RANCHER LABS

March 2017

Deploying and Scaling Kubernetes with Rancher

Vishal Biyani and Girish Shilamkar

Contents

Introduction

1. Overview of Kubernetes capabilities

1.1 Introduction

A lot has happened within the container ecosystem in the past few years to shape how software is built and deployed. To manage a fleet of containers running microservices, one needs robust cluster management capabilities that can handle scheduling, service discovery, load balancing, resource monitoring and isolation, and more. For years, Google has used a cluster manager called Borg to run thousands of jobs, supporting thousands of applications, running on multiple clusters. Google has taken the best aspects of Borg and open-sourced them in the Kubernetes project, opening up a powerful tool for running and managing containers at scale.

In this eBook, we will review capabilities of Kubernetes, deploy Kubernetes with Rancher, then deploy and scale some sample multi-tier applications. But before we dive into details, let's first cover the general capabilities and concepts of Kubernetes. If you have some basic familiarity with Kubernetes, then you can safely skip rest of this chapter and jump to Chapter 2. The Kubernetes 101 walkthrough provided by the Kubernetes project itself provides a strong starting point for reviewing these concepts as well.

1.2 Kubernetes Concepts and Terminology

Let's take some time to understand some basic concepts and Kubernetes terminology:

Cluster
A cluster is a set of machines (physical or virtual) on which your applications are managed and run. For Kubernetes, all machines are managed as a cluster (or set of clusters, depending on the topology used).

Node
A logical machine unit (physical or virtual), which is part of a larger cluster on which you can run your applications.

Pod
A co-located group of containers and their storage is called a pod. For example, it makes sense to have database processes and data containers as close as possible - ideally they should be in same pod.

Label
Labels are names given to resources to classify them, and are always a key pair of name and value. The key-value pairs can be used to filter, organize and perform mass operations on a set of resources. Think of labels as a role, group, or any similar mechanism given to a container or resource. One container can have a database role, while the other can be a load-balancer. Similarly, all pods could be labeled by geography, with applied values like US, EU, APAC, etc. If done in the right manner, labels can act as a powerful way to classify resources of various types.

Selector
A selector expression matches labels to filter certain resources. For example, you may want to search for all pods that belong to a certain service, or find all containers that have a specific tier label value as database. Labels and selectors are inherently two sides of the same coin. You can use labels to classify resources and use selectors to find them and use them for certain actions.

Replication Controller
Replication Controllers (RC) are an abstraction used to manage pod lifecycles. One of key uses of replication controllers is to maintain a certain number of pods. This is also useful when you want to enable certain number of pods for scaling, or ensure that at least one pod. It is a best practice to use replication controllers to define pod lifecycles, rather than to create pods directly.

Replica Sets
Replica Sets define how many replicas of each pod will be running. They also monitor and ensure the required number of pods are running, replacing pods that die. Replica Sets can act as replacements for Replication Controllers.

Service
A service is an abstraction on top of pods which provides a single IP address and DNS name by which the pods can be accessed. This load balancing configuration is much easier to manage, and helps scale pods seamlessly.

Volume
A volume is a directory with data which is accessible to a container. The volume co-terminates with the pods that encloses it.

Name
A name by which a resource is identified.

Namespace
Namespace provides additional qualification to a resource name. This is especially helpful when multiple teams/projects are using same cluster and there is a potential for name collision. You can think of namespace as a virtual wall between multiple clusters.

Annotation
An annotation is a Label but with much larger data capacity. Typically, this data is not readable by humans and not easy to filter through. Annotation is useful only for storing data which may not be searched but is required by the resource (for example, storing strong keys, etc).

1.3 Kubernetes Functionalities

At bare minimum, any container orchestration platform needs the ability to run and schedule containers. But to manage containers effectively, additional features are needed. Here, we will look at the functionalities that Kubernetes has to manage containers, and along the way introduce some additional concepts that Kubernetes uses.

1.3.1 Co-Locating Related Processes

Most of applications today are built from multiple components or layers, and sometimes these components must be tightly coupled to each other. Logically, it makes sense to co-locate tightly coupled components as close to enable easier network communication and shared storage usage. Kubernetes enables co-locating related containers through pods.

1.3.2 Data and Storage

By their very nature, containers are short-lived. If a container reboots, the data inside it is lost; hence, the introduction of Docker volumes. Docker volumes lack a defined lifecycle like the containers (as of this publish date). In contrast, Kubernetes volumes are tied to the lifecycle for the container with which the volumes are associated. Kubernetes also has several volume types to cater to various use cases.

1.3.3 Secret Management

Applications use secrets such as passwords, SSH keys and API tokens all the time. To prevent disclosing the secrets in the definition files that define containers/clusters, Kubernetes encodes them in Secret objects for later referral in the definition files.

1.3.4 Application Health

Long-running applications may eventually break, or degrade. Kubernetes provides a way to check application health with HTTP endpoints using liveness probes. Some applications start but are not ready to serve requests: for example, when building a cache at the start of the application. In these situations, Kubernetes provides a readiness probe. Lastly, for applications that must terminate deliberately but gracefully, Kubernetes provides termination hooks to execute certain activities before the container is terminated.

1.3.5 Container Management and Scaling

If a container unexpectedly goes down, it is important that it is replaced by a new one. To achieve this, Kubernetes uses a replication controller, which ensures that a certain number of replicas of a pod are always running. In cases where only one replica of a pod needs to be running, its replication factor can be set to 1. in which case Kubernetes will bring it back up if it goes down. Autoscaling of pods replicas can be setup based on other conditions, such as CPU utilization.

1.3.6 Service Registry and Discovery

Consider a vast cluster running a large number of nodes. When a container fails on a given node, it may be launched on a different node. How do you ensure that all other containers connecting to that failed container receive the IP address of the replacement container? This is an important consideration in a microservices architecture where you must dynamically manage service endpoints.

While Docker allows networking at the host level only (and Docker Swarm works across hosts), Kubernetes makes network management much easier, by enabling any pod to talk to other pods within same namespace, irrespective of the host. This makes exposing ports and managing links between different services much easier.

1.3.7 Load Balancing

Every application that is scaled needs load balancing. In Kubernetes, load balancing can be implemented using an abstraction called a “service,” or with an ingress-type resource. A service masks underlying pods/containers and instead represents them as a single entity. The ingress

resource works by exposing the underlying containers through a dynamically created load balancer. We will look at both concepts in detail in later sections of book.

1.3.8 Rolling Updates

Many applications cannot be taken down for updates for an extended period of time, and in some cases, cannot be taken down at all. Rolling updates ensure that a minimum number of instances are always available, while instances are taken out of the pool for updates. Typically, the strategy is to take some percentage of instances out of service, upgrade them, and then put them back into service before repeating the process with another set of instances.

Kubernetes supports rolling updates with the use of “deployment” and “rolling-update” abstractions. Deployments are a fairly recent addition to the project, but provide a powerful and declarative way to control how service updates are performed and is recommended over rolling-updates.

1.3.9 Resource Monitoring

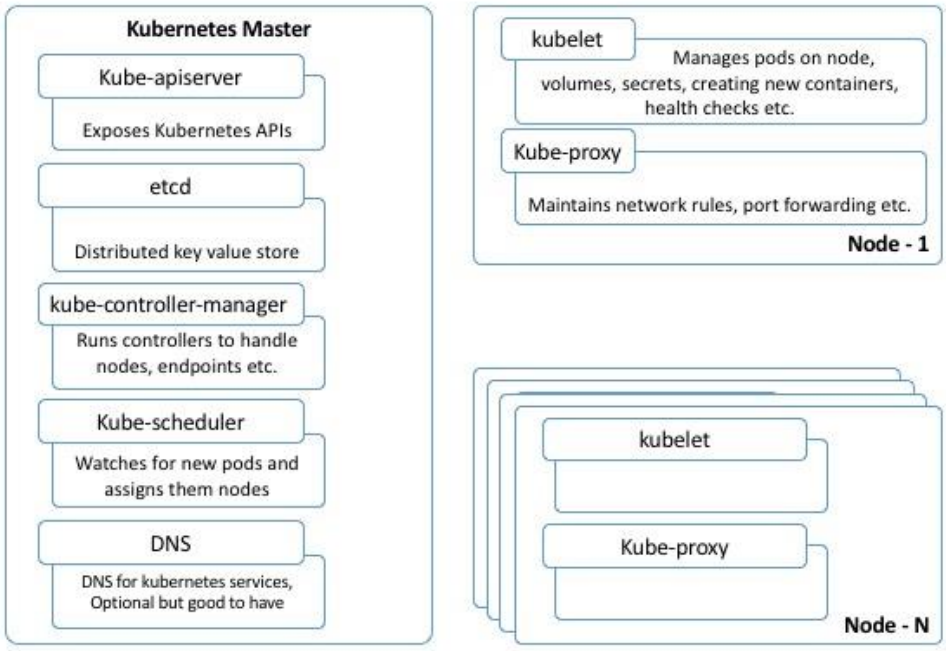
Knowing the health of your systems is critical to running a large cluster. Kubernetes monitors the clusters at multiple levels. Heapster is used to aggregate vital metrics, while the kubelet node agent queries cAdvisor to fetch data from containers and provide to Heapster. The performance data can be stored in InfluxDB and visualized by Grafana, or it can be fed to Google Cloud monitoring for storage and visualization.

1.3.10 Log Management

Fetching and analyzing log data is critical to understanding what is happening with a given cluster. Internal Kubernetes components use log library to log data; kubectl (the command line interface) can be used to fetch log data from containers. This data can be fed to an ELK (Elasticsearch, Logstash and Kibana) stack or Google Cloud logging for further analysis and visualization.

1.4 Kubernetes Components

Kubernetes works in a master-node mode, where a master can manage a large number of nodes. Some components run only on masters, some components run only on node and provide all management support needed on node.



The master can be run in HA mode with a multi-master setup. Apart from components listed for master as shown in the above diagram, there are optional components such as: user interface, container resource monitoring and logging-related components.

1.5 Summary

Kubernetes provides cluster management for containerized workloads and simplifies container orchestration at scale. In this chapter, we looked at Kubernetes terminologies, and how Kubernetes can help companies adopt containers. In the chapters that follow, we will look at how Kubernetes and Rancher can be used to deploy applications and perform various container operations.

2. Deploying Kubernetes with Rancher

2.1 Rancher Overview

Rancher is an open source software platform for deploying and managing containers in production. It includes commercially-supported distributions of Kubernetes, Mesos, and Docker Swarm for container orchestration, and allows teams to transparently view and manage the infrastructure and containers supporting their applications. Rancher provides built-in authentication, networking, storage, and load-balancing capabilities as well.

2.2 Native Kubernetes Support in Rancher

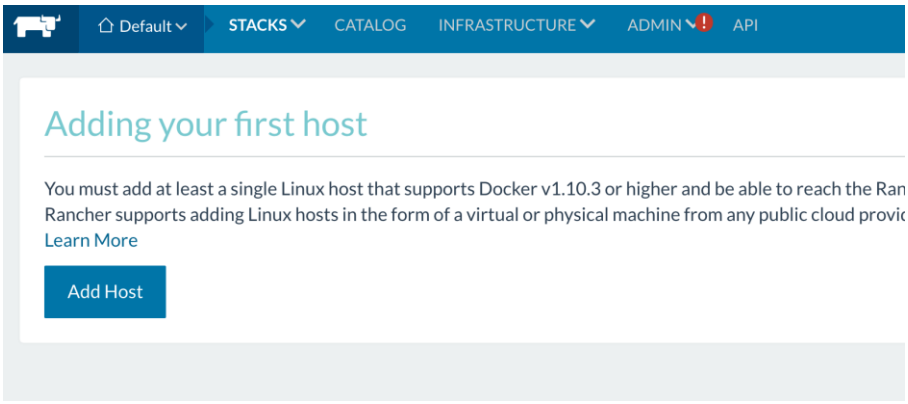
Rancher natively supports Kubernetes and allows users to control its features through a simple and intuitive UI. Kubernetes can be launched in a matter of minutes with a single click through Rancher. Multiple teams and access policies for their clusters can be managed through Rancher, as the platform integrates with LDAP, AD, and GitHub for authentication. Once the cluster is up, any number of hosts can be added and Rancher then provides complete visibility for both developer and operations teams into infrastructure and applications.

Additionally, Rancher provides an application catalog with templates to deploy complex applications in a single-click. The templates that back the application catalog can be stored in a Git repo, and can be shared across teams. Rancher DNS is a drop-in replacement for Sky DNS thus providing transparent, scalable and simplified network management across the cluster.

2.3 Setting Up a Rancher Kubernetes Environment

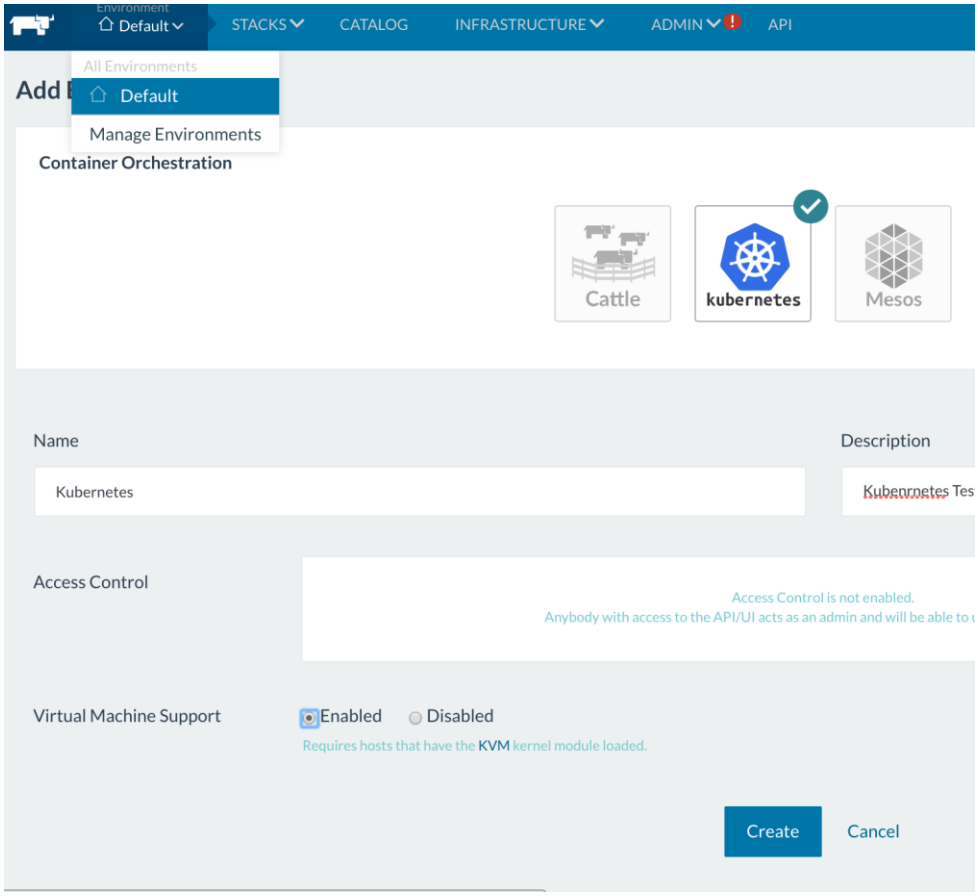
Setting up a Rancher server is easy. You can set one up by following instructions here, or if you wish to use Vagrant, you can clone the repo here and run vagrant up.

When you deploy Rancher server, you should see a screen that looks like this:



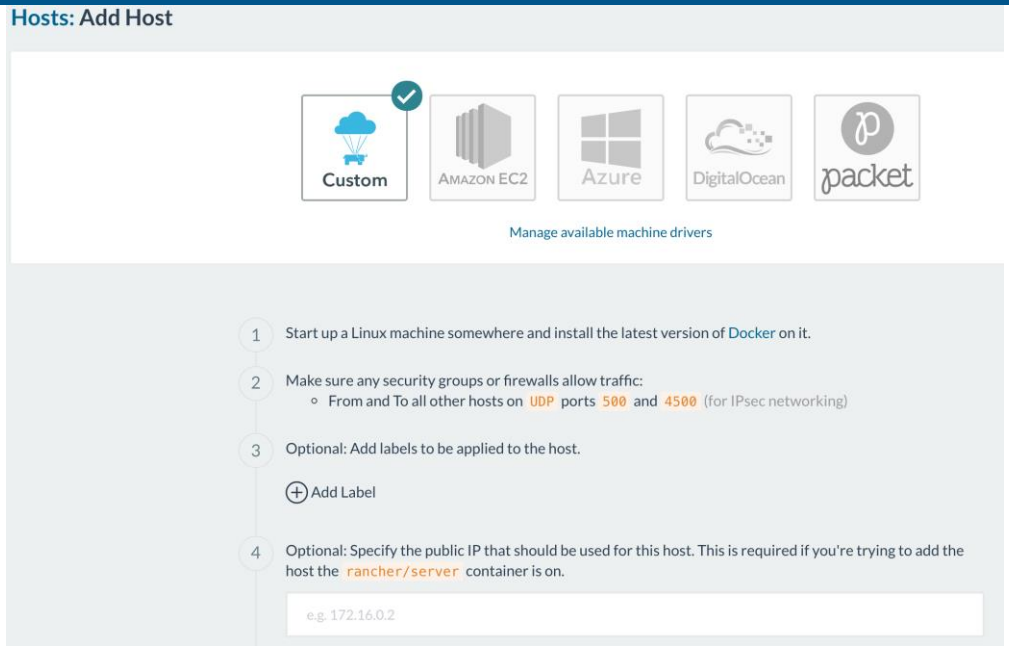
Let's first add an environment for Kubernetes. An environment in Rancher is a logical entity for sharing deployments and resources with different sets of users, and the platform allows you to

choose among several different orchestration frameworks (Kubernetes, Mesos, Docker Swarm, and Cattle). Everything is done in context of an environment. Environments in Rancher can serve as logical separations for different teams within a company, such as development or QA.

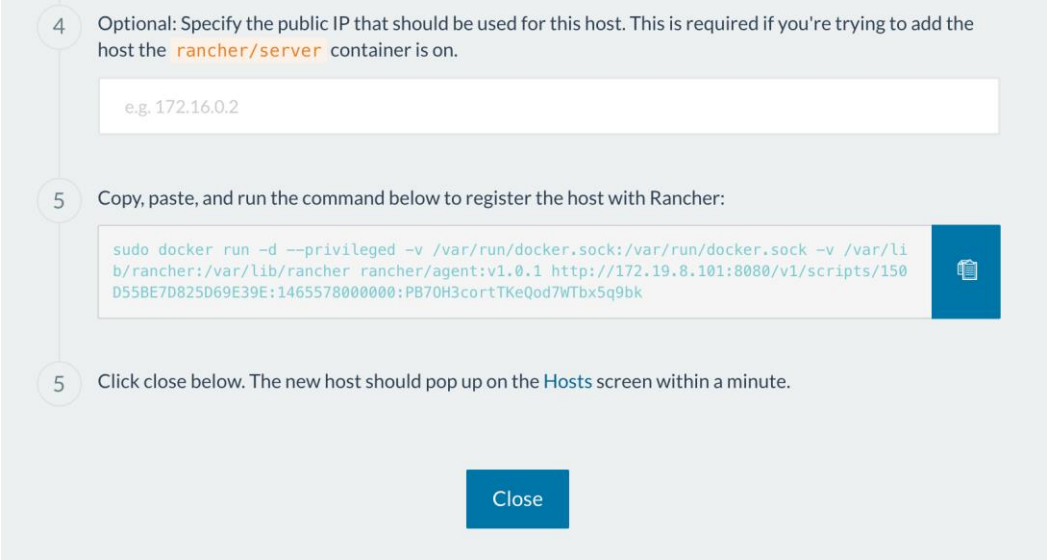


Once the Kubernetes environment has been created, we can add hosts to the it. Choose “Add Hosts” within the newly-created Kubernetes environment from the drop-down menu at top of the screen, where we can add a host machine from some of public clouds or from a custom stack. In this example, we’ll choose the custom method. If you are still using the Vagrantfile from Git repo, set up three nodes as described below. You can of course change this as per your use case.

