FUJITSU

FUJITSU Software ServerView Cloud Load Control V1.0

Cluster Management Guide

J2UL-2089-01ENZ0(00)
November 2015

Contents

About this Manual

This manual describes how to create and manage clusters in an OpenStack environment with FUJITSU Software ServerView Cloud Load Control - hereafter referred to as Cloud Load Control (CLC). In addition, it describes how to get started with deploying applications to the cluster nodes.

This manual is structured as follows:

Chapter	Description
Introduction on page 7	Introduces Kubernetes, its architecture and basic concepts.
Setting up a Cluster on page 10	Describes how to create and provision a cluster using the CLC Horizon plugin.
Getting Started with Using a Cluster on page 17	Describes how to prepare the Kubernetes command-line interface, and provides a walkthrough of building a simple application and deploying it to a cluster.
Deleting a Cluster on page 37	Describes how to de-provision and delete a cluster.
Glossary on page 38	Definitions of terms used in this manual.

Intended Audience

This manual is directed to people who want to use CLC for provisioning and managing clusters in an OpenStack environment (cluster operators) and for deploying applications to a cluster. It assumes that you are familiar with the CLC concepts as described in the *Overview* manual.

In addition, the following knowledge is required:

- Working with OpenStack environments
- Developing containerized Web applications

Notational Conventions

This manual uses the following notational conventions:

Add	The names of graphical user interface elements like menu options are shown in boldface.
`init`	System names, for example command names and text that is entered from the keyboard, are shown in Courier font.
`<variable>`	Variables for which values must be entered are enclosed in angle brackets.
`[option]`	Optional items, for example optional command parameters, are enclosed in square brackets.
`one | two`	Alternative entries are separated by a vertical bar.
`{one | two}`	Mandatory entries with alternatives are enclosed in curly brackets.

Abbreviations

This manual uses the following abbreviations:

CentOS	Community ENTerprise Operating System
CLC	Cloud Load Control
HOT	Heat orchestration template
IaaS	Infrastructure as a Service
KVM	Kernel-based Virtual Machine
OSS	Open Source software
PaaS	Platform as a Service
SaaS	Software as a Service

Available Documentation

The following documentation on CLC is available:

- *Overview* - A manual introducing CLC. It is written for everybody interested in CLC.
- *Installation Guide* - A manual written for OpenStack operators who install CLC in an existing OpenStack environment.
- *Cluster Management Guide* - A manual written for cluster operators who use CLC for provisioning and managing clusters, and for cluster users who deploy applications to a cluster.

Related Web References

The following Web references provide information on open source offerings integrated with CLC:

- *OpenStack*: Documentation of OpenStack, the underlying platform technology.
- *OpenStack Horizon*: Documentation of the OpenStack Horizon dashboard.
- *Kubernetes*: Information on Kubernetes, the core of CLC.
- *Docker*: Information on the container technology used by Kubernetes.

More detailed Web references provided in this manual are subject to change without notice.

Export Restrictions

Exportation/release of this document may require necessary procedures in accordance with the regulations of your resident country and/or US export control laws.

Trademarks

Trademarks of other companies are used in this manual only to identify particular products or systems.

LINUX is a registered trademark of Linus Torvalds.

UNIX is a registered trademark of the Open Group in the United States and in other countries.

The OpenStack® Word Mark and OpenStack Logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation in the United States and other countries and are used with the OpenStack Foundation's permission.

Red Hat is a trademark or a registered trademark of Red Hat Inc. in the United States and other countries.

Kubernetes is a registered trademark of Google Inc.

Docker is a trademark of Docker Inc.

ServerView is a registered trademark of FUJITSU LIMITED.

Other company names and product names are trademarks or registered trademarks of their respective owners.

Other company and product names in this manual are trademarks or registered trademarks of their respective owners.

Copyrights

Copyright (c) FUJITSU LIMITED 2015

All rights reserved, including those of translation into other languages. No part of this manual may be reproduced in any form whatsoever without the written permission of FUJITSU LIMITED.

High Risk Activity

The Customer acknowledges and agrees that the Product is designed, developed and manufactured as contemplated for general use, including without limitation, general office use, personal use, household use, and ordinary industrial use, but is not designed, developed and manufactured as contemplated for use accompanying fatal risks or dangers that, unless extremely high safety is secured, could lead directly to death, personal injury, severe physical damage or other loss (hereinafter "High Safety Required Use"), including without limitation, nuclear reaction control in nuclear facility, aircraft flight control, air traffic control, mass transport control, medical life support system, missile launch control in weapon system. The Customer shall not use the Product without securing the sufficient safety required for the High Safety Required Use. In addition, FUJITSU (or other affiliate's name) shall not be liable against the Customer and/or any third party for any claims or damages arising in connection with the High Safety Required Use of the Product.

1 Introduction

Cloud Load Control (CLC) is an out-of-the-box solution for companies that are looking for a preassembled and enterprise-ready distribution of tools for provisioning and managing clusters and containers on top of OpenStack-based cloud computing platforms. CLC automates the setup and operation of a workload management system in OpenStack based on modern Linux container and clustering technology: *Kubernetes* and *Docker*.

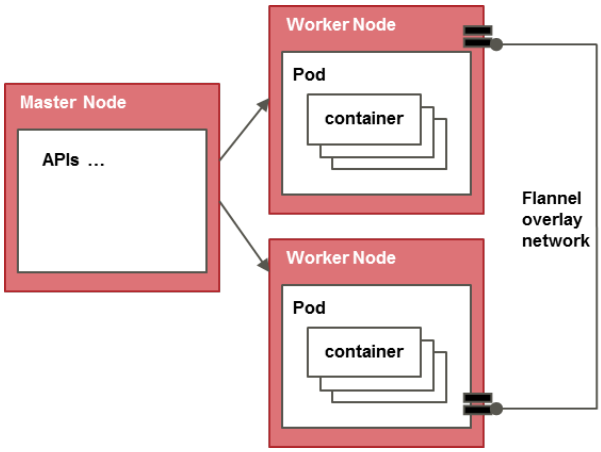
CLC comes with a plugin for the OpenStack dashboard (Horizon) for automatically setting up, managing, and deleting clusters.

1.1 Kubernetes Basics

Kubernetes is an open source orchestration system for Docker containers. Using Kubernetes, you can deploy and manage containers from one system (a Kubernetes master node) to run on other systems (Kubernetes worker nodes). The preassembled base image for the nodes provided by CLC is CentOS Atomic host.

While Docker defines the container format and builds and manages individual containers, Kubernetes is the orchestration tool for deploying and managing sets of Docker containers.

Using the OpenStack dashboard extension of CLC, the setup of a Kubernetes cluster is completely automated. CLC comes with Heat templates for automatically orchestrating and booting a cluster. The master node is used to deploy one or more containers in what is referred to as a pod. The master pushes that pod to a Kubernetes worker node on which the containers run. The worker node provides the run-time environment for the containers (CentOS Atomic host). Containers isolate applications from each other and the underlying infrastructure while providing a layer of protection for the applications.



When creating a cluster with CLC, a master node as well as a specified number of worker nodes are automatically set up. All nodes are provisioned with the preassembled CentOS Atomic host image. A Flannel overlay network is created. Flannel is a virtual network that provides a subnet to each host for use with the container runtime environments. Kubernetes assumes that each pod has a unique, routable IP address inside the cluster. The overlay network reduces the complexity of port mapping.

Below you can find descriptions of basic Kubernetes concepts. For details, refer to the *Kubernetes* documentation.

Pods

A pod is a collection of one or more containers that are symbiotically grouped.

In the Kubernetes model, containers are not deployed to specific hosts, but pod definitions describe the containers that are to be run. When you create a pod, you declare that you want the containers in it to be running. If the containers are not running, Kubernetes will continue to (re-)create them for you in order to drive them to the desired state. This process continues until you delete the pod.

Pod definitions are stored in configuration files (in `yaml` or `json` format).

Volumes

A container file system only lives as long as the container does. To achieve a more persistent storage, you declare a volume as part of your pod, and mount it to a container.

Labels

In Kubernetes, labels are used for organizing pods in groups. Labels are key-value pairs that are attached to each object in Kubernetes. Label selectors can be passed along to the Kubernetes API server to retrieve a list of objects which match the specified label selectors.

Labels are a core concept for two additional building blocks of Kubernetes: replication controllers and services.

Replication Controller

Kubernetes implements "self-healing" mechanisms, such as auto-restarting, re-scheduling, and replicating pods by means of a replication controller. This Kubernetes API object defines a pod in terms of a template that the system then instantiates as the specified number of pods. The replicated set of pods may constitute an entire application, a micro-service, or a layer in a multi-tier application.

