Running Kubernetes on your Raspberry Pi homelab

Build your own cloud at home and start experimenting with Kubernetes

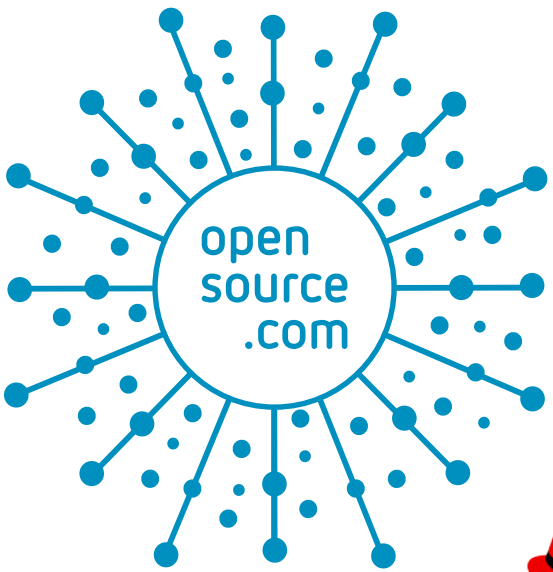
ABOUT OPENSOURCE.COM

What is Opensource.com?

OPENSOURCE.COM publishes stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Submit a story idea: opensource.com/story

Email us: open@opensource.com



Supported by
Red Hat

CHRIS COLLINS

CHRIS COLLINS IS AN SRE at Red Hat and a Community Moderator for OpenSource.com. He is a container and container orchestration, DevOps, and automation evangelist, and will talk with anyone interested in those topics for far too long and with much enthusiasm.

Prior to working at Red Hat, Chris spent thirteen years with Duke University, variously as a Linux systems administrator, web hosting architecture and team lead, and an automation engineer.

In his free time, Chris enjoys brewing beer, woodworking, and being a general-purpose geek.

FOLLOW CHRIS COLLINS

Twitter: @ChrisInDurham

CHAPTERS

Introduction

THE MOST EFFECTIVE WAY TO LEARN a new technology is to use it. Building a homelab has become increasingly accessible thanks to the affordability and versatility of the Raspberry Pi single-board computer. There are thousands of things you can do with a Raspberry Pi, including experimenting with Kubernetes.

Kubernetes was initially released in 2014 and has since defined how we build and interact with cloud technology. But if you think Kubernetes is too elusive or challenging to grok outside of an enterprise environment, think again. In this eBook, author Chris Collins demonstrates how to get started running Kubernetes on your Raspberry Pi. He also provides several different techniques for optimizing your homelab's Kubernetes clusters. Each section of this guide can be treated in isolation or as a holistic project. Regardless of your day job, reading these tutorials and trying them out at your own pace is sure to boost your cloud technology prowess.

Modify a disk image to create a Raspberry Pi-based homelab

Create a "private cloud at home" with a Raspberry Pi or other single-board computer.

BUILDING A HOMELAB [1] can be a fun way to entertain yourself while learning new concepts and experimenting with new technologies. Thanks to the popularity of single-board computers (SBCs), led by the Raspberry Pi [2], it is easier than ever to build a multi-computer lab right from the comfort of your home. Creating a "private cloud at home" is also a great way to get exposure to cloud-native technologies for considerably less money than trying to replicate the same setup with a major cloud provider.

This article explains how to modify a disk image for the Raspberry Pi or another SBC, pre-configure the host for SSH (secure shell), and disable the service that forces interaction for configuration on first boot. This is a great way to make your devices "boot and go," similar to cloud instances. Later, you can do more specialized, in-depth configurations using automated processes over an SSH connection.

Also, as you add more Pis to your lab, modifying disk images lets you just write the image to an SD card, drop it into the Pi, and go!

Decompress and mount the image

For this project, you need to modify a server disk image. I used the Fedora Server 31 ARM [3] image during testing. After you download the disk image and verify its checksum [4], you need to decompress and mount it to a location on the host computer's file system so you can modify it as needed.

You can use the **xz** [5] command to decompress the Fedora Server image by using the --decompress argument:

```
xz --decompress Fedora-Server-armhfp-X-y.z-sda.raw.xz
```

This leaves you with a raw, decompressed disk image (which automatically replaces the **.xz** compressed file). This raw disk image is just what it sounds like: a file containing all the data that would be on a formatted and installed disk. That includes partition information, the boot partition, the root partition, and any other partitions. You need to mount the partition you intend to work in, but to do that, you need information about where that partition starts in the disk image and the size of the sectors on the disk, so you can mount the file at the right sector.

Luckily, you can use the **fdisk** [6] command on a disk image just as easily as on a real disk and use the **--list** or **-l** argument to view the list of partitions and their information:

```
# Use fdisk to list the partitions in the raw image:
$ fdisk -l Fedora-Server-armhfp-31-1.9-sda.raw
Disk Fedora-Server-armhfp-X-y.z-sda.raw: 3.2 GiB, 3242196992 bytes,
  6332416 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xdaad9f57

Device                                Boot   Start     End Sectors  Size Id Type
Fedora-Server-armhfp-X-y.z-sda.raw1         8192  163839  155648   76M  c W95 F
Fedora-Server-armhfp-X-y.z-sda.raw2 *     163840 1163263  999424  488M 83 Linux
Fedora-Server-armhfp-X-y.z-sda.raw3     1163264 6047743 4884480  2.3G 83 Linux
```

All the information you need is available in this output. Line 3 indicates the sector size, both logical and physical: (512 bytes / 512 bytes).

The list of devices shows the partitions inside the raw disk image. The first one, **Fedora-Server-armhfp-X-y.z-sda.raw1**

is no doubt the bootloader partition because it is the first, small (only 76MB), and type W95 FAT32 (LBA), as identified by the Id "c," a FAT32 partition for booting off the SD card.

The second partition is not very large either, just 488MB. This partition is a Linux native-type partition (Id 83), and it probably is the Linux boot partition containing the kernel and initramfs [7].

The third partition is what you probably want: it is 2.3GB, so it should have the majority of the distribution on it, and it is a Linux-native partition type, which is expected. This should contain the partition and data you want to modify.

The third partition starts on sector 1163264 (indicated by the "Start" column in the output of **fdisk**), so your mount offset is **595591168**, calculated by multiplying the sector size (512) by the start sector (1163264) (i.e., **512 * 1163264**). This means you need to mount the file with an offset of 595591168 to be in the right place at the mount point.

ARMed (see what I did there?) with this information, you can now mount the third partition to a directory in your homedir:

```
$ mkdir ~/mnt
$ sudo mount -o loop,offset=595591168
  Fedora-Server-armhfp-X-y.z-sda.raw ~/mnt
$ ls ~/mnt
```

Work directly within the disk image

Once the disk image has been decompressed and mounted to a spot on the host computer, it is time to start modifying the image to suit your needs. In my opinion, the easiest way to make changes to the image is to use chroot to change the working root of your session to that of the mounted image. There's a tricky bit, though.

When you change root, your session will use the binaries from the new root. Unless you are doing all of this from an ARM system, the architecture of the decompressed disk image will not be the same as the host system you are using. Even inside the chroot, the host system will not be able to make use of binaries with a different architecture. At least, not natively.

Luckily, there is a solution: **qemu-user-static**. From the Debian Wiki [8]:

> "[qemu-user-static] provides the user mode emulation binaries, built statically. In this mode QEMU can launch Linux processes compiled for one CPU on another CPU… If binfmt-support package is installed, qemu-user-static package will register binary formats which the provided emulators can handle, so that it will be possible to run foreign binaries directly."

This is exactly what you need to be able to work in the non-native architecture inside your chroot. If the host system is Fedora, install the **qemu-user-static** package with DNF, and restart **systemd-binfmt.service**:

```
# Enable non-native arch chroot with DNF, adding new binary
  format information
# Output suppressed for brevity
$ dnf install qemu-user-static
$ systemctl restart systemd-binfmt.service
```

With this, you should be able to change root to the mounted disk image and run the **uname** command to verify that everything is working:

```
sudo chroot ~/mnt/ /usr/bin/uname -a -r
Linux marvin 5.5.16-200.fc31.x86_64 #1 SMP Wed Apr 8 16:43:33
  UTC 2020 armv7l armv7l armv7l GNU/Linux
```

Running **uname** from within the changed root shows **armv7l** in the output—the architecture of the raw disk image—and not the host machine. Everything is working as expected, and you can continue on to modify the image.

Modify the disk image

Now that you can change directly into the ARM-based disk image and work in that environment, you can begin to make changes to the image itself. You want to set up the image so it can be booted and immediately accessed without having to do any additional setup directly on the Raspberry Pi. To do this, you need to install and enable sshd (the OpenSSH daemon) and add the authorized keys for SSH access.

And to make this behave more like a cloud environment and realize the dream of a private cloud at home, add a local user, give that user sudo rights, and (to be just like the cloud heavyweights) allow that user to use sudo without a password.

So, your to-do list is:

- Install and enable SSHD (SSHD is already installed and enabled in the Fedora ARM image, but you may need to do this manually for your distribution)
- Set up a local user
- Allow the local user to use sudo (without a password, optional)
- Add authorized keys
- Allow root to SSH with the authorized keys (optional)

I use the GitHub feature that allows you to upload your public SSH keys and make them available at **https://github.com/<your_github_username>.keys** [9]. I find this to be a convenient way to distribute public keys, although I am paranoid enough that I always check that the downloaded keys match what I am expecting. If you don't want to use this method, you can copy your public keys into the chroot directly from your host computer, or you can host your keys on a web server that you control in order to use the same workflow.

To start modifying the disk image, **chroot** into the mounted disk image again, this time starting a shell so multiple commands can be run:

```
# Output of these commands (if any) are omitted for brevity
$ sudo chroot ~/mnt /bin/bash

# Install openssh-server and enable it (already done on Fedora)
$ dnf install -y openssh-server
$ systemctl enable sshd.service

# Allow root to SSH with your authorized keys
$ mkdir /root/.ssh

# Download, or otherwise add to the authorized_keys file, your
  public keys
# Replace the URL with the path to your own public keys
$ curl https://github.com/clcollins.keys -o /root/.ssh/
  authorized_keys
$ chmod 700 /root/.ssh
$ chmod 600 /root/.ssh/authorized_keys

# Add a local user, and put them in the wheel group
# Change the group and user to whatever you desire
groupadd chris
useradd -g chris -G wheel -m -u 1000 chris

# Download or add your authorized keys
# Change the homedir and URL as you've done above
mkdir /home/chris/.ssh
curl https://github.com/clcollins.keys -o
  /home/chris/.ssh/authorized_keys
chmod 700 /home/chris/.ssh
chmod 600 /home/chris/.ssh/authorized_keys
chown -R chris.chris /home/chris/.ssh/

# Allow the wheel group (with your local user) to use suso
  without a password
echo "%wheel ALL=(ALL) NOPASSWD:ALL" >>
  /etc/sudoers.d/91-wheel-nopasswd
```

This is all that generally needs to be done to set up SSH into a Raspberry Pi or other single-board computer on first boot. However, each distribution has its own quirks. For example, Rasbian already includes a local user, **pi**, and does not use the **wheel** group. So for Raspbian, it may be better to use the existing user or to delete the **pi** user and replace it with another.

In the case of Fedora ARM, the image prompts you to finish setup on first boot. This defeats the purpose of the changes you made above, especially since it blocks startup entirely until setup is complete. Your goal is to make the Raspberry Pi function like part of a private cloud infrastructure, and that workflow includes setting up the host remotely via SSH when it starts up. Disable the initial setup, controlled by the service **initial-setup.service**:

```
# Disable the initial-setup.service for both the multi-user and
  graphical targets
unlink /etc/systemd/system/multi-user.target.wants/
  initial-setup.service
unlink /etc/systemd/system/graphical.target.wants/
  initial-setup.service
```

While you are in the change root, you can make any other changes you might want for your systems or just leave it at that and follow the cloud-native workflow of configuration over SSH after first boot.

Recompress and install the modified image

With these changes to your system completed, all that is left is to recompress the disk image and install it on an SD card for your Raspberry Pi.

Make sure to exit the chroot, then unmount the disk image:

```
$ sudo umount ~/mnt/
```

Just as you decompressed the image initially, you can use the **xz** command again to compress the image. By using the **--keep** argument, **xz** will leave the raw image rather than cleaning it up. While this uses more disk space, leaving the uncompressed image allows you to make incremental changes to the images you are working with without needing to decompress them each time. This is great for saving time while testing and tweaking images to your liking:

```
# Compress the raw disk image to a .xz file, but keep the raw
  disk image
xz --compress Fedora-Server-armhfp-31-1.9-sda.raw --keep
```

The compression takes a while, so take this time to stand up, stretch, and get your blood flowing again.

Once the compression is done, the new, modified disk image can be copied to an SD card to use with a Raspberry Pi. The standard **dd** method to copy the image to the SD card works fine, but I like to use Fedora's **arm-image-installer** because of the options it provides when working with unedited images. It also works great for edited images and is a little more user-friendly than the **dd** command.

Make sure to check which disk the SD card is on and use that for the **--media** argument:

```
# Use arm-image-installer to copy the modified disk image to the
  SD card
arm-image-installer --image=Fedora-Server-armhfp-X-y.z-sda.raw.xz
   --target=rpi3 --media=/dev/sdc --norootpass --resizefs -y
```

Now you are all set with a new, modified Fedora Server ARM image for Raspberry Pis or other single board computers, ready to boot and immediately SSH into with your modifications. This method can also be used to make other changes, and you can use it with other distributions' raw disk images if you prefer them to Fedora. This is a good base to start building a private-cloud-at-home homelab. In future articles, I will guide you through setting up a homelab using cloud technologies and automation.

Further reading

A lot of research went into learning how to do the things in this article. Two of the most helpful sources I found for learning how to customize disk images and work with non-native architectures are listed below. They were extremely helpful in rounding the corner from "I have no idea what I'm doing" to "OK, I can do this!"

