WORKBOOK

PVD903-RKE2 WORKBOOK

Pascal van Dam

Pascal Van Dam

"Let us orchestrate your success!" #K8SMastery

Author(s): Pascal van Dam

Contents:

1 Install RKE2 for single CP K8S cluster

1.1 Introduction

In this lab we get you familiar with the RKE2 installation by setting up a RKE2 cluster consisting of a single master node and 1 or 2 worker nodes. Please remember that in RKE2 lingo, a master- or controlplane node is called a `rke2-server` and a (worker-) node is called a `rke2-agent`.

1.1.1 Requirements

To be able to execute this lab, you need at least 2 Ubuntu 22.04 (LTS) systems. If you would like to add more `rke2-agent` nodes you can always add more machines. The virtual machines must be provided with at least 4 GB Memory, 2 CPUs and enough storage to store some containers (approximately 8 GB free space should be ok). This lab has been tested on both x86_64 (AMD64) as wel as aarch64 (ARM64) systems. So they should also work on ARM64 architectures like RPi4, RPi5, ARM64 ODROIDs, RockPi4, OrangePi4 (plus) etc. RKE2 does not support RISCV64 yet.

Furthermore, you need an Internet connection to be able to access the RKE2 distribution and download some containers from quay.io and docker.io.

In this lab we assume that there is an unprivileged user, named 'student01' available on this system. This user should be able to execute privileged commands using the `sudo` utility. Of course your user may have a different name, important is the access to `sudo su - root` privileges.

If not already done, you can create on all nodes this user as root:

```
▶ useradd -m -s /bin/bash student

▶ passwd student01

▶ echo "student ALL=(ALL) NOPASSWD: ALL" | tee /etc/sudoers.d/89-student
```

Login again as the user student, and proceed this lab.

Bring your installation up-to-date, using:

```
▶ sudo apt update

▶ sudo apt upgrade -y
```

If systemd or the kernel is updated during this process, you need to reboot your system. If you are not sure: reboot it anyway.

```
▶ sudo systemctl reboot
```

Install the following software requirements:

```
▶ sudo apt -y install vim info wget curl elinks man-db manpages \
      bash-completion psmisc jq ipvsadm yamllint conntrack \
      apt-transport-https pinfo
```

Execute the update of your system and the installation of the software on each node.

1.1.2 Network Configuration

It is assumed that you have internet connection and for this lab that the local system firewall (UFW on Ubuntu) is disabled.

Please ensure it is disabled:

```
▶ sudo systemctl disable ufw --now
```

No further tuning of the network stack like with `kubeadm` is needed as the RKE2 installer binary will take care of it.

1.1.3 Swap

A requirement for Kubernetes is to have swap disabled, otherwise the Kubelet service will not start on the masters and nodes. The idea of Kubernetes is to directly terminate overcommitting workloads as not to jeopardize the workload on the K8S cluster that nicely abides to the rules. Remember the principle of `Cattle vs. Pets` in K8S.

```
▶ sudo systemctl disable --now swap.target
▶ sudo systemctl mask swap.target

▶ sudo sed -i '/swap/d' /etc/fstab

▶ sudo swapoff -a
```

Again, repeat this on every node.

1.2 Install RKE2

We will now use a 2 step process to install K8S using the RKE2 distribution on your systems:

1. Install RKE2 server on the system what will fulfill the role of master node for our K8S cluster.
2. Join the other servers that will play the role of (worker-)nodes by installing and configuring rke2-server on these.

In the examples shown here under we will have 3 systems:

Nr	Hostname	Role	Config
1	k8sc903n01	RKE2 server	2cpu, 8GiB RAM
2	k8sc903n02	RKE2 agent #1	2cpu, 8GiB RAM
3	k8sc903n03	RKE2 agent #2	2cpu, 8GiB RAM

For all the steps, use your normal unprivileged user. When elevated rights are needed, the examples will clearly show the use of the prefix sudo. Do NOT, I repeat, do NOT run these commands under the `root` user.

1.2.1 Install RKE2 server node

Log on to the system that will take the role of your RKE2 server node. On this node execute the following steps:

- Step 1: Downloading the RKE2 binary and install it as a rke2-server binary:

```
▶ curl -sfL https://get.rke2.io | sudo sh -
```

Note:

Not specifying the INSTALL_RKE2_TYPE environment variable will always result in installing a 'rke2-server' node.

- Step 2: Enabling install and configuration of the RKE2 server node

```
▶ sudo systemctl enable rke2-server.service --now
```

Now we have to wait until the systemctl command returns. In the meantime we can take a look in the installation/configuration process of the rke2-server by consulting it's log using journalctl. This can be done by backgrounding the systemctl process (Press Ctrl-Z and type bg) or by accessing the system using a new terminal session.

The optional command to view the rke2-server installation progress is:

```
▶ sudo journalctl -u rke2-server -lf
```

Note:

This will run a `tail` on the rke2-server log. You can abort the log viewing/following by pressing Ctrl-C

Note:

In the logs you will find a lot of error conditions and timeouts. This is normal behaviour as the different components of RKE2/K8S need to come up, find eachother and stabilize. It's all about loosely coupled components, remember?

Once the `systemctl` command returns we can resume with arranging our access to the RKE2/K8S cluster.

- Step 3: Create the directory .kube in your homedir and copy RKE2's kubeconfig file into it. We must not forget to set the proper owner/group to it so we can access it.

```
▶ mkdir -p ~/.kube

▶ sudo cp /etc/rancher/rke2/rke2.yaml ~/.kube/config

▶ sudo chown ${USER}:${USER} ~/.kube/config
```

Note:

RKE2 has a different name for the kubeconfig file as `kubeadm` has. For RKE2 it's called `rke2.yaml` and located in `/etc/rancher/rke2/rke2.yaml`

Note:

If not already done so, you will need to install the `kubectl` so that we can access our RKE2 K8S cluster. The commands for installing it are:

- Step 4: Install `kubectl` binary to access our cluster if not already done so:

```
▶ curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/
↪linux/amd64/kubectl"

▶ sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

Now we can check if our RKE2 cluster or actually still only the rke2-server node is coming up healthy:

- Step 5: Check with `kubectl get nodes` if the rke-server node is healthy.

```
▶ kubectl get nodes
```

```
NAME          STATUS   ROLES                       AGE     VERSION
k8sc903n01    Ready    control-plane,etcd,master   3m20s   v1.28.2+rke2r1
```

- Step 6: If the rke2-server node is not ready yet we can watch or troubleshoot the install by taking a look at the K8S engine room, the `kube-system` namespace:

```
▶ kubectl get pods
```

```
NAME                                          READY   STATUS      RESTARTS   ␣
↪AGE
cloud-controller-manager-k8sc903n01           1/1     Running     0          ␣
↪2m12s
etcd-k8sc903n01                               1/1     Running     0          ␣
↪2m12s
helm-install-rke2-canal-2tsct                 0/1     Completed   0          ␣
↪2m24s
helm-install-rke2-coredns-hthzg               0/1     Completed   0          ␣
↪2m24s
kube-apiserver-k8sc903n01                     1/1     Running     0          ␣
↪2m14s
kube-controller-manager-k8sc903n01            1/1     Running     0          ␣
↪2m24s
kube-proxy-k8sc903n01                         1/1     Running     0          ␣
↪2m14s
kube-scheduler-k8sc903n01                     1/1     Running     0          ␣
↪2m24s
rke2-canal-47kgn                              2/2     Running     0          ␣
↪2m12s
rke2-coredns-rke2-coredns-67f86d96c-8j5b8     1/1     Running     0          ␣
↪2m24s
rke2-coredns-rke2-coredns-autoscaler-d97d9cd9f-b9dqf   1/1     Running     0          ␣
↪2m24s
rke2-ingress-nginx-controller-4tj8g           1/1     Running     0          ␣
↪2m12s
rke2-metrics-server-c6fb46b64-fvg29           1/1     Running     0          ␣
↪2m24s
rke2-snapshot-controller-59cc9cd8f4-9h65p     1/1     Running     0          ␣
↪2m24s
rke2-snapshot-validation-webhook-54c5989b65-l4dpt      1/1     Running     0          ␣
↪2m24s
```

Note:

In the `kube-system` namespace you should be able to see all PODs either in `Running` or `Completed` state. Any other state either states that your RKE2 cluster is not (yet) ready or has some serious issues. Do check with `journalctl -u rke2-server -lf` for clues on the root-cause of the issue(s)

Note:

By default on RKE2 server-nodes, the so called `noschedule` taint is not configured. This means that master nodes do allow to have customer-workload scheduled

1.2.2 Prepare for installing RKE2 agent node(s)

To prepare for the installing of any RKE2 agent nodes we need to retrieve the so called `node-token` from the rke2-server node. This token can be found in `/var/lib/rancher/rke2/server/node-token`

Please lookup the token using the following command and store in a safe place. You will need it here after.

```
▶ sudo cat /var/lib/rancher/rke2/server/node-token
```

This token needs to be mentioned in a `config.yaml` file for the rke2-agent installations on the rke2-agent nodes.

Kindly create a file called `rke2-join-agent-config.yaml` in your home directory and put the following info in it:

```
server: https://<rke2-server-node>:9345
token: <node-token>
```

- For `server` please add the hostname of your rke2-server node.
- For `token` please add the node-token retrieved in above step.

Your `rke2-join-agent-config.yaml` file should look a little like this:

```
server: https://k8sc903n01:9345
token:␣
↪K1066bf857b5cb1b9a40d111ace22fac1177a4bdc19e6424c2a678e0b4273fb8cf5::server:ff544d6ba9b39ac62a817199d4249e
```

Note:

This `config.yaml` file is the way we are going to enable and configure add-ons and customizations to our cluster later in these labs.

Now that we have create the `rke2-join-agent-config.yaml` file we need to copy it to each of the rke2-agent nodes that we would like to add to our cluster. In our case where we have:

- k8sc903n02
- k8sc903n03

as our rke2-nodes we are going to copy the `rke2-join-agent-config.yaml` to each of them:

```
▶ scp rke2-agent-config.yaml k8sc903n02:

▶ scp rke2-agent-config.yaml k8sc903n03:
```

1.2.3 Installing and adding RKE2 agent nodes to the K8S cluster

Please login on your the node that will become your first rke2-agent. In our case that will be the `k8sc903n02`.

We need to execute the following 4 steps to join the rke2-agent to the existing rke2-server:

- Step 1: We need to install the RKE2 agent binary.

```
▶ curl -sfL https://get.rke2.io | sudo INSTALL_RKE2_TYPE="agent" sh -
```

Note:

This is of course actually the same command we executed on the rke2-server node, but now we have set the `INSTALL_RKE2_TYPE="agent"` ENV var.

- Step 2: We need to copy the `rke2-join-agent-config.yaml` file we crafted on the rke2-server and copy it to the proper RKE2 directory, so the RKE2 agent install process can use the information it to join our rke2-agent to the already available rke2-server.

```
▶ sudo mkdir -p /etc/rancher/rke2

▶ sudo cp rke2-join-agent-config.yaml /etc/rancher/rke2/config.yaml
```

- Step 3: Now we need to enable and install the rke2-agent service.

```
▶ sudo systemctl enable rke2-agent --now
```

Now we have to wait once more until the systemctl command returns. In the meantime we can take a look in the installation/configuration process of the rke2-agent by consulting it's log using journalctl. This can be done by backgrounding the systemctl process (Press Ctrl-Z and type bg) or by accessing the system using a new terminal session.

The optional command to view the rke2-server installation progress is:

```
▶ sudo journalctl -u rke2-agent -lf
```

Note:

This will run a `tail` on the rke2-agent log. You can abort the log viewing/following by pressing Ctrl-C

Note:

In the logs you will find a lot of error conditions and timeouts. This is normal behaviour as the different components of RKE2/K8S need to come up, find eachother and stabilize. Once more, it's all about loosely coupled components, remember?

Once the `systemctl` command returns we can resume with checking if our rke2-agent has joined our K8S cluster.

- Step 4: Verifying if our rke2-agent node has succesfully joined the cluster. For this we go back to the rke2-server node, in the example case, k8sc903n01 and use `kubectl get nodes` to ascertain that the new rke2-node has been added to the K8S cluster.

