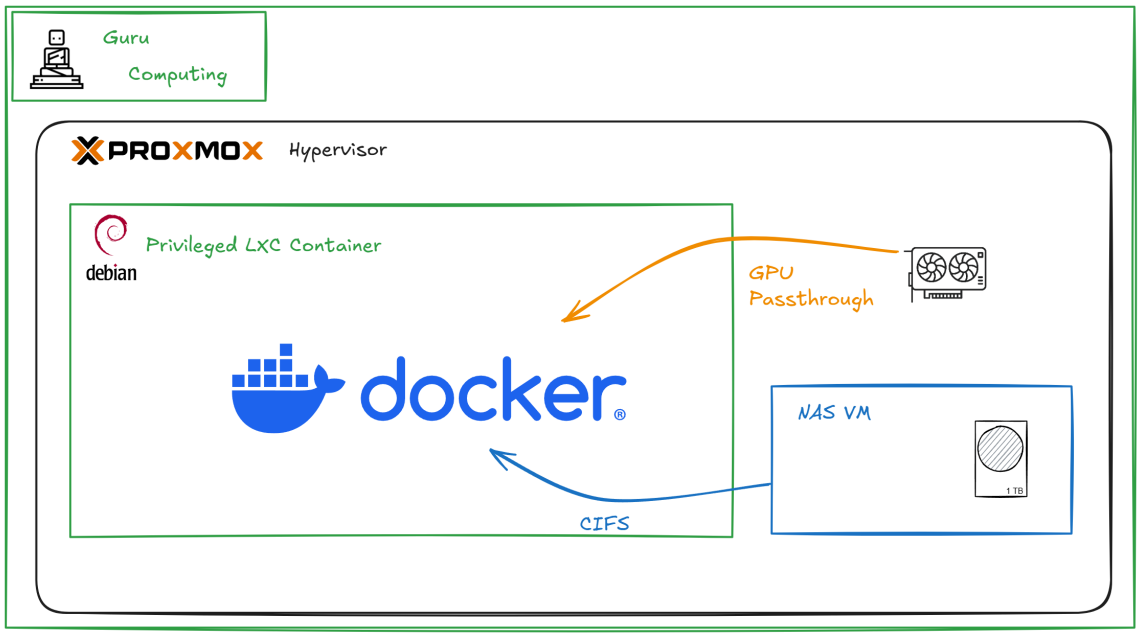
Running Docker in a Proxmox Container



Introduction

If you have been a historical follower of my guides, you'll see that most of my guides assume you will be operating a docker installation, usually on a baremetal (fedora or debian) instance. And it works great!

So if baremetal docker hosts are a perfectly functional way to self-host containers, why would you consider running docker for your homelab any other way? Well, the reason for this guide is that we want to take advantage of the features of proxmox:

- Resource isolation: being able to run a full docker instance on the machine while being able to, separately, run multiple other Linux (or Windows!) instances and virtual machines on the same device.

> **Info**
>
> This is especially useful if you have software expecting access to the **docker socket** (IE: Authentik, Kasm, or CICD Runners. You can set up an isolated docker host just for them to avoid security concerns about their access).

- Snapshots! Being able to easily back up the virtual machine or docker instance, and revert potentially risky changes.
- Guides! Like this one! Every time I write one of these, I have to be spinning up lab instances just for demo purposes and destroying them afterwards. Wouldn't it be great to be performing this on the same hardware as my homelab?

Pre-Requisites

To follow this guide, you need:

- An x86 device (not ARM) running Proxmox
- That's it!

But why a Proxmox Container?

Proxmox has two mechanisms of creating virtual linux environments to host workloads in:

- A Virtual Machine (like Hyper-V, VMware, Nutanix, etc. etc.)
- An **LXC** container (a similar concept to docker, but for fully interactive environments instead of purpose built application environments)

Proxmox themselves recommend using a Virtual Machine if you're running docker: not a container.

> If you want to run application containers, for example, Docker images, it is recommended that you run them inside a Proxmox QEMU VM. This will give you all the advantages of application containerization, while also providing the benefits that VMs offer, such as strong isolation from the host and the ability to live-migrate, which otherwise isn't possible with containers.

However using a LXC container still works! If you want the simple approach, do what proxmox says: Use a VM to host your docker environment. But there are some advantages to using a container instead:

- You can maximise your resources, if you are working on a resource constrained device (since you don't have to pre-allocate memory in a LXC container, whereas you do for a VM, as well as generally less overhead)
- You can easily share hardware devices with the host, such as GPU, NFS/CIFS Mounts, USB devices, or hard-disks.

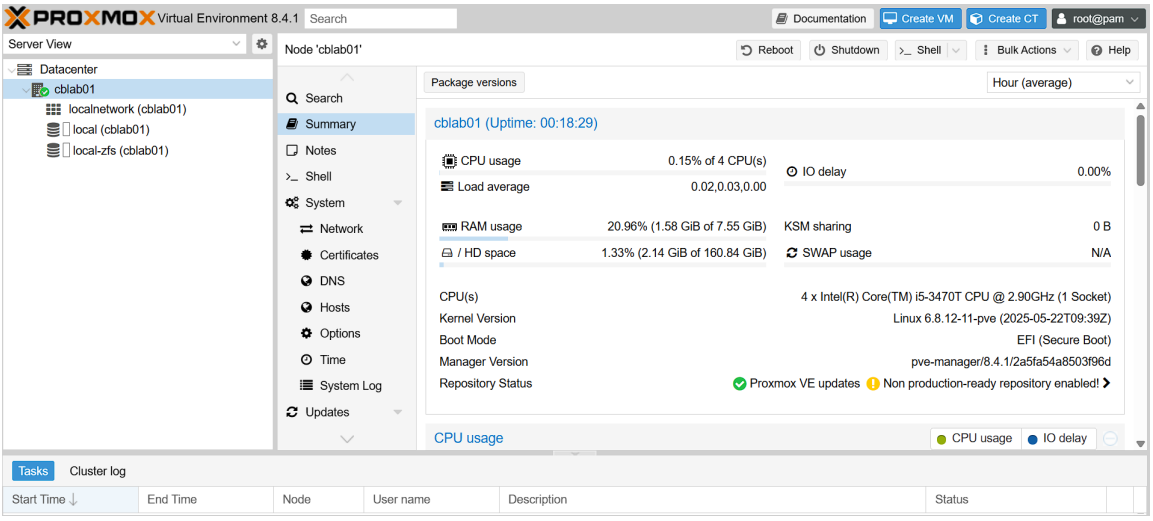
> **Info**
>
> You can also share GPU and Disks with a virtual machine, but it typically requires a feature called **PCI-Passthrough**. This functionally turns off the device for the host and moves it entirely into the virtual machine, which has some limitations and complications.

So with that in mind, we will be looking at using a container for our primary docker host instead.

