:
  type: my.new.ns.AppName
servers:
  ?:
    type: io.murano.applications.ServerReplicationGroup
    numItems: $.appConfiguration.numNodes
    provider:
      ?:
        type: io.murano.applications.TemplateServerProvider
    template:
      ?:
        type: io.murano.resources.LinuxMuranoInstance
        flavor: $.instanceConfiguration.flavor
        ... <other instance properties>
    serverNamePattern: $.instanceConfiguration.unitNamingPattern
```

The `servers` property accepts instance of the `ServerReplicationGroup` class, and in turn it requires input of the `numItems` and `provider` properties.

After the deployment, the `scaleOut` and `scaleIn` public methods (actions) become available in the dashboard UI.

For a working example of such application, see the `com.example.apache.ApacheHttpServer` package version 1.0.0.

Library overview

The framework includes several groups of classes:

replication.yaml

Classes that provide the capability to replicate the resources.

servers.yaml

Classes that provide instances grouping and replication.

component.yaml

Classes that define common application workflows.

events.yaml

Class for handling events.

baseapps.yaml

Base classes for applications.

As it is described in the *QuickStart* section, the application makes use of the Application development framework by inheriting from one of the base application classes, such as `SingleServerApplication`, `MultiServerApplication`, `MultiServerApplicationWithScaling`. In turn, these classes are inheritors of the standard `Application` class and the `SoftwareComponent` class. The latter class binds all of the framework capabilities.

The `SoftwareComponent` class inherits both `Installable` and `Configurable` classes which provide boilerplate code for the installation and configuration workflow respectively. They also contain empty

methods for each stage of the workflow (e.g. `onBeforeInstall`, `onInstallServer`), which are the places where application developers can add their own customization code.

The entry point to execute deployment of the software component is its `deployAt` method which requires instance of one of the inheritors of the `serverGroup` class. It is the object representing the group of servers the application should be deployed to. The application holds such an object as one of its properties. It can be a single server (`SingleServerGroup` subclass), a prepopulated list of servers (`ServerList` subclass) or a list of servers that are dynamically generated in runtime (`ServerReplicationGroup` subclass).

`ServerReplicationGroup` or, more precisely, one of its parent classes `ReplicationGroup` controls the number of items it holds by releasing items over the required amount and requesting creation of the new items in runtime from the `ReplicaProvider` class which acts like an object factory. In case of servers, it is `TemplateServerProvider` which creates new servers from the given template. Replication is done during the initial deployment and during the scaling actions execution.

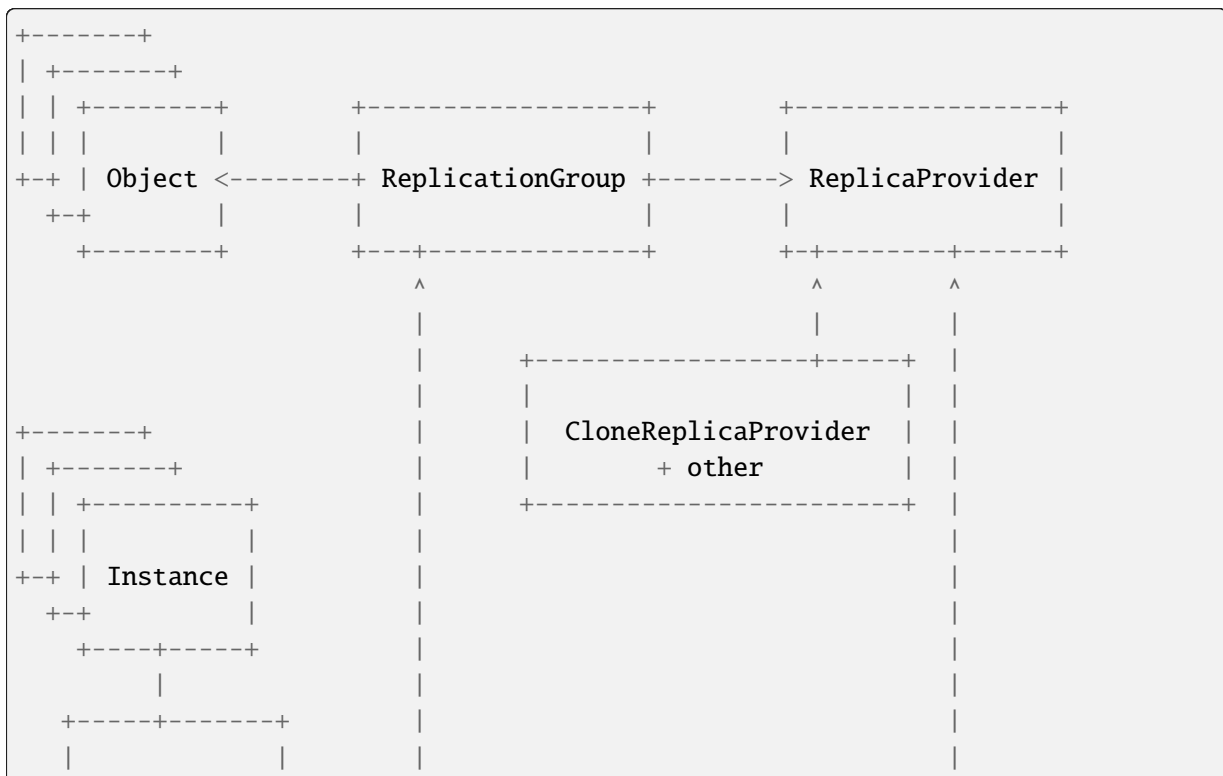
Framework detailed description

This section provides technical description of all the classes present in the application development library, their hierarchy and usage.

Scaling primitives

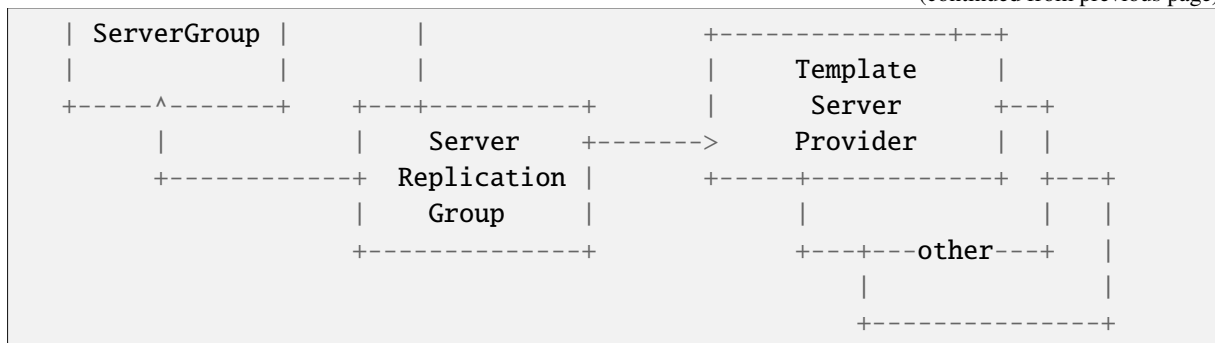
There is an ability to group similar resources together, produce new copies of the same resources or release the existing ones on request. Now it is implemented for instances only, other resources may be added later.

The following is the hierarchy of classes that provide grouping and replication of resources:



(continues on next page)

(continued from previous page)



ReplicationGroup

A base class which holds the collection of objects generated in runtime in its `items` output property and contains a reference to a `ReplicaProvider` object in its `provider` property which is used to dynamically generate the objects in runtime.

Input properties of this class include the `minItems` and `maxItems` allowing to limit the number of objects it holds in its collection.

An input-output property `numItems` allows to declaratively change the set of objects in the collection by setting its size.

The `deploy()` method is used to apply the replica settings: it drops the objects from the collection if their number exceeds the number specified by the `numItems` or generate some new if there are not enough of them.

The `scale()` method is used to increase or decrease the `numItems` by some number specified in the `delta` argument of the method, but in range between `maxItems` and `minItems`.

ReplicaProvider

A class which does the object replication. The base one is abstract, its inheritors should implement the abstract `createReplica` method to create the actual object. The method accepts the `index` parameter to properly parametrize the newly created copy and optional `owner` parameter to use it as an owner for the newly created objects.

The concrete implementations of this class should define all the input properties needed to create new instances of object. Thus the provider actually acts as a template of the object it generates.

CloneReplicaProvider

An implementation of `ReplicaProvider` capable to create replicas by cloning some user-provided object, making use of the `template()` contract.

PoolReplicaProvider

Replica provider that takes replicas from the prepopulated pool instead of creating them.

RoundrobinReplicaProvider

Replica provider with a load balancing that returns replica from the prepopulated list. Once the provider runs out of free items it goes to the beginning of the list and returns the same replicas again.

CompositeReplicaProvider

Replica provider which is a composition of other replica providers. It holds the collection of providers in its `providers` input property. Its `ReplicaProvider` method returns a new replica created by the first provider in that list. If that value is *null*, the replica created by the second provider is returned, and so on. If no not-null replicas are created by all providers, the method returns null.