Hostname	Details	Docker Version
ranch-svr	Rancher Master	Latest
Ranch-def	Default environment	Latest
Ranch-Kubernetes	Kubernetes environment	1.10.3 (Kubernetes compatible)



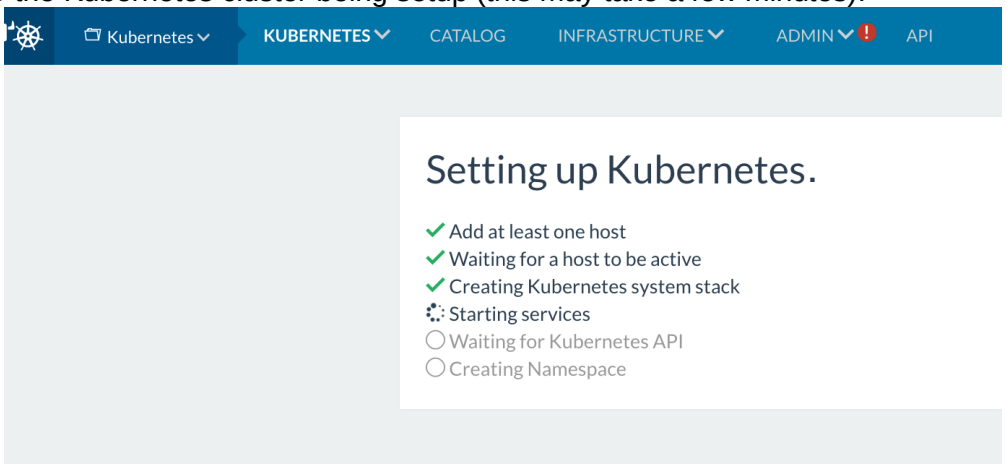
As we add the details for the hosts, the command to register the host with Rancher is modified. Copy this command and log into a host other than Rancher master on which you want to setup your Kubernetes cluster:



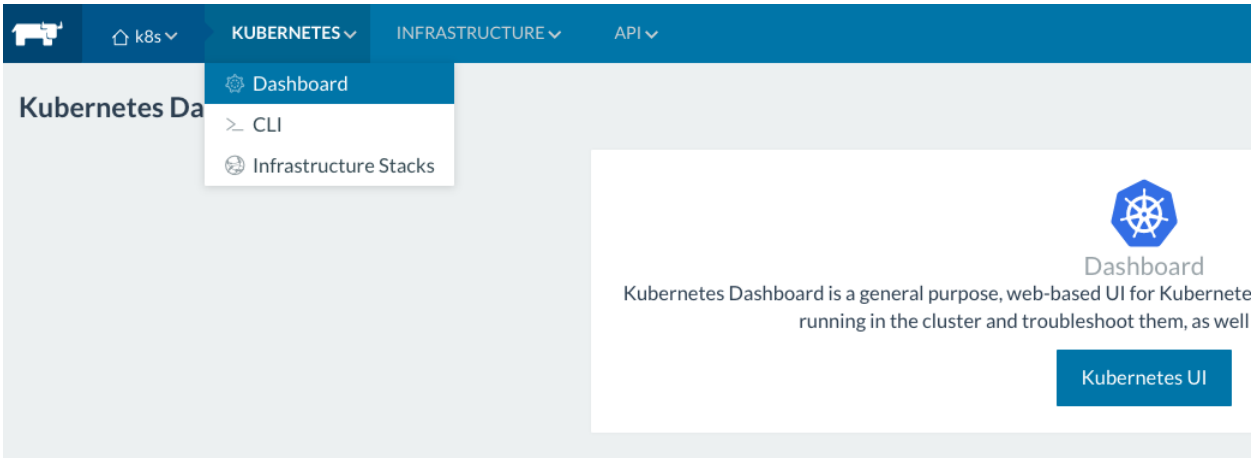
Once you run above command on a new machine, the host(s) tries to contact the Rancher server with the key. The server then verifies the key and registers the agent. Based on the environment to which the agent belongs, further instructions are sent to agent to bring it to its desired state. In our case, since this is the first and only node in the Kubernetes environment, a complete Kubernetes stack is already setup on the node (note: starting with Rancher 1.1, it is suggested to have at least three hosts for your Kubernetes deployment; having these hosts for etcd ensures a highly-available setup).

```
vagrant@ranch-svr:~$ sudo docker run -d --privileged -v /var/run/docker.sock:/var/run/docker.sock -v
ancher:/var/lib/rancher rancher/agent:v1.0.1 http://172.19.8.101:8080/v1/scripts/ED500E4A6AD6A1317A2
0000:nTYTN60WZa9TyeSIrVI3ozUq4U
Unable to find image 'rancher/agent:v1.0.1' locally



v1.0.1: Pulling from rancher/agent
5a132a7e7af1: Already exists
fd2731e4c50c: Already exists
28a2f68d1120: Already exists
a3ed95caeb02: Already exists
91d0cd4547c7: Downloading [=>                                                  ] 3.227 MB/83.51 MB
664cee9fec33: Download complete
371d94bd3bdf: Download complete
```

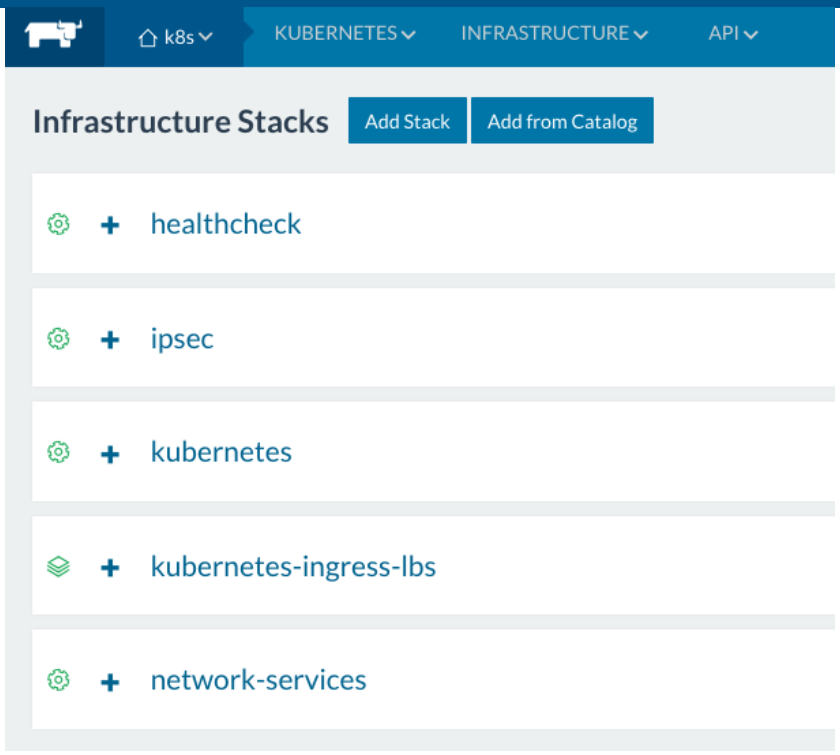
You'll see the Kubernetes cluster being setup (this may take a few minutes):



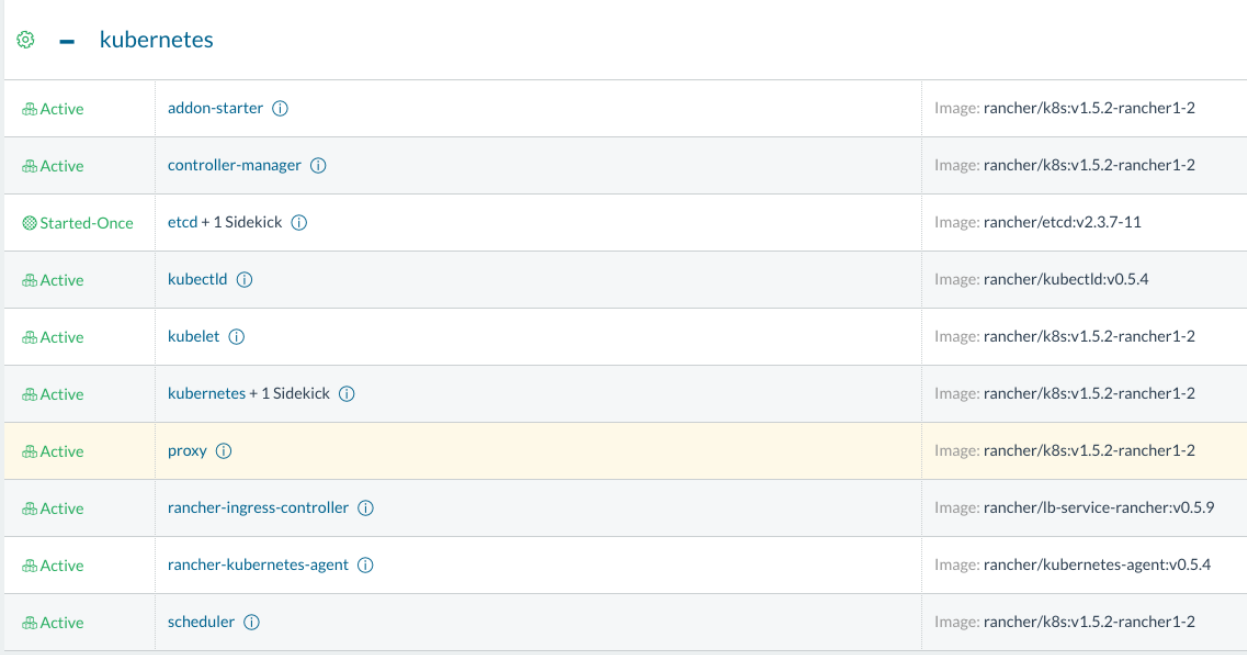
Once the cluster has been set up, the Kubernetes menu is populated. If you choose the "Dashboard" option from the Kubernetes drop-down menu, you will notice a link to the Kubernetes UI :



If you navigate to "Infrastructure stacks" from Kubernetes menu, you will notice various groups of stacks which have been deployed as part of setting up Kubernetes:



After expanding the Kubernetes stack, you will see various components of Kubernetes:



- *Controller-manager* is a core control loop which continuously watches the state of clusters and takes actions if needed to bring it to the desired state.
- *etcd* is a highly available (HA) key-value pair store for all persistent data for the cluster. The etcd server should only be accessible by Kubernetes API server as it may contain sensitive information.
- *kubectld* is the daemon which runs kubectl.

- *kubelet* is an agent node and runs on every node in the cluster to manage containers running on that host.
- *kubernetes* is the API server which provides all CRUD operations on cluster through a API.
- *proxy* is another component which is running on every node in Kubernetes cluster and provides a simple network and load balancer across hosts.
- *scheduler* manages pods which are not yet assigned to nodes and schedules them.
- *Rancher-kubernetes-agent* manages the communication between Rancher and the Kubernetes cluster
- *Rancher-ingress-controller* will leverage the existing Kubernetes load balancing functionality within Rancher and convert what's in the Kubernetes ingress to a load balancer in Rancher.

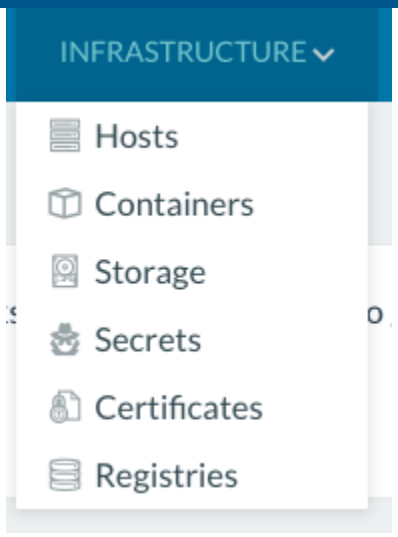
2.4 How Rancher Extends Kubernetes for User-Friendly Container Management

As you might have noticed in previous section, launching Kubernetes is a breeze in Rancher. But Rancher has several features that make it easy to manage the cluster:

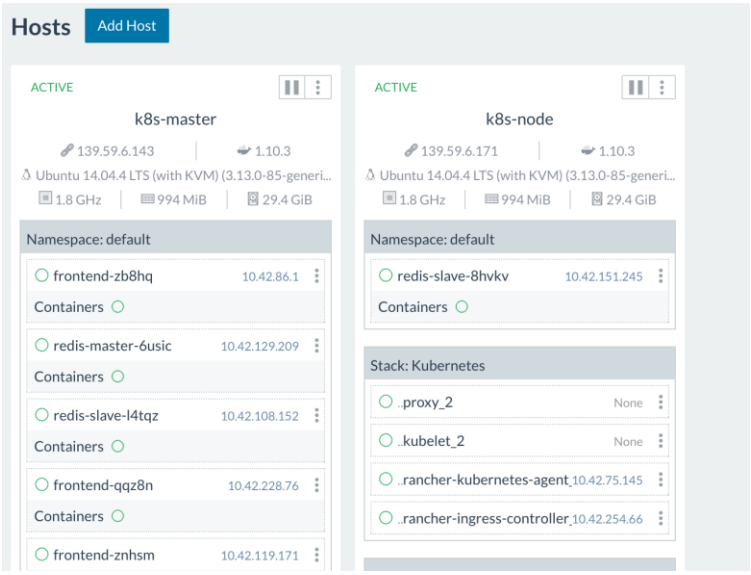
- Rancher simplifies Kubernetes networking by providing Rancher DNS as a drop-in replacement for SkyDNS. The Kubernetes cluster can then span across multiple resource pools or clouds.
- Rancher enables teams to set up and manage multiple Kubernetes environments, and provides role-based access management (RBAC) for both teams and individuals for each environments.
- Rancher's intuitive user interface allows you to execute CRUD operations on all of Kubernetes objects such as pods, replication controllers and services. Users can even manage underlying containers, view logs for those containers, and execute shell right from UI.
- Rancher has built-in credentials management.
- Rancher can provide access to kubectl from the Rancher UI itself.
- The Rancher load balancer allows traffic routing from hosts to Kubernetes services and pods

2.4.1 Infrastructure Visibility

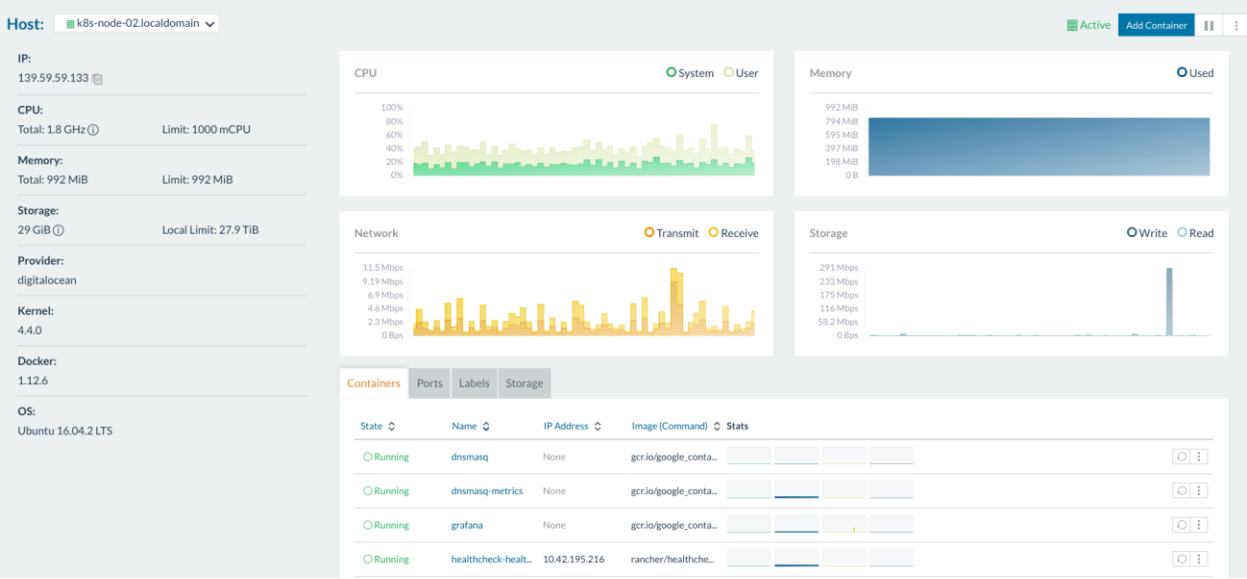
The Rancher UI provides complete visibility into the infrastructure on which Kubernetes clusters are running. The infrastructure menu will list various resources as displayed below:



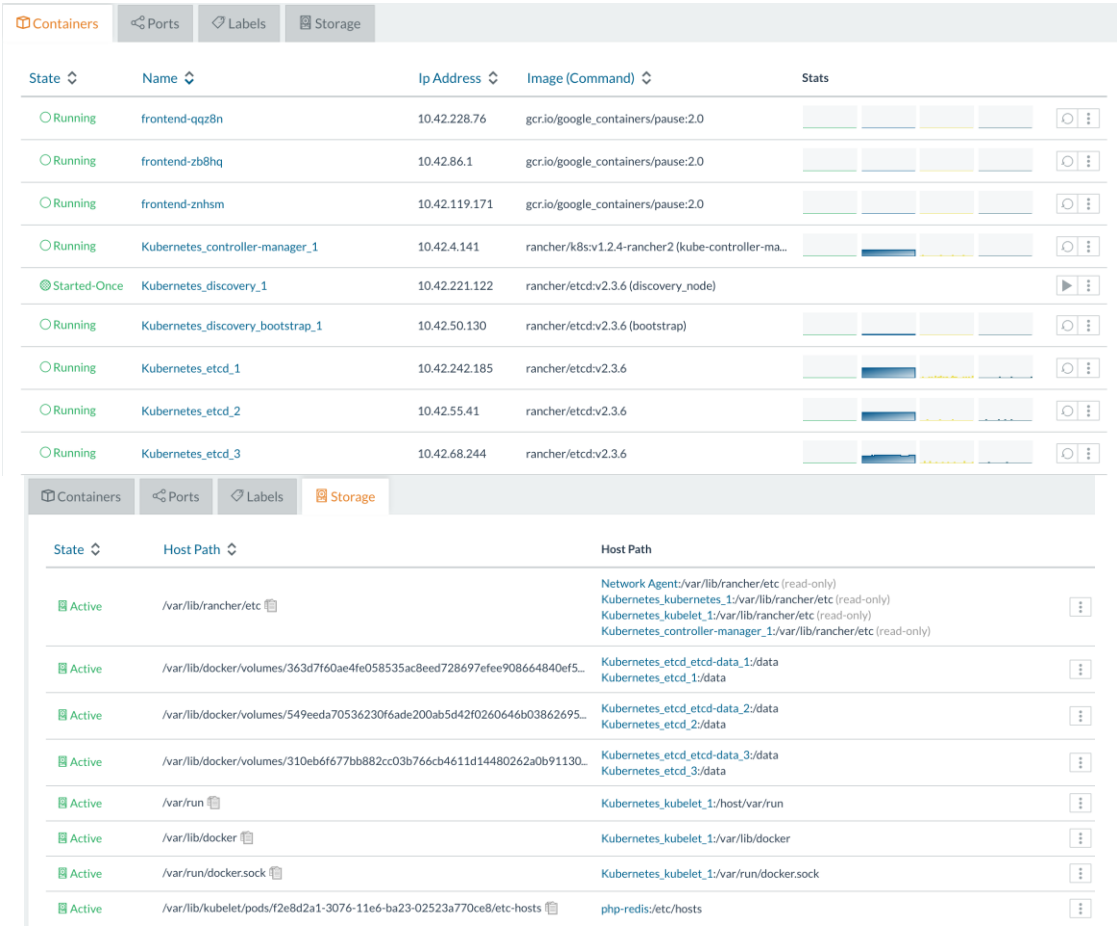
The Hosts tab provides visibility into all hosts, the containers running in those hosts and their overall status. You can edit host labels by editing host definitions here.



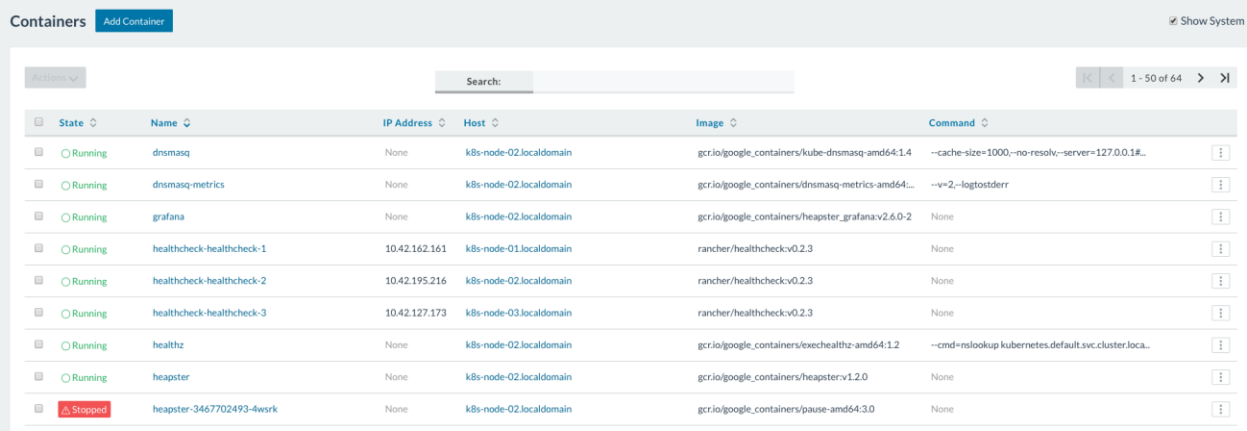
When you click on an individual host, you can view detailed information about the host:



The second part of host screen shows details about the containers running on the host, the ports open on that host and storage mounts used by containers, etc.



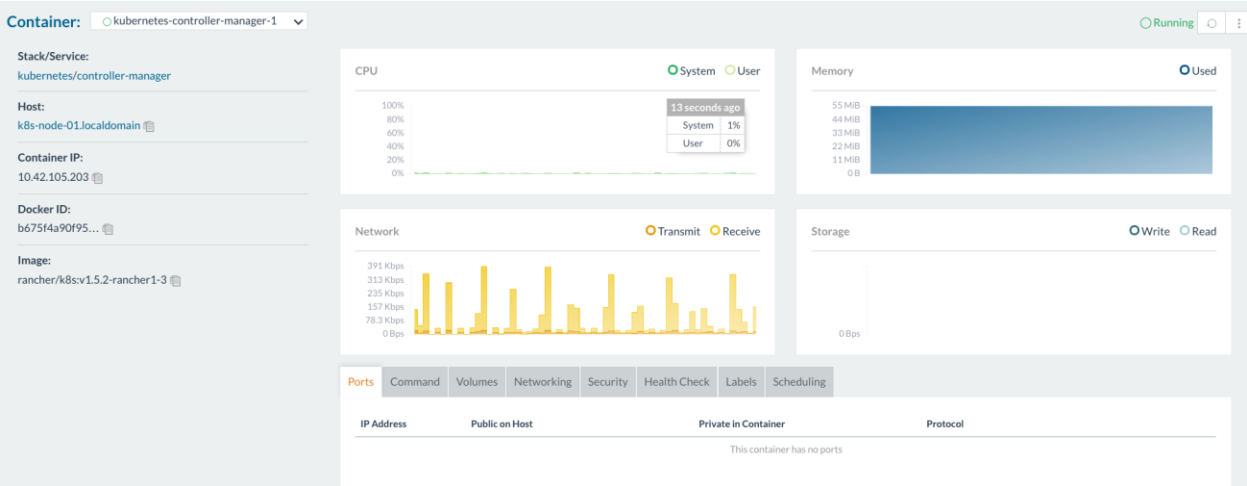
The containers menu provides information for all containers in the environment, including the hosts on which they're running, their IP addresses, and statuses etc. You can also search for specific containers in the search bar at top.



From context menu on right hand side, you can execute shell directly into the running container, and view the container logs:

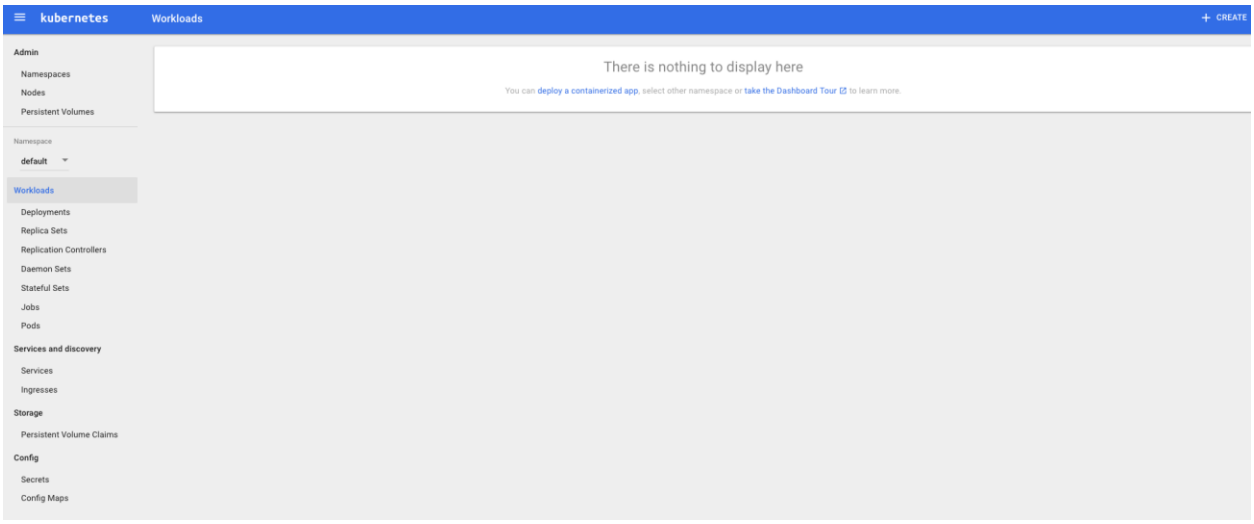
If you click on any single container from the list, you will see detailed, vital information about that container such as CPU, memory, network and disk consumption. Information about labels,

volumes and ports is also provided in bottom section. This level of detail and access to underlying infrastructure provide a secure way to access the hosts.