Getting Started

So here is our proxmox instance:

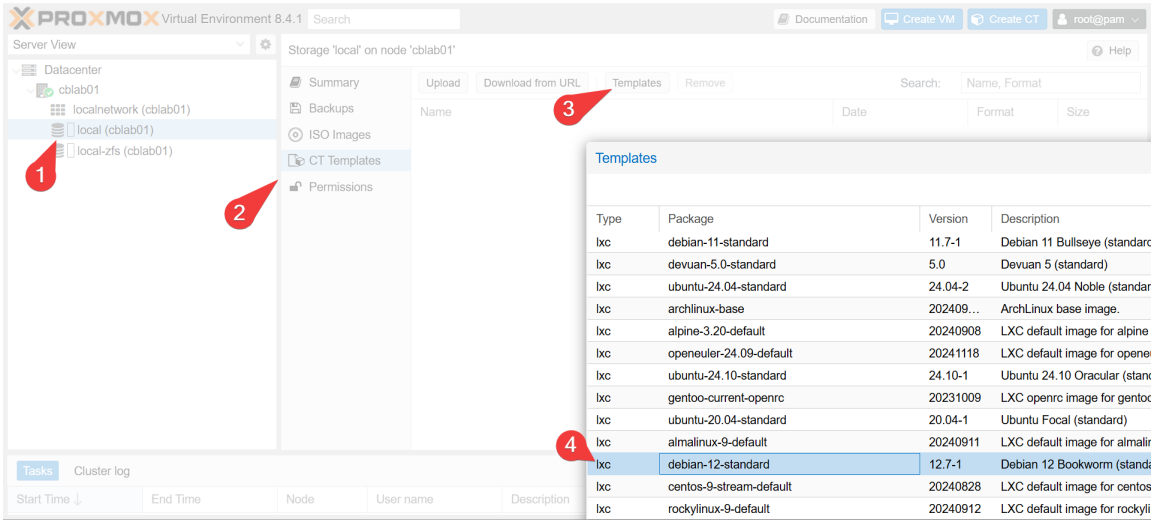
> **Info**
>
> If you are using proxmox with a **single drive**, you will want to install proxmox as either **zfs (raid 0)** or **btrfs (raid 0)** (btrfs is still experimental but will work, if you have strong btrfs tendencies)



The very first thing we will want to do is choose a base LXC template for our docker host. Given that proxmox runs on top of debian, the natural choice for a container is also debian (though it doesn't have to be).

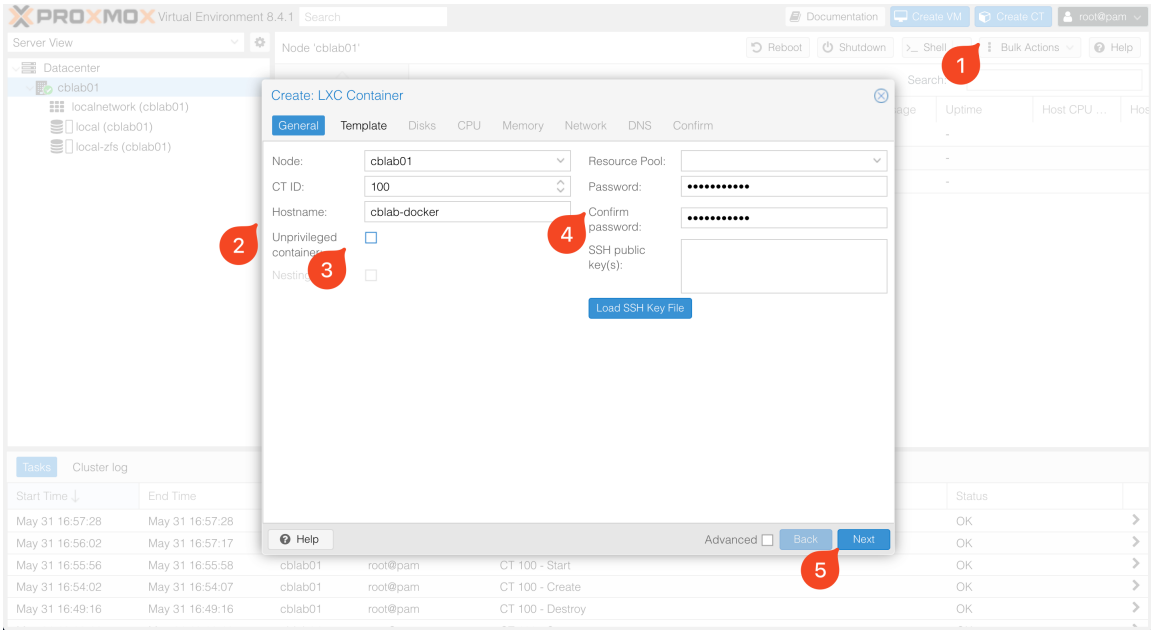
Creating the LXC Docker Container

- Click on your primary storage (local, typically) and then choose "CT Templates". Choose "templates" and download "Debian 12" out of the mix.



- Choose "Create CT" at the top and choose a hostname and password. uncheck **Unprivileged Container**

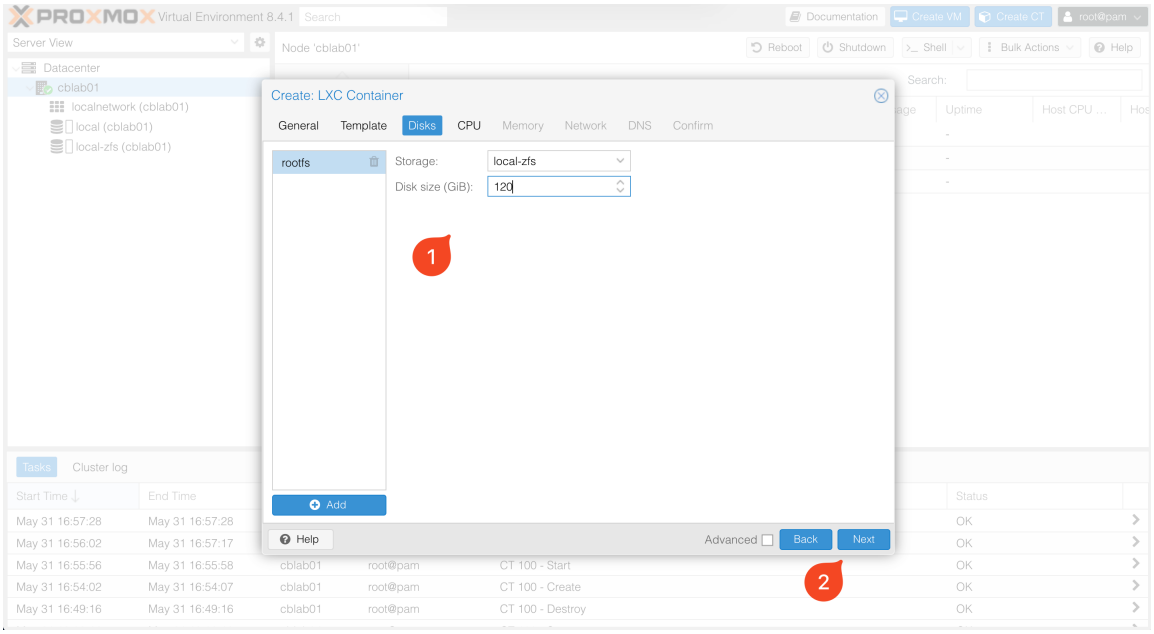
> **Info**
>
> Unchecking **unprivileged** (obviously) creates a **privileged** container. While you can do most docker functions inside an unprivileged container, some features (like mounting a NFS or CIFS share) require the container to be privileged to function. \nA privileged container has some security implications (namely, weakening the degree of isolation from Proxmox), but isn't necessarily insecure as a result. You can read more about that here.