- How to modify a raw disk image of your custom Linux distro
- Using DNF wiki

Links

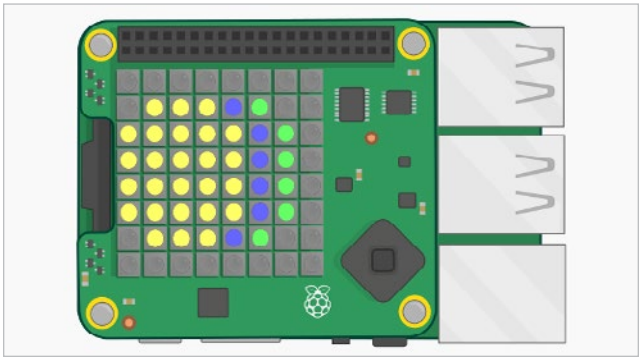
[1] https://opensource.com/article/19/3/home-lab
[2] https://opensource.com/resources/raspberry-pi
[3] https://arm.fedoraproject.org/
[4] https://arm.fedoraproject.org/verify.html
[5] https://tukaani.org/xz/
[6] https://en.wikipedia.org/wiki/Fdisk
[7] https://wiki.debian.org/initramfs
[8] https://wiki.debian.org/RaspberryPi/qemu-user-static
[9] https://github.com/<your_github_username>.keys

Build a Kubernetes cluster with the Raspberry Pi

Install Kubernetes on several Raspberry Pis for your own "private cloud at home" container service.

KUBERNETES [1] is an enterprise-grade container-orchestration system designed from the start to be cloud-native. It has grown to be the de-facto cloud container platform, continuing to expand as it has embraced new technologies, including container-native virtualization and serverless computing.

Kubernetes manages containers and more, from micro-scale at the edge to massive scale, in both public and private cloud environments. It is a perfect choice for a "private cloud at home" project, providing both robust container orchestration and the opportunity to learn about a technology in such demand and so thoroughly integrated into the cloud that its name is practically synonymous with "cloud computing."



Nothing says "cloud" quite like Kubernetes, and nothing screams "cluster me!" quite like Raspberry Pis. Running a local Kubernetes cluster on cheap Raspberry Pi hardware is a great way to gain experience managing and developing on a true cloud technology giant.

Install a Kubernetes cluster on Raspberry Pis

This exercise will install a Kubernetes 1.18.2 cluster on three or more Raspberry Pi 4s running Ubuntu 20.04. Ubuntu 20.04 (Focal Fossa) offers a Raspberry Pi-focused 64-bit ARM (ARM64) image with both a 64-bit kernel and user-space. Since the goal is to use these Raspberry Pis for running a Kubernetes cluster, the ability to run AArch64 container images is important: it can be difficult to find 32-bit images for common software or even standard base images. With its ARM64 image, Ubuntu 20.04 allows you to use 64-bit container images with Kubernetes.

AArch64 vs. ARM64; 32-bit vs. 64-bit; ARM vs. x86

Note that AArch64 and ARM64 are effectively the same thing. The different names arise from their use within different communities. Many container images are labeled AArch64 and will run fine on systems labeled ARM64. Systems with AArch64/ARM64 architecture are capable of running 32-bit ARM images, but the opposite is not true: a 32-bit ARM system cannot run 64-bit container images. This is why the Ubuntu 20.04 ARM64 image is so useful.

Without getting too deep in the woods explaining different architecture types, it is worth noting that ARM64/AArch64 and x86_64 architectures differ, and Kubernetes nodes running on 64-bit ARM architecture cannot run container images built for x86_64. In practice, you will find some images that are not built for both architectures and may not be usable in your cluster. You will also need to build your own images on an AArch64-based system or jump through some hoops to allow your regular x86_64 systems to build AArch64 images. In a future article in the "private cloud at home" project, I will cover how to build AArch64 images on your regular system.

For the best of both worlds, after you set up the Kubernetes cluster in this tutorial, you can add x86_64 nodes to it later. You can schedule images of a given architecture to run on the appropriate nodes by Kubernetes' scheduler through the use of Kubernetes taints and tolerations [2].

Enough about architectures and images. It's time to install Kubernetes, so get to it!

Requirements

The requirements for this exercise are minimal. You will need:

- Three (or more) Raspberry Pi 4s (preferably the 4GB RAM models)
- Install Ubuntu 20.04 ARM64 on all the Raspberry Pis

To simplify the initial setup, read *Modify a disk image to create a Raspberry Pi-based homelab* [3] to add a user and

SSH authorized_keys to the Ubuntu image before writing it to an SD card and installing on the Raspberry Pi.

Configure the hosts

Once Ubuntu is installed on the Raspberry Pis and they are accessible via SSH, you need to make a few changes before you can install Kubernetes.

Install and configure Docker

As of this writing, Ubuntu 20.04 ships the most recent version of Docker, v19.03, in the base repositories and can be installed directly using the apt command. Note that the package name is docker.io. Install Docker on all of the Raspberry Pis:

```
# Install the docker.io package
$ sudo apt install -y docker.io
```

After the package is installed, you need to make some changes to enable cgroups [4] (Control Groups). Cgroups allow the Linux kernel to limit and isolate resources. Practically speaking, this allows Kubernetes to better manage resources used by the containers it runs and increases security by isolating containers from one another.

Check the output of docker info before making the following changes on all of the RPis:

```
# Check `docker info`
# Some output omitted
$ sudo docker info
(...)
 Cgroup Driver: cgroups
(...)
WARNING: No memory limit support
WARNING: No swap limit support
WARNING: No kernel memory limit support
WARNING: No kernel memory TCP limit support
WARNING: No oom kill disable support
```

The output above highlights the bits that need to be changed: the cgroup driver and limit support.

First, change the default cgroups driver Docker uses from `cgroups` to `systemd` to allow systemd to act as the cgroups manager and ensure there is only one cgroup manager in use. This helps with system stability and is recommended by Kubernetes. To do this, create or replace the `/etc/docker/daemon.json` file with:

```
# Create or replace the contents of /etc/docker/daemon.json to
  enable the systemd cgroup driver

$ sudo cat > /etc/docker/daemon.json <<EOF
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "100m"
  },
  "storage-driver": "overlay2"
}
EOF
```

Enable cgroups limit support

Next, enable limit support, as shown by the warnings in the `docker info` output above. You need to modify the kernel command line to enable these options at boot. For the Raspberry Pi 4, add the following to the `/boot/firmware/cmdline.txt` file:

- `cgroup_enable=cpuset`
- `cgroup_enable=memory`
- `cgroup_memory=1`
- `swapaccount=1`

Make sure they are added to the end of the line in the `cmdline.txt` file. This can be accomplished in one line using `sed`:

```
# Append the cgroups and swap options to the kernel command line
# Note the space before "cgroup_enable=cpuset", to add a space
  after the last existing item on the line
$ sudo sed -i '$ s/$/ cgroup_enable=cpuset cgroup_enable=memory
  cgroup_memory=1 swapaccount=1/' /boot/firmware/cmdline.txt
```

The **sed** command matches the termination of the line (represented by the first `$`), replacing it with the options listed (it effectively appends the options to the line).

With these changes, Docker and the kernel should be configured as needed for Kubernetes. Reboot the Raspberry Pis, and when they come back up, check the output of docker `info` again. The `Cgroups driver` is now `systemd`, and the warnings are gone.

Allow iptables to see bridged traffic

According to the documentation, Kubernetes needs iptables to be configured to see bridged network traffic. You can do this by changing the `sysctl` config:

```
# Enable net.bridge.bridge-nf-call-iptables and -iptables6
cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
$ sudo sysctl --system
```

Install the Kubernetes packages for Ubuntu

Since you are using Ubuntu, you can install the Kubernetes packages from the Kubernetes.io Apt repository. There is not

currently a repository for Ubuntu 20.04 (Focal), but Kubernetes 1.18.2 is available in the last Ubuntu LTS repository: Ubuntu 18.04 (Xenial). The latest Kubernetes packages can be installed from there.

Add the Kubernetes repo to Ubuntu's sources:

```
# Add the packages.cloud.google.com atp key
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg
  | sudo apt-key add -

# Add the Kubernetes repo
cat <<EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF
```

When Kubernetes adds a Focal repository—perhaps when the next Kubernetes version is released—make sure to switch to it.

With the repository added to the sources list, install the three required Kubernetes packages: kubelet, kubeadm, and kubectl:

```
# Update the apt cache and install kubelet, kubeadm, and kubectl
# (Output omitted)
$ sudo apt update && sudo apt install -y kubelet kubeadm kubectl
```

Finally, use the `apt-mark hold` command to disable regular updates for these three packages. Upgrades to Kubernetes need more hand-holding than is possible with the general update process and will require manual attention:

```
# Disable (mark as held) updates for the Kubernetes packages
$ sudo apt-mark hold kubelet kubeadm kubectl
kubelet set on hold.
kubeadm set on hold.
kubectl set on hold.
```

That is it for the host configuration! Now you can move on to setting up Kubernetes itself.

Create a Kubernetes cluster

With the Kubernetes packages installed, you can continue on with creating a cluster. Before getting started, you need to make some decisions. First, one of the Raspberry Pis needs to be designated the Control Plane (i.e., primary) node. The remaining nodes will be designated as compute nodes.

You also need to pick a network CIDR [5] to use for the pods in the Kubernetes cluster. Setting the `pod-network-cidr` during the cluster creation ensures that the `podCIDR` value is set and can be used by the Container Network Interface (CNI) add-on later. This exercise uses the Flannel [6] CNI. The CIDR you pick should not overlap with any CIDR currently used within your home network nor one managed by your router or DHCP server. Make sure to use a subnet that is larger than you expect to need: there are ALWAYS more pods than you initially plan for! In this example, I will use 10.244.0.0/16, but pick one that works for you.

With those decisions out of the way, you can initialize the Control Plane node. SSH or otherwise log into the node you have designated for the Control Plane.

Initialize the Control Plane

Kubernetes uses a bootstrap token to authenticate nodes being joined to the cluster. This token needs to be passed to the `kubeadm init` command when initializing the Control Plane node. Generate a token to use with the `kubeadm token generate` command:

```
# Generate a bootstrap token to authenticate nodes joining the
  cluster
$ TOKEN=$(sudo kubeadm token generate)
$ echo $TOKEN
d584xg.xupvwv7wllcpmwjy
```

You are now ready to initialize the Control Plane, using the `kubeadm init` command:

```
# Initialize the Control Plane
# (output omitted)
$ sudo kubeadm init --token=${TOKEN} --kubernetes-version=v1.18.2
  --pod-network-cidr=10.244.0.0/16
```

If everything is successful, you should see something similar to this at the end of the output:

```
Your Kubernetes control-plane has initialized successfully!
To start using your cluster, you need to run the following as a
regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options
listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the
following on each as root:

kubeadm join 192.168.2.114:6443 --token zqqoy7.9oi8dpkfmqkop2p5 \
  --discovery-token-ca-cert-hash sha256:71270ea1372144 22221319
  c1bdb9ba6d4b76abfa2506753703ed654a90c4982b
```

Make a note of two things: first, the Kubernetes `kubectl` connection information has been written to `/etc/kubernetes/`

admin.conf. This kubeconfig file can be copied to ~/.kube/config, either for root or a normal user on the master node or to a remote machine. This will allow you to control your cluster with the kubectl command.

Second, the last line of the output starting with kubernetes join is a command you can run to join more nodes to the cluster.

After copying the new kubeconfig to somewhere your user can use it, you can validate that the Control Plane has been installed with the kubectl get nodes command:

```
# Show the nodes in the Kubernetes cluster
# Your node name will vary
$ kubectl get nodes
NAME         STATUS   ROLES    AGE     VERSION
elderberry   Ready    master   7m32s   v1.18.2
```

Install a CNI add-on

A CNI add-on handles configuration and cleanup of the pod networks. As mentioned, this exercise uses the Flannel CNI add-on. With the podCIDR value already set, you can just download the Flannel YAML and use kubectl apply to install it into the cluster. This can be done on one line using kubectl apply -f - to take the data from standard input. This will create the ClusterRoles, ServiceAccounts, and DaemonSets (etc.) necessary to manage the pod networking.

Download and apply the Flannel YAML data to the cluster:

```
# Download the Flannel YAML data and apply it
# (output omitted)
$ curl -sSL https://raw.githubusercontent.com/coreos/flannel/
  v0.12.0/Documentation/kube-flannel.yml | kubectl apply -f -
```

Join the compute nodes to the cluster

With the CNI add-on in place, it is now time to add compute nodes to the cluster. Joining the compute nodes is just a matter of running the kubeadm join command provided at the end of the kube init command run to initialize the Control Plane node. For the other Raspberry Pis you want to join your cluster, log into the host, and run the command:

```
# Join a node to the cluster - your tokens and ca-cert-hash
  will vary
$ sudo kubeadm join 192.168.2.114:6443 --token \
  zqqoy7.9oi8dpkfmqkop2p5 --discovery-token-ca-cert-hash \
  sha256:71270ea137214422221319c1bdb9ba6d4b76abfa2506753703 \
  ed654a90c4982b
```

Once you have completed the join process on each node, you should be able to see the new nodes in the output of kubectl get nodes:

```
# Show the nodes in the Kubernetes cluster
# Your node name will vary
$ kubectl get nodes
NAME         STATUS   ROLES    AGE     VERSION
elderberry   Ready    master   7m32s   v1.18.2
gooseberry   Ready    <none>   2m39s   v1.18.2
huckleberry  Ready    <none>   17s     v1.18.2
```

Validate the cluster

At this point, you have a fully working Kubernetes cluster. You can run pods, create deployments and jobs, etc. You can access applications running in the cluster from any of the nodes in the cluster using Services [7]. You can achieve external access with a NodePort service or ingress controllers.

To validate that the cluster is running, create a new namespace, deployment, and service, and check that the pods running in the deployment respond as expected. This deployment uses the quay.io/clcollins/kube-verify:01 image—an Nginx container listening for requests (actually, the same image used in the article *Add nodes to your private cloud using Cloud-init* [8]). You can view the image Containerfile here [9].

Create a namespace named kube-verify for the deployment:

```
# Create a new namespace
$ kubectl create namespace kube-verify
# List the namespaces
$ kubectl get namespaces
NAME              STATUS   AGE
default           Active   63m
kube-node-lease   Active   63m
kube-public       Active   63m
kube-system       Active   63m
kube-verify       Active   19s
```

Now, create a deployment in the new namespace:

```
# Create a new deployment
$ cat <<EOF | kubectl create -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kube-verify
  namespace: kube-verify
  labels:
    app: kube-verify
spec:
  replicas: 3
  selector:
    matchLabels:
      app: kube-verify
  template:
```

```
    metadata:
      labels:
        app: kube-verify
    spec:
      containers:
      - name: nginx
        image: quay.io/clcollins/kube-verify:01
        ports:
        - containerPort: 8080
EOF
deployment.apps/kube-verify created
```

Kubernetes will now start creating the deployment, consisting of three pods, each running the `quay.io/clcollins/kube-verify:01` image. After a minute or so, the new pods should be running, and you can view them with `kubectl get all -n kube-verify` to list all the resources created in the new namespace:

```
# Check the resources that were created by the deployment
$ kubectl get all -n kube-verify
NAME                               READY   STATUS              RESTARTS   AGE
pod/kube-verify-5f976b5474-25p5r   0/1     Running             0          46s
pod/kube-verify-5f976b5474-sc7zd   1/1     Running             0          46s
pod/kube-verify-5f976b5474-tvl7w   1/1     Running             0          46s

NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/kube-verify   3/3     3            3           47s

NAME                                     DESIRED   CURRENT   READY   AGE
replicaset.apps/kube-verify-5f976b5474   3         3         3       47s
```

You can see the new deployment, a replicaset created by the deployment, and three pods created by the replicaset to fulfill the `replicas: 3` request in the deployment. You can see the internals of Kubernetes are working.