As soon as the pods are created, the system continuously monitors their health and that of the machines they are running on. In the case a pod fails, for example, due to a machine failure or a software problem, the replication controller automatically creates a new pod on a healthy machine. In this way, the number of pods specified in the template is maintained at the desired level of replication.

Multiple pods from the same or different applications can share the same machine. Note that a replication controller is needed also for a single, non-replicated pod if you want it to be re-created when the pod or its machine fails.

Services

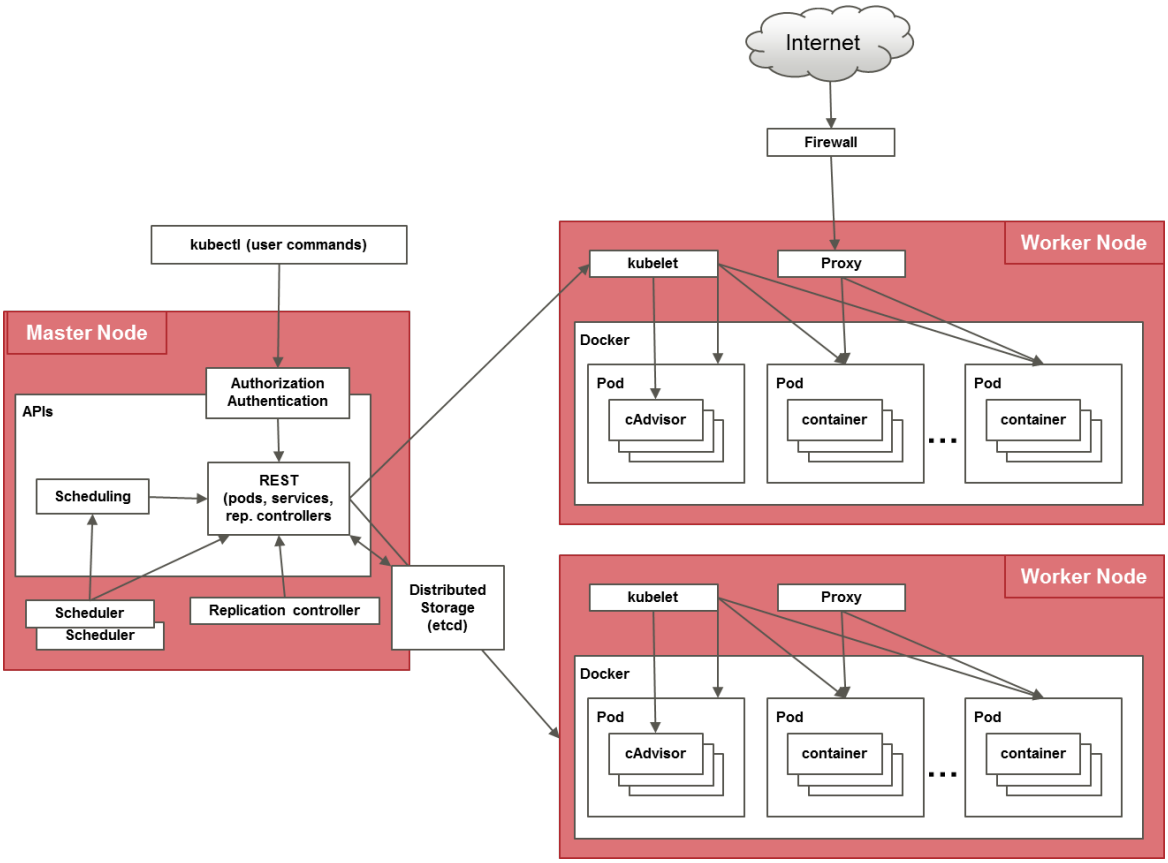
Once you have a replicated set of pods, you need an abstraction that enables connectivity between the layers of your application. For example, if you have a replication controller managing your backend jobs, you do not want to have to reconfigure your frontends whenever you re-scale your backends.

A service basically combines an IP address and a label selector to form a simple, static access point for connecting to a micro-service in your application (running in one or several containers in one or several pods).

When created, each service is assigned a unique IP address. This address does not change while the service is alive. Pods can be configured to talk to the service. In this way, the pods know that communication to the service will be automatically load-balanced to a pod which is a member of the set identified by the label selector in the service.

1.2 Kubernetes Architecture

The following picture illustrates the basic architecture of Kubernetes:



The master node exposes the Kubernetes APIs, through which tasks are defined. The master node is where you run commands (`kubectl`) to manage containers and deploy workloads.
All persistent states are stored in `etcd`, so that configuration data is reliably available.
The Kubernetes API mainly processes REST operations, validates them, and updates the corresponding objects in `etcd`. The scheduler binds unscheduled pods to worker nodes. The replication controller ensures the health of the cluster.

An agent (a `kubelet`) on each worker node instance polls for instructions or state changes (persisted in `etcd`) and acts to execute them on the host. Each worker node also runs a simple network proxy. cAdvisor (Container Advisor) provides cluster users an overview of the resource usage and performance characteristics of their containers. It is a daemon that collects, aggregates, processes, and exports information about running containers.

For further details, refer to the *Kubernetes* documentation.

2 Setting up a Cluster

This chapter describes the prerequisites and preparations as well as the actual steps of provisioning a cluster using the OpenStack dashboard.

2.1 Prerequisites and Preparation

The following prerequisites must be fulfilled before you can provision a cluster using the OpenStack dashboard and start working with it:

- You must have access to a fully functional OpenStack platform. Access as a user with administrative rights is required.
- The OpenStack operator must have installed CLC to this platform.
- Internet access without proxy settings, in particular access to *Google Container Registry* and *Docker Hub*. This is required because when an application is provisioned in a Kubernetes cluster, Kubernetes tries to download a Docker container image.
- When you access the OpenStack dashboard for the first time, you need to create a key pair to enable SSH access to the cluster nodes.
- It is recommended to create a custom flavor in OpenStack which defines at least the minimum resource configuration for the provisioning of a cluster and its nodes.
- CLC supports the following Web browsers:
  - Google Chrome 45.0
  - Microsoft Internet Explorer 11.0
  - Mozilla Firefox 40.0

2.1.1 Creating a Key Pair for SSH Access

Key pairs are SSH credentials that are injected into an OpenStack instance when it is launched. Each OpenStack project should have at least one key pair.

On the host which you use to access the OpenStack dashboard, generate a key pair as follows:

```
ssh-keygen -t rsa -f cloud.key
```

This command generates a private key (`cloud.key`) and a public key (`cloud.key.pub`).

Then proceed as follows:

1. Log in to the OpenStack dashboard as an administrator.
2. Go to **Project > Orchestration > Clusters**.
3. Click **Create Cluster**.
4. On the **Access & Security** tab, click the plus (+) sign.
5. In the **Key Pair Name** field, enter a name for the key pair. It is recommended to use letters, numbers, and hyphens only.
6. Copy the contents of the public key file you just generated, and paste it into the **Public Key** field of the **Import Key Pair** page.
7. Proceed with the provisioning of your cluster as described in *Creating a Cluster* on page 12.

When an instance is created in OpenStack, it is automatically assigned a fixed IP address in the network to which it is assigned. This IP address is permanently associated with the node (instance) until it is terminated. However, in addition to the fixed IP address, you also assign

floating IP addresses to the cluster nodes when you create a cluster. Unlike fixed IP addresses, floating IP addresses are able to have their associations modified at any time, regardless of the state of the nodes involved.

After launching a node, you can log in to it using the private key:

```
ssh -i cloud.key minion@<floating IP of node>
```

2.1.2 Defining a Custom Flavor

Flavors are virtual hardware templates in OpenStack, defining sizes for RAM, disks, number of cores, and other resources. The default OpenStack installation provides several flavors. It is recommended to create a flavor which defines the minimum configuration for the creation of a Kubernetes cluster with 2 CentOS Atomic host image nodes.

Proceed as follows:

1. Log in to the OpenStack dashboard as an administrator.
2. Go to **Admin > System > Flavors**.
3. Click **Create Flavor**.
4. In the **Create Flavor** window, on the **Flavor Information** tab, enter or select the parameters for the flavor.
   - **Name**: Enter a name for the flavor.
   - **ID**: Leave this field blank. OpenStack generates the ID.
   - **VCPUs**: Enter the number of virtual CPUs to use. The minimum value is `1`.
   - **RAM (MB)**: Enter the amount of RAM to use in megabytes. The minimum recommended value is `1024`.
   - **Root Disk (GB)**: Enter the amount of disk space in gigabytes to use for the root (/) partition. The minimum recommended value is `10`.
   - **Ephemeral Disk (GB)**: Enter the amount of disk space in gigabytes to use for the ephemeral partition. If unspecified, the value is `0` by default. Ephemeral disks provide local disk storage for the life cycle of a VM instance. When a VM is terminated, all data on the ephemeral disk is lost. Ephemeral disks are not included in any snapshots.
   - **Swap Disk (MB)**: Enter the amount of swap space in megabytes to use. If unspecified, the default is `0`.
5. On the **Flavor Access** tab, you can control access to the flavor: You can specfiy which OpenStack projects can use the flavor by moving projects from the **All Projects** column to the **Selected Projects** column.

