CLoud Application Platform

Release 0.2.1

Otávio Napoli

Jun 06, 2021

CONTENTS

CHAPTER

ONE

INTRODUCTION

CLoud Application Provider (CLAP) provides a user-friendly command line tool to create, manage and interact with individual instances or a set of instances hosted in public cloud providers (such as AWS, Google Cloud and Microsoft Azure), as well as easily creates, manages, resizes and interacts with compute clusters hosted in public cloud providers. It was firstly inspired on elasticluster project, a tool that allows automated setup of compute clusters (MPI, Spark/Hadoop, etc.) and Ansible, a framework used for automation.

Its main features includes:

- YAML-Style configuration files to define nodes, logins and cloud configurations.
- User-friendly interface to create, setup, manage, interact and stop multiple instances hosted different cloud providers at the same time, transparently.
- Easy and fast creation and configuration of multiple compute clusters hosted in public cloud providers at same time.
- Growing and shrinking running clusters.
- Role system to easily perform actions in different heterogeneous nodes via Ansible. playbooks.
- Easy-to-use python API to bring nodes up and configure them (via ansible or SSH commands).

CHAPTER

TWO

CONTENTS

2.1 Introduction

2.1.1 Installation Guide

To install CLAP in a linux-based system follow the instructions below.

1. Install requirement packages

```
gcc g++ git libc6-dev libffi-dev libssl-dev virtualenv python3 python3-pip
```

Note: CLAP requires Python 3.7 or higher.

2. Clone the git repository and enter inside clap's directory

```
git clone https://github.com/lmcad-unicamp/CLAP.git clap
cd clap
```

3. Set execution flags of the install script using the `chmod` command. Then just run the `install.sh` script!

```
chmod +x install.sh
./install.sh
```

4. To use CLAP, you will need to activate the `virtual-env`, for each shell you are using it. Inside the `clap` root directory, where the `git` repository was cloned use the following command:

```
source clap-env/bin/activate
```

5. Finally, test CLAP, via the CLI interface. The `clapp` command should be available to use at this point.

```
clapp --help

clapp node list
```

Note: As CLAP is at development stage, use the `update.sh` periodically to fetch updates!

2.1.2 Quickly CLAP usage description

To use CLAP you will first need to provide some information about how to launch instances in the cloud. By default, CLAP holds all information about configurations in the `~/.clap/configs` directory. The `~/.clap/configs/providers.yaml` file describes how to connect to the cloud provider, the `~/.clap/configs/logins.yaml` file describes how to login into machines and the `~/.clap/configs/instances.yaml` describe the instances that can be used with CLAP. The *configuration section* will guide you to write all these configuration sections easily.

Once configurations written, the *usage section* will show you how to execute CLAP commands based on the configurations written. CLAP can be used to start, configure and manage single or multiple cloud's instances using the *node module* as well as entire compute clusters using the *cluster module*.

2.1.3 Quickly CLAP directory architecture description

By default, CLAP holds all of it information needed inside the `~/.clap` directory (where ~ stands for the user home directory). The minimal structure of `~/.clap` directory is shown below:

```
~/ (home directory)
└── .clap/
    ├── configs/
    │   ├── clusters/
    │   ├── instances.yaml
    │   ├── logins.yaml
    │   └── providers.yaml
    ├── roles/
    │   ├── actions.d/
    │   ├── group_vars/
    │   │   └── all.yml
    │   └── roles/
    ├── private/
    └── storage/
```

- The `~/.clap/configs/providers.yaml` YAML file inside the `~/.clap/configs` directory holds the information about the cloud provider and how to connect to it.
- The `~/.clap/configs/logins.yaml` file holds information about how to connect to an instance (e.g. login user, keyfile, etc)
- The `~/.clap/configs/instances.yaml` holds the information about the instances to launch, i.e. the instance templates.
- The `roles` directory store role's files and actions, used to perform action in several nodes. For more detailed information about roles and actions refer to the *roles section*
- The `private` stores keys and passwords files used to connect to the cloud provider and to the instance itself. Every key/secret files needed in the configuration files must be placed inside this directory (usually with 0400 permissions).
- The `storage` directory store metadata information used by CLAP.

2.2 Basic Configuration Setup

In order to create compute nodes and interact with them you will need provide some information about how to connect to the cloud provider (*providers configuration*), how to the login into the machines (*logins configuration*) and details about the cloud's virtual machines that can be used (*instances configuration*). The following sections will show how to configure these sections and the valid values for each one. All configuration files use the YAML File Format as default format.

Note: YAML use spaces instead of tabs. Be careful to do not messing up!

2.2.1 Cloud Provider Configuration

The `~/.clap/configs/providers.yaml` file defines all properties needed to connect to a specific cloud provider, such as the region, IAM access keys, among others. In this file you can define multiple provider configurations that is used by other configurations. An example `providers.yaml` file is shown below.

```
aws-east-1-config:                              # Name of the provider configuration ID
    provider: aws                               # Provider (currently only 'aws')
    access_keyfile: ec2_access_key.pub          # Name of the file in the ~/.clap/
↪private/ directory containing the IAM AWS access key ID
    secret_access_keyfile: ec2_access_key.pem   # Name of the file in the ~/.clap/
↪private directory containing the IAM AWS Secret Access Key (access ID)
    region: us-east-1                           # The availability zone you want to use

my-cool-config-2:
    provider: aws
    access_keyfile: acesss.pub
    secret_access_keyfile: private_access.pem
    region: us-east-2

my-cool-config-3:
    provider: aws
    ...
```

The YAML dictionary keys (`aws-east-1-config`, `my-cool-config-2` and `my-cool-config-3` in the above example) are the provider configuration names (provider IDs) that can be referenced in other files. The values for each provider ID are specific cloud provider information. You can define as many provider configurations as you want just adding a new provider ID and the values for it. Note that each provider ID must be unique. The valid values for a provider configuration showed in the table below.

Table 1: Valid cloud provider configuration key and values

Name	Valid Values or Type	Description
provider	**valid values**: aws	Name of the cloud provider to be used
access_keyfile	**type**: string	**Name of the file** containing the AWS access key ID. The file must be placed at `~/.clap/private` and this field must be filled only with the name of file, not the whole path.
secret_access_keyfile	**type**: string	**Name of the file** containing the AWS Secret Access Key (access ID). The file must be placed at `~/.clap/private` and this field must be filled only with the name of file, not the whole path.
region	**type**: string	The availability zone you want to use (e.g. `us-east-1`)
vpc (optional)	**type**: string	Name or ID of the AWS Virtual Private Cloud to provision resources in.

Note: For CLAP, **all keys** must be stored at `~/.clap/private/` directory with 400 permission (use the `chmod 400` command to set the read-only permission).

Note for AWS provider

IAM Access keys consist of two parts: an access key ID (for example, `AKIAIOSFODNN7EXAMPLE`) and a secret access key (for example, `wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`). These keys are **required** and is used to connect to the aws provider via third-party applications (see the AWS Access Keys documentation for more information). So you must place your access key ID string inside a file in the `~/.clap/private/`. For instance, copy and paste access key ID in a file and save at `~/.clap/private/ec2_access_key.pub`, or other filename and the same for the secret access key.

