# Practice M5: Virtualization and Containerization (openSUSE)

This practice assumes that you are working in an on-premise environment that allows for nested virtualization

All tasks can be achieved under different configurations (host OS and/or virtualization solution) with the appropriate adjustments

This practice is oriented towards **openSUSE Leap 15.3**

## Part 1: Virtualization with KVM

For this part we will need an infrastructure like this one:

Graphical user interface

Description automatically generated with medium confidence

The desktop machine is for testing the remote administration of KVM, it can be skipped

The server machine should be able to interact with the virtualization instructions of the CPU and should have enough memory (for example 4 GB) and disk space (either by extending the existing one, or by adding a new one)

Network settings are the ones, used during the demonstration. They must be adjusted according to your setup

Preparation

If your system is using **wicked** as network management solution, switch to **Network Manager**, it will make things easier for this part

One way to do this is to log on the machine using the console (not remotely) and then execute the following steps

Install **Network Manager** packages

**sudo zipper install NetworkManager**

Disable the **wicked** services

**sudo systemctl disable --now wicked**

**sudo systemctl disable --now wicked**

Enable the Network Manager service

**sudo systemctl enable --now NetworkManager**

Using the **nmcli** or **nmtui** commands, adjust both the connections and their settings (IP addresses, masks, etc.)

As a result of the above, we will assume that we have two connections – **eth0** and **eth1** (each named after the device) and only the **eht0** has **static address** (as shown on the picture)

### Install KVM

Log on to the **SERVER** machine with account that has administrative privileges

Let’s first check if our machine supports hardware assisted virtualization by executing:

**egrep -cwo 'vmx|svm' /proc/cpuinfo**

If the result is a number bigger than **0** we can safely continue. Otherwise, we must check in case of physical machine, if the processor supports virtualization and if it is enabled in BIOS, or in case of a virtual machine, if the virtualization instructions are passed on (nested virtualization)

Of course, there many other ways to do the same check

Now, let’s install the necessary binaries

**sudo zypper install libvirt qemu-kvm virt-install**

The system may need a reboot. If so, do it

Check if the **libvirtd** daemon is running

**systemctl status libvirtd**

And if not, start it and enable it to start on boot

**sudo systemctl enable --now libvirtd**

Now, we have **KVM** installed and ready

### Configure KVM networking (default / NAT)

Once the service is started, a default network is created

We can see it if we ask for information about our networks with

**ip a**

There should be one named **virbr0**

If not, let’s check if there is network configuration stored in **/etc/libvirt/qemu/networks/default.xml**

There should be one but most probably it is not marked for auto-start. Let’s check with

**sudo ls -al /etc/libvirt/qemu/networks/autostart**

If it returns no files or symbolic links, this means that there isn’t any network set to auto-start

Let’s first examine the network configuration with

**sudo vi /etc/libvirt/qemu/networks/default.xml**

We can see what the configuration looks like and the DHCP range of addresses

We should not modify this file directly. We should use **virsh** instead

Let’s create a symbolic link and restart the **libvirtd** service

**sudo ln -s /etc/libvirt/qemu/networks/default.xml /etc/libvirt/qemu/networks/autostart/default.xml**

**sudo systemctl restart libvirtd**

Lastly, if we go to the folder where the usual network scripts reside, we won’t find this virtual network, as it is managed by **libvirt**

Depending on the distribution and package selection during the installation, we could end up with such a network, even if we do not plan to use virtualization

Let’s use the appropriate utility to query and manage virtual networks

For example, in order to get list of all virtual networks, we can execute:

**sudo virsh net-list**

We can see that the **default** network is currently **active** and it is configured as **autostart**

If the default network is not there, we can ask for all networks with

**sudo virsh net-list --all**

And if it appears as **inactive**, start it with

**sudo virsh net-start default**

The **virsh** tool provides up with a huge amount of commands

**virsh help**

We can even use the tool in an interactive manner. In order to do it, just execute

**sudo virsh**

Now, that we are in, we can ask for the supported commands by typing **help**

One of the commands, we know already. Let’s ask for the list of virtual networks with

**net-list**

Here, we can use the **Tab** key to have auto-completion. For example, we can type **net-** and hit the **Tab** key

Let’s complete the command to **net-info**

Then, if we hit the **Tab** key two more times, we will end up with

**net-info --network default**

And if we press enter, we will get information about our **default** virtual network