```
▶ kubectl get nodes
```

```
NAME         STATUS   ROLES                       AGE     VERSION
k8sc903n01   Ready    control-plane,etcd,master   15m38s   v1.28.2+rke2r1
k8sc903n02   Ready    control-plane,etcd,master   3m14s    v1.28.2+rke2r1
```

You can repeat adding more rke2-agents by repeating the above 3 or 4 steps.

1.2.4 Validating our RKE2 K8S cluster

If all the nodes have been joined, we can validate our cluster. For this we are going to verify the cluster in a number of steps:

- Step 1: Verify if all the nodes are up-and-ready

```
▶ kubectl get nodes
```

For a 1x rke2-server, 2 rke2-agents cluster the output would look like this: .. code-block:: console

```
NAME STATUS ROLES AGE VERSION k8sc903n01 Ready control-plane,etcd,master 18m42s
v1.28.2+rke2r1 k8sc903n02 Ready control-plane,etcd,master 6m18s v1.28.2+rke2r1 k8sc903n03 Ready
control-plane,etcd,master 3m18s v1.28.2+rke2r1
```

The status should be ready. If not we need to check the node(s) with `kubectl describe node <node-name>`

- Step 2: Verify the healthiness of the K8S engine room:

```
▶ kubectl get pods -n kube-system -n node
```

On the master node the following `static` PODS should be in Running state and no 'recent' restarts should be mentioned:

1. cloud-controller-manager
2. etcd
3. kube-apiserver
4. controller-manager
5. kuber-scheduler

For each of the nodes (server and agents) you should see an instance of the following PODs, state should be `Running`:

1. rke2-canal
2. kube-proxy
3. rke2-ingress-nginx-controller

On a 2-node cluster you should see 2 instances of each. On a 3-node cluster you should see of course 3 instances of the above PODs. They are running managed by so called DaemonSets, making sure that every node will have each once instance of these specific PODs

More PODs will be running in the `kube-system` namespace. Check the `rke2-coredns-rke2-coredns` Pods in particular. Also these should have state `Running`. They are of great importance as they will provide the service discovery / hostname resolving in our cluster.

The PODs with names helm-install-* having states like 'Completed' are of no concern. It means these PODs that have been created by K8S Jobs have done their work succesfully. In this case bringing up the Container Network Interface (CNI) CANAL DaemonSet and the coredns Deployment.

Note:

In the `kube-system` namespace you should be able to see all PODs either in `Running` or `Completed` state. Any other state either states that your RKE2 cluster is not (yet) ready or has some serious issues. Do check with `journalctl -u rke2-server -lf` on the rke2-server node or `journalctl -u rke2-agent` on the rke2-agent nodes for clues on the root-cause of the issue(s)

2 Install RKE2 on ARM64 architectures

2.1 Introduction

In this lab we are going to work with installing RKE2 on non x86_64 / AMD64 architectures like aarch64 / ARM64

2.1.1 Requirements

To be able to execute this lab, you need at least 2 Ubuntu 22.04 (LTS) systems on ARM64. These systems can be for example Raspberry PI4s, Rock5b, OrangePI5 etc. If you would like to add more `rke2-agent` nodes you can always add more machines. The ARM systems must have been equipped with at least 4 GB Memory and enough storage to store some containers (approximately 8 GB free space should be ok). Mind the RKE2 does not support RISCV64 based systems yet.

Furthermore, you need an Internet connection to be able to access the RKE2 distribution and download some containers from quay.io and docker.io.

In this lab we assume that there is an unprivileged user, named 'student01' available on this system. This user should be able to execute privileged commands using the `sudo` utility. Of course your user may have a different name, important is the access to `sudo su - root` privileges.

If not already done, you can create on all nodes this user as root:

```
$> useradd -m -s /bin/bash student

$> passwd student01

$> echo "student ALL=(ALL) NOPASSWD: ALL" | tee /etc/sudoers.d/89-student
```

Login again as the user student, and proceed this lab.

Bring your installation up-to-date, using:

```
▶ sudo apt update

▶ sudo apt upgrade -y
```

If systemd or the kernel is updated during this process, you need to reboot your system. If you are not sure: reboot it anyway.

```
▶ sudo systemctl reboot
```

Install the following software requirements:

```
▶ sudo apt -y install vim info wget curl elinks man-db manpages \
      bash-completion psmisc jq ipvsadm yamllint conntrack \
      apt-transport-https pinfo
```

Execute the update of your system and the installation of the software on each node.

2.1.2 Network Configuration

It is assumed that you have internet connection and for this lab that the local system firewall (UFW on Ubuntu) is disabled.

Please ensure it is disabled:

```
▶ sudo systemctl disable ufw --now
```

No further tuning of the network stack like with `kubeadm` is needed as the RKE2 installer binary will take care of it.

2.1.3 Swap

A requirement for Kubernetes is to have swap disabled, otherwise the Kubelet service will not start on the masters and nodes. The idea of Kubernetes is to directly terminate overcommitting workloads as not to jeopardize the workload on the K8S cluster that nicely abides to the rules. Remember the principle of `Cattle vs. Pets` in K8S.

```
▶ sudo systemctl disable --now swap.target

▶ sudo systemctl mask swap.target

▶ sudo sed -i '/swap/d' /etc/fstab

▶ sudo swapoff -a
```

Again, repeat this on every node.

2.1.4 Raspberry Pi OS

Note:

You can skip this part if you are not running `Rapberry Pi OS`. So if you are setting RKE2 up on a Raspberry PI but running with Ubuntu 22.04 LTS or with another ARM64 based system running Ubuntu 22.04, you will not have to do these preparations.

If your ARM64 system is a Raspberry PI running Raspberry PI OS it currently lacks the configuration of cgroups-v2 in the kernel commandline. This will prevent RKE2 or any K8S version from installing on your system. In order to fix this you will need to edit your `/boot/cmdline.txt` and the kernel options `cgroup_enable=memory cgroup_memory=1` to it like this:

```
console=serial0,115200 console=tty1 root=PARTUUID=<redacted> rootfstype=ext4 fsck.repair=yes␣
↪rootwait cgroup_enable=memory cgroup_memory=1
```

After you have changed the file you will need to reboot your Raspberry PI:

```
▶ sudo reboot
```

As Raspberry PI OS does not by default install the `iptables` package, we will need to do that manually on these platforms:

```
▶ sudo apt install iptables -y
```

2.2 Install RKE2

We will now use a 2 step process to install K8S using the RKE2 distribution on your systems:

1. Install RKE2 server on the system what will fulfill the role of master node for our K8S cluster.
2. Join the other servers that will play the role of (worker-)nodes by installing and configuring rke2-server on these.

In the examples shown here under we will have 3 systems:

Nr	Hostname	Role	Config
1	rpb58-n01	RKE2 server	4cpu, 8GiB RAM
2	rock5b-n11	RKE2 agent #1	8cpu, 16GiB RAM
3	rock5b-n12	RKE2 agent #2	8cpu, 16GiB RAM

For all the steps, use your normal unprivileged user. When elevated rights are needed, the examples will clearly show the use of the prefix sudo. Do NOT, I repeat, do NOT run these commands under the `root` user.

2.2.1 Install RKE2 server node

For ARM64 based architectures we need to do this step a little bit different from X86_64/AMD64 based architectures. The reason for this is that the first release where RKE2 supports ARM64 platforms is: `v1.27.3+rke2r1`. And event that one is from experience still flaky. When we are using the default install instructions it will still try to happily install a v1.26.x version for us that simply isn't available for ARM64. We can use ENV variables however to request RKE2 to install a version that does have ARM64 support and runs stable. We will choose version `v1.27.8+rke2r1` for our setup.

Log on to the system that will take the role of your RKE2 server node. On this node execute the following steps:

- Step 1: Downloading the RKE2 binary version `v1.27.8+rke2r1`

```
▶ curl -sfL https://get.rke2.io | sudo INSTALL_RKE2_VERSION=v1.27.8+rke2r1 sh -
```

Note:

Not specifying the INSTALL_RKE2_TYPE environment variable will always result in installing a 'rke2-server' node.

Note:

If you want you can install later version from the 1.27 or the 1.28 release. Kindly look at:

- https://docs.rke2.io/release-notes/v1.27.X
- https://docs.rke2.io/release-notes/v1.28.X
- etc.

which versions are available to you.

- Step 2: Enabling install and configuration of the RKE2 server node

```
▶ sudo systemctl enable rke2-server.service --now
```

Now we have to wait until the systemctl command returns. In the meantime we can take a look in the installation/configuration process of the rke2-server by consulting it's log using journalctl. This can be done by backgrounding the systemctl process (Press Ctrl-Z and type bg) or by accessing the system using a new terminal session.

The optional command to view the rke2-server installation progress is:

```
▶ sudo journalctl -u rke2-server -lf
```

Note:

This will run a `tail` on the rke2-server log. You can abort the log viewing/following by pressing Ctrl-C

Note:

In the logs you will find a lot of error conditions and timeouts. This is normal behaviour as the different components of RKE2/K8S need to come up, find eachother and stabilize. It's all about loosely coupled components, remember?

Once the `systemctl` command returns we can resume with arranging our access to the RKE2/K8S cluster.

- Step 3: Create the directory .kube in your homedir and copy RKE2's kubeconfig file into it. We must not forget to set the proper owner/group to it so we can access it.

```
▶ mkdir -p ~/.kube

▶ sudo cp /etc/rancher/rke2/rke2.yaml ~/.kube/config

▶ sudo chown ${USER}:${USER} ~/.kube/config
```

Note:

RKE2 has a different name for the kubeconfig file as `kubeadm` has. For RKE2 it's called `rke2.yaml` and located in `/etc/rancher/rke2/rke2.yaml`

Note:

If not already done so, you will need to install the `kubectl` so that we can access our RKE2 K8S cluster. The commands for installing it are:

- Step 4: Install `kubectl` binary to access our cluster if not already done so:

```
▶ curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/
↪linux/amd64/kubectl"

▶ sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

Now we can check if our RKE2 cluster or actually still only the rke2-server node is coming up healthy:

- Step 5: Check with `kubectl get nodes` if the rke-server node is healthy.

```
▶ kubectl get nodes
```

```
NAME         STATUS   ROLES                       AGE     VERSION
rpb58-n01    Ready    control-plane,etcd,master   3m20s   v1.28.2+rke2r1
```

- Step 6: If the rke2-server node is not ready yet we can watch or troubleshoot the install by taking a look at the K8S engine room, the `kube-system` namespace:

```
▶ kubectl get pods
```

```
NAME                                                READY   STATUS      RESTARTS
↪AGE
cloud-controller-manager-rpb58-n01                  1/1     Running     0
↪2m12s
etcd-rpb58-n01                                      1/1     Running     0
↪2m12s
helm-install-rke2-canal-2tsct                       0/1     Completed   0
↪2m24s
helm-install-rke2-coredns-hthzg                     0/1     Completed   0
↪2m24s
kube-apiserver-rpb58-n01                            1/1     Running     0
↪2m14s
kube-controller-manager-rpb58-n01                   1/1     Running     0
↪2m24s
kube-proxy-rpb58-n01                                1/1     Running     0
↪2m14s
kube-scheduler-rpb58-n01                            1/1     Running     0
↪2m24s
rke2-canal-47kgn                                    2/2     Running     0
↪2m12s
rke2-coredns-rke2-coredns-67f86d96c-8j5b8           1/1     Running     0
↪2m24s
rke2-coredns-rke2-coredns-autoscaler-d97d9cd9f-b9dqf   1/1     Running     0
↪2m24s
rke2-ingress-nginx-controller-4tj8g                 1/1     Running     0
↪2m12s
rke2-metrics-server-c6fb46b64-fvg29                 1/1     Running     0
↪2m24s
rke2-snapshot-controller-59cc9cd8f4-9h65p           1/1     Running     0
↪2m24s
rke2-snapshot-validation-webhook-54c5989b65-l4dpt   1/1     Running     0
↪2m24s
```

Note:

In the `kube-system` namespace you should be able to see all PODs either in `Running` or `Completed` state. Any other state either states that your RKE2 cluster is not (yet) ready or has some serious issues. Do check with `journalctl -u rke2-server -lf` for clues on the root-cause of the issue(s)

Note:

By default on RKE2 server-nodes, the so called `noschedule` taint is not configured. This means that master nodes do allow to have customer-workload scheduled

2.2.2 Prepare for installing RKE2 agent node(s)

To prepare for the installing of any RKE2 agent nodes we need to retrieve the so called `node-token` from the rke2-server node. This token can be found in `/var/lib/rancher/rke2/server/node-token`

Please lookup the token using the following command and store in a safe place. You will need it here after.

```
▶ sudo cat /var/lib/rancher/rke2/server/node-token
```

This token needs to be mentioned in a `config.yaml` file for the rke2-agent installations on the rke2-agent nodes.