2.2.2 Login Configuration

The `~/.clap/configs/logins.yaml` defines all properties needed to access a started virtual machine via SSH, such as login user name, SSH key file used to access, etc. In this file you can define multiple login information that is used by other configurations. An example `logins.yaml` file is shown below.

```
ubuntu-login:                                    # Name of the login config (Login ID)
    user: ubuntu                                 # Login name used to SSH into the␣
↪virtual machine
    keypair_name: key_us_east_1                  # Name of the keypair to use on the␣
↪cloud provider
    keypair_public_file: key_us_east_1.pub       # Name of the file in the ~/.clap/
↪private directory containing the RSA/DSA public key corresponding to the private key␣
↪file
    keypair_private_file: key_us_east_1.pem      # Name of the file in the ~/.clap/
↪private directory containing a valid SSH private key to be used to connect to the␣
↪virtual machine.
    sudo: true                                   # True if the sudo_user can execute␣
↪commands as root by running the sudo command
    sudo_user: root                              # (OPTIONAL) Login name of the super␣
↪user (default is root)
```

(continues on next page)

(continued from previous page)

```
example-centos:
    user: centos
    ...
```

The YAML dictionary keys (`ubuntu-login` and `example-centos` in the above example) are login's configuration name (also called login ID). The values are the specific information about that configuration. You can define as many login configurations as you want just adding a new login ID and the values for it. Note that each login ID must be unique. The valid values for a login configuration are:

Table 2: Valid login configuration key and values

Name	Values/Type	Description
user	**type**: string	Name of the user used to perform SSH into the virtual machine.
keypair_name	**type**: string	Name of the keypair used on the cloud provider.
keypair_public_file	**type**: string	Name of the file in the `~/.clap/private` directory containing the RSA/DSA public key corresponding to the private key file.
keypair_private_file	**type**: string	Name of the file in the `~/.clap/private` directory containing a valid SSH private key to be used to connect to the virtual machine.
sudo	**type**: boolean	True if the sudo user can execute commands as root by running the sudo command.
sudo_user (optional)	**type**: string	Optional login name of the super user (default is root)

The keypair is used to login to the machine without password (and perform SSH).

Note for AWS users

For AWS users, the keypair can be generated in the menu: `EC2 --> Network & Security --> Key Pairs`. A keypair can be created using the `create key pair` button providing an unique keypair name (this name is used in the `keypair_name` field of the login configuration). When a keypair is created, a private key file is generated to download. This is the **private key file** (used to login to the instances).

For CLAP, all key files must be placed in the `~/.clap/private/` directory with 400 permission. In the **keypair_private_file** login configuration field, the name of the private key file inside the `~/.clap/private/` must be inserted (e.g. **only** the file name: `key_us_east_1.pem.pem` and not `~/.clap/private/key_us_east_1.pem.pem`)

If you have a private key, the public key can be obtained with the command `ssh-keygen -y -f /path_to_key_pair/my-key-pair.pem` (where `my-key_pair.pem` is the private key file. See AWS Keypair Documentation for more details). The generated public key must be saved to a file and placed at `~/.clap/private/` directory with 400 permission. So, in the `keypair_public_file` field of the login configuration, the name of the public key file must be inserted.

2.2.3 Instance Templates Configuration

To start virtual machines in a cloud, you must first setup some instance templates in the `~/.clap/configs/instances.yaml` file. The template contains information about the virtual machine to start, including its flavor (i.e. instance type, for instance `t2.micro` on AWS clouds), security group information, AMI used, the disk size and others. The instance template references the provider configuration and login configuration by its ID. An example of `instances.yaml` file is shown below.

```
ubuntu-instance-aws:                    # Name of the instance template (instance␣
↪template ID)
    provider: aws-east-1-config         # Provider configuration ID
    login: ubuntu-login                 # Login configuration ID
    flavor: t2.medium                   # The VM "size" to use. Different cloud␣
↪providers call it differently: could be "instance type", "instance size" or "flavor".
    image_id: ami-07d0cf3af28718ef8     # Disk image ID to use in the VM. Amazon EC2␣
↪uses IDs like ami-123456
    security_group: xxx-secgroup        # Name of security group to use when starting␣
↪the instance
    boot_disk_size: 10                  # (OPTIONAL) Size of the instance's root␣
↪filesystem volume, in Gibibytes (GiB)
    boot_disk_device: /dev/sda1         # (OPTIONAL) Device name of the instance's root␣
↪file system in the block device mapping
    boot_disk_type: gp2                 # (OPTIONAL) Root filesystem volume storage type,
↪ one of gp2 (general purpose SSD), io1 (provisioned IOPS SSD), or standard (the␣
↪default).
    placement_group: XXX                # (OPTIONAL) Placement group to enable low-
↪latency networking between compute nodes
    image_userdata: '...'               # (OPTIONAL) Shell script to be executed (as␣
↪root) when the machine starts.
    network_ids:  subnet-abcdfefxx      # (OPTIONAL) Subnet IDs the nodes will be␣
↪connected to

instance-t2small-us-east-1:
    provider: aws-east-1-config
    ...
```

The YAML dictionary keys (`ubuntu-instance-aws` and `instance-t2small-us-east-1` in the above example) are the name of the instance templates (also called instance template ID) and the values are the specific information about that instance template. You can define as many instance templates configurations as you want just adding a new instance template ID and the values for it. Note that each instance template ID must be unique. Commands will use the instance template ID to start instances based on this information. The valid values for the instance templates are:

Table 3: Valid instance template key and values

Name	Values/Type	Description
provider	**type**: string	The ID of the provider configuration to be used for this instance. The ID must match the provider ID at `providers.yaml`
login	**type**: string	The ID of the login configuration to be used for this instance. The ID must match the login ID at `logins.yaml`
flavor	**type**: string	The provider instance type to use (e.g. `t2.micro`, `c5.large`, etc)
image_id	**type**: string	Disk image ID to use in the VM (basically the OS to be used). Amazon EC2 uses IDs like `ami-123456`. Note that the image_id is dependent of the provider region and a error may be raised if an invalid AMI id is specified
security_group	**type**: string	Name of security group to use when starting the instance
boot_disk_size (optional)	**type**: string	Size of the instance's root filesystem volume, in Gibibytes (GiB)
boot_disk_device (optional)	**type**: string	Device name of the instance's root file system in the block device mapping. For AWS, see block device mapping docs for more information
boot_disk_type (optional)	**type**: string	Root filesystem volume storage type, one of gp2 (general purpose SSD), io1 (provisioned IOPS SSD), or standard (default). See Root filesystem volume storage type for more information
placement_group (optional)	**type**: string	Placement group to enable low-latency networking between compute nodes. See placement groups for more information
network_ids (optional)	**type**: string	Subnet ID that the nodes of the cluster will be connected to

2.3 Basic Usage

CLAP is a platform to start, stop and manage cloud's instances (called CLAP nodes or simply, nodes) at different cloud providers transparently, based on configuration files. Also, it offers mechanisms to perform actions via SSH commands or Ansible playbooks in single nodes or in a set of nodes in a row. To provide this, in a modular way, CLAP provides modules to allow performing several operations. You can use `clapp --help` command to list the available modules.

The most common modules are: `node`, `role` and `cluster`.

2.3.1 Node Module

The node module provides mechanisms to create, manage and interact with cloud's instances. It provides the following features:

- Start nodes based on the instance templates with the `start` command.
- Stop (terminate) already started nodes using the `stop` command.
- Pause or resume already instantiated nodes using the `pause` and `resume` commands, respectively.
- Check the status of a node (if its accessible by SSH) using the `alive` command.
- List started nodes using the `list` command.
- Execute a shell command via SSH, using the `execute` command.

- Execute an Ansible Playbook using the `playbook` command.
- Obtain a shell session (via SSH) using the `connect` command.
- Add and remove tags from nodes using `add-tag` and `remove-tag` commands.
- List all available instance templates configurations using the `list-templates` command.

All these commands are detailed below.

Command `node start`

To launch a cloud's instance based on an instance template, defined in the `~/.clap/configs/instances.yaml` file, you can use the command below, where the `ubuntu-instance-aws` refers to the instance template ID defined in the `~/.clap/configs/instances.yaml` file. In this way, you need to configure the files only once and launch instances at any time.

```
clapp node start ubuntu-instance-aws
```

Once instances are successfully started, CLAP will assign an unique node ID to each instance, used to perform other CLAP operation. Also, CLAP will try to login at the instance with the login information provided, via SSH.

To launch more than one instance with the same instance template ID, you can put the desired number after the instance template ID preceded by an `:` character. For instance, the command below, launches 4 `ubuntu-instance-aws` instances in a row.

```
clapp node start ubuntu-instance-aws:4
```

You can also launch different instances in a row using the same command, but just appending more instance template IDs to it, as below. The above command launches 2 `ubuntu-instance-aws` VMs and 2 `example-instance-aws` VMs in a row.

```
clapp node start ubuntu-instance-aws:2 example-instance-aws:2
```

Command `node list`

The `clapp node list` command can be used to show managed CLAP's nodes. An example output of this command is shown below.

```
* Node: ebcd658bacdf485487543cbcc721d1b3, config: type-a, nickname: MarjoryLang, status:␣
↪reachable, ip: 3.87.87.154, tags: {}, roles: [], creation at: 21-05-21 14:11:55
Listed 1 nodes
```

The node id (`ebcd658bacdf485487543cbcc721d1b3` in the above example) is used across all other modules and commands to perform commands in this node.