You may have noticed that there is a **net-start** command but not a **net-stop** command

The opposite of **net-start** is **net-destroy**

Let’s stop our **default** virtual network with

**net-destroy default**

Now, exit the utility with **quit** and ask for network related information with **ip a**

We should not see the **default** virtual network listed. The reason for this is that we stopped it a second ago

Let’s check if the configuration is still there

**sudo ls -al /etc/libvirt/qemu/networks**

Oh, it is still there

Let’s start it again with

**sudo virsh net-start default**

Should we want to delete it, we must use the **net-undefine** command

### Configure KVM networking (host only)

Okay, let’s create a host only network with DHCP capabilities

For this purpose, we will need a network configuration to start with. The easiest way to get one is to dump the one of the **default** network with:

**sudo virsh net-dumpxml --network default > hostonly.xml**

Now, open it for editing with:

**vi hostonly.xml**

Change to match the following:

<network>

<name>**hostonly**</name>

<bridge name=**'virbr2'** stp='on' delay='0'/>

<mac address=**'52:54:00:b9:42:98'**/>

<ip address=**'192.168.100.1'** netmask='255.255.255.0'>

<dhcp>

<range start=**'192.168.100.2'** end=**'192.168.100.254'**/>

</dhcp>

</ip>

</network>

You should pay particular attention to the MAC address

Now, let’s apply the configuration

**sudo virsh net-define --file hostonly.xml**

Let’s check the networks with

**sudo virsh net-list**

It is not there as this command shows only active networks. Instead, we must execute:

**sudo virsh net-list --inactive**

There it is. Let’s mark it for **autostart**

**sudo virsh net-autostart --network hostonly**

And start it

**sudo virsh net-start --network hostonly**

We can check both with **net-list** and **ip a**

### Configure KVM networking (external)

In order to execute successfully the next steps, your **SERVER** machine must be equipped with two network adapters connected to your LAN

Assuming that the second interface is **eth1**, let’s execute **nmcli conn** to get list of connections

If there is a connection named **Wired connection 1** or something else pointing to the second adapter, execute the following (or similar statement) to delete it

**sudo nmcli conn delete "Wired connection 1"**

Now, let’s add a new bridge

**sudo nmcli conn add type bridge ifname virbr1 autoconnect yes con-name virbr1 stp off**

Let’s attach the network adapter to the bridge

**sudo nmcli conn add type bridge-slave autoconnect yes con-name eth1 ifname eth1 master virbr1**

Now, restart both **Network Manager** and **libvirtd**

**sudo systemctl restart NetworkManager**

**sudo systemctl restart libvirtd**

Please note, that this interface is not manageable by **libvirt**

### Create a VM in KVM (locally)

For this exercise and the ones that follow, we must have an install media deployed in **/var/lib/libvirt/images/**

Let’s assume that we copied the **Alpine Linux** image under the stated folder

Before we continue, we must install one additional package – **virt-viewer**

**sudo zypper install virt-viewer**

Now, in a terminal session type:

**sudo virt-install --hvm --name=vm-alpine --vcpus=1 --ram=512 --os-type=linux --os-variant=alpinelinux3.8 --disk path=/var/lib/libvirt/images/vm-alpine.qcow2,size=8 --cdrom=/var/lib/libvirt/images/alpine.iso --network=bridge:virbr0 --graphics spice --noautoconsole**

This will create a virtual machine with 1 vCPU, 512 MB RAM, and 8 GB HDD. It will be connected to the default virtual network

If you wonder how we came up with the **os-variant** value and what are the possible values, you can check with

**osinfo-query os**

Should we want a thin provisioned hard disk, we can create it upfront with

**sudo qemu-img create -f qcow2 /var/lib/libvirt/images/thin-disk.qcow2 -o size=12G,preallocation=off**

Then, we can attach it to a machine. We should bear in mind the performance implications

Okay, but how we can interact with the installation process?

