DTrove Documentation

Release 0.1

Robert Myers

July 30, 2014

Contents

Fork of openstack trove written in Django

The main difference is that the this project does not use a guest agent on the host machines. Instead all the commands to manage the instances are done either thru ssh or the console on the compute hosts.

If you like the project fork it on github.

CHAPTER 1

Quickstart Guide

The main way to run dtrove is in single superuser. Basically using the same account to create all database clusters. The plan is to make the api honor the same access controls as the *Cloud Providers* you have chosen.

1.1 Install

You have to go bleeding edge on a project like this, so break out your trusty *git* tool and prepare to get dirty

1. Clone this repository:

```
$ git clone https://github.com/rmyers/dtrove.git
$ cd dtrove
$ mkvirtualenv dtrove -r requirements.txt
$ pip install -e .
```

2. Now you are ready to add this to your django settings file:

```
INSTALLED_APPS = (
    ...
    'dtrove',
    'rest_framework',
)
```

3. For the default Openstack Provider set the following:

```
OS_USERNAME=your_nova_user.
OS_PASSWORD=your_password
OS_PROJECT_ID=your_project_id
OS_NOVA_URL=http://localhost:5000
```

> **Note:** See the *Configuration Options* for details about all the options available.

4. First run the migrations:

```
$ python manage.py migrate
```

1.2 Creating Clusters

First

CHAPTER 2

Further Reading

See the following sections for more details on the setup of dtrove.

2.1 Datastores

2.1.1 DTrove Datastore Base Classes

The base classes that define the interface for all datastores.

If you wish to create your own datastore, simply subclass the base and override the methods you need to for example:

```
from dtrove.datastores import base
from dtrove.commands import run

class MyBackyardDBManager(base.BaseManager):

    def backup():
        with self.conn as connection:
            run('/bin/bash my_backup.sh')

    @property
    def name():
        return 'ClassNameOverriddenDB'
```

Templates for the managers live in *dtrove/datastores/<manager_name>* where manager name by default is the lowercase manager name with the 'Manager' removed. For example 'MySQLManager' would be 'mysql'.

More on templates later.

class dtrove.datastores.base.**BaseManager**(*datastore*)

Manager Base

Parameters **datastore** – The actual datastore version that is being managed. The datastores have the image and package information to install on the guest vm's.

backup(*instance*)

Preform a backup on the remote instance.

name

Returns the name of the manager ex: MySQLManager = mysql

prepare(*instance*)

Install and configure the datastore on the instance.

render_config_file(*instance*)
Load and render a config file for this datastore.

restart()
Restart the datastore.

service_name = None
Name of the service ie 'mysql'

start()
Start the datastore

stop()
Stop the datastore

2.1.2 Implementations

This is a list of the current built in Managers

- dtrove.datastores.mysql.MySQLManager
- dtrove.datastores.redis.RedisManager
- dtrove.datastores.pgsql.PostgresManager

2.2 Cloud Providers

This is the interface that the provisioning system uses to create new nodes/volumes/backups for the clusters.

These classes are meant to be used in the task system as they are usually long running operations. During the execution of a long running process the status should be updated on the base objects, usually a dtrove.models.Instance.

Here is a example:

```
from celery import shared_task
from dtrove.models import Instance
from dtrove.provider import get_provider

provider = get_provider()

@shared_task
def create(instance_id):
    instance = Instance.objects.get(pk=instance_id)
    provider.create(instance)
```

class dtrove.providers.base.**BaseProvider**
Base Provider Interface

attach_volume(*instance*)
Creates and mounts a volume on an instance

Parameters **instance** (dtrove.models.Instance) – An instance object to delete

Raises dtrove.providers.base.ProviderError If the volume create and mount failed

The provider should update instance with the following info:

•*volume*: Reference to the actual volume id on the provider

create (*instance*, ***kwargs*)
Creates a new instance

Parameters **instance** (dtrove.models.Instance) – An instance object to create

Raises dtrove.providers.base.ProviderError If the create failed

Typically the provider should respond with a 202 message and work in the background to create the instance. So during creation of the instance on the provider this should update the progress field of the instance.

