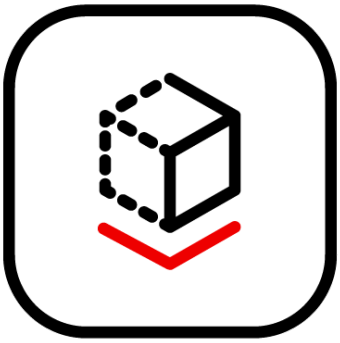
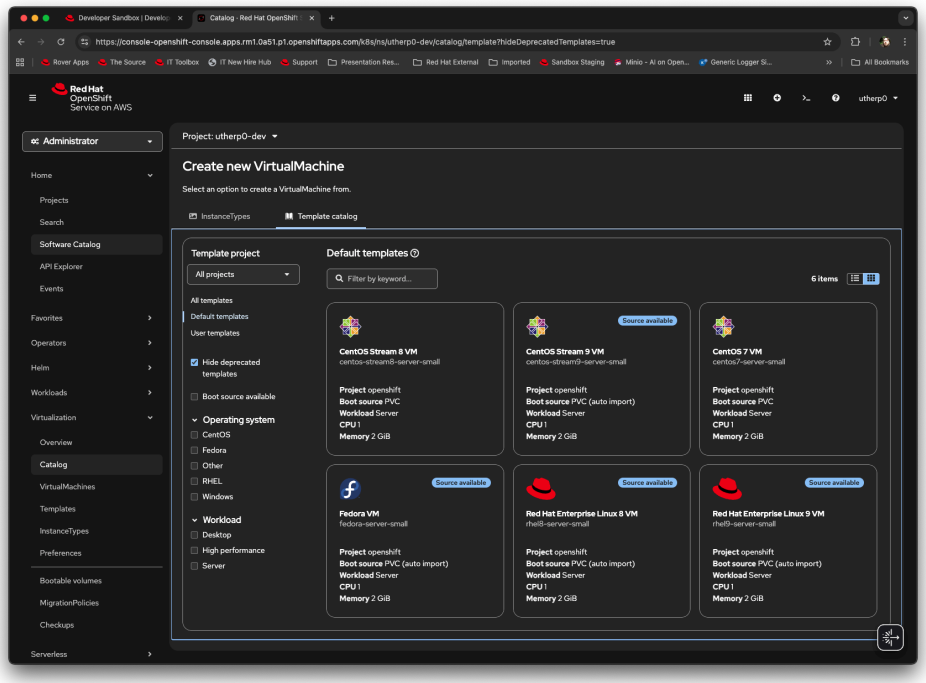


Hands-on Workshop (Sandbox based)
Containers Primer workshop with Podman on Fedora

Step 1 - SETUP

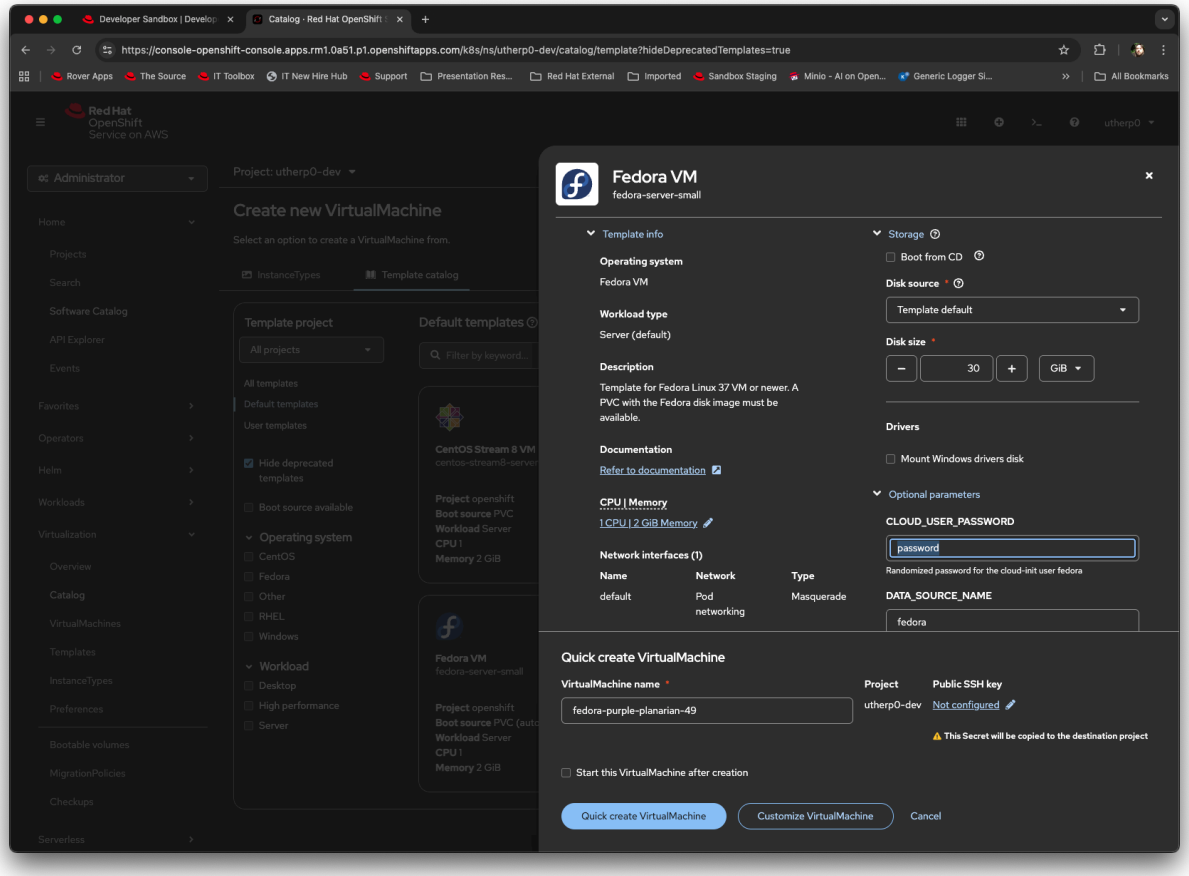
Log on to the Developer Sandbox (follow instructions on how to create an account if you don't already have one) - https://console.redhat.com/openshift/sandbox