While still on the terminal, execute the following to find out

**sudo virsh domdisplay --domain vm-alpine**

Now, open a browser and navigate to **spice://127.0.0.1:5900**

Confirm that you want to use the **Remote Viewer** application

You can follow the steps if you want to complete the installation process

Otherwise, you can return to the terminal and (forcibly) stop the VM. First, get the list of VMs

**sudo virsh list**

Now, stop the VM

**sudo virsh destroy vm-alpine**

Then, remove the VM with

**sudo virsh undefine vm-alpine**

And finally, delete the virtual disk

**sudo rm -f /var/lib/libvirt/images/vm-alpine.qcow2**

Or execute this command to combine the above two commands

**sudo virsh undefine vm-alpine --remove-all-storage**

### Create a VM in KVM (local CLI, but access it remotely)

In order to allow remote access to the virtual machine’s console, you must change the creation command

**sudo virt-install --hvm --name=vm-alpine1 --vcpus=1 --ram=512 --os-type=linux --os-variant=alpinelinux3.8 --disk path=/var/lib/libvirt/images/vm-alpine1.qcow2,size=8 --cdrom=/var/lib/libvirt/images/alpine.iso --network=bridge:virbr0 --graphics spice,listen=0.0.0.0 --noautoconsole**

Now, we must ask what the connection string is with

**sudo virsh domdisplay vm-alpine1**

Once, we have it, we can continue our interaction with the virtual machine from another host

Please have in mind that you may need to adjust the firewall on the **KVM** host

### Access a VM from Windows client

First, you need to set a machine for remote access

Second, you must download and install the **SPICE** client. It can be downloaded from here:

<https://virt-manager.org/download/sources/virt-viewer/virt-viewer-x64-10.0-1.0.msi>

### Working with KVM (remote CLI)

Okay, so far, we worked directly on the KVM server. What if we want to control it remotely?

Log on to the **DESKTOP** machine

Open a terminal session and install the CLI tool

**sudo zypper install libvirt-client**

So far, when we worked on the **KVM** host, we used **virsh** without any parameters, like

**virsh**

Basically, it is the same as if we executed it like

**virsh -c qemu:///system**

In order to access a KVM machine remotely, we must extend the command to

**virsh -c qemu+ssh://<username>@<kvm-ip-address>/system**

Once, we establish a session, we can continue interactively with the regular commands

Alternatively, instead of the interactive shell, we can put commands on the command line

**virsh -c qemu+ssh://<username>@<kvm-ip-address>/system <command>**

Before we continues, let’s install one more package

**sudo dnf install virt-install**

Using the new connection approach, we can extend the command, we used to create our virtual machine to

**virt-install --connect qemu+ssh://<username>@<kvm-ip-address>/system --hvm --name=vm-alpine2 --vcpus=1 --ram=512 --os-type=linux --os-variant=alpinelinux3.8 --disk path=/var/lib/libvirt/images/vm-alpine2.qcow2,size=8 --cdrom=/var/lib/libvirt/images/alpine.iso --network=bridge:virbr0 --graphics spice,listen=0.0.0.0 --noautoconsole**

Next, we can ask what the final connection string is

**virsh -c qemu+ssh://<username>@<kvm-ip-address>/system domdisplay vm-alpine2**

Using the received string, we can connect and work with the machine

Please have in mind that you may need to adjust the firewall on the **KVM** host

### Working with KVM (remote GUI)

While still on the **DESKTOP** machine, open a terminal and install both **Virtual Machine Manager** and **OpenSSH AskPass** component

**sudo zypper install virt-manager openssh-askpass**

Should we need it, we can add the **Remote Viewer** (**virt-viewer**) as well

Open the **Virtual Manager** application

Click on **File** and then **Add Connection** to create a new connection to our **KVM** server

Make sure that the **QEMU/KVM** option is selected in the **Hypervisor** drop-down list

Select the **Connect to remote host over SSH** option

Enter the **KVM** host IP address in the **Hostname** field

Click on the **Connect** button

If this is the first time you are connecting to the host via SSH, enter **Yes** in the dialog window and click **OK**

Enter the passphrase and click **OK**

Now, we should have a connection to our **KVM** host

Here, we can do everything a lot easier

First, let’s explore a bit

Select the connection and click on **Details** in the context menu

Explore the **Overview**, **Virtual Networks**, and **Storage** tabs

Close the **Connection Details** window

Now, let’s create a VM

Click on **File** and then **New Virtual Machine**

Check that the correct **KVM** host is selected in the **Connection** drop-down list

Make sure that the **Local install media (ISO image or CDROM)** option is selected