The provider should update instance with the following info:

•*server*: Reference to the actual server id on the provider

•*addr*: The public facing ip address to this server

•*user*: The inital user that was created

The task, status and progress are properties that are cached on the dtrove.models.Instance Instance object.

destroy (*instance*)
Destroys an instance

Parameters **instance** (dtrove.models.Instance) – An instance object to delete

Raises dtrove.providers.base.ProviderError If the delete failed

This should delete the instance and poll until the instance is deleted.

flavors (*datastore=None*)
Return a list of flavors available

Parameters **datastore** (*str*) – (optional) datastore to filter flavors by

snapshot (*instance*)
Creates a snapshot of an instance

Parameters **instance** (dtrove.models.Instance) – An instance object to delete

Raises dtrove.providers.base.ProviderError If the snapshot failed

supports_snapshots = True
Whether of not this provider supports snapshots

supports_volumes = True
Whether or not this provider supports creating and attaching volumes

update_status (*instance*)
Updates the status of an instance

Parameters **instance** (dtrove.models.Instance) – An instance object to create

Raises dtrove.providers.base.ProviderError If the status failed

Returns tuple of (status, progress)

A call to this method should set the properties on the instance. For example here:

```
def update_status(self, instance):
    obj = self.get(instance.id)

    instance.message = obj.error_message
    instance.server_status = obj.status
    instance.progress = obj.progress
```

```
        return instance.server_status, instance.progress
```

- *status* property should be a string of the current status
- *progress* property should be an int which is the percent complete

exception dtrove.providers.base.**ProviderError**
An exception during a provisoner action.

dtrove.providers.base.**get_provider**()
Return the current provider class

2.3 Tasks

These handle all the remote operations that happen. Such as building a new cluster, or run a scheduled backup. We are using celery as our worker daemon, giving us a powerful system to schedule or chain tasks together.

2.3.1 Execution Tasks

We can use the *perform* task to run any command on the remote systems.

dtrove.tasks.**perform**(*instance_id*, *name*, **cmds*)
Execute a remote task

Parameters

- **instance_id** (*int*) – ID of the instance to connect to
- **name** (*str*) – Identifier of this task
- **cmds** (*str*) – The actual commands to run

For example (To ruin someones day):

```
perform.delay(1, 'destory', 'rm -rf /', 'echo everything gone')

Will produce
[root@10.0.0.1] run: rm -rf /
out:
[root@10.0.0.1] run: echo everything gone
out: everything gone
Disconnecting from 127.0.0.1...
done.
```

2.3.2 Build Tasks

These tasks handle creating and managing the servers themselves.

dtrove.tasks.**create_server**(*instance_id*, *volume_id=None*)
Create a nova server instance

Parameters

- **instance_id** (*int*) – ID of the instance to build
- **volume_id** (*str*) – Optional volume id to attach

dtrove.tasks.**create_volume**(*instance_id*)
Create a volume for the instance

Parameters **instance_id** (*int*) – ID of the instance

dtrove.tasks.**attach_volume**(*instance_id*, *volume_id*)
Attach a volume to a nova server instance

Parameters

- **instance_id** (*int*) – ID of the instance to build
- **volume_id** (*str*) – Optional volume id to attach

2.4 DTrove Models

These objects persist infomation about the clusters, datastores and other information to the Dtrove database. Creating instances of these are pretty straight forward if you know anything about Django. When a user adds a Cluster from the API or the web front end the underlining servers are created. Celery workers are spawn to create the nodes and then report back the status. For example:

```
>>> from dtrove.models import *
>>> ds = Datastore.objects.get(version='5.5')
>>> ds
<Datastore: mysql - 5.5>
>>> c = Cluster.objects.create(name='my_cluster', datastore=ds, size=10)
>>> c
<Cluster: my_cluster>
>>> c.datastore.name
u'mysql - 5.5'
>>> c.datastore.status
u'spawning'
```

Once the cluster is in an 'active' state further operations can be done.

2.4.1 Available models

class dtrove.models.**Cluster**(**args*, ***kwargs*)
Datastore Cluster

This is the Main User facing object it defines the datastore type along with the size of the cluster and the other user options.

Parameters

- **name** (*str*) – Name of the cluster
- **datastore** (dtrove.models.Datastore) – Datastore of the cluster
- **size** (*int*) – Number of nodes in the cluster

When a user creates a new Cluster the system handles provisioning the underlining instances and checking on the health of them.