This provider can be used to have some default provider with the ability to fall back to the next options if the preferable one is not successful.

Servers replication

ServerGroup

A class that provides static methods for deployment and releasing resources on the group of instances.

The `deployServers()` static method accepts instance of `ServerGroup` class and a list of servers as the parameters and deploys all servers from the list in the environment which owns the server group, unless server is already deployed.

The `releaseServers()` static method accepts a list of servers as the parameter and consequentially calls `beginReleaseResources()` and `endReleaseResources()` methods on each server.

ServerList

A class that extends the `ServerGroup` class and holds a group of prepopulated servers in its `servers` input property.

The `deploy()` method calls the `deployServers()` method with the servers defined in the `servers` property.

The `.destroy()` method calls the `releaseServers()` method with the servers defined in the `servers` property.

SingleServerGroup

Degenerate case of a `ServerGroup` which consists of a single server. Has the `server` input property to hold a single server.

CompositeServerGroup

A server group that is composed of other server groups.

ServerReplicationGroup

A subclass of the `ReplicationGroup` class and the `ServerGroup` class to replicate the `Instance` objects it holds.

The `deploy()` method of this group not only generates new instances of servers but also deploys them if needed.

TemplateServerProvider

A subclass of `ReplicaProvider` which is used to produce the objects of one of the `Instance` class inheritors by creating them from the provided template with parameterization of the hostnames. The resulting hostname looks like `'Server {index}{groupName}'`.

May be passed as `provider` property to objects of the `ServerReplicationGroup` class.

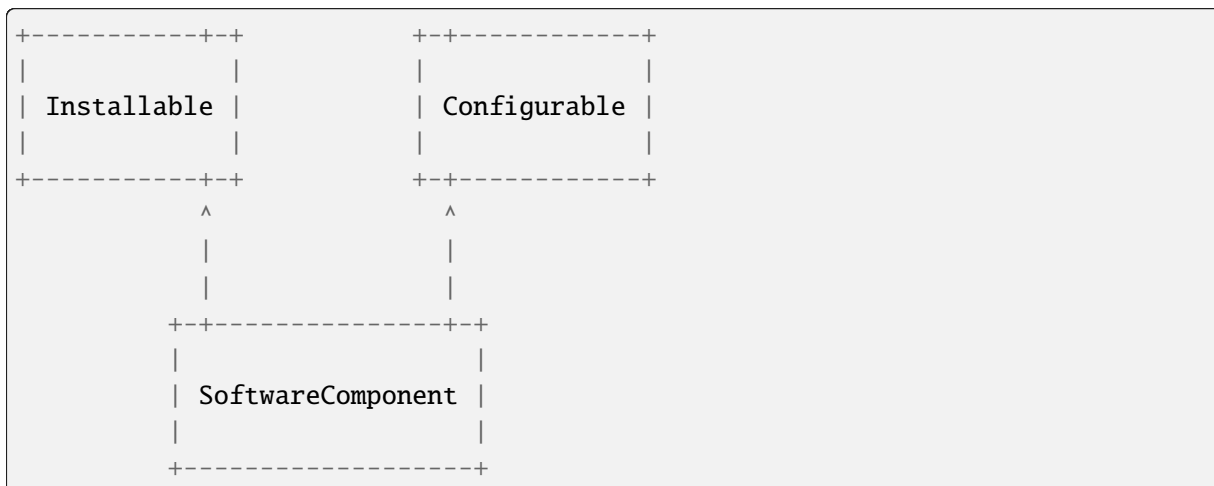
other replica providers

Other subclasses of `ReplicaProvider` may be created to produce different objects of `Instance` class and its subclasses depending on particular application needs.

Classes for grouping and replication of other kinds of resources are to be implemented later.

Software Components

The class to handle the lifecycle of the application is the `SoftwareComponent` class which is a subclass of `Installable` and `Configurable`:



The hierarchy of the `SoftwareComponent` classes is used to define the workflows of different application lifecycles. The general logic of the application behaviour is contained in the methods of the base classes and the derived classes are able to implement the handlers for the custom logic. The model is event-driven: the workflow consists of the multiple steps, and most of the steps invoke appropriate `on%StepName%` methods intended to provide application-specific logic.

Now 'internal' steps logic and their 'public' handlers are split into the separate methods. It should improve the developers' experience and simplify the code of the derived classes.

The standard workflows (such as Installation and Configuration) are defined by the `Installable` and `Configurable` classes respectively. The `SoftwareComponent` class inherits both these classes and defines its deployment workflow as a sequence of Installation and Configuration flows. Other future implementations may add new workflow interfaces and mix them in to change the deployment workflow or add new actions.

Installation workflow consists of the following methods:



(continues on next page)

(continued from previous page)



Method	Arguments	Description
install	serverGroup	Entry point of the installation workflow. Iterates through all the servers of the passed ServerGroup and calls the <code>checkServerIsInstalled</code> method for each of them. If at least one of the calls has returned <i>false</i> , calls a <code>beforeInstall</code> method. Then, for each server which returned <i>false</i> as the result of the <code>checkServerIsInstalled</code> calls the <code>installServer</code> method to do the actual software installation. After the installation is completed on all the servers and if at least one of the previous calls of <code>checkServerIsInstalled</code> returned <i>false</i> , the method runs the <code>completeInstallation</code> method. If all the calls to <code>checkServerIsInstalled</code> return <i>true</i> , this method concludes without calling any others.
check-ServerIsInstalled	server	Checks if the given server requires a (re)deployment of the software component. By default checks for the value of the attribute <i>installed</i> of the instance. May be overridden by subclasses to provide some better logic (e.g. the app developer may provide code to check if the given software is pre-installed on the image which was provisioned on the VM).
beforeInstall	servers, serverGroup	Reports the beginning of installation process, sends notification about this event to all objects which are subscribed for it (see <i>Event notification pattern</i> section for details) and calls the public event handler <code>onBeforeInstall</code> .
onBeforeInstall	servers, serverGroup	Public handler of the <i>beforeInstall</i> event. Empty in the base class, may be overridden in subclasses if some custom pre-install logic needs to be executed.
installServer	server, serverGroup	Does the actual software deployment on a given server by calling an <code>onInstallServer</code> public event handler (with notification on this event). If the installation completes successfully sets the <i>installed</i> attribute of the server to <i>true</i> , reports successful installation and returns <i>null</i> . If an exception encountered during the invocation of <code>onInstallServer</code> , the method handles that exception, reports a warning and returns the server. The return value of the method indicates to the <code>install</code> method how many failures encountered in total during the installation and with what servers.
onInstallServer	server, serverGroup	An event-handler method which is called by the <code>installServer</code> method when the actual software deployment is needed. It is empty in the base class. The implementations should override it with custom logic to deploy the actual software bits.