Kindly create a file called `rke2-join-agent-config.yaml` in your home directory and put the following info in it:

```
server: https://<rke2-server-node>:9345
token: <node-token>
```

- For `server` please add the hostname of your rke2-server node.
- For `token` please add the node-token retrieved in above step.

Your `rke2-join-agent-config.yaml` file should look a little like this:

```
server: https://rpb58-n01:9345
token:␣
↪K1066bf857b5cb1b9a40d111ace22fac1177a4bdc19e6424c2a678e0b4273fb8cf5::server:ff544d6ba9b39ac62a817199d4249e
```

Note:

This `config.yaml` file is the way we are going to enable and configure add-ons and customizations to our cluster later in these labs.

Now that we have create the `rke2-join-agent-config.yaml` file we need to copy it to each of the rke2-agent nodes that we would like to add to our cluster. In our case where we have:

- rock5b-n11
- rock5b-n12

as our rke2-nodes we are going to copy the `rke2-join-agent-config.yaml` to each of them:

```
▶ scp rke2-join-agent-config.yaml rock5b-n11:

▶ scp rke2-join-agent-config.yaml rock5b-n12:
```

2.2.3 Installing and adding RKE2 agent nodes to the K8S cluster

Please login on your the node that will become your first rke2-agent. In our case that will be the `rock5b-n11`.

We need to execute the following 4 steps to join the rke2-agent to the existing rke2-server:

- Step 1: We need to install the RKE2 agent binary.
  Please note this needs to be the same `INSTALL_RKE2_VERSION` as we used on the server.

```
▶ curl -sfL https://get.rke2.io | sudo INSTALL_RKE2_VERSION=v1.27.8+rke2r1 INSTALL_RKE2_TYPE=
↪"agent" sh -
```

Note:

This is of course actually the same command we executed on the rke2-server node, but now we have set the `INSTALL_RKE2_TYPE="agent"` ENV var.

- Step 2: We need to copy the `agent-config.yaml` file we crafted on the rke2-server and copy it to the proper RKE2 directory, so the RKE2 agent install process can use the information it to join our rke2-agent to the already available rke2-server.

```
▶ sudo mkdir -p /etc/rancher/rke2

▶ sudo cp rke2-join-agent-config.yaml /etc/rancher/rke2/config.yaml
```

- Step 3: Now we need to enable and install the rke2-agent service.

```
▶ sudo systemctl enable rke2-agent --now
```

Now we have to wait once more until the systemctl command returns. In the meantime we can take a look in the installation/configuration process of the rke2-agent by consulting it's log using journalctl. This can be done by backgrounding the systemctl process (Press Ctrl-Z and type bg) or by accessing the system using a new terminal session.

The optional command to view the rke2-server installation progress is:

```
▶ sudo journalctl -u rke2-agent -lf
```

Note:

This will run a `tail` on the rke2-agent log. You can abort the log viewing/following by pressing Ctrl-C

Note:

In the logs you will find a lot of error conditions and timeouts. This is normal behaviour as the different components of RKE2/K8S need to come up, find eachother and stabilize. Once more, it's all about loosely coupled components, remember?

Once the `systemctl` command returns we can resume with checking if our rke2-agent has joined our K8S cluster.

- Step 4: Verifying if our rke2-agent node has succesfully joined the cluster. For this we go back to the rke2-server node, in the example case, rpb58-n01 and use `kubectl get nodes` to ascertain that the new rke2-node has been added to the K8S cluster.

```
▶ kubectl get nodes
```

```
NAME          STATUS   ROLES                         AGE     VERSION
rpb58-n01    Ready    control-plane,etcd,master    15m38s   v1.28.2+rke2r1
rock5b-n11    Ready     control-plane,etcd,master    3m14s    v1.28.2+rke2r1
```

You can repeat adding more rke2-agents by repeating the above 3 or 4 steps.

2.2.4 Validating our RKE2 K8S cluster

If all the nodes have been joined, we can validate our cluster. For this we are going to verify the cluster in a number of steps:

- Step 1: Verify if all the nodes are up-and-ready

```
▶ kubectl get nodes
```

For a 1x rke2-server, 2 rke2-agents cluster the output would look like this: .. code-block:: console

> NAME STATUS ROLES AGE VERSION rpb58-n01 Ready control-plane,etcd,master 18m42s v1.28.2+rke2r1 rock5b-n11 Ready control-plane,etcd,master 6m18s v1.28.2+rke2r1 rock5b-n12 Ready control-plane,etcd,master 3m18s v1.28.2+rke2r1

The status should be ready. If not we need to check the node(s) with `kubectl describe node <node-name>`

- Step 2: Verify the healthiness of the K8S engine room:

```
▶ kubectl get pods -n kube-system -n node
```

On the master node the following `static` PODS should be in Running state and no 'recent' restarts should be mentioned:

1. cloud-controller-manager
2. etcd

3. kube-apiserver
4. controller-manager
5. kuber-scheduler

For each of the nodes (server and agents) you should see an instance of the following PODs, state should be `Running`:

1. rke2-canal
2. kube-proxy
3. rke2-ingress-nginx-controller

On a 2-node cluster you should see 2 instances of each. On a 3-node cluster you should see of course 3 instances of the above PODs. They are running managed by so called DaemonSets, making sure that every node will have each once instance of these specific PODs

More PODs will be running in the `kube-system` namespace. Check the `rke2-coredns-rke2-coredns` Pods in particular. Also these should have state `Running`. They are of great importance as they will provide the service discovery / hostname resolving in our cluster.

The PODs with names helm-install-* having states like 'Completed' are of no concern. It means these PODs that have been created by K8S Jobs have done their work succesfully. In this case bringing up the Container Network Interface (CNI) CANAL DaemonSet and the coredns Deployment.

Note:

In the `kube-system` namespace you should be able to see all PODs either in `Running` or `Completed` state. Any other state either states that your RKE2 cluster is not (yet) ready or has some serious issues. Do check with `journalctl -u rke2-server -lf` on the rke2-server node or `journalctl -u rke2-agent` on the rke2-agent nodes for clues on the root-cause of the issue(s)

3 Installing and configuring add-ons for RKE2

3.1 Introduction

In this lab we are going to work with installing RKE2 with supplied add-ons and we will learn how to configure these.

We will practice using/configuring the boxed-in add-ons:

- RKE2 supplied metrics server
- RKE2 supplied Ingress-nginx controller

And we will practice installing and configuring our own add-ons:

- OpenEBS storage provisioner
- Cilium Container Network Interface

3.2 Requirements

A working RKE2 cluster on AMD64 or ARM64.

3.3 Boxed in add-ons

Out of the box RKE2 supplies you with:

1. Canal CNI for K8S Networking
2. CoreDNS for K8S DNS services and service discovery
3. K8S Metrics Server for insights in resource usage using `kubectl top` etc.
4. ETCD snapshot controller for backing up and restoring the `ETCD` cluster database
5. Ingress controller of the type Ingress-nginx
6. Helm controller to install additional components using `Helm charts`

In this part we will focus on working with the `K8S Metrics Server` and the `Ingress-nginx controller`

3.3.1 K8S Metrics Server

We don't need to configure anything for the `Metrics Server` and can use it out of the box:

```
▶ kubectl top nodes
```

```
NAME         CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
rock5b-n11   360m         4%     3461Mi          22%
rock5b-n12   305m         3%     1203Mi          7%
rpb58-n01    274m         6%     2215Mi          27%
```

This will show use the resource usage of each `node` in our cluster.

With `kubectl top pods` we can see the resource usage of the PODs in a certain namespace:

```
▶ kubectl top pods -n default
```

```
NAME                              CPU(cores)   MEMORY(bytes)
tst-containers-68c464d94f-b65nm   7m           8Mi
```

Note:

The `K8S Metrics Server` will also provide metrics for `K8S Horizontal Pod Autoscaling (HPA)`

3.3.2 Ingress-nginx controller

Architecture

Let's look at how the `ingress-nginx` controller is deployed with RKE2:

```
▶ kubectl get pods -n kube-system -l app.kubernetes.io/name=rke2-ingress-nginx
```

This will display all PODs belonging to the `ingress-nginx-controller`.

```
NAME                                 READY   STATUS    RESTARTS   AGE
rke2-ingress-nginx-controller-2p4vv   1/1     Running   0          8h
rke2-ingress-nginx-controller-6nllx   1/1     Running   0          8h
rke2-ingress-nginx-controller-qfj9h   1/1     Running   0          86m
```

Note that it's deployed as a daemonset. The single generated name part to the POD name will hint at it and a `kubectl get ds rke2-ingress-nginx-controller -n kube-system` will prove it.

As these are PODs in a DaemonSet we do not get access to them using a service. PODs in a DaemonSet are normally accessed using the NODE IP addresses and can use so called `low ports`.

To access the `ingress-nginx-controller` we can use a command like:

```
▶ curl -k https://rock5b-n12
```

You will receive output like:

```
*   Trying 10.8.62.238:443...
* Connected to rock5b-n11 (10.8.62.238) port 443 (#0)
* ALPN: offers h2,http/1.1
* TLSv1.3 (OUT), TLS handshake, Client hello (1):
* TLSv1.3 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (IN), TLS handshake, Server key exchange (12):
* TLSv1.2 (IN), TLS handshake, Server finished (14):
* TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
* TLSv1.2 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.2 (OUT), TLS handshake, Finished (20):
* TLSv1.2 (IN), TLS handshake, Finished (20):
* SSL connection using TLSv1.2 / ECDHE-RSA-AES128-GCM-SHA256
* ALPN: server accepted h2
* Server certificate:
*  subject: O=Acme Co; CN=Kubernetes Ingress Controller Fake Certificate
*  start date: Dec  8 23:23:28 2023 GMT
*  expire date: Dec  7 23:23:28 2024 GMT
*  issuer: O=Acme Co; CN=Kubernetes Ingress Controller Fake Certificate
*  SSL certificate verify result: self-signed certificate (18), continuing anyway.
* using HTTP/2
* h2h3 [:method: GET]
* h2h3 [:path: /]
* h2h3 [:scheme: https]
* h2h3 [:authority: rock5b-n11]
* h2h3 [user-agent: curl/7.88.1]
* h2h3 [accept: */*]
* Using Stream ID: 1 (easy handle 0x5556038fca90)
> GET / HTTP/2
> Host: rock5b-n11
> user-agent: curl/7.88.1
> accept: */*
>
< HTTP/2 404
< date: Sat, 09 Dec 2023 08:24:46 GMT
< content-type: text/html
< content-length: 146
< strict-transport-security: max-age=15724800; includeSubDomains
<
<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx</center>
</body>
</html>
* Connection #0 to host rock5b-n11 left intact
```

This is definitely coming from the `ingress-nginx-controller` as you can see from the `server certificate` part of the output.