Now, create a Service to expose the Nginx "application" (or, in this case, the Welcome page) running in the three pods. This will act as a single endpoint through which you can connect to the pods:

```
# Create a service for the deployment
$ cat <<EOF | kubectl create -f -
apiVersion: v1
kind: Service
metadata:
  name: kube-verify
  namespace: kube-verify
spec:
  selector:
    app: kube-verify
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
EOF
service/kube-verify created
```

With the service created, you can examine it and get the IP address for your new service:

```
# Examine the new service
$ kubectl get -n kube-verify service/kube-verify
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
kube-verify   ClusterIP   10.98.188.200   <none>        80/TCP    30s
```

You can see that the `kube-verify` service has been assigned a ClusterIP (internal to the cluster only) of `10.98.188.200`. This IP is reachable from any of your nodes, but not from outside of the cluster. You can verify the containers inside your deployment are working by connecting to them at this IP:

```
# Use curl to connect to the ClusterIP:
# (output truncated for brevity)
$ curl 10.98.188.200
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
```

Success! Your service is running and Nginx inside the containers is responding to your requests.

At this point, you have a running Kubernetes cluster on your Raspberry Pis with a CNI add-on (Flannel) installed and a test deployment and service running an Nginx webserver. In the large public clouds, Kubernetes has different ingress controllers to interact with different solutions, such as the recently-covered Skipper [10] project. Similarly, private clouds have ingress controllers for interacting with hardware load balancer appliances (like F5 Networks' load balancers) or Nginx and HAProxy controllers for handling traffic coming into the nodes.

In a future article, I will tackle exposing services in the cluster to the outside world by installing your own ingress controller. I will also look at dynamic storage provisioners and StorageClasses for allocating persistent storage for applications, including making use of the NFS server you set up in a previous article, *Turn your Raspberry Pi homelab into a network filesystem* [11], to create on-demand storage for your pods.

Go forth, and Kubernetes

"Kubernetes" (κυβερνήτης) is Greek for pilot—but does that mean the individual who steers a ship as well as the action of guiding the ship? Eh, no. "Kubernan" (κυβερνάω) is Greek for "to pilot" or "to steer," so go forth and Kubernan, and if you see

me out at a conference or something, give me a pass for trying to verb a noun. From another language. That I don't speak.

Disclaimer: As mentioned, I don't read or speak Greek, especially the ancient variety, so I'm choosing to believe something I read on the internet. You know how that goes. Take it with a grain of salt, and give me a little break since I didn't make an "It's all Greek to me" joke. However, just mentioning it, I, therefore, was able to make the joke without actually making it, so I'm either sneaky or clever or both. Or, neither. I didn't claim it was a good joke.

So, go forth and pilot your containers like a pro with your own Kubernetes container service in your private cloud at home! As you become more comfortable, you can modify your Kubernetes cluster to try different options, like the aforementioned ingress controllers and dynamic StorageClasses for persistent volumes.

This continuous learning is at the heart of DevOps [12], and the continuous integration and delivery of new services mirrors the agile methodology, both of which we have embraced as we've learned to deal with the massive scale enabled by the cloud and discovered our traditional practices were unable to keep pace.

Look at that! Technology, policy, philosophy, a tiny bit of Greek, and a terrible meta-joke, all in one article!

Links

[1] https://opensource.com/resources/what-is-kubernetes
[2] https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/
[3] https://opensource.com/article/20/5/disk-image-raspberry-pi
[4] https://en.wikipedia.org/wiki/Cgroups
[5] https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing
[6] https://github.com/coreos/flannel
[7] https://kubernetes.io/docs/concepts/services-networking/service/
[8] https://opensource.com/article/20/5/create-simple-cloud-init-service-your-homelab
[9] https://github.com/clcollins/homelabCloudInit/blob/master/simpleCloudInitService/data/Containerfile
[10] https://opensource.com/article/20/4/http-kubernetes-skipper
[11] https://opensource.com/article/20/5/nfs-raspberry-pi
[12] https://opensource.com/tags/devops

How Cloud-init can be used for your Raspberry Pi homelab

Automate adding new devices and users to your homelab while getting to know a cloud-industry standard.

CLOUD-INIT [1] is a standard—it would not be a stretch to say it is *the* standard—that cloud providers use to provide initialization and configuration data to cloud instances. It is used most often on the first boot of a new instance to automate network setup, account creation, and SSH (secure shell) key installation—anything required to bring a new system online so that it is accessible by the user.

In a previous article, *Modify a disk image to create a Raspberry Pi-based homelab* [2], I showed how to customize the operating system image for single-board computers like the Raspberry Pi to accomplish a similar goal. With Cloud-init, there is no need to add custom data to the image. Once it is enabled in your images, your virtual machines, physical servers, even tiny Raspberry Pis can behave like cloud instances in your own "private cloud at home." New machines can just be plugged in, turned on, and automatically become part of your homelab [3].

To be honest, Cloud-init is not designed with homelabs in mind. As I mentioned, you can easily modify the disk image for a given set of systems to enable SSH access and configure them after the first boot. Cloud-init is designed for large-scale cloud providers that need to accommodate many customers, maintain a small set of images, and provide a mechanism for those customers to access instances without customizing an image for each of them. A homelab with a single administrator does not face the same challenges.

Cloud-init is not without merit in the homelab, though. Education is one of my goals for the private cloud at home project, and setting up Cloud-init for your homelab is a great way to gain experience with a technology used heavily by cloud providers, large and small. Cloud-init is also an alternative to other initial-configuration options. Rather than customizing each image, ISO, etc. for every device in your homelab and face tedious updates when you want to make changes, you can just enable Cloud-init. This reduces technical debt—and is there anything worse than personal technical debt? Finally, using Cloud-init in your homelab allows your private cloud instances to behave the same as any public cloud instances you have or may have in the future—a true hybrid cloud [4].

About Cloud-init

When an instance configured for Cloud-init boots up and the service (actually, four services in systemd implementations to handle dependencies during the boot process) starts, it checks its configuration for a datasource to determine what type of cloud it is running in. Each major cloud provider has a datasource [5] configuration that tells the instance where and how to retrieve configuration information. The instance then uses the datasource information to retrieve configuration information provided by the cloud provider, such as networking information and instance-identification information, and configuration data provided by the customer, such as authorized keys to be copied, user accounts to be created, and many other possible tasks.

After retrieving the data, Cloud-init then configures the instance: setting up networking, copying the authorized keys, etc., and finally completing the boot process. Then it is accessible to the remote user, ready for further configuration with tools like Ansible [6] or Puppet [7] or prepared to receive a workload and begin its assigned tasks.

Configuration data

As mentioned above, the configuration data used by Cloud-init comes from two potential sources: the cloud provider and

the instance user. In a homelab, you fill both roles: providing networking and instance information as the cloud provider and providing configuration information as the user.

The cloud provider metadata file

In your cloud provider role, your homelab datasource will offer your private cloud instances a metadata file. The metadata [8] file contains information such as the instance ID, cloud type, Python version (Cloud-init is written in and uses Python), or a public SSH key to be assigned to the host. The metadata file may also contain networking information if you're not using DHCP (or the other mechanisms Cloud-init supports, such as a config file in the image or kernel parameters).

The user-provided user-data file

The real meat of Cloud-init's value is in the user-data [9] file. Provided by the user to the cloud provider and included in the datasource, the user-data file is what turns an instance from a generic machine into a member of the user's fleet. The user-data file can come in the form of an executable script, working the same as the script would in normal circumstances, or as a cloud-config YAML file, which makes use of Cloud-init's modules [10] to perform configuration tasks.

Datasource

The datasource is a service provided by the cloud provider that offers the metadata and user-data files to the instances. Instance images or ISOs are configured to tell the instance what datasource is being used.

For example, Amazon AWS provides a link-local [11] file that will respond to HTTP requests from an instance with the instance's custom data. Other cloud providers have their own mechanisms, as well. Luckily for the private cloud at home project, there are also NoCloud data sources.

NoCloud [12] datasources allow configuration information to be provided via the kernel command as key-value pairs or as user-data and metadata files provided as mounted ISO filesystems. These are useful for virtual machines, especially paired with automation to create the virtual machines.

There is also a NoCloudNet datasource that behaves similarly to the AWS EC2 datasource, providing an IP address or DNS name from which to retrieve user data and metadata via HTTP. This is most helpful for the physical machines in your homelab, such as Raspberry Pis, NUCs [13], or surplus server equipment. While NoCloud could work, it requires more manual attention—an anti-pattern for cloud instances.

Cloud-init for the homelab

I hope this gives you an idea of what Cloud-init is and how it may be helpful in your homelab. It is an incredible tool that is embraced by major cloud providers, and using it at home can be educational and fun and help you automate adding new physical or virtual servers to your lab. Future articles will detail how to create both simple static and more complex dynamic Cloud-init services and guide you in incorporating them into your private cloud at home.

Links

[1] https://cloudinit.readthedocs.io/
[2] https://opensource.com/article/20/5/disk-image-raspberry-pi
[3] https://opensource.com/article/19/3/home-lab
[4] https://www.redhat.com/en/topics/cloud-computing/what-is-hybrid-cloud
[5] https://cloudinit.readthedocs.io/en/latest/topics/datasources.html
[6] https://www.ansible.com/
[7] https://puppet.com/
[8] https://cloudinit.readthedocs.io/en/latest/topics/instancedata.html#
[9] https://cloudinit.readthedocs.io/en/latest/topics/format.html
[10] https://cloudinit.readthedocs.io/en/latest/topics/modules.html
[11] https://en.wikipedia.org/wiki/Link-local_address
[12] https://cloudinit.readthedocs.io/en/latest/topics/datasources/nocloud.html
[13] https://en.wikipedia.org/wiki/Next_Unit_of_Computing

Add nodes to your private cloud using Cloud-init

Make adding machines to your private cloud at home similar to how the major cloud providers handle it.

CLOUD-INIT [1] IS A WIDELY UTILIZED industry-standard method for initializing cloud instances. Cloud providers use Cloud-init to customize instances with network configuration, instance information, and even user-provided configuration directives. It is also a great tool to use in your "private cloud at home" to add a little automation to the initial setup and configuration of your homelab's virtual and physical machines—and to learn more about how large cloud providers work. For a bit more detail and background, see my previous article on Cloud-init and why it is useful [2].

Boot process for a Linux server running Cloud-init (Chris Collins, CC BY-SA 4.0)

Admittedly, Cloud-init is more useful for a cloud provider provisioning machines for many different clients than for a homelab run by a single sysadmin, and much of what Cloud-init solves might be a little superfluous for a homelab. However, getting it set up and learning how it works is a great way to learn more about this cloud technology, not to mention that it's a great way to configure your devices on first boot.

This tutorial uses Cloud-init's "NoCloud" datasource, which allows Cloud-init to be used outside a traditional cloud provider setting. This article will show you how to install Cloud-init on a client device and set up a container running a web service to respond to the client's requests. You will also learn to investigate what the client is requesting from the web service and modify the web service's container to serve a basic, static Cloud-init service.

Set up Cloud-init on an existing system

Cloud-init probably is most useful on a new system's first boot to query for configuration data and make changes to customize the system as directed. It can be included in a disk image for Raspberry Pis and single-board computers [3] or added to images used to provision virtual machines. For testing, it is easy to install and run Cloud-init on an existing system or to install a new system and then set up Cloud-init.

As a major service used by most cloud providers, Cloud-init is supported on most Linux distributions. For this example, I will be using Fedora 31 Server for the Raspberry Pi, but it can be done the same way on Raspbian, Ubuntu, CentOS, and most other distributions.

Install and enable the cloud-init services

On a system that you want to be a Cloud-init client, install the Cloud-init package. If you're using Fedora:

```
# Install the cloud-init package
dnf install -y cloud-init
```

Cloud-init is actually four different services (at least with systemd), and each is in charge of retrieving config data and performing configuration changes during a different part of the boot process, allowing greater flexibility in what can be done. While it is unlikely you will interact much with these services directly, it is useful to know what they are in the event you need to troubleshoot something. They are:

- cloud-init-local.service
- cloud-init.service
- cloud-config.service
- cloud-final.service

Enable all four services:

```
# Enable the four cloud-init services
systemctl enable cloud-init-local.service
systemctl enable cloud-init.service
systemctl enable cloud-config.service
systemctl enable cloud-final.service
```

Configure the datasource to query

Once the service is enabled, configure the datasource from which the client will query the config data. There are a large number of datasource types [4], and most are configured for specific cloud providers. For your homelab, use the NoCloud datasource, which (as mentioned above) is designed for using Cloud-init without a cloud provider.

NoCloud allows configuration information to be included a number of ways: as key/value pairs in kernel parameters, for using a CD (or virtual CD, in the case of virtual machines) mounted at startup, in a file included on the filesystem, or, as in this example, via HTTP from a specified URL (the "NoCloud Net" option).

The datasource configuration can be provided via the kernel parameter or by setting it in the Cloud-init configuration file, `/etc/cloud/cloud.cfg`. The configuration file works very well for setting up Cloud-init with customized disk images or for testing on existing hosts.

Cloud-init will also merge configuration data from any `*.cfg` files found in `/etc/cloud/cloud.cfg.d/`, so to keep things cleaner, configure the datasource in `/etc/cloud/cloud.cfg.d/10_datasource.cfg`. Cloud-init can be told to read from an HTTP datasource with the seedfrom key by using the syntax:

```
seedfrom: http://ip_address:port/
```

The IP address and port are the web service you will create later in this article. I used the IP of my laptop and port 8080; this can also be a DNS name.