completeInstallation	servers, serverGroup, failedServer	It is executed after all the <code>installServer</code> methods were called. Checks for the number of errors reported during the installation: if it is greater than the value of <code>allowedInstallFailures</code> property, an exception is raised to interrupt the deployment workflow. Otherwise the method emits notification on this event, calls an <code>onCompleteInstallation</code> event handler and then reports the successful completion of the installation workflow.
onCompleteInstallation	servers, serverGroup, failedServer	An event-handler method which is called by the <code>completeInstallation</code> method when the component installation is about to be completed. Default implementation is empty. Inheritors may implement this method to add some final handling, reporting etc.

Configuration workflow consists of the following methods:



Method	Arguments	Description
configure	serverGroup	Entry point of the configuration workflow. Calls a <code>checkClusterIsConfigured</code> method. If the call returns <i>true</i> , workflow exits without any further action. Otherwise for each server in the <code>serverGroup</code> it calls <code>checkServerIsConfigured</code> method and gets the list of servers that need reconfiguration. The <code>preConfigure</code> method is called with that list. At the end calls the <code>completeConfiguration</code> method.
checkClusterIsConfigured	serverGroup	Has to return <i>true</i> if the configuration (i.e. the values of input properties) of the component has not been changed since it was last deployed on the given server group. Default implementation calls the <code>getConfigurationKey</code> method and compares the returned result with a value of <code>configuration</code> attribute of <code>serverGroup</code> . If the results match returns <i>true</i> otherwise <i>false</i> .
getConfigurationKey	None	Should return some values describing the configuration state of the component. This state is used to track the changes of the configuration by the <code>checkClusterIsConfigured</code> and <code>checkServerIsConfigured</code> methods. Default implementation returns a synthetic value which gets updated on every environment redeployment. Thus the subsequent calls of the <code>configure</code> method on the same server group during the same deployment will not cause the reconfiguration, while the calls on the next deployment will reapply the configuration again. The inheritors may redefine this to include the actual values of the configuration properties, so the configuration is reapplied only if the appropriate input properties are changed.
checkServerIsConfigured	server, serverGroup	It is called to check if the particular server of the server group has to be reconfigured thus providing more precise control compared to cluster-wide <code>checkClusterIsConfigured</code> . Default implementation calls the <code>getConfigurationKey</code> method and compares the returned result with a value of <code>configuration</code> attribute of the server. If the results match returns <i>true</i> otherwise <i>false</i> . This method gets called only if the <code>checkClusterIsConfigured</code> method returned <i>false</i> for the whole server group.
preConfigure	servers, serverGroup	Reports the beginning of configuration process, calls the <code>configureSecurity</code> method, emits the notification and calls the public event handler <code>onPreConfigure</code> . This method is called once per the server group and only if the changes in configuration are detected.
configureSecurity	servers, serverGroup	Intended for configuring the security rules. It is empty in the base class. Fully implemented in the <code>OpenStackSecurityConfigurable</code> class which is the inheritor of <code>Configurable</code> .
onPreConfigure	servers, serverGroup	Public event-handler which is called by the <code>preConfigure</code> method when the (re)configuration of the component is required. Default implementation is empty. Inheritors may implement this method to set various kinds of cluster-wide states or output properties which may be of use at later stages of the workflow.
configureServer	server, serverGroup	Does the actual software configuration on a given server by calling the <code>onConfigureServer</code> public event handler. Before that reports the beginning of the configuration and emits the notification. If the configuration completes successfully calls the <code>getConfigurationKey</code> method and sets the <code>configuration</code> attribute of the server to resulting value thus saving the configuration applied to a given server. Returns <i>null</i> to indicate successful configuration. If configuration is not successful returns <i>false</i> .

The `OpenStackSecurityConfigurable` class extends `Configurable` by implementing the `configureSecurity` method of the base class and adding the empty `getSecurityRules` method.

Method	Arguments	Description
<code>getSecurityRules</code>	None	Returns an empty dictionary in default implementation. Inheritors which want to add security rules during the app configuration should implement this method and make it return a list of dictionaries describing the security rules with the following keys: <ul style="list-style-type: none"> • <code>FromPort</code> (port number, e.g. 80). • <code>ToPort</code> (port number, e.g. 80). • <code>IpProtocol</code>: (string, e.g. 'tcp'). • <code>External</code>: (boolean: <i>true</i> means that the inbound traffic to the given port (or port range) may originate from outside of the environment; <i>false</i> means that only the VMs spawned by this or other apps of the current environment may connect to this port). • <code>Ethertype</code>: (optional, can be 'IPv4' or 'IPv6').
<code>configureSecurity</code>	<code>servers,</code> <code>serverGroup</code>	Gets the list of security rules provided by the <code>getSecurityRules</code> method and adds security group with these rules to the Heat stacks of all regions which the component's servers are deployed to

Consider the following example of this class usage:

```

Namespaces:
  ==: com.example.apache
  apps: io.murano.applications

Name: ApacheHttpServer

Extends:
  - apps:MultiServerApplicationWithScaling
  - apps:OpenStackSecurityConfigurable

Methods:
  getSecurityRules:
    Body:
      - Return:
          - ToPort: 80
            FromPort: 80
            IpProtocol: tcp
            External: true
          - ToPort: 443
            FromPort: 443
            IpProtocol: tcp
            External: true
  
```

In the example above, the `ApacheHttpServer` class is configured to create a security group with two security rules allowing network traffic over HTTP and HTTPS protocols on its deployment.

The `SoftwareComponent` class inherits both `Installable` and `Configurable` and adds several addi-

tional methods.

Method	Arguments	Description
deployAt	serverGroup	Binds all workflows into one process. Consequentially calls <code>deploy</code> method of the <code>serverGroup</code> , <code>install</code> and <code>configure</code> methods inherited from the parent classes.
report	message	Reports a message using environment's reporter.
detectSuccess	allowedFailures, serverGroup, failedServer	Static method that returns <code>true</code> in case the actual number of failures (number of <code>failedServers</code>) is less than or equal to the <code>allowedFailures</code> . The latter can be one of the following options: <code>none</code> , <code>one</code> , <code>two</code> , <code>three</code> , <code>any</code> , 'quorum'. <code>any</code> allows any number of failures during the installation or configuration. <code>quorum</code> allows failure of less than a half of all servers.

Event notification pattern

The `Event` class may be used to issue various notifications to other MuranoPL classes in an event-driven manner.

Any object which is going to emit the notifications should declare the instances of the `Event` class as its public Runtime properties. You can see the examples of such properties in the `Installable` and `Configurable` classes:

Name: `Installable`

Properties:

beforeInstallEvent:

Contract: `$.class(Event).NotNull()`

Usage: Runtime

Default:

name: `beforeInstall`

The object which is going to subscribe for the notifications should pass itself into the `subscribe` method of the event along with the name of its method which will be used to handle the notification:

```
$event.subscribe($subscriber, handleFoo)
```

The specified handler method must be present in the subscriber class (if the method name is missing it will default to `handle%Eventname%`) and have at least one standard (i.e. not `VarArgs` or `KwArgs`) argument which will be treated as `sender` while invoking.

The `unsubscribe` method does the opposite and removes object from the subscribers of the event.

The class which is going to emit the notification should call the `notify` method of the event and pass itself as the first argument (`sender`). All the optional parameters of the event may be passed as `varargs`/`kwargs` of the `notify` call. They will be passed all the way to the handler methods.

This is how it looks in the `Installable` class:

beforeInstall:

Arguments:

(continues on next page)

(continued from previous page)

```

- servers:
  Contract:
    - $.class(res:Instance).NotNull()
- serverGroup:
  Contract: $.class(ServerGroup).NotNull()
Body:
- ...
- $this.beforeInstallEvent.notify($this, $servers, $serverGroup)
- ...

```

The `notifyInParallel` method does the same, but invokes all handlers of subscribers in parallel.

Base application classes

There are several base classes that extend standard `io.murano.Application` class and `SoftwareComponent` class from the application development library.

SingleServerApplication

A base class for applications running a single software component on a single server only. Its `deploy` method simply creates the `SingleServerGroup` with the `server` provided as an application input.

MultiServerApplication

A base class for applications running a single software component on multiple servers. Unlike `SingleServerApplication`, it has the `servers` input property instead of `server`. It accepts instance of one of the inheritors of the `ServerGroup` class.

MultiServerApplicationWithScaling

Extends `MultiServerApplication` with the ability to scale the application by increasing (scaling out) or decreasing (scaling in) the number of nodes with the application after it is installed. The differences from `MultiServerApplication` are:

- the `servers` property accepts only instances of `ServerReplicationGroup` rather than any `ServerGroup`
- the additional optional `scaleFactor` property accepts the number by which the app is scaled at once; it defaults to 1
- the `scaleOut` and `scaleIn` public methods are added

Application developers may as well define their own classes using the same approach and combining base classes behaviour with the custom code to satisfy the needs of their applications.

Application developer's cookbook

If you have not written murano packages before, start from the existing *Step-by-Step* guide. It contains general information about murano packages development process. Additionally, see the *MuranoPL reference*.

Load applications from a local directory

Normally, whenever you make changes to your application, you have to package it, re-upload the package to the API, and delete the old package from the API. This makes developing and testing murano applications troublesome and time-consuming. Murano-engine provides a way to speed up the edit-upload-deploy loop. This can be done with the `load_packages_from` option. Murano-engine examines any directories mentioned in this option before accessing the API. Therefore, you do not even need to package the application into a ZIP archive and any changes you make are instantly available to the engine, if you do not plan to check or change the application UI. To check your application's appearance in the OpenStack dashboard, upload the application for the first run. Additionally, re-upload the package using the OpenStack dashboard or CLI each time you update the application UI.

To load an application from a local directory, modify the `load_packages_from` parameter in murano config `[engine]` section.