- Pick your template, then move to disks and create a relatively large disk (it will have to

store your cached docker images, and potentially your docker volumes depending on how you want to configure your storage).

> **Info**
>
> Proxmox will default to **thinly provisioning** storage (for both VMs and containers) which means that the disk size only represents the **max** size. Note that you **can** overprovision storage (allocate more storage then exists). Attempting to **use** more storage than exists will end in a bad time for everybody.



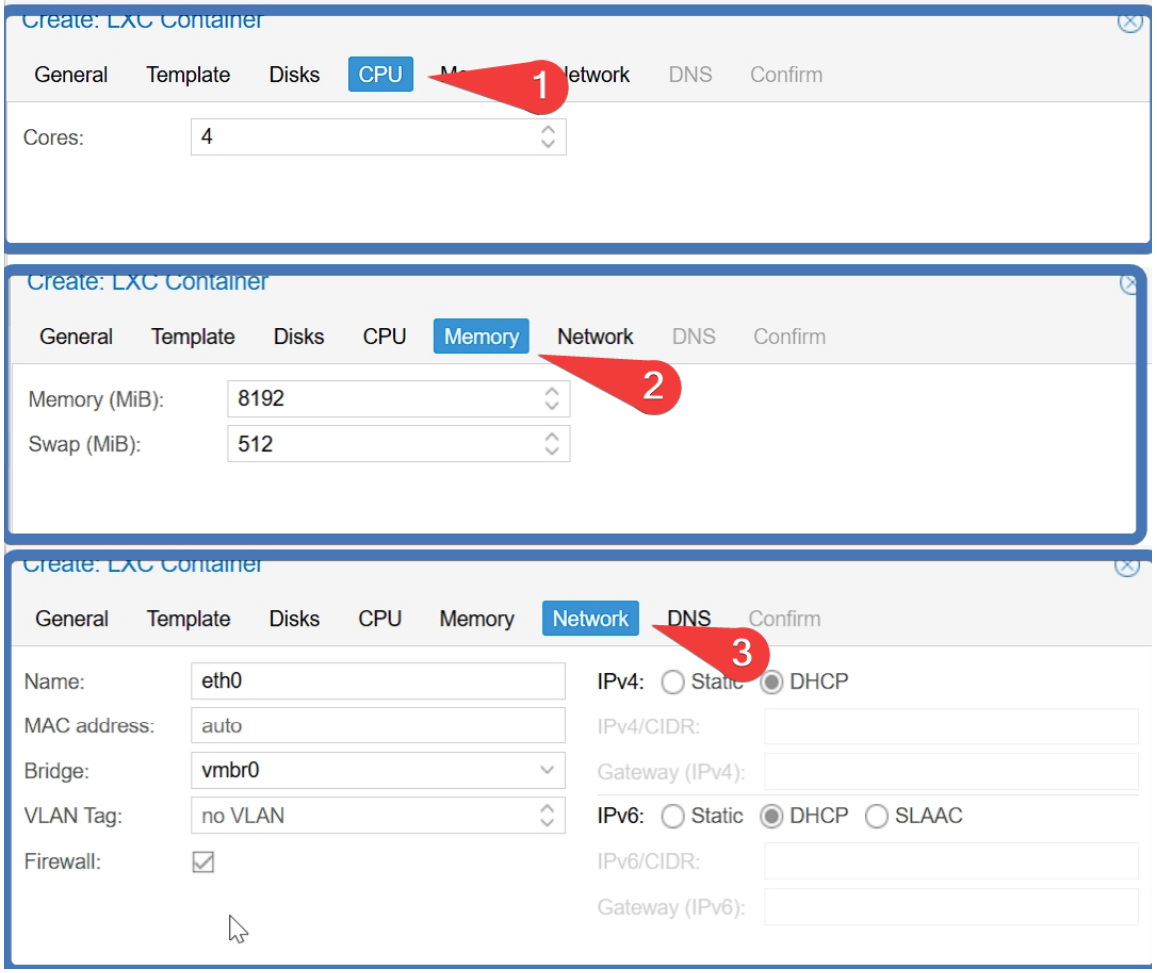
- Create as many CPUs as you want to allocate. For the docker host I typically will set it to the same amount as the host (you can find this info on the "node" part of the sidebar, 1 down from the datacenter).

> **Info**
>
> Both **VMs** and **Containers** will also thinly provision **CPUs**. Each **CPU** (or vCPU) is representative of a **thread**, not a **core**. \nSimilar to storage, you can also overprovision CPUs. However, if you have multiple containers/VMs attempting to max out the CPU utilisation, overprovisioning will lead to reduced performance as devices fight for scheduling

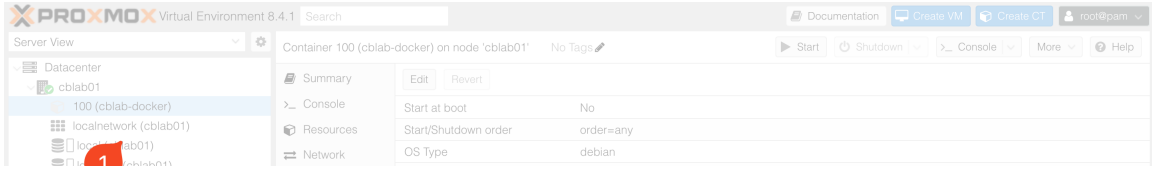
- Set your memory, again it can match the max as it is a limit, not a preallocation. Swap can be left at defaults.

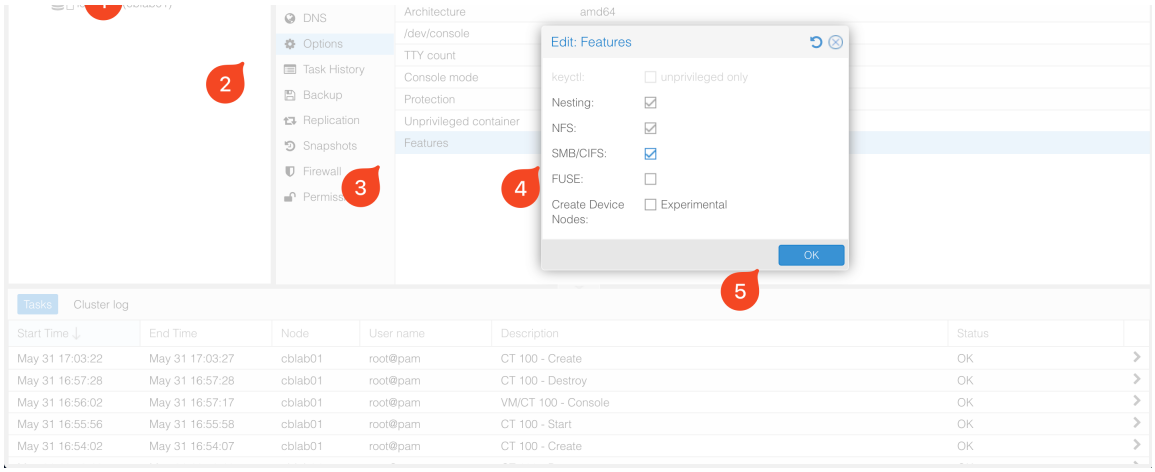
> **Info**
>
> Containers provide memory limits. VMs, however, **preallocate** memory, which is a clear advantage for **LXC** containers over **VM**s.