When you are connecting to the `Ingress Controller` on port 80 or 443 on any of your nodes this will connect you to the appropriate service as directed by the `Ingress` API resource. Mind the difference between the `Controller` (aka the software) and the `Ingress` aka the configuration telling the `Controller` what to serve when someone is connecting to it. You can place a LB that using a Virtual IP (VIP) address that will connect you to any node that has the `Ingress Controller` listening on port 80 and/or 443.

Using the Ingress controller

Let's deploy 2 microservices, a blue and a green one that will show some ASCII art in color.

Deploying the demo-blue microservice

1. Creating the demo-blue deployment

Listing 1: deploy-demo-blue.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: demo-blue
  name: demo-blue
spec:
  replicas: 3
  selector:
    matchLabels:
      app: demo-blue
  strategy: {}
  template:
    metadata:
      labels:
        app: demo-blue
    spec:
      containers:
      - image: quay.io/pamvdam/containers:1.4
        name: containers
        env:
        - name: COLOR
          value: blue
        ports:
        - containerPort: 8080
```

2. Creating the demo-blue service

   Create a file called `svc-blue.yaml` with the following content:

Listing 2: svc-blue.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app: demo-blue
  name: demo-blue
spec:
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    app: demo-blue
```

Deploying the demo-green microservice

1. Creating the demo-green deployment

Listing 3: deploy-demo-blue.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: demo-green
  name: demo-green
spec:
  replicas: 3
  selector:
    matchLabels:
      app: demo-green
  strategy: {}
  template:
    metadata:
      labels:
        app: demo-green
    spec:
      containers:
      - image: quay.io/pamvdam/containers:1.4
        name: containers
        env:
        - name: COLOR
          value: green
        ports:
        - containerPort: 8080
```

2. Creating the demo-green service
   Create a file called `svc-green.yaml` with the following content:

Listing 4: svc-blue.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app: demo-green
  name: demo-green
spec:
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    app: demo-green
```

Configuring ingress for both microservices

1. Create a file called `demo-ingress.yaml` with the following content:

Listing 5: deploy-ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  creationTimestamp: null
  name: containers
spec:
  rules:
  - host: blue.containers.k8s
    http:
      paths:
      - backend:
          service:
            name: demo-blue
            port:
              number: 8080
        path: /
        pathType: Exact
  - host: green.containers.k8s
    http:
      paths:
      - backend:
          service:
            name: demo-green
            port:
              number: 8080
        path: /
        pathType: Exact
```

1. Apply the `ingress` (configuration):

```
kubectl apply -f demo-ingress.yaml
```

Configuring ingress for both microservices

1. Testing the access through the ingress for the blue service

```
curl -H 'Host: blue.containers.k8s' rpb58-n01
```

Listing 6: curl-blue.out

```
                    (((((((((
                .(((((((((((((((((.
            .((((((((((((&((((((((((((.
         /((((((((((((((((@(((((((((((((((/
       ((((((((((((((((((@((((((((((((((((((
      *(((((##((((((@@@@@@@@@@@@(((((((%#(((((*
      (((((((@@@(@@@@#((@@@@((#@@@@(@@@(((((((
     *(((((((((@@@@(((((@@@@(((((@@@@((((((((,
      (((((((((@@@%@@@@((@@@@((@@@@%@@@(((((((((
    .(((((((((@@((((@@@@@@@@@@@@((((@@(((((((((.
    (((((((((&@@(((((@@@(((@@@(((((@@&(((((((((
    (((((((((&@@@@@@@@@@@@#(#@@@@@@@@@@@@&(((((((((
   ((((((@@@@@@@@@@(((((@@@@@@@@(((((&@@@@@@@@((((((
   ((((((((((((%@@((((%@@@(@@@@%((((@@&((((((((((((
    ((((((((((((@@@@((@@%(((%@@((@@@((((((((((((
      ((((((((((#@@@@%(((((&@@@@#((((((((((
       /((((((((((@@@@@@@@@@@@@@@@((((((((((/
         (((((((((@@((((((((((@@(((((((((
           (((((((&(((((((((((((&(((((((
            /(((((((((((((((((((((((((/
              ((((((((((((((((((((((
This container is running in KUBERNETES on demo-blue-68f6f4f4b6-gnnwx (10.42.0.8)
```

2. Testing the access through the ingress for the green service

```
curl -H 'Host: green.containers.k8s' rpb58-n01
```

Listing 7: curl-green.out

```
                    (((((((((
                .(((((((((((((((((.
            .((((((((((((&((((((((((((.
         /((((((((((((((((@(((((((((((((((/
       ((((((((((((((((((@((((((((((((((((((
      *(((((##((((((@@@@@@@@@@@@(((((((%#(((((*
      (((((((@@@(@@@@#((@@@@((#@@@@(@@@(((((((
     *(((((((((@@@@(((((@@@@(((((@@@@((((((((,
      (((((((((@@@%@@@@((@@@@((@@@@%@@@(((((((((
    .(((((((((@@((((@@@@@@@@@@@@((((@@(((((((((.
    (((((((((&@@(((((@@@(((@@@(((((@@&(((((((((
    (((((((((&@@@@@@@@@@@@#(#@@@@@@@@@@@@&(((((((((
   ((((((@@@@@@@@@@(((((@@@@@@@@(((((&@@@@@@@@((((((
   ((((((((((((%@@((((%@@@(@@@@%((((@@&((((((((((((
    ((((((((((((@@@@((@@%(((%@@((@@@((((((((((((
      ((((((((((#@@@@%(((((&@@@@#((((((((((
       /((((((((((@@@@@@@@@@@@@@@@((((((((((/
         (((((((((@@((((((((((@@(((((((((
           (((((((&(((((((((((((&(((((((
            /(((((((((((((((((((((((((/
              ((((((((((((((((((((((
This container is running in KUBERNETES on demo-green-7c8dfff7db-t6hbk (10.42.2.9)
```

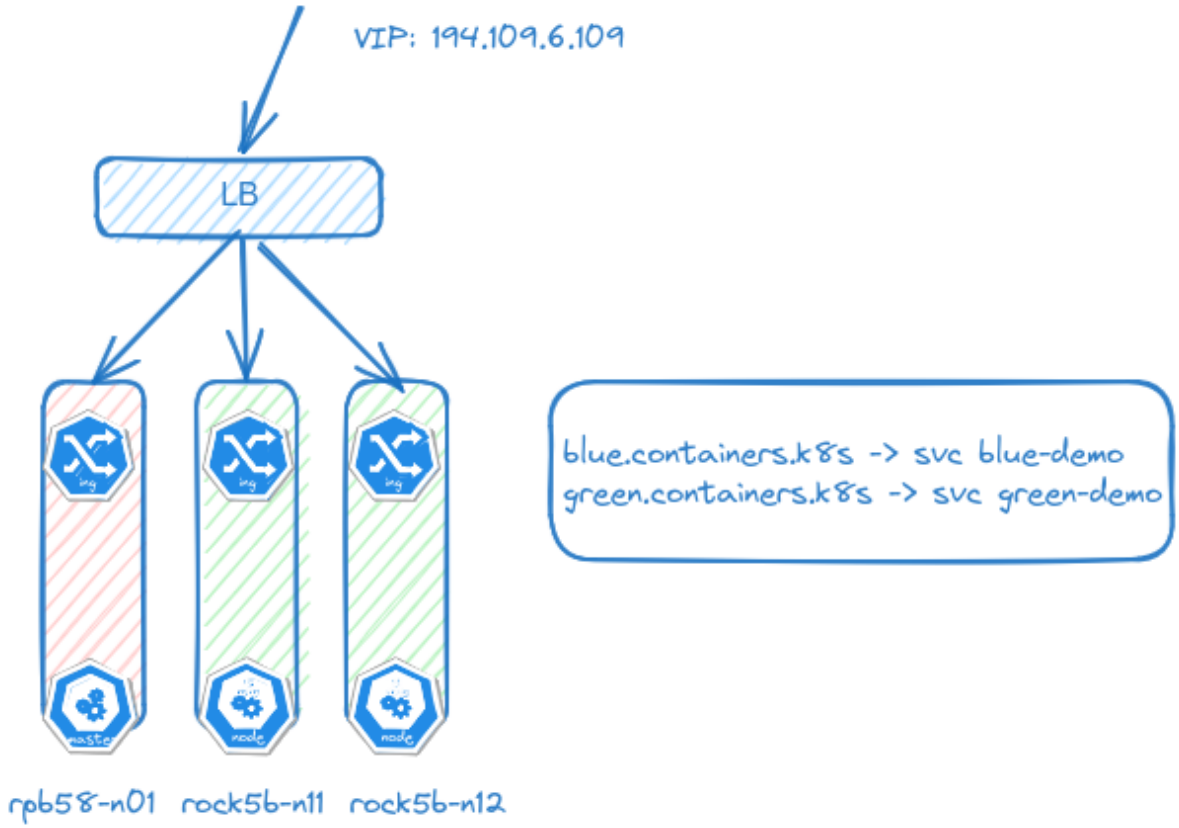
3. Testing the access through the ingress for an unconfigured URL/service

```
curl -H 'Host: black.containers.k8s' rpb58-n01
```

Listing 8: curl-black.out

```
<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

In a picture:



3.3.3 Custom Add-ons

Introduction

Custom add-ons in RKE2 are added using the Helm chart controller that is running in the kube-system namespace of RKE2. The Helm controller is deployed by default on an RKE2 installation.

This Helm chart controller can be provided with YAML files during bootstrap that have been stored in the directory /var/lib/rancher/rke2/server/manifests.

Post-setup, so when your RKE2/K8S cluster is already fully running, you can use the Helm chart controller to deploy applications/tools using Helm charts.

The YAML needed to deploy a Helm chart using the Helm chart controller looks like this:

Listing 9: helmchart-openebs.yaml

```
apiVersion: helm.cattle.io/v1
kind: HelmChart
metadata:
  name: openebs
  namespace: kube-system
spec:
  chart: openebs
  repo: https://openebs.github.io/charts
  targetNamespace: openebs
```

Table 1: OpenEBS HelmChart YAML Explanation

Field	Value	Explanation
apiVersion	helm.cattle.io/v1	Specifies the API version of the Helm chart CRD.
kind	HelmChart	Indicates the kind of Kubernetes resource; a Helm chart managed by RKE2.
metadata.name	openebs	The name of the HelmChart resource; a unique identifier within the namespace.
metadata.namespace	kube-system	The namespace where the HelmChart resource will be created; typically 'kube-system' for RKE2.
spec.chart	openebs	The name of the Helm chart to be deployed.
spec.repo	https://openebs.github.io/charts	The repository URL where the Helm chart is hosted.
spec.targetNamespace	openebs	The target namespace where the Helm chart will be deployed; must be created beforehand if it doesn't exist.
spec.valuesContent	-	The contents of the values file

This is a simple example without the Helm charts values included. It will roll out the OpenEBS Helm chart using the default values that come with it.