Command `node alive`

This command updates several node's information (such as IP) and check if the node is reachable (if a SSH connection can be established).

The node's status can be:

- **started**: when the VM is up.
- **reachable**: when the VM is up and a SSH connection was successfully established.
- **unreachable**: when the SHH connection was not successfully established.
- **paused**: when VM is paused.
- **stopped**: when VM is terminated.

Note: CLAP does not check the status of VM periodically. Use this command to update node status and information.

Command `node stop`

The `clapp node stop` command can be used to **terminate** an running VM (destroying it). The syntax is shown below:

Command `node pause`

The `clapp node pause` command can be used to **pause** an running instance. When a node is paused, its status is changed to **paused** and its public IP is changed to `None`.

Note: The command has no effect for nodes that already been paused.

Command `node resume`

The `clapp node resume` command can be used to **resume** a paused instance. When a node is resumed, it status is changed to **started**. Then, it checked if it is alive, testing its connection and updating its public IP (and changing its status to **reachable**).

Note: The command has no effect at nodes that were not paused. It will only check for its aliveness.

Command `node connect`

The `clapp node connect` command can be used to obtain a shell to a specific node.

Note: The connection may fail if node has an invalid public IP or a invalid login information. You may want to check if node is alive first to update node's information.

Command `node execute`

The `clapp node execute` command can be used to execute a shell command on an reachable node. The syntax is shown below:

```
Usage: clapp node execute [OPTIONS] [NODE_ID]...

  Execute a shell command in nodes (via SSH)

Options:
  -t, --tags TEXT         Filter nodes by tags. There are two formats: <key> or
                          <key>=<val>
  -cmd, --command TEXT    Shell Command to be executed in nodes  [required]
  --timeout INTEGER       Timeout to execute command in host (0 to no timeout)
                          [default: 0]
  -a, --additional TEXT   Additional arguments to connection. Format:
                          <key>=<val>
  --help                  Show this message and exit.
```

One or more nodes can be passed as argument, or can be selected based on their tags. The `--command` parameter specify the command that will be executed in nodes.

An example is shown below, executing a simple `ls -lha` command in the node `ebcd658bacdf485487543cbcc721d1b3`

```
clapp node execute ebcd658bacdf485487543cbcc721d1b3 -cmd "ls -lha"
```

And the result:

```
------------------------ ebcd658bacdf485487543cbcc721d1b3 ------------------------
return code ebcd658b: 0
stdout ebcd658b: drwxr-xr-x 5 ubuntu ubuntu 4.0K May 21 17:12 .
stdout ebcd658b: drwxr-xr-x 3 root   root   4.0K May 21 17:12 ..
stdout ebcd658b: -rw-r--r-- 1 ubuntu ubuntu  220 Apr  4  2018 .bash_logout
stdout ebcd658b: -rw-r--r-- 1 ubuntu ubuntu 3.7K Apr  4  2018 .bashrc
stdout ebcd658b: drwx------ 2 ubuntu ubuntu 4.0K May 21 17:12 .cache
stdout ebcd658b: drwx------ 3 ubuntu ubuntu 4.0K May 21 17:12 .gnupg
stdout ebcd658b: -rw-r--r-- 1 ubuntu ubuntu  807 Apr  4  2018 .profile
stdout ebcd658b: drwx------ 2 ubuntu ubuntu 4.0K May 21 17:12 .ssh
```

Note: You may want to check for nodes aliveness first.

Command `node playbook`

The `clapp node playbook` command can be used to execute an Ansible playbook in a set of reachable nodes. The syntax is shown below:

```
clapp node playbook [OPTIONS] [NODE_ID]...

    Execute an Ansible playbook in a set of nodes.

    The NODE_ID argument is a list of strings (optional) and can filter nodes to
```

(continues on next page)

(continued from previous page)

```
    execute the playbook by their node ids

Options:
    -p, --playbook TEXT    Path of the playbook to be executed  [required]
    -t, --tags TEXT        Filter nodes by tags. There are two formats: <key> or
                          <key>=<val>
    -e, --extra TEXT       Extra variables to be passed. Format: <key>=<val>
    -nv, --node-vars TEXT  Host variables to be passed. Format:
                          <node_id>:<key>=<val>,<key>=<val>,...
    --help                 Show this message and exit.
```

One or more nodes can be passed as argument, or can be selected based on their tags.

The `--playbook` parameter specify the playbook to execute in nodes.

The `--extra` parameter can be used to pass keyword arguments to the playbook.

The `--node-vars` parameter can be used to pass keyword arguments to a specific node when building the inventory.

An example is shown below. The playbook `install_packages.yml` is executed in node `ebcd658bacdf485487543cbcc721d1b3`. Extra playbook variables (in jinja format, e.g. "{{ var1 }}") will be replaced by the extra variables informed. In the example below the playbook's variable `packages` will be replaced by `gcc`.

```
clapp node playbook ebcd658bacdf485487543cbcc721d1b3 -p install_packages.yml -e
↪"packages=gcc"
```

Command `node add-tag`

This `clapp node add-tag` command adds a tag to a set of nodes and has the following syntax:

```
Usage: clapp node add-tag [OPTIONS] NODE_ID...

  Add tags to a set of nodes.

  The NODE_ID argument is a list of node_ids to add tags.

Options:
  -t, --tags TEXT  Tags to add. Format: <key>=<val>  [required]
  --help           Show this message and exit.
```

One or more nodes can be passed as argument. The `tags` parameter must be a keyword value in the format `key=value`. You can add as many tags to a node as you want. An example of adding tags is shown below:

```
clapp node add-tag ebcd658bacdf485487543cbcc721d1b3 -t x=10
```

Where tag `x=10` is added to nodes `ebcd658bacdf485487543cbcc721d1b3`.

Command `node remove-tag`

This `clapp tag remove` command removes a tag from a set of nodes and has the following syntax:

```
clapp node remove-tag [OPTIONS] NODE_ID...

  Remove tags from a set of nodes.

  The NODE_ID argument is a list of node_ids to remove tags.

Options:
  -t, --tags TEXT  Tags to remove. Format: <key>  [required]
  --help           Show this message and exit.
```

One or more nodes can be passed as argument. The `tag` parameter must be a string. The tags from nodes that matches to the informed tag is removed (tag and value).

2.3.2 Role Module

The role module allows to perform pre-defined actions to a set of nodes that belongs to a role. When a node is added to a role, it is said that this node is ready to perform tasks of this role. Thus, each role defines their set of specific actions that can be performed to nodes that belongs to that particular role.

In this way, the role module consists of three steps:

1. Add nodes to a role.
2. Perform role's action to nodes that belongs to a role.
3. Optionally, remove nodes from the group.

The nodes of a role can also be logically divided in hosts. Thus, role actions can be performed to all nodes of the role or to a subset of nodes of role (hosts).

CLAP's roles and actions

Role's actions are Ansible playbooks that are executed when an action is invoked (e.g. using `role action` command). By default CLAP's roles are stored in the `~/.clap/roles/` directory and each role consists in at minimum of two files:

- A YAML description file describing the actions that can be performed (and informing the playbook that must be called) and, optionally, the hosts (subset of role's nodes to execute the playbook)
- The Ansible Playbook called when each action is invoked.

You can see some roles shared with CLAP and their requirements at *Roles shared with CLAP* section.

Role description file

The role's description files are python files placed at `~/.clap/groups/actions.d` directory. The name of the YAML file defines the role's name. Each role description file defines the key `actions` and, optionally, the `hosts` key. Inside `actions` key, each dictionary defines a role action where the key name is the action name and the values informs characteristic of that action.