- Set your network settings, I prefer to set DHCP over static.

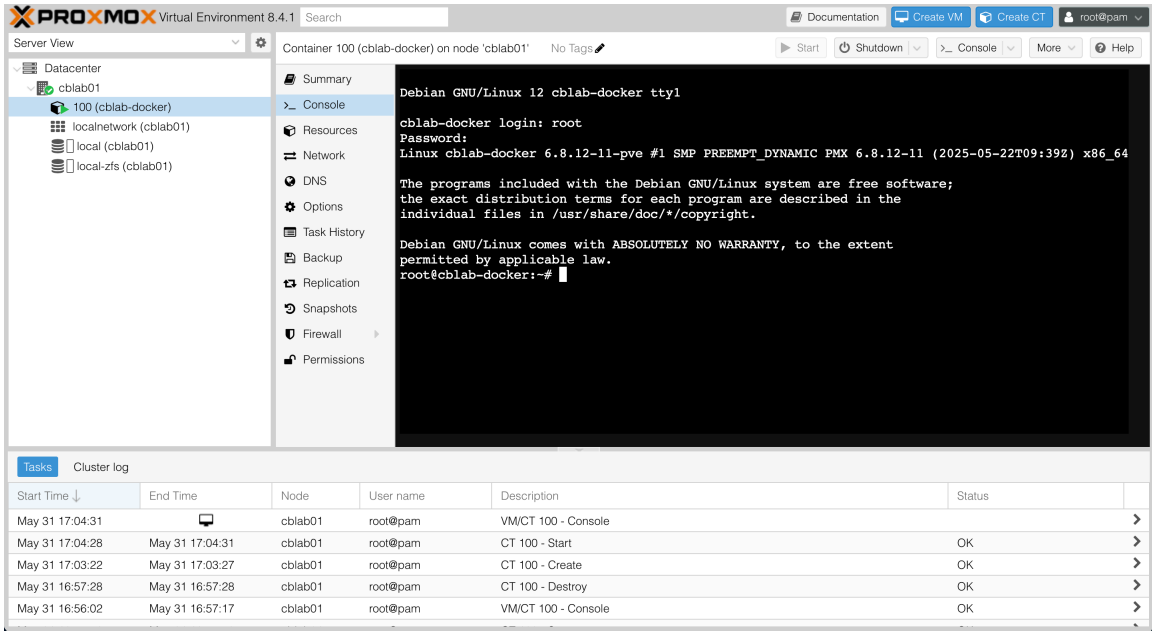


- Before starting, go into the container options and enable:
- Nesting
- SMB/CIFS
- NFS
- Also enable "Start at Boot" (assuming you want the container to start on boot)





- Finally, start up your new docker host!

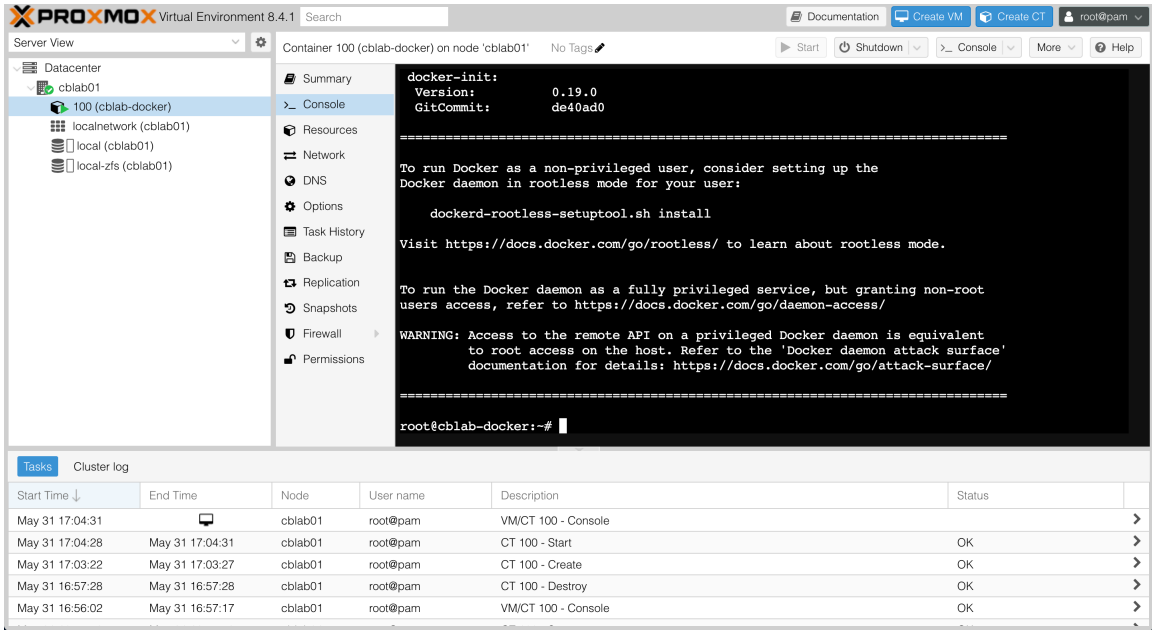


Installing Docker

Installing docker is easy! Docker makes it convenient by publishing a shell script to auto-install the latest version at https://get.docker.com/. We'll also install some basic utilities since the base LXC container is pretty barebones

```
apt-get update
apt-get dist-upgrade -y
apt-get install -y curl wget tmux micro htop
curl -fsSL https://get.docker.com | bash
```

> **Warning**
>
> Always review any scripts you plan to pipe into bash like this first. you're basically trusting an internet source (`get.docker.com` in this case) to execute on your machine.

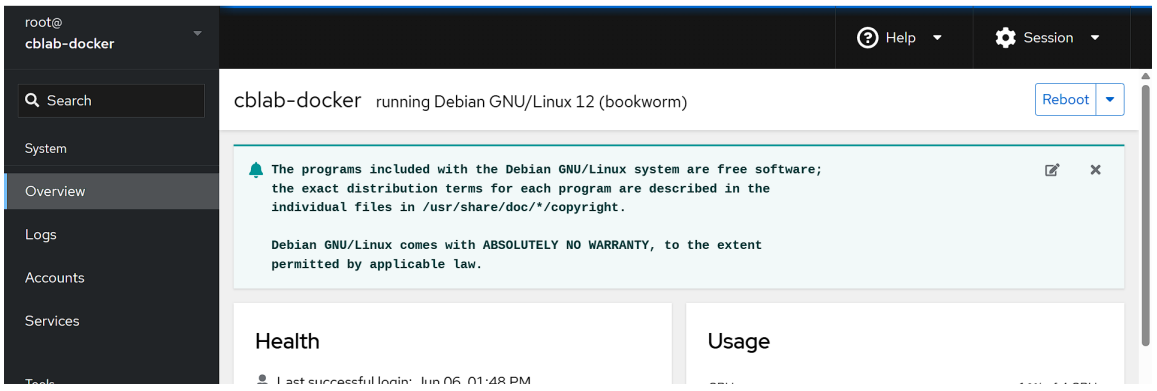


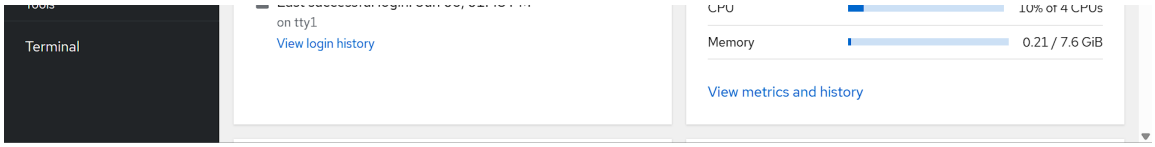
Easy!