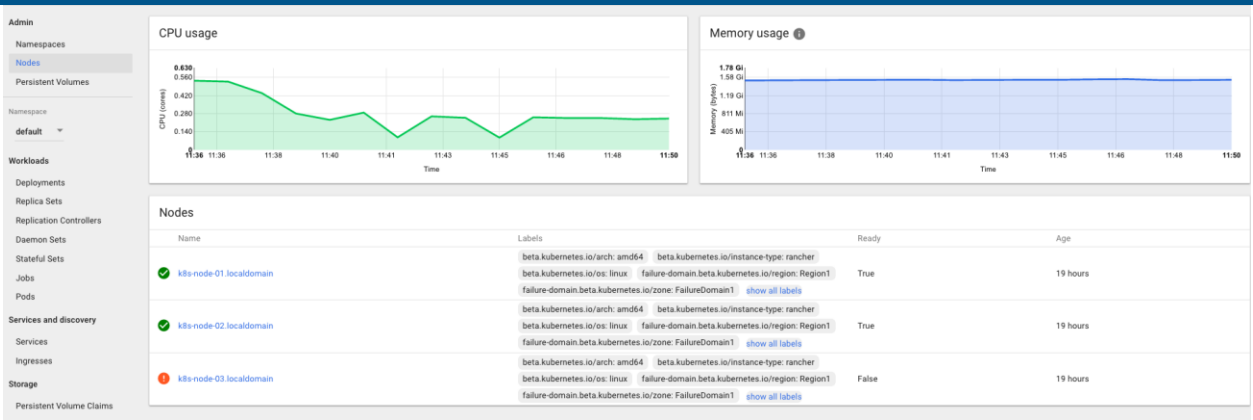


2.4.2 Kubernetes Dashboard

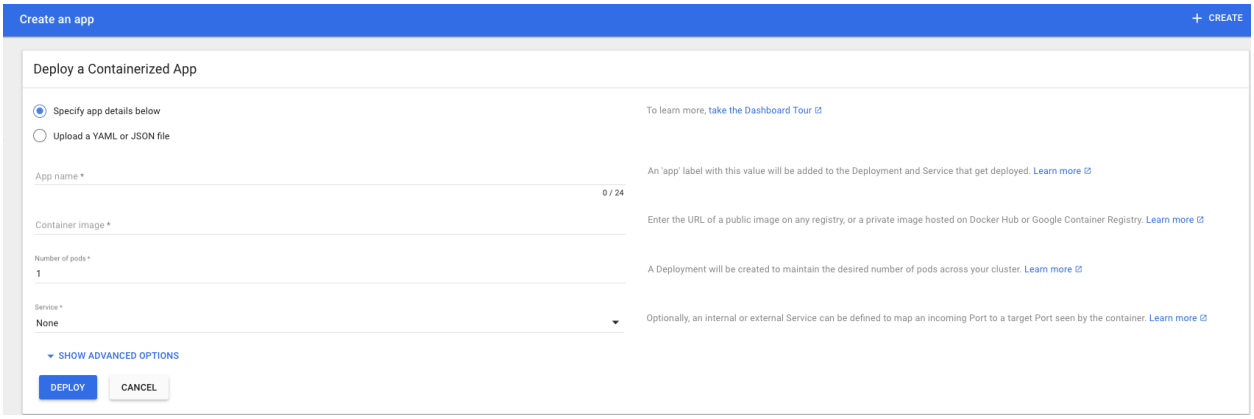
As of Rancher 1.4, Rancher uses the Kubernetes Dashboard for providing a concise and uniform view of your deployment



The left hand side menu provides quick navigation between namespaces and multiple types of objects such as Services, Deployments, Secrets etc. The nodes section provides a quick overview of the nodes in the system:



For creating a new type of object, you can use the create option on right top corner. You can input all parameters one by one or simply upload a JSON/YAML format file with specifications of the object to be created.



2.4.3 GUI-Based CRUD Operations for Kubernetes

In this section, we will create a guestbook application using CRUD operations on Kubernetes objects. We will use templates from the guestbook sample application in the Kubernetes examples. We will show you how to create one service and one replication controller, which can be used as a basis for other services and replication controllers. If you don't have a copy of the guestbook, ensure that you clone it before you proceed.

We will deploy following components one by one to deploy the complete Guestbook application:

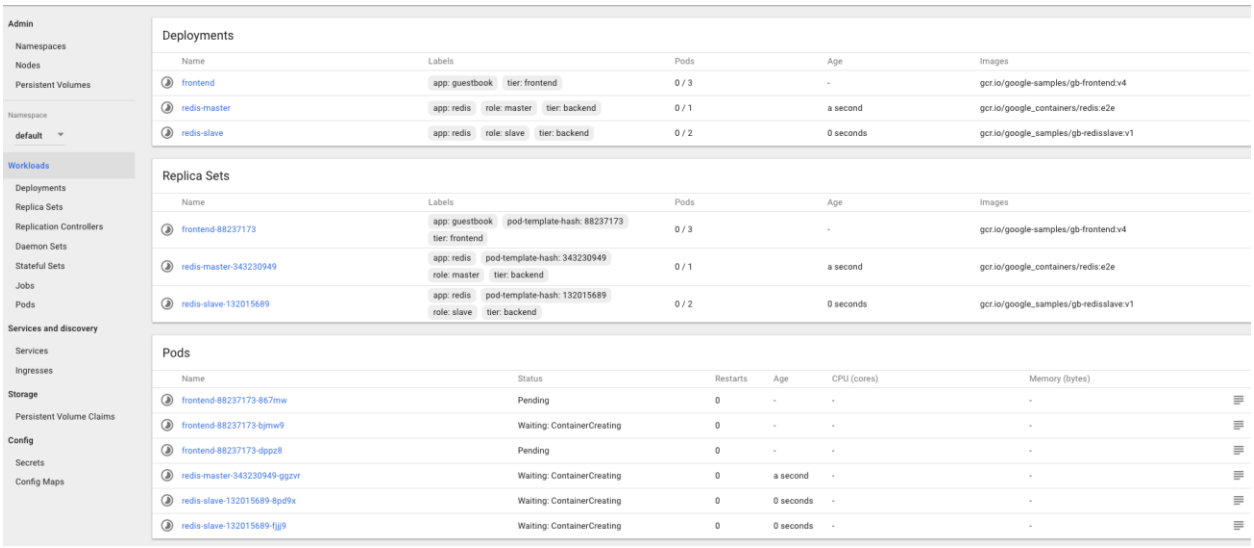
- Service definitions for:
  - FrontEnd component :
  - Redis Master
  - Redis Slave component
- Deployment definitions for:
  - Front End
  - Redis Master
  - Redis Slave

Open the "frontend-service.yaml" and uncomment the line with content "type: LoadBalancer", after changes the code should look like:

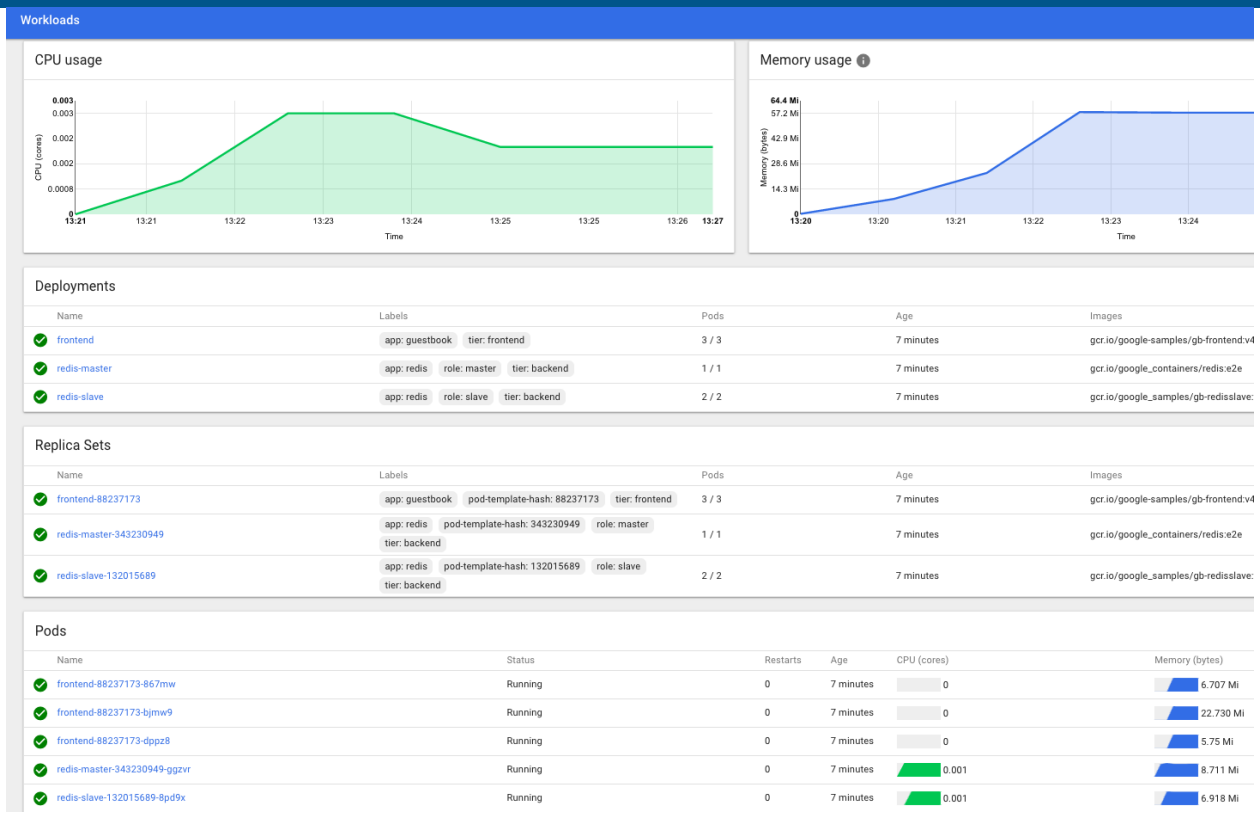
```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # if your cluster supports it, uncomment the following to automatically create
  # an external load-balanced IP for the frontend service.
  type: LoadBalancer
  ports:
    # the port that this service should serve on
  - port: 80
  selector:
    app: guestbook
    tier: frontend
```

Open the Kubernetes Dashboard, click on “Create” and upload the newlymodified service file. Similarly also deploy other .yml files in the guestbook directory.

After you have created all Services and RCs, you will see the complete stack being created:

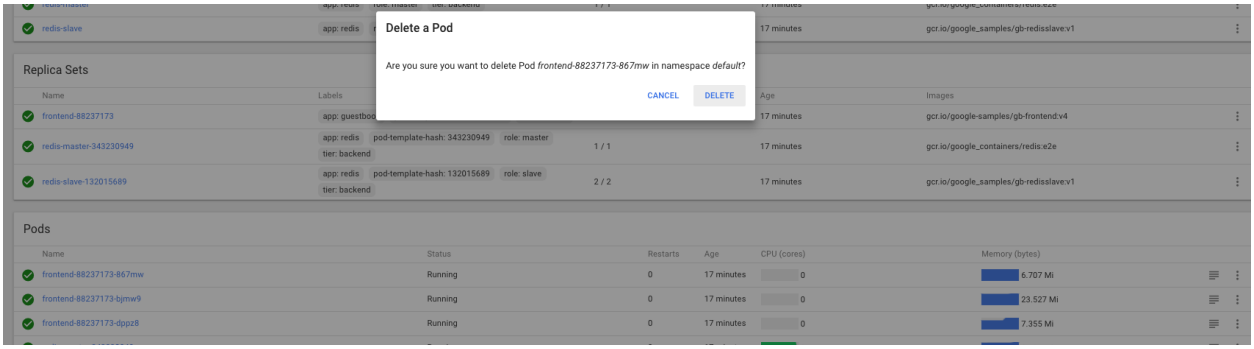


After a few minutes you will notice all deployments and PODs are in GREEN state:



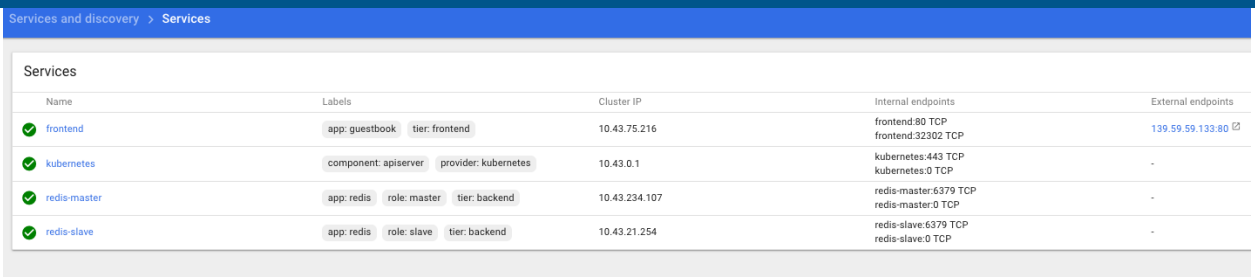
There are some key points to note here:

- There were three pods created for the frontend, based on the RC definition. It is a Kubernetes best practice to not create pods directly and to only create them through RCs.
- You can also get the details of each of the containers that are running in a pod, along with some basic information about its status.

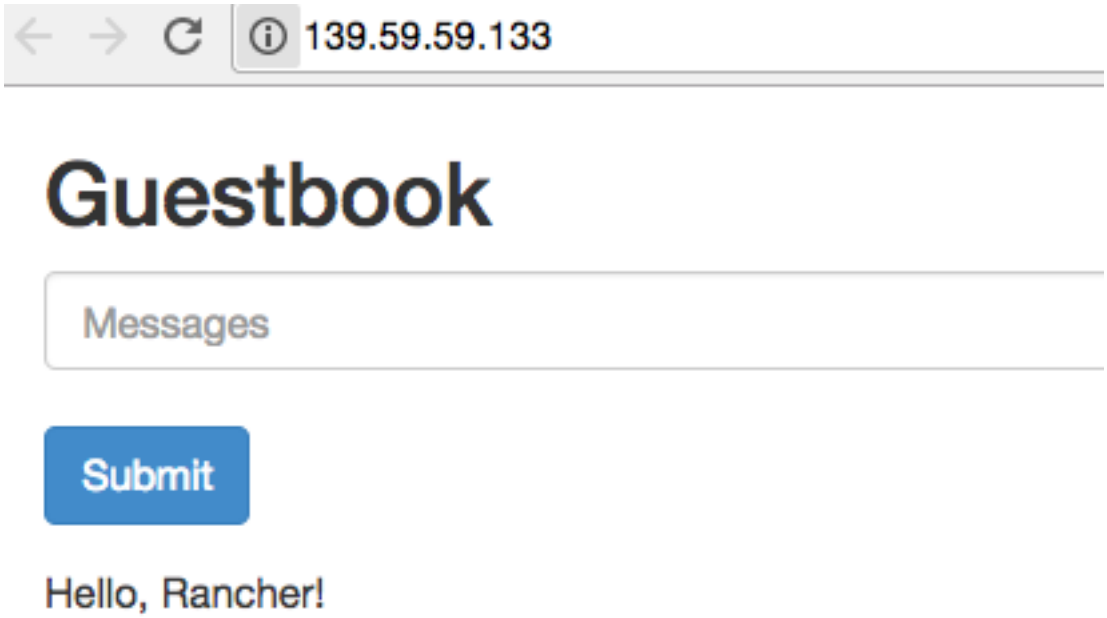


If you delete a POD, the replication controller will ensure that another POD is created immediately.

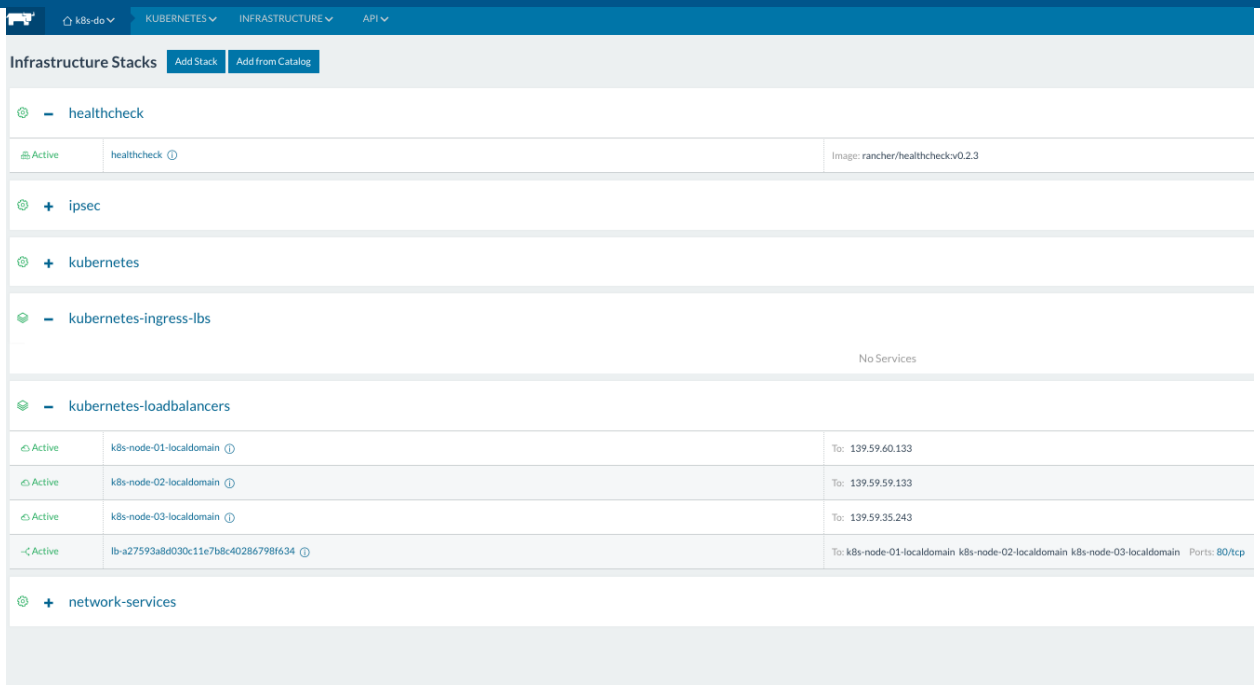
Now let's try to access the service we just deployed. Navigate to Services and you will notice that the service we deployed with "LoadBalancer" type has created a public endpoint:



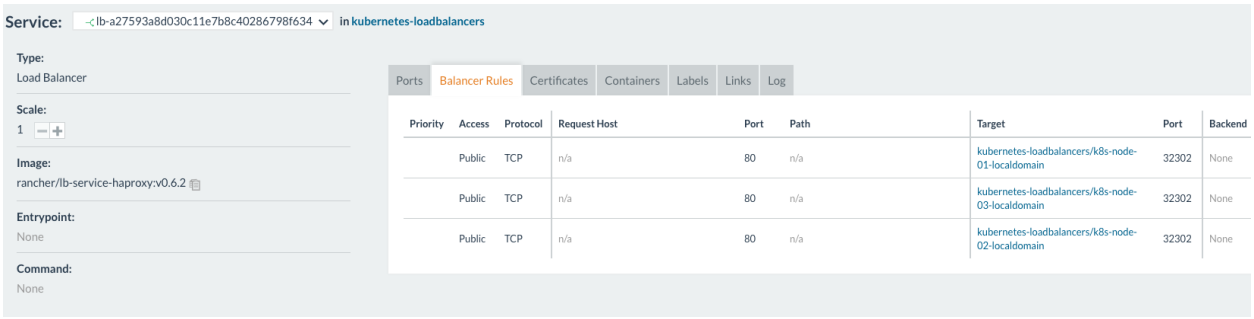
And if you access the same, you will see the guestbook application:



If you are using a cloud provider, it is natural that the LoadBalancer type service will create a load balancer and attach the containers to it. But in this case, Rancher is doing some work. Returning to the Rancher dashboard, you will notice that Rancher has created a HAProxy load balancer and abstracted the service being exposed:

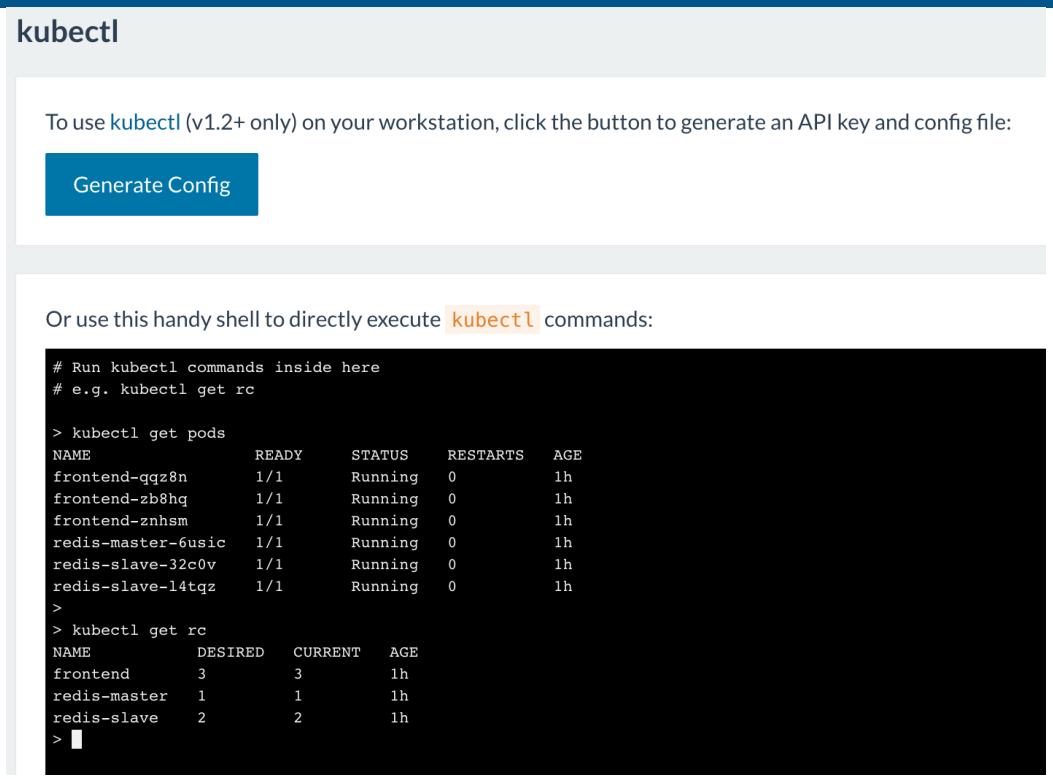


If you click on the Load Balancer above, you will see:



2.4.4 Usingkubectl - Credential Management and Web Access

Rancher provides a GUI-based way to interact with Kubernetes, which supplements kubectl, the command line interface to Kubernetes. Rancher exposes this CLI through the UI; it can be accessed by selecting Kubectl from the Kubernetes drop-down menu. You can then execute commands through kubectl to get information or to change cluster configuration.