An example role description file is shown below, for a role named `commands-common` (placed at `~/.clap/roles/actions.d/commands-common.yaml`).

```
---
actions:                                                    # Defines the actions of␣
↪this group
    setup:                                                  # Action called setup
        playbook: roles/commands-common_setup.yml           # Playbook to be␣
↪executed when this group action is invoked

    copy:                                                   # Action called copy
        playbook: roles/commands-common_copy.yml            # Playbook to be␣
↪executed when this group action is invoked
        description: Copy files from localhost to remote hosts  # Optional action's␣
↪description
        vars:                                               # Optional variables␣
↪required
        - name: src                                         # src variable
          description: Source files/directory to be copied      # Optional variable's␣
↪description
          optional: no                                      # Informs if this␣
↪variable is optional
        - name: dest                                        # dest variable
          description: Destination directory where files will be placed # Optional␣
↪variable's description

    fetch:
        playbook: roles/commands-common_fetch.yml
        description: Fetch files from remote hosts to localhost
        vars:
        - name: src
          description: Source files/directory to be fetched
        - name: dest
          description: Destination directory where files will be placed

hosts:                                                      # (optional) List of␣
↪hosts that are used in this role. The host name can be used in the playbooks.
- master
- slave
```

Note: Action's playbook is relative to the `~/.clap/roles/` directory.

For role's description files, `actions` dictionary is required, and `hosts` optional. The keys inside `actions` dictionary are the action names and the possible values for each action are described in table below.

Table 4: Valid values for actions

Name	Type	Description
`playbook`	path	Playbook to be executed when this action is invoked. The path is relative to `~/.clap/roles/` directory.
`description` (optional)	string	Action's descriptive information
`vars` (optional)	List of variable dictionaries	List informing variables needed for this action

And optionally, the actions can define their variables to use. The possible values are listed table below

Table 5: Valid action's values

Name	Type	Description
`name`	string	Name of the variable
`description` (optional)	string	Variable's descriptive information
`optional` (optional)	boolean	Inform if variable is optional (default is `no`)

Finally the hosts specify the `hosts` used by role actions. It's optional and when specified Ansible playbooks can segment their execution using the `hosts` variable at each play. If no hosts are specified you must use `hosts: all` to perform the action over all nodes that belong to the role.

Command `role list`

The `clapp role list` command can be used to list all available role and their respective actions and hosts. An example of output is shown below

```
* name: commands-common
  Has 7 actions and 2 hosts defined
    actions: copy, fetch, install-packages, reboot, run-command, run-script, update-
↪packages
    hosts: h1, h2

* name: ec2-efs
  Has 3 actions and 0 hosts defined
    actions: mount, setup, unmount
    hosts:

* name: spits
  Has 6 actions and 2 hosts defined
    actions: add-nodes, job-copy, job-create, job-status, setup, start
    hosts: jobmanager, taskmanager

Listed 3 roles
```

Command `role add`

The `clapp role add` command can be used to add a node to a role. The syntax is shown below:

```
clapp role add [OPTIONS] ROLE

  Add a set of nodes to a role.

  The ROLE argument specify the role which the nodes will be added.

Options:
  -n, --node TEXT         Nodes to be added. Can use multiple "-n" commands and
                          it can be a list of colon-separated nodes as
                          "<node>,<node>,..." or
                          "<role_host_name>:<node>,<node>". The formats are
                          mutually exclusive  [required]
  -nv, --node-vars TEXT   Node's arguments. Format
                          <node_id>:<key>=<value>,<key>=<val>
  -hv, --host-vars TEXT   Role's host arguments. Format
                          <host_name>:<key>=<value>,...
  -e, --extra TEXT        Extra arguments. Format <key>=<value>
  --help                  Show this message and exit.
```

The nodes can be supplied with `--node` parameter using two formats (mutually exclusive): with host or without host.

If the role does not define any host, nodes must be informed supplying only their node ids in the `--node` parameter. Multiple `--node` parameters can be used to indicate multiple nodes ids. Besides that, multiple nodes ids can be passed to `--node` parameter by separating them with comma. The both examples below add nodes `ebcd658bacdf485487543cbcc721d1b3` and `455e9c5da5c4417abc757f587a31c105` to role `commands-common`.

```
clapp role add commands-common -n ebcd658bacdf485487543cbcc721d1b3 -n␣
↪455e9c5da5c4417abc757f587a31c105
clapp role add commands-common -n ebcd658bacdf485487543cbcc721d1b3,
↪455e9c5da5c4417abc757f587a31c105
```

If the role defines one or more hosts, the `--node` parameter can be supplied with the "<node>,<node>,..." format (1) or with the "<role_host_name>:<node>,<node>" format (2) (both are mutually exclusive). If the format (1) is used, the nodes are added to all role's hosts defined . Two examples are shown below, one for format (1) and other for format (2).

```
clapp role add commands-common -n ebcd658bacdf485487543cbcc721d1b3 -n␣
↪455e9c5da5c4417abc757f587a31c105
clapp role add commands-common -n masters:ebcd658bacdf485487543cbcc721d1b3 -n␣
↪slaves:455e9c5da5c4417abc757f587a31c105
```

Supposing the role `commands-common` defines 2 hosts: `masters` and `slaves`, the first one adds nodes `ebcd658bacdf485487543cbcc721d1b3` and `ebcd658bacdf485487543cbcc721d1b3` to both role's host. The second one adds node `ebcd658bacdf485487543cbcc721d1b3` as commands-common masters and node `455e9c5da5c4417abc757f587a31c105` as commands-common slaves host.

The `--extra` parameter can be used to pass keyword arguments to the playbook.

The `--node-vars` parameter can be used to pass keyword arguments to a specific node when building the inventory.

The `--host-vars` parameter can be used to pass keyword arguments to a hosts.

Note: If the role's setup action is defined this action is immediately executed when adding a role to a node. If this action fails, the node is not added to the role.

Command `role action`

The `clapp role action` command can be used to perform an action in all nodes belonging to a particular role. The syntax is shown below:

```
clapp role action [OPTIONS] ROLE

  Perform an group action at a set of nodes.

  The ROLE argument specify the role which the action will be performed.

Options:
  -a, --action TEXT      Name of the group's action to perform  [required]
  -n, --node TEXT        Nodes to perform the action. Can use multiple "-n"
                         commands and it can be a list of colon-separated node
                         as "<node>,<node>,..." or
                         "<role_host_name>:<node>,<node>". The formats are
                         mutually exclusive. If not is passed, the action will
                         be performed in all nodes that belongs to the role.
  -nv, --node-vars TEXT  Node's arguments. Format
                         <node_id>:<key>=<value>,<key>=<val>
  -hv, --host-vars TEXT  Role's host arguments. Format
                         <host_name>:<key>=<value>,...
  -e, --extra TEXT       Extra arguments. Format <key>=<value>
  --help                 Show this message and exit.
```

The `--node` parameter is optional and if is not supplied, the role action will be executed in all nodes that belongs to the role. If `--node` parameter is supplied it may be in two formats (mutually exclusive): with host or without host.

If nodes are informed in format without host, the selected nodes will be automatically placed in their correct hosts (if any). Otherwise, the nodes will be placed in informed hosts.

Examples are shown below:

```
clapp role action commands-common -a install-packages -n␣
↪ebcd658bacdf485487543cbcc721d1b3 -e packages=gcc
clapp role action commands-common -a install-packages -n␣
↪masters:ebcd658bacdf485487543cbcc721d1b3 -e packages=gcc
clapp role action commands-common -a install-packages -e packages=gcc
```

The first command perform `install-packages` action, from `commands-common` role in nodes `ebcd658bacdf485487543cbcc721d1b3`. The node's hosts are the same when the nodes added. The second command perform `install-packages` action, from `commands-common` role in node `ebcd658bacdf485487543cbcc721d1b3`. The node's hosts acts only as `masters`, additional hosts from this node are discarded. The last command perform `install-packages` action, from `commands-common` role at all nodes that belongs to `commands-common`. For all commands, the extra variable `package` with value `gcc` is passed.

The `--extra` parameter can be used to pass keyword arguments to the playbook.