Install Cockpit

This step isn't necessary, but it's nice to have a dedicated web interface for administration that isn't through the proxmox console.

```
apt-get install -y --no-install-recommends cockpit cracklib-runtime
# allow root to log into cockpit
rm -f /etc/cockpit/disallowed-users
touch /etc/cockpit/disallowed-users
```





Log in with Visual Studio Code

you can follow my @Using Remote VSCode guide to set up vscode with the microsoft containers extension to perform docker management.

- Also make your working folder (`/mnt/containers` is what I use, but it can be arbitrary)

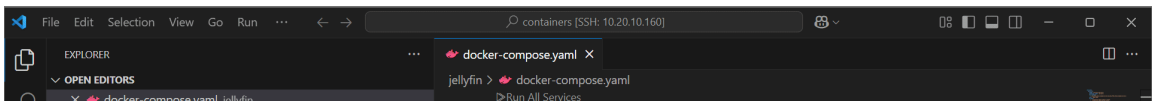
```
mkdir /mnt/containers
```

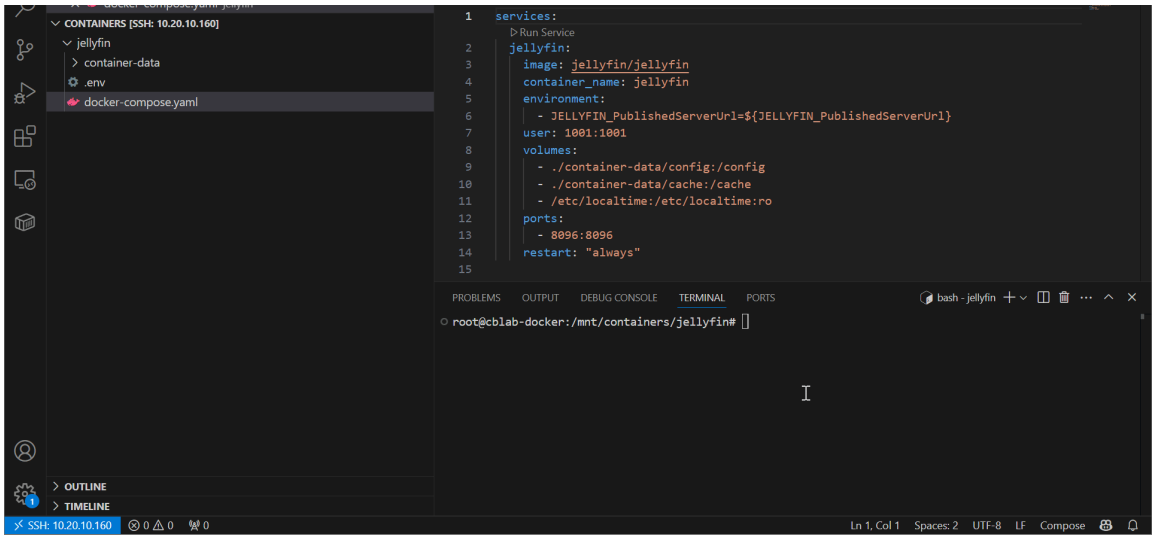
Use Docker

Let's start up a jellyfin instance to make sure it works.

> **Info**
>
> You will notice that this script will run jellyfin as user `1001` and not as `root`. Running containers as non-root is best practice for any docker container you plan to run in "production"

```
#!/bin/bash
# Variables. change the url to your IP
JELLYFIN_PublishedServerUrl="http://10.20.10.160:8096"
WORKING_DIR="/mnt/containers/jellyfin"

######
### main script.
######
# Assumption: ran as root

mkdir -p "${WORKING_DIR}"

# create the primary compose file
cat << 'EOF' > "${WORKING_DIR}/docker-compose.yaml"
services:
  jellyfin:
    image: jellyfin/jellyfin
    container_name: jellyfin
    environment:
      - JELLYFIN_PublishedServerUrl=${JELLYFIN_PublishedServerUrl}
    user: 1001:1001
    volumes:
      - ./container-data/config:/config
      - ./container-data/cache:/cache
      - /etc/localtime:/etc/localtime:ro
    ports:
      - 8096:8096
    restart: "always"
EOF

# create the env file
cat << EOF > "${WORKING_DIR}/.env"
JELLYFIN_PublishedServerUrl=${JELLYFIN_PublishedServerUrl}
EOF
chown 0:0 "${WORKING_DIR}/.env"
chmod 600 "${WORKING_DIR}/.env"

# make the relevant folders and give user permissions
mkdir -p "${WORKING_DIR}/container-data/config"
chown 1001:1001 "${WORKING_DIR}/container-data/config"
mkdir -p "${WORKING_DIR}/container-data/cache"
chown 1001:1001 "${WORKING_DIR}/container-data/cache"

# prompt to start the container
read -p "bring up the container? (y/n): " -n 1 -r
echo    # (optional) move to a new line
if [[ $REPLY =~ ^[Yy]$ ]]
then
  # start the container
  cd "${WORKING_DIR}"
  docker compose up -d
fi
```





It works!

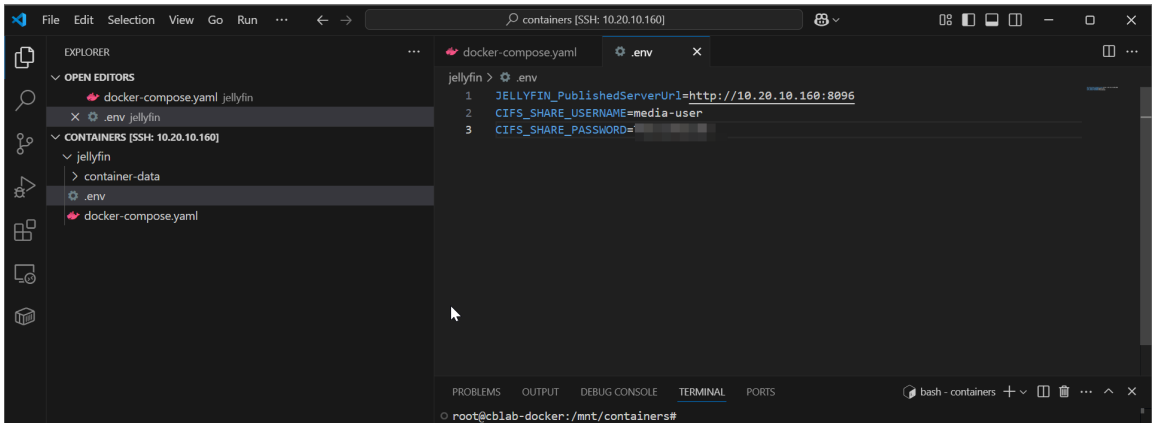
> **Info**
>
> It works even better if you start using a reverse proxy to provide TLS encryption. You can read an example of that in my caddy guide.