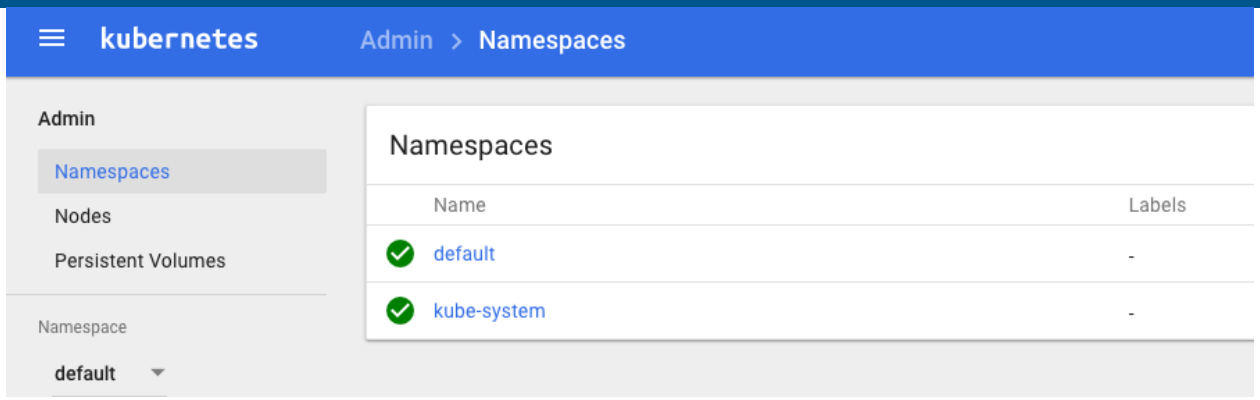
You can also generate the configuration file from “Generate config” button at the top – and along with a local executable of kubectl, interact with the Kubernetes cluster from your own machine.

One of important things to understand in this context is how your access in UI will affect what you do. Every user in a Rancher environment is either a “user” or an “admin”. As the name suggests, the admin has all the privileges associated with the environment, whereas users can be split further into specific roles (the same user can have different roles). The following table explains four roles and their access level within the environment.

	Owner	Member	Restricted	Read-Only
Operations on Stacks, Services, and Catalog	Yes	Yes	Yes	No
Operations on Hosts	Yes	Yes	View Only	View Only
Managing user access	Yes	No	No	No

2.4.5 Manage Kubernetes Namespaces

Namespaces are virtual clusters in Kubernetes that can sit on top of the same physical cluster. They provide logical separations between teams and their environments as needed. You can view existing namespaces in Kubernetes Dashboard by clicking on Namespace option in left hand side menu:



You can also add namespaces with simple YAML configuration:

```
apiVersion: v1
kind: Namespace
metadata:
  name: test-namespace
```

3 Deploying a Multi-Service Application

3.1 Defining Multi-Service Application

In Chapter 2, we deployed a guestbook application to illustrate CRUD operations in Rancher (the guestbook templates are here, detailed information and example application is available at here). In this chapter, we cover the details of services, load balancing, service discovery and finally persistence with cloud disk such as Google Cloud. You can also look at other examples such as WordPress with MySQL (beginner level, and available here), and pet store example which is advanced and must be done after finishing this basic exercise (available here).

The Guestbook application has three components:

- Frontend: The UI which takes user inputs and persists to Redis. There can be more than one node for frontend component load balanced by a load balancer
- Redis Master: The master node of Redis used to write data from the frontend. At the moment, Redis does not support multi-master out of the box; we will stick to a simple single node master in this example.
- Redis Slave: Redis slave is used to read data by frontend. We are going to scale this to two nodes so a load balancer will be needed.

3.2 Designing a Kubernetes service for an Application

Services are abstractions that hide the underlying changes in pods. Let's define a service for the frontend object and look at some important aspects of how it affects behavior.

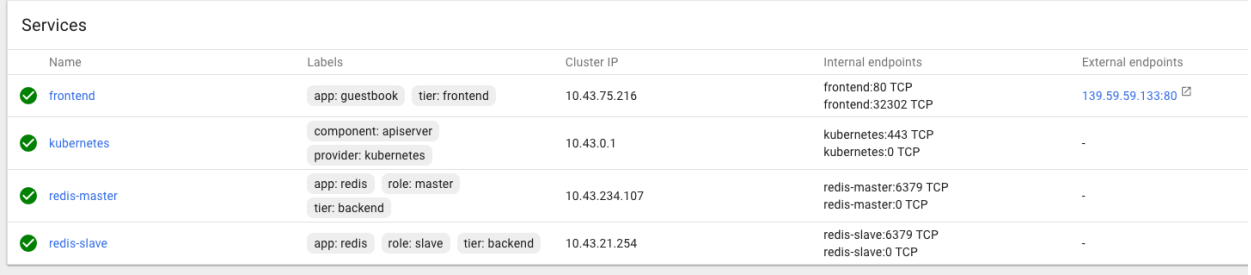
```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  type: NodePort
  ports:
    - port: 80
    - protocol: "TCP"
    - targetPort: 80
  selector:
    name: frontend
```

In the first block, we define a service named "frontend": this will create a service object in Kubernetes with a IP (usually referred to as the cluster IP). For all other services, the cluster IP is resolved for the service name and traffic forwarded to the pods. Since the selector for this service is "frontend", this service will only target the pods which have a name label as frontend and are running port 80. The selector expression is evaluated continuously and posted on EndPoint object within Kubernetes. For example, if we scale from two pods to three, traffic will be routed to the new pod.

The targetPort is same as port if not specified, and the default protocol is TCP; we can omit those lines if we want to stick to defaults. One of key points to understand here is that "port" refers to the Kubernetes exposed port, whereas targetPort is the port where the container is listening – but none of these refer to port of host on which the pod is running. If one service wants to talk to

another service within Kubernetes cluster, the port of the service will be used. But if you want the service to be exposed to the outside world we have to use NodePort or LoadBalancer in “type” (Called ServiceType). If you don’t specify any value, then the default value is ClusterIP, which allows connections only from within cluster.

NodePort proxies the port from service definition on the host and forwards all the traffic to intended container and port. NodePort is chosen randomly from a pre-configured range, or can be specified in the definition. The service is then available on each host on which a pod is running at NodePort. NodePort can be used when you have only one node of a given component and don’t want to put additional load balancer layer. NodePort is often used when you don’t want to use the load balancer provided by the cluster, want to use an external one instead, or your cloud provider does not fully support Kubernetes. The following screenshot shows the services and their type along with other information.



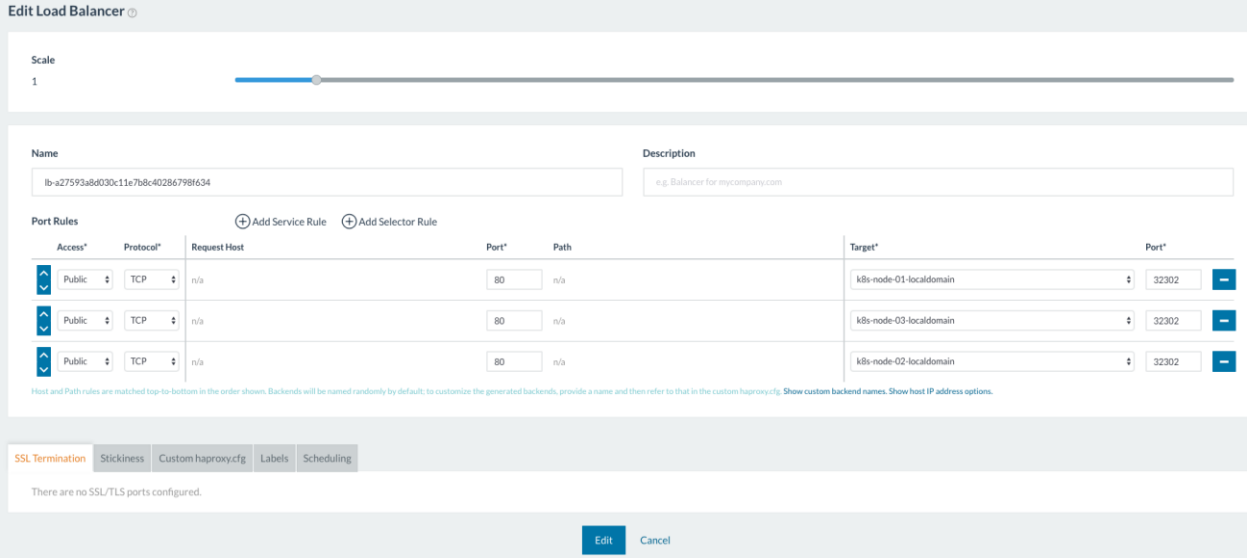
You can also create services without a selector; this is done to abstract external services such as a database or services in another Kubernetes namespace. Since the (pod) selector does not exist in these cases, we have to manually create the EndPoint object so that the service refers to external service. The service definition remains in same format except the selector block and EndPoint declaration looks like the following:

```
apiVersion: v1
kind: EndPoint
metadata:
  name: frontend
subsets:
  addresses:
    ip: “192.168.17.99”
  ports:
    port: “8090”
```

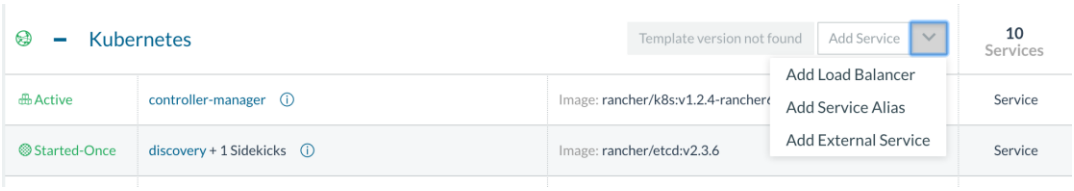
3.3 Load Balancing using Rancher Load Balancing services

We have built the frontend service using NodePort in the earlier section; now let’s build the service using LoadBalancer type. The definition of service is same except “type” has value of “LoadBalancer”. Once the service is successfully created, navigate with the Rancher UI to the Kubernetes menu, select “Infrastructure stacks” and look for the section labeled “kubernetes-loadbalancers”. You will notice that a load balancer has been created:

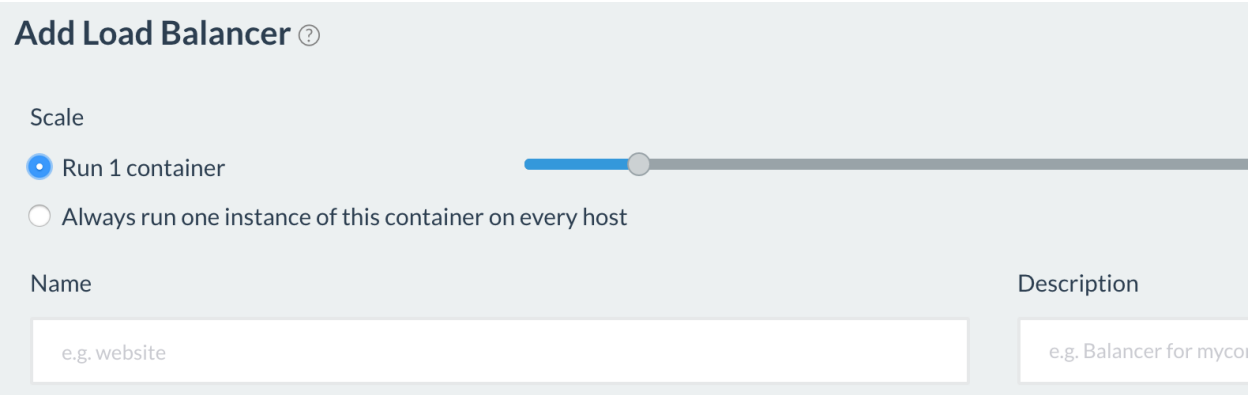
If you choose to edit the load balancer, you'll see more options around scaling, routing etc. The load balancer created by Rancher uses haproxy, and allows for additional configuration options in the global and default sections.



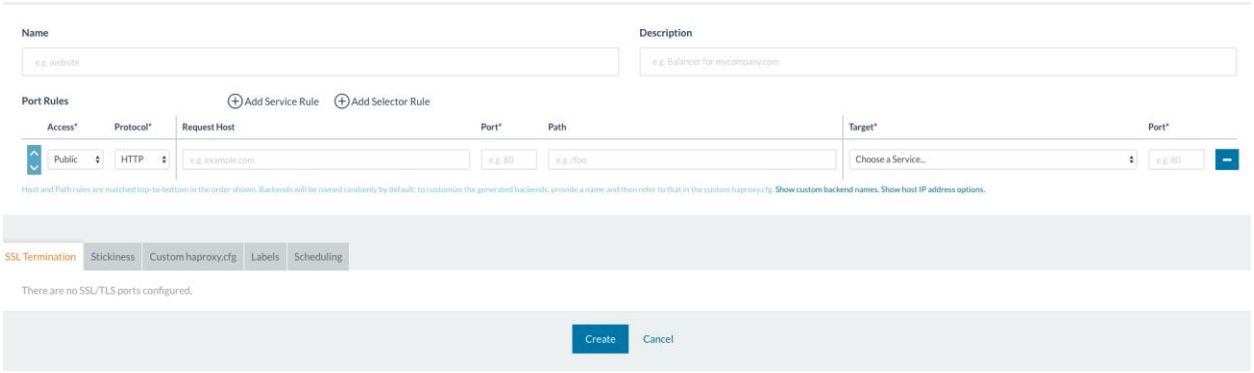
Although this load balancer was created automatically as part of the service definition, we can explicitly create a load balancer and with more fine-grained options. The system section has option to add load balancer:



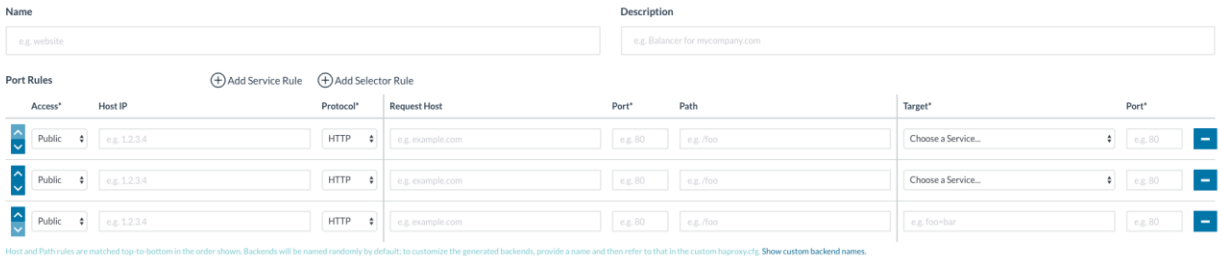
You can run a fixed number of containers of load balancer or you can choose to run one instance of load balancer on each host. The number of instances of load balancer should be less than or equal to number of hosts to avoid port conflict. If you try to create more load balancers than hosts, then load balancer will wait for more hosts to come up and will be in activating state till more hosts are available. Additionally, for fixed number of load balancers are affected by the scheduling rules which we will go through shortly.



Rancher supports L4 load balancing and forwards the ports to targets. You can configure multiple ports and services, and the load balancer will forward traffic to a combination of host and ports in a round robin fashion.



The load balancer can also be configured as an L7 load balancer by setting some advanced options. If you don't configure the additional optional choices, then it will work like a L4 LB.



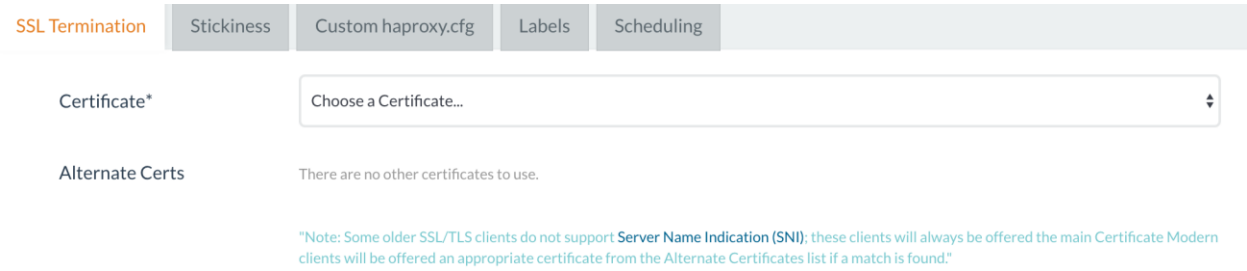
In the above screen, we can configure different hostnames and request paths to be routed to different target services. For example, an incoming request to Rancher.com can be routed to a web application, while Rancher.com/demo can be routed to a completely different web application. Even wildcard can be used for example *.example.com for targeting a service. Since you can define multiple rules – it is possible that there is an overlap in matching rules to incoming requests and hence following precedence order is used:

1. **HOSTNAME WITH NO WILDCARDS AND URL**
2. **HOSTNAME WITH NO WILDCARDS**
3. **HOSTNAME WITH WILDCARDS AND URL**
4. **HOSTNAME WITH WILDCARDS**
5. **URL**
6. **DEFAULT (NO HOSTNAME, NO URL)**

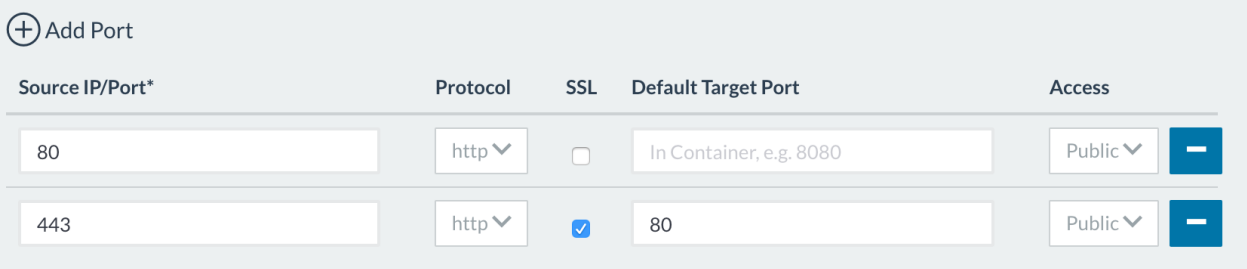
Since the load balancer was designed as an L4 load balancer, and further enhanced to be an L7 load balancer, there is one catch you should keep in mind: if you define two services (let's call them S1 and S2) and two listening ports (P1 and P2), then the mapping will be done for all four combinations of services and ports (S1-P1, S1-P2, S2-P1, S2-P2). This will introduce traffic from the P1 to S2, or conversely, P2 to S1. To prevent this, you will need to add dummy rules as shown in the example below (shown for S1 on port 81, and S2 on port 80).

```
lb-test:
  ports:
  - 80:80
  - 81:81
  labels:
    io.Rancher.loadbalancer.target.service1: 80=80
    io.Rancher.loadbalancer.target.service1: dummy1:81=81
    io.Rancher.loadbalancer.target.service2: 81=81
    io.Rancher.loadbalancer.target.service2: dummy2:80=80
  tty: true
  image: Rancher/load-balancer-service
  links:
  - service1:service1
  - service2:service2
  stdin_open: true
```

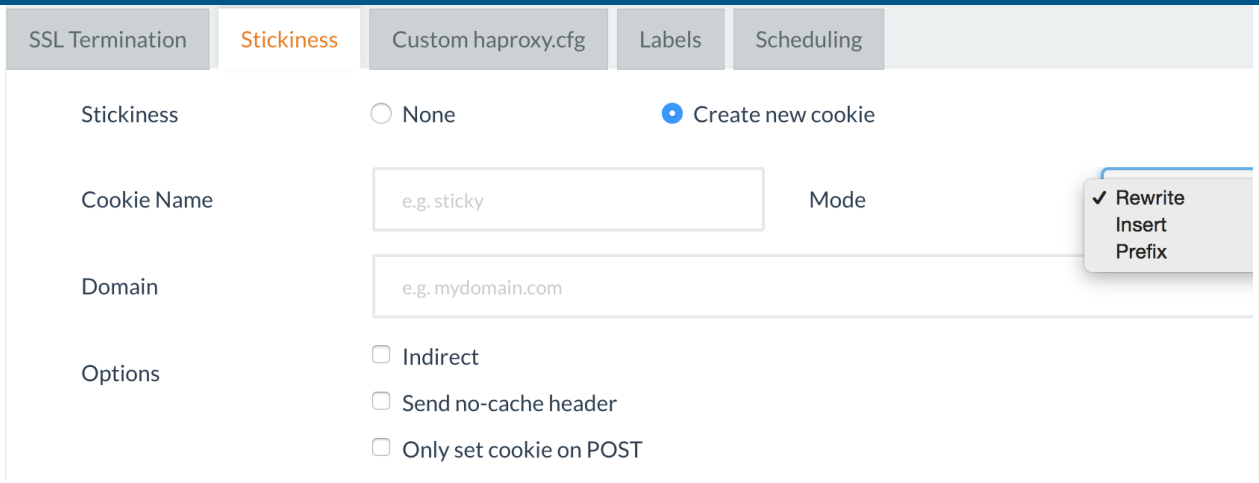
If you have chosen one of the listening ports to be “SSL” then you get options to choose the certificate for the same.



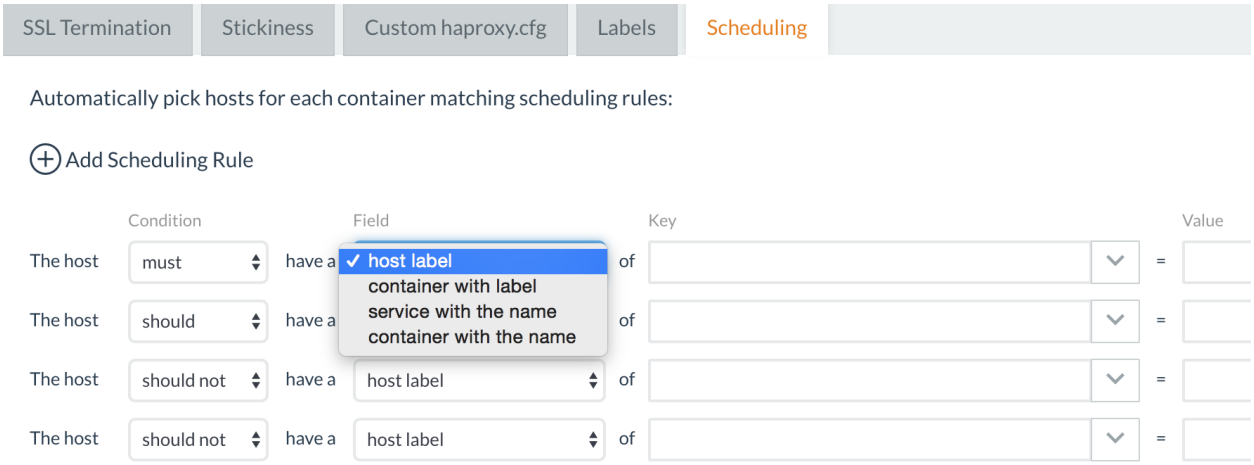
If you want to serve traffic from both HTTP and HTTPS, this can be achieved by using two listening ports and mapping the target for the SSL-checked port to the HTTP port:



The load balancer also supports stickiness on requests using cookie. You can define a cookie for all request, and responses and can be customized based on needs of session stickiness:



Labels and scheduling rules together provide a way to control which hosts will run the load balancer. The ability to set conditions at the host-, container- and service-level altogether make it possible to set very fine-grained scheduling policies in Rancher (note: if you have choose to run only one container on each host, then you will only see host labels).