Select **Virtualization/Catalog**, then **Template Catalog**.



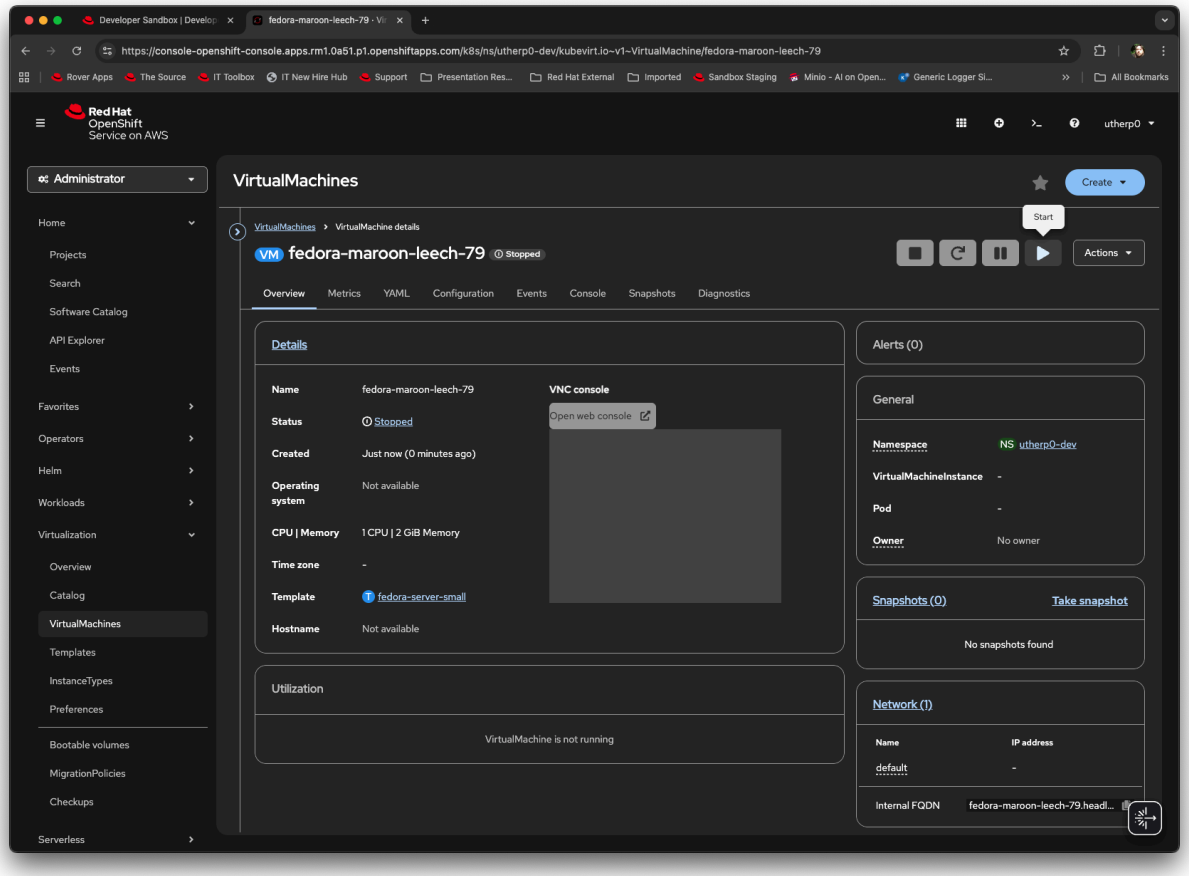
Click on the **Fedora** tile.