Click on **Forward**

Click on **Browse** to select an install media and click **Choose Volume**

If the operating system is not recognized, select the closest one

Click on **Forward**

Adjust the **Memory** and **CPU** settings

Click on **Forward**

Adjust the virtual hard disk size

Click on **Forward**

Enter the desired **name**

Select the **Customize configuration before install** option

Expand **Network selection** to explore the network connection

Finally, click **Finish**

Explore the settings (change the boot order and where the SPICE will listen) and once done, click the **Begin installation** button

The machine will start and will guide you through the installation process

If by any chance, you cannot connect and see the console, refer to the next section

### Manage VMs

While still on the **DESKTOP** machine, open the **Virtual Machine Manager** if not open

Connect to our **KVM** host

Select a running machine and invoke its context menu

In the **Shut Down** section we have (graceful) **Reboot** and **Shut Down** and (forced) **Force Reset** and **Force Off**

While the machine is powered on, we can **migrate** it to another **KVM** host

We can **delete** it while running (it will be forced to turn off) together with its disk files

Let’s click on **Open** option in the context menu

There, from the **View** menu, you can switch between the VM’s **console**, **details** (settings), and **snapshots**

By default, the **console** mode is loaded

Should you need to adjust for example, where the console is available, or the boot order, or any other setting, this is the mode you are interested in

Here, we can add or remove hardware as well

Let’s add one additional hard disk and remove the virtual CD drive

### Clone and snapshot VMs

While still on the **DESKTOP** machine, open the **Virtual Machine Manager** if not open

Connect to our **KVM** host

Select a machine (no matter is it running or not) and invoke its context menu

Open the virtual machine and switch to **Snapshots**

Click the plus button on the bottom-left to create a snapshot

With a VM powered off, invoke its context menu

Select **Clone**

Adjust the **Name**, **Networking**, and **Storage** options and click **Clone**

Don’t forget to adjust OS settings in the clone

Power on the new machine and adjust its OS settings like **hostname**, **static IP address** (if any), **password**, and etc.

### Sysprep & Customize

The process of adjusting the guest OS settings can be automated by applying the so-called **system preparation** or **sysprep**

For this purpose, we must install an additional package

**sudo zypper install guestfs-tools**

For example, we can set the root user’s password of our VM

**sudo virt-sysprep -v -x -d <our-vm-name> --root-password password:12345**

In a similar fashion, we can customize our VM for example by adding packages

For this, we must use the **customizer** tool

For example, let’s add the **htop** and **lynx** packages

**sudo virt-customize -v -x -d <our-vm-name> --install htop,lynx**

Now, if we power of the machine, we will see that both the password has been changed and the packages are available

Please note, that **Windows** guests are not (fully) supported currently by both tools

### Monitor KVM

We can install **top** like utility in order to monitor the VMs

Let’s install it on the **KVM** host

**sudo zypper install virt-top**

Now that we have it, start it and explore it

Return on the **DESKTOP** machine and start the **Virtual Machine Manager** if not started

Connect to our **KVM** host

Select the connection and click on **Edit** and then **Preferences**

Switch to **Polling**

Select the parameters you want to monitor

Adjust the polling interval

Close the dialog box

Then, open the **View** menu and select the charts you want to see

If you open a connection a machine and switch to **Details** and then to **Performance**, now you will see more detailed information

### Manage KVM VMs with Cockpit

*Please note, that currently, cockpit is supported only on* ***openSUSE Tumbleweed***

Log on to the **SERVER** and open a terminal session

Install both **Cockpit** and one of its modules

**sudo zypper install cockpit cockpit-machines**

Start cockpit and configure it for auto start

**sudo systemctl enable --now cockpit.socket**

Check if there is an exception added for cockpit in the firewall

**sudo firewall-cmd --list-service**

And if not, add it. The default port is **9090/tcp** or add the **cockpit** service

Open a browser and navigate to **https://<kvm-host-ip>:9090/**

Enter your credentials

Navigate to **Virtual Machines** section and explore what can you do

## Part 2: Containerization with LXC

In this part we will experiment with **Linux Containers** or **LXC**

We will need just a one machine. We can create a new one, or reset (if you made a snapshot) the **SERVER** VM

Graphical user interface

Description automatically generated

### Install and configure LXC

Open a terminal session and execute the following to install **LXC**

**sudo zypper install lxc**

Let’s start the daemon

**sudo systemctl enable --now lxc**

Now, we can check the configuration with

**sudo lxc-checkconfig**

Everything should be fine, only green labels

We can explore the **templates** folder

**ls -l /usr/share/lxc/templates/**

There are not so many items here. The template that we will use mostly is the **download** one

