CCP Documentation

Release

OpenStack Foundation

May 13, 2017

Contents

CHAPTER 1

User docs

Quick Start

This guide provides a step by step instruction of how to deploy CCP on bare metal or a virtual machine.

Recommended Environment

CCP was tested on Ubuntu 16.04 x64. It will probably work on different OSes, but it's not officialy supported.

CCP was tested on the environment created by Kargo, via fuel-ccp-installer, which manages k8s, calico, docker and many other things. It will probably work on different setup, but it's not officialy supported.

Current tested version of different components are:

Component	Min Version	Max Version	Comment
Kubernetes	1.2.4	1.3.5	1.3.0 to 1.3.3 won't work
Docker	1.10.0	1.12.0	
Calico-node	0.20.0	0.21.0	

Additionaly, you will need to have working kube-proxy, kube-dns and docker registry.

If you don't have a running k8s environment, please check out this guide

Warning: All further steps assume that you already have a working k8s installation.

Deploy CCP

Install CCP CLI

Note: Some commands below may require root permissions

To clone the CCP CLI repo:

```
git clone https://git.openstack.org/openstack/fuel-ccp
```

To install CCP CLI and Python dependencies use:

```
pip install fuel-ccp/
```

Create CCP CLI configuration file:

```
mkdir /etc/ccp
cat > /etc/ccp/ccp.yaml << EOF
builder:
  push: True
registry:
  address: "127.0.0.1:31500"
repositories:
  skip_empty: True
EOF
```

Append default topology and edit it, if needed:

```
cat fuel-ccp/etc/topology-example.yaml >> /etc/ccp/ccp.yaml
```

Append global CCP configuration:

```
cat >> /etc/ccp/ccp.yaml << EOF
configs:
    private_interface: eth0
    public_interface: eth1
    neutron_external_interface: eth2
EOF
```

Make sure to adjust it to your environment, since the network configuration of your environment may be different.

- `private_interface` - should point to eth with private ip address.
- `public_interface` - should point to eth with public ip address (you can use private iface here, if you want to bind all services to internal network)
- `neutron_external_interface` - should point to eth without ip addr (it actually might be non-existing interface, CCP will create it).

Fetch CCP components repos:

```
ccp fetch
```

Build CCP components and push them into the Docker Registry:

```
ccp build
```

Deploy OpenStack:

```
ccp deploy
```

If you want to deploy only specific components use:

```
ccp deploy -c COMPONENT_NAME1 COMPONENT_NAME2
```

For example:

```
ccp deploy -c etcd mariadb keystone
```

Check deploy status

By default, CCP deploying all components into "ccp" k8s namespace. You could set context for all kubectl commands to use this namespace:

```
kubectl config set-context ccp --namespace ccp
kubectl config use-context ccp
```

Get all running pods:

```
kubectl get pod -o wide
```

Get all running jobs:

```
kubectl get job -o wide
```

Note: Deployment is successful when all jobs have "1" (Successful) state.

Deploying test OpenStack environment

Install openstack-client:

```
pip install python-openstackclient
```

openrc file for current deployment was created in the current working directory. To use it run:

```
source openrc-ccp
```

Run test environment deploy script:

```
bash fuel-ccp/tools/deploy-test-vms.sh -a create -n NUMBER_OF_VMS
```

This script will create flavor, upload cirrios image to glance, create network and subnet and launch bunch of cirrios based VMs.

Accessing horizon and nova-vnc

Currently, we don't have any external proxy (like Ingress), so, for now, we have to use k8s service "nodePort" feature to be able to access internal services.

Get nodePort of horizon service:

```
kubectl get service horizon -o yaml | awk '/nodePort: / {print $NF}'
```

Use external ip of any node in cluster plus this port to access horizon.

Get nodePort of nova-novncproxy service:

```
kubectl get service nova-novncproxy -o yaml | awk '/nodePort: / {print $NF}'
```

Take the url from Horizon console and replace "nova-novncproxy" string with an external IP of any node in cluster plus nodeport from the service.

Cleanup deployment

To cleanup your environment run:

```
ccp cleanup
```

This will delete all VMs created by OpenStack and destroy all neutron networks. After it's done it will delete all k8s pods in this deployment.

Monitoring and Logging with StackLight

This section provides information on deploying StackLight, the monitoring and logging system for CCP.

Warning: StackLight requires Kubernetes 1.4 or higher, and its deployment will fail with Kubernetes 1.3 and lower. So before deploying StackLight make sure you use an appropriate version of Kubernetes.

Overview

StackLight is composed of several components. Some components are related to logging, and others are related to monitoring.