3.4 Service Discovery

There are two ways Kubernetes can implement service discovery: through environment variables and through DNS.

Environment variables are set by Kubernetes. They support Docker linking, and simple semantics like ${SERVICE_NAME}_SERVICE_HOST, etc. By convention, names are upper case, and dashes are converted to underscores. If you run “docker inspect CONTAINER_ID” on one of the containers, you will see that many variables have been set for the container. One of shortcomings with environment variables is that you have to maintain their order of creation. Since the environment variables are set for the container, you will need to bring up a dependent service first, then the service using the environment variable.

```
"Env": [
            "KUBERNETES_PORT_443_TCP_PORT=443",
            "FRONTEND_PORT_80_TCP_PORT=80",
```

```
"FRONTEND_PORT_80_TCP_PROTO=tcp",
"FRONTEND_PORT_80_TCP_ADDR=10.43.89.247",
"REDIS_MASTER_SERVICE_PORT=6379",
"REDIS_SLAVE_SERVICE_HOST=10.43.115.100",
"KUBERNETES_PORT=tcp://10.43.0.1:443",
"FRONTEND_PORT=tcp://10.43.89.247:80",
"REDIS_SLAVE_SERVICE_PORT=6379",
"REDIS_MASTER_PORT_6379_TCP=tcp://10.43.31.149:6379",
"REDIS_SLAVE_PORT=tcp://10.43.115.100:6379",
"REDIS_SLAVE_PORT_6379_TCP_ADDR=10.43.115.100",
"KUBERNETES_SERVICE_PORT=443",
"REDIS_MASTER_SERVICE_HOST=10.43.31.149",
"REDIS_SLAVE_PORT_6379_TCP_PROTO=tcp",
"REDIS_SLAVE_PORT_6379_TCP_PORT=6379",
"KUBERNETES_SERVICE_HOST=10.43.0.1",
```

A cleaner way to implement service discovery is with the DNS server. In a Rancher environment, the SkyDNS service is built-in with Kubernetes environment (and briefly discussed in section **Error! Reference source not found.**).

New records are created in DNS when new services and pods are started, and they are updated if those pods or services change. A service is mapped to its cluster IP and every service in a namespace is directly accessible to other services. If you want to access a service in a different namespace, this can be done with <service_name>.<stack_name>

3.5 Storage

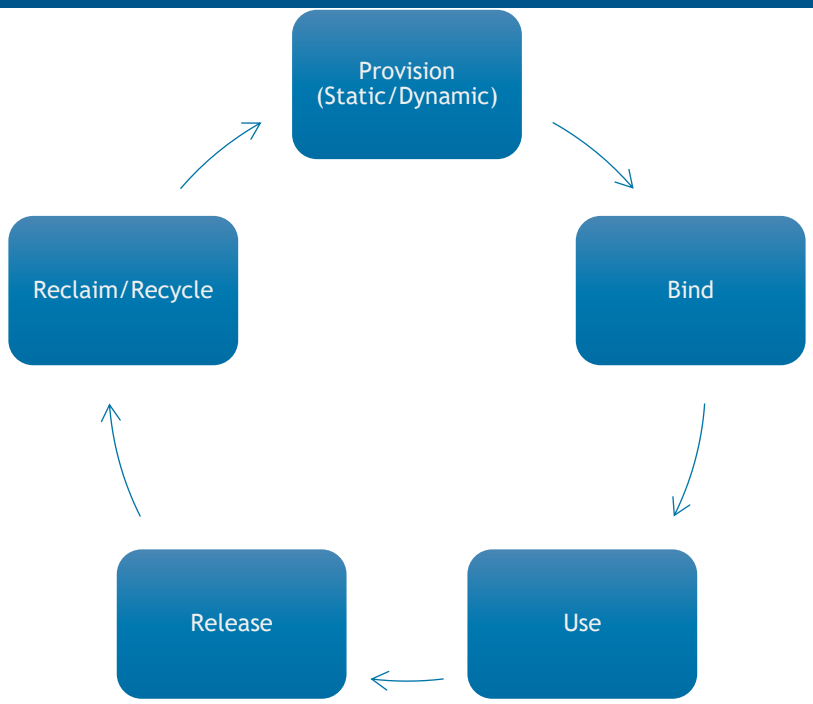
In the example above, we did not use any persistent storage for storing Redis data. This means the data will be lost if the host restarts, or the container is shifted to a new host. To make data accessible and persistent across hosts, we will need to use some sort of persistent disk. Here, we will use Google Cloud Engine persistent disk for our application before discussing other storage options.

The storage in Kubernetes is achieved using PersistentVolume mechanism, where there are three primary kinds of objects:
StorageClass: Specifies what kind of storage is available. For example, if you have a hard disk and solid state disk both available, storages available, then you might define two StorageClasses: HD, and SSD

PeristentVolume: This is the actual storage on network that is provisioned by either an administrator in advance (Static Provisioning), or on demand in by a cloud provider (Dynamic Provisioning). This object abstracts the underlying storage for the Kubernetes user; underlying storage can be any NFS, EBS, etc.

PesistentVolumeClaim: This represents the request by a pod or user for attaching a storage to a running container. If a match is found from pool of persistent volumes, then it is used. Otherwise dynamic provisioning is requested.

Note that every persistent volume follows a lifecycle which is roughly described in below diagram:



We will assume dynamic provisioning in this example. First you define a storage class to define what kind of storage you offer:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: disk-ssd
```

And then you request a claim in the container spec for the available storage classes:

```
        volumeMounts:
        - name: redis-disk
          mountPath: /cassandra-datastore
  volumeClaimTemplates:
  - metadata:
      name: redis-datastore
      annotations:
        volume.beta.kubernetes.io/storage-class: fast
```

We have used the Google Cloud Engine persistent disk in this example, but Kubernetes provides some more types of persistent volumes. The table below provides a quick overview of some key volume types:

Volume Type	Description
emptyDir	An initially empty directory used by container to store data. The data persists on host across container crashes, but if the pod is removed for any reason then data in emptyDir is deleted.
hostPath	Mounts a directory from the host into container. While most applications would not want to use this there are some useful cases for

	example a container running cAdvisor can mount and look at /dev/cgroups
gcePersistentDisk	A Google cloud engine disk which is unmounted when container is removed but can be re-mounted while data persists. A gce PD can also be mounted in READ only mode on multiple containers. The disk must be created/available in google cloud engine before it can be used.
awsElasticBlockStore	Similar to gcePersistentDisk, persists beyond container lifecycle. Must be created before using.
nfs	A NFS server which can be mounted by multiple containers. NFS allows writing to multiple containers.
Flocker	Flocker is an open source cluster volume manager. Flocker volumes are not tied to the lifecycle of the container to which it is mounted; data written to these volumes persists beyond the lifetime of the container.
gitRepo	Mounts an empty directory and then clones the content of a Git repo into that directory. Especially useful when you want to fetch configuration or standard files from a repo.
Secret	Secrets can be mounted as volumes (for example for passwords).
downloadAPI	Mounts an empty directory, then copies data from API into that directory in plain text format.

3.6 Secrets

In a large cluster where many sensitive components' data and code components might be running, it is important that the secrets and authentication are managed well. Kubernetes provides multiple ways to manage authentication between systems. We will create a secrets file in context of our application and go over service accounts.

Creating Secrets
The simplest way to create secrets is to use the kubectl command line and pass files which have value of username and password:

```
kubectl create secret generic redis-pass --from-file=./username.txt --from-file=./password.txt
```

The above command assumes username.txt has username and password.txt has the password. The username and passwords are B64 encoded and stored in object named "redis-pass" in above case. We can also manually encode the values of username/password into b64 and then create a secret object:

```
apiVersion: v1
kind: Secret
metadata:
  name: redis-pass
type: Opaque
data:
  password: c3VwZXJ0b3VnaHBhc3N3b3Jk
  username: c3VwZXJ1c2Vy
```

Using Secrets
Secrets can be used as data volumes or environment variables within a pod. A secret can be used by more than one service or pod. For example, the following code snippet shows how to use the secret as an environment variable:

```
env:
  - name: SECRET_USERNAME
    valueFrom:
      secretKeyRef:
        name: mysecret
        key: username
```

Service accounts are another way to manage communication between services or pods, and are typically used to access an API server to derive the share state of cluster and interact with cluster. By default, all calls happen through a “default” service account, but additional service accounts can be created. **Keep in mind that service accounts are meant for non-human components and should not be mixed with actual user accounts created for humans to access cluster.** A service account definition looks like this:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: elk
```

In the definition for a replication controller, you can specify the service account to be used:

```
spec:
   serviceAccount: elk
```

You can add secrets to the service account as well (for example, for fetching Docker images from a secured registry or for authenticating with certain systems).

4 Container Operations

4.1 Continuous Deployment – Service Upgrades and Rollbacks

In Kubernetes, there are two options for deployments. Rolling-updates are an older method of doing deployments; these work only with replication controllers. We recommend using the new Deployment object for any new application deployments. We will quickly go over the rolling-update method, and then cover deployment objects in more detail.

For the rolling-update method, first create a replication controller with 4 replicas and nginx version 1.7.9. Wait for the replication controller to bring all 4 pods up:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 4
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

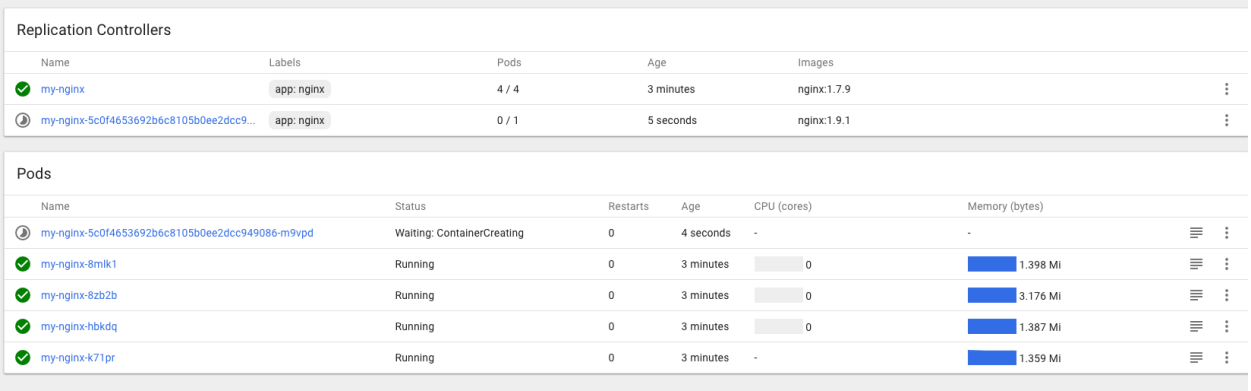
Once the RC is up, let's go to the kubectl web console in one tab, and open the replication controller section in another browser tab.

```
kubectl rolling-update my-nginx –image=nginx:1.9.1
```

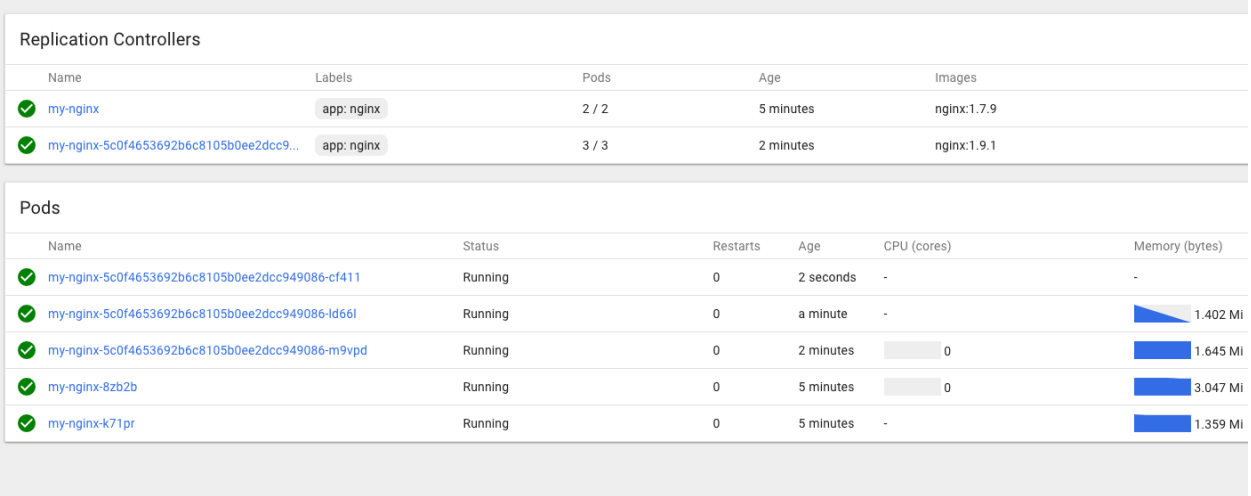
You will notice in the logs that Kubernetes decided to deploy only one pod at a time, and maintained a maximum of 5 pods at any given time.

```
> kubectl rolling-update my-nginx --image=nginx:1.9.1
W0613 17:36:52.012428     555 request.go:627] Throttling request took 189.489291ms, request: GET:http://master/api/
mespaces/default/pods?labelSelector=app%3Dnginx
W0613 17:36:52.211640     555 request.go:627] Throttling request took 193.432987ms, request: GET:http://master/api/
mespaces/default/replicationcontrollers/my-nginx-0a06b69648204650163bfcc5a2114f17
W0613 17:36:52.411648     555 request.go:627] Throttling request took 198.09957ms, request: POST:http://master/api/
mespaces/default/replicationcontrollers
Created my-nginx-0a06b69648204650163bfcc5a2114f17
W0613 17:36:52.611653     555 request.go:627] Throttling request took 185.878031ms, request: GET:http://master/api/
mespaces/default/replicationcontrollers/my-nginx
W0613 17:36:52.811635     555 request.go:627] Throttling request took 196.428577ms, request: PUT:http://master/api/
mespaces/default/replicationcontrollers/my-nginx
Scaling up my-nginx-0a06b69648204650163bfcc5a2114f17 from 0 to 4, scaling down my-nginx from 4 to 0 (keep 4 pods av
le, don't exceed 5 pods)
Scaling my-nginx-0a06b69648204650163bfcc5a2114f17 up to 1
Scaling my-nginx down to 3
```

If we switch to the Kubernetes dashboard, you will notice it has created a new replication controller and is deploying new images in the new RC.



As the update moves forward, more pods are added to new replication controller and removed from the previous one:



Essentially, rolling-updates are a way to upgrade a given number of pods at a time by moving pods from one replication controller to another.

Deployments are the recommended method for managing updates to pods and replication controllers. With a Deployment object, you specify only the target state to be achieved, and the Kubernetes deployment controller will ensure the state is changed accordingly. Specifying the desired state in a declarative manner gives Deployments a lot of flexibility and power. For example, Deployments in Kubernetes are great for the following use cases:

(1) To create a new replication controller with six nodes of nginx, you can use a Deployment and specify the desired state
(2) You can check status of a deployment for its success or for failures
(3) You can rollback an earlier deployment if the version is found to be unstable
(4) You can pause and resume a deployment – this is especially handy for executing canary deployments in which certain resources are upgraded and then deployment is paused. If the results look positive, then the deployment can be resumed (otherwise they can be aborted).
(5) You can look at history of deployment

We will first look at a couple of use cases from this list, then some parameters that can be fine-tuned to alter behavior of Deployments.

First, let's create a Deployment. Add a replication controller and paste the following block. Note that once you create a Deployment from one of the options, you won't see it in the UI but you can check it from kubectl command console. However, you will see associated pods in the appear in the UI as they're created. You can keep a tab open with kubectl command console to monitor things as they happen.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Once you create the deployment using Add Service/RC option, you can see the Deployment in kubectl:

```
> kubectl get deployment
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         3         3            3           5m
>
```

Now let's upgrade the Deployment to a higher version of nginx. Note that once upgraded, you you are unable to add the same Deployment throught the UI, as it will conflict with the previous name. So let's execute the following command in kubectl:

```
kubectl edit deployment/nginx-deployment
```

This will open an editor inside the console. Update the nginx image version from 1.7.9 to 1.9.1 and save the file. Once we have saved, we can watch the Deployment to check how the upgrade is going:

```
> kubectl edit deployment/nginx-deployment
deployment "nginx-deployment" edited
> kubectl get deployment
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         4         2            2           9m
>
>
> kubectl get deployment
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         4         2            2           9m
>
>
> kubectl get deployment
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         3         3            3           9m
>
>
```

We can also check history of replication controllers to see how the Deployment created a new replication controller and deactivated the old one:

```
> kubectl get rs
NAME                          DESIRED   CURRENT   AGE
nginx-deployment-1564180365   3         3         1m
nginx-deployment-2035384211   0         0         6m
>
```

We can rollback to previous version, since we did not provide “--record” flag while executing commands – we see nothing under “change-cause”.

```
>
> kubectl rollout undo deployment/nginx-deployment
deployment "nginx-deployment" rolled back
>
> kubectl rollout history deployment/nginx-deployment
deployments "nginx-deployment":
REVISION        CHANGE-CAUSE
2               <none>
3               <none>

>
```

The “kubectl describe deployments” command will give you a lot more information on deployments.

```
Events:
  FirstSeen     LastSeen        Count   From                            SubobjectPath   Type            Reason          M
essage
  ---------     --------        -----   ----                            -------------   --------        ------          -
------
  18m           13m             3       {deployment-controller }                        Normal          ScalingReplicaSe
t       Scaled up replica set nginx-deployment-2035384211 to 3
  9m            9m              1       {deployment-controller }                        Normal          ScalingReplicaSe
t       Scaled up replica set nginx-deployment-1564180365 to 1
  9m            9m              1       {deployment-controller }                        Normal          ScalingReplicaSe
t       Scaled down replica set nginx-deployment-2035384211 to 2
  9m            9m              1       {deployment-controller }                        Normal          ScalingReplicaSe
t       Scaled up replica set nginx-deployment-1564180365 to 2
  9m            9m              1       {deployment-controller }                        Normal          ScalingReplicaSe
t       Scaled up replica set nginx-deployment-1564180365 to 3
  9m            9m              1       {deployment-controller }                        Normal          ScalingReplicaSe
t       Scaled down replica set nginx-deployment-2035384211 to 0
  3m            3m              1       {deployment-controller }                        Normal          DeploymentRollba
ck      Rolled back deployment "nginx-deployment" to revision 1
  3m            3m              1       {deployment-controller }                        Normal          ScalingReplicaSe
t       Scaled up replica set nginx-deployment-2035384211 to 1
  3m            3m              1       {deployment-controller }                        Normal          ScalingReplicaSe
t       Scaled down replica set nginx-deployment-1564180365 to 2
  3m            3m              1       {deployment-controller }                        Normal          ScalingReplicaSe
t       Scaled up replica set nginx-deployment-2035384211 to 2
  3m            3m              3       {deployment-controller }                        Normal          ScalingReplicaSe
t       (events with common reason combined)


> 
```

4.1.1 A closer look at deployments

There are quite a few parameters in the deployment YAML file, but in this section we will look at how some of important parameters within the deployment file can be tweaked to meet different requirements.

Strategy
The strategy parameter has two types: ReCreate and RollingUpdate. Syntax for the strategy type looks like below:

```
.spec.strategy.type=Recreate
.spec.strategy.type=RollingUpdate
```

Recreate will, unsurprisingly, recreate existing pods by killing then redeploying them. Similarly, RollingUpdate will apply a rolling update to the existing pods. For RollingUpdates there are additional parameters to control how the rolling updates are done.

```
.spec.strategy.rollingUpdate.maxUnavailable
.spec.strategy.rollingUpdate.maxSurge
```

For rolling updates, only a certain number of pods are taken down and updated at one time; this process is repeated until all the pods are upgraded. MaxUnavailable specifies how many pods can be down at any time during an upgrade. This can be an absolute number, or a percentage of pods. The default value for MaxUnavailable is 1.

MaxSurge is another option which controls how many pods can be created over and above the desired number of pods. Similar to MaxUnavailable, this parameter can be set as an absolute number or as a percentage, and also defaults to 1. If we set MaxSurge and MaxUnavailable both to 20%, for ten pods, during deployment Kubernetes will immediately scale to 12 pods in total and then 2 pods will be taken out of the older pool.

```
.spec.rollbackTo
```

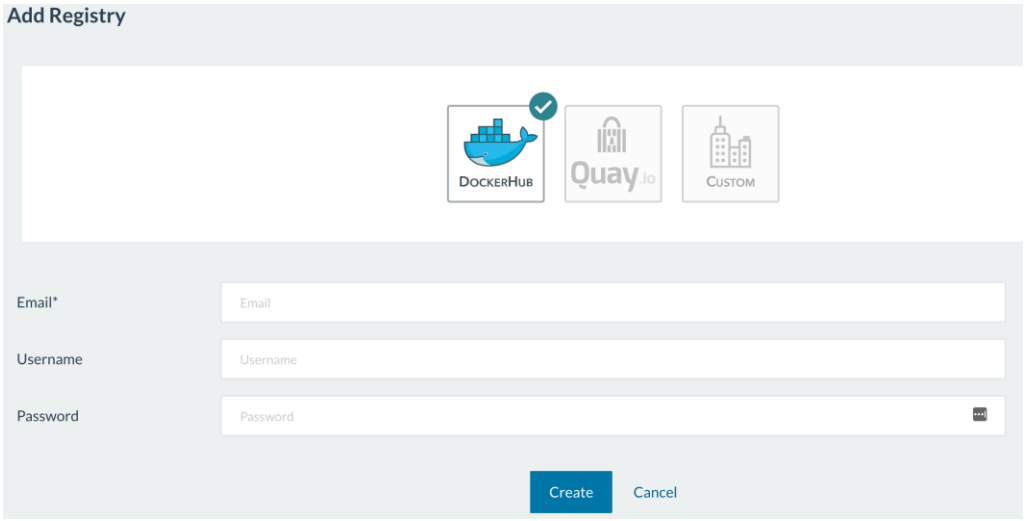
`.spec.rollbackTo.revision`

rollbackTo contains the configuration to which the Deployment is rolling back. Setting this field will cause a rollback and the field will be removed from Deployment object. You can further specify the exact “revision” to which rollback should take place. The default is considered the last version in history but specifying revision is helpful in some cases.

4.2 Rancher Private Registry Support for Kubernetes

In Rancher, you can configure private registries, and then use container images from those registries for template definitions. Rancher supports Docker Hub, Quay.io or any other private registry. You can also configure an insecure or internal certificate registry, though these require bypassing a certificate check in Docker configuration files on all nodes. Each environment requires its own registry assignment(s).