   Only projects in the **Selected Projects** column can use the flavor. If there are no projects in this column, all projects can use the flavor.
6. Click **Create Flavor**.

The flavor is added to the list from which you can choose when creating a cluster.

The following table lists the OpenStack default flavors and your newly created one:

Flavor	VCPUs	Disk (in GB)	RAM (in MB)
m1.tiny	1	1	512
<YourFlavor>	1	10	1024
m1.small	1	20	2048

Flavor	VCPUs	Disk (in GB)	RAM (in MB)
m1.medium	2	40	4096
m1.large	4	80	8192
m1.xlarge	8	160	16384

Note: The CentOS Atomic host image also requires RAM and system resources (about 750 MB). You must take this into account when you create nodes. When you create a cluster, for example, with a flavor defining 1024 MB of RAM, you only have about 275 MB left for running applications in the pods on your nodes. Make sure to have enough resources available when setting up a cluster.

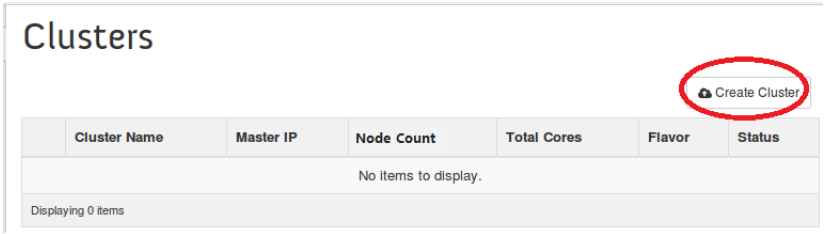
2.2 Creating a Cluster

CLC comes with a plugin for the OpenStack dashboard (*Horizon*) which adds a panel for cluster management to the dashboard.

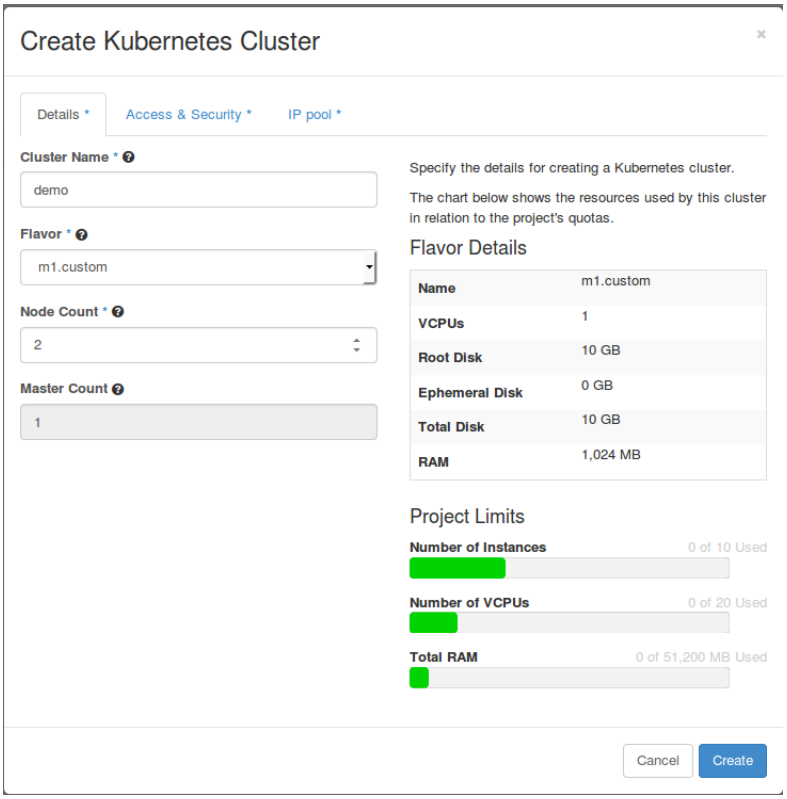
To set up a cluster using the dashboard:

1. Log in to the OpenStack dashboard as an administrator.
2. Go to **Project > Orchestration > Clusters**.

   The panel for cluster management provided by CLC is opened:



3. Click **Create Cluster**:



4. On the **Details** tab, enter the following information:
   - **Cluster Name**

     Name of the cluster. The name must be unique within the OpenStack project, start with a letter and only contain alphanumeric characters, numbers, underscores, periods, and hyphens. The cluster name may consist of a maximum of 200 characters.
   - **Flavor**

     Flavors are virtual hardware templates in OpenStack, defining sizes for RAM, disks, number of cores, and other resources. The default OpenStack installation provides several flavors. It is recommended to create and use a flavor which defines the minimum configuration for the creation of a Kubernetes cluster with CentOS Atomic host image nodes:
     - 1 vCPU (virtual CPU) must be present for each node in the cluster. In your OpenStack environment, this vCPU must not be shared with other VMs.
     - 1024 MB of RAM virtual machine memory.
     - 10 GB of virtual root disk size.

     Flavors which are not large enough for cluster environments are filtered out from the list of flavors that can be chosen.

     Refer to *Defining a Custom Flavor* on page 11 for information on how to create a flavor.
   - **Node Count**

Number of worker nodes to be created in the cluster to be provisioned. As an example, specify `2`.

As you type, in the **Project Limits** area, you see the resources the cluster will use in relation to the quota set for the current project.

Quota are operational limits. CLC uses the default OpenStack project quota. Without sensible quota, a single project could exceed all available physical resources, and thus make the whole OpenStack environment unstable. The colors have the following meaning:

- Blue: The resources already used by the number of instances in OpenStack defined in the project's quota before the cluster is created.
- Green: The resources that will be used as soon as the cluster is created.
- Red: The number of nodes you specified will exceed the resources defined in the default quota of the OpenStack project. Thus you either need to increase the project quota, or specify a lower number of nodes in the **Node Count** field.

A Kubernetes cluster always consists of a master node and one or several worker nodes. The master node is added to the amount of nodes that you specify in the **Node Count** field.

- **Master Count** (read-only)

  The number of Kubernetes master nodes. Currently, there can only be 1 master.

5. On the **Key Pair for Access & Security** tab, select a key pair or create a new one by clicking the plus (+) sign.

   Key pairs define how you log in to a cluster node after it has been launched. Cloud images support public key authentication rather than conventional user name/password authentication. Key pairs are SSH credentials that are injected into an OpenStack instance when it is launched. Each project should have at least one key pair.

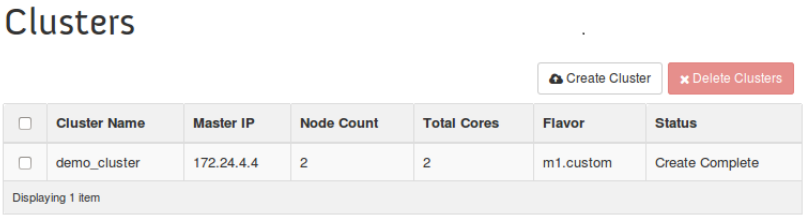
   When using the dashboard for the first time, you need to create a key pair for the project. Refer to *Creating a Key Pair for SSH Access* on page 10 for details on how to create a key pair.

6. On the **IP Pool** tab, select a network.

   An OpenStack administrator can set up several networks and subnets. The network defines the range of floating IP addresses the nodes in the cluster to be provisioned can be assigned.