Expand the ‘Optional Parameters’ and change the CLOUD_USER_PASSWORD to ‘password’ (for simplicity *only*)



Click on ‘Quick Create VirtualMachine’

When the VM panel appears, click on the Start icon (as shown below):



Allow the VM to complete initialisation. Once it has finished the small VNC window will show four lines of static text.

Click on the ‘Open Web Console’ link above the VNC.

Switch back to the OpenShift Ux (shown above) and click on **Home/Projects** (see below for why you need to do this).

Switch the the VM console tab, log into the Fedora machine - username is ‘fedora’ and password is ‘password’ (these are shown at the top of the console for reference)

In the VM type:

```
None
sudo dnf install git -y
```

CAVEATS ABOUT SANDBOX USAGE

Sandbox is a shared and hosted system provided for playing with OpenShift. To avoid misuse it disconnects often and will idle workloads periodically. If your VM is idled the active containers will be removed; if that happens simply repeat the commands to create the containers and continue.

Also the VNC will disconnect periodically as well. In the exercises we will shift the focus of the OpenShift Ux away from the VM overview; this takes the VNC connection and will disconnect the web-based one if the page is left up (hence we change the OpenShift Ux to point at the welcome page before starting to interact with the VM).

INITIAL PODMAN TESTS

In the VM console type:

```
None
podman info
```

Look for the setting for ‘graphRoot’ - this is the filesystem where Podman maintains its local storage, images, file-layers, volumes etc.

```
None
podman pull quay.io/ilawson/ocpnode
```

This may take a minute or two, it is downloading the image (via the file layers) from the quay.io external registry to the Podman local storage.

```
None
podman images
```

This will show the current images available in the local Podman storage.

```
None
podman history ocpnode:latest
```

This will show the stored history for each of the file-layers that makes up the ocpnode:latest image.

BUILD AN IMAGE USING PODMAN AND CONTAINER FILES

In the VM console type:

```
None
git clone https://github.com/utherp0/containerlab
```

Hint - when interacting with files in Fedora/RHEL you can be lazy and hit [TAB] after giving the start of a command or word

Try:

```
None
cd co[TAB]
```

I.e. type co and then hit TAB, it will fill in the rest of the directory name

```
None
cd example1
podman build -t example1 -f ContainerFile
```

This will execute the commands specified in the ContainerFile in the example1 directory and build an image. To check the image has built, try:

```
None
podman images
podman history example1:latest
```

To see how we built this image, try:

```
None
cat ContainerFile
```

Now, to show how file layers work in the build, we will edit the ContainerFile:

```
None
vi ContainerFile
```

vi is an esoteric old editor that uses keystrokes for commands. Move the cursor under the line ‘RUN mkdir /var/www/html/images’ and press ‘i’ to enter INSERT mode

Add the following:

```
None

RUN echo "Hello!"
```

(with the linebreak for tidiness)

Hit [ESCAPE] and ‘wq!’ to write the file and quit the editor.

Now type ‘history’, find the number of the command we executed to do the build, and repeat it using ‘!(the number in the history)’ - saves typing.

Type:

```
None
podman history example1:latest
```

Notice the new layer added (it does nothing, but each command in the ContainerFile will generate a layer in the image definition).

Edit the ContainerFile again using ‘vi ContainerFile’. Move the cursor down to the ‘RUN echo’ line we added, and hit ‘dd’, which deletes this line. Again, hit [ESCAPE] and ‘wq!’ to write and exit the editor.