To add a new registry, go to Infrastructure → Registries. You will see if any registry is already configured before adding a new one (adding a custom repository requires entering additional information):



Once you have added a registry, you can access its container images with following format of the image:

[registry-name]/[namespace]/[imagename]:[version]

By default, the repository is assumed to be Docker Hub. If you are using an image from Docker Hub, you can use the short form of the above:

[imagename]:[version]

If a registry is insecure, you need to add an additional parameter in DOCKER_OPTS in the Docker configuration file.:

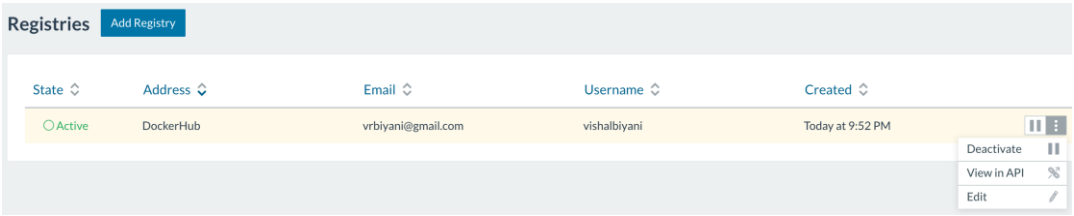
DOCKER_OPTS="$DOCKER_OPTS --insecure-registry=${DOMAIN}:${PORT}"

If you are using a certificate with the repository that is internally signed, then you need to add the certificate to certs.d directory of Docker and append the certificate to the certificate chain. This will need to be completed for all hosts and requires a restart of Docker daemon:

```
# Download the certificate from the domain
$ openssl s_client -showcerts -connect ${DOMAIN}:${PORT} </dev/null 2>/dev/null|openssl x509 -
outform PEM >ca.crt
# Copy the certificate to the appropriate directories
$ sudo cp ca.crt /etc/Docker/certs.d/${DOMAIN}/ca.crt
# Append the certificate to a file
$ cat ca.crt | sudo tee -a /etc/ssl/certs/ca-certificates.crt
# Restart the Docker service to have the changes take affect
$ sudo service Docker restart
```

Once a registry is added, there are certain actions that can be done and it is important to understand how it affects behavior.

- You can edit the registry to re-enter username, email or password
- You can deactivate a registry. After deactivation,
    - No new images can be fetched from the deactivated registry (but images already in use will continue to work).
    - Any user who has access to the environment can re-activate a deactivated registry without being asked for his or her password; if a registry should not be re-activated, it's best to delete it.



4.3 Container Monitoring

4.3.1 Monitoring with Prometheus

Prometheus is a modern and popular monitoring and alerting system, built at SoundCloud and eventually open sourced in 2012 – it handles multi-dimensional time series data really well. We will deploy a full Prometheus suite to show you what all we can monitor in a Kubernetes cluster using Prometheus. We will deploy the Prometheus stack using Helm Chart. We will discuss the Helm chart in Section 5 in detail. Please note that the official helm chart for Prometheus needs persistent disk to start. You can also check out detailed blog post on converting Cattle templates to Kubernetes templates at Rancher blog here. To install Prometheus, from Kubectl shell:

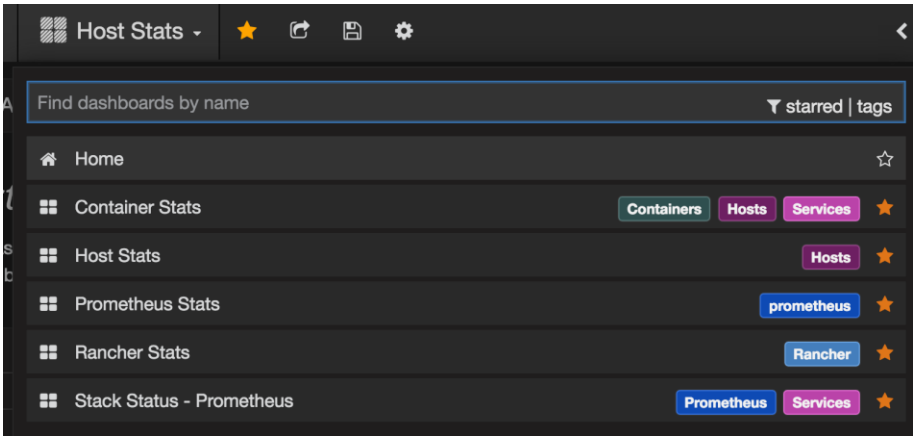
```
helm install --name prom-release stable/prometheus
```

Below is a quick overview of how each component participates in monitoring:

- Prometheus: the core component that scrapes and stores data.
- Prometheus node exporter: gets host level metrics and exposes them to Prometheus.

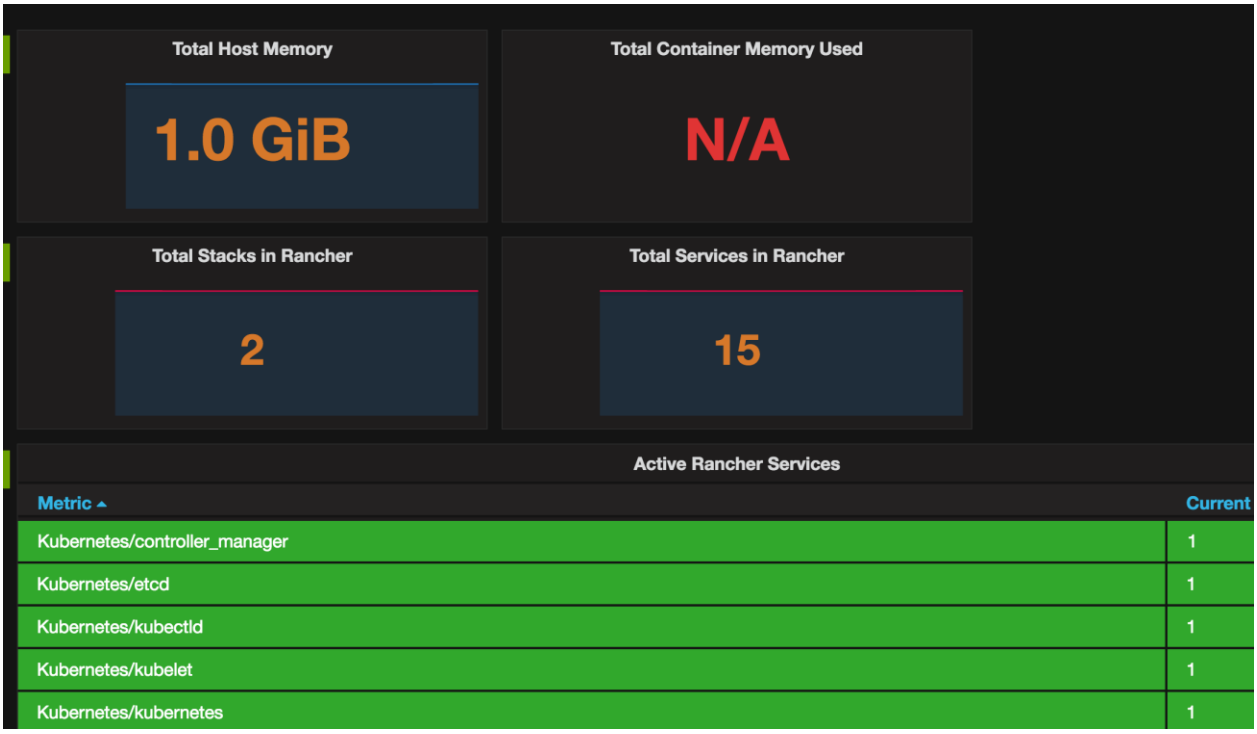
- Ranch-eye: is an haproxy and exposes cAdvisor stats to Prometheus
- Grafana: visualization for data
- InfluxDB: time series database specifically used to store data from Rancher server which is exported via the Graphite connector for Rancher.
- Prom-ranch-exporter: is a simple node.js application which helps in querying Rancher server for states of a stack or service.

Once Prometheus has been successfully deployed, we can examine the various dashboards available in Grafana to see what data is available. Note that you will have to use the IP for the host where Grafana container is deployed and NodePort for the Grafana service. Grafana has five distinct dashboards for different use case; let's cover these one by one.



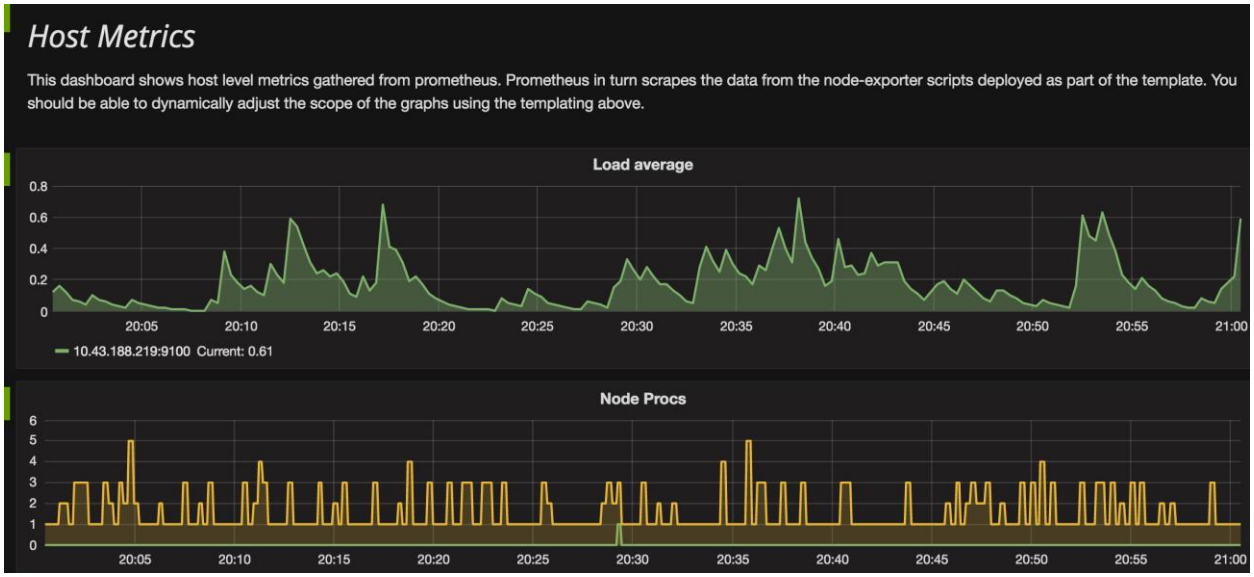
Container Stats

Container stats provides information on the host machines coming from cAdvisor and from the Prometheus Rancher exporter.



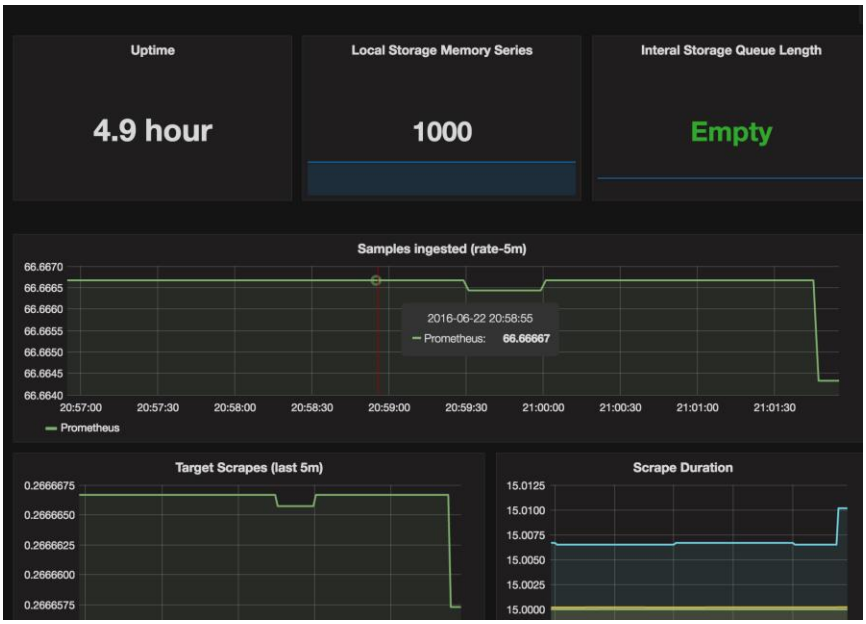
Host Stats

The host stats dashboard shows the metrics related to the host, as gathered by Prometheus.



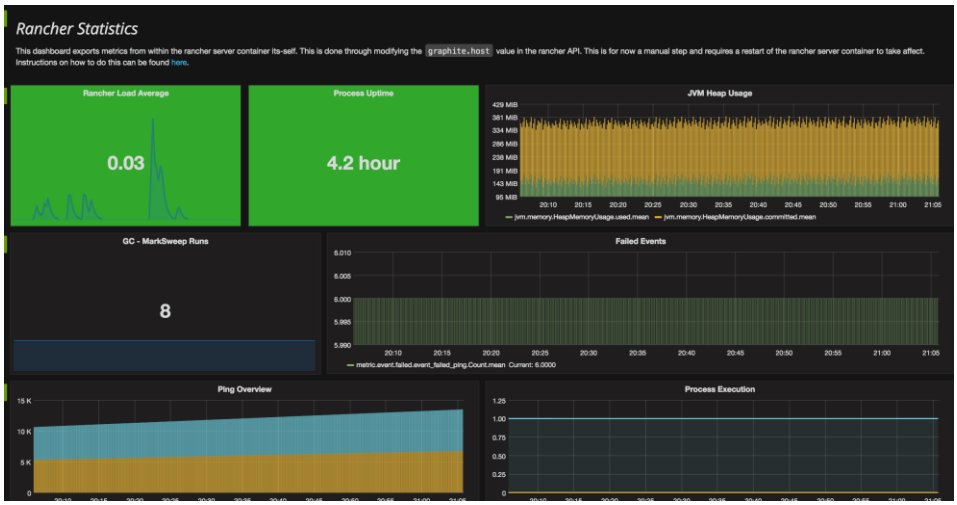
Prometheus Stats

The Prometheus Stats dashboard contains information about Prometheus itself, such as how long the Prometheus service has been running, scrape duration, and more.



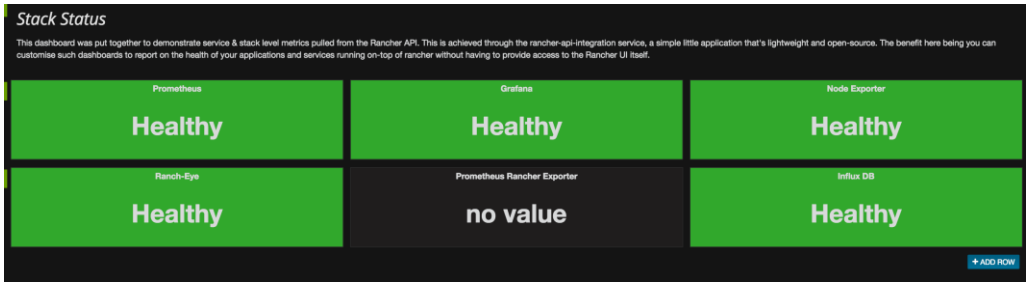
Rancher Stats

The Rancher Stats dashboard uses the Rancher graphite connector to provide information about Rancher. Rancher graphite needs to be configured to point to the InfluxDB graphite host and NodePort. Once this is done, you will need to restart the Rancher server. The metrics are stored in Rancher DB in InfluxDB and then picked up by Grafana for visualization.



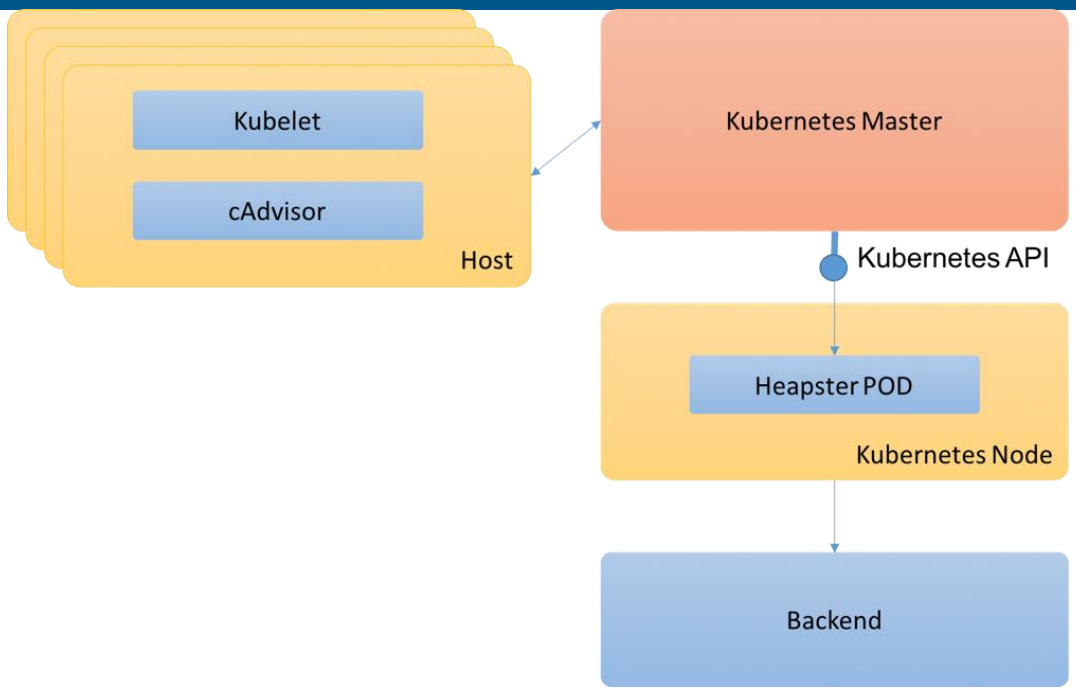
Stack Status – Prometheus

Stack status pulls information from the Rancher API about the health of individual containers.

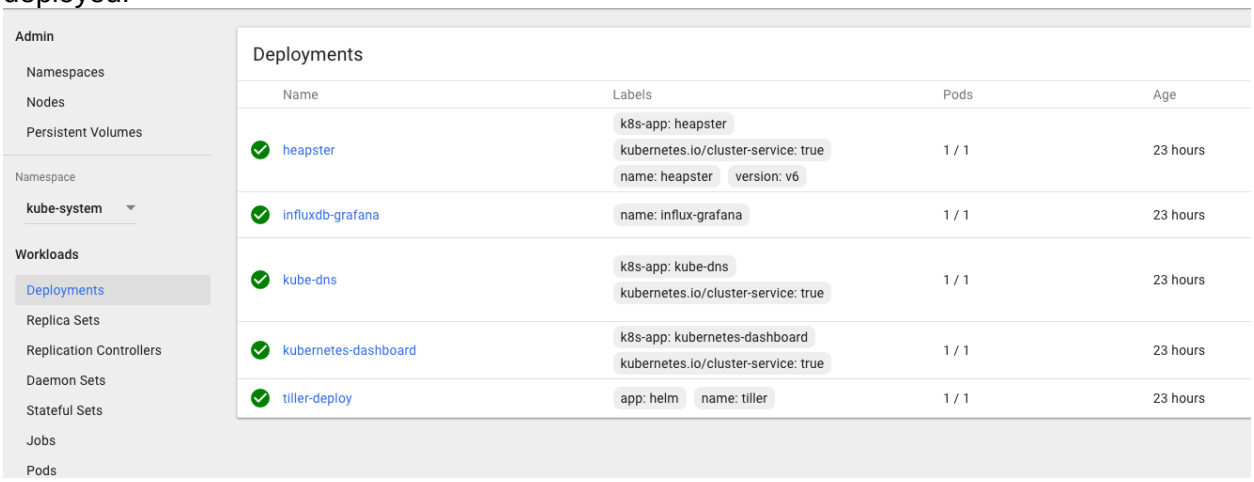


4.4 Monitoring with Heapster

Heapster is a framework for monitoring and analyzing the performance of computing clusters, and acts as an aggregator of performance data fetched from Kubernetes. It exposes a variety of metrics at the node, container, pod, and namespace level through a REST endpoint. Heapster natively supports Kubernetes and CoreOS, and a variety of backends can be used to store and visualize the data. Data for all containers on a host is fetched by cAdvisor, which is then integrated into a kubelet binary. The data is queried by Heapster and aggregated based on pods, services etc. The data is then stored into one of the configured backends, such as InfluxDB.



Heapster is already installed inside the kube-system namespace in the Kubernetes cluster created using Rancher (You can take a look at manifests for Heapster here:). If you switch to kube-system namespace in Dashboard, you will see the components related to Heapster deployed:



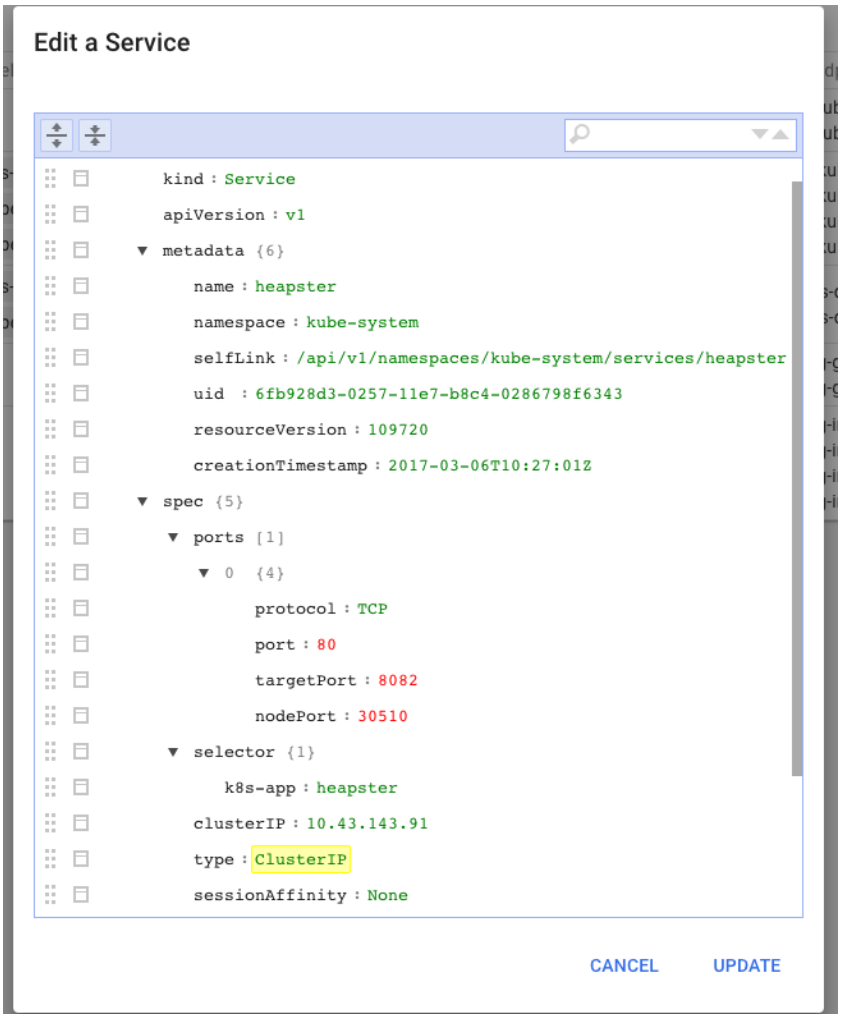
Notice that the Heapster container takes its arguments for source and target based on where the data will be stored.

```
spec:
   containers:
   - name: Heapster
     image: kubernetes/Heapster:canary
     imagePullPolicy: Always
     command:
     - /Heapster
     - --source=kubernetes:https://kubernetes.default
     - --sink=influxdb:http://monitoring-influxdb:8086
```

Change the “influxdb-grafana-controller.YAML” to use the following key-value pair (instead of what is provided in the YAML file).

```
- name: GF_SERVER_ROOT_URL
value: /
```

We will change the type of service for the Grafana Dashboard to LoadBalancer so we can access the UI of the Heapster dashboard. Edit the “Monitoring – Grafana” service and change the type from “Cluster IP” to “LoadBalancer”



Once you have setup Heapster, you will notice metrics will start flowing into the Grafana UI.