7. Click **Create**.

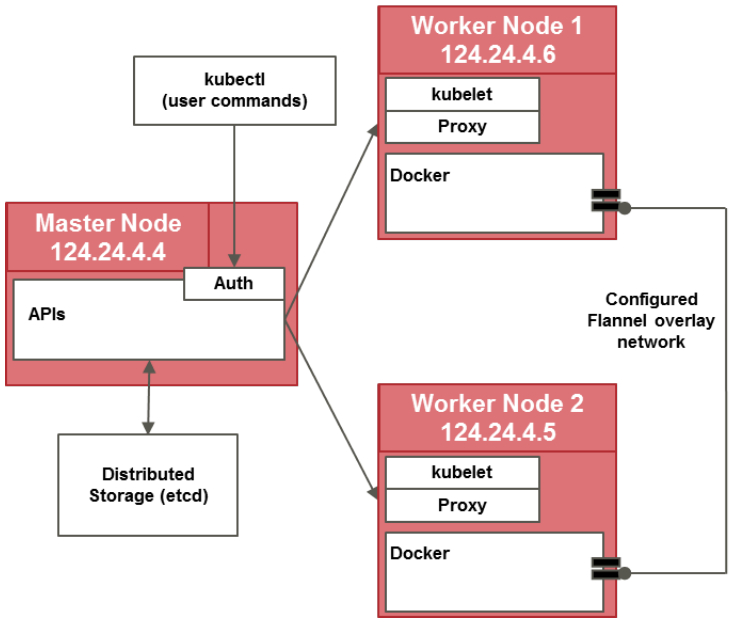
CLC starts provisioning the Kubernetes cluster. The panel for cluster management provided by CLC now looks, for example, as follows:



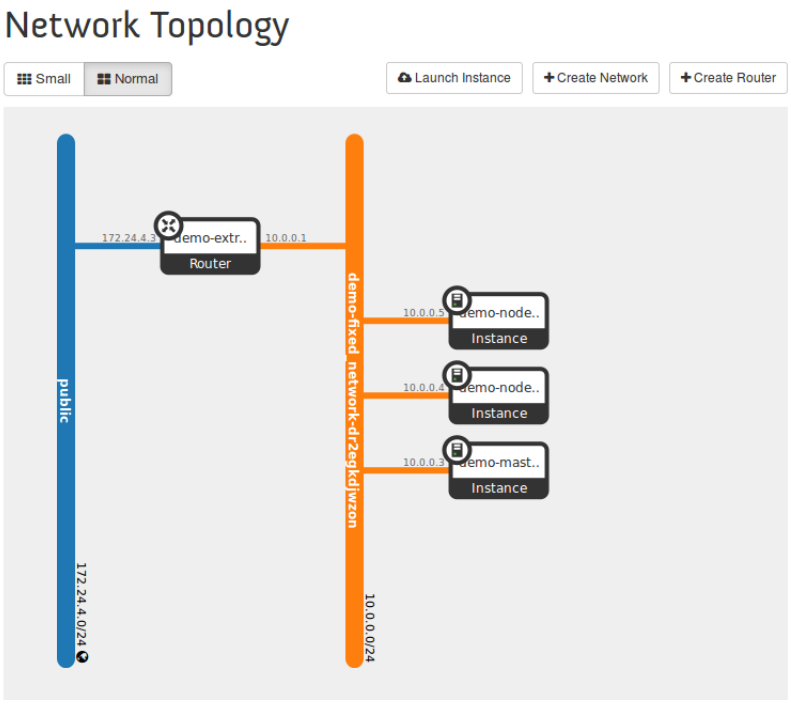
- **Cluster Name**: The name of the cluster as specified when you created the cluster.
- **Master IP**: IP address of the master node.
- **Node Count**: Number of provisioned worker nodes.
- **Total Cores**: Number of vCPU cores used by the provisioned worker nodes.

- **Flavor**: The flavor used for the cluster as selected when you created the cluster.
- **Status**: The status of the cluster. This is, for example, `Create in Progress` when Kubernetes starts to provision the cluster, or `Create Complete` when the provisioning of the cluster is completed.

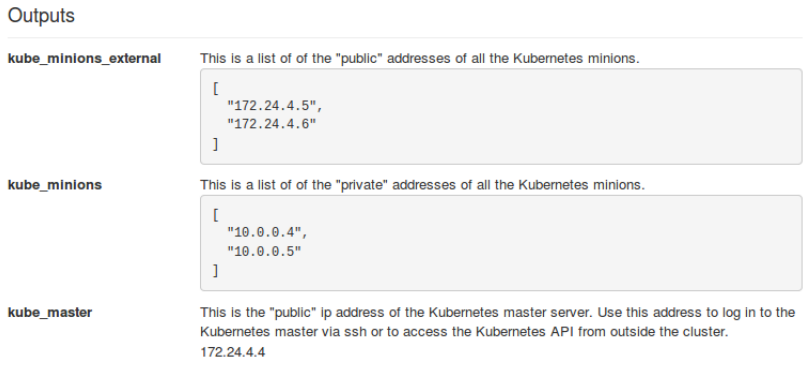
When the provisioning of the cluster is finished, a stack representing the cluster is created in OpenStack. The CLC Heat orchestration template boots the Kubernetes cluster with the specified number of worker nodes and one master node:



The networking between the containers is configured using Flannel, a network fabric layer that provides each node with an individual subnet for container communication. You can see this configuration in the OpenStack dashboard (**Project > Network > Network Topology**):



The newly created stack represents the cluster in OpenStack. When you take a look at the stack (**Project > Orchestration > Stacks**), you see the floating IP addresses of the master and the worker nodes:



The floating IP addresses can also be viewed on an instance level (**Project > Compute > Instances**).

Note:	You can always check the status of the cluster and its health with **Project > Orchestration > Stacks**.

3 Getting Started with Using a Cluster

This chapter describes how to install the Kubernetes command-line interface (CLI), which controls the Kubernetes cluster manager. The guestbook tutorial walks you through building a simple, multi-tier Web application, a guestbook, and deploying it to a cluster.

3.1 Preparing the Kubernetes CLI

To deploy and manage applications on Kubernetes, you use the Kubernetes command-line tool, `kubectl`. It lets you inspect your cluster resources, create, delete, and update components, and much more. You will use it to look at your new cluster and bring up example applications.

So, before you can start with the deployment of workloads to the cluster nodes, you need to make the Kubernetes command-line interface (CLI) available. The binaries of the CLI are available in the `<InstDir>/cli` folder.

To be able to execute the `kubectl` commands from any folder, link `kubectl` to your `/usr/local/bin` folder, for example:

```
sudo ln -s <InstDir>/cli/kubectl /usr/local/bin/kubectl
```

To manage your Kubernetes environment, and to deploy workloads into the containers, you run the `kubectl` commands. `kubectl` controls the Kubernetes cluster manager.

For a complete list of the parameters that are supported, you can execute the following command:

```
kubectl --help
```

For details on the parameters for `kubectl`, you can also refer to *Google's Kubernetes CLI description*.

Note: If the `kubectl` command cannot be executed, the executable bit might not be set for the file. You can check this, for example, by using the `ls -l` command. If the executable bit is not set, you can set it with the following command:

```
chmod +x kubectl
```

For samples of how to use the `kubectl` commands, refer to *Guestbook Tutorial* on page 17.

3.2 Guestbook Tutorial

This tutorial walks you through building a simple, multi-tier Web application, a guestbook, and deploying it into a Kubernetes cluster. The guestbook allows visitors to enter text in a log on a Web page, and to see the last few entries.

In the example, you will see how you can set up a Web service on an IP address that is accessible from outside the cluster, and how to implement load balancing to a set of replicated servers backed by replicated Redis worker nodes. Redis is an open source, advanced key-value cache and store. It is often referred to as a data structure server since keys can contain strings, hashes, lists, sets, and bitmaps. You can run atomic operations on these types.

The example explains a number of important container concepts. It shows how to define pods, define a set of replicated pods, use labels for selection, and set up services.

Kubernetes uses the concept of a pod, which is a group of dependent containers that share networking and a filesystem. Pods are created within a Kubernetes cluster through specification

files that are pushed to the Kubernetes API. Kubernetes then schedules pod creation on a node in the cluster.

The example consists of:

- A replicated PHP Web frontend.
- A Redis master (for storage) and a replicated set of Redis workers.

The Web frontend interacts with the Redis master via Javascript Redis API calls.

CLC provides the files used in this example in the `<InstDir>/examples/guestbook` folder. `<InstDir>` is the folder where the CLC installation package is located.

3.2.1 Prerequisites

Make sure that the following prerequisites are fulfilled before you start with the steps in this tutorial:

- The example requires a running Kubernetes cluster. Provision a cluster with two nodes using the OpenStack dashboard. For details, refer to *Creating a Cluster* on page 12.
- The example requires Kubernetes API information from the master node. To achieve this, set the following environment variable:

  ```
  export K8S_APISERVER=http://<master floating ip>:8080
  ```

  You can look up the floating IP address of the master node in the OpenStack dashboard (**Project > Compute > Instances**).
- The commands provided in the tutorial assume that you are working in the following folder:

  `<InstDir>/examples/guestbook`
- To be able to execute the Kubernetes command-line commands (`kubectl`) from the above folder, it is assumed that you have prepared `kubectl` as described in *Preparing the Kubernetes CLI* on page 17.
- The cluster worker nodes must have Internet access, in particular, access to *Docker Hub*.
- The cluster worker nodes must have Internet access, in particular access to *Google Container Registry* and *Docker Hub*.