The "logging" components:

- `heka` – for collecting logs
- `elasticsearch` – for storing/indexing logs
- `kibana` – for exploring and visualizing logs

The "monitoring" components:

- `stacklight-collector` – composed of Snap and Hindsight for collecting and processing metrics
- `influxdb` – for storing metrics as time-series
- `grafana` – for visualizing time-series

For fetching the StackLight repo (`fuel-ccp-stacklight`) and building the StackLight Docker images please refer to the *Quick Start* section as StackLight is not different from other CCP components for that matter. If you followed the *Quick Start* the StackLight images may be built already.

The StackLight Docker images are the following:

- `ccp/cron`
- `ccp/elasticsearch`
- `ccp/grafana`
- `ccp/heka`
- `ccp/hindsight`
- `ccp/influxdb`
- `ccp/kibana`

Deploy StackLight

The StackLight components are regular CCP components, so the deployment of StackLight is done through the CCP CLI like any other CCP component. Please read the *Quick Start* section and make sure the CCP CLI is installed and you know how to use it.

StackLight may be deployed together with other CCP components, or independently as a separate deployment process. You may also want to deploy just the "logging" components of StackLight, or just the "monitoring" components. Or you may want to deploy all the StackLight components at once.

In any case you will need to create StackLight-related roles in your CCP configuration file (e.g. `/etc/ccp/ccp.yaml`) and you will need to assign these roles to nodes.

For example:

```
nodes:
  node1:
    roles:
      - stacklight-backend
      - stacklight-collector
  node[2-3]:
    roles:
      - stacklight-collector
roles:
  stacklight-backend:
    - influxdb
    - grafana
  stacklight-collector:
    - stacklight-collector
```

In this example we define two roles: `stacklight-backend` and `stacklight-collector`. The role `stacklight-backend` is assigned to `node1`, and it defines where `influxdb` and `grafana` will run. The role `stacklight-collector` is assigned to all the nodes (`node1`, `node2` and `node3`), and it defines where `stacklight-collector` will run. In most cases you will want `stacklight-collector` to run on every cluster node, for node-level metrics to be collected for every node.

With this, you can now deploy `influxdb`, `grafana` and `stacklight-collector` with the following CCP command:

```
ccp deploy -c influxdb grafana stacklight-collector
```

Here is another example, in which both the "monitoring" and "logging" components will be deployed:

```
nodes:
  node1:
    roles:
      - stacklight-backend
      - stacklight-collector
  node[2-3]:
    roles:
      - stacklight-collector
roles:
  stacklight-backend:
    - influxdb
    - grafana
    - elasticsearch
    - kibana
  stacklight-collector:
    - stacklight-collector
    - heka
    - cron
```

And this is the command to use to deploy all the StackLight services:

```
ccp deploy -c influxdb grafana elasticsearch kibana stacklight-collector heka cron
```

To check the deployment status you can run:

```
kubectl --namespace ccp get pod -o wide
```

and check that all the StackLight-related pods have the `RUNNING` status.

Accessing the Grafana and Kibana interfaces

As already explained in *Quick Start* CCP does not currently include an external proxy (such as Ingress), so for now the Kubernetes `nodePort` feature is used to be albe to access services such as Grafana and Kibana from outside the Kubernetes cluster.

This is how you can get the node port for Grafana:

```
$ kubectl get service grafana -o yaml | awk '/nodePort: / {print $NF}'
31124
```

And for Kibana:

```
$ kubectl get service kibana -o yaml | awk '/nodePort: / {print $NF}'
31426
```

Developer docs

How To Contribute

General info

1. Bugs should be filed on launchpad, not GitHub.
2. Please follow OpenStack Gerrit Workflow to contribute to CCP.
3. Since CCP has multiple Git repositories, make sure to use Depends-On Gerrit flag to create cross repository dependencies.

Useful documentation

- Please follow our *Quick Start* guide to deploy your environment and test your changes.
- Please refer to *CCP Docker images guide*, while making changes to Docker files.
- Please refer to *Application definition contribution guide*, while making changes to `service/*` files.

CCP Docker images guide

This guide covers CCP specific requirements for defining Docker images.

Docker files location

All docker files should be located in `docker/<component_name>` directory, for example:

```
docker/horizon
docker/keystone
```

The docker directory may contain multiple components.

Docker directory structure

Each docker directory should contain a `Dockerfile.j2` file. Dockerfile.j2 is a file which contains Docker build instructions in a Jinja2 template format. You can add additional files, which will be used in Dockerfile.j2, but only Dockerfile.j2 can be a Jinja2 template in this directory.