```
[engine]
...
load_packages_from = /path/to/murano/applications
...
```

Note: The murano-engine scans the directory structure and seeks application manifests. Therefore, you can point the `load_packages_from` parameter to a cloned version of the murano-apps repository.

Deploy environment using CLI

The standard way to deploy an application in murano is by using the murano dashboard (OpenStack dashboard plug-in). However, if the OpenStack dashboard is not available or some sort of automation is required, murano provides the capability to deploy environments through CLI. It is a powerful tool that allows users and application developers make arbitrary changes to apps object-model. This can be useful in early stages of application development to experiment with different object models of an application. You can read more about it in *Deploying environments using CLI*

Application unit test framework

An application unit test framework was created to make development process easier. With this framework you can check different scenarios of application deployment without running real deployments.

For more information about application unit tests, see *Application unit tests*.

Garbage collection system in MuranoPL

A garbage collection system (GC) manages the deallocation of resources in murano. The garbage collection system implementation is based on the execution of special `.destroy()` methods that you may define in MuranoPL classes. These methods contain logic to deallocate any resources that were allocated by MuranoPL objects. During deployment all objects that are not referenced by any other object and that are not present in the object model anymore is deleted by GC.

- The `.destroy()` methods are executed for each class in the class hierarchy of the object that has this method. Child classes cannot prevent parent classes `.destroy` from being called and cannot call base classes implementation manually
- `.destroy()` methods for class hierarchy are called in reversed order from that of `.init()` - starting from the actual object type and up to the `io.murano.Object` class
- If object *Bar* is owned (directly or indirectly) by object *Foo* then *Bar* is going to be destroyed before *Foo*. There is a way for *Foo* to get notified on *Bar*'s destruction so that it can prepare for it. See below for details.
- For objects that are not related to each other the destruction order is undefined. However objects may establish destruction dependency between them to establish the order.
- Unrelated objects might be destroyed in different green threads.
- Any exceptions thrown in the `.destroy()` methods are muted (but still logged).

Destruction dependencies may be used to notify *Foo* of *Bar*'s destruction even if *Bar* is not owned by *Foo*. If you subscribe *Foo* to *Bar*'s destruction, the following will happen:

- *Foo* will be notified when *Bar* is about to be destroyed.
- If both *Foo* and *Bar* are going to be destroyed in the same garbage collection execution, *Bar* will be destroyed before *Foo*.

Garbage collector methods

Murano garbage collector class (`io.murano.system.GC`) has the following methods:

collect()

Initiates garbage collection of unreferenced objects of current deployment. Usually, it is called by murano `ObjectStore` object during deployment. However, it can be called from MuranoPL code like `io.murano.system.GC.collect()`.

isDestroyed(object)

Checks if the object was already destroyed during a GC session and thus its methods cannot be called.

isDoomed(object)

Can be used within the `.destroy()` method to check if another object is also going to be destroyed.

subscribeDestruction(publisher, subscriber, handler=null)

Establishes a destruction dependency from the `subscriber` to the object passed as `publisher`. This method may be called several times with the same arguments. In this case, only a single destruction dependency will be established. However, the same amount of calls of `unsubscribeDestruction` will be required to remove it.

The `handler` argument is optional. If passed, it should be the name of an instance method defined by the caller class to handle the notification of `publisher` destruction. The following argument will be passed to the handler method:

object

A target object that is going to be destroyed. It is not recommended persisting the reference to this object anywhere. This will not prevent the object from being garbage collected but the object will be moved to the "destroyed" state. This is an advanced feature that should not be used unless it is absolutely necessary.

unsubscribeDestruction(publisher, subscriber, handler=null)

Removes the destruction dependency from the `subscriber` to the object passed as `publisher`. The method may be called several times with the same arguments without any side effects. If `subscribeDestruction` was called more than once, the same (or more) amount of calls to `unsubscribeDestruction` is needed to remove the dependency.

The `handler` argument is optional and must correspond to the handler passed during subscription if it was provided.

Using destruction dependencies

To use direct destruction dependencies in your murano applications, use the methods from MuranoPL `io.murano.system.GC`. To establish a destruction dependency, call the `io.murano.system.GC.subscribeDestruction` method in you application code:

```
.init:
  Body:
    - If: $.publisher
      Then:
        - sys:GC.subscribeDestruction($.publisher, $this, ↵
↵onPublisherDestruction)
```

In the example above, `onPublisherDestruction` is a *Foo* object method that will be called when *Bar* is destroyed. If you do not want to do something specific with the destroyed object omit the third parameter. The destruction dependencies will be persisted between deployments and deserialized from the objects model to murano object.

Managing Sensitive Data in Murano

Overview

If you are developing a Murano application that manages sensitive data such as passwords, user data, etc, you may want to ensure this is stored in a secure manner in the Murano backend.

Murano offers two *yaql* functions to do this, *encryptData* and *decryptData*.

Note: Barbican or a similar compatible secret storage backend must be configured to use this feature.

Configuring

Murano makes use of [Castellan](#) to manage encryption using a supported secret storage backend. As of OpenStack Pike, [Barbican](#) is the only supported backend, and hence is the one tested by the Murano community.

To configure Murano to use Barbican, place the following configuration into *murano-engine.conf*:

```
[key_manager]
auth_type = keystone_password
auth_url = <keystone_url>
username = <username>
password = <password>
user_domain_name = <domain_name>
```

Similarly, place the following configuration into *_50_murano.py* to configure the murano-dashboard end:

```
KEY_MANAGER = {
    'auth_url': '<keystone_url>/v3',
    'username': '<username>',
    'user_domain_name': '<domain_name>',
    'password': '<password>',
    'project_name': '<project_name>',
    'project_domain_name': '<domain_name>'
}
```

Note: Horizon config must be valid Python, so the quotes above are important.

Example

encryptData(foo): Call to encrypt string *foo* in storage. Will return a *uuid* which is used to retrieve the encrypted value.

decryptData(foo_key): Call to decrypt and retrieve the value represented by *foo_key* from storage.

There is an example application available in the murano [repository](#).

6.1.15 Installing Murano API via WSGI

This document is a guide to deploy murano using two WSGI mode uwsgi and mod_wsgi of Apache.

Please note that if you intend to use mode uwsgi, you should install `mode_proxy_uwsgi` module. For example on deb-base system:

```
# sudo apt-get install libapache2-mod-proxy-uwsgi
# sudo a2enmod proxy
# sudo a2enmod proxy_uwsgi
```

WSGI Application

The function `murano.httppd.init_application` will setup a WSGI application to run behind `uwsgi` and `mod_wsgi`

Murano API behind uwsgi

Create a `murano-api-uwsgi` file with content below:

```
[uwsgi]
chmod-socket = 666
socket = /var/run/uwsgi/murano-wsgi-api.socket
lazy-apps = true
add-header = Connection: close
buffer-size = 65535
hook-master-start = unix_signal:15 gracefully_kill_them_all
thunder-lock = true
plugins = python
enable-threads = true
worker-reload-mercy = 90
exit-on-reload = false
die-on-term = true
master = true
processes = 2
wsgi-file = <path-to-murano-bin-dir>/murano-wsgi-api
```

Start `murano-api`:

```
# uwsgi --ini /etc/murano/murano-api-uwsgi.ini
```

Murano API behind mod_wsgi

Create `/etc/apache2/murano.conf` with content below:

```
Listen 8082

<VirtualHost *:8082>
    WSGIDaemonProcess murano-api processes=1 threads=10 user=%USER% display-
    ↪name=%{GROUP} %VIRTUALENV%
    WSGIProcessGroup murano-api
    WSGIScriptAlias / %MURANO_BIN_DIR%/murano-wsgi-api
    WSGIApplicationGroup %{GLOBAL}
    WSGIPassAuthorization On
    AllowEncodedSlashes On
    <IfVersion >= 2.4>
        ErrorLogFormat "%{cu}t %M"
    </IfVersion>
    ErrorLog /var/log/%APACHE_NAME%/murano_api.log
    CustomLog /var/log/%APACHE_NAME%/murano_api_access.log combined
```

(continues on next page)

(continued from previous page)

```
<Directory %MURANO_BIN_DIR%>
  <IfVersion >= 2.4>
    Require all granted
  </IfVersion>
  <IfVersion < 2.4>
    Order allow,deny
    Allow from all
  </IfVersion>
</Directory>
</VirtualHost>
```

Then on deb-based systems copy or symlink the file to `/etc/apache2/sites-available`. For rpm-based systems the file will go in `/etc/httpd/conf.d`.

Enable the murano site. On deb-based systems:

```
# a2ensite murano
# systemctl reload apache2.service
```

On rpm-based systems:

```
# systemctl reload httpd.service
```


FIRST APP GUIDE

A guide for developing your first Murano application.

7.1 My first Murano App getting started guide

This directory contains the "My first Murano App getting started guide" tutorial.

The tutorials work with an application that can be found in the [openstack/murano-apps](#) repository.

7.1.1 Prerequisites

To build the documentation, you must install the Graphviz package.

/source

The `/source` directory contains the tutorial documentation as `reStructuredText` (RST).

To build the documentation, you must install `Sphinx` and the `OpenStack docs.openstack.org Sphinx theme` (`openstackdocstheme`). When you invoke `tox`, these dependencies are automatically pulled in from the top-level `test-requirements.txt`.

You must also install `Graphviz` on your build system.

The document is build as part of the docs build, for example using:

```
tox -e docs
```

/samples

The code samples in this guide are located in this directory.

Contents

Who is this guide for

What is the use case

What you will learn

Before the start

What you need

Deploy Murano

Develop Murano app for Plone

Develop standalone Plone Murano app (single VM)

Plone server requirements

Define host VM requirements

Host VM operating system requirements

Host VM hardware resources requirements

Define preinstalled software and libraries requirements

Define what the PloneServerApp should do

Create and debug sh-script that fully deploys the Plone server on a single VM

Create Murano package for your app

Upload and deploy your Murano app to OpenStack cloud

Develop cluster Plone Murano app (multi VM)

Develop basic server-client Murano app

Add load-balancing to the Plone cluster

Add scalability to the Plone cluster

Add self-healing to the Plone cluster

Debugging and troubleshooting your Murano app

Publish your Murano app in the application catalog

Join the OpenStack community

Prepare testing environment

Contribute your code to Murano-apps

Contribute your code to App-catalog

APPLICATION DEVELOPER DOCUMENTATION

Learn how to compose an application package and get it ready for uploading to Murano.

8.1 FAQ

There are too many files in Murano package, why not to use a single Heat Template?

To install a simple Apache service to a new VM, Heat Template is definitely simpler. But the Apache service is useless without its applications running under it. Thus, a new Heat Template is necessary for every application that you want to run with Apache. In Murano, you can compose a result software to install it on a VM on-the-fly: it is possible to select an application that can run under Apache dynamically. Or you can set a VM where Apache is installed as a parameter. This way, the files in the application package allow to compose compound applications with multiple configuration options. For any single combination you need a separate Heat Template.

The Application section is defined in the UI form. Can I remove it?

No. The `Application` section is a template for Murano object model which is the instruction that helps you to understand the environment structure that you deploy. While filling the forms that are auto-generated from the `UI.yaml` file, object model is updated with the values entered by the user. Eventually, the Murano engine receives the resulted object model (`.json` file) after the environment is sent to the deploy.

The Templates section is defined in the UI form. What's the purpose?

Sometimes, the user needs to create several instances with the same configuration. A template defined by a variable in the `Templates` section is multiplied by the value of the number of instances that are set by the user. A YAQL `repeat` function is used for this operation.

Some properties have Usage, others do not. What does this affect?

`Usage` indicates how a particular property is used. The default value is `In`, so sometimes it is omitted. The `Out` property indicates that it is not set from outside, but is calculated in the class methods and is available for the `read` operation from other classes. If you don't want to initialize in the class constructor, and the property has no default value, you specify `Out` in the `Usage`.

Can I use multiple inheritance in my classes?

Yes. You can specify a list of parent classes instead of a single string in the regular YAML notation. The list with one element is also acceptable.

There are FullName and Name properties in the manifest file. What's the difference between them?

Name is displayed in the web UI catalog, and `FullName` is a system name used by the engine to get the class definition and resolve the class interconnections.

How does Murano know which class is the main one?

There is no main class term in the MuranoPL. Everything depends on a particular object model and an instance class representing the instance. Usually, an entry-point class has exactly the same name as the package `FullName`, and it uses other classes.

What is the difference between `$variable` and `$.variable` in the class definitions?

By default, `$` represents a current object (similar to `self` in Python or `this` in C++/Java/C#), so `$.variable` accesses the object field/property. In contrast, `$variable` (without a dot) means a local method variable. Note that `$` can change its value during execution of some YAQL functions like `select`, where it means a current value. A more safe form is to use a reserved variable `$this` instead of `$`. `$this.variable` always refers to an object-level value in any context.

CONTRIBUTOR DOCUMENTATION

- If you are a new contributor to Murano please refer: *So You Want to Contribute...*

9.1 So You Want to Contribute...

For general information on contributing to OpenStack, please check out the [contributor guide](#) to get started. It covers all the basics that are common to all OpenStack projects: the accounts you need, the basics of interacting with our Gerrit review system, how we communicate as a community, etc. Below will cover the more project specific information you need to get started with Murano.

9.1.1 Communication

- IRC channel #murano at OFTC
- Mailing list (prefix subjects with [murano] for faster responses) <http://lists.openstack.org/cgi-bin/mailman/listinfo/openstack-discuss>

9.1.2 Contacting the Core Team

Please refer the [murano Core Team](#) contacts.

9.1.3 New Feature Planning

murano features are tracked on [Launchpad](#).

9.1.4 Task Tracking

We track our tasks in [Launchpad](#). If you're looking for some smaller, easier work item to pick up and get started on, search for the 'low-hanging-fruit' tag.

9.1.5 Reporting a Bug

You found an issue and want to make sure we are aware of it? You can do so on [Launchpad](#).

9.1.6 Getting Your Patch Merged

All changes proposed to the murano project require one or two +2 votes from murano core reviewers before one of the core reviewers can approve patch by giving Workflow +1 vote.

9.1.7 Project Team Lead Duties

All common PTL duties are enumerated in the [PTL guide](#).

9.2 Contributor Guide

9.2.1 How to contribute

9.2.2 Development guidelines

Conventions

High-level overview of Murano components

Coding guidelines

There are several significant rules for the Murano developer:

- Follow PEP8 and OpenStack style guidelines.
- Do not import functions. Only module imports are accepted.
- Make commits as small as possible. It speeds up review of the change.
- Six library usage rule: use it only when really necessary (for example if existing code will not work in python 3 at all).
- Mark application name in the 1st line of commit message for murano-apps repository, i.e. [Apache] or [Kubernetes].
- Prefer code readability over performance unless the situations when performance penalty can be proven to be big.
- Write Py3-compatible code. If that's impossible leave comment.

Rules for MuranoPL coding style:

- Use camelCase for MuranoPL functions/namespaces/variables/properties, PascalCase for class names.
- Consider using `$this` instead of `$` where appropriate.

Debug tips

9.2.3 Murano plug-ins

Murano plug-ins help to extend the capability of murano. There are two types of murano plug-ins which serve different purposes:

- Extend murano Core Library by implementing additional functionality.
- Add new package type classes.

This section contains the following topics:

MuranoPL extension plug-ins

Murano plug-ins allow extending MuranoPL with new classes. Therefore, using such plug-ins applications with MuranoPL format, you access some additional functionality defined in a plug-in. For example, the Magnum plug-in, which allows murano to deploy applications such as Kubernetes using the capabilities of the Magnum client.

MuranoPL extension plug-ins can be used for the following purposes:

- Providing interaction with external services.

For example, you want to interact with the OpenStack Image service to get information about images suitable for deployment. A plug-in may request image data from glance during deployment, performing any necessary checks.

- Enabling connections between murano applications and external hardware

For example, you have an external load balancer located on a powerful hardware and you want your applications launched in OpenStack to use that load balancer. You can write a plug-in that interacts with the load balancer API. Once done, add new apps to the pool of your load balancer or make any other configurations from within your application definition.

- Extending Core Library class functionality, which is responsible for creating networks, interaction with murano-agent, and others

For example, you want to create networks with special parameters for all of your applications. You can just copy the class that is responsible for network management from the Murano Core library, make the desired modification, and load the new class as a plug-in. Both classes will be available, and it is up to you to decide which way to create your networks.

- Optimization of frequently used operations. Plug-in classes are written in Python, therefore, the opportunity for improvement is significant.

Murano provides a number of optimization opportunities depending on the improvement needs. For example, classes in the Murano Core Library can be rewritten in C and used from Python code to improve their performance in particular use cases.

MuranoPL package type plug-ins

The only package type natively supported by Murano is MuranoPL. However, it is possible to extend Murano with support for other formats of application definitions. TOSCA CSARs and HOT templates are the two examples of alternate ways to define applications.

Package types plug-ins are normal Python packages that can be distributed through PyPI and installed using **pip** or its alternatives. It is important that the plug-in be installed to the same Python instance that is used to run Murano API and Murano Engine. For multi-node Murano deployments, plug-ins need to be installed on each node.

To associate a plug-in with a particular package format, it needs to have a special record in `[entry_points]` section of `setup.cfg` file:

```
io.murano.plugins.packages =
    Name/Version = namespace:Class
```

For example:

```
[entry_points]
io.murano.plugins.packages =
    Cloudify.TOSCA/1.0 = murano_cloudify_plugin.cloudify_tosca_
↪package:CloudifyToscaPackage
```

This declaration maps particular pair of format-name/version to Python class that implements Package API interface for the package type. It is possible to specify several different format names or versions and map them to single or different Python classes. For example, it is possible to specify

```
[entry_points]
io.murano.plugins.packages =
    Cloudify.TOSCA/1.0 = murano_cloudify_plugin.cloudify_tosca_
↪package:CloudifyToscaPackage
    Cloudify.TOSCA/1.1 = murano_cloudify_plugin.cloudify_tosca_
↪package:CloudifyToscaPackage
    Cloudify.TOSCA/2.0 = murano_cloudify_plugin.cloudify_tosca_
↪package:CloudifyToscaPackage_v2
```

Note: A single Python plug-in package may contain several Murano plug-ins including of different types. For example, it is possible to combine MuranoPL extension and package type plug-ins into a single package.

Tooling for package preparation

Some package formats may require additional tooling to prepare package ZIP archive of desired structure. In such cases it is expected that those tools will be provided by plug-in authors either as part of the same Python package (by exposing additional shell entry points) or as a separate package or distribution.

The only two exceptions to this rule are native MuranoPL packages and HOT packages that are built into Murano (there is no need to install additional plug-ins for them). Tooling for those two formats is a part of `python-muranoclient`.

Package API interface reference

Plug-ins expose API for the rest of Murano to interact with the package by implementing *murano.packages.package.Package* interface.

Class initializer:

```
def __init__(self, format_name, runtime_version, source_directory, manifest):
```

- **format_name**: name part of the format identifier (string)
- **runtime_version**: version part of the format identifier (instance of *semantic_version.Version*)
- **source_directory**: path to the directory where package content was extracted (string)
- **manifest**: contents of the manifest file (string->string dictionary)

Note: implementations must call base class (*Package*) initializer passing the first three of these arguments.

Abstract properties that must be implemented by the plug-in:

```
def full_name(self):
```

- Fully qualified name of the package. Must be unique within package scope of visibility (string)

```
def version(self):
```

- Package version (not to confuse with format version!). An instance of *semantic_version.Version*

```
def classes(self):
```

- List (or tuple) of MuranoPL class names (FQNs) that package contains

```
def requirements(self):
```

- Dictionary of requirements (dependencies on other packages) in a form of key-value mapping from required package FQN string to SemVer version range specifier (instance of *semantic_version.Spec* or string representation supported by Murano versioning scheme)

```
def package_type(self):
```

- Package type: "Application" or "Library"