3.2.2 Step One: Create and Start the Redis Master

The first step is to create and start up pod for the Redis master. A replication controller is used to create the pod. Even though it is a single pod, the replication controller is still useful for monitoring cluster health and restarting the pod, if required.

Use the file `redis-master-controller.json` which describes a single pod running a Redis key-value server in a container:

```
{
   "kind":"ReplicationController",
   "apiVersion":"v1",
   "metadata":{
      "name":"redis-master",
      "labels":{
         "name":"redis-master"
      }
   },
   "spec":{
      "replicas":1,
      "selector":{
         "name":"redis-master"
      },
      "template":{
         "metadata":{
            "labels":{
               "name":"redis-master"
            }
         },
         "spec":{
            "containers":[
               {
                  "name":"master",
                  "image":"redis",
                  "ports":[
                     {
                        "containerPort":6379
                     }
                  ]
               }
            ]
         }
      }
   }
}
```

Although the Redis server runs just with a single replicated pod, a replication controller is used to enforce that exactly one pod keeps running: In the event of the node going down, the replication controller ensures that the Redis master gets restarted on a healthy node.

For details on pod configuration, refer to the ***Kubernetes*** documentation.

Create and start the Redis master pod:

```
kubectl --server=$K8S_APISERVER create -f redis-master-controller.json
```

View the replication controllers of your cluster:

```
kubectl --server=$K8S_APISERVER get rc
```

Verify that the `redis-master` pod is running:

```
kubectl --server=$K8S_APISERVER get pods
```

A single `redis-master` pod is listed, showing the name of the instance the pod is running on. It may take about 30 seconds for the pod to be placed on an instance, and for its status to change from `Pending` to `Running`.

3.2.3 Step Two: Create and Start the Service for the Redis Master

A Kubernetes service is a named load balancer which proxies traffic to one or more containers. A service provides a way to group pods under a common access policy.

The proxying is achieved by using the `labels` metadata defined for the `redis-master` pod above. As mentioned already, there is only one master in Redis. However, we still want to create a service for the master node because this provides a deterministic way of routing the single master using an elastic IP address:

The implementation of the service creates a virtual IP address which clients can access and which is transparently proxied to the pods in a service. The services in a Kubernetes cluster are discoverable inside other containers via environment variables. Services find the containers to load-balance based on the pod labels.

Each node runs a `kube-proxy` process which programs IP table rules to access service IP addresses and redirect them to the correct backends. This provides a highly available load-balancing solution with low performance overhead by balancing client traffic from a node on that same node.

In the guestbook example, you have a set of pods (here it is just one, the `redis-master`) that each expose port `6379` and have the `"name=redis-master"` label. When setting up the service for the Redis master, you communicate to the service the pods to proxy based on this pod label.

Look at the file `redis-master-service.json` and use it for creating a service for the Redis master:

```
{
   "kind":"Service",
   "apiVersion":"v1",
   "metadata":{
      "name":"redis-master",
      "labels":{
         "name":"redis-master"
      }
   },
   "spec":{
      "ports": [
        {
          "port":6379,
          "targetPort":6379
        }
      ],
      "selector":{
         "name":"redis-master"
      }
   }
}
```

The `selector` field of the service determines which pods will receive the traffic sent to the service. The configuration specifies that we want this service to point to pods labeled with `"name"="redis-master"`. The `port` and `targetPort` information defines the port the service proxy will run at.

Create and start the service for the Redis master:

```
kubectl --server=$K8S_APISERVER create -f redis-master-service.json
```

Check the services:

```
kubectl --server=$K8S_APISERVER get services
```

The list includes the `redis-master` service:

```
NAME          LABELS            SELECTOR          IP(S)           PORT(S)
redis-master  name=redis-master name=redis-master 10.254.145.149 6379/TCP
```

All pods can now see the Redis master running on port `6379`. A service can map an incoming port to any `targetPort` in the backend pod.

`targetPort` defaults to `port` if it is omitted in the configuration. For simplicity's sake, it is omitted in the subsequent configurations.

The traffic flow from the worker nodes to the master can be described as follows:

- A Redis slave connects to the `port` on the Redis master service.
- Traffic is forwarded from the service `port` on the node where the service is running to the `targetPort` on the pod in a node the service listens to.

Thus, once created, the service proxy on each worker node is configured to set up a proxy on the specified port (`6379`).

3.2.4 Step Three: Create and Start the Replicated Redis Worker Pods

Although the Redis master is a single pod, the Redis workers are set up as replicated pods. In Kubernetes, a replication controller is responsible for managing multiple instances of a replicated pod. The replication controller ensures that a specified number of pod replicas are running at any one time. A configuration file defines the set of pods, and the replication controller is responsible for achieving this set. If there are too many pods, it will kill some. If there are too few, which may occur because of pod or node failure, it will start more.

In the sample below, two replicas are to be created. Use the file `redis-slave-controller.json` to set up the two Redis worker pods:

```
{
   "kind":"ReplicationController",
   "apiVersion":"v1",
   "metadata":{
      "name":"redis-slave",
      "labels":{
         "name":"redis-slave"
      }
   },
   "spec":{
      "replicas":2,
      "selector":{
         "name":"redis-slave"
      },
      "template":{
         "metadata":{
            "labels":{
               "name":"redis-slave"
            }
         },
         "spec":{
            "containers":[
               {
                  "name":"slave",
                  "image":"kubernetes/redis-slave",
                  "ports":[
                     {
                        "containerPort":6379
                     }
                  ]
               }
            ]
         }
      }
   }
}
```

The configuration file above defines a pod template to be used by the replication controller, and indicates that 2 replicas of this pod be maintained.

Create and start the Redis worker pods:

```
kubectl --server=$K8S_APISERVER create -f redis-slave-controller.json
```

View the replication controllers of your cluster:

```
kubectl --server=$K8S_APISERVER get rc
```

Now you can list the pods in the cluster to verify that the master and workers are running:

```
kubectl --server=$K8S_APISERVER get pods
```

You will see a single Redis master and two Redis worker pods.

3.2.5 Step Four: Create and Start the Redis Worker Service

Just like for the master, a service to proxy connections to the Redis worker nodes is to be set up. In addition to discovery, the service provides transparent load balancing to Web application clients.

The service specification for the workers is in the file `redis-slave-service.json`:

```
{
   "kind":"Service",
   "apiVersion":"v1",
   "metadata":{
      "name":"redis-slave",
      "labels":{
         "name":"redis-slave"
      }
   },
   "spec":{
      "ports": [
        {
          "port":6379,
          "targetPort":6379
        }
      ],
      "selector":{
         "name":"redis-slave"
      }
   }
}
```

This time, the selector for the service is `"name"="redis-slave"`. This identifies the pods running the Redis worker nodes. It is also helpful to set labels on the service itself, as shown in this example, to make it easy to locate the service using the `kubectl --server=$K8S_APISERVER get services -l "label=redis-slave"` command.

Create and start the service using the service specification described above:

```
$ kubectl --server=$K8S_APISERVER create -f redis-slave-service.json
```

List the services. They now include the `redis-slave` service:

```
kubectl --server=$K8S_APISERVER get services
```

Sample output:

```
NAME          LABELS            SELECTOR          IP(S)           PORT(S)
redis-master  name=redis-master name=redis-master 10.127.252.216 6379/TCP
redis-slave   name=redis-slave  name=redis-slave  10.0.72.62      6379/TCP
```

3.2.6 Step Five: Create and Start the Guestbook Frontend Pods

Now that you have the backend of your guestbook up and running, start its frontend pods as Web servers.

This tutorial uses a simple PHP frontend. It is configured to talk to either the Redis worker or master services, depending on whether the request is a read (worker) or a write (master) request.The PHP frontend exposes a simple AJAX interface, and serves an angular-based

UX (user experience). Like the Redis worker pods, it is a replicated service instantiated by a replication controller.

The PHP frontend can now leverage write requests to the load-balancing Redis worker nodes, which can be highly replicated.