### Set up LXC networking

There are two options for **LXC** networking – to use networking components, managed by **libvirt** or let **lxc** manage all by itself

Let’s configure **lxc** to handle the networking

First, we must install **dnsmasq**

**sudo zypper install dnsmasq**

Open the default configuration file

**sudo vi /etc/default/lxc**

Set the **USE\_LXC\_BRIDGE** directive (close to the end of the file) to **true**

Save the file and close it

Restart both **lxc-net** and **lxc** services

Now, if you execute **ip a** command, you will see **lxcbr0**

We, must adjust one more configuration file

**sudo vi /etc/lxc/default.conf**

Make sure that the following is present

**lxc.net.0.type = veth**

**lxc.net.0.link = lxcbr0**

**lxc.net.0.flags = up**

Save the file and restart **LXC** daemon

### Create containers

Let’s create our first container

We should know in advance what are our options for **distribution**, **release**, and **architecture**

We can check the available templates here: <https://images.linuxcontainers.org/>

Suppose that we don’t know this, and thus we want to create a container in an interactive fashion

The following command will do the job

**sudo lxc-create -t download -n con1**

If we get notified with an error that states "**ERROR: Unable to fetch GPG key from keyserver**" then we must modify the above command to

**sudo lxc-create -t download -n con1 -- --no-validate**

We will be presented with all the available options

Let’s select **alpine** for **distribution**, **3.14** for **release**, and **amd64** for **architecture**

After a while, the container will be available on our machine

Now, that we know what options we have, let’s create second container of the same type, but this time using

**sudo lxc-create --template download --name con2 -- --no-validate --dist alpine --release 3.14 --arch amd64**

As it appears, as usual, we have two ways to specify options – short and long

### Work with containers

Let’s start our first container, execute

**sudo lxc-start --name con1**

In order to check what containers, exist on our system and what is their state, we can execute

We can modify a little bit the output. Let’s execute

**sudo lxc-ls --fancy**

Now, we can see which one is running, what is its IP address, and so on

Okay, now that we know the IP address of our container, let’s try to create a ssh session

No success. It is refused

Okay, let’s establish a console session to the container

**sudo lxc-attach --name con1**

Now, that we are in, we can execute any arbitrary command like **ip**, **uname**, **hostnamectl**, etc.

Let’s test if we have internet connectivity from inside

**ping softuni.org**

Everything seems to be fine

Can we create files? Let’s try. Create a simple **/readme.txt** file

**echo "Hello world!" > /readme.txt**

Check if the file is there and we can read its content

It appears that all is good so far

Let’s check if there is an **OpenSSH** service installed and running

**rc-service sshd describe**

Okay. Let’s install one

**apk add openssh**

Let’s start it now

**rc-service sshd start**

**rc-update add sshd**

And close the connection to the container

**exit**

Didn’t we destroy the container? Let’s check

**sudo lxc-ls --fancy**

Let’s try to establish a ssh session

**ssh root@<container-ip>**

It appears that we don’t know the password …

Of course, we can establish new session to the container and set one, but let’s explore another approach

### Explore containers

Let’s examine the container closer. Ask for some system information for the container

**sudo lxc-info --name con1**

Okay, we saw some information, but where is stored this container?

Let’s check what configuration options we have

**sudo lxc-config -l**

Hm, let’s ask for the value of **lxc.lxcpath**

**sudo lxc-config lxc.lxcpath**

Now, let’s visit this folder and explore its content

**ls -al /var/lib/lxc**

It appears that every container of ours has own folder here

Let’s check what is the content of the **con1** folder

**sudo ls -al /var/lib/lxc/con1**

There is a file named **config** and another folder **rootfs**

There should not by a surprise that the **config** file contains the configuration for the container

**sudo cat /var/lib/lxc/con1/config**

Let’s check the folder

**sudo ls -al /var/lib/lxc/con1/rootfs**

We can even see our **readme.txt** file. Can we read it?

**sudo cat /var/lib/lxc/con1/rootfs/readme.txt**

Sure, we can. But can we write to it as well?

**echo "Changed by the host" | sudo tee -a /var/lib/lxc/con1/rootfs/readme.txt**

Obviously, we can

Can we change or set the **root** password for the container?