If you already have an RKE2 cluster running you can deploy this Helm-chart by creating the above file (helm-chart-openebsyaml) and submitting it to your cluster with the following command:

```
▶ kubectl create ns openebs

▶ kubectl apply -f helmchart-openebs.yaml
```

Once you have done this, the helm-chart-controller will wake up and process the YAML. The helm chart will be pulled from it's specified repo and deployed. Please note that the helm-chart-controller does not create the namespaces for you. You will need to that upfront yourself, like in the example.

You can check which helmcharts have been deployed using the helm-chart-controller by issuing the following command:

```
▶ kubectl get helmcharts -A
```

You can remove helmchart deploy done by the helm-chart-controller by deleting the helmchart yaml. E.g. if you would like to delete above OpenEBS chart you will need to issue a command like this:

```
▶ kubectl delete helmchart openebs -n kube-system
```

Remember that by default the helmcharts themselves get deployed in namespace kube-system where the helm-chart-controller is running. The applications described by the helmcharts can be deployed in any namespace specified as long as it's already created.

In the case you want to update an already deployed helmchart for example because you would like to reconfigure it, you can update the helmchart YAML and re-apply it. E.g:

Listing 10: helmchart-openebs-values.yaml

```
apiVersion: helm.cattle.io/v1
kind: HelmChart
metadata:
  name: openebs
  namespace: kube-system
spec:
  chart: openebs
  repo: https://openebs.github.io/charts
  targetNamespace: openebs
  valuesContent: |-
    # Enable LocalPV
    localpv:
      enabled: true
      hostpath:
        basePath: "/var/openebs/local"
      device:
        basePath: "/dev"

    # Enable NDM (Node Disk Manager)
    ndm:
      enabled: true

    # Enable Node Disk Operator
    ndmOperator:
      enabled: true

    # Additional feature configurations
    featureGates:
      CSI: "true"
      AdmissionWebhook: "true"
```

```
▶ kubectl apply -f helmchart-openebs.yaml
```

The helm-chart will be updated and when needed a redeployment will be triggerd, like when applied with the CLI helm tool.

We have now described using the helm-controller when the cluster is already up-and-running. You can also use helm-charts and the helm-chart-controller when installing your RKE2 cluster so it will directly have all the add-ons you want after install. In order to do this you will need to add the helmchart yaml manifests on the node that will be your first RKE2 server node. All YAML files in the directory /var/lib/rancher/rke2/server/manifests will be processed by the RKE2 server. Please note that this declarative use of K8S so we never know upfront if the YAML will lead to the correct deployment of the desired add-on. So we have to make sure that all requirements have been arranged for.

In the case of helmcharts that the helm-chart-controller will pick up once it comes alive on the RKE2 cluster, we must make sure that the namespace where our add-on will be deployed by the helmchart is pre-created.

Best practice is to prepend a piece of YAML to the helmchart file that will create the namespace upfront. In the case of OpenEBS that gets deployed in the openebs namespace we can use a YAML manifest like this:

Listing 11: helmchart-openebs-values-ns.yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: openebs
---
apiVersion: helm.cattle.io/v1
kind: HelmChart
metadata:
  name: openebs
  namespace: kube-system
spec:
  chart: openebs
  repo: https://openebs.github.io/charts
  targetNamespace: openebs
```

If you place this file in the directory `/var/lib/rancher/rke2/server/manifests` it will be processed by the RKE2 server setup process. Please note this is after installing the RKE2 server binary and prior to starting the `rke2-server` service with `systemctl enable rke2-server --now`. As such you must pre-create the directory structure `/var/lib/rancher/rke2/server/manifests` in order to be able to place the files in it.

OpenEBS

What is OpenEBS actually?

OpenEBS is an open-source storage platform that provides containerized block storage for cloud-native and other environments. It's designed to be an easy-to-use, scalable, and agile storage solution, especially well-suited for Kubernetes environments. OpenEBS delivers us:

- Containerized Storage Volumes
  OpenEBS enables the creation of highly available and scalable storage volumes that can be dynamically provisioned and managed, much like other cloud-native resources.
- Kubernetes Integration
  It is deeply integrated with Kubernetes, making it a natural fit for Kubernetes-based applications. It uses the Container Storage Interface (CSI) to seamlessly integrate with the Kubernetes ecosystem.
- Flexibility and Choice of Storage Engines
  OpenEBS offers a variety of storage engines, such as Jiva, cStor, and LocalPV, to cater to different use cases like high performance, resilience, or simplicity. Users can choose the most appropriate engine for their specific requirements.
- Replication and High Availability
  It supports replication of data across multiple nodes, ensuring high availability and resilience of data.
- Snapshotting and Cloning
  OpenEBS allows for taking snapshots of data volumes and cloning them, which can be useful for backup/restore operations and test/dev environments.
- Use Cases
  It's widely used for use cases like database storage, logging, and monitoring systems in Kubernetes, providing a persistent and reliable storage layer.

To install OpenEBS at bootstrap time of your RKE2 system kindly execute the following steps:

1. Install RKE2 server using the `curl | sh` command described in one of the previous 2 modules.
2. Prior to starting up RKE2 with `sudo systemctl enable rke2-server` now executed the following commands:

```
▶ sudo mkdir -p /var/lib/rancher/rke2/server/manifests
```

1. Create the following file `helm-chart` file `helmchart-openebs.yaml`, if needed adapt it:

Listing 12: helmchart-openebs-values-ns.yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: openebs
---
apiVersion: helm.cattle.io/v1
kind: HelmChart
metadata:
  name: openebs
  namespace: kube-system
spec:
  chart: openebs
  repo: https://openebs.github.io/charts
  targetNamespace: openebs
```

1. Copy the file to the directory /var/lib/rancher/rke2/server/manifests

```
▶ sudo cp helmchart-openebs.yaml /var/lib/rancher/rke2/server/manifests
```

1. Now enable and start the `rke2-server` systemd service:

```
▶ sudo systemctl enable rke2-server --now
```

This only needs to be done once on the first rke2-server. You should not repeat it on any other `rke2-server` or ``rke2-agent` you are going to add. The helmchart will propagate the API resources over the rest of your cluster when needed.

If you would like to install `openebs` using the `helm-chart-controller` when you already have an up-and-running cluster you can create the `helmchart-openebs.yaml` file, create the `openebs` namespace in RKE2 and submitting this `YAML` file to the RKE2 cluster.

Listing 13: helmchart-openebs.yaml

```
apiVersion: helm.cattle.io/v1
kind: HelmChart
metadata:
  name: openebs
  namespace: kube-system
spec:
  chart: openebs
  repo: https://openebs.github.io/charts
  targetNamespace: openebs
```

```
▶ kubectl create ns openebs

▶ kubectl apply -f helmchart-openebs.yaml
```

You can check if the OpenEBS installation went well by checking the `openebs` namespace:

```
▶ kubectl get pods -n openebs
```

You should see an output like this:

Listing 14: openebs-1.out

```
NAME                                          READY   STATUS    RESTARTS   AGE
openebs-localpv-provisioner-6b4f46dd8c-jckv4  1/1     Running   0          116m
openebs-ndm-4q5xw                             1/1     Running   0          116m
openebs-ndm-cf5v2                             1/1     Running   0          116m
openebs-ndm-operator-64fc5fc9c-kwvj5          1/1     Running   0          116m
openebs-ndm-pflxj                             1/1     Running   0          116m
```

Of course the proof of the pudding is in the eating; so let's try to get a Persistent Volume (PV) provisioned:

1. Create a YAML file called `testpvc.yaml` like this:

Listing 15: test-pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: test-pvc
  namespace: default
spec:
  storageClassName: openebs-hostpath
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Mi
```

1. Submit the YAML file to the RKE2 server:

```
▶ kubectl apply -f test-pvc.yaml
```

1. After a little time, verify if a PV was created:

```
▶ kubectl get pvc
```

As you can see the PVC is still pending for a PV to be provisioned. This is by design as OpenEBS will wait for the first consumer of the PVC before it will actually start provisioning the PV. So, let's add the demand side.

Kindly create another YAML file describing a POD consuming the `test-pvc` PV.

Listing 16: test-pvc-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: my-openebs-pod
  namespace: default
  label:
    run: my-openebs-pod
spec:
  initContainers:
    - name: init-web-content
      image: busybox
      command: ['sh', '-c', 'echo "<H1>Welcome friends of RKE2!</H1>" > /web-root/index.html
↪']
      volumeMounts:
        - name: storage
          mountPath: /web-root
  containers:
    - name: my-container
      image: nginx
      volumeMounts:
        - name: storage
          mountPath: "/usr/share/nginx/html"
  volumes:
    - name: storage
      persistentVolumeClaim:
        claimName: test-pvc
```

And submit it:

```
▶ kubectl apply -f test-pvc-pod.yaml
```

After a little while check again:

```
▶ kubectl get pvc

▶ kubectl get pv

▶ kubectl get pods -l run=my-openebs-pod -o wide
```

The PVC should be bound, the PV should be created and the POD should be online.

You can get the POD IP using `kubectl get pods -l run=my-openebs-pod -o wide` and try to `curl` it:

```
▶ kubectl get pods -l run=my-openebs-pod -o wide

▶ curl <pod-ip>
```

You should see someting like:

```
<H1>Welcome friends of RKE2!</H1>
```

Cilium CNI

By default RKE2 will come outfitted with the CANAL CNI out of the box. CANAL is actually Calico piggybacked over Flannel. We have the ease of networking of Flannel (runs everywhere) and still we can wield the nice security features like Network Policies that come with Calico CNI.

But sometimes, you would like to have another CNI installed like pure Calico (without the Flannel part) or the very powerful and modern Cilium CNI with all the good stuff that it brings, like built-in Ingres, Clustermesh, Service Mesh, Load Balacning and Observation tools.

In this part we are going to practice with installing Cilium instead of CANAL at setup time. Changing the CNI on an already running RKE2 cluster is a real challenge if at all possible, so we are not going to chase that here.

In order to have a pristine new RKE2 cluster installed with RKE2 instead of the default CANAL we need to execute some steps:

1. Specify in an RKE2 server config file that we are going to use Cilium instead of CANAL
2. Optionally have Cilium take over `kube-proxy` functionality so we need to disable it in RKE2 K8S
3. Using the `helm-chart-controller` supply helm-chart for Cilium and any additional YAML config files for Cilium

Important; as written earlier, this can only be done easily at the setup of a new RKE2 cluster, so that will also be our starting point.

Let's get started:

1. Install the RKE2 server binary with `curl` like described in the previous examples.
   Do NOT start the `rke2-server`. We will first need to deploy some files in the RKE2 directories.
2. Create a RKE2 config file to skip CANAL and select Cilium for our CNI
   Kindly create the following yaml file called `rke2-config-cilium.yaml` with the following contents:

   Listing 17: code/cilium/rke2-server-cilium-config.yaml

   ```
   cni: cilium
   disable-kube-proxy: true
   ```

   In this example we are also disabling `kube-proxy` in RKE2 so that Cilium CNI can take over that funcionality in which it can do much more efficient.
3. Create the directory `/etc/rancher/rke2` and copy the `rke2-server-cilium-config.yaml` file into it.
   ..code:: bash
   - sudo mkdir -p /etc/rancher/rke2
   - sudo cp rke2-server-cilium-config.yaml /etc/rancher/rke2
4. Create a `helmchart` yaml file for the `helm-chart-controller` to deploy Cilium. Call it `rke2-cilium-config.yaml`. An example is included here:

Listing 18: code/cilium/rke2-server-cilium-config.yaml

```
# /var/lib/rancher/rke2/server/manifests/rke2-cilium-config.yaml
---
apiVersion: helm.cattle.io/v1
kind: HelmChartConfig
metadata:
  name: rke2-cilium
  namespace: kube-system
spec:
  version: 1.14.4
  valuesContent: |-
    kubeProxyReplacement: strict
    k8sServiceHost: k8sc903n01
    k8sServicePort: 6443
    cni:
      chainingMode: "none"
    hubble:
      enabled: true
      relay:
        enabled: true
      ui:
        enabled: true
    ingressController:
      enabled: true
      loadBalancerMode: "shared"
    clustermesh:
      config:
        enabled: true
    l2announcements:
      enabled: true
      leaseDuration: "3s"
      leaseRenewDeadline: "1s"
      leaseRetryPeriod: "500ms"
    externalIPs:
      enabled: true
    devices:
    - ens2
```

5. Copy the `rke2-cilium-config.yaml` to the `/var/lib/rancher/rke2/server/manifests` directory.

```
► sudo mkdir -p /var/lib/rancher/rke2/server/manifests

► sudo cp rke2-cilium-config.yaml /var/lib/rancher/rke2/server/manifests
```

6. Create and copy any extra YAML files fo cilium under `/var/lib/rancher/rke2/server/manifests`

   Tips:

   - Please do check the cilium config files as these need to be tailored to your environment. Especially the naming of the network devices. This goes beyond the scope of this workbook.
   - Setting up a cluster on ARM64 with Cilium can take a little bit longer then you are used too with CANAL.