The replication controller and its pod template are described in the file `frontend-controller.json`:

```
{
   "kind":"ReplicationController",
   "apiVersion":"v1",
   "metadata":{
      "name":"frontend",
      "labels":{
         "name":"frontend"
      }
   },
   "spec":{
      "replicas":3,
      "selector":{
         "name":"frontend"
      },
      "template":{
         "metadata":{
            "labels":{
               "name":"frontend"
            }
         },
         "spec":{
            "containers":[
               {
                  "name":"php-redis",
                  "image":"kubernetes/example-guestbook-php-redis",
                  "ports":[
                     {
                        "containerPort":80
                     }
                  ]
               }
            ]
         }
      }
   }
}
```

The `frontend-controller.json` file specifies 3 replicas of the server. Using this file, you can start your guestbook frontends with the following command:

```
kubectl --server=$K8S_APISERVER create -f frontend-controller.json
```

View the replication controllers for your cluster:

```
kubectl --server=$K8S_APISERVER get rc
```

A list of three controllers is shown:

```
CONTROLLER    CONTAINER(S) IMAGE(S)                    SELECTOR          REPLICAS
frontend      php-redis    example-guestbook-php-redis name=frontend     3
redis-master  master       redis                       name=redis-master 1
redis-slave   slave        redis-slave                 name=redis-slave  2
```

Once the servers are up, you can list the pods in the cluster to verify that they are all running:

```
kubectl --server=$K8S_APISERVER get pods
```

You will see a single Redis master pod, two Redis workers, and three frontend pods, for example:

```
NAME                  READY     STATUS    RESTARTS   AGE
frontend-1ltts        1/1       Running   0          8m
frontend-j68k0        1/1       Running   0          8m
frontend-vojnc        1/1       Running   0          8m
redis-master-bsaxn    1/1       Running   0          20m
redis-slave-2co2l     1/1       Running   0          14m
redis-slave-7favm     1/1       Running   0          14m
```

The PHP file implementing the guestbook application is downloaded from Docker Hub. The code for the PHP service looks like this:

```
<?

set_include_path('.:/usr/share/php:/usr/share/pear:/vendor/predis');

error_reporting(E_ALL);
ini_set('display_errors', 1);

require 'predis/autoload.php';

if (isset($_GET['cmd']) === true) {
  header('Content-Type: application/json');
  if ($_GET['cmd'] == 'set') {
    $client = new Predis\Client([
      'scheme' => 'tcp',
      'host'   => getenv('REDIS_MASTER_SERVICE_HOST') ?:
 getenv('SERVICE_HOST'),
      'port'   => getenv('REDIS_MASTER_SERVICE_PORT'),
    ]);
    $client->set($_GET['key'], $_GET['value']);
    print('{"message": "Updated"}');
  } else {
    $read_port = getenv('REDIS_MASTER_SERVICE_PORT');

    if (isset($_ENV['REDISSLAVE_SERVICE_PORT'])) {
      $read_port = getenv('REDISSLAVE_SERVICE_PORT');
    }
    $client = new Predis\Client([
      'scheme' => 'tcp',
      'host'   => getenv('REDIS_MASTER_SERVICE_HOST') ?:
 getenv('SERVICE_HOST'),
      'port'   => $read_port,
    ]);

    $value = $client->get($_GET['key']);
    print('{"data": "' . $value . '"}');
  }
} else {
  phpinfo();
} ?>
```

3.2.7 Step Six: Create and Start a Guestbook Web Service

As with the other pods, a service is to be created to group the guestbook frontend pods.

A Kubernetes service provides the following mechanisms for load balancing:

- A stable address to ephemeral pods (by proxying the traffic to the right location).
- A simple round robin-based load balancing mechanism to a group of pods. Services can be used for internal communication between pods as well as for external communication. This is achieved by setting the type to `NodePort` as in the example below.

The service specification for the guestbook Web service is in the file `frontend-service.json`:

```
{
   "kind":"Service",
   "apiVersion":"v1",
   "metadata":{
      "name":"frontend",
      "labels":{
         "name":"frontend"
      }
   },
   "spec":{
      "ports": [
        {
          "port":80,
          "targetPort":80,
          "nodePort":30001
        }
      ],
      "type": "NodePort",
      "selector":{
         "name":"frontend"
      }
   }
}
```

Create and start the service:

```
kubectl --server=$K8S_APISERVER create -f frontend-service.json
```

List the services. They now include the `frontend` service, for example:

```
NAME          LABELS            SELECTOR          IP(S)           PORT(S)
frontend      name=frontend     name=frontend     10.254.50.106   80/TCP
redis-master  name=redis-master name=redis-master 10.254.145.149 6379/TCP
redis-slave   name=redis-slave  name=redis-slave  10.254.190.3    6379/TCP
```

You have exposed your service on an external port on all nodes in your cluster. If you want to expose this service to the external Web, you may need to set up firewall rules for the service ports (`tcp:30001`) to serve traffic. Refer to the Kubernetes documentation for further details on frontend services.

3.2.8 Externally Access the Guestbook Service

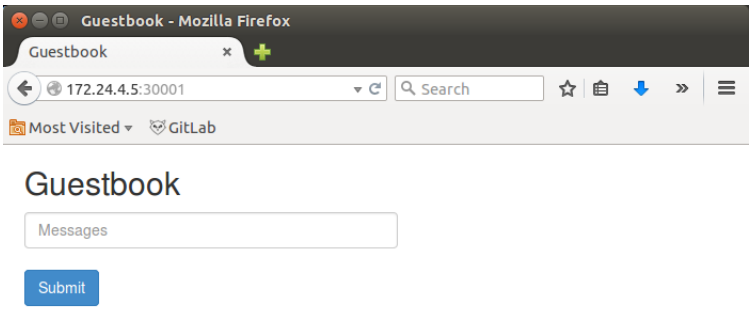
The pods that you have set up can be reached through the frontend service, but you will notice that the `10.254.50.106` IP address (the IP address of the frontend service) cannot be reached from outside the Kubernetes cluster.

In our example, the way of exposing the frontend service to an external IP address is defined by the `NodePort` parameter in the `type` field. This type defines how the service can be accessed. In the example, the access takes place using a cluster IP and by exposing the service on a port on each node of the cluster (the same port on each node).

This means that each node will proxy that specific node port into the guestbook service. The same port number is used on every node. This port is reported in the guestbook service's `nodePort` field (`30001`). The system allocates this port.

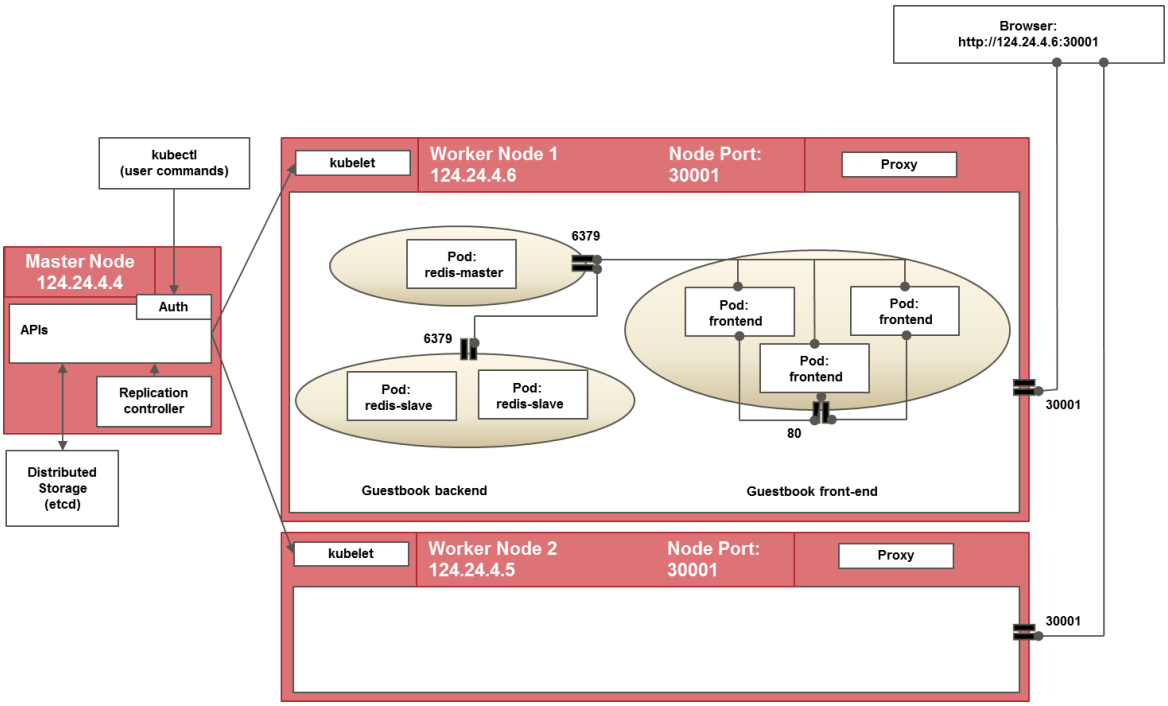
Access to the guestbook is thus possible using the following address:

```
http://<any worker node floating ip>:30001
```



3.2.9 Summary

The following illustration shows the components of your cluster after having completed the tutorial. For the sake of simplicity, the pods are shown to be on the first worker node only. Usually, they will be distributed evenly to all nodes in the cluster.