Create the `/etc/cloud/cloud.cfg.d/10_datasource.cfg` file:

```
# Add the datasource:
# /etc/cloud/cloud.cfg.d/10_datasource.cfg

# NOTE THE TRAILING SLASH HERE!
datasource:
  NoCloud:
    seedfrom: http://ip_address:port/
```

That's it for the client setup. Now, when the client is rebooted, it will attempt to retrieve configuration data from the URL you entered in the seedfrom key and make any configuration changes that are necessary.

The next step is to set up a webserver to listen for client requests, so you can figure out what needs to be served.

Set up a webserver to investigate client requests

You can create and run a webserver quickly with Podman [5] or other container orchestration tools (like Docker or Kubernetes). This example uses Podman, but the same commands work with Docker.

To get started, use the Fedora:31 container image and create a Containerfile (for Docker, this would be a Dockerfile) that installs and configures Nginx. From that Containerfile, you can build a custom image and run it on the host you want to act as the Cloud-init service.

Create a Containerfile with the following contents:

```
FROM fedora:31

ENV NGINX_CONF_DIR "/etc/nginx/default.d"
ENV NGINX_LOG_DIR "/var/log/nginx"
ENV NGINX_CONF "/etc/nginx/nginx.conf"
ENV WWW_DIR "/usr/share/nginx/html"

# Install Nginx and clear the yum cache
RUN dnf install -y nginx \
      && dnf clean all \
      && rm -rf /var/cache/yum

# forward request and error logs to docker log collector
RUN ln -sf /dev/stdout ${NGINX_LOG_DIR}/access.log \
    && ln -sf /dev/stderr ${NGINX_LOG_DIR}/error.log

# Listen on port 8080, so root privileges are not required for
  podman
RUN sed -i -E 's/(listen)([[:space:]]*)
  (\[\:\:\]\:)?80;$/\1\2\38080 default_server;/' $NGINX_CONF
EXPOSE 8080

# Allow Nginx PID to be managed by non-root user
RUN sed -i '/user nginx;/d' $NGINX_CONF
RUN sed -i 's/pid \/run\/nginx.pid;/pid
  \/tmp\/nginx.pid;/' $NGINX_CONF

# Run as an unprivileged user
USER 1001

CMD ["nginx", "-g", "daemon off;"]
```

Note: The example Containerfile and other files used in this example can be found in this project's GitHub repository [6].

The most important part of the Containerfile above is the section that changes how the logs are stored (writing to STDOUT rather than a file), so you can see requests coming into the server in the container logs. A few other changes enable you to run the container with Podman without root privileges and to run processes in the container without root, as well.

This first pass at the webserver does not serve any Cloud-init data; just use this to see what the Cloud-init client is requesting from it.

With the Containerfile created, use Podman to build and run a webserver image:

```
# Build the container image
$ podman build -f Containerfile -t cloud-init:01 .

# Create a container from the new image, and run it
# It will listen on port 8080
$ podman run --rm -p 8080:8080 -it cloud-init:01
```

This will run the container, leaving your terminal attached and with a pseudo-TTY. It will appear that nothing is happening at first, but requests to port 8080 of the host machine will be routed to the Nginx server inside the container, and a log message will appear in the terminal window. This can be tested with a curl command from the host machine:

```
# Use curl to send an HTTP request to the Nginx container
$ curl http://localhost:8080
```

After running that curl command, you should see a log message similar to this in the terminal window:

```
127.0.0.1 - - [09/May/2020:19:25:10 +0000] "GET / HTTP/1.1"
  200 5564 "-" "curl/7.66.0" "-"
```

Now comes the fun part: reboot the Cloud-init client and watch the Nginx logs to see what Cloud-init requests from the webserver when the client boots up.

As the client finishes its boot process, you should see log messages similar to:

```
2020/05/09 22:44:28 [error] 2#0: *4 open()
  "/usr/share/nginx/html/meta-data" failed (2: No such file or
  directory), client: 127.0.0.1, server: _, request:
  "GET /meta-data HTTP/1.1", host: "instance-data:8080"
127.0.0.1 - - [09/May/2020:22:44:28 +0000] "GET /meta-data
  HTTP/1.1" 404 3650 "-" "Cloud-Init/17.1" "-"
```

Note: Use Ctrl+C to stop the running container.

You can see the request is for the /meta-data path, i.e., `http://ip_address_of_the_webserver:8080/meta-data`. This is just a GET request—Cloud-init is not POSTing (sending) any data to the webserver. It is just blindly requesting the files from the datasource URL, so it is up to the datasource to identify what the host is asking. This simple example is just sending generic data to any client, but a larger homelab will need a more sophisticated service.

Here, Cloud-init is requesting the instance metadata [7] information. This file can include a lot of information about the instance itself, such as the instance ID, the hostname to assign the instance, the cloud ID—even networking information.

Create a basic metadata file with an instance ID and a hostname for the host, and try serving that to the Cloud-init client.

First, create a metadata file that can be copied into the container image:

```
instance-id: iid-local01
local-hostname: raspberry
hostname: raspberry
```

The instance ID can be anything. However, if you change the instance ID after Cloud-init runs and the file is served to the client, it will trigger Cloud-init to run again. You can use this mechanism to update instance configurations, but you should be aware that it works that way.

The local-hostname and hostname keys are just that; they set the hostname information for the client when Cloud-init runs.

Add the following line to the Containerfile to copy the metadata file into the new image:

```
# Copy the meta-data file into the image for Nginx to serve it
COPY meta-data ${WWW_DIR}/meta-data
```

Now, rebuild the image (use a new tag for easy troubleshooting) with the metadata file, and create and run a new container with Podman:

```
# Build a new image named cloud-init:02
podman build -f Containerfile -t cloud-init:02 .

# Run a new container with this new meta-data file
podman run --rm -p 8080:8080 -it cloud-init:02
```

With the new container running, reboot your Cloud-init client and watch the Nginx logs again:

```
127.0.0.1 - - [09/May/2020:22:54:32 +0000] "GET
  /meta-data HTTP/1.1" 200 63 "-" "Cloud-Init/17.1" "-"
2020/05/09 22:54:32 [error] 2#0: *2 open()
  "/usr/share/nginx/html/user-data" failed (2: No such file or
  directory), client: 127.0.0.1, server: _, request:
  "GET /user-data HTTP/1.1", host: "instance-data:8080"
127.0.0.1 - - [09/May/2020:22:54:32 +0000]
  "GET /user-data HTTP/1.1" 404 3650 "-" "Cloud-Init/17.1" "-"
```

You see that this time the /meta-data path was served to the client. Success!

However, the client is looking for a second file at the /user-data path. This file contains configuration data provided by the instance owner, as opposed to data from the cloud provider. For a homelab, both of these are you.

There are a large number of user-data modules [8] you can use to configure your instance. For this example, just use the write_files module to create some test files on the client and verify that Cloud-init is working.

Create a user-data file with the following content:

```
#cloud-config

# Create two files with example content using the write_files module
write_files:
 - content: |
    "Does cloud-init work?"
   owner: root:root
   permissions: '0644'
   path: /srv/foo
 - content: |
   "IT SURE DOES!"
   owner: root:root
   permissions: '0644'
   path: /srv/bar
```

In addition to a YAML file using the user-data modules provided by Cloud-init, you could also make this an executable script for Cloud-init to run.

After creating the user-data file, add the following line to the Containerfile to copy it into the image when the image is rebuilt:

```
# Copy the user-data file into the container image
COPY user-data ${WWW_DIR}/user-data
```

Rebuild the image and create and run a new container, this time with the user-data information:

```
# Build a new image named cloud-init:03
podman build -f Containerfile -t cloud-init:03 .

# Run a new container with this new user-data file
podman run --rm -p 8080:8080 -it cloud-init:03
```

Now, reboot your Cloud-init client, and watch the Nginx logs on the webserver:

```
127.0.0.1 - - [09/May/2020:23:01:51 +0000]
  "GET /meta-data HTTP/1.1" 200 63 "-" "Cloud-Init/17.1" "-"
127.0.0.1 - - [09/May/2020:23:01:51 +0000]
  "GET /user-data HTTP/1.1" 200 298 "-" "Cloud-Init/17.1" "-
```

Success! This time both the metadata and user-data files were served to the Cloud-init client.

Validate that Cloud-init ran

From the logs above, you know that Cloud-init ran on the client host and requested the metadata and user-data files, but did it do anything with them? You can validate that Cloud-init wrote the files you added in the user-data file, in the write_files section.

On your Cloud-init client, check the contents of the `/srv/foo` and `/srv/bar` files:

```
# cd /srv/ && ls
bar foo
# cat foo
"Does cloud-init work?"
# cat bar
"IT SURE DOES!"
```

Success! The files were written and have the content you expect.

As mentioned above, there are plenty of other modules that can be used to configure the host. For example, the user-data file can be configured to add packages with apt, copy SSH authorized_keys, create users and groups, configure and run configuration-management tools, and many other things. I use it in my private cloud at home to copy my authorized_keys, create a local user and group, and set up sudo permissions.

What you can do next

Cloud-init is useful in a homelab, especially a lab focusing on cloud technologies. The simple service demonstrated in this article may not be super useful for a homelab, but now that you know how Cloud-init works, you can move on to creating a dynamic service that can configure each host with custom data, making a private cloud at home more similar to the services provided by the major cloud providers.

With a slightly more complicated datasource, adding new physical (or virtual) machines to your private cloud at home can be as simple as plugging them in and turning them on.

Links

[1] https://cloudinit.readthedocs.io/

[2] https://opensource.com/article/20/5/cloud-init-raspberry-pi-homelab

[3] https://opensource.com/article/20/5/disk-image-raspberry-pi

[4] https://cloudinit.readthedocs.io/en/latest/topics/datasources.html

[5] https://podman.io/

[6] https://github.com/clcollins/homelabCloudInit/tree/master/simpleCloudInitService/data

[7] https://cloudinit.readthedocs.io/en/latest/topics/instancedata.html#what-is-instance-data

[8] https://cloudinit.readthedocs.io/en/latest/topics/modules.html

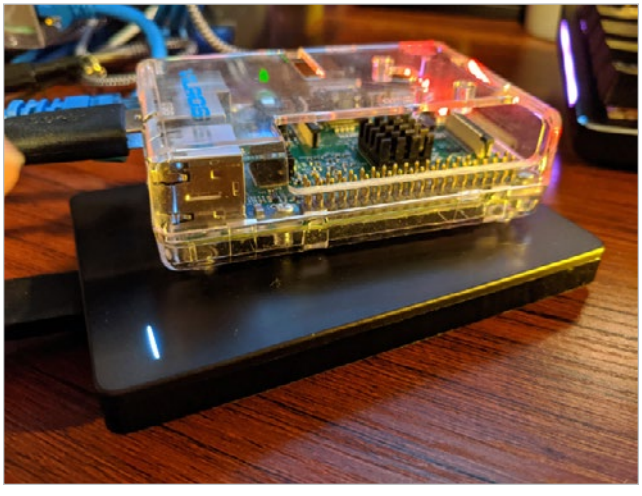
Turn your Raspberry Pi homelab into a network filesystem

Add shared filesystems to your homelab with an NFS server.

A SHARED FILESYSTEM is a great way to add versatility and functionality to a homelab. Having a centralized filesystem shared to the clients in the lab makes organizing data, doing backups, and sharing data considerably easier. This is especially useful for web applications load-balanced across multiple servers and for persistent volumes used by Kubernetes [1], as it allows pods to be spun up with persistent data on any number of nodes.

Whether your homelab is made up of ordinary computers, surplus enterprise servers, or Raspberry Pis or other single-board computers (SBCs), a shared filesystem is a useful asset, and a network filesystem (NFS) server is a great way to create one.

I have written before about setting up a "private cloud at home," [2] a homelab made up of Raspberry Pis or other SBCs and maybe some other consumer hardware or a desktop PC. An NFS server is an ideal way of sharing data between these components. Since most SBCs' operating systems (OSes) run off an SD card, there are some challenges. SD cards suffer from increased failures, especially when used as the OS disk for a computer, and they are not made to be constantly read from and written to. What you really need is a real hard drive: they are generally cheaper per gigabyte than SD cards, especially for larger disks, and they are less likely to sustain failures. Raspberry Pi 4's now come with USB 3.0 ports, and USB 3.0 hard drives are ubiquitous and affordable. It's a perfect match. For this project, I will use a 2TB USB 3.0 external hard drive plugged into a Raspberry Pi 4 running an NFS server.



(Chris Collins, CC BY-SA 4.0)

Install the NFS server software

I am running Fedora Server on a Raspberry Pi, but this project can be done with other distributions as well. To run an NFS server on Fedora, you need the nfs-utils package, and luckily it is already installed (at least in Fedora 31). You also need the rpcbind package if you are planning to run NFSv3 services, but it is not strictly required for NFSv4.

If these packages are not already on your system, install them with the **dnf** command:

```
# Intall nfs-utils and rpcbind
$ sudo dnf install nfs-utils rpcbind
```

Raspbian is another popular OS used with Raspberry Pis, and the setup is almost exactly the same. The package names differ, but that is about the only major difference. To install an NFS server on a system running Raspbian, you need the following packages:

- **nfs-common**: These files are common to NFS servers and clients
- **nfs-kernel-server**: The main NFS server software package

Raspbian uses **apt-get** for package management (not dnf, as Fedora does), so use that to install the packages:

```
# For a Raspbian system, use apt-get to install the NFS packages
$ sudo apt-get install nfs-common nfs-kernel-server
```

Prepare a USB hard drive as storage

As I mentioned above, a USB hard drive is a good choice for providing storage for Raspberry Pis or other SBCs, especially because the SD card used for the OS disk image is not ideal. For your private cloud at home, you can use cheap USB 3.0 hard drives for large-scale storage. Plug the disk in and use **fdisk** to find out the device ID assigned to it, so you can work with it.

```
# Find your disk using fdisk
# Unrelated disk content omitted
$ sudo fdisk -l

Disk /dev/sda: 1.84 TiB, 2000398933504 bytes, 3907029167 sectors
Disk model: BUP Slim BK
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xe3345ae9

Device     Boot Start        End    Sectors  Size Id Type
/dev/sda1        2048 3907028991 3907026944  1.8T 83 Linux
```

For clarity, in the example output above, I omitted all the disks except the one I'm interested in. You can see the USB disk I want to use was assigned the device **/dev/sda**, and you can see some information about the model (**Disk model: BUP Slim BK**), which helps me identify the correct disk. The disk already has a partition, and its size confirms it is the disk I am looking for.

Note: Make sure to identify the correct disk and partition for your device. It may be different than the example above.