We can use the **chroot** technique to set the password

**sudo chroot /var/lib/lxc/con1/rootfs/ passwd**

Now, that we have a password, let’s try to establish a SSH session

**ssh root@<container-ip>**

No luck again. Maybe it is forbidden for the root to use password authentication via SSH

Let’s attach back to the container

**sudo lxc-attach --name con1**

Check and modify the SSH configuration

**vi /etc/ssh/sshd\_config**

Add the following, save and close the file

**PermitRootLogin yes**

Restart the service

**rc-service sshd restart**

Exit the container

**exit**

Try again

**ssh root@<container-ip>**

Voila, finally we can establish SSH session between the host and the container

While we are here, let’s check out the **/readme.txt** file

All changes are there

Close the connection with **exit**

### Stop, start, auto-start

We can stop a container from inside or from outside

If we have not established a session to the container, do it

Let’s stop our container from the inside. Just execute the regular

**poweroff**

And bam, the container is stopped. We can check with

**sudo lxc-ls --fancy**

Let’s start it again

**sudo lxc-start --name con1**

Let’s test alternative use of the **lxc-attach** command

**sudo lxc-attach --name con1 -- uname -a**

So, if we can pass commands this way, maybe we can turn it off as well. Let’s try

**sudo lxc-attach --name con1 -- poweroff**

It appears that we can turn off a container this way as well

Let’s start it again

**sudo lxc-start --name con1**

And try alternative approach for stopping a container

**sudo lxc-stop --name con1**

Should we want to kill it instead of stopping it gently, we can execute

**sudo lxc-stop --name con1 --kill**

If we substitute the **--kill** switch with **--reboot**, we will restart the container

We can configure a container to auto-start by setting the **lxc.start.auto** parameter to **1** in the container’s configuration file

**sudo vi /var/lib/lxc/con1/config**

Then we can ask all containers that are set to auto-start to start with

**sudo lxc-autostart**

### Clones and linked clones

We can clone stopped containers

Let’s clone our **con2** container into **con3**

**sudo lxc-copy --name con2 --newname con3**

We can check the list of containers

**sudo lxc-ls --fancy**

Sometimes, we would need to create a test clone fast just for a simple test. We can think of this like some kind of a linked clone. Let’s create one

**sudo lxc-copy --name con2 --newname con2-test -B overlay -s**

It happens faster than the traditional cloning

Let’s check what happens in the back end with the storage. Compare folders of **con2** and **con2-test**

**sudo ls -al /var/lib/lxc/con2/**

**sudo ls -al /var/lib/lxc/con2-test/**

There is additional folder **delta0** which contains changes over the original data. Currently there should be only a few. For example, the hostname

**sudo ls -al /var/lib/lxc/con2-test/delta0/etc**

### Snapshots

We can create snapshot as well. Again, this should be done on **stopped containers** (so if the container is running, stop it first)

Let’s create a snapshot of our first container

Should we want to attach a comment, we must create a text file to hold it

**echo "snapshot 1" > snapshot-comment**

And then the snapshot itself

**sudo lxc-snapshot --name con1 --comment snapshot-comment**

We can get the list of snapshots of a container with

**sudo lxc-snapshot --name con1 --list**

In order to see the comments as well, we must extend to command to

**sudo lxc-snapshot --name con1 --list --showcomments**

We can restore or revert a container to a snapshot with

**sudo lxc-snapshot --name con1 --restore snap0**

Alternatively, we can restore a snapshot as a new container

**sudo lxc-snapshot --name con1 --restore snap0 --newname con1-snap**

### Monitoring

We can monitor what is happening with

**sudo lxc-top**

To close it, press the **Q** key

### Deleting containers

We can remove a container with

**sudo lxc-destroy --name con2-test**

If the container has snapshots, we can delete them together with the container

**sudo lxc-destroy --name con1 --snapshots**

Remove all other containers

### Install LXD

**LXD** is an improvement over the classic **LXC**

In order to install it, we must execute

**sudo zypper install lxd**

Add current user to the **LXD** group

**sudo usermod -aG lxd <user>**

Restart the **LXC** and start the **LXD** daemons

**sudo systemctl restart lxc**

**sudo systemctl enable --now lxd**

Switch to the **root** user

**sudo su**

Create the initial configuration of the **LXD**

**lxd init**

Follow the wizard (reuse the existing bridge) and once done, exit to the regular user. For storage backend select **dir**