3.2.10 Cleanup the Guestbook

If you are in a live Kubernetes cluster and want to remove the replication controllers and service - and, by doing so, remove the pods you created while working through this tutorial, you can use a script like the one below:

```
kubectl --server=$K8S_APISERVER stop -f redis-master-controller.json
kubectl --server=$K8S_APISERVER stop -f redis-slave-controller.json
kubectl --server=$K8S_APISERVER stop -f frontend-controller.json
kubectl --server=$K8S_APISERVER delete -f redis-master-service.json
kubectl --server=$K8S_APISERVER delete -f redis-slave-service.json
kubectl --server=$K8S_APISERVER delete -f frontend-service.json
```

When you are done with your cluster altogether, you can remove it using the OpenStack dashboard (**Project > Orchestration > Clusters > Delete Cluster**).

3.3 Advanced Topics

3.3.1 Scaling Your Application

When load on your application grows or shrinks, it is easy to manually scale with `kubectl`. For instance, to increase the number of Redis worker replicas from 2 to 3, execute the following command:

```
kubectl --server=$K8S_APISERVER scale rc redis-slave --replicas=3
```

Note: The CentOS Atomic host image also requires RAM and system resources (about 750 MB). You must take this into account when you scale your pods. When you create a cluster, for example, with a flavor defining 1024 MB of RAM, you only have about 275 MB left for running applications in your pods. Make sure to have enough resources available when increasing the number of pods.

3.3.2 Resource Limits

In Kubernetes, there is no information on how many resources a container requires when it has been launched, nor is there information on how many additional resources the container might require once it is running. For this reason, Kubernetes distributes pods evenly to all available cluster nodes. This can lead to unevenly distributed resource consumption.

When creating a pod, you can optionally specify how much CPU and memory (RAM) each container running in the pod needs. When such resource limits are defined fro containers, the Kubernetes scheduler is able to make better decisions about which nodes to place pods on. CPU and memory are each a resource type. A resource type has a base unit. CPU is specified in units of cores. Memory is specified in units of bytes.

Take a look at the `redis-master-controller-with-resource-limits.json` file for a sample pod definition which sets the resource limits to half a CPU core and 500 MB of RAM:

```
{
   "kind":"ReplicationController",
   "apiVersion":"v1",
   "metadata":{
      "name":"redis-master",
      "labels":{
         "name":"redis-master"
      }
   },
   "spec":{
      "replicas":1,
      "selector":{
         "name":"redis-master"
      },
      "template":{
         "metadata":{
            "labels":{
               "name":"redis-master"
            }
         },
         "spec":{
            "containers":[
               {
                  "name":"master",
                  "image":"kubernetes/example-guestbook-php-redis",
                  "resources": {
                        "limits": {
                           "memory":"500Mi",
                           "cpu":"500m"
                  },
                  "ports":[
                     {
                        "containerPort":6379
                     }
                  ]
               }
            ]
         }
      }
   }
}
```

To apply the resource limits definition to the Redis master controller created in the guestbook tutorial, proceed as follows:

1. Stop the Redis master replication controller:

```
kubectl --server=$K8S_APISERVER stop rc redis-master
```

2. Create a new replication controller that includes the resource limits definition:

```
kubectl --server=$K8S_APISERVER create \
 -f \
 examples/guestbook/redis-master-controller-with-resource-limits.json
```

3. Test that the resource limits definition is applied. For example, scale the Redis master controller, and specify that 5 replicas are to be created:

```
kubectl --server=$K8S_APISERVER scale redis-master --replicas=5
```

   Assuming that the master node has a capacity of 1 GB of RAM and one CPU core, only two pods will be running; the remaining 3 replicas are in a pending state.

   The Kubernetes scheduler bookkeeping service substracts the specified container resource limits from the node capacity. In the sample above, the Docker process is physically run with an upper limit of 500 MB of RAM and an unlimited amount of swap space; in addition, the Docker process is half-weighted in the CPU scheduling.

Be aware of the following when defining resource limits:

- The RAM limit must be above the maximum memory consumption of the application. Otherwise, memory swapping starts which can degrade the overall system performance.
- Even if the complete capacity of a node is distributed to pods for which resource limits are defined, you can still schedule pods without resource limits.

3.3.3 Recovering from Overload

If you have provisioned a cluster and put too much load on its nodes, you will notice that deployed applications will slow down and eventually fail. In such a case, you can react by implementing health checks so that the system can recover automatically (see below).

If you ignore the error situation and put even more load on the nodes, the core services of your operating system may eventually start to fail. The operating system can no longer recover safely and must be rebooted.

3.3.4 Health Checking

By default, replication controllers only check whether a pod is still alive. If not, the pod is restarted automatically. This is the simplest form of health-checking, also called "process level health checking": The Kubelet agent on the worker nodes constantly asks the Docker daemon if a container process is still running, and if not, the container process is started. In the example you have run in this tutorial, this health checking was actually already enabled. It is automatically on for every single container that runs in Kubernetes.

However, in many cases this low-level health checking is not sufficient. The checks of the replication controller are only useful in case an application crashes. If an application does no longer respond, for example, due to a deadlock or a memory leakage, the checks cannot resolve the problem.

Kubernetes supports user-implemented application health-checks. These checks are performed by the Kubelet agent to ensure that your application is operating correctly. Using a management system to perform periodic health checking and repair of your application provides for a system outside of your application itself which is responsible for monitoring the application and taking action to fix it.

It is important that the system is outside of the application itself, because if your application fails and the health checking agent is part of your application, it may fail as well. Again, the health check monitor is the Kubelet agent running on the worker nodes.

The health checks are configured in the `livenessProbe` section of your container configuration. The probes can be HTTP or shell commands. You can also specify an `initialDelaySeconds`, which is a grace period from when the container is started to when health checks are performed.

Below you find an example configuration for the Redis master pod with an HTTP health check. This sample requires that the application provides an HTTP interface. The sample is available in the `frontend-controller-with-health-check.json` file:

```
{
   "kind":"ReplicationController",
   "apiVersion":"v1",
   "metadata":{
      "name":"frontend",
      "labels":{
         "name":"frontend"
      }
   },
   "spec":{
      "replicas":3,
      "selector":{
         "name":"frontend"
      },
      "template":{
         "metadata":{
            "labels":{
               "name":"frontend"
            }
         },
         "spec":{
            "containers":[
               {
                  "name":"php-redis",
                  "image":"kubernetes/example-guestbook-php-redis",
                  "livenessProbe": {
                       "httpGet": {
                        "path": "/",
                        "port": 80
                  },
                  "initialDelaySeconds": 30,
                  "timeoutSeconds": 1
              },
                  "ports":[
                     {
                        "containerPort":80
                     }
                  ]
               }
            ]
         }
      }
   }
}
```

To apply the health checking to the frontend replication controller created in the guestbook tutorial, proceed as follows:

1. Stop the frontend replication controller:

```
kubectl --server=$K8S_APISERVER stop rc frontend
```

2. Create a new replication controller that includes the health checking:

```
kubectl --server=$K8S_APISERVER create \
   -f examples/guestbook/frontend-controller-with-health-check.json
```

3. You can simulate the restart behavior of the pod by changing the path to a URL that does not exist (for example, `"path"="/non-existing"`). You will then see that the Kubelet service restarts the pods:

```
kubectl --server=$K8S_APISERVER get pods
```

The output looks something like this:

```
NAME                  READY     STATUS    RESTARTS   AGE
frontend-1ltts        1/1       Running   1          8m
frontend-j68k0        1/1       Running   1          8m
frontend-vojnc        1/1       Running   1          8m
redis-master-bsaxn    1/1       Running   0          20m
redis-slave-2co2l     1/1       Running   0          14m
redis-slave-7favm     1/1       Running   0          14m
```

3.3.5 Lifecycle of Pods: Container Environment

Containers can be managed by the Kubelet agent running on the Kubernetes worker nodes. Contrary to the Kubernetes cluster API (`kubectl`) which provides an API for creating and managing containers, the Kubernetes container environment provides access to cluster information to the containers.

This information allows for building applications that are "cluster aware".

The Kubernetes container environment also defines several "hooks" that are exposed to hook handlers defined as part of individual containers. The hook handlers are optional. Container hooks help to facilitate the building of reliable and scalable applications in a Kubernetes cluster.