The `--node-vars` parameter can be used to pass keyword arguments to a specific node when building the inventory.

The `--host-vars` parameter can be used to pass keyword arguments to a hosts.

Command `role remove`

The `clapp role action` command can be used to perform an action in all nodes belonging to a particular role. The syntax is shown below:

```
clapp role remove [OPTIONS] ROLE

  Perform an group action at a set of nodes.

  The ROLE argument specify the role which the action will be performed.

Options:
  -n, --node TEXT  Nodes to perform the action. Can use multiple "-n" commands
                   and it can be a list of colon-separated node as
                   "<node>,<node>,..." or "<role_host_name>:<node>,<node>".
                   The formats are mutually exclusive. If not is passed, the
                   action will be performed in all nodes that belongs to the
                   role.  [required]
  --help           Show this message and exit.
```

The `--node` parameter is used to inform the nodes to remove from a role. The parameter can be supplied using two formats (mutually exclusive): with host or without host. If host is passed, the node is removed from the host's role else the node is removed from all hosts in the role (if any). An example is shown below:

```
clapp role remove commands-common -n ebcd658bacdf485487543cbcc721d1b3 -n␣
↪455e9c5da5c4417abc757f587a31c105
clapp role remove commands-common -n masters:ebcd658bacdf485487543cbcc721d1b3 -n␣
↪slaves:455e9c5da5c4417abc757f587a31c105
```

The first example remove nodes `ebcd658bacdf485487543cbcc721d1b3` and `455e9c5da5c4417abc757f587a31c105` from role `commands-common` and from all `commands-common` role hosts (if any). The second example removes node `ebcd658bacdf485487543cbcc721d1b3` from host called `masters` from `commands-common` role and node `455e9c5da5c4417abc757f587a31c105` from hosts called `slaves` from `commands-common` role.

2.3.3 Cluster Module

The cluster module allows CLAP to work with cluster, which is a set of CLAP's nodes tagged with a specific tag. A CLAP's cluster is created taking as input configuration files, in YAML format, which will create nodes and setup each of them properly. After created, the cluster can be resized (adding or removing nodes), paused, resumed, stopped, among other things.

By default, the CLAP's cluster module will find configurations inside `~/clap/configs/clusters` directory. At next sections, we will assume that files will be created inside this directory (in `.yaml` format).

The next section will guide you to write a cluster configuration and then, module's commands will be presented.

Cluster Configuration

To create a CLAP's cluster you will need to write:

- **Setup configuration sections**: which define a series of groups and actions that must be performed.
- **Cluster configuration sections**: which define a set of nodes that must be created and the setups that must be performed in each node.

Setups and cluster section may be written in multiple different files (or at the same file), as CLAP's cluster modules will read all files (and setups and clusters configurations, respectively) inside the cluster's directory.

Setup Configuration Sections

Setup configuration sections define a series of roles and/or actions that must be performed at cluster's nodes. An example of a setup configuration section is shown below.

```
# Setup configurations must be declared inside setups key
setups:

    # This is a setup configuration called setup-common
    setup-common:
        roles:
        - name: commands-common         # Add nodes to commands-common role
        - name: another-role            # Add nodes to another-role role
        actions:
        - role: commands-common
          action: update-packages       # Perform action update-packages from role␣
↪commands-common
        - command: "git init"           # Perform shell command 'git init'

    # This is a setup configuration called setup-spits-jobmanager
    setup-spits-jobmanager:
        roles:
        - name: spits/jobmanager        # Add nodes to spits' role as jobmanager host

    # This is a setup configuration called setup-spits-taskmanager
    setup-spits-taskmanager:
        roles:
        - name: spits/taskmanager       # Add nodes to spits' role as taskmanager host
```

Setup configurations must be written inside `setups` YAML-dictionary. You can define as many setup configurations as you want, even at different files but each one must have a unique name. Inside the `setups` section, each dictionary represents a setup configuration. The dictionary key (`setup-common`, `setup-spits-jobmanager` and `setup-spits-taskmanager` in above example) represent the setup configuration ID.

Each setup configuration may contain two dictionaries: `roles` and `actions` (both are optional). Both sections, for a setup configuration is described in the next two subsections.

Roles key at setups configuration

The `role` section inside a setup configuration tells to add nodes, whose perform this setup, to the defined roles. The `roles` section contains a **list** describing each role that the nodes must be added. Also, the role is always added in the order defined in the list.

Each element of the list must have a `name` key, which describe the name of the role that the node must be added. For instance, the `setup-common` at above example, defines two roles which nodes that perform this setup must be added: `commands-common` and `another-role` (in this order).

Optionally an `extra` key can be defined by each role, as a dictionary. The key and values is passed as `extra` parameter similar to the `role add` module command. For instance, the setup below, will add nodes that perform this setup (`setup-common-2`) to role `example-role` passing variables, `foo` and `another_var` with values `bar` and `10`, respectively.

```
# Setup configurations must be declared inside setups key
setups:

    # This is a setup configuration called setup-common
    setup-common-2:
        roles:
        - name: example-group     # Add nodes to example-role role
          extra:
            foo: bar
            another_var: 10
```

Actions key at an setups configuration

The `actions` section inside a setup configuration tells to perform actions at nodes which perform this setup. The `actions` section contains a **list** describing each action that must be performed (in order). There are three types of actions:

- **role action**: will perform an role action. Thus, the `role` and `action` keys must be informed. The `role` key will tell the name of the role and the `action` key will tell which action from that role which will be performed. Optionally, an `extra` dictionary can be informed to pass keyword variables to the role's action.
- **playbook**: will execute an Ansible Playbook. Thus, the `playbook` key must be informed, telling the absolute path of the playbook that will be executed. Optionally an `extra` dictionary can be informed to pass keyword variables to the playbook.
- **command**: will execute a shell command. Thus, the `command` key must be informed, telling which shell command must be executed.

Some action examples are briefly shown below:

```
# Setup configurations must be declared inside setups key
setups:

    # This is a setup configuration called setup-common. The actions are executed␣
↪sequentially
    another-setup-example:
        actions:
        # Perform mount action from role nfs-client, passing the variable mount_path␣
↪with value /mnt
        - action: mount
```

(continues on next page)

(continued from previous page)

```
        role: nfs-client
        extra:
          mount_path: /mnt
      # Execute the playbook /home/my-cool-ansible-playbook with an variable foo with␣
↪value bar
      - playbook: /home/my-cool-ansible-playbook
        extra:
          foo: bar
      # Execute a shell command: hostname
      - command: hostname
      # Perform reboot action from commands-common role
      - role: commands-common
        action: reboot
```

Note: If a setup configuration contains both `roles` and `actions` sections, the `roles` section will **always** be executed before `actions` section.

Cluster Configuration Sections

The cluster configuration defines a set of nodes that must be created and setups that must be executed. Clusters are written inside `clusters` YAML-dictionary key and each dictionary inside `clusters` key denotes a cluster (where the dictionary key is the cluster's name). Above is an example of a cluster configuration:

```
# Clusters must be defined inside clusters key
clusters:
  # This is the cluster name
  my-cool-cluster-1:
    # Nodes that must be created when a cluster is instantiated
    nodes:
      # Node named master-node
      master-node:
        type: aws-instance-t2.large   # Instance type that must be created (must match␣
↪instances.yaml name)
        count: 1                      # Number of instances that must be created
        setups:                       # Optionally, list of setups to be executed when␣
↪the master-nodes is created (reference setup configuration names, at setups section)
        - another-example-setup
        - master-setup

      # Node named taskmanager
      slave-nodes:
        type: aws-instance-t2.small  # Instance type that must be created (must match␣
↪instances.yaml name)
        count: 2                     # Number of instances that must be created
        min_count: 1                 # Minimum desired number of instances that must␣
↪effectively be created
        setups:                      # Optionally, list of setups to be executed when␣
↪the slave-nodes is created
        - setup-slave-node
```

Clusters must have the `nodes` section, which defines the nodes that must be created when the cluster is instantiated. As example above, each cluster's node have a type (`master-node` and `slave-node`) and values, that specify the cluster's node characteristics. Each node may have the values listed in is table below.