4 Backing up and restoring RKE2 clusters

4.1 Introduction

In this lab we are going to learn how we can:

1. Configure scheduled snapshots of our RKE2/ECTD store.
2. Make ad-hoc snapshots of our RKE2/ETCD store.
3. Restore an ETCD snapshot to recover our RKE2 cluster.

4.1.1 Requirements

- An up-and-running RKE2 cluster, it doens't matter if it's having a single server control plane or a multi-server HA controlplane. We will cover backing up en restoring both.

4.1.2 How it works

The `rke2-server` takes care of ad hoc and periodic snapshotting of the cluster's etcd database. The snapshot files are placed by default in the directory `/var/lib/rancher/rke2/server/db/snapshots`. RKE2 will keep a default retention of 5 snapshots. When more then the configured `etcd-snapshot-retention` snapshots have been created, the oldest one will get pruned. If needed you can also prune snapshots yourself.

4.1.3 Setting up etcd snapshotting

By default ectd snapshotting is enabled when you setup your RKE2 server with the defaults.

You can configure the etcd snapshotting mechanism by creating or adapting your `/etc/rancher/rke2/config.yaml` file.

E.g:

Listing 1: config.yaml

```
etcd-snapshot-schedule-cron: "10 */4 * * *"
etcd-snapshot-retention: 10
```

The following options can be used:

Table 1: RKE2 etcd Snapshotting Configuration Options

Field	Default	Explanation
etcd-snapshot-schedule-cron	"*/5 * * * *"	Cron schedule to take automated etcd snapshots. For example, "*/5 * * * *" means a snapshot is taken every 5 minutes.
etcd-snapshot-retention	5	The number of snapshots to retain. For example, '5' means the five most recent snapshots are kept.
etcd-snapshot-dir	/var/lib/rancher/rke2/server/db/snapshots	The directory where etcd snapshots are stored. Can be set to a custom path.
etcd-snapshot-compress	false	Enable or disable compressed snapshots. Set to 'true' to enable.
etcd-s3	false	Enable or disable backup to S3. Set to 'true' to enable.
etcd-s3-bucket	""	The name of the S3 bucket where snapshots are stored, if S3 backup is enabled.
etcd-s3-region	""	The region of the S3 bucket, if S3 backup is enabled.
etcd-s3-endpoint	""	The endpoint URL for S3, if S3 backup is enabled.
etcd-s3-access-key	""	The access key for S3, if S3 backup is enabled.
etcd-s3-secret-key	""	The secret key for S3, if S3 backup is enabled.
etcd-s3-folder	""	The folder within the S3 bucket to store snapshots, if S3 backup is enabled.
etcd-s3-skip-ssl-verify	false	Skip SSL certificate verification for S3, if S3 backup is enabled. Useful in testing environments.

4.1.4 Making an ad hoc etcd snapshot

To make an ad hoc ectd snapshot, issue the following command:

```
▶ sudo rke2 etcd-snapshot now
```

4.1.5 Listing available etcd snapshots

To list the available etcd snapshots on the RKE2 server, use the following command:

```
▶ sudo rke2 etcd-snapshot ls
```

In the output you will recognize ad hoc and scheduled backups. Your output will look something like this:

Listing 2: Output of `rke2 etcd-snapshot ls`

```
Name                                Location
↪                                 Size     Created
on-demand-rpb58-n01-1702240686         file:///var/lib/rancher/rke2/server/db/snapshots/on-
↪demand-rpb58-n01-1702240686         10305568 2023-12-10T20:38:06Z
on-demand-rpb58-n01-1702240858         file:///var/lib/rancher/rke2/server/db/snapshots/on-
↪demand-rpb58-n01-1702240858         10305568 2023-12-10T20:40:58Z
on-demand-rpb58-n01-1702240917         file:///var/lib/rancher/rke2/server/db/snapshots/on-
↪demand-rpb58-n01-1702240917         10305568 2023-12-10T20:41:57Z
etcd-snapshot-rpb58-n01-1702243200.zip file:///var/lib/rancher/rke2/server/db/snapshots/etcd-
↪snapshot-rpb58-n01-1702243200.zip 1276185  2023-12-10T21:20:00Z
etcd-snapshot-rpb58-n01-1702243804     file:///var/lib/rancher/rke2/server/db/snapshots/etcd-
```

(continues on next page)

(continued from previous page)

```
↪snapshot-rpb58-n01-1702243804     6012960  2023-12-10T21:30:04Z
etcd-snapshot-rpb58-n01-1702244403     file:///var/lib/rancher/rke2/server/db/snapshots/etcd-
↪snapshot-rpb58-n01-1702244403     6340640  2023-12-10T21:40:03Z
on-demand-rpb58-n01-1702244969         file:///var/lib/rancher/rke2/server/db/snapshots/on-
↪demand-rpb58-n01-1702244969         6340640  2023-12-10T21:49:29Z
etcd-snapshot-rpb58-n01-1702245002     file:///var/lib/rancher/rke2/server/db/snapshots/etcd-
↪snapshot-rpb58-n01-1702245002     6340640  2023-12-10T21:50:02Z
```

4.1.6 Restoring a etcd snapshot

We will discuss 2 scenario's:

1. Restoring of a single RKE2 server (single controlplane)
2. Restoring of multiple HA RKE2 servers (HA controlplane)

Restoring of a single RKE2

- Step 1: Bring down the RKE2 server

```
▶ sudo systemctl stop rke2-server
```

- Step 2: Select an etcd snapshot to restoreetcd-snapshot-rpb58-n01-1702245002

```
▶ sudo rke2 etcd-snapshot ls
```

Listing 3: Output of `rke2 etcd-snapshot ls`

```
Name                                  Location                                                ␣
↪                                       Size     Created
on-demand-rpb58-n01-1702240686        file:///var/lib/rancher/rke2/server/db/snapshots/
↪on-demand-rpb58-n01-1702240686         10305568 2023-12-10T20:38:06Z
on-demand-rpb58-n01-1702240858        file:///var/lib/rancher/rke2/server/db/snapshots/
↪on-demand-rpb58-n01-1702240858         10305568 2023-12-10T20:40:58Z
on-demand-rpb58-n01-1702240917        file:///var/lib/rancher/rke2/server/db/snapshots/
↪on-demand-rpb58-n01-1702240917         10305568 2023-12-10T20:41:57Z
etcd-snapshot-rpb58-n01-1702243200.zip file:///var/lib/rancher/rke2/server/db/snapshots/
↪etcd-snapshot-rpb58-n01-1702243200.zip 1276185  2023-12-10T21:20:00Z
etcd-snapshot-rpb58-n01-1702243804    file:///var/lib/rancher/rke2/server/db/snapshots/
↪etcd-snapshot-rpb58-n01-1702243804     6012960  2023-12-10T21:30:04Z
etcd-snapshot-rpb58-n01-1702244403    file:///var/lib/rancher/rke2/server/db/snapshots/
↪etcd-snapshot-rpb58-n01-1702244403     6340640  2023-12-10T21:40:03Z
on-demand-rpb58-n01-1702244969        file:///var/lib/rancher/rke2/server/db/snapshots/
↪on-demand-rpb58-n01-1702244969         6340640  2023-12-10T21:49:29Z
etcd-snapshot-rpb58-n01-1702245002    file:///var/lib/rancher/rke2/server/db/snapshots/
↪etcd-snapshot-rpb58-n01-1702245002     6340640  2023-12-10T21:50:02Z
```

Say we wanted to restore the snapshot of dd `2023-12-10T21:50:02Z`

```
▶ sudo rke2 server --cluster-reset \
                   --cluster-reset-restore-path= \
                   /var/lib/rancher/rke2/server/db/snapshots/etcd-snapshot-rpb58-n01-
↪1702245002
```

This assumes that the snapshot path is the default one. The name of the snapshot file itself can be found in above listing of the snapshots.

Next step is to start the RKE2 server again with the restored etcd database:

```
▶ sudo systemctl start rke2-server
```

Check if the server is up-and-running again:

```
▶ kubectl get nodes
```

5 RKE2 and FIPS/CIS installs

5.1 Introduction

In this lab we are going to practice with installing RKE2 according to the security specifications of FIPS and CIS 1.23. We are going to assess how much of the CIS 1.23 requirements are covered out of the box after a CIS 1.23 profile install of RKE2 and how to close the final gaps in addition to mitigating the ones that we cannot close.

5.1.1 Requirements

The requirements for installing hardened RKE2 in this lab are the same as installing RKE2 with single control plane specifications. For simplicity we will repeat these requirements below.

To be able to execute this lab, you need at least 2 Ubuntu 22.04 (LTS) systems. If you would like to add more `rke2-agent` nodes you can always add more machines. The virtual machines must be provided with at least 4 GB Memory, 2 CPUs and enough storage to store some containers (approximately 8 GB free space should be ok). This lab has been tested on both x86_64 (AMD64) as wel as aarch64 (ARM64) systems. So they should also work on ARM64 architectures like RPi4, RPi5, ARM64 ODROIDs, RockPi4, OrangePi4 (plus) etc. RKE2 does not support RISCV64 yet.

Furthermore, you need an Internet connection to be able to access the RKE2 distribution and download some containers from quay.io and docker.io.

In this lab we assume that there is an unprivileged user, named 'student01' available on this system. This user should be able to execute privileged commands using the `sudo` utility. Of course your user may have a different name, important is the access to `sudo su - root` privileges.

If not already done, you can create on all nodes this user as root:

```
$> useradd -m -s /bin/bash student

$> passwd student01

$> echo "student ALL=(ALL) NOPASSWD: ALL" | tee /etc/sudoers.d/89-student
```

Login again as the user student, and proceed this lab.

Bring your installation up-to-date, using:

```
[~ $> sudo apt update

[~ $> sudo apt upgrade -y
```

If systemd or the kernel is updated during this process, you need to reboot your system. If you are not sure: reboot it anyway.

```
[~ $> sudo systemctl reboot
```

Install the following software requirements:

```
[~ $> sudo apt -y install vim info wget curl elinks man-db manpages \
      bash-completion psmisc jq ipvsadm yamllint conntrack \
      apt-transport-https pinfo
```

Execute the update of your system and the installation of the software on each node.

5.2 Install RKE2

We will now use a 2 step process to install K8S using the RKE2 distribution on your systems:

1. Install RKE2 server on the system what will fulfill the role of master node for our K8S cluster.
2. Join the other servers that will play the role of (worker-)nodes by installing and configuring rke2-server on these.