4.5 Ingress Support

Typically, services and pods have IP addresses which are only reachable within the cluster. But in many practical scenario, there is a need to provide an externally reachable URL or to provide a virtual host based on name etc. Ingress are set of rules which enable incoming connections to reach cluster services or pods. Ingress was introduced in Kubernetes 1.1 and in addition to an ingress definition, you also need an ingress controller to satisfy those rules.

Imagine an ingress resource as set of rules, and the ingress controller as the engine which satisfies those rules. Rancher comes with a built-in ingress controller and converts ingress rules into load balancing functionality. Rancher listens to server events on Kubernetes, and creates and updates the ingress controller accordingly. Let's set up a simple ingress, and then explore more complex use cases where ingress can be useful.

The first step is to create a service which we want to expose. Here we will use a simple nginx container, and create a service and replication controller for it:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  labels:
    k8s-app: nginx-service
spec:
  ports:
  - port: 90
    targetPort: 80
    protocol: TCP
    name: http
  selector:
    k8s-app: nginx-service
---
apiVersion: v1
kind: ReplicationController
metadata:
```

```
  name: nginx-service
spec:
  replicas: 1
  selector:
    k8s-app: nginx-service
  template:
    metadata:
      labels:
        k8s-app: nginx-service
    spec:
      terminationGracePeriodSeconds: 60
      containers:
      - name: nginx-service
        image: nginx:latest
        ports:
        - containerPort: 80
```

This will simply create an nginx container, wrapped with a service. Note that with this configuration, there is no way to access the service from outside of the cluster. To create, view, and delete an ingress you will have to use kubectl command line. A sample ingress definition looks like this:

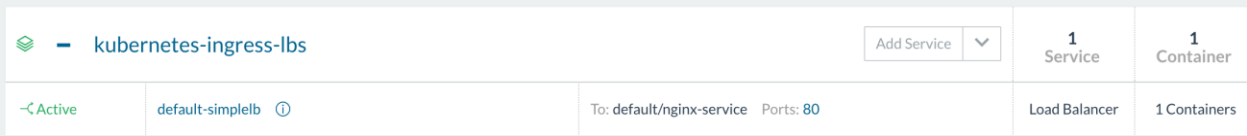
```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: simplelb
spec:
  rules:
  - host: ingress.example.com                    ## The incoming host name
    http:
      paths:
      - path: /test                              ## Path to service
        backend:
          serviceName: backend-service
          servicePort: 90                        ## service_port
```

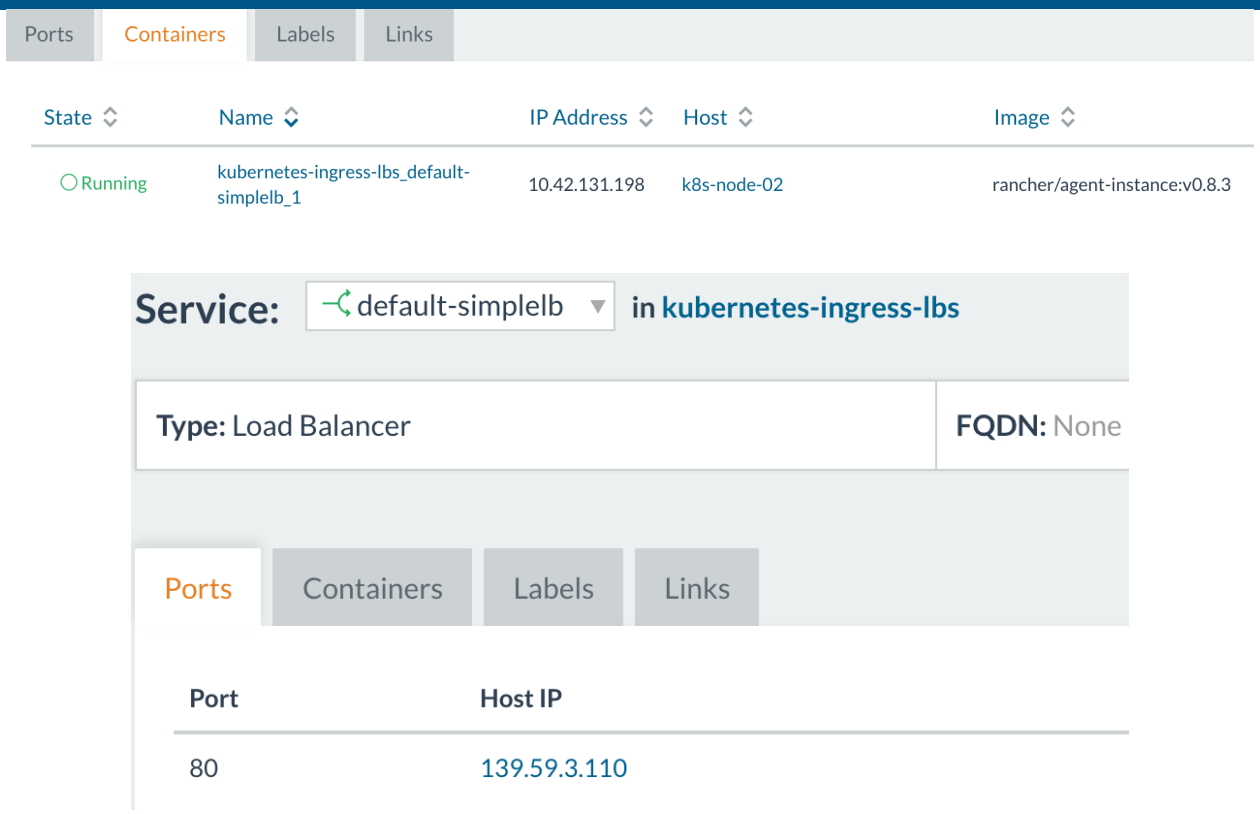
In the above example, we have defined a single rule with a host, path and corresponding backend service and port. We can define multiple host and path combinations pointing to backend services.

For our nginx service, we will define simple ingress as shown in the code snippet below. Since host is not specified, it defaults to an asterisk “*” and the path defaults to root path.

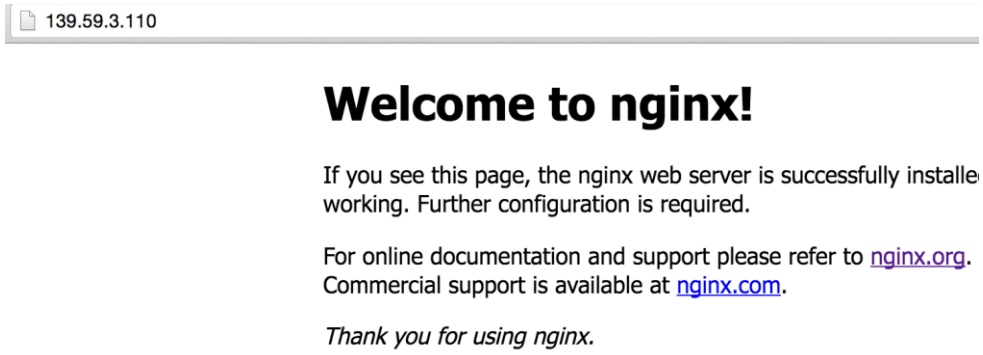
```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: simplelb
spec:
  backend:
    serviceName: nginx-service
    servicePort: 90
```

Once you have created the ingress rule, you will notice that Rancher automatically creates a corresponding load balancer. To view the load balancer, you can go to the Kubernetes System view.





As you can see, a load balancer and appropriate routing has been created by Rancher based on ingress rules definition. If we now click the Host IP at port 80, we can see the nginx hompage:



4.5.1 Ingress Use cases

Host based routing
Multiple host URLs can be diverted to different services by defining ingress rules through a single load balancer.

```
- host: foo.bar.com
  http:
    paths:
```

```
      - backend:
          serviceName: nginx-service-1
          servicePort: 90
  - host: bar.foo.com
    http:
      paths:
      - backend:
          serviceName: nginx-service-2
          servicePort: 90
```

Path based routing
Different paths of a service can be load-balanced with ingress rules:

```
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: nginx-service-1
          servicePort: 90
      - path: /bar
        backend:
          serviceName: nginx-service-2
          servicePort: 90
```

Scaling ingress
In default configuration, the ingress rules definition will create a single load balancer on a single host. You can use annotations to create multiple load balancers spread across multiple hosts and using a non-default port. For the following example to work, you will need at least two Kubernetes nodes with port 99 available:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: scaledlb
  annotations:
    scale: "2"
    http.port: "99"
spec:
```

Ingress with TLS
Ingress can be also be set up with TLS. As a prerequisite, you will need to upload the certificate in Rancher. You can then use TLS by specifying the port and the secretname in the tls field:

```
metadata:
  name: tlslb
  annotations:
    https.port: "444"
spec:
  tls:
   - secretName: foo
   backend:
```

4.6 Container Logging

With large container deployments, it is important to collect log data from the containers for analysis. The container logs might contain valuable information for finding the root cause of any issues. In a Kubernetes cluster, there are a few functionalities required to build a logging platform:

- Collecting logs from containers running on a host
- Forwarding the logs to a central logging platform
- Formatting, filtering, analyzing and visualizating the logs.

There can be more areas in the platform such as alerting, monitoring etc. but we will not cover those here.

For analysis and visualization, we will use the famous ELK (Elasticsearch, Logstash, Kibana) stack. To collect and forward the logs to the logging platform, we will use LogSpout (though there are other options like FluentD available).

If you are running Kubernetes on Google Cloud Platform, you also have the option of using Google cloud logging services to analyze and visualize the logs (in addition to using the ELK stack on Google Cloud).

4.6.1 Using ELK Stack and logspout

Let's first set up a simple ELK cluster on Kubernetes. In the real world, Elasticsearch should be set up with data, client and master as separate components so that they can be scaled easily (there are templates and detailed documentation for Kubernetes here). For Logstash and Kibana, we will use the images and templates developed by Paulo here with some minor modifications.

Let's first create service for Elasticsearch. Note that we are creating the service in the "elk" namespace.

```
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch
  namespace: elk
  labels:
    component: elasticsearch
spec:
  type: LoadBalancer
  selector:
    component: elasticsearch
  ports:
  - name: http
    port: 9200
    protocol: TCP
  - name: transport
    port: 9300
    protocol: TCP
```

Similarly, the replication controller is also created in the elk namespace.

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: es
  namespace: elk
  labels:
    component: elasticsearch
```

```
spec:
  replicas: 1
  template:
    metadata:
      labels:
        component: elasticsearch
    spec:
      serviceAccount: elasticsearch
      containers:
      - name: es
        securityContext:
          capabilities:
            add:
              - IPC_LOCK
        image: quay.io/pires/docker-elasticsearch-kubernetes:1.7.1-4
        env:
        - name: KUBERNETES_CA_CERTIFICATE_FILE
          value: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
        - name: NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        - name: "CLUSTER_NAME"
          value: "myesdb"
        - name: "DISCOVERY_SERVICE"
          value: "elasticsearch"
        - name: NODE_MASTER
          value: "true"
        - name: NODE_DATA
          value: "true"
        - name: HTTP_ENABLE
          value: "true"
        ports:
        - containerPort: 9200
          name: http
          protocol: TCP
        - containerPort: 9300
          name: transport
          protocol: TCP
        volumeMounts:
        - mountPath: /data
          name: storage
      volumes:
      - name: storage
        emptyDir: {}
```

As you may have noticed, we are using emptyDir type storage. This type is good for our quick demo, but for real-world use cases, it is preferable to use persistent volumes.

The templates for Logstash and Kibana are very similar and you can check them out here. However, let's look at some of the important aspects before we launch the full stack:

For Kibana, we are provide the Elasticsearch URL as an environment variable:

```
        - name: KIBANA_ES_URL
          value: "http://elasticsearch.elk.svc.cluster.local:9200"
        - name: KUBERNETES_TRUST_CERT
          value: "true"
```

We are using a custom Logstash image, and again the Elasticsearch URL is provided as an environment variable:

```
        - name: ES_HOST_NAME
          value: http\:\/\/elasticsearch.elk.svc.cluster.local\:9200
```

Once all three containers are up and running you can do a quick verification with following checks and confirm there are no issues in ELK setup.

- Elasticsearch will be reachable at container port 9200 and corresponding host/NodePort combination if the stack has been set up correctly. You should see text similar to the one below in response:

```
{
  "status" : 200,
  "name" : "Master Mold",
  "cluster_name" : "myesdb",
  "version" : {
    "number" : "1.7.1",
    "build_hash" : "b88f43fc40b0bcd7f173a1f9ee2e97816de80b19",
    "build_timestamp" : "2015-07-29T09:54:16Z",
    "build_snapshot" : false,
    "lucene_version" : "4.10.4"
  },
  "tagline" : "You Know, for Search"
}
```

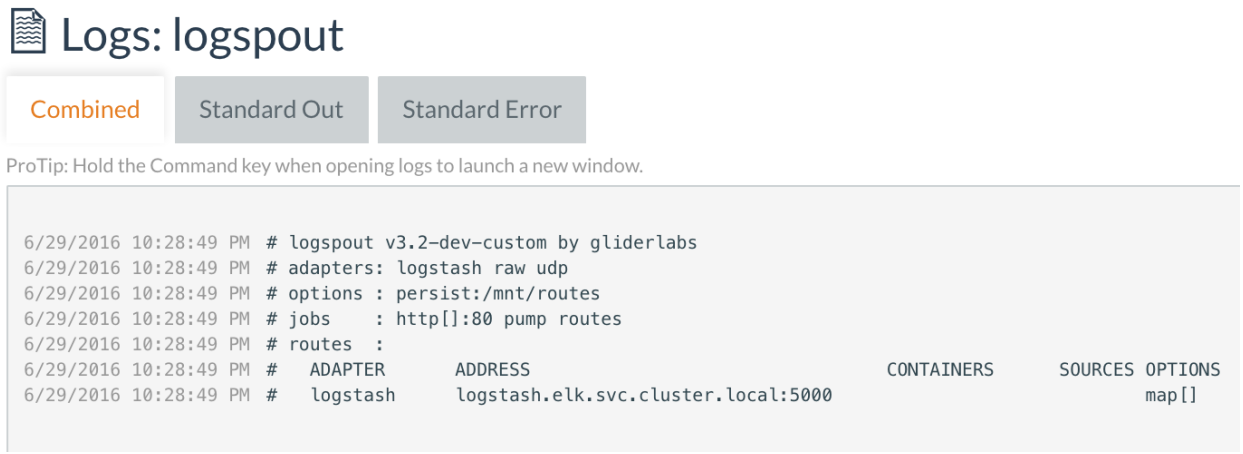
- Kibana UI will be reachable at container port 5601 and corresponding host/NodePort combination. When you begin, there won't be any data in Kibana (which is expected as you have not pushed any data).

The next step is to set up the log aggregator from container hosts, and forward them to Logstash for further processing. Here, we will use Logspout, a generic log forwarder. Logspout runs inside a Docker container itself, and attaches to all containers running. The logs from all containers are forwarded to the configured destination. In our case, the destination is Logstash so we will need Logstash plugin. We have built a custom Docker image which has Logstash plugin baked in (source code here). Logspout takes a parameter "ROUTE_URIS" which is where the logs are routed to. Following is the template for the replication controller for LogSpout:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: logspout
  namespace: elk
  labels:
    component: elk
    role: logspout
spec:
  replicas: 1
  selector:d
    component: elk
    role: logspout
  template:
    metadata:
      labels:
        component: elk
        role: logspout
    spec:
      serviceAccount: elk
```

```
containers:
- name: logspout
  image: infracloud/logspout-Logstash-docker
  env:
  - name: ROUTE_URIS
    value: "Logstash://Logstash.elk.svc.cluster.local:5000"
  volumeMounts:
  - mountPath: "/var/run/docker.sock"
    name: dockersock
volumes:
- hostPath:
    path: "/var/run/docker.sock"
  name: "dockersock"
```

For this to work, Logstash needs to be listening in on port 5000 with UDP. We have already put this configuration into our Docker container for Logstash. Now you should see the following logs in LogSpout pod:



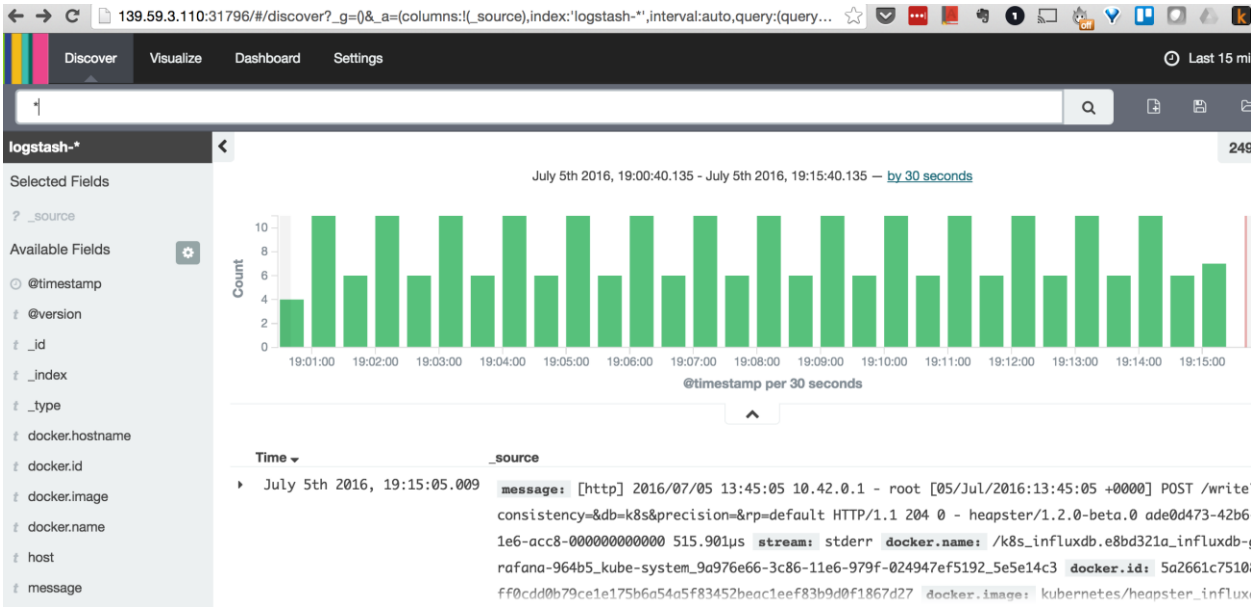
And in Logstash:

Assuming all above steps succeeded, data should be flowing in Elasticsearch now. We can verify this by looking at indexes of Elasticsearch:

139.59.3.110:30102/_cat/indices?v

```
health status index               pri rep docs.count docs.deleted store.size pri.store.size
yellow open   logstash-2016.07.03   5   1      25350            0      5.2mb          5.2mb
yellow open   logstash-2016.06.30   5   1      25618            0      5.4mb          5.4mb
yellow open   .kibana               1   1          2            0      5.7kb          5.7kb
yellow open   logstash-2016.06.29   5   1      11045            0      2.7mb          2.7mb
yellow open   logstash-2016.07.01   5   1      25867            0      5.4mb          5.4mb
yellow open   logstash-2016.07.02   5   1      25470            0      5.3mb          5.3mb
yellow open   logstash-2016.07.04   5   1      40261            0      7.7mb          7.7mb
yellow open   logstash-2016.07.05   5   1      14558            0      6.4mb          6.4mb
```

The final step is to configure which indexes Kibana should look for in Elasticsearch. Once you have configured the pattern with Logstash-*, you will see the log data trending in default dashboard of Kibana:



What we have set up is a simple data pipeline, with all components required for aggregating, logging, analyzing, and visualizing log data.

4.7 Auto Scaling

We can achieve auto scaling in Kubernetes in a Rancher environment using Kubernetes native auto scaling.

In Kubernetes, manual scaling can be implemented using the “kubectl scale” command on almost all resource types. Auto scaling in Kubernetes is achieved by using the Horizontal Pod Autoscaling (HPA). Let’s take a closer look at how HPA works.