Reboot the system or close your session and re-open it

Test if you can execute commands against **LXD**. Check the version of both the client and server

**lxc version**

Ask for detailed information

**lxc info**

### Bash command completion

It is available by default

### Create and work with containers

Let’s first try to search for images

**lxc image list images:**

Wow, the list is quite long. Let’s filter it a bit

**lxc image list images: | grep -i opensuse**

Okay, start an openSUSE Leap container with

**lxc launch images:opensuse/15.3/amd64 opensuse1**

And another one

**lxc launch images:opensuse/15.3/amd64 opensuse2**

Now, check what local images do we have

**lxc image list**

We can check the running containers with

**lxc list**

We can use the usual **stop**, **restart**, and **start** actions. For example, restart the **first** container and stop and then start the **second** one

**lxc restart opensuse1**

**lxc stop opensuse2 && lxc start opensuse2**

To get some more detailed information about a container, execute

**lxc info opensuse1**

We can execute commands in the containers from the host. For example

**lxc exec opensuse1 uname**

If we need to specify parameters, we must change the command to

**lxc exec opensuse1 -- uname -a**

Should we need to establish a session to the container and work there, we can do

**lxc exec opensuse1 bash**

From now on, we can execute commands as if this is a regular host

Install for example **NGINX** and start it and then close the session

Use **curl** on the host to check if the default **NGINX** page is accessible

### Delete containers

Stopped containers we can delete with

**lxc delete opensuse1**

Or if the container is running, we can modify the command to

**lxc delete --force opensuse1**

## Part 3: Containerization with Docker

Let’s check another approach on containerization

We will need just one machine, like in the previous part

Graphical user interface

Description automatically generated

### Prepare and install Docker

We can jump directly to install the Docker platform

**sudo zypper install docker**

Add our user to the **docker** group

**sudo usermod -aG docker <username>**

Close and re-open the session

Start and enable the service

**sudo systemctl enable --now docker**

### Pre-flight check

Check that **Docker CLI** is accessible and the **Docker** daemon reachable

Get some information about both the client and the daemon

**docker version**

Ask for system-wide information with

**docker info**

In the output of the second command we can see detailed information about the installation

If both return information and no errors, we can move on

### Hello container world

Now, that we have **Docker** up and running, let’s execute a simple hello world container

**docker run hello-world**

We should see a message that our installation is correctly set up

There is an additional example there, let’s execute it

**docker run -it ubuntu bash**

Wow, we end up in an Ubuntu container. It tastes exactly like a running Ubuntu

Explore a bit. Browse the filesystem, check information about the kernel, check the network connectivity, etc.

Once done, close the container session with

**exit**

### Searching for images

We cannot know all images by name, so we can do a search by a keyword

On the terminal this done like

**docker search opensuse**

Unfortunately, the output is not very informative

Alternatively, you can visit the famous **Docker Hub** at <https://hub.docker.com/search>

Let’s go there and ask again for **opensuse**

Now, visit the official repositories of openSUSE and explore different tags and read the instructions

### Download (install) an image

So far, we executed one **run** command on a brand-new installation without any images

What happed behind the scenes? **Docker**, knowing that we don’t have the image locally, went out, found one for us, downloaded it and then ran a container out of it

There is another approach. Knowing what we need, we can issue a special command to download the image locally

**docker pull opensuse/leap**

There is a default tag **latest** applied secretly if we do not specify on. So, the above command is in fact this one

**docker pull opensuse/leap:latest**

Once the image has been downloaded (it can take some time), we can check the list of our local images with

**docker image ls**

### Create a container

Let’s use a little bit modified version of the hello world **run** command

**docker run -it opensuse/leap bash**

This will start a container and will create an interactive terminal session with **BASH** to it

It looks and feels exactly like a real thing

Try a few commands

Once you are done execute the **exit** command

You will be returned in your host session

Now, let’s check what containers we have running with

**docker container ls**

It appears that there aren’t any. It is true, we just terminated the only one. When we executed **exit**, we closed the **BASH** session which was the only running process (in a way) in the container. So, we effectively stopped the container as well

If we modify the command a little bit to

**docker container ls -a**

We will see all containers running and stopped

Okay, so how we exit from container without terminating it? There is a key combination **Ctrl+P** and then **Ctrl+Q**. Try it and see yourself. First create a container with