Each partition created on a drive gets a special universally unique identifier (UUID). The computer uses the UUID to make sure it is mounting the correct partition to the correct location using the **/etc/fstab** config file. You can retrieve the UUID of the partition using the **blkid** command:

```
# Get the block device attributes for the partition
# Make sure to use the partition that applies in your case. It
  may differ.
$ sudo blkid /dev/sda1

/dev/sda1: LABEL="backup" UUID="bd44867c-447c-4f85-8dbf-
           dc6b9bc65c91" TYPE="xfs" PARTUUID="e3345ae9-01"
```

In this case, the UUID of **/dev/sda1** is **bd44867c-447c-4f85-8dbf-dc6b9bc65c91**. Yours will be different, so make a note of it.

Configure the Raspberry Pi to mount this disk on startup, then mount it

Now that you have identified the disk and partition you want to use, you need to tell the computer how to mount it, to do so whenever it boots up, and to go ahead and mount it now. Because this is a USB disk and might be unplugged, you will also configure the Raspberry Pi to not wait on boot if the disk is not plugged in or is otherwise unavailable.

In Linux, this is done by adding the partition to the **/etc/fstab** configuration file, including where you want it to be mounted and some arguments to tell the computer how to treat it. This example will mount the partition to **/srv/nfs**, so start by creating that path:

```
# Create the mountpoint for the disk partition
$ sudo mkdir -p /srv/nfs
```

Next, modify the **/etc/fstab** file using the following syntax format:

```
<disk id>   <mountpoint>   <filesystem type>   <options>   <fs_freq> <fs_passno>
```

Use the UUID you identified earlier for the disk ID. As I mentioned in the prior step, the mountpoint is **/srv/nfs**. For the filesystem type, it is usually best to select the actual filesystem, but since this will be a USB disk, use **auto**.

For the options values, use **nosuid,nodev,nofail**.

An aside about man pages:

That said, there are a *lot* of possible options, and the manual (man) pages are the best way to see what they are. Investigating the man page for for fstab is a good place to start:

```
# Open the man page for fstab
$ man fstab
```

This opens the manual/documentation associated with the fstab command. In the man page, each of the options is broken down to show what it does and the common selections. For example, **The fourth field (fs_mntopts)** gives some basic information about the options that work in that field and directs you to **man (8) mount** for more in-depth description of the mount options. That makes sense, as the **/etc/fstab** file, in essence, tells the computer how to automate mounting disks, in the same way you would manually use the mount command.

You can get more information about the options you will use from mount's man page. The numeral 8, in parentheses, indicates the man page section. In this case, section 8 is for *System Administration tools and Daemons*.

Helpfully, you can get a list of the standard sections from the man page for **man**.

Back to mountng the disk, take a look at **man (8) mount**:

```
# Open Section 8 of the man pages for mount
$ man (8) mount
```

In this man page, you can examine what the options listed above do:

- **nosuid**: Do not honor the suid/guid bit. Do not allow any files that might be on the USB disk to be executed as root. This is a good security practice.
- **nodev**: Do not interpret characters or block special devices on the file system; i.e., do not honor any device nodes

that might be on the USB disk. Another good security practice.
- **nofail**: Do not log any errors if the device does not exist. This is a USB disk and might not be plugged in, so it will be ignored if that is the case.

Returning to the line you are adding to the **/etc/fstab** file, there are two final options: **fs_freq** and **fs_passno**. Their values are related to somewhat legacy options, and *most* modern systems just use a **0** for both, especially for filesystems on USB disks. The fs_freq value relates to the dump command and making dumps of the filesystem. The fs_passno value defines which filesystems to **fsck** on boot and their order. If it's set, usually the root partition would be **1** and any other filesystems would be **2**. Set the value to **0** to skip using **fsck** on this partition.

In your preferred editor, open the **/etc/fstab** file and add the entry for the partition on the USB disk, replacing the values here with those gathered in the previous steps.

```
# With sudo, or as root, add the partition info to the /etc/
  fstab file
UUID="bd44867c-447c-4f85-8dbf-dc6b9bc65c91"    /srv/nfs
     auto    nosuid,nodev,nofail,noatime 0 0
```

Enable and start the NFS server

With the packages installed and the partition added to your **/etc/fstab** file, you can now go ahead and start the NFS server. On a Fedora system, you need to enable and start two services: **rpcbind** and **nfs-server**. Use the **systemctl** command to accomplish this:

```
# Start NFS server and rpcbind
$ sudo systemctl enable rpcbind.service
$ sudo systemctl enable nfs-server.service
$ sudo systemctl start rpcbind.service
$ sudo systemctl start nfs-server.service
```

On Raspbian or other Debian-based distributions, you just need to enable and start the **nfs-kernel-server** service using the **systemctl** command the same way as above.

RPCBind

The rpcbind utility is used to map remote procedure call (RPC) services to ports on which they listen. According to the rpcbind man page:

> "When an RPC service is started, it tells rpcbind the address at which it is listening, and the RPC program numbers it is prepared to serve. When a client wishes to make an RPC call to a given program number, it first contacts rpcbind on the server machine to determine the address where RPC requests should be sent."

In the case of an NFS server, rpcbind maps the protocol number for NFS to the port on which the NFS server is listening. However, NFSv4 does not require the use of rpcbind. If you use only NFSv4 (by removing versions two and three from the configuration), rpcbind is not required. I've included it here for backward compatibility with NFSv3.

Export the mounted filesystem

The NFS server decides which filesystems are shared with (exported to) which remote clients based on another configuration file, **/etc/exports**. This file is just a map of host internet protocol (IP) addresses (or subnets) to the filesystems to be shared and some options (read-only or read-write, root squash, etc.). The format of the file is:

```
<directory>     <host or hosts>(options)
```

In this example, you will export the partition mounted to **/srv/nfs**. This is the "directory" piece.

The second part, the host or hosts, includes the hosts you want to export this partition to. These can be specified as a single host with a fully qualified domain name or hostname, the IP address of the host, a number of hosts using wildcard characters to match domains (e.g., *.example.org), IP networks (e.g., classless inter-domain routing, or CIDR, notation), or netgroups.

The third piece includes options to apply to the export:

- **ro/rw**: Export the filesystem as read only or read write
- **wdelay**: Delay writes to the disk if another write is imminent to improve performance (this is *probably* not as useful with a solid-state USB disk, if that is what you are using)
- **root_squash**: Prevent any root users on the client from having root access on the host, and set the root UID to **nfsnobody** as a security precaution

Test exporting the partition you have mouted at **/srv/nfs** to a single client—for example, a laptop. Identify your client's IP address (my laptop's is **192.168.2.64**, but yours will likely be different). You could share it to a large subnet, but for testing, limit it to the single IP address. The CIDR notation for just this IP is **192.168.2.64/32**; a **/32** subnet is just a single IP.

Using your preferred editor, edit the /etc/exports file with your directory, host CIDR, and the rw and root_squash options:

```
# Edit your /etc/exports file like so, substituting the
  information from your systems
/srv/nfs    192.168.2.64/32(rw,root_squash)
```

Note: If you copied the **/etc/exports** file from another location or otherwise overwrote the original with a copy, you may need to restore the SELinux context for the file. You can do this with the **restorecon** command:

```
# Restore the SELinux context of the /etc/exports file
$ sudo restorecon /etc/exports
```

Once this is done, restart the NFS server to pick up the changes to the **/etc/exports** file:

```
# Restart the nfs server
$ sudo systemctl restart nfs-server.service
```

Open the firewall for the NFS service

Some systems, by default, do not run a firewall service [3]. Raspbian, for example, defaults to open iptables rules, with ports opened by different services immediately available from outside the machine. Fedora server, by contrast, runs the firewalld service by default, so you must open the port for the NFS server (and rpcbind, if you will be using NFSv3). You can do this with the **firewall-cmd** command.

Check the zones used by firewalld and get the default zone. For Fedora Server, this will be the FedoraServer zone:

```
# List the zones
# Output omitted for brevity
$ sudo firewall-cmd --list-all-zones

# Retrieve just the default zone info
# Make a note of the default zone
$ sudo firewall-cmd --get-default-zone

# Permanently add the nfs service to the list of allowed ports
$ sudo firewall-cmd --add-service=nfs --permanent

# For NFSv3, we need to add a few more ports, nfsv3,
  rpc-mountd, rpc-bind
$ sudo firewall-cmd --add-service=(nfs3,mountd,rpc-bind)

# Check the services for the zone, substituting the default
  zone in use by your system
$ sudo firewall-cmd --list-services --zone=FedoraServer

# If all looks good, reload firewalld
$ sudo firewall-cmd --reload
```

And with that, you have successfully configured the NFS server with your mounted USB disk partition and exported it to your test system for sharing. Now you can test mounting it on the system you added to the exports list.

Test the NFS exports

First, from the NFS server, create a file to read in the **/srv/nfs** directory:

```
# Create a test file to share
echo "Can you see this?" >> /srv/nfs/nfs_test
```

Now, on the client system you added to the exports list, first make sure the NFS client packages are installed. On Fedora systems, this is the **nfs-utils** package and can be installed with **dnf**. Raspbian systems have the **libnfs-utils** package that can be installed with **apt-get**.

Install the NFS client packages:

```
# Install the nfs-utils package with dnf
$ sudo dnf install nfs-utils
```

Once the client package is installed, you can test out the NFS export. Again on the client, use the mount command with the IP of the NFS server and the path to the export, and mount it to a location on the client, which for this test is the **/mnt** directory. In this example, my NFS server's IP is **192.168.2.109**, but yours will likely be different:

```
# Mount the export from the NFS server to the client host
# Make sure to substitute the information for your own hosts
$ sudo mount 192.168.2.109:/srv/nfs /mnt

# See if the nfs_test file is visible:
$ cat /mnt/nfs_test
Can you see this?
```

Success! You now have a working NFS server for your homelab, ready to share files with multiple hosts, allow multi-read/write access, and provide centralized storage and backups for your data. There are many options for shared storage for homelabs, but NFS is venerable, efficient, and a great option to add to your "private cloud at home" homelab. Future articles in this series will expand on how to automatically mount NFS shares on clients and how to use NFS as a storage class for Kubernetes Persistent Volumes.

Links

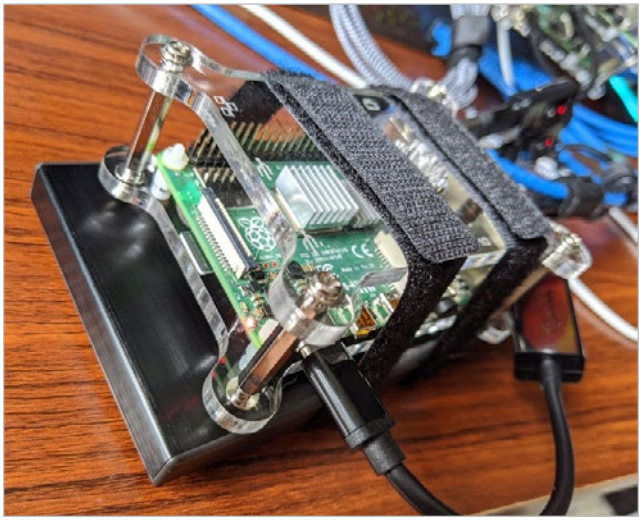
[1] https://opensource.com/resources/what-is-kubernetes
[2] https://opensource.com/article/20/5/disk-image-raspberry-pi
[3] https://opensource.com/article/18/9/linux-iptables-firewalld

Provision Kubernetes NFS clients on a Raspberry Pi homelab

Create dynamic persistent volumes on a Raspberry Pi Kubernetes cluster with the NFS-client provisioner.

EPHEMERAL CONTAINERS are useful, but sometimes data needs to persist between containers or be shared among multiple containers. The solution is mounting external volumes inside the containers, and this is done in Kubernetes with persistent volumes. In large, public cloud deployments, Kubernetes has integrations with the cloud providers' block-storage backends, allowing developers to create claims for volumes to use with their deployments, and Kubernetes works with the cloud provider to create a volume and mount it in the developers' pods.

You can replicate this same behavior in a "private cloud at home" Kubernetes cluster using the network filesystem (NFS)-client provisioner from the Kubernetes Incubator [1] external storage project.



Chris Collins, CC BY-SA 4.0

In a previous article, I explained how to set up an NFS server with a Raspberry Pi [2]. You can use this NFS server to back the storage provided by the NFS-client provisioner. The provisioner runs a container that mounts an NFS export from your NFS server and carves it up into "volumes" when a persistent volume claim is created, requesting volumes for a pod. Kubernetes supports NFS volume types natively [3] and handles mounting the volumes inside the containers when a pod starts.

The NFS-client provisioner gives the benefit of dynamically provisioned volumes and makes a homelab cluster behave similarly to a Kubernetes cluster in the cloud. In addition, because these are NFS volumes, they can support multi-read/multi-write operations, so multiple pods can have the volumes mounted at the same time. This is useful for load-balanced services like a webserver, where multiple pods can be spread across multiple nodes to handle traffic and provide greater availability. NFS volumes can also be created that support only read/write-once operations. This is better for database or other instances, where the software writing to the volume does not handle multiple write operations nicely.

Persistent volumes, persistent volume claims, and storage classes

Persistent volumes [4] are volumes backed by non-ephemeral storage that can be mounted as a volume inside a container. Data written into that volume persists across container restarts and can be mounted into new pods when they replace old ones. In the case of NFS volumes, the volumes can be mounted into multiple containers at the same time, allowing the data to be shared.

Developers can request a persistent volume from Kubernetes with a **persistent volume claim** (PVC) [5]. This is exactly what it sounds like: a request for a volume that matches certain conditions, such as access mode or size, and that can be claimed and used for a project. Persistent volumes request specific volume types using storage classes.

Storage classes [6] describe types of volumes that can be claimed, allowing developers to choose volume types that will work best with what they need. Administrators can create multiple types of storage classes to meet specific needs, like access mode or size (as mentioned above) or even speed/IOPS classes or different backend technologies. Storage classes can also specify a particular provisioner or software in charge of creating and managing the volumes.

Cluster administrators can maintain a large pool of pre-provisioned volumes created manually to satisfy the storage classes, but that requires hands-on work and does not scale well. With a dynamic volume provisioner, software can handle the creation of volumes on-demand when a new claim is created. By default, the NFS-client provisioner has a single storage class, and PVCs that request volumes from this storage class are fulfilled by the provisioner.