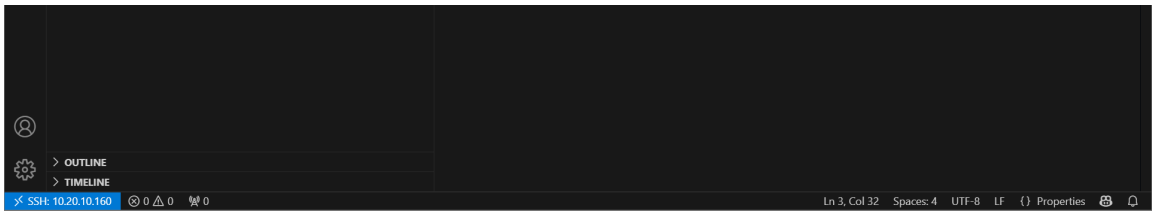
Mounting Network Storage

Let's say you want to have a media drive passed through to the container, because you... I don't know... followed this guide. The good news is that since we created a privileged container with CIFS capabilities, this is actually quite simple.

- Add your shared user's username and password to jellyfin's .env file:

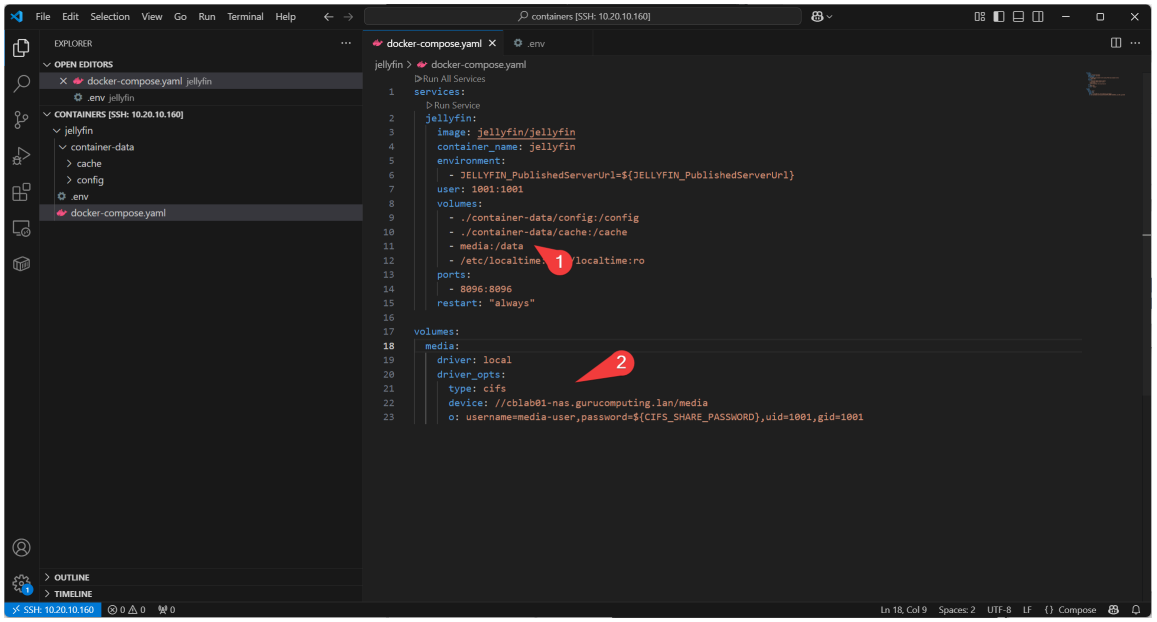
```
JELLYFIN_PublishedServerUrl=<your jellyfin url>
CIFS_SHARE_USERNAME=<your username>
CIFS_SHARE_PASSWORD=<your password>
```



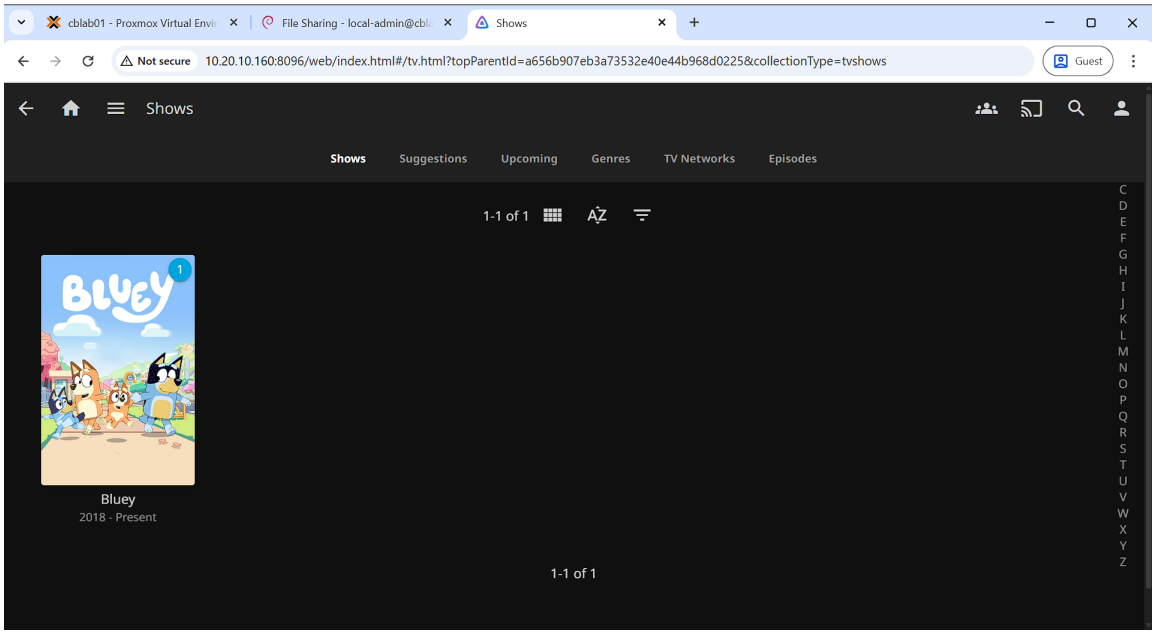


- Change your jellyfin docker compose to include the shared volume

```
services:
  jellyfin:
    image: jellyfin/jellyfin
    container_name: jellyfin
    environment:
      - JELLYFIN_PublishedServerUrl=${JELLYFIN_PublishedServerUrl}
    user: 1001:1001
    volumes:
      - ./container-data/config:/config
      - ./container-data/cache:/cache
      - media:/data
      - /etc/localtime:/etc/localtime:ro
    ports:
      - 8096:8096
    restart: "always"

volumes:
  media:
    driver: local
    driver_opts:
      type: cifs
      device: //cblab01-nas.gurucomputing.lan/media
      o: username=media-user,password=${CIFS_SHARE_PASSWORD},uid=1001,gid=1001
```



- restart the container with `docker-compose up -d` and confirm you can get to the shared volume.