**docker run -it --name con1 opensuse/leap bash**

Let’s check if our last container is still running

**docker container ls**

And now, how to return to the container? We can attach back to it. First, we need to know its ID or name. This can be seen from the output of the above command

But wait, we know the name because we set it to custom value

Now, to return to the second (still running) container, we must execute

**docker container attach <container-id-or-name>**

Exit the session again but do not terminate the container

### Exchange data

Let’s create a folder on the host. For example, **~/data**

Create a simple **readme.txt** file there with some content

Now, let’s create a container, but a modified **run** command

**docker run -it --name con2 --volume ~/data:/data opensuse/leap bash**

Being in the container, navigate to the file and check its content

Add something at the end of the file or add a new file

Exit (and terminate) the container

On the host, go and see if the changes persisted. They should be there

### Communicating with a container

Let’s run a **NGINX** container and expose its port to the host

**docker run --name nginx1 -d -p 8080:80 nginx**

We can check for running containers

**docker container ps**

Now, either using a browser or for example curl, we can check the following address

[**http://localhost:8080**](http://localhost:8080)

We should be able to access it even outside the host but first we must take care of the firewall

**sudo firewall-cmd --add-port 8080/tcp --permanent**

Reload the rules

**sudo firewall-cmd --reload**

Now, open a browser and navigate to

**http://<docker-host-ip>:8080**

Let’s stop the container and combine this technique with the data exchange one

**docker container stop nginx1**

Create a **~/html** folder

**mkdir html**

And put a simple **index.html** file in it

**echo "<h1>Hello from Docker</h1>" > ~/html/index.html**

Now, run a **NGINX** container, but with this command

**docker run --name nginx2 -d -p 8080:80 -v ~/html:/usr/share/nginx/html:ro nginx**

Check the result. Nice, a? 😉

### Build own container image #1 (from running container)

Spin up a container with

**docker container run --name con-templ -it ubuntu**

Add a **readme.txt** file in the root folder

**echo "Running container" > readme.txt**

Exit the container without stopping it (use the key combination)

Create image out of the running container with

**docker container commit con-templ con-image**

Check that indeed we have created a new image

**docker image ls**

Let’s try to create a container out of our own image

**docker container run -it con-image**

Check if the file is there

**cat readme.txt**

Close the connection and stop the container

**exit**

### Build own container image #2 (from file)

Let’s create our own image from (almost) scratch

Create **~/hello-nginx** folder

**mkdir ~/hello-nginx**

Create a **~/hello-nginx/Dockerfile** there

**vi ~/hello-nginx/Dockerfile**

Containing the following text

**FROM ubuntu**

**RUN apt-get update && apt-get install -y nginx**

**ADD index.html /var/www/html/index.html**

**ENTRYPOINT ["/usr/sbin/nginx","-g","daemon off;"]**

**EXPOSE 80**

Create **~/hello-nginx/index.html** by executing the following command

**echo "<h1>Hello from Docker</h1>" > ~/hello-nginx/index.html**

Enter the folder

**cd ~/hello-nginx**

And execute

**docker build -t hello-nginx .**

Now, start a container with

**docker run --name my-nginx -d -p 8000:80 hello-nginx**

Check the result

### Attach to a container

We can execute another process in a container that is running in detached mode

Let’s create a **BASH** shell in the last **NGINX** container we started

**docker container exec -it my-nginx bash**

We can do some stuff. For example, explore where our custom index is

**ls -al /var/www/html/index.html**

Once we are done, we can type **exit** to close the session

If we check for the running containers, we will see that it is still running

Let’s now start another container in interactive mode

**docker container run -it --name my-ubuntu ubuntu bash**

Detach from its session with **Ctrl+P** and then **Ctrl+Q**

Now, should you want to attach back, you can execute

**docker container attach my-ubuntu**

We know already, that if we type **exit**, we will terminate the container. Let’s do it

### Clean up containers and images

We can stop and then remove running containers or directly force their removal no matter if they are running or not

**docker container rm <container-id-or-name> --force**

After we remove the containers, we can remove the images as well

**docker image rm <image-id>:<tag>**

We can automate the whole process of container stopping and deleting

To stop all running containers, we can execute

**docker container stop $(docker container ls -q)**

We can remove all stopped containers with

**docker container prune**

Add **--force** at the end if you do not like to be asked

In order to remove all downloaded images, we can execute

**docker image rm $(docker image ls -q)**