The NFS-client provisioner

The NFS-client provisioner [7] is part of the Kubernetes Incubator project. In a Kubernetes cluster, this provisioner runs in a container that mounts an NFS export from an existing NFS server—it does not run an NFS server itself. With the container, it listens for PVCs that match its storage class, creates directories within the NFS export, and reports each directory to Kubernetes as a persistent volume. Kubernetes can then mount the volume into a container that uses the volumes from that PVC.

Installation of the NFS-client provisioner can be done with a Helm chart, as described in the project's README [8], but both for educational reasons and because this tutorial is about running a Kubernetes cluster on Raspberry Pis and needs a few adjustments, you will install it manually on a private-cloud-at-home cluster.

Prerequisites

There are two prerequisites before you can use the NFS-client provisioner:

1. Find the details of the NFS server (the IP address and path to the export)
2. Install the NFS libraries on each of the Kubernetes nodes that might run pods that need the NFS volumes

If you installed your own NFS server, such as the one in *Turn your Raspberry Pi homelab into a network filesystem* [9], you should already know the server's IP address and the path to its exported filesystem. Make sure the export list includes all the Kubernetes nodes that might run pods with the NFS volumes.

You also need to install the NFS libraries on each of these nodes, so they will be able to mount the NFS volumes. On Ubuntu, Raspbian, or other Debian-based operating systems, you can install the `nfs-common` package using `apt`. For Red Hat-based distributions, install the `nfs-utils` package.

With the prerequisites out of the way, you can move on to installing the NFS-client provisioner onto the Kubernetes cluster.

Installation

The NFS-client provisioner is deployed using standard Kubernetes objects. You can find these in the Kubernetes Incubator external storage project within the `nfs-client` directory. To get started, clone the `https://github.com/kubernetes-incubator/external-storage` repository:

```
# Clone the external-storage repo
# (output omitted)
$ git clone https://github.com/kubernetes-incubator/external-
  storage.git
```

The specific pieces needed to install the NFS-client provisioner are in nfs-client/deploy:

```
$ cd external-storage/nfs-client/deploy
$ ls
class.yaml           deployment.yaml  rbac.yaml        test-pod.yaml
deployment-arm.yaml  objects          test-claim.yaml
```

Note that the `objects` directory contains everything from its parent directory, just broken out to a single file per object.

Before doing anything else, you must create the appropriate role-based access control (RBAC) permissions to allow the NFS-client provisioner service account to mount volumes, create PVCs, etc. They can be created on the cluster with the `kubectl create` command.

If you plan to deploy the provisioner in a namespace other than the default namespace, make sure to change the namespace in the RBAC and deployment files first:

```
# Create the RBAC permissions needed by the provisioner
kubectl create -f rbac.yaml
serviceaccount/nfs-client-provisioner created
clusterrole.rbac.authorization.k8s.io/nfs-client-provisioner-
  runner created
clusterrolebinding.rbac.authorization.k8s.io/run-nfs-client-
  provisioner created
role.rbac.authorization.k8s.io/leader-locking-nfs-client-
  provisioner created
rolebinding.rbac.authorization.k8s.io/leader-locking-nfs-
  client-provisioner created
```

Next, create a deployment for the NFS-client provisioner pod. If you created your Kubernetes cluster by following the *Build a Kubernetes cluster with the Raspberry Pi* [10] instructions or you created your own on an ARM-based system, you will need to modify and deploy using the deployment-arm.yaml file. If you are using an x86_64-based system, use the `deployment.yaml` file.

Edit the file with your editor of choice. You need to change four things. First, set the three environment variables from the `.spec.template.containers.env` list:

- Change the value for `PROVISIONER_NAME` to `nfs-storage` (optional; this just makes it a little more human-friendly).
- Change the `NFS_SERVER` value to the IP address of your NFS server.
- Change the `NFS_PATH` value to the path of your NFS export.

Finally, under `.spec.template.spec.volumes.nfs`, change the server and path values to the same ones you set for the `NFS_SERVER` and `NFS_PATH`, respectively.

For example, in an NFS server and export path of `192.168.2.123:/srv`, the `deployment-arm.yaml` file would look like this:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nfs-client-provisioner
  labels:
    app: nfs-client-provisioner
  namespace: default
spec:
  replicas: 1
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: nfs-client-provisioner
  template:
    metadata:
      labels:
        app: nfs-client-provisioner
    spec:
      serviceAccountName: nfs-client-provisioner
      containers:
        - name: nfs-client-provisioner
          image: quay.io/external_storage/nfs-client-
                 provisioner-arm:latest
          volumeMounts:
            - name: nfs-client-root
              mountPath: /persistentvolumes
          env:
            - name: PROVISIONER_NAME
              value: nfs-storage
            - name: NFS_SERVER
              value: 192.168.2.123
            - name: NFS_PATH
              value: /srv
      volumes:
        - name: nfs-client-root
          nfs:
            server: 192.168.2.123
            path: /srv
```

Once the `deployment-arm.yaml` file has been modified, create the deployment with the `kubectl create` command:

```
# Create the deployment
$ kubectl create -f deployment-arm.yaml
deployment.apps/nfs-client-provisioner created

# Check that the deployment created the provisioner pod correctly
$ kubectl get po
NAME                                     READY   STATUS    RESTARTS   AGE
nfs-client-provisioner-6ddfb9bb6d-x4zwt  1/1     Running   0          54s
```

If everything worked correctly, the NFS-client provisioner is now running in your cluster. If the pod has the status `ContainerCreating`, and it does not eventually change to Running, check for any relevant events using the `kubectl get events` command. Make sure that the user "nobody" has write permissions on the export directory on the NFS server. If there are no issues, move on to creating the storage class.

The `class.yaml` file needs to be modified to set the `provisioner` value to `nfs-storage` or whatever you set for the `PROVISIONER_NAME` value in the `deployment-arm.yaml`. This tells Kubernetes which provisioner needs to be used to fulfill PVCs for this storage class. Assuming you choose `nfs-storage`, the `class.yaml` file should look like this:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: managed-nfs-storage
provisioner: nfs-storage # or choose another name, must match
                         deployment's env PROVISIONER_NAME'
parameters:
  archiveOnDelete: "false"
```

Create the storage class with the `kubectl create` command:

```
# Create the storage class
$ kubectl create -f class.yaml
storageclass.storage.k8s.io/managed-nfs-storage created

# Verify the storage class was created
$ kubectl get storageClass
NAME                  PROVISIONER   RECLAIMPOLICY   \
                       VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION   AGE
managed-nfs-storage   nfs-storage   Delete      Immediate       false
```

Success! The NFS-client provisioner is now installed in the Kubernetes cluster and prepared to dynamically allocate persistent storage volumes for your pods in response to persistent volumes claims against the `managed-nfs-storage` storage class.

Test your new volume provisioner

With the provisioner installed and configured, you can test it out by creating a PVC that requests a persistent volume and a pod to mount the volume. Luckily, test objects are provided with the files used for the deployment: `test-claim.yaml` and `test-pod.yaml`.

Before creating a PVC and a pod, take a look at what is already there. Unless you have already created some, there should not be any persistent volumes nor PVCs:

```
# Look for existing persistent volumes
$ kubectl get persistentvolumes
No resources found in default namespace.
```

```
# Look for existing persistent volume claims
$ kubectl get persistentvolumeclaims
No resources found in default namespace.
```

Now, create a new PVC from the `test-claim.yaml` file:

```
# Create a test PVC
$ kubectl create -f test-claim.yaml
persistentvolumeclaim/test-claim created

$ kubectl get persistentvolumeclaims
NAME         STATUS   VOLUME
CAPACITY   ACCESS MODES   STORAGECLASS          AGE
test-claim   Bound    pvc-bdf41489-abe4-4508-8adc-74e80f70c626  \
               1Mi        RWX            managed-nfs-storage   7s
```

From the output above, you can see a PVC named test-claim was created and bound. This claim is tied to a volume named `pvc-bdf41489-abe4-4508-8adc-74e80f70c626` that was created automatically by the NFS-client provisioner. Also, note the `STORAGECLASS` is called `managed-nfs-storage`—the name of the storage class you created.

Do not worry about the `CAPACITY` value. The NFS-client provisioner does not have a way to enforce storage quota; NFS does not have that feature. The `1Mi` is a placeholder.

Now that the PVC and its associated persistent volume have been created by the NFS-client provisioner, log into your NFS server and check out what happened on that end:

```
# From the NFS host, with the directory used for the NFS export
$ ls -la
total 4
drwxr-xr-x  3 nobody  root      73 May 25 18:46 .
drwxr-xr-x 21 root   root    4096 May 25 17:37 ..
drwxrwxrwx  2 nobody nogroup    6 May 25 18:45 default-test-
  claim-pvc-bdf41489-abe4-4508-8adc-74e80f70c626
```

A new directory has been created that follows the naming convention `namespace-pvc_name-pv_name`. This directory is initially empty. You can create a test pod to mount this directory via NFS and write to it.

First, if your cluster is using Raspberry Pis or other ARM-based hosts, the `test-pod.yaml` file needs to be modified to use a busybox image created for ARM hosts. The default will pull from the `gcr.io` registry but does not have the correct architecture for the sh binary, resulting in “exec format” errors. If your Kubernetes cluster is running on x86_64 hosts, you can skip this step.

Change the `test-pod.yaml` file to use the `docker.io/aarch64/busybox:latest` container image. Your file should look like this:

```
kind: Pod
apiVersion: v1
metadata:
  name: test-pod
spec:
  containers:
  - name: test-pod
    image: docker.io/aarch64/busybox:latest
    # image: gcr.io/google_containers/busybox:1.24
    command:
     - "/bin/sh"
    args:
     - "-c"
      - "touch /mnt/SUCCESS && exit 0 || exit 1"
    volumeMounts:
      - name: nfs-pvc
        mountPath: "/mnt"
  restartPolicy: "Never"
  volumes:
    - name: nfs-pvc
      persistentVolumeClaim:
        claimName: test-claim
```

The pod described in the file will create a busybox container, mount the NFS volume from the `test-claim` persistent volume to `/mnt` inside the container, and create a file named SUCCESS.

```
# Creae the test pod container
$ kubectl create -f test-pod.yaml
pod/test-pod created

# Validate the container ran without problem
$ kubectl get po
NAME                                     READY   STATUS      RESTARTS   AGE
nfs-client-provisioner-6ddfb9bb6d-x4zwt  1/1     Running     0          20m
test-pod                                 0/1     Completed   0          65s
```

If the container ran correctly and the status is `Completed`, then check the contents of the volume on your NFS server:

```
# From the NFS server, within the directory for the PVC
$ ls default-test-claim-pvc-bdf41489-abe4-4508-8adc-74e80f70c626/
SUCCESS
```

Success indeed! The pod was able to mount the NFS volume created by the NFS-client provisioner and write to it!

Clean up the test pod and PVC with the `kubectl delete` command:

```
# Cleanup the test-pod pod
$ kubectl delete po test-pod
pod "test-pod" deleted

# Cleanup the test-claim pvc
$ kubectl delete pvc test-claim
persistentvolumeclaim "test-claim" deleted
```

Turn up the volume(s)

Thanks to the NFS-client provisioner and your NFS server, you can now request persistent volumes to use with your pods by creating PVCs, and the claims will be automatically filled with dynamically provisioned NFS volumes. This mirrors, in most aspects, how Kubernetes works with dynamic provisioners in large public clouds and allows you to use a workflow like one you would use with a public cloud provider. In fact, a benefit of storage classes is the abstraction created between the PVCs and the volume providers. This allows you to use storage classes with the same name and different providers between on-premises private clouds and any of the public cloud providers, making it easier to "lift-and-shift" workloads between them.

By using NFS, you can support multi-read and multi-write operations on the volumes, too, allowing multiple pods to use the volumes at the same time. As mentioned, this is great for load-balanced web servers or similar services and allows you to scale your services across multiple nodes at the same time.

Best of all, the NFS-client provisioner is automatic, dynamically creating volumes within the NFS export so that they do not have to be manually provisioned ahead of time!

Take your new NFS-client provisioner out for a spin, and create some projects that need persistent storage. Perhaps try (shameless plug incoming) deploying InfluxDB and Grafana on Kubernetes to collect Twitter stats [11].

Links

[1] https://github.com/kubernetes-incubator/external-storage
[2] https://opensource.com/article/20/5/nfs-raspberry-pi
[3] https://github.com/kubernetes-incubator/external-storage
[4] https://kubernetes.io/docs/concepts/storage/persistent-volumes/
[5] https://kubernetes.io/docs/concepts/storage/persistent-volumes/#persistentvolumeclaims
[6] https://kubernetes.io/docs/concepts/storage/storage-classes/
[7] https://github.com/kubernetes-incubator/external-storage/tree/master/nfs-client
[8] https://github.com/kubernetes-incubator/external-storage/tree/master/nfs-client#with-helm
[9] https://opensource.com/article/20/5/nfs-raspberry-pi
[10] https://opensource.com/article/20/5/kubernetes-raspberry-pi
[11] https://opensource.com/article/19/2/deploy-influxdb-grafana-kubernetes

Use this script to find a Raspberry Pi on your network

Identify a specific Raspberry Pi in your cluster with a script that triggers an LED to flash.

WE'VE ALL BEEN THERE. "I'm going to get this Raspberry Pi [1] to try out. They look kinda cool." And then, like tribbles on an Enterprise, suddenly you have Kubernetes clusters [2] and NFS servers [3] and Tor proxies [4]. Maybe even a hotel booking system! [5]

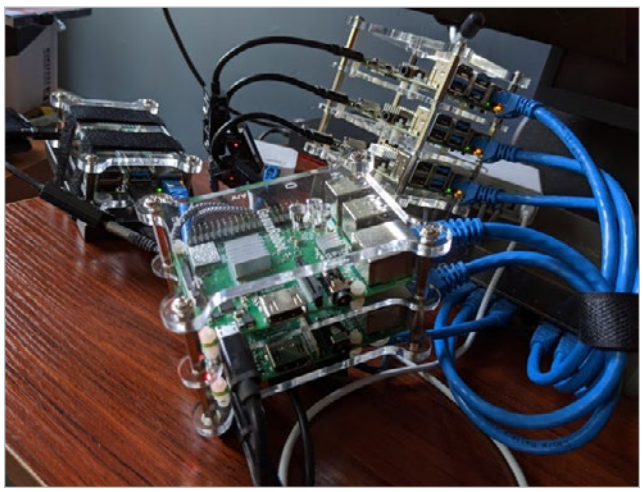
Pis cover the desk. They spill out onto the floor. Carrier boards for Raspberry Pi compute modules installed into lunchboxes litter the shelves.