```
def display_name(self):
```

- Human-readable name of the package as presented to the user (string)

```
def description(self):
```

- Package description (string or None)

```
def author(self):
```

- Package author (string or None)

```
def supplier(self):
```

- Package supplier (string or None)

def tags(self):

- List or tags for the package (list of strings)

def logo(self):

- Package (application) logo file content (str or None)

def supplier_logo(self):

- Package (application) supplier logo file content (str or None)

def ui(self):

- YAML-encoded string containing application's form definition (string or None)

Abstract methods that must be implemented by the plug-in:

def get_class(self, name):

- Returns string containing MuranoPL code (YAML-encoded string) for the class whose fully qualified name is in "name" parameter (string)

def get_resource(self, name):

- Returns path for resource file whose name is in "name" parameter (string)

Properties that can be overridden in the plug-in:

def format_name(self):

- Canonical format name for the plug-in. Usually the same value that was passed to class initializer

def runtime_version(self):

- Format version. Usually the same value that was passed to class initializer (semantic_version.Version)

def blob(self):

- Package file (.zip) content (str)

PackageBase class

Usually, there is no need to manually implement all the methods and properties described. There is a *murano.packages.package.PackageBase* class that provides typical implementation of most of required properties by obtaining corresponding value from manifest file.

When inheriting from PackageBase class, plug-in remains responsible for implementation of:

- *ui* property
- *classes* property
- *get_class* method

This allows plug-in developers to concentrate on dynamic aspects of the package type plug-in while keeping all static aspects (descriptions, logos and so on) consistent across all package types (at least those who inherit from *PackageBase*).

Creating a Murano plug-in

Murano plug-in is a setuptools-compliant python package with `setup.py` and all other necessary files. For more information about defining stevedore plug-ins, see [stevedore documentation](#).

The structure of the demo application package

The package must meet the following requirements:

- It must be a ZIP archive.
- The root folder of the archive must contain a `manifest.yaml` file.
- The manifest must be a valid YAML file representing key-value associative array.
- The manifest should contain a `Format` key, that is, a format identifier. If it is not present, "MuranoPL/1.0" is used.

Murano uses the `Format` attribute of the manifest file to find an appropriate plug-in for a particular package type. All interactions between the rest of Murano and package file contents are done through the plug-in interface alone.

Because Murano never directly accesses files inside the packages, it is possible for plug-ins to dynamically generate MuranoPL classes on the fly. Those classes will be served as adapters between Murano and third-party systems responsible for deployment of particular package types. Thus, for Murano all packages remain to be of MuranoPL type though some of them are "virtual".

The format identifier has the following format: `Name/Version`. For example, `Heat.HOT/1.0`. If name is not present, it is assumed to be MuranoPL (thus `1.0` becomes `MuranoPL/1.0`). Version strings are in SemVer three-component format (major.minor.patch). Missing version components are assumed to be zero (thus `1.0` becomes `1.0.0`).

Installing a plug-in

To use a plug-in, install it on murano nodes in the same Python environment with murano engine service.

To install a plug-in:

1. Execute the plug-in setup script.

Alternatively, use a package deployment tool, such as pip:

```
cd plugin_dir
pip install .
```

2. Restart murano engine. After that, it will be possible to upload and deploy the applications that use the capabilities that a plug-in provides.

Plug-in versioning

Plug-ins located in Murano repository have the same version as Murano. Therefore, to use a specific version of such plug-in, checkout to this version. Then specify the version of plug-in classes in your application's manifest file as usual:

```
Require:
  murano.plugins.example: 2.0.0
```

It should be standard SemVer format version string consisting of three parts: Major.Minor.Patch. For more information about versioning, refer to *Versioning*.

Note: Enable Glare to use versioning.

Organization

Documentation

Documentation helps users understand what your plug-in does. For plug-ins located in the Murano repository, create a `README.rst` file in the main folder of the plug-in. The `README.rst` file may contain information about the plug-in and an installation guide.

Code

The code of your plug-in may be located in the following repositories:

- Murano repository. In this case, the plug-in should be located in the `murano/contrib/plugins` folder.
- A separate repository. In this case, create your own project.

Bugs

All bugs for specific plug-ins are reported in their projects. Bugs related to plug-ins located in Murano repository should be reported in the [Murano](#) project.

9.2.4 Development environment

9.2.5 Testing

Testing guidelines

Continuous Integration service

UI testing

Tempest tests

Automated testing machinery

CI design

CI jobs

9.2.6 Documentation guidelines

9.2.7 Backporting to stable/branches

Since murano is a big-tent OS project it largely follows the [OpenStack stable branch guide](#)

Upstream support phases

1. Phase I (first 6 months): All bugfixes (which meet the stable port criteria, described in OS stable branch policy) are appropriate
2. Phase II (6-12 months): Only critical bugfixes and security patches are acceptable
3. Phase III (more than 12 months): Only security patches are acceptable

In order to accept a change into \$release it must first be accepted into all releases back to master.

There are two notable exceptions to the support phases rule:

- murano-apps repository: We recognise, that murano apps have different lifecycle than main murano repository. Most of the time new apps are being written for already released versions of murano, not for master. Having a rich collection of apps is one of the goals of murano-apps repository, therefore we accept backports of apps and app features to previous release branches. This is done on a case by case basis and should be discussed with PTL and Murano core members on IRC or Mailing List. However we believe, that submitting an app to stable branch only means that author of the patch is not going to support the app. Therefore for the app to get backported it still has to be first accepted to master and all subsequent releases.
- murano core library patches: Murano Core Library is an app, that provides core functionality and classes for other murano apps. It shares a lot of properties of regular murano apps and the rationale behind allowing backports of MuranoPL code from master to stable branches is basically the same: low regression risks during upgrades, high adoption impact. However since core library is much more sensitive app, backports to it should be taken more seriously and should be discussed on IRC and Mailing List and receive PTL's approval.

These two exceptions do not mean, that we're free to backport any code from master to stable branches. Instead they show, that murano team recognises the importance of these two areas of murano project and treats exceptions to those slightly more liberally than to other parts of murano project.

Bug nomination process

Whenever you file a bug, or see a bug, that you think is eligible for backporting in stable branch nominate it for the corresponding series. If bug reporter does not nominate the bug for eligible branch this is done by murano bug supervisor during triaging/confirmation process. In case it is not clear whether the bug is eligible or not or if you do not have permissions to nominate a bug for series you can set *\$release-backport-potential* tag (for example *liberty-backport-potential*). Murano team is holding bi-weekly meetings on IRC (as part of regular community meetings) to triage and nominate bugs for stable backports.

OTHER DOCUMENTATION