Containers that participate in a cluster lifecycle become "cluster native".

Hooks are different from events provided by Docker and other systems. We differentiate between **output events** and **input hooks**. Events are output from the container and represent log entries, this means that they provide a log of what has already happened (history). Hooks provide real-time notification about situations that are actually are happening in a specific situation. The hooks are triggered by Kubernetes after the start of a container or before shutdown.

Below you find an example for using hooks. Container hooks provide information to the container about events in its management lifecycle. For example, a container receives a `PostStart` hook immediately after the container has been started.

Have a look at the `redis-master-controller-with-lifecycle.json` file provided with the guestbook tutorial:

```
{
"kind":"ReplicationController",
"apiVersion":"v1",
"metadata":{
   "name":"redis-master",
   "labels":{
      "name":"redis-master"
   }
},
"spec":{
   "replicas":1,
   "selector":{
      "name":"redis-master"
   },
   "template":{
      "metadata":{
         "labels":{
            "name":"redis-master"
         }
      },
      "spec":{
         "containers":[
            {
               "name":"master",
               "image":"redis",
               "lifecycle":{
                     "postStart": {
                        "exec": {
                            "command" : ["touch","/hello-world.txt"]
                        }
                     }
               },

               "ports":[
                  {
                     "containerPort":6379
                  }
                  ]
               }
            ]
         }
      }
   }
}
```

This example creates a dummy file during the start process of the container. This file does not change any functionality. It is used for demonstration purposes only.

To apply the hook to the Redis master replication controller, proceed as follows:

1. Stop the `redis-master` replication controller.

```
kubectl --server=$K8S_APISERVER stop rc redis-master
```

2. Create a new replication controller that includes a lifecycle hook:

```
kubectl --server=$K8S_APISERVER create \
   -f examples/guestbook/redis-master-controller-with-lifecycle.json
```

3. Verify that the `hello-world.txt` file exists:

```
kubectl --server=$K8S_APISERVER get pods
```

This command retrieves the generated ID for the `redis-master` (for example, `4f23c`).

Check for the file with the following command:

```
kubectl --server=$K8S_APISERVER exec redis-master-4f23c ls /
```

where `-4f23c` is the generated ID of the Redis master pod.

3.3.6 Connecting to Applications: Port Forwarding

The Kubernetes command-line interface (`kubectl`) offers the possibility to forward connections to a local port on a pod (`port-forward`). In the guestbook tutorial, for example, the pods that you have set up can be reached by proxying the port specified with `nodePort` into the guestbook service. The same port number is used on every node.

In the example below you see how to use `kubectl port-forward` for connecting to a Redis database. This connection may be useful, for example, for database debugging.

Proceed as follows:

1. Verify that the Redis master is listening on port `6379`:

```
kubectl get pods redis-master \
  -t='{{(index (index .spec.containers 0).ports 0).containerPort}}\
  {{"\n"}}'
```

The result should be `6379`.

2. Forward the port `6379` on the local workstation to the port `6379` of the Redis master pod:

```
kubectl port-forward -p redis-master 6379:6379
```

3. Verify that the connection is successful, run the following command on the local workstation:

```
redis-cli
```

The result should be something like this:

```
127.0.0.1:6379> ping
PONG
```

4. You can now proceed with debugging the Redis database from your local workstation.

3.3.7 Security Settings

The Kubernetes API is served by the Kubernetes `apiserver` process. Typically, there is one `apiserver` process running on a single Kubernetes master node.

With this version of CLC, the Kubernetes API server serves HTTP requests on port `8080`. This means that calls to the API server are open; no authentication is required for accessing the API server. It is assumed that you are operating CLC in a trusted environment.

If you want to manually protect your clusters and implement secure communication, you need to spend some thoughts on security issues. For example:

- Ensure that port `6443` is used for communication inside your clusters, and port `443` for calls from outside the cluster.
- Specify security group settings in OpenStack.
- Set certificates for the Kubernetes API server.
- Define an authentication mode: Kubernetes can be set up to use one of the following authentication methods for authenticating users for API calls:
  - Client certificate authentication
  - Token authentication
  - Basic authentication

  In general, it is recommended to use client certificate authentication since this is the most secure way of communicating with the API server.

Note: When implementing authorization depending on namespaces or resources, the users restricted by a namespace or resource cannot use the `kubectl` client for accessing the Kubernetes API. Instead, the REST API can be used.

By default, a Kubernetes cluster instantiates a default namespace when provisioning the cluster to hold the default set of pods, services, and replication controllers used by the cluster. In this space, Kubernetes resources come and go, and the restrictions on who can or cannot modify resources are unconstrained. You can create additional namespaces or enforce authentication per resource so that, for example, a group of users can maintain a space in the cluster where they can get a view on the list of pods, services, and replication controllers they use to build and run their application. For details, refer to the Kubernetes documentation.

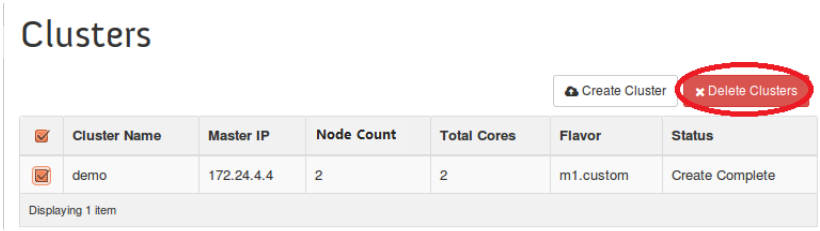
- Optionally define authorizations in Kubernetes. In Kubernetes, authorization happens as a separate step from authentication. Authorization applies to all HTTP accesses on the main API server port.

For details, refer to the Kubernetes documentation.

4 Deleting a Cluster

You can use the Cluster Management UI for terminating and deprovisioning an existing cluster:

1. Log in to the OpenStack dashboard as an administrator.
2. Go to **Project > Orchestration > Clusters**.
   The panel for cluster management provided by CLC is opened:



3. Select the cluster to be deleted and click **Delete Clusters**.
4. Confirm the deletion.
   The nodes of the cluster are deprovisioned and deleted, and all resources used by them are freed.

Note:	When you started to create a cluster and click **Delete Cluster** while the provisioning is still in progress, deleting the cluster will fail. You can remove the respective stack in OpenStack (**Project > Orchestration > Stacks**).

Glossary

Cloud

A metaphor for the Internet and an abstraction of the underlying infrastructure it conceals.

Container

A Linux-based technology, where a single kernel runs multiple instances on a single operating system. Containers form an additional layer of abstraction and automation of operating-system-level virtualization.

Cluster

A group of servers and other resources that act like a single system and enable high availability, load balancing and parallel processing.

DevStack

The OpenStack development environment. It can be used to demonstrate starting and running OpenStack services and provide examples of using them from a command line. DevStack has evolved to support a large number of configuration options and alternative platforms and support services.

Docker

An Open Source project that automates the deployment of applications inside software containers. Docker uses resource isolation features of the Linux kernel to allow independent containers to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines.

Flavor

A virtual hardware template used in OpenStack for defining the compute, memory, and storage capacity of a virtual server.

Heat

A service to orchestrate multiple composite cloud applications.

Infrastructure as a Service (IaaS)

The delivery of computer infrastructure (typically a platform virtualization environment) as a service.

Label

In the context of cluster orchestration, a facility for organizing and selecting groups of objects based on key-value pairs.

Platform as a Service (PaaS)

The delivery of a computing platform and solution stack as a service.

Pod

An application-specific logical host in a containerized environment. Pods may contain one or more tightly coupled containers.

Pods are the smallest deployable units that can be created, scheduled, and managed by Kubernetes. Pods can be created individually, the usage of *Replication Controllers*, however, is recommended.

Replication Controller

An abstraction which ensures that a specific number of *Pod* replicas are running at all times. If a pod or host goes down, the replication controller ensures that enough pods get recreated elsewhere. A replication controller manages the lifecycle of pods.

Service

In the context of cluster orchestration, a layer of abstraction for providing a single, stable name and address for accessing a set of *Pods*. Services act as basic load balancers.

Software as a Service (SaaS)

A model of software deployment where a provider licenses an application to customers for use as a service on demand.

Stack

A set of OpenStack resources created and managed by the Orchestration service according to a given template (in CLC, a Heat Orchestration Template (HOT)).

Tenant

A group of users in OpenStack. Also referred to as "project".

Workload

Abstraction of the actual amount of work that applications or services perform in a given period of time, or the average amount of work handled by an application or service at a particular point in time.