In the examples shown here under we will have 3 systems:

Nr	Hostname	Role	Config
1	k8sc903n01	RKE2 server	2cpu, 8GiB RAM
2	k8sc903n02	RKE2 agent #1	2cpu, 8GiB RAM
3	k8sc903n03	RKE2 agent #2	2cpu, 8GiB RAM

5.2.1 Install RKE2 server node

While RKE2 is designed to be hardened by default, it will not pass all Kubernetes CIS controls without modification. To fully pass the CIS benchmark, we will make a few manual modifications. The first of which will be done on the host operating system, for RKE2 does not modify this. The second patch of modifications must be made because certain CIS controls for Network Policies and Pod Security Standards restrict the functionality of our cluster. When we do chose our security over our freedom, we need to opt into having RKE2 configure these for us, by starting RKE2 with the `profile` flag set to `cis-1.23`.

5.2.2 More requirements

In addition to installing RKE2, we also have some host-level requirements when opting for CIS hardening.

5.3 Ensure protect-kernel-defaults are set

When RKE2 is installed via tarball, as is usually the case on OSes that do not use RPMs, such as, in our case, Ubuntu, we use the following commands to set the kernel default protection flags:

```
$> sudo cp -f /usr/local/share/rke2/rke2-cis-sysctl.conf /etc/sysctl.d/60-rke2-cis.conf
$> sudo systemctl restart systemd-sysctl
```

When your OS does use RPMs, such as CentOS on mac, RKE2 is usually installed via RPM, YUM or DNF, and you can use the following commands to set the kernal default protection flags instead:

```
$> sudo cp -f /usr/share/rke2/rke2-cis-sysctl.conf /etc/sysctl.d/60-rke2-cis.conf
$> sudo systemctl restart systemd-sysctl
```

We can verify whether we have set our flags correctly by running

Which should display the flags such as below:

```
$> vm.panic_on_oom=0
$> vm.overcommit_memory=1
$> kernel.panic=10
$> kernel.panic_on_oops=1
```

To give some additional information about what this exactly does: #. vm.panic_on_oom=0 states that when our virtual machine runs out of memory, we do not want to panic but instead kill some other processes to free up memory #. vm.overcommit_memory=1 will allow the virtual machine to allocate more memory than is physically available. The kernel will handle situations where we try to allocate memory that does not exist, hence why this will allow to use our available more efficiently. #. kernel.panic=10 sets the amount of seconds the kernel will continue running before rebooting after encountering a panic to 10 seconds. This gives the kernel enough time to write and store a log containing the cause of the panic, and maybe back some data up, without causing a down-time that is too long for the user's service. #. kernel.panic_on_oops=1 is a safety measure that states that, when the kernel encounters an oops (a non-fatal kernel error), it should panic either way and reboot just to be safe.

5.4 create the etcd user

Finally, we need to create an etcd `user` and group to achieve our full CIS Benchmark.

For this we run the following command

```
$> sudo useradd -r -c "etcd user" -s /sbin/nologin -M etcd -U
```

5.4.1 Hardening RKE2

In order to properly harden our RKE2 server, we start by creating a `server-config.yaml` file to store the necessary flag. But before we do that, we must first create the directory where we need to store this yaml-file, if it does not exist already:

```
$> sudo mkdir -p /etc/rancher/rke2
```

```
$> sudo vi /etc/rancher/rke2/config.yaml
```

And edit it to contain the following information

```
profile: cis-1.23
```

Next, we need to boot our server using the following command, which will run it with our specified flags:

```
$> sudo systemctl enable rke2-server.service --now
```

We can verify our server has boot correctly by running

```
$> sudo journalctl -u rke2-server -lf
```

Finally, we copy the kube config file and retrieve the node-token to assign nodes to it later:

```
$> mkdir -p ~/.kube

$> sudo cp /etc/rancher/rke2/rke2.yaml ~/.kube/config

$> sudo chown ${USER}:${USER} ~/.kube/config

$> sudo cat /var/lib/rancher/rke2/server/node-token
```

Make sure to save the token obtained by running these commands, as we will need it in the next chapter.

5.4.2 installing and configuring kubectl

On the RKE2 server, install and configure kubectl by running the following code in a file called `install-kubectl.sh`

```
#!/bin/sh
ARCH="$(uname -m)"

case "$ARCH" in
 x86_64)
   ARCH="amd64"
   ;;
 aarch64)
   ARCH="arm64"
   ;;
esac

curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/
↪linux/${ARCH}/kubectl"
sudo mv kubectl /usr/local/bin
sudo chmod +x /usr/local/bin/kubectl
kubectl get nodes
```

Of course we execute the file by calling the following commands:

```
$> chmod +x install-kubectl.sh

$> ./install-kubectl.sh
```

5.4.3 agent-config.yaml

On the RKE2 server, create a file called `agent-config.yaml` with the following content. Recall that the node-token is copied from the last line of the last chapter, `Hardening RKE2`:

```
server: https://<rke2-server-hostname>:9345
token: <node-token>
```

5.4.4 installing a node

On every agent node you want to connect to the master node:

Make a file called `rke-agent.sh` containing the following code and execute it:

```
# Download and install the RKE2 agent binary

curl -sfL https://get.rke2.io | sudo INSTALL_RKE2_TYPE="agent" sh -

# Copy the agent-config.yaml file from rke2-server to rke2-agent's /etc

sudo mkdir -p /etc/rancher/rke2

sudo cp agent-config.yaml /etc/rancher/rke2/config.yaml

sudo systemctl enable rke2-agent.service --now

# Enable and start rke2-agent service to configure RKE2 agent node

sudo systemctl enable rke2-agent.service --now

# Optionally verify RKE2 agent logs with:

sudo journalctl -u rke2-agent -lf
```

6 Upgrading RKE2 installations

6.1 Introduction

In this lab we are going to practice with the 2 different methods we can use to upgrade our RKE2 clusters:

1. Manually upgrading the cluster by upgrading/installing a new RKE2 server and agent version.
2. Automatically upgrading our cluster by using the Rancher Upgrade Controller

6.1.1 Requirements

An up-and-running RKE2 cluster with at least one rke2-server node and one rke2-agent node.

6.1.2 Manual upgrade

First we are going to practice with the manual upgrade. This should work on single controlplane (1 rke-servers) as well as HA controlplane (multiple rke2-server) configuration. In contrast to the automated upgrade method this will also work on ARM64 based RKE2 clusters.

In our example the IST situation will be an 8 node cluster running version v1.27.8+rke2r1

Listing 1: Output of `kubectl get nodes` prio to upgrade

```
NAME         STATUS   ROLES                       AGE     VERSION
k8sc270n01   Ready    control-plane,etcd,master   4m6s    v1.27.8+rke2r1
k8sc270n02   Ready    <none>                      2m13s   v1.27.8+rke2r1
k8sc270n03   Ready    <none>                      2m13s   v1.27.8+rke2r1
k8sc270n04   Ready    <none>                      2m12s   v1.27.8+rke2r1
k8sc270n05   Ready    <none>                      2m21s   v1.27.8+rke2r1
k8sc270n06   Ready    <none>                      2m15s   v1.27.8+rke2r1
k8sc270n07   Ready    <none>                      97s     v1.27.8+rke2r1
k8sc270n08   Ready    <none>                      97s     v1.27.8+rke2r1
```

Preparation

1. Backup the cluster
2. Create an ETCD snapshot

   ```
   ▶ sudo rke2 etcd-snapshot save
   ```

3. Export kubernetes resources

   ```
   ▶ kubectl get --all-namespaces --export -o yaml > all-deployments.yaml
   ```

   Mind that this will not export all the api-resources. A K8S backuptool is highly recommended
4. Backup Node Configuration
   Save copies of your RKE2 server and agent configuration files.
5. Read the release notes for the new RKE2 release you want to upgrade to, to be aware of any changes that could impact your cluster or applications.
6. Test in a Staging Environment
   If possible, replicate your production environment and perform a test upgrade to identify potential issues.

Upgrade the control-plane(s)

The control-plane nodes (rke2-servers) need to be upgraded one-by-one.

1. Drain the first (or only) rke2-server. Here that's `k8sc270n01`:

   ```
   ▶ kubectl drain k8sc270n01 --ignore-daemonsets --delete-local-data
   ```

   Explanation:
   - The `kubectl drain <node-name> --ignore-daemonsets --delete-local-data` command is used in Kubernetes to prepare a node for maintenance or upgrade. Here's what each part of the command does:
     - `kubectl drain <node-name>` This is the basic command to drain a node in Kubernetes. "Draining" a node involves safely evicting all the pods running on it, except for those that cannot be moved, such as static pods managed directly by the Kubelet. This is done to ensure that the node can be safely taken down for maintenance or upgrade without disrupting the services running on the cluster.
     - `--ignore-daemonsets` DaemonSets are a type of workload in Kubernetes that ensures that all (or some) nodes run a copy of a pod. When you drain a node, you usually don't want to remove these DaemonSet-managed pods. The –ignore-daemonsets flag tells kubectl drain to ignore pods that are managed by DaemonSets. Without this flag, the drain command would fail because it cannot evict these pods as they are automatically recreated on the node.
     - `--delete-emptydir-data` This flag tells the drain command to continue even if there are pods using emptyDir volumes on the node. The emptyDir volume is a temporary directory that shares a pod's lifetime. Using this flag means any data in an emptyDir volume will be deleted when the pod is evicted. Without this flag, the drain command would fail if such pods were present, as the command by default does not evict pods with local storage to prevent data loss.

1. Stop the RKE2 Service on the RKE2 server

```
▶ sudo systemctl stop rke2-server
```

2. Download the RKE2 install script for the RKE2 version you want to upgrade to (here v1.28.4+rke2r1)

```
▶ curl -sfL https://get.rke2.io | sudo INSTALL_RKE2_VERSION=v1.28.4+rke2r1 sh -
```

3. Start the RKE2 service on this rke2-server node again

```
▶ sudo systemctl start rke2-server
```

4. Verify the status of the service to see if rke2-server is running fine

```
▶ sudo systemctl status rke2-server
```

5. Use kubectl to check if the node is properly upgraded and in ready state

```
▶ kubectl get nodes
```

6. Uncordon the rke2-server node (here k8sc270n01)

```
▶ kubectl uncordon k8sc270n01
```

Listing 2: Output of `kubectl uncordon`

```
node/k8sc270n01 uncordoned
```

And check with `kubectl get nodes` again the status after uncordoning

```
▶ kubectl get nodes
```

Listing 3: Output of `kubectl get nodes` post kubectl uncordon

```
NAME         STATUS   ROLES                       AGE   VERSION
k8sc270n01   Ready    control-plane,etcd,master   43m   v1.28.4+rke2r1
k8sc270n02   Ready    <none>                      42m   v1.27.8+rke2r1
k8sc270n03   Ready    <none>                      42m   v1.27.8+rke2r1
k8sc270n04   Ready    <none>                      42m   v1.27.8+rke2r1
k8sc270n05   Ready    <none>                      42m   v1.27.8+rke2r1
k8sc270n06   Ready    <none>                      42m   v1.27.8+rke2r1
k8sc270n07   Ready    <none>                      41m   v1.27.8+rke2r1
k8sc270n08   Ready    <none>                      41m   v1.27.8+rke2r1
```

7. Last check, take a look at the K8S engineroom in the `kube-system` namespace

```
▶ kubectl get pods -n kube-system
```