…or maybe that's just me?

I'll bet if you have one Raspberry Pi, you've got at least two others, though, and gosh darn it, they all look the same.

This was the situation I found myself in recently while testing a network filesystem (NFS) server I set up on one of my Raspberry Pis. I needed to plug in a USB hard drive, but … to which one? Ol' Lingonberry Pi was the chosen host, and I was SSH'd into her, but which actual, *physical* RPi was she? There was no way of knowing…

Or was there?



So, so many Raspberry Pis. Which one is Lingonberry? Chris Collins, CC BY-SA 4.0

At a previous job, I sometimes worked on servers in our data centers, and some of them had a neat feature: an ID button on the front of the server that, when pressed, started an LED flashing on the front and back of the server. If I needed to deal with the other side of the server, I could press the ID button, then walk *allllll* the way around to the other side of the rack, and easily find the right server.

I needed something like this to find Lingonberry.

There aren't any buttons on the Pis, but there are LEDs, and after a quick Google search, I learned that one of them is *controllable* [6]. *Cue maniacal laughter*.

There are three important bits to know. First, the LED path: on Raspberry Pis, at least those running Ubuntu 20.04, the front (and user-controllable) LED is found at `/sys/class/leds/led0`. If you navigate to it, you'll find it is a symlink to a directory that has a number of files in it. The two important files are `trigger` and `brightness`.

The `trigger` file controls what lights up the LED. If you `cat` that file, you will find a list:

```
none usb-gadget usb-host rc-feedback rfkill-any rfkill-none
  kbd-scrolllock kbd-numlock kbd-capslock kbd-kanalock
  kbd-shiftlock kbd-altgrlock kbd-ctrllock kbd-altlock
  kbd-shiftllock kbd-shiftrlock kbd-ctrlllock kbd-ctrlrlock
  timer oneshot disk-activity disk-read disk-write ide-disk
  mtd nand-disk heartbeat backlight gpio cpu cpu0 cpu1 cpu2
  cpu3 default-on input panic mmc1 [mmc0] bluetooth-power
  rfkill0 unimac-mdio--19:01:link unimac-mdio--19:01:1Gbps
  unimac-mdio--19:01:100Mbps unimac-mdio--19:01:10Mbps
```

The item in brackets indicates what triggers the LED; in the example above, it's `[mmc0]`—the disk activity for when the SD card plugged into the Raspberry Pi. The trigger file isn't a normal file, though. Rather than editing it directly, you change the trigger by echoing one of the triggers into the file.

To identify Lingonberry, I needed to temporarily disable the [mmc0] trigger, so I could make the LED work how I wanted it to work. In the script, I disabled all the triggers by echoing "none" into the trigger file:

```
# You must be root to do this
$ echo none >trigger
```

```
$ cat trigger
[none] usb-gadget usb-host rc-feedback rfkill-any rfkill-
  none kbd-scrolllock kbd-numlock kbd-capslock kbd-kanalock
  kbd-shiftlock kbd-altgrlock kbd-ctrllock kbd-altlock kbd-
  shiftllock kbd-shiftrlock kbd-ctrlllock kbd-ctrlrlock timer
  oneshot disk-activity disk-read disk-write ide-disk mtd
  nand-disk heartbeat backlight gpio cpu cpu0 cpu1 cpu2 cpu3
  default-on input panic mmc1 mmc0 bluetooth-power rfkill0
  unimac-mdio--19:01:link unimac-mdio--19:01:1Gbps
  unimac-mdio--19:01:100Mbps unimac-mdio--19:01:10Mbps
```

In the contents of the trigger file above, you can see `[none]` is now the selected trigger. Now the LED is off and not flashing.

Next up is the `brightness` file. You can control whether the LED is on (1) or off (0) by echoing either 0 or 1 into the file. Alternating 1 and 0 will make the LED blink, and doing it with a one-second sleep in the middle produces a regular on/off blink unlike any of the activity that would otherwise trigger the LED. This is perfect for identifying the Raspberry Pi.

Finally, if you do not set the trigger file back to a trigger, it remains off. That's not what you want most of the time—it's better to see the disk activity. This means you have to make sure that any script you write will reset the trigger when it's finished or interrupted. That calls for a signal trap [7]. A trap will capture the `SIGINT` or `SIGTERM` (or other) signals and execute some code before quitting. This way, if the script is interrupted—say if you press **CTRL+C** to stop it—it can still reset the trigger.

With this newfound knowledge, I was able to bang out a script (available under the MIT License [8]) pretty quickly and toss it onto my Raspberry Pis:

```
#!/bin/sh

set -o errexit
set -o nounset

trap quit INT TERM

COUNT=0

if ! [ $(id -u) = 0 ]; then
   echo "Must be run as root."
   exit 1
fi

LED="/sys/class/leds/led0"

if [[ ! -d $LED ]]
then
  echo "Could not find an LED at ${LED}"
  echo "Perhaps try '/sys/class/leds/ACT'?"
  exit 1
fi

function quit() {
  echo mmc0 >"${LED}/trigger"
}

echo -n "Blinking Raspberry Pi's LED - press CTRL-C to quit"

echo none >"${LED}/trigger"

while true
do
  let "COUNT=COUNT+1"
  if [[ $COUNT -lt 30 ]]
  then
    echo 1 >"${LED}/brightness"
    sleep 1
    echo 0 >"${LED}/brightness"
    sleep 1
  else
    quit
    break
  fi
done
```

This script checks that the LED control directory exists, disables the `[mmc0]` trigger, and then starts a loop blinking the LED on and off every second. It also includes a trap to catch `INT` and `TERM` signals and resets the trigger. I copied this script onto all my Raspberry Pis, and any time I need to identify one of them, I just run it. It worked perfectly to identify Ol' Lingonberry, so I could set up the disks for the NFS server, and I've used it a number of times since then.

One thing to note—the path to the LED might be different in other distributions. There are also other LEDs in the `/sys/class/leds` directory, but they are not controllable by the user; they are hooked into different bits of the firmware of the Raspberry Pi.

Links

[1] https://opensource.com/resources/raspberry-pi
[2] https://opensource.com/article/20/6/kubernetes-raspberry-pi
[3] https://opensource.com/article/20/5/nfs-raspberry-pi
[4] https://opensource.com/article/20/4/tor-proxy-raspberry-pi
[5] https://opensource.com/article/20/4/qloapps-raspberry-pi
[6] https://www.raspberrypi.org/forums/viewtopic.php?t=12530
[7] https://tldp.org/LDP/Bash-Beginners-Guide/html/sect_12_02.html
[8] https://opensource.org/licenses/MIT

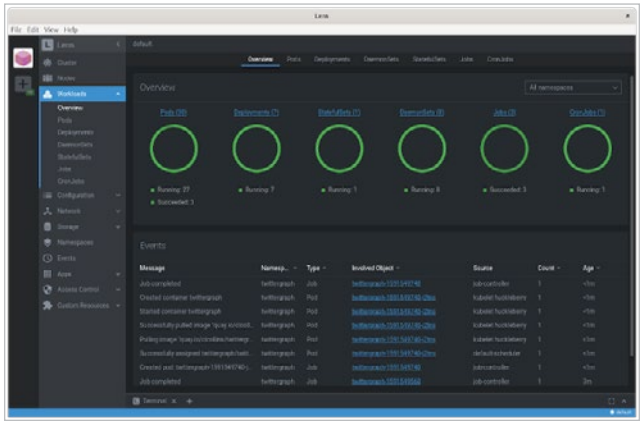
Manage your Kubernetes cluster with Lens

Lens is a useful, attractive, open source user interface for working with Kubernetes clusters.

AS MORE WORKLOADS are migrated to containerized environments, it becomes challenging to manage those larger numbers of containers and the connections between them and other systems. As the scale and complexity of a containerized environment increase past a human's ability to manage, container orchestration platforms such as Kubernetes [1] become increasingly important. Such platforms, however, come with their own management challenges that require metrics, observability, and a user-friendly interface to present their huge amount of complexity.

Enter Lens

Lens [2], which bills itself as "the Kubernetes IDE," is a useful, attractive, open source user interface (UI) for working with Kubernetes clusters. Out of the box, Lens can connect to Kubernetes clusters using your kubeconfig file and will display information about the cluster and the objects it contains. Lens can also connect to—or install—a Prometheus stack and use it to provide metrics about the cluster, including node information and health.



An overview of workloads on the cluster. (Chris Collins, CC BY-SA 4.0)

Like Kubernetes' dashboard and OpenShift, Lens provides live updates on the state of objects in the cluster and metrics collected by Prometheus [3].

Get started

Installing Lens is straightforward. AppImage [4] packages are available for Linux, and there are binaries available for macOS and Windows clients. This tutorial explains how to download and use the Lens AppImage to install and use Lens on Linux.

According to AppImage's FAQ [5], an AppImage is "a downloadable file for Linux that contains an application and everything the application needs to run." An application packaged as an AppImage is just that—a single executable file that can be downloaded and run.

The AppImage for Lens can be downloaded from the Lens Releases [6] page on GitHub. After you download it, mark the file executable with chmod, and then either execute it directly or copy it to a place in your `$PATH`:

```
# Download the 3.4.0 AppImage for Lens, mark it executable and
  copy it to your $PATH
# (output omitted for brevity)

$ wget https://github.com/lensapp/lens/releases/download/
  v3.4.0/Lens-3.4.0.AppImage
$ chmod +x Lens-3.4.0.AppImage
$ sudo mv Lens-3.4.0.AppImage /usr/sbin/lens
```

Then you can start Lens by typing `lens` on the command line.

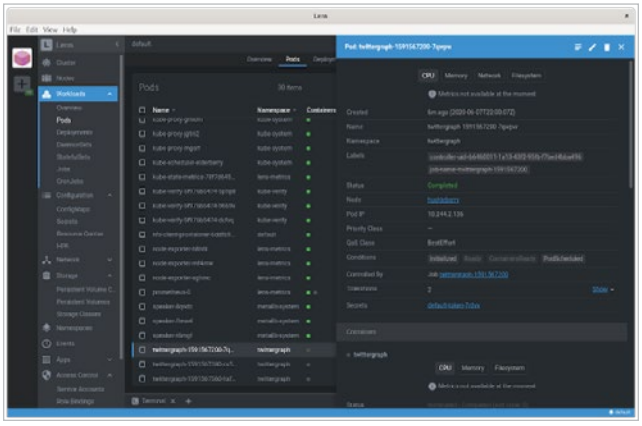
Connect Lens to a Kubernetes cluster

Once you launch Lens, connect it to a Kubernetes cluster by clicking the + icon in the top-left corner and selecting a kubeconfig. Next, a drop-down box will appear containing any Kubernetes contexts from your `~/.kube/config` file, or you can select a custom one. Because cluster and authentication information about the cluster for any context is included in the kubeconfig file, Lens treats each context as a different cluster, unfortunately.

This is particularly unhelpful compared with how OpenShift creates context information in the kubeconfig file automatically for any project (namespace) you switch to. As a site-reliability engineer (SRE) working on hundreds of

clusters, I had dozens and dozens of "clusters" to choose from when setting up Lens. In practice, I found it best to select the default context for any cluster. You can manage all namespaces and workloads once Lens has connected, and there's no need to add them all.

Once it's connected, Lens will display a ton of information about your cluster. You can see the workloads that are running: pods and deployments, daemon sets, cron jobs, etc. You can also view information about config maps and secrets, networking information, storage, namespaces, and events. Each will let you drill down into the information about a given object, and you can even edit the objects directly in Lens.



Details of pods running on the cluster. (Chris Collins, CC BY-SA 4.0)

Gather metrics about your cluster

One of Lens' incredibly helpful features is its ability to connect to a Prometheus stack installed in your cluster to gather metrics about the cluster and its nodes for both current and historical data. This is great for getting at-a-glance information about the cluster right within the Lens UI without having to go to an external dashboard. However, the information presented is not comprehensive–it's good for an overview, but you may still wish to utilize a visualization tool such as Grafana with a more complicated dashboard to gather more specialized information.

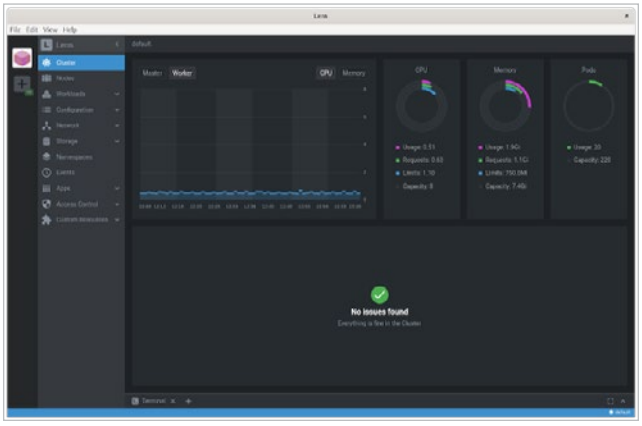
Along with being able to connect to an existing Prometheus stack provisioned in the cluster, Lens can install applications on your behalf, too. This is very useful for enthusiasts running Kubernetes clusters in their homelabs to be able to deploy and connect to Prometheus in a single click.

Install Prometheus with Lens

If you have been following along with this series, especially *Build a Kubernetes cluster with the Raspberry Pi* [7], you will have a Kubernetes cluster provisioned in your homelab for education and tinkering. One thing the vanilla cluster lacks is metrics, and this is a great opportunity to add Prometheus to the cluster and install the kube-state-metrics [8] service to gather information about the cluster.

To install it, just right-click on the cluster icon in the top-left corner of the Lens UI (after connecting to the cluster, of course) and select **Settings**. Under **Features** on the Settings page, you will find a **Metrics** section and a button to install Prometheus. Click **Install** to deploy the Prometheus stack to your cluster, and Lens will auto-detect its existence and begin displaying metrics. (It will take a minute—the new Prometheus has to collect some metrics first.)

I also appreciate that Lens links directly to the manifests used to deploy this stack, so you can verify what will be created before doing it, if you want.



Hardware utilization metrics about the cluster. (Chris Collins, CC BY-SA 4.0)

Fix kube-state-metrics

Unfortunately, while Prometheus will install just fine on a Raspberry Pi-based cluster, the kube-state-metrics service will fail. Currently, the kube-state-metrics project does not build an AArch64/ARM64 image, so pods created from that image will continuously crash with exec format error messages in the logs.

Luckily this issue is being tracked [9], and the kube-state-metrics project is working toward building the infrastructure to produce official ARM images. Until then, however, you can use a community-developed image and patch the kube-state-metrics deployment directly using Lens.