Table 6: Cluster's nodes valid parameters

Name	Type	Description
`type`	string	Instance type that must to be created. The type must match the node name at `instances.yaml` file
`count`	Integer	Number of instances of this type to be launched
`min_count` (OPTIONAL)	Positive integer (less then or equal `count` parameter)	Minimum number of instances of this type that must effectively be launched. If this parameter is not supplied the value of `count` parameter is assumed
`setups`	List of strings	List with the name of the setup configurations that must be executed after nodes are created

When a cluster is created, the instance types specified in the each node section is created with the desired `count` number. The cluster is considered created when all nodes are effectively created. The `min_count` parameter at each node specify the minimum number of instances of that type that must effectively be launched. If some instances could not be instantiated (or created wwith less than `min_count` parameter) the cluster creation process fails and all nodes are terminated.

After the cluster is created, i.e. the minimum number of nodes of each type is successfully created, the `setups` for each node is executed, in order. If some setup does not execute correctly, the cluster remains created and the `setup` phase can be executed again.

Controlling cluster's setups execution phases

CLAP's cluster module also offers some other facilities to configure the cluster. By default the cluster module create nodes and run the setup from each node type. You can control the flow of the setup execution using some optional keys at your cluster configuration. The keys: `before_all`, `before`, `after` and `after_all` can be plugged into a cluster's configuration, in order to execute setups in different set of nodes, before and after the nodes setups. These keys takes a list of setups to execute. CLAP's setup phases are executed in the order, as shown in table bellow.

Table 7: Cluster's setups execution phases (in order)

Phase name	Description
`before_all` (#1)	Setups inside this key are executed in all cluster's nodes before specific setup of the nodes (#3).
`before` (#2)	Setups inside this key are executed only in nodes that are currently being added to the cluster, before the setup specific setup of the nodes (#3). Its useful when resizing cluster, i.e., adding more nodes. This phase is always executed at cluster creation, as all created nodes are being added to the cluster.
`node` (#3)	The setup for each node is executed. The setup (inventory generated) is executed only at nodes of this type
`after` (#4)	Setups inside this key are executed only in nodes that are currently being added to the cluster, after the setup specific setup of the nodes (#3). Its useful when resizing cluster, i.e., adding more nodes. This phase is always executed at cluster creation, as all created nodes are being added to the cluster.
`after_all` (#5)	Setups inside this key are executed in all cluster's nodes after specific setup of the nodes (#3).

Note: All setups are optional

An example is shown below:

```
# Clusters must be defined inside clusters key
clusters:
  # This is the cluster name
  my-cool-cluster-1:
    # These setups are executed at all cluster's nodes, before setups at nodes section
    before_all:
    - my-custom-setup-1

    # These setups are executed at nodes that are currently being added to cluster,␣
↪before setups at nodes section
    before:
    - my-custom-setup-2

    # These setups are executed at nodes that are currently being added to cluster,␣
↪after setups at nodes section
    after:
    - my-custom-setup-3
    - my-custom-setup-4

    # These setups are executed at all cluster's nodes, after setups at nodes section
    after_all:
    - final_setup

    # Nodes that must be created when a cluster is instantiated
    nodes:
      # Node named master-node
      master-node:
        type: aws-instance-t2.large   # Instance type that must be created (must match␣
↪instances.yaml name)
        count: 1                      # Number of instances that must be created
        setups:                       # Optionally, list of setups to be executed when␣
↪the master-nodes is created (reference setup configuration names, at setups section)
        - another-example-setup
        - master-setup

      # Node named taskmanager
      slave-nodes:
        type: aws-instance-t2.small  # Instance type that must be created (must match␣
↪instances.yaml name)
        count: 2                     # Number of instances that must be created
        min_count: 1                 # Minimum desired number of instances that must␣
↪effectively be created
        setups:                      # Optionally, list of setups to be executed when␣
↪the slave-nodes is created
        - setup-slave-node
```

In the above example, supposing you are creating a new cluster, after the creation of nodes the following setups are executed (in order):

- `before_all` setups: `my-custom-setup-1` at all nodes
- `before` setups: `my-custom-setup-2` at all nodes
- `nodes` setups (not necessary in order): `another-example-setup` and `master-setup` at `master-nodes` nodes

and `setup-slave-node` at `slave-nodes` nodes.

- `after` setups: `my-custom-setup-3` and `my-custom-setup-4` at all nodes
- `after_all` setups: `final_setup` at all nodes

Now supposing you are resizing the already created cluster (adding more `slave-nodes` to it), the `before_all` and `after_all` setups will be executed in all cluster's nodes (including the new ones, that are being added) and `before`, `nodes` and `after` phase setups will only be executed at nodes that are being added to the the cluster.

Other cluster's setups optional keys

The `options` key can be plugged at a cluster configuration allowing some special options to cluster. The `options` key may have the following parameters:

Table 8: code-block:: none Cluster's options keys

Option name	Description
`ssh_to`	Connect to a specific node when performing the `cluster connect` command

A example is shown below:

```
# Clusters must be defined inside clusters key
clusters:
  # This is the cluster name
  my-cool-cluster-1:
    # Additional cluster's options (optional)
    options:
      # When connecting to a cluster, connect to a master-node
      ssh_to: master-node

    # Nodes that must be created when a cluster is instantiated
    nodes:
      # Node named master-node
      master-node:
        type: aws-instance-t2.large   # Instance type that must be created (must match␣
↪instances.yaml name)
        count: 1                      # Number of instances that must be created
        setups:                       # Optionally, list of setups to be executed when␣
↪the master-nodes is created (reference setup configuration names, at setups section)
        - another-example-setup
        - master-setup

      # Node named taskmanager
      slave-nodes:
        type: aws-instance-t2.small  # Instance type that must be created (must match␣
↪instances.yaml name)
        count: 2                     # Number of instances that must be created
        min_count: 1                 # Minimum desired number of instances that must␣
↪effectively be created
        setups:                      # Optionally, list of setups to be executed when␣
↪the slave-nodes is created
        - setup-slave-node
```

Command `cluster start`

Start a cluster given a cluster configuration name. The syntax of the command is shown below

```
clapp cluster start [OPTIONS] CLUSTER_TEMPLATE

  Start cluster based on a cluster template.

  The CLUSTER TEMPLATE is the ID of the cluster configuration at cluster
  configuration files.

Options:
  -n, --no-setup  Do not perform setup  [default: False]
  --help          Show this message and exit.
```

By default, the CLAP's cluster module search for configurations at all `.yaml` files inside `~/.clap/configs/clusters` directory. After cluster is created, the setups are automatically executed. You can omit this phase by using the `--no-setup` option.

An example of the command is shown below, which starts a cluster called `example-cluster`.

```
clapp cluster start example-cluster
```

Note:

- After the cluster's creation a new `cluster_id` will be assigned to it. Thus, multiple clusters with same cluster configuration can be launched Also, all commands will reference to `cluster_id` to perform their actions.
- When a cluster is started its initial configuration is copied to cluster metadata. If you update the cluster configuration while having already started clusters use the `clapp cluster update` command to update the cluster configuration.

Command `cluster setup`

Setup an existing cluster. The command has the following syntax:

```
clapp cluster setup [OPTIONS] CLUSTER_ID

  Perform cluster setup operation at a cluster.

  The CLUSTER_ID argument is the id of the cluster to perform the setup

Options:
  -a, --at TEXT  Stage to start the setup action  [default: before_all]
  --help         Show this message and exit.
```

Given the `cluster_id`, the command will execute all setup phases in all cluster nodes. Some phases of the setup pipeline can be skipped informing at which phase the setup must begin with the `at` parameter. Examples are shown below:

```
clapp cluster setup cluster-faa4017e10094e698aed56bb1f3369f9
clapp cluster setup cluster-faa4017e10094e698aed56bb1f3369f9 --at "before"
```

In the above examples, the first one setups all cluster nodes from `cluster-faa4017e10094e698aed56bb1f3369f9`, the second one setups all nodes, but starting at `before` phase.

Note: The `before_all` and `after_all` phases will be executed at all cluster's nodes, even if setting the `nodes` parameter.