Listing 4: Output of `kubectl get pods -n kube-system`

```
NAME                                                READY   STATUS      RESTARTS
↪   AGE
cloud-controller-manager-k8sc270n01                 1/1     Running     1 (3m34s
↪ago)   3m31s
etcd-k8sc270n01                                     1/1     Running     0
↪   3m31s
helm-install-rke2-canal-r42kv                       0/1     Completed   0
↪   2m59s
helm-install-rke2-coredns-b826c                     0/1     Completed   0
↪   2m59s
helm-install-rke2-ingress-nginx-t8l52               0/1     Completed   0
↪   2m59s
helm-install-rke2-metrics-server-wfsrz              0/1     Completed   0
↪   2m59s
helm-install-rke2-snapshot-controller-crd-2mxpb     0/1     Completed   0
↪   2m59s
```

(continues on next page)

(continued from previous page)

```
helm-install-rke2-snapshot-controller-vslzx            0/1     Completed   0
↪    2m59s
helm-install-rke2-snapshot-validation-webhook-nghrd    0/1     Completed   0
↪    2m59s
kube-apiserver-k8sc270n01                              1/1     Running     0
↪    3m31s
kube-controller-manager-k8sc270n01                     1/1     Running     1 (3m34s
↪ago)   3m31s
kube-proxy-k8sc270n01                                  1/1     Running     0
↪    2m53s
kube-proxy-k8sc270n02                                  1/1     Running     0
↪    43m
kube-proxy-k8sc270n03                                  1/1     Running     0
↪    43m
kube-proxy-k8sc270n04                                  1/1     Running     0
↪    43m
kube-proxy-k8sc270n05                                  1/1     Running     0
↪    43m
kube-proxy-k8sc270n06                                  1/1     Running     0
↪    43m
kube-proxy-k8sc270n07                                  1/1     Running     0
↪    43m
kube-proxy-k8sc270n08                                  1/1     Running     0
↪    43m
kube-scheduler-k8sc270n01                              1/1     Running     0
↪    3m31s
rke2-canal-4dwkx                                       2/2     Running     0
↪    43m
rke2-canal-64c6q                                       2/2     Running     0
↪    43m
rke2-canal-8tfwm                                       2/2     Running     8 (14m ago)
↪    43m
rke2-canal-bmj8q                                       2/2     Running     2 (7m9s ago)
↪    43m
rke2-canal-dtmn9                                       2/2     Running     0
↪    43m
rke2-canal-fmfdp                                       2/2     Running     9 (9m21s
↪ago)   43m
rke2-canal-hr7w9                                       2/2     Running     2 (32m ago)
↪    45m
rke2-canal-vd7sm                                       2/2     Running     0
↪    43m
rke2-coredns-rke2-coredns-6b795db654-ct25j             1/1     Running     0
↪    107s
rke2-coredns-rke2-coredns-6b795db654-dvz44             1/1     Running     0
↪    106s
rke2-coredns-rke2-coredns-autoscaler-945fbd459-99hxz   1/1     Running     0
↪    107s
rke2-ingress-nginx-controller-75xff                    1/1     Running     0
↪    44m
rke2-ingress-nginx-controller-gd6dc                    1/1     Running     0
↪    42m
rke2-ingress-nginx-controller-hmf69                    1/1     Running     0
↪    42m
rke2-ingress-nginx-controller-hs6hz                    1/1     Running     0
↪    42m
rke2-ingress-nginx-controller-jh55l                    1/1     Running     0
↪    42m
rke2-ingress-nginx-controller-n4bgc                    1/1     Running     0
↪    42m
rke2-ingress-nginx-controller-pl4zt                    1/1     Running     0
```

(continues on next page)

(continued from previous page)

```
↪   41m
rke2-ingress-nginx-controller-pr6gr                  1/1     Running     0
↪   42m
rke2-metrics-server-544c8c66fc-jvfl8                 1/1     Running     0
↪   2m40s
rke2-snapshot-controller-7d6476d7cb-2klcb            1/1     Running     1 (3m41s
↪ago)   5m56s
rke2-snapshot-validation-webhook-5649fbd66c-w8m4x    1/1     Running     0
↪   5m57s
```

8. Repeat above steps for every node in your control-plane.
   Only when all the control plane nodes have been successfully upgraded continue with upgrading the worker nodes aka rke2-agents

Upgrade RKE2 agents

Also here the rke2-agent nodes are upgraded one-by-one!

1. Drain the next rke2-agent. Here that's `k8sc270n02`:

   ```
   ▶ kubectl drain k8sc270n02 --ignore-daemonsets --delete-local-data
   ```

2. Stop the RKE2 Service on this RKE2 agent

   ```
   ▶ sudo systemctl stop rke2-agent
   ```

3. Download the RKE2 install script for the RKE2 version you want to upgrade to (here v1.28.4+rke2r1)

   ```
   ▶ curl -sfL https://get.rke2.io | sudo INSTALL_RKE2_TYPE="agent" INSTALL_RKE2_
   ↪VERSION=v1.28.4+rke2r1 sh -
   ```

4. Start the RKE2 service on this rke2-agent node again

   ```
   ▶ sudo systemctl start rke2-agent
   ```

5. Verify the status of the service to see if rke2-agent is running fine

   ```
   ▶ sudo systemctl status rke2-agent
   ```

6. Use kubectl to check if this node is properly upgraded and in ready state

   ```
   ▶ kubectl get nodes
   ```

7. Uncordon this rke2-agent node (here k8sc270n02)

   ```
   ▶ kubectl uncordon k8sc270n02
   ```

   And check with `kubectl get nodes` again the status after uncordoning

   ```
   ▶ kubectl get nodes
   ```

8. Repeat above steps for every node that has the role of worker / rke2-agent
   Only when all the rke2-agent nodes have been successfully upgraded the upgrade process is completed.

6.1.3 Automatic upgrades

RKE2 let's you automate the upgrade of your RKE2 cluster using a special upgrade-controller. This operator let's you plan the upgrade of both control-plane and worker nodes. The operator is available for ARM64 but the container images needed for the upgrade on ARM64 platforms are not. So this method is currently only supported on AMD64 platforms.

In order to make use of the upgrade controller one needs to:

1. Install the Rancher upgrade controller.
2. Create upgrade plans for control-plane and worker nodes (CRDs).
3. Flag each nodes when they are ready for upgrade

Install the Rancher upgrade controller

To install the `Rancher upgrade controller` aka `system-upgrade-controller` log on to one of your `rke2-server` nodes and issue the following command:

```
▶ kubectl https://github.com/rancher/system-upgrade-controller/releases/download/v0.13.2/
↪system-upgrade-controller.yaml
```

This will create the namespace `system-upgrade` and deploy the `system-upgrade-controller` in it. Any jobs that will be deployed by the `system-upgrade-controller` will appear in this namespace. You can verify the deployment of the system-upgrade-controller with the following command:

```
▶ kubectl get pods -n system-upgrade
```

Create and submit the upgrade plan

The `system-upgrade-controller` consumes Custom Resources called `plans`. These plans are just plain YAML describing how the `rke2-server` and `rke2-agent` nodes should be upgraded.

An example:

Listing 5: upgrade-rke2-plan.yaml

```
# Server plan
apiVersion: upgrade.cattle.io/v1
kind: Plan
metadata:
  name: server-plan
  namespace: system-upgrade
  labels:
    rke2-upgrade: server
spec:
  concurrency: 1
  nodeSelector:
    matchExpressions:
       - {key: rke2-upgrade, operator: Exists}
       - {key: rke2-upgrade, operator: NotIn, values: ["disabled", "false"]}
       # When using k8s version 1.19 or older, swap control-plane with master
       - {key: node-role.kubernetes.io/control-plane, operator: In, values: ["true"]}
  serviceAccountName: system-upgrade
  cordon: true
  drain:
    force: true
  upgrade:
    image: rancher/rke2-upgrade
  version: v1.28.4+rke2r1
---
# Agent plan
apiVersion: upgrade.cattle.io/v1
```

(continues on next page)

(continued from previous page)

```
kind: Plan
metadata:
  name: agent-plan
  namespace: system-upgrade
  labels:
    rke2-upgrade: agent
spec:
  concurrency: 1
  nodeSelector:
    matchExpressions:
      - {key: rke2-upgrade, operator: Exists}
      - {key: rke2-upgrade, operator: NotIn, values: ["disabled", "false"]}
      # When using k8s version 1.19 or older, swap control-plane with master
      - {key: node-role.kubernetes.io/control-plane, operator: NotIn, values: ["true"]}
  prepare:
    args:
    - prepare
    - server-plan
    image: rancher/rke2-upgrade
  serviceAccountName: system-upgrade
  cordon: true
  drain:
    force: true
  upgrade:
    image: rancher/rke2-upgrade
  version: v1.28.4+rke2r1
```

This YAML example contains actually 2 plans.

1. One plan for upgrading the `rke2-server` nodes
2. One plan for upgrading the `rke2-agent` nodes

Server Plan

The *server-plan* is designed to upgrade RKE2 server nodes.

- **Concurrency**: Specifies the number of nodes to upgrade simultaneously. Set to *1* to upgrade one node at a time.
- **NodeSelector**: Determines which nodes are targeted for the upgrade. In this plan, nodes with the label *rke2-upgrade* and not labeled as *disabled* or *false* are selected. Also, nodes labeled as *control-plane* are targeted.
- **ServiceAccountName**: The service account used to perform the upgrade.
- **Cordon**: If set to *true*, it cordons the node before upgrading, preventing new pods from being scheduled on it.
- **Drain**: Drains the node of workloads before upgrading. *force: true* forces eviction of pods.
- **Upgrade Image**: The Docker image used to perform the upgrade, specified as *rancher/rke2-upgrade*.
- **Version**: The target version for the upgrade, *v1.28.4+rke2r1* in this case.

Agent Plan

The *agent-plan* upgrades the RKE2 agent nodes.

- **Concurrency**: As with the server plan, it's set to *1*.
- **NodeSelector**: Selects agent nodes for the upgrade, excluding those labeled as *control-plane*.
- **Prepare**: Prepares agent nodes by invoking the *server-plan*. This ensures agents are compatible with server versions.
- **Other Parameters**: Similar to the server plan, including cordon, drain, upgrade image, and version.

Flag the nodes for upgrade

This step is forgotten by a lot of K8S operators. Nothing will happen unless we will flag the NODES to be suitable for upgrading. This means you need to set a label (according to the above plan: `rke2-upgrade=true`) on each node we have in our cluster that needs to be upgraded. In most cases, that means all of the nodes. You can do that manually or use a script like this:

```
for node in `kubectl get node -o name | awk -F '/' '{print $2}'`
do
    kubectl label node ${node} rke2-upgrade=true --overwrite
done
```

Please note that the upgrade process will not remove the labels, so you must do that manually after the upgrade has been succesfully completed, to make sure you don't get instant surprises on the planning of the next upgrade.

Upgrade Process

1. **Server Upgrade**: The system-upgrade-controller first upgrades server nodes as per the *server-plan*.
2. **Agent Upgrade**: After server nodes are upgraded, agent nodes are upgraded following the *agent-plan*.

Jobs will be created in the `system-upgrade` namespace for each node that will be upgraded. Nodes that have been upgraded can be recognized with `kubectl get node <nodename>` as they will report the newly upgraded version. You can also see their corresponding job having the `completed` status when you do a `kubectl get pods -n system-upgrade`

During and after the upgrade you might see some PODs from upgrade JOBs in 'Unknown' state. This should not give cause to concern. If you do an:

- kubectl get pods -n system-upgrade -o wide

You will see that each job actually runs on the NODE it's going to upgrade. Hence, once the NODE gets down that JOB will be drained with the rest of the PODs. A new POD will be deployed after the NODE comes up again to finalize the upgrade tasks.

Precautions and Preparation

- **Backup**: Ensure you have a complete backup of the cluster.
- **Monitoring**: Monitor the upgrade process, especially to check for any errors or issues.
- **Version Compatibility**: Ensure the target version is compatible with your current cluster setup.
- **Node Health**: Check the health of all nodes before starting the upgrade.