Go back into the cluster information, click on **Workloads**, and select **Deployments**. A list of all the Kubernetes deployment objects in the cluster will appear in the pane on the right. You should be able to pick out the kube-state-metrics deployment easily by the angry red entry under the **Conditions** column that indicates the crash-looping pod issue.

Select the kube-state-metrics deployment, and details of the object slide out from the right in an overlay window. In the upper-right corner of this window is a pencil icon. Click that icon to open an editor window with the YAML representation of the kube-state-metrics deployment. Scroll down, and edit the **.spec.template.spec.containers.image** value. By de-

fault, this value points to the official image: **quay.io/coreos/kube-state-metrics:v1.9.5**. Replace this value with **docker.io/carlosedp/kube-state-metrics:v1.9.5**.

In that same deployment are **nodeAffinity** rules, one of which forces the deployment to only run pods on hosts with amd64 architecture. This won't work for an AArch64/ARM64 Raspberry Pi, and with the updated image above, it's not useful. Find the key **beta.kubernetes.io/arch** in **.spec.template.spec.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms**:

```
- key: beta.kubernetes.io/arch
  operator: In
  values:
  - amd64
```

Delete this key entirely, and that will allow the pods from the deployment to be scheduled on your Raspbery Pi nodes. Click Save. This will trigger the deployment to roll out new kube-state-metrics pods with an ARM64 architecture, and they should become ready and begin reporting the metrics directly to Prometheus.

Lens lets you see clearly

Kubernetes is complex, and any tool that makes it easier to visualize and work with Kubernetes clusters can lower the barrier of entry for new folks and make life considerably easier for experienced Kubernetes administrators. Lens knocks this out of the park with an attractive, intuitive, and easy-to-use UI for managing one or more clusters, from the 10,000-foot view down into the nitty-gritty of individual Kubernetes objects. Lens also helps display metrics about the cluster and makes installing and using a Prometheus stack to display the metrics almost push-button.

I am extremely impressed with Lens and use it to manage several Kubernetes clusters in my own homelab, and I hope you find it useful as well.

Links

[1] https://opensource.com/resources/what-is-kubernetes
[2] https://k8slens.dev/
[3] https://opensource.com/sitewide-search?search_api_views_fulltext=prometheus
[4] https://opensource.com/article/20/6/appimages
[5] https://docs.appimage.org/user-guide/faq.html#question-what-is-an-appimage
[6] https://github.com/lensapp/lens/releases/latest
[7] https://opensource.com/article/20/6/kubernetes-raspberry-pi
[8] https://github.com/kubernetes/kube-state-metrics
[9] https://github.com/kubernetes/kube-state-metrics/issues/1037

Install a Kubernetes load balancer on your Raspberry Pi homelab with MetalLB

Assign real IPs from your home network to services running in your cluster and access them from other hosts on your network.

KUBERNETES IS DESIGNED to integrate with major cloud providers' load balancers to provide public IP addresses and direct traffic into a cluster. Some professional network equipment manufacturers also offer controllers to integrate their physical load-balancing products into Kubernetes installations in private data centers. For an enthusiast running a Kubernetes cluster at home, however, neither of these solutions is very helpful.

Kubernetes does not have a built-in network load-balancer implementation. A bare-metal cluster, such as a Kubernetes cluster installed on Raspberry Pis for a private-cloud homelab [1], or really any cluster deployed outside a public cloud and lacking expensive professional hardware, needs another solution. MetalLB [2] fulfills this niche, both for enthusiasts and large-scale deployments.

MetalLB is a network load balancer and can expose cluster services on a dedicated IP address on the network, allowing external clients to connect to services inside the Kubernetes cluster. It does this via either layer 2 (data link) [3] using Address Resolution Protocol (ARP) [4] or layer 4 (transport) [5] using Border Gateway Protocol (BGP) [6].

While Kubernetes does have something called Ingress [7], which allows HTTP and HTTPS traffic to be exposed outside the cluster, it supports *only* HTTP or HTTPS traffic, while MetalLB can support any network traffic. It is more of an apples-to-oranges comparison, however, because MetalLB provides resolution of an unassigned IP address to a particular cluster node and assigns that IP to a Service, while Ingress uses a specific IP address and internally routes HTTP or HTTPS traffic to a Service or Services based on routing rules.

MetalLB can be set up in just a few steps, works especially well in private homelab clusters, and within Kubernetes clusters, it behaves the same as public cloud load-balancer integrations. This is great for education purposes (i.e., learning how the technology works) and makes it easier to "lift-and-shift" workloads between on-premises and cloud environments.

ARP vs. BGP

As mentioned, MetalLB works via either ARP or BGP to resolve IP addresses to specific hosts. In simplified terms, this means when a client attempts to connect to a specific IP, it will ask "which host has this IP?" and the response will point it to the correct host (i.e., the host's MAC address).

With ARP, the request is broadcast to the entire network, and a host that knows which MAC address has that IP address responds to the request; in this case, MetalLB's answer directs the client to the correct node.

With BGP, each "peer" maintains a table of routing information directing clients to the host handling a particular IP for IPs and the hosts the peer knows about, and it advertises this information to its peers. When configured for BGP, MetalLB peers each of the nodes in the cluster with the network's router, allowing the router to direct clients to the correct host.

In both instances, once the traffic has arrived at a host, Kubernetes takes over directing the traffic to the correct pods.

For the following exercise, you'll use ARP. Consumer-grade routers don't (at least easily) support BGP, and even higher-end consumer or professional routers that do support BGP can be difficult to set up. ARP, especially in a small home network, can be just as useful and requires no configuration on the network to work. It is considerably easier to implement.

Install MetalLB

Installing MetalLB is straightforward. Download or copy two manifests from MetalLB's GitHub repository [8] and apply them to Kubernetes. These two manifests create the namespace MetalLB's components will be deployed to and the components themselves: the MetalLB controller, a "speaker" daemonset, and service accounts.

Install the components

Once you create the components, a random secret is generated to allow encrypted communication between the speakers (i.e., the components that "speak" the protocol to make services reachable).

(Note: These steps are also available on MetalLB's website.)

The two manifests with the required MetalLB components are:

- https://raw.githubusercontent.com/metallb/metallb/v0.9.3/manifests/namespace.yaml
- https://raw.githubusercontent.com/metallb/metallb/v0.9.3/manifests/metallb.yaml

They can be downloaded and applied to the Kubernetes cluster using the kubectl apply command, either locally or directly from the web:

```
# Verify the contents of the files, then download and pipe then
  to kubectl with curl
# (output omitted)
$ kubectl apply -f https://raw.githubusercontent.com/metallb/
  metallb/v0.9.3/manifests/namespace.yaml
$ kubectl apply -f https://raw.githubusercontent.com/metallb/
  metallb/v0.9.3/manifests/metallb.yaml
```

After applying the manifests, create a random Kubernetes secret for the speakers to use for encrypted communications:

```
# Create a secret for encrypted speaker communications
$ kubectl create secret generic -n metallb-system memberlist
  --from-literal=secretkey="$(openssl rand -base64 128)"
```

Completing the steps above will create and start all the MetalLB components, but they will not do anything until they are configured. To configure MetalLB, create a configMap that describes the pool of IP addresses the load balancer will use.

Configure the address pools

MetalLB needs one last bit of setup: a configMap with details of the addresses it can assign to the Kubernetes Service LoadBalancers. However, there is a small consideration. The addresses in use do not need to be bound to specific hosts in the network, but they must be free for MetalLB to use and not be assigned to other hosts.

In my home network, IP addresses are assigned by the DHCP server my router is running. This DHCP server should not attempt to assign the addresses that MetalLB will use. Most consumer routers allow you to decide how large your subnet will be and can be configured to assign only a subset of IPs in that subnet to hosts via DHCP.

In my network, I am using the subnet `192.168.2.1/24`, and I decided to give half the IPs to MetalLB. The first half of the subnet consists of IP addresses from `192.168.2.1` to `192.168.2.126`. This range can be represented by a `/25` subnet: `192.168.2.1/25`. The second half of the subnet can similarly be represented by a `/25` subnet: `192.168.2.128/25`. Each half contains 126 IPs—more than enough for the hosts and Kubernetes services. Make sure to decide on subnets appropriate to your own network and configure your router and MetalLB appropriately.

After configuring the router to ignore addresses in the `192.168.2.128/25` subnet (or whatever subnet you are using), create a configMap to tell MetalLB to use that pool of addresses:

```
# Create the config map
$ cat <<EOF | kubectl create -f -
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
data:
  config: |
    address-pools:
    - name: address-pool-1
      protocol: layer2
      addresses:
      - 192.168.2.128/25
EOF
```

The example configMap above uses CIDR [9] notation, but the list of addresses can also be specified as a range:

```
addresses:
 - 192.168.2.128-192.168.2.254
```

Once the configMap is created, MetalLB will be active. Time to try it out!

Test MetalLB

You can test the new MetalLB configuration by creating an example web service, and you can use one from a previous article [10] in this series: Kube Verify. Use the same image to test that MetalLB is working as expected: `quay.io/clcollins/kube-verify:01`. This image contains an Nginx server listening for requests on port 8080. You can view the Containerfile [11] used to create the image. If you want, you can instead build your own container image from the Containerfile and use that for testing.

If you previously created a Kubernetes cluster on Raspberry Pis, you may already have a Kube Verify service running and can skip to the section [12] on creating a LoadBalancer-type of service.

If you need to create a kube-verify namespace

If you do not already have a `kube-verify` namespace, create one with the `kubectl` command:

```
# Create a new namespace
$ kubectl create namespace kube-verify
# List the namespaces
$ kubectl get namespaces
NAME              STATUS   AGE
default           Active   63m
kube-node-lease   Active   63m
kube-public       Active   63m
kube-system       Active   63m
metallb-system    Active   21m
kube-verify       Active   19s
```

With the namespace created, create a deployment in that namespace:

```
# Create a new deployment
$ cat <<EOF | kubectl create -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kube-verify
  namespace: kube-verify
  labels:
    app: kube-verify
spec:
  replicas: 3
  selector:
    matchLabels:
      app: kube-verify
  template:
    metadata:
      labels:
        app: kube-verify
    spec:
      containers:
      - name: nginx
        image: quay.io/clcollins/kube-verify:01
        ports:
        - containerPort: 8080
EOF
deployment.apps/kube-verify created
```

Create a LoadBalancer-type Kubernetes service

Now expose the deployment by creating a LoadBalancer-type Kubernetes service. If you already have a service named `kube-verify`, this will replace that one:

```
# Create a LoadBalancer service for the kube-verify deployment
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
  name: kube-verify
  namespace: kube-verify
spec:
  selector:
    app: kube-verify
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: LoadBalancer
EOF
```

You could accomplish the same thing with the `kubectl expose` command:

```
kubectl expose deployment kube-verify -n kube-verify
  --type=LoadBalancer --target-port=8080 --port=80
```

MetalLB is listening for services of type LoadBalancer and immediately assigns an external IP (an IP chosen from the range you selected when you set up MetalLB). View the new service and the external IP address MetalLB assigned to it with the `kubectl get service` command:

```
# View the new kube-verify service
$ kubectl get service kube-verify -n kube-verify
NAME         TYPE          CLUSTER-IP     EXTERNAL-IP    PORT(S)        AGE
kube-verify  LoadBalancer  10.105.28.147  192.168.2.129  80:31491/TCP   4m14s

# Look at the details of the kube-verify service
$ kubectl describe service kube-verify -n kube-verify
Name:                     kube-verify
Namespace:                kube-verify
Labels:                   app=kube-verify
Annotations:              <none>
Selector:                 app=kube-verify
Type:                     LoadBalancer
IP:                       10.105.28.147
LoadBalancer Ingress:     192.168.2.129
Port:                     <unset>  80/TCP
TargetPort:               8080/TCP
NodePort:                 <unset>  31491/TCP
Endpoints:                10.244.1.50:8080,10.244.1.51:8080,10.244.2.36:8080
Session Affinity:         None
External Traffic Policy:  Cluster
Events:
  Type    Reason        Age    From                Message
  ----    ------        ----   ----                -------
  Normal  IPAllocated   5m55s  metallb-controller  Assigned IP
                                                   "192.168.2.129"
  Normal  nodeAssigned  5m55s  metallb-speaker     announcing from node
                                                   "gooseberry"
```

In the output from the `kubectl describe` command, note the events at the bottom, where MetalLB has assigned an IP address (yours will vary) and is “announcing” the assign-

ment from one of the nodes in your cluster (again, yours will vary). It also describes the port, the external port you can access the service from (80), the target port inside the container (port 8080), and a node port through which the traffic will route (31491). The end result is that the Nginx server running in the pods of the `kube-verify` service is accessible from the load-balanced IP, on port 80, from anywhere on your home network.

For example, on my network, the service was exposed on `http://192.168.2.129:80`, and I can `curl` that IP from my laptop on the same network:

```
# Verify that you receive a response from Nginx on the load-balanced IP
$ curl 192.168.2.129
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title>Test Page for the HTTP Server on Fedora</title>
(further output omitted)
```

MetalLB FTW

MetalLB is a great load balancer for a home Kubernetes cluster. It allows you to assign real IPs from your home network to services running in your cluster and access them from other hosts on your home network. These services can even be exposed outside the network by port-forwarding traffic through your home router (but please be careful with this!). MetalLB easily replicates cloud-provider-like behavior at home on bare-metal computers, Raspberry Pi-based clusters, and even virtual machines, making it easy to "lift-and-shift" workloads to the cloud or just familiarize yourself with how they work. Best of all, MetalLB is easy and convenient and makes accessing the services running in your cluster a breeze.

Links

[1] https://opensource.com/article/20/6/kubernetes-raspberry-pi
[2] https://metallb.universe.tf/
[3] https://en.wikipedia.org/wiki/Data_link_layer
[4] https://en.wikipedia.org/wiki/Address_Resolution_Protocol
[5] https://en.wikipedia.org/wiki/Transport_layer
[6] https://en.wikipedia.org/wiki/Border_Gateway_Protocol
[7] https://kubernetes.io/docs/concepts/services-networking/ingress/
[8] https://github.com/metallb/metallb
[9] https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing
[10] https://opensource.com/article/20/6/kubernetes-raspberry-pi
[11] https://github.com/clcollins/homelabCloudInit/blob/master/simpleCloudInitService/data/Containerfile
[12] https://opensource.com/article/20/7/homelab-metallb#loadbalancer