Dockerfile format

Please refer to the official Docker documentation which covers the Dockerfile format. CCP has some additional requirements, which is:

1. Use as few layers as possible. Each command in Dockerfile creates a layer, so make sure you're grouping multiple RUN commands into one.
2. If it's possible, please run container from the non-root user.
3. If you need to copy some scripts into the image, please place them into the `/opt/ccp/bin` directory.
4. Only one process should be started inside container. Do not use runit, supervisord or any other init systems, which will allow to spawn multiple processes in container.
5. Do not use CMD and ENTRYPOINT commands in Dockerfile.j2.
6. All OpenStack services should use `openstack-base` parent image in FROM section. All non-OpenStack services should use `base-tools` parent image in FROM section.

Here is an example of valid Dockerfile.j2: Keystone Dockerfile

Supported Jinja2 variables

Only specific variables can actually be used in Dockerfile.j2:

1. `namespace` - Used in the FROM section, renders into image namespace, by default into `ccp`.
2. `tag` - Used in the FROM section, renders into image tag, by default into `latest`.
3. `maintainer` - Used in the MAINTAINER section, renders into maintainer email, by default into "MOS Microservices <mos-microservices@mirantis.com>"
4. `copy_sources` - Used anywhere in the Dockerfile. please refer to corresponding documentation section below.
5. Additionaly, you could use variables with software versions, please refer to *Application definition contribution guide* for details.

copy_sources

The CCP CLI provides additional feature for Docker images creation, which will help to use git repositories inside Dockerfile, it's called `copy_sources`.

This feature uses configuration from `service/files/defaults.yaml` from the same repository or from global config, please refer to *Application definition contribution guide* for details.

Testing

After making any changes in docker directory, you should test it via build and deploy.

To test building, please run:

```
ccp build -c <component_name>
```

For example:

```
ccp build -c keystone
```

Make sure that image is built without errors.

To test the deployment, please build new images using the steps above and after run:

```
ccp deploy
```

Please refer to *Quick Start* for additional information.

Application definition contribution guide

This guide covers CCP specific DSL, which is used by the CCP CLI to populate k8s objects.

Application definition files location

All application definition files should be located in the `service/` directory, as a `component_name.yaml` file, for example:

```
service/keystone.yaml
```

All templates, such as configs, scripts, etc, which will be used for this service, should be located in `service/<component_name>/files`, for example:

```
service/files/keystone.conf.j2
```

All files inside this directory are Jinja2 templates. Default variables for these templates should be located in `service/component_name/files/defaults.yaml` inside the `configs` key.

Understanding globals and defaults config

There are three config locations, which the CCP CLI uses:

1. `Global defaults` - fuel_ccp/resources/defaults.yaml in `fuel-ccp` repo.
2. `Component defaults` - service/files/defaults.yaml in each component repo.
3. `Global config` - Optional. Set path to this config via "–deploy-config /path" CCP CLI arg.

Before deployment, CCP will merge all these files into one dict, using the order above, so "component defaults" will override "global defaults" and "global config" will override everything.

Global defaults

This is project wide defaults, CCP keeps it inside fuel-ccp repository in `fuel_ccp/resources/defaults.yaml` file. This file defines global variables, that is variables that are not specific to any component, like eth interface names.

Component defaults

Each component repository could contain a `service/files/defaults.yaml` file with default config for this component only.

Global config

Optional config with global overrides for all services. Use it only if you need to override some defaults.

Config keys types

Each config could contain 3 keys:

- `configs`
- `versions`
- `sources`
- `nodes`

Each key has its own purpose and isolation, so you have to add your variable to the right key to make it work.

configs key

Isolation:

- Used in service templates files (service/files/).
- Used in application definition file service/component_name.yaml.

Allowed content:

- Any types of variables allowed.

Example:

```
configs:
  keystone_debug: false
```

So you could add "{{ keystone_debug }}" variable to you templates, which will be rendered into "false" in this case.

versions key

Isolation:

- Used in Dockerfile.j2 only.

Allowed content:

- Only versions of different software should be kept here.

For example:

```
versions:
 influxdb_version: "0.13.0"
```

So you could add this to influxdb Dockerfile.j2:

```
curl https://dl.influxdata.com/influxdb/releases/influxdb_{{ influxdb_version }}_amd64.deb
```

sources key

Isolation:

- Used in Dockerfile.j2 only.

Allowed content:

- This key has a restricted format, examples below.

Remote git repository example:

```
sources:
  openstack/keystone:
    git_url: https://github.com/openstack/keystone.git
    git_ref: master
```

Local git repository exaple:

```
sources:
  openstack/keystone:
    source_dir: /tmp/keystone
```

So you could add this to Dockerfile.j2:

```
{{ copy_sources("openstack/keystone", "/keystone") }}
```

CCP will use the chosen configuration, to copy git repository into Docker container, so you could use it latter.

network_topology key

Isolation:

- Used in service templates files (service/files/).