Command `cluster grow`

Start and add a new node to cluster, based on its cluster's node name. The command has the following syntax:

```
clapp cluster grow [OPTIONS] CLUSTER_ID

  Start more nodes at a cluster by cluster node type.

  The CLUSTER_ID argument is the id of the cluster to add more nodes.

Options:
  -n, --node TEXT  Type of node to start. Format: <node_type>:<num>
                   [required]
  -n, --no-setup   Do not perform setup  [default: False]
  --help           Show this message and exit.
```

The `--node` parameter determines how much nodes will be added to cluster. If `--no-setup` is provided no setup phase will be executed.

Command `cluster list`

List all available CLAP's clusters.

Command `cluster alive`

Check if all nodes of the cluster are alive.

Command `cluster resume`

Resume all nodes of the cluster.

Command `cluster pause`

Pause all nodes of the cluster.

Command `cluster stop`

Stop all nodes of the cluster, terminating them (destroying).

Command `cluster list-templates`

List all available cluster templates at `~/clap/configs/clusters` directory.

Command `cluster update`

Update a cluster configuration of an already created cluster. The command's syntax is shown below.

```
clapp cluster update [OPTIONS] CLUSTER_ID

  Perform cluster setup operation at a cluster.

  The CLUSTER_ID argument is the id of the cluster to perform the setup

Options:
  -c, --config TEXT  New cluster config name
  --help             Show this message and exit.
```

If `--config` option is provided, the cluster configuration will be replaced with the informed configuration. Otherwise, the cluster will be updated with the same configuration.

Note: The configurations will be searched in `~/clap/configs/clusters` directory.

Command `cluster connect`

Get a SSH shell to a node of the cluster. Given a `cluster_id` it will try to get an SSH shell to a node type specified in `ssh_to` cluster configuration option. If no `ssh_to` option is informed at cluster's configuration the command will try to connect to any other node that belongs to the cluster.

Command `cluster execute`

Execute a shell command in nodes of the cluster.

Command `cluster playbook`

Execute an Ansible Playbook in nodes of the cluster.

2.4 Roles shared with CLAP

Here are some roles shared by default with CLAP. Setup action is **always** executed when adding a node to a role. Also, variables needed by actions must be passed via `extra` parameter, as keyword value.

2.4.1 Role `commands-common`

This role provide means to execute common known commands in several machines in the role, such as: reboot, copy files to nodes, copy and execute shell scripts, among others. Consider add nodes to this role to quickly perform common commands in several nodes in a row.

The following actions is provided by this role:

- `copy`: Copy a file from the localhost to the remote nodes
- `fetch`: Fetch files from the remote nodes to the localhost
- `reboot`: Reboot a machine and waits it to become available
- `run-command`: Execute a shell command in the remote hosts
- `run-script`: Transfer a script from localhost to remote nodes and execute it in the remote hosts
- `update-packages`: Update packages in the remote hosts

Hosts

No host must be specified by this role.

Action `commands-common copy`

Copy a file from the localhost to the remote nodes

Required Variables

Table 9: `commands-commands copy` action variables

Name	Type	Description
`src`	path	File to be copied to the remote hosts. If the path **is not absolute** (it is relative), it will search in the role's files directory else the file indicated will be copied. If the path is a directory, it will be recursive copied.
`dest`	path	Destination path where the files will be put into at remote nodes

Examples

```
clapp role action commands-common copy --extra src="/home/ubuntu/file" -e dest="~"
```

The above command copy the file at `/home/ubuntu/file` (localhost) the the ~ directory of the nodes.

Action `commands-common fetch`

Fetch files from the remote nodes to the localhost

Required Variables

Table 10: `commands-common fetch` action variables

Name	Type	Description
`src`	path	File to be copied from the remote hosts. If the file is a directory, it will be recursive copied.
`dest`	path	Destination path where the files will be put into (localhost)

Examples

```
clapp role action commands-common fetch --extra src="~/file" --extra dest="/home/ubuntu/
↪fetched_files/"
```

The above command fetch a file at `~/file` directory from the nodes and place at the `/home/ubuntu/fetched_files/` directory of the localhost.

Action `commands-common install-packages`

Install packages in the remote hosts

Required Variables

Table 11: `commands-common install-packages` action variables

Name	Type	Description
`packages`	string	Comma-separated list of packages to install.

Examples

```
clapp role action commands-common install-packages --extra "packages=openmpi-bin,openmpi-
↪common"
```

The above command will install `openmpi-bin` and `openmpi-common` packages to remote hosts

Action `commands-common reboot`

Reboot a machine and waits it to become available

Required Variables

This action does not require any additional variable to be passed.

Examples

```
clapp role action commands-common reboot
```

The command reboot all machines belonging to the `commands-common` role.

Action `commands-common run-command`

Execute a shell command in the remote hosts

Required Variables

Table 12: `commands-common run-command` action variables

Name	Type	Description
`cmd`	string	String with the command to be executed in the nodes
`workdir` (optional)	path	Change into this directory before running the command. If none is passed, home directory of the remote node will be used

Examples

```
clapp role action commands-common run-command --extra cmd="ls"
clapp role action commands-common run-command --extra cmd="ls" -e "workdir=/bin"
```

In the above command (first one) runs the command `ls` in the remote nodes, the second one runs the command `ls` in the remote nodes, after changing to the "/bin" directory

Action `commands-common run-script`

Transfer a script from localhost to remote nodes and execute it in the remote hosts

Required Variables

Table 13: `commands-common run-script` action variables

Name	Type	Description
`src`	string	Shell script file to be executed in the remote nodes. The file will be first copied (from localhost) to the nodes and after will be executed. Note: the script file **must begin** with the bash shebang (`#!/bin/bash`). Also the script filepath must be **absolute** else, if relative path is passed, Ansible search in the role's file directory. The script will be deleted from nodes after execution.
`args` (optional)	string	Command-line arguments to be passed to the script.
`workdir` (optional)	path	Change into this directory before running the command. If none is passed, home directory of the remote node will be used (Path must be absolute for Unix-aware nodes)

Examples

```
clapp role action commands-common run-script --extra src="/home/ubuntu/echo.sh"
clapp role action commands-common run-script --extra src="/home/ubuntu/echo.sh" -e args=
↪"1 2 3"
clapp role action commands-common run-script --extra src="/home/ubuntu/echo.sh" -e args=
↪"1 2 3" -e workdir="/home"
```

The above command (first one) will copy the `/home/ubuntu/echo.sh` script from localhost to the remote nodes and execute it (similar to run `bash -c echo.sh` in the hosts).

The above command (second one) will copy the `/home/ubuntu/echo.sh` script from localhost to the remote nodes and execute it using the arguments "1 2 3" (similar to run `bash -c echo.sh 1 2 3` in the hosts).

The above command (third one) is similar to the second one but will execute the script in the `/home` directory.

Action `commands-common update-packages`

Update packages in the remote hosts

Required Variables

This action does not require any additional variable to be passed

Examples

```
clapp role action commands-common update-packages
```

The above command will update the package list from remote hosts (similar to `apt update` command)

2.4.2 Group ec2-efs

This role setup and mount an network EFS filesystem on AWS provider. The following actions are provided by the role.

- `setup`: Install nfs client
- `mount`: Mount an EFS filesystem
- `umount`: Unmount EC2 File System

Hosts

No hosts must be specified by this role.

Action ec2-efs setup

Install nfs client at remote host. This action is executed when nodes are added to the role.

Required Variables

This action does not require any additional variable to be passed

Action ec2-efs mount

Mount an AWS EC2 EFS filesystem at remote host.

Required Variables

Table 14: `ec2-efs mount` action variables

Name	Type	Description
`efs_mount_ip`	string	Mount IP of the filesystem (see AWS EFS Documentation for more information)
`efs_mount_point` (OPTIONAL)	path	Directory path where the filesystem will be mounted. Default path is: `/efs`
`efs_owner` (OPTIONAL)	string	Name of the user owner (e.g. ubuntu). Default user is the currently logged user
`efs_group` (OPTIONAL)	string	Name of the group owner (e.g. ubuntu). Default group is the currently logged user
`efs_mount_permissions` (OPTIONAL)	string	Permission used to mount the filesystem (e.g. 0644). Default permission is `0744`

Examples

```
clapp role action ec2-efs mount --extra "efs_mount_ip="192.168.0.1" -e "efs_mount_point=/
↪tmp"
```

The above command will mount the EFS Filesystem from `192.168.0.1` it at `/tmp` with `744` permissions (read-write-execute for user and read-only for group and others).