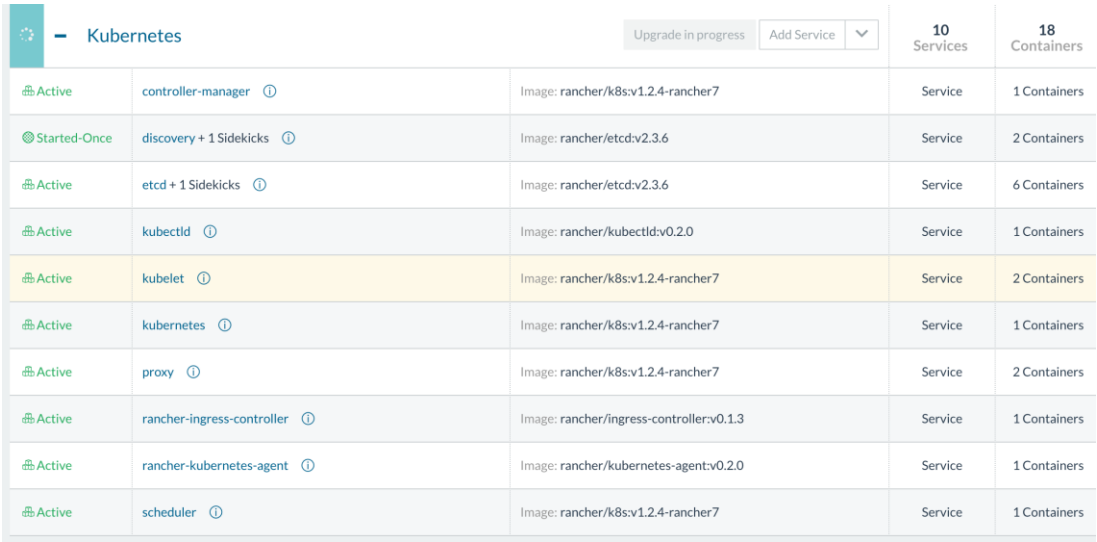
- HPA Prerequisites:
  - HPA is triggered by pod CPU utilization.
  - Heapster is required for querying CPU utilization

    - The containers in the pod must have a CPU request set; HPA will not considering the pods without a CPU request for auto scaling.
- Working
    - HPA is designed as an API resource and controller – resource invokes actions on controller and controller makes sure that target state is achieved.
    - HPA watches CPU of pods at an interval which is controlled by parameter --*horizontal-pod-autoscaler-sync-period* in controller manager.
    - The actual CPU utilization is compared with the one requested by pod containers based on algorithm here
    - To change the state to desired state a new resource called “Scale” was introduced. The scale resource can be queried for RC/deployment/Replica Set to retrieve current number of resources and then send request to adjust desired number of resources.
    - Please note that some of these resource types and APIs were introduced in versions as latest as 1.2 so the grouping of APIs might change in later versions.
- Via kubectl
    - The auto scaling feature can also be interacted with by kubectl CLI. For example to scale a replication controller named frontend when the CPU utilization hits 85%:

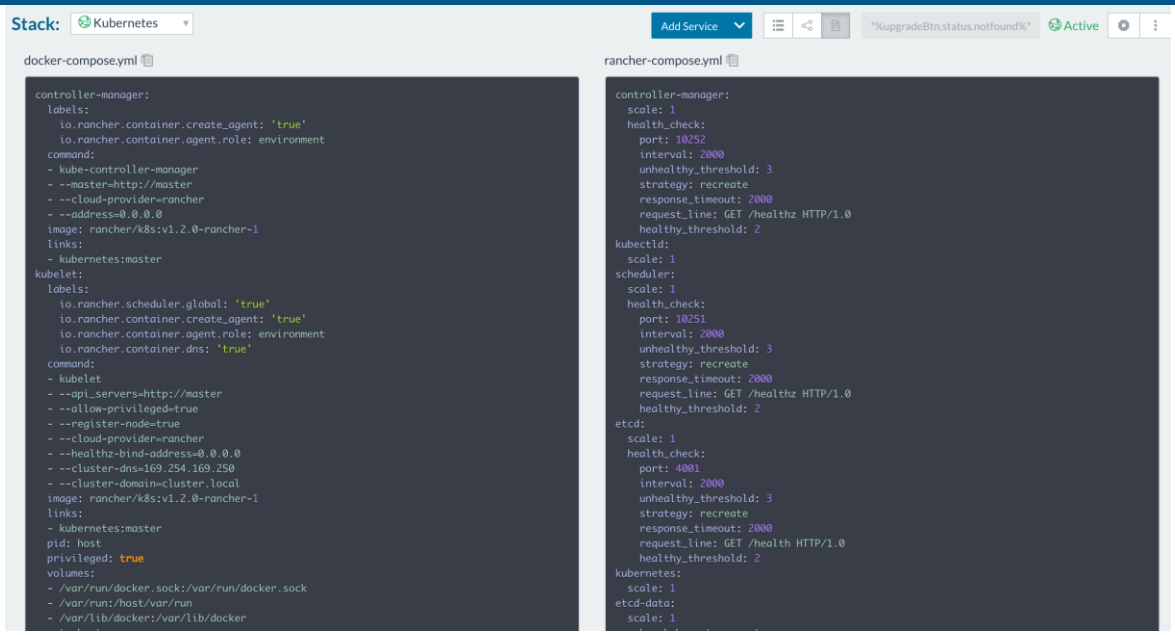
```
kubectl autoscale rc frontend --max=7 --cpu-percent=85
```

4.8 Kubernetes System Stack Upgrades in Rancher

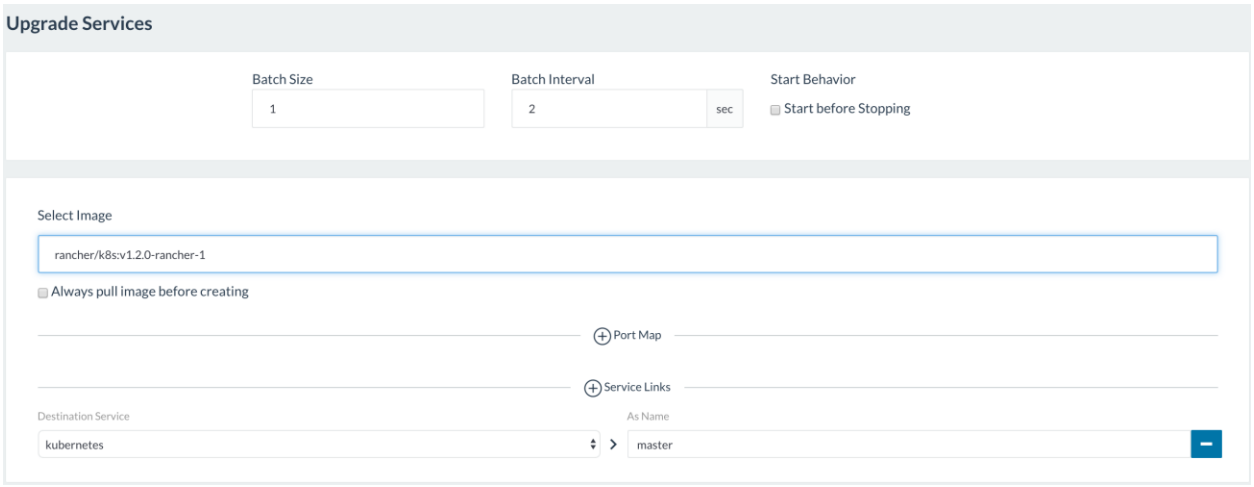
If you look at the System menu in Kubernetes tab, you will see all containers which running in the Kubernetes stack. This entire stack has been deployed via Rancher:



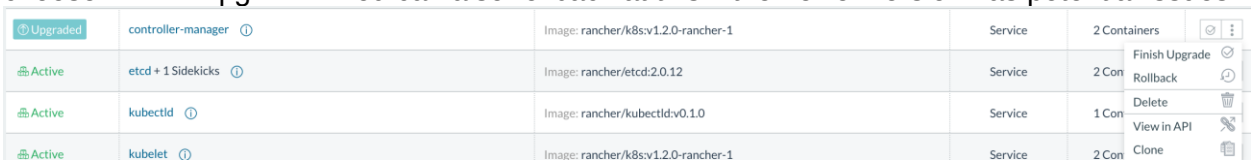
By clicking on the Compose YAML icon, you can also view the docker-compose and rancher-compose files which together have created the Kubernetes stack:



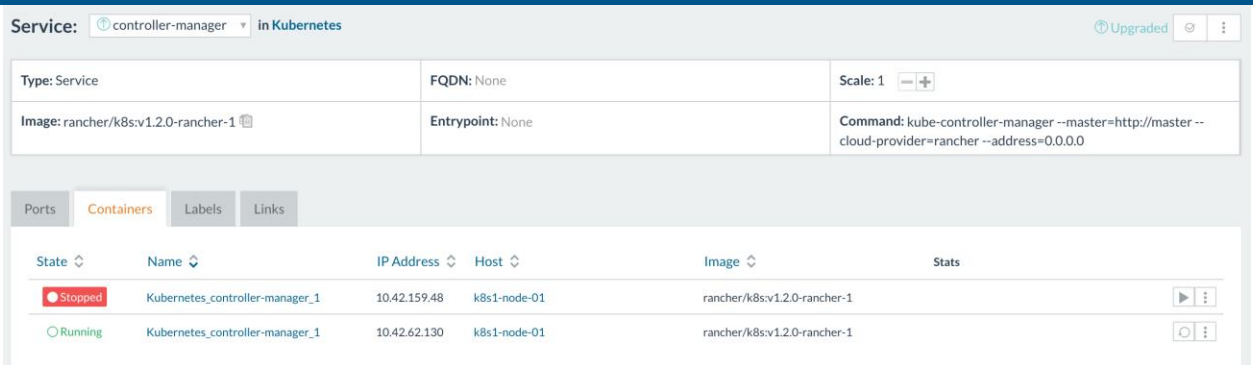
As you can see, the containers used for the Kubernetes stack use specific versions of Kubernetes and Rancher. For example, the controller-manager container uses Rancher/k8s:v1.2.0-Rancher-1. If we want to upgrade this container to a newer version, we can go to the context menu on right hand side, and choose upgrade (or simply click on upgrade icon). We can upgrade the version of container and control certain options such as batch size and the delay between batch upgrades.



The upgrade process creates a new version of the container and keeps it on standby until you choose “Finish upgrade”. You can also rollback at this if the newer version has potential issues.

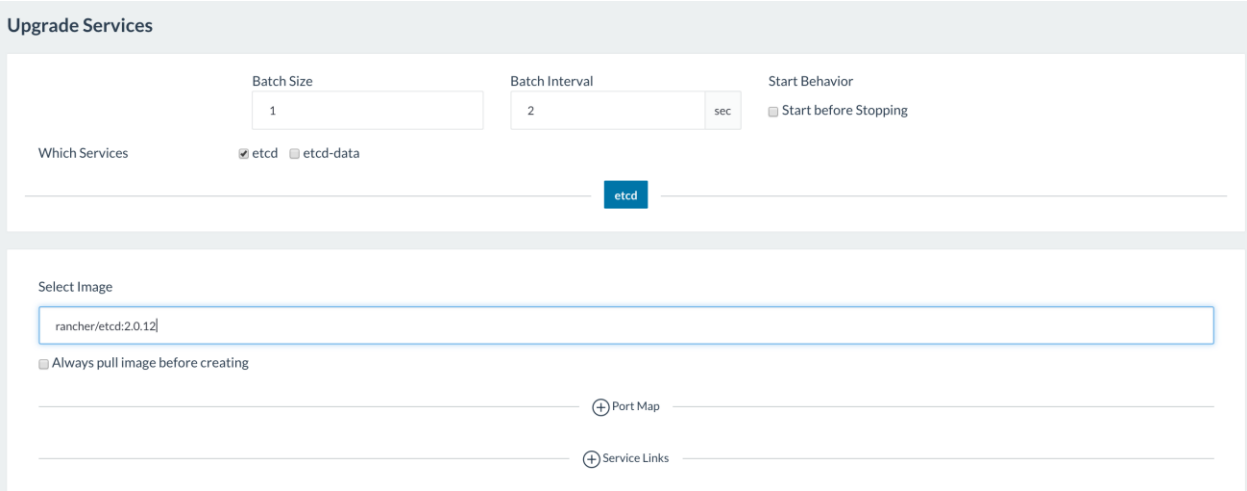


Let’s look at details of controller-manager – you will notice that there is one node running, one stopped, and that the upgrade is waiting for confirmation on top right hand side. At this point, we can inspect the logs to verify the upgrade and decide to upgrade or rollback.

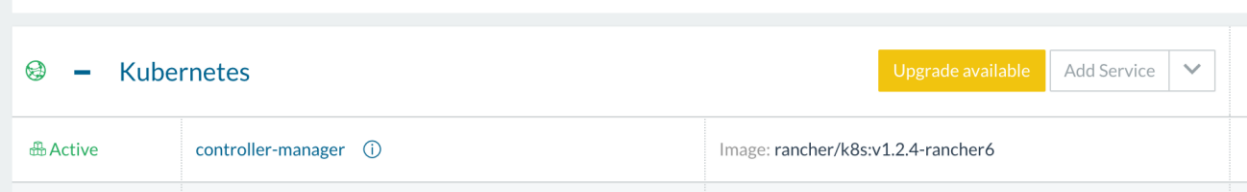


If upgrade option is chosen – the new container will continue running and old one will be discarded. If rollback is chosen, then the inverse happens: the new one is stopped and old one is reinitialized.

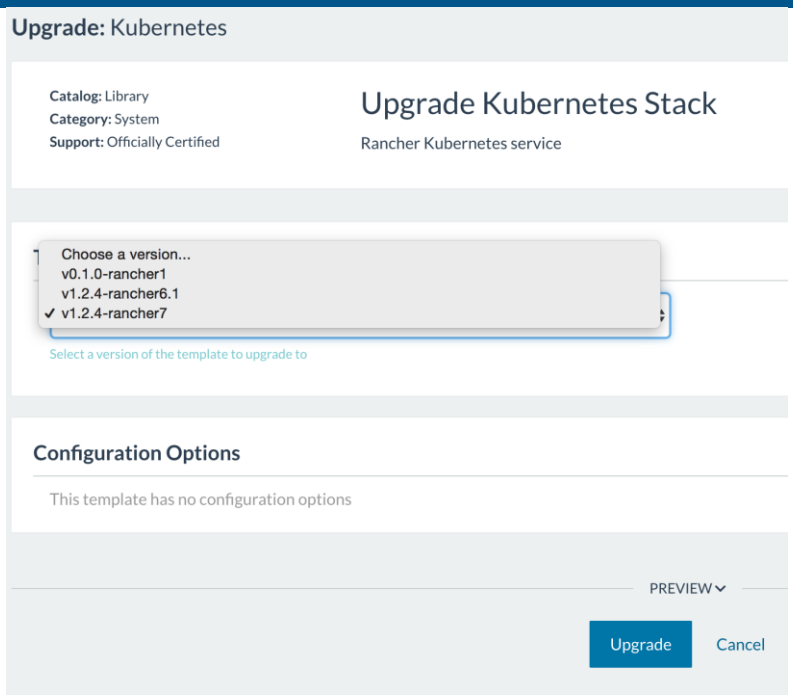
If a container has a sidekick or data-container attached to it, you can choose to upgrade both of them together too.



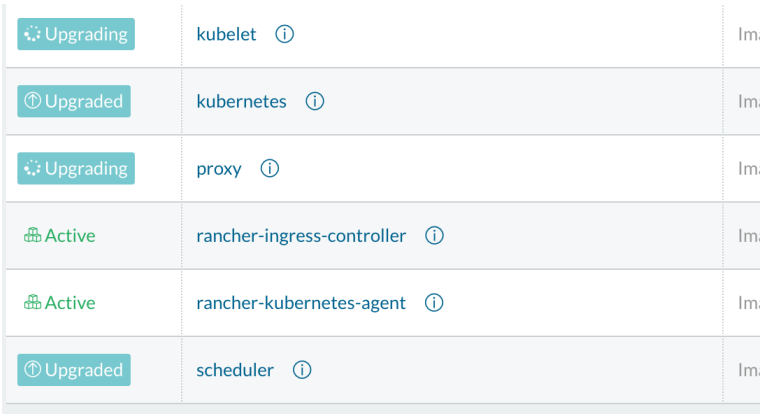
Although it is possible this way to upgrade individual components, it is not advisable to do so. Starting with Rancher v1.1, Rancher provides an “Upgrade available” button right at top of Kubernetes cluster when the upgrade is available:



You can choose the versions of Kubernetes and Rancher for the upgrade:



Rancher then will only upgrade components which have changed:



It is recommended to upgrade the entire Kubernetes stack through the upgrade menu, as this method has been tested and verified to work with all components in the stack.

5 Managing packages in Kubernetes

5.1 Introduction to Helm and Charts

When we deployed some sample apps in earlier sections, you might have noticed that we had to deploy multiple services to create a complete application. In a way, multiple manifests can be grouped into a single package which can be more easily shared. This is the exact problem solved by Helm, the package manager for Kubernetes.

Let's first understand some basic terminology Helm package management uses:

> A **chart** is a package in Helm terminology that contains all that is needed to run a specific application on Kubernetes
>
> **Tiller** is the server that runs inside the Kubernetes cluster and handles installing, managing and tracking the charts in the cluster. Helm is the client portion, or CLI, with which users interact and issue commands. Helm client talks to Tiller server.

Charts are stored in a repository, typically in a version-controlled source control system such as Github. The official chart repository can be found here. Finally, when a user installs a chart into a cluster, a new release is created and tracked.

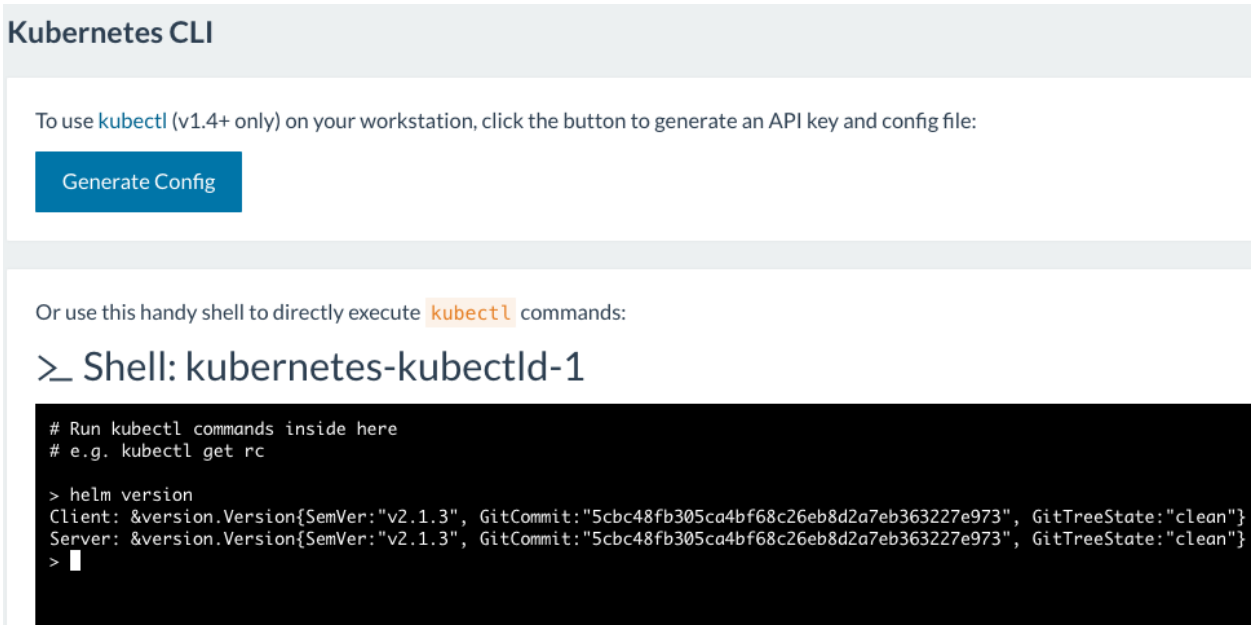
5.2 Structure of Helm Charts

A chart for a given application can contain the following files:

Chart.yml	Information about the chart in YAML format
LICENSE	License info (if applicable) for chart
README.md	README file
values.yml	Configuration values which could be used as default if the user does not provide any values
charts/	Any other charts on which this chart depends
templates/	Template files which are combined with values from values.yml and CLI provided values to produce manifests
templates/Notes.txt	Any info about usage of charts

5.3 Using Helm

Rancher comes with Tiller, Helm, and the Kubernetes CLI built-in. If you navigate to Kubernetes CLI in Rancher UI, you can quickly verify that Helm is configured appropriately:



To search charts in the repo:

```
> helm search mysql
NAME            VERSION DESCRIPTION
stable/mysql    0.2.5   Fast, reliable, scalable, and easy to use open-...
stable/mariadb  0.5.9   Fast, reliable, scalable, and easy to use open-...
>
```

To get more information about a specific chart you can inspect the chart:

```
> helm inspect stable/mysql
description: Fast, reliable, scalable, and easy to use open-source relational database
  system.
engine: gotpl
home: https://www.mysql.com/
icon: https://www.mysql.com/common/logos/logo-mysql-170x115.png
keywords:
- mysql
- database
- sql
maintainers:
- email: viglesias@google.com
  name: Vic Iglesias
name: mysql
sources:
- https://github.com/kubernetes/charts
- https://github.com/docker-library/mysql
version: 0.2.5

---
## mysql image version
## ref: https://hub.docker.com/r/library/mysql/tags/
##
imageTag: "5.7.14"

## Specify password for root user
##
## Default: random 10 character string
# mysqlRootPassword: testing
```

Now let's install the Jenkins chart so we can track it later:

```
> helm install --name jenkins-r1 stable/jenkins
NAME:   jenkins-r1
LAST DEPLOYED: Tue Mar  7 12:18:26 2017
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Secret
NAME                TYPE      DATA      AGE
jenkins-r1-jenkins  Opaque    2         0s

==> v1/ConfigMap
NAME                DATA      AGE
jenkins-r1-jenkins  1         0s

==> v1/Service
NAME                CLUSTER-IP     EXTERNAL-IP   PORT(S)                          AGE
jenkins-r1-jenkins  10.43.41.103   <pending>     8080:30068/TCP,50000:32388/TCP   0s

==> extensions/Deployment
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
jenkins-r1-jenkins  1         0         0            0           0s

==> v1/PersistentVolumeClaim
NAME                STATUS    VOLUME    CAPACITY   ACCESSMODES   AGE
jenkins-r1-jenkins  Pending                                      0s
```

We can then get the status of latest deployment

```
>
> helm status jenkins-r1
LAST DEPLOYED: Tue Mar  7 12:18:26 2017
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> extensions/Deployment
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
jenkins-r1-jenkins  1         0         0            0           1m
```

What we have done is installed the stock Jenkins chart without any customization. In the real world we need to customize the package by changing certain parameters. The default parameters are in values.yml and we can override that by using a custom values.yml file.

```
helm install --name jenkins-r1 -f values.yaml stable/jenkins
```

6 Additional Resources

Documentation

- Kubernetes documentation: primary resource for Kubernetes concepts, components, and best practices
- Rancher documentation: primary resource for Rancher concepts, components, and best practices
- Helm Documentation: Provides in depth information about using Helm package manager in Kubernetes

Rancher Articles and Walkthroughs

The following articles, previously published on Rancher.com, may be of interest for users interested in running containerized applications at scale, and making the most of Rancher's capabilities

- Launching Microservices Deployments on Kubernetes with Rancher: an additional quick walkthrough on using Kubernetes with Rancher
- Converting a template from Cattle to Kubernetes: walks through the process of taking a template used in Rancher's Cattle framework, and making it easily deployable in a Kubernetes environment.
- Rancher Controller for the Kubernetes Ingress Feature: An in-depth look on how Rancher works with one of the key features of Kubernetes
- Creating a MongoDB Replica Set with the Rancher Kubernetes Catalog: a walkthrough introduction to using replica sets and the Rancher Catalog.

Additional eBooks

- Building a CI/CD Pipeline with Rancher and Docker: a hands-on guidebook for introducing containers into your build, test, and deployment processes
- Comparing Kubernetes, Mesos, and Docker Swarm: an in-depth guide comparing the most common orchestration tools on the market

7 About the Authors

Girish Shilamkar is founder and CEO at infraCloud technologies. Girish started hacking on distributed file systems right from his university days. Girish has worked on HPC file systems, Lustre, GPFS and has multiple contributions to Linux Kernel & Apache Cloudstack projects. Girish has worked with startups to build storage and cloud platforms while providing technical leadership and building high-performance teams. Girish's interests are Containers, Microservices, Cloud adoption and migration, QA Engineering. When not at screen, Girish can be found on the golf course, driving through mountains.

Vishal Biyani is founder & CTO at infraCloud technologies. Vishal has worked across the whole spectrum of SDLC from developing code to deploying it and supporting customer tickets. Vishal's roles spanned from consulting Fortune 500 customers on DevOps assessment to hands down platform building for internet scale companies. Vishal is a DevOps practitioner, likes to work in Agile environments with focus on TDD. Vishal's interests span continuous delivery, enterprise DevOps, containers and security. When not typing, Vishal can be found cycling, photographing or flipping pages.

infraCloud technologies is a programmable infrastructure company with leadership in building highly scalable Container, Cloud & DevOps solutions. infraCloud is also a Rancher partner