GPU Passthrough

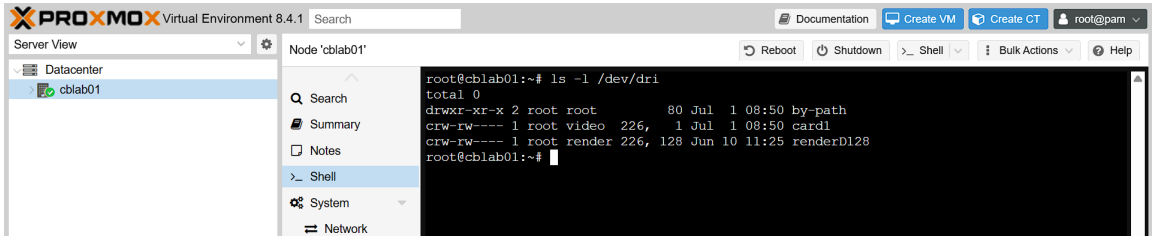
One of the trickier problems compared to baremetal is leveraging GPU acceleration for transcoding media, for services like jellyfin. This used to be especially difficult before proxmox 8, but these days it is relatively easy to perform.

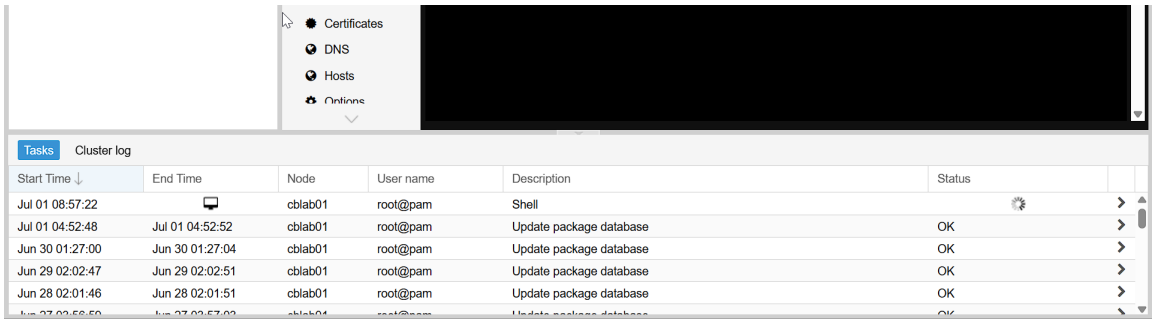
> **Info**
>
> This guide will assume you are wanting to utilise an **intel** GPU. Nvidia GPUs are significantly more challenging to get going, baremetal or otherwise.

- Find the **gpu** path in the proxmox host.

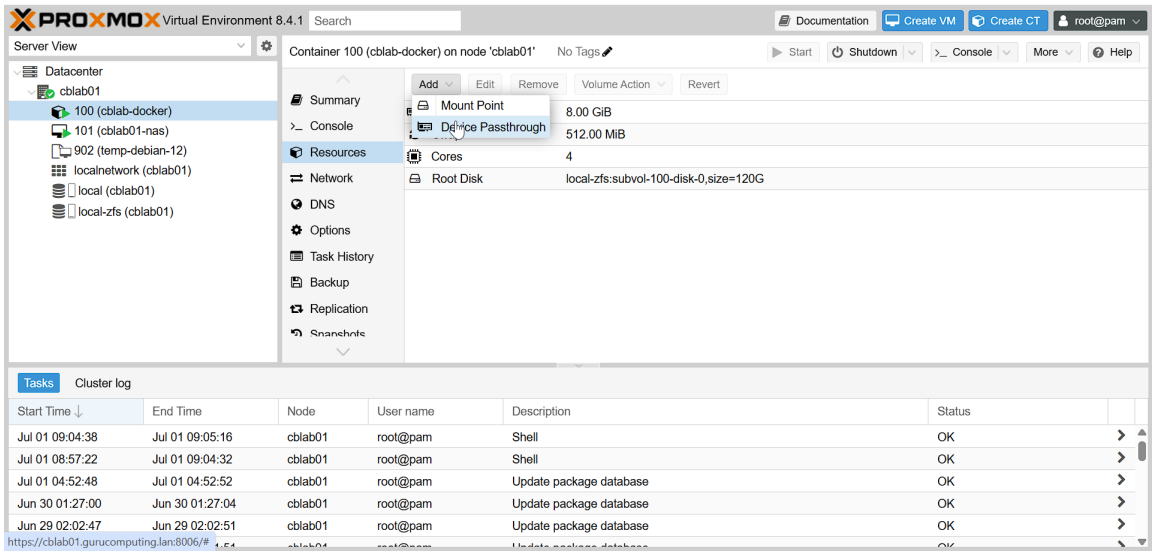
```
ls -l /dev/dri
```

It will almost certainly be `/dev/dri/renderD128`

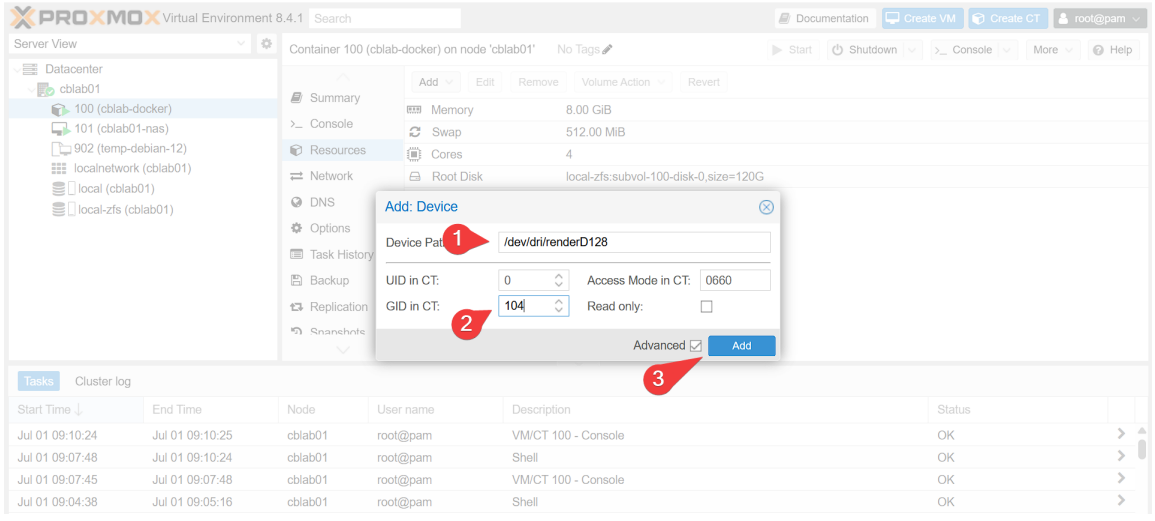




- If you need to confirm which GPU is which, you can check the render → driver mapping using `ls -l /sys/class/drm/renderD*/device/driver`
- In the **container settings**, add a **resource** for device passthrough