Action `ec2-efs umount`

Unmount the EC2 File System

Required Variables

Table 15: `ec2-efs umount` action variables

Name	Type	Description
`efs_mount_point` (OPTIONAL)	path	Directory path where the filesystem will be mounted. Default path is: `/efs`

Examples

```
clapp role action ec2-efs umount --nodes node-0 --extra efs_mount_point="/efs"
```

The above command will unmount EC2 EFS filesystem at `/efs` directory from `node-0`

2.4.3 Role `spits`

Install spits runtime for the SPITS programming model in nodes, deploy SPITS applications and collect results from execution. The following actions are provided by this role.

- `add-nodes`: This action informs to the job manager node, the public address of all task managers.
- `job-copy`: Copy the results (job directory) from the job manager to the localhost.
- `job-create`: Create a SPITS job in nodes
- `job-status`: Query job manager nodes the status and the metrics of a running SPITS job
- `setup`: Install SPITS runtime and its dependencies at nodes
- `start`: Start a SPITS job at job manager and task manager nodes

Note: For now, shared filesystem is **not supported** for SPITS runtime.

Warning: SPITS application are started using random TCP ports. For now, your security group must allows the communication from/to random IP addresses and ports. So, set inbound and outbound rules from you security group to allow the communication from anywhere to anywhere at any port.

Hosts

This role defines two host types:

- `jobmanager`: Nodes where job manager will be executed for a job
- `taskmanager`: Nodes where task manager will be executed for a job

Typical Workflow

The `spits` role is used to run SPITS applications. For each SPITS application to run, you must create a SPITS job, with an unique Job ID. One node can execute multiple SPITS jobs.

Thus, a typical workflow for usage is:

1. Add job manager desired nodes to `spits/jobmanager` role and task manager desired nodes to `spits/taskmanager`
2. Use `job-create` action the create a new SPITS job in all machines belonging to `spits` role (filter nodes if you want to create a job at selected nodes only).
3. Use `start` action to start the SPITS job manager and SPITS task manager at nodes to run the SPITS job
4. Use the `add-nodes` action to copy public addresses from task managers nodes to the job manager node.
5. Optionally, check the job status using the `job-status` action.
6. When job is finished, use `job-copy` action to get the results.

Action `spits add-nodes`

This action informs to the job manager node, the public address of all task managers.

Required Variables

Table 16: `spits add-nodes` action variables

Name	Type	Description
`jobid`	string	Unique job identifier (must match the job ID used in the `job-create` action)
`PYPITS_PATH` (OPTIONAL)	path	Directory path where the pypits will be installed (default: `${HOME}/pypits/`)
`SPITS_JOB_PATH` (OPTIONAL)	path	Directory path where the spits jobs will be created (default: `${HOME}/spits-jobs/`)

Examples

```
clapp role action spits add-nodes --extra "jobid=my-job-123"
```

The above example will add all task manager addresses, from nodes belonging to the `spits/taskmanager` role to the `spits/jobmanager` nodes at job `my-job-123`. At this point, the job manager nodes recognizes all task managers.

Note:

- This action is not needed if job manager and task managers are running at same node

Action `spits job-copy`

Copy the results (job directory) from the job manager to the localhost

Required Variables

Table 17: `spits job-copy` action variables

Name	Type	Description
`jobid`	string	Unique job identifier (must match the job ID used in the `job-create` action)
`outputdir`	path	Path where job will be copied to
`PYPITS_PATH` (OPTIONAL)	path	Directory path where the pypits will be installed (default: `${HOME}/pypits/`)
`SPITS_JOB_PATH` (OPTIONAL)	path	Directory path where the spits jobs will be created (default: `${HOME}/spits-jobs/`)

Examples

```
clapp role action spits job-copy -e "jobid=my-job-123" -e "outputdir=/home/app-output"
```

The above example will copy the entire job folder (including logs/results) to the localhost and put at `/home/app-output` directory.

Action `spits job-create`

Create a SPITS job in nodes to run an SPITS application. If you are using a shared filesystem, use this action in only one node and set the `SPITS_JOB_PATH` variable to the desired location.

Required Variables

Table 18: `spits job-create` action variables

Name	Type	Description
`jobid`	string	Unique job ID to identify the SPITS job.
`spits_binary`	path	Absolute path to the SPITS binary (at localhost) that will be copied to nodes
`spits_args`	string	Arguments that will be passed to the SPITS binary when executing the SPITS application
`PYPITS_PATH` (OPTIONAL)	path	Directory path where the pypits will be installed (default: `${HOME}/pypits/`)
`SPITS_JOB_PATH` (OPTIONAL)	path	Directory path where the spits jobs will be created (default: `${HOME}/spits-jobs/`)

Examples

```
clapp role action spits job-create --extra "jobid=my-job-123" -e "spits_binary=/home/xxx/
↪spits-app" -e "spits_args=foo bar 10"
```

The above example create the a job called `my-job-123` in all nodes belonging to the `spits` role. The job will execute the SPITS runtime with the binary `/home/xxx/spits-app` (that will be copied from localhost to nodes) with arguments `foo bar 10`.

Action `spits job-status`

Query job manager nodes the status and the metrics of a running SPITS job

Required Variables

Table 19: `spits job-status` action variables

Name	Type	Description
`jobid`	string	Unique job identifier (must match the job ID used in the `job-create` action)
`PYPITS_PATH` (OPTIONAL)	path	Directory path where the pypits will be installed (default: `${HOME}/pypits/`)
`SPITS_JOB_PATH` (OPTIONAL)	path	Directory path where the spits jobs will be created (default: `${HOME}/spits-jobs/`)

Examples

```
clapp role action spits job-status --extra "jobid=my-job-123"
```

The above example query the status of a SPITS job with ID `my-job-123` from nodes belonging to `spits/jobmanager` role. The job status will be displayed at the command output (in green).

Action `spits setup`

Install SPITS runtime and its dependencies at nodes

Required Variables

This action does not require any additional variable to be passed. Optional variables can be passed.

Table 20: `spits setup` action variables

Name	Type	Description
`PYPITS_PATH` (OPTIONAL)	path	Directory path where the pypits will be installed (default: `${HOME}/pypits/`)
`SPITS_JOB_PATH` (OPTIONAL)	path	Directory path where the spits jobs will be created (default: `${HOME}/spits-jobs/`)

Examples

```
clapp role add -n jobmanager:node-0 -n taskmanager:node-1,node-2
```

The above example install SPITS runtime at `node-0`, `node-1` and `node-2`. `node-0` is set as job manager host and nodes `node-1` and `node-2` are set as task manager host.

Action `spits start`

Start a SPITS job at job manager and task manager nodes

Required Variables

Table 21: `spits start` action variables

Name	Type	Description
`jobid`	string	Unique job identifier (must match the job ID used in the `job-create` action)
`jm_args`	string	Arguments to be passed to the job manager SPITS runtime
`tm_args`	string	Arguments to be passed to the task manager SPITS runtime
`PYPITS_PATH` (OPTIONAL)	path	Directory path where the pypits will be installed (default: `${HOME}/pypits/`)
`SPITS_JOB_PATH` (OPTIONAL)	path	Directory path where the spits jobs will be created (default: `${HOME}/spits-jobs/`)

Examples

```
clapp role action spits start --extra "jobid=my-job-123" -e "jm_args=-vv"
```

The above example starts job managers and task managers for job `my-job-123` in nodes belonging to `spits` role. Also, job managers SPITS runtime are executed passing the `-vv` parameter.

Note: The `job-create` action must be used before to create the SPITS job at nodes belonging to `spits` role.

2.5 API Reference

This page contains auto-generated API reference documentation[1].

[1] Created with sphinx-autoapi

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search