Allowed content:

- This key is auto-created by entrypoint script and populated with container network topology, based on the following variables: `private_interface` and `public_interface`.

You could use it to get the private and public eth IP address. For example:

```
bind = network_topology["private"]["address"]
listen = network_topology["public"]["address"]
```

nodes and roles key

Isolation:

- Not used in any template file, only used by the CCP CLI to create a cluster topology.

Allowed content:

- This key has a restricted format, example of this format can be found in `fuel-ccp` git repository in `etc/topology-example.yaml` file.

“CCP_*” env variables

Isolation:

- Used in service templates files (service/files/).

Allowed content:

- This variables are created from the application definition `env` key. Only env keys which start with “CCP_” will be passed to config hash.

This is mainly used to pass some k8s related information to container, for example, you could use it to pass k8s node hostname to container via this variable:

Create env key:

```
env:
  - name: CCP_NODE_NAME
    valueFrom:
      fieldRef:
        fieldPath: spec.nodeName
```

Use this variable in some config:

```
{{ CCP_NODE_NAME }}
```

Application definition language

Please refer to *Application definition language* for detailed description of CCP DSL syntax.

Application definition language

There is a description of current syntax of application definition framework.

Application definition template

```
service:
    name: service-name
    ports:
        - internal-port:external-port
    daemonset: true
    host-net: true
    containers:
        - name: container-name
          image: container-image
          probes:
              readiness: readiness.sh
              liveness: liveness.sh
          volumes:
              - name: volume-name
                type: host
                path: /path
          pre:
              - name: service-bootstrap
                dependencies:
                    - some-service
                    - some-other-service
                type: single
                command: /tmp/bootstrap.sh
                files:
                    - bootstrap.sh
                user: user
              - name: db-sync
                dependencies:
                    - some-dep
                command: some command
                user: user
          daemon:
              dependencies:
                  - demon-dep
              command: daemon.sh
              files:
                  - config.conf
              user: user
          post:
              - name: post-command
                dependencies:
                    - some-service
                    - some-other-service
                type: single
                command: post.sh
                files:
                    - config.conf

files:
    config.conf:
        path: /etc/service/config.conf
        content: config.conf.j2
        perm: "0600"
        user: user
    bootstrap.sh:
        path: /tmp/bootstrap.sh
        content: bootstrap.sh.j2
        perm: "0755"
```

Parameters description

service

Name	Description	Required	Schema	Default
name	Name of the service.	true	string	
containers	List of containers under multi-container pod	true	container array	
ports	k8s Service will be created if specified (with NodePort type for now) Only internal or both internal:external ports can be specified	false	internal-port: external-port array	
daemonset	Create DaemonSet instead of Deployment	false	boolean	false
hostnet		false	boolean	false

container

Name	Description	Required	Schema	Default
name	Name of the container. It will be used to track status in etcd	true	string	
image	Name of the image. registry, namespace, tag will be added by framework	true	string	
probes	Readiness, liveness or both checks can be defined. Exec action will be used for both checks	false	dict with two keys: liveness: cmd readiness: cmd	
volumes		false	volume array	
pre	List of commands that need to be executed before daemon process start	false	command array	
daemon		true	command	
post	The same as for "pre" except that post commands will be executed after daemon process has been started	false	command array	
env	An array of environment variables defined in kubernetes way.	false	env array	

volume

Name	Description	Required	Schema	Default
name	Name of the volume	true	string	
type	host and empty-dir type supported for now	true	one of: [”host”, “empty-dir”]	
path	Host path that should be mounted (only if type = “host”)	false	string	
mount-path	Mount path in container	false	string	path
readOnly	Mount mode of the volume	false	bool	False

command

Name	Description	Required	Schema	Default
name	Name of the command. Required only for *pre* and *post* with type *single*	–	string	
command		true	string	
dependencies	These keys will be polled from etcd before commands execution	false	string array	
type	type: single means that this command should be executed once per openstack deployment. For commands with type: single Job object will be created type: local (or if type is not specified) means that command will be executed inside the same container as a daemon process.	false	one of: [”single”, “local”]	local
files	List of the files that maps to the keys of files dict. It defines which files will be rendered inside a container	false	file keys array	
user		false	string	

files

Name	Description	Required	Schema	Default
Name of the file to refer in files list of commands		false	file array	

file

Name	Description	Required	Schema	Default
path	Destination path inside a container	true	string	
content	Name of the file under {{ service_repo }}/service/files directory. This file will be rendered inside a container and moved to the destination defined with path	true	string	
perm		false	string	
user		false	string	

CHAPTER 3

Indices and tables

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