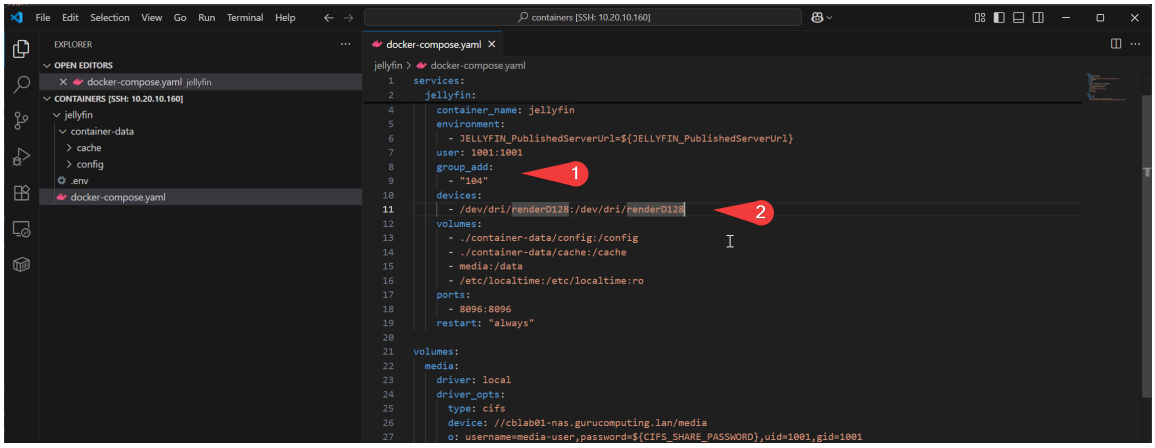
- Add the `/dev/dri/renderD128` (mind the capitalisation). Under **advanced** settings, set the same user and group (`104` for render group and `0` for user). **shutdown the container** and turn back on.



- fix up your jellyfin docker compose to take advantage of the newly available gpu

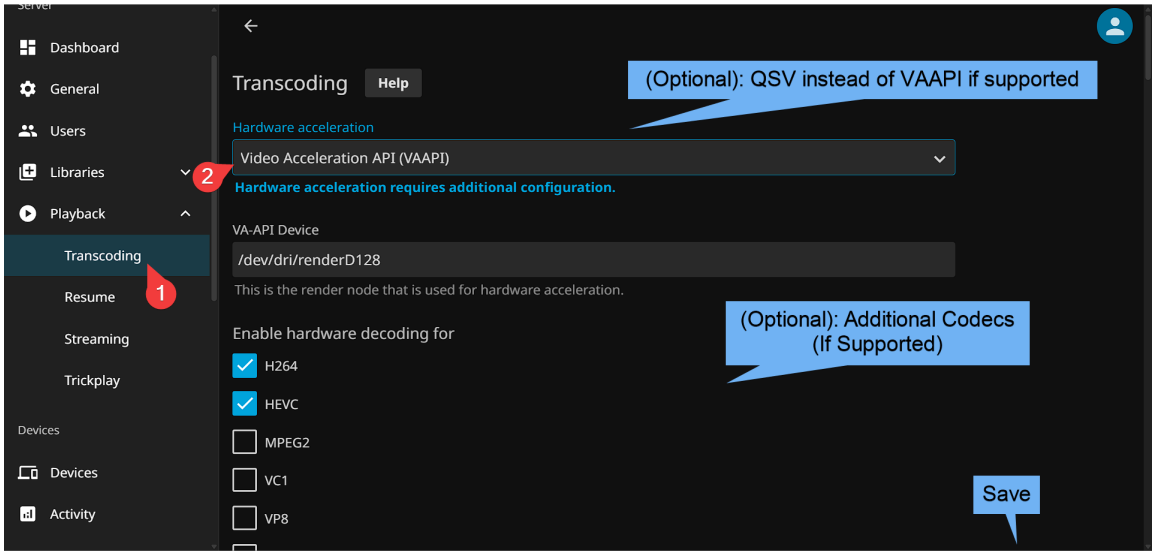
> **Warning**
>
> You **cannot** use the linuxserver version of jellyfin with the **user** and **group_add** docker compose directives (among many other problems with the linuxserver images). Given that jellyfin has an official image, we are assuming you will use the official image.

```
services:
  jellyfin:
    image: jellyfin/jellyfin
    container_name: jellyfin
    environment:
      - JELLYFIN_PublishedServerUrl=${JELLYFIN_PublishedServerUrl}
    user: 1001:1001
    group_add:
      - "104"
    devices:
      - /dev/dri/renderD128:/dev/dri/renderD128
    volumes:
      - ./container-data/config:/config
      - ./container-data/cache:/cache
      - media:/data
      - /etc/localtime:/etc/localtime:ro
    ports:
      - 8096:8096
    restart: "always"

volumes:
  media:
    driver: local
    driver_opts:
      type: cifs
      device: //cblab01-nas.gurucomputing.lan/media
      o: username=media-user,password=${CIFS_SHARE_PASSWORD},uid=1001,gid=1001
```

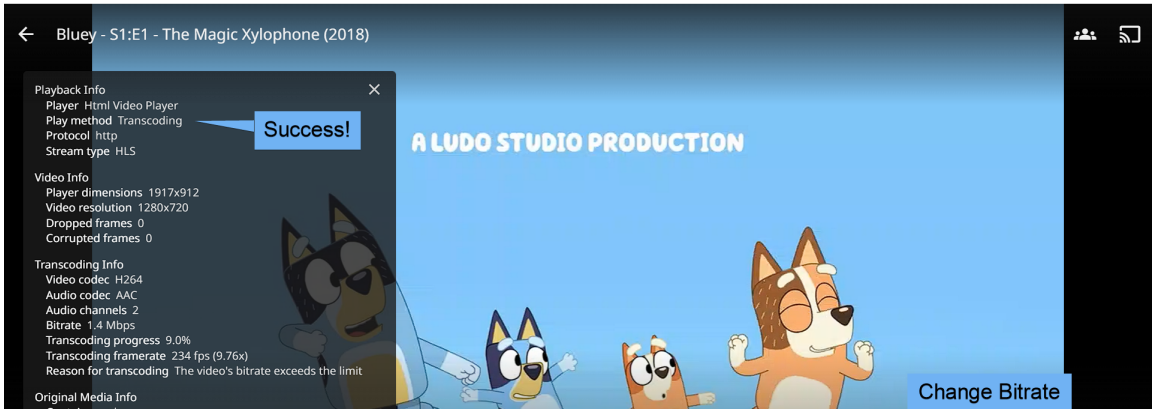


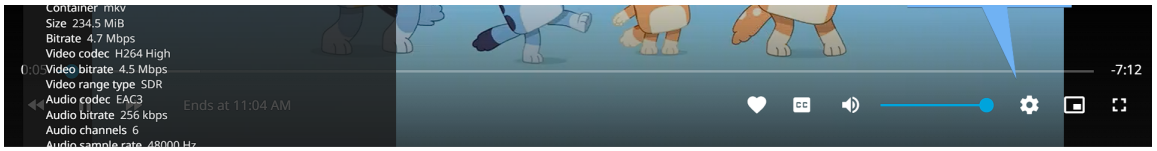
- Bring up with `docker compose up -d` and turn on hardware acceleration in jellyfin. We are defaulting to **VAAPI**, but if your GPU supports it, you can use **Quicksync/QSV** instead for some performance gain.

> **Info**
>
> What you can or cannot hardware accelerate depends heavily on what generation GPU you are using. You can see a full list here. H264/265 decoding is widely supported, encoding and other options less so.



- Bring up a video to play, change the bitrate, and (crossing fingers) see hardware accelerated transcoding!





Conclusion

We have now successfully set up a working docker environment within a proxmox LXC container! This can give us the benefits of both proxmox (such as proxmox backup server!) *and* a working docker environment, with all of the benefits that entitles. Fantastic!