After the cluster is created the user can then use the datastores, or perform certain actions:

•Schedule automated backups

•Import from a previous datastore export

•Export the data into a transportable type

•Change configuration parameters

The posibilities are endless!

add_node (*count=1*)
Add a node to the cluster.

class dtrove.models.**Datastore** (**args*, ***kwargs*)
Datastore

This represents a version of a specific datastore that is available

Parameters

- **manager_class** (dtrove.config.DTROVE_DATASTORE_MANAGERS) – The dotted path to the actual manager object
- **version** (*str*) – The version of the datastore ex: 5.6
- **image** (*str*) – The id of the VM image to use to create new instances
- **packages** (*str*) – Comma separated list of packages to install

The main purpose of this object is to link the datastore manager to a list of packages to install and the base image that is used to create instances.

By creating an datastore that makes it available for users to select and install a cluster from it.

The *manager_class* property should be an importable subclass of the dtrove.datastores.base.BaseManager class.

manager
The manager object initialize with this datastores information

name
The display name of the datastore (manager.name - version)

class dtrove.models.**Instance** (**args*, ***kwargs*)
Instance

This contains the information about the actual server instance that runs the datastore in the cluster.

Parameters

- **name** (*str*) – Name of the instance
- **cluster** (dtrove.models.Cluster) – Cluster that this instance is a part of
- **key** (dtrove.models.Key) – SSH Key pair object for this instance
- **user** (*str*) – The user which the SSH Key is connected to
- **addr** (*ipaddr*) – IP Address of this server
- **server** (*str*) – UUID of the nova server instance

Note: This model is internal only, all interactions are handled thought the cluster model and or the manager class on the cluster

---

Remote operations can be preformed on this instance by using the connection_info property like so:

```
from fabric.api import sudo, settings
from fabric.network import disconnect_all
from dtrove.models import Instance
```

```
instance = Instance.objects.first()

with settings(**instance.connection_info):
    sudo('rm -rf /etc/trove/*')

# Always remember to disconnect ssh sessions
disconnect_all()
```

See the dtrove.datastores.base.BaseManager class for more examples.

connection_info
Provides the connection info from the key stored for this server

message
Error message of the last server task

progress
Progress of the current server task

server_status
Status of the server

class dtrove.models.**Key**(**args*, ***kwargs*)
SSH Key Pair

This holds the public and private keys for connecting to a remote host.

Note: This should have encrypted fields. Don't put anything really secure in here. This is just a proof of concept.

Parameters

- **name** (*str*) – Name of the key
- **passphrase** (*str*) – Passphrase for the key
- **private** (*str*) – Text of the private key
- **public** (*str*) – Text of the public key

These keys are attached to an instance, the idea is that each cluster would have it's own ssh key assigned to it. That way there isn't a single master key that own's your entire network.

You should also use a passphrase for each key even though it is not a required field.

You can access the information in this key in the dtrove.models.Instance.connection_info property on the instance(s) the key is attached to.

classmethod create(*save=True*)
Factory method to create a new private/public key pair

2.5 Configuration Options

This extends the normal django settings module so we can simply call:

```
from dtrove import config
NOVA_USER = config.NOVA_USERNAME
```

This allows us to have a consistent place to define possible settings and document them as well as provide some defaults and warnings. To modify any of these values simply edit your django settings file.

class dtrove.**config**

This defines the available options to configure dtrove.

These are set or overridden in the django settings file.

DTROVE_DATASTORE_MANAGERS = [('dtrove.datastores.mysql.MySQLManager', 'mysql'), ('dtrove.datastores.redis.Red

List of available datastores. This should be a list of tuples:

```
[('path.to.Manager', 'manager_name'), ...]
```

DTROVE_MAX_CLUSTER_SIZE = 5

Max number of nodes in a cluster

DTROVE_PREFIX = 'sudo '

Prefix for the remote commands this has access to all the instance properties. EX:

```
DTROVE_PREFIX = 'sudo /usr/sbin/vzctl exec %(local_id)s '
```

DTROVE_PROVIDER = 'dtrove.providers.openstack.Provider'

The *Cloud Providers* that you have enabled.

DTROVE_SSH_USER = 'root'

Default SSH user to use when connecting to hosts

OS_AUTH_URL = 'http://0.0.0.0:5000/v2.0'

The url of the identity service for openstack.

OS_NOVA_BYPASS_URL = None

Actual url to use instead of the endpoint from the catalog.

OS_NOVA_ENDPOINT_TYPE = 'publicURL'

Type of the nova endpoint.

OS_PASSWORD = None

The openstack management user's password

OS_PROJECT_ID = None

OS_TENANT_ID or OS_PROJECT_ID of the openstack user.

OS_USERNAME = None

The openstack management username

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

d