Now type:

```
None
podman build -t example:newbuild -f ContainerFile .
podman history example1:newbuild
```

Note it doesn't have the 'echo' layer any more, and also notice we have a new image and tag.

```
None
podman images
```

Note we have two separate Images with the same base name but different tags and IDs

RUNNING A CONTAINER

Type:

```
None
podman images | grep newbuild
```

Make sure the image exists. Now type:

```
None
podman run -dt -p 8080:8080/tcp example1:newbuild
podman ps
```

You will now have a running container (with a webserver inside as defined by the ContainerFile we used).

Now try this:

```
None
podman run -dt -p 8080:8080/tcp example1:newbuild
```

This will fail - we already have an active container listening on the 808 port *of the container runtime host.* Try this:

```
None
podman run -dt -p 8081:8080/tcp example:newbuild
podman ps
```

You should now have two active containers - both of them *think* they are running on port 8080, but in actuality the container runtime host is handling re-direction of the physical ports, in this case one is listening on 8080 physically, and the other is listening on 8081.

Try:

```
None
curl http://localhost:8080
curl http://localhost:8081
```

Both containers are offering a webserver, you should see a basic http source file provided from both URLs.

For observability, do:

```
None
podman ps
```

Make a mental note of the names of the containers. Now try:

```
None
podman top (the name of the first container)
podman logs (the name of the second container)
```

‘Top’ will give you the processes consuming the most resource from the first container. ‘Logs’ gives you the direct output of the second (probably just a single line about not being able to determine the DNS).

GETTING INTO A CONTAINER

Using one of the names of the containers, type:

```
None
podman exec -it (name) /bin/bash
```

You are now inside the running container. Most containers are built on reduced sets of the OS, so a lot of the ‘dangerous’ commands are not installed, such as ‘ps’ or ‘top’.

Type:

```
None
df -h
```

This shows you the disk systems mounted in the “OS” - remember that a container is just a file system with delusions of grandeur; it thinks it is a full OS.

Note the ‘overlay’ file system mapped to the root directory (“/”). This is the core filesystem generated by the container runtime host for the container to work within. This is important, as the following exercise shows.

DEMONSTRATING THE EPHEMERAL NATURE OF THE CONTAINER

Whilst inside the container (from the last exercise, repeat if you have exited), type:

```
None
cd /tmp
touch myfile.txt
ls -al
exit
```

The ‘ls’ should have shown that the file was created in the /tmp directory. Use ‘podman ps’ to get the port of the container you were logged into (i.e. the physical, whether you were logged into the 8080 or the 8081 physical container). Now, using the name of the container you just logged into, type:

```
None
podman stop (name)
```

This removes the container.

```
None
podman run -dt -p (port):8080/tcp example1:newbuild
```

Basically we are recreating the container just removed. Do a ‘podman ps’ to get the name of the new copy, and the type:

```
None
podman exec -it (name) /bin/bash
cd /tmp
ls -al
exit
```

No files. The container has been recreated using the immutable original image; the file we created previously no longer exists.

USING PODMAN VOLUMES TO RETAIN DATA

Type:

```
None
podman volume ls
```

Podman will have no volumes setup currently. To create one, called ‘vol1’, type:

```
None
podman volume create vol1
podman volume ls
```

Podman should now acknowledge you have a piece of shared storage you can use for your containers.

Now try:

```
None
podman run -it -v vol1:/myvolume example1:newbuild /bin/bash
```

Note that with -it rather than -dt this will create a container interactively, dropping you straight into it (using /bin/bash). Now type:

```
None
df -h
```

Note that we have an additional filesystem mounted into the container; this is actually the external file system offered via the volume. Now type:

```
None
cd /myvolume
ls -al
touch hello.txt
ls -al
exit
```

This will show the filesystem is empty, create a file called hello.txt, show it in the directory and then exit. When you exit, because the container was created interactively (-i) it will shutdown and be removed. Do a ‘podman ps’ to see that it has been deleted.

Now type:

```
None
podman run -dt -v vol1:/anothervolume example1:newbuild
```

This will create a running container without dropping you straight into it. Do a ‘podman ps’ to get the new container’s name, and then:

```
None
podman exec -it (name) /bin/bash
df -h
cd /anothervolume
```

```
ls
echo "Wibble" >> wibble.txt
exit
```

Note that the original file still exists in the physical volume.

USING VOLUMES TO TRANSFER FILES BETWEEN DIFFERENT CONTAINERS

Kill the active container (use ‘podman ps’ to get the name, and ‘podman stop (name)’ to remove it.

Now use ‘podman images’ and make sure you still have the ocpnode:latest image (from the early examples) in the local storage.

Now type:

```
None
podman run -dt -v vol1:/transfer ocpnode:latest
podman ps
podman exec -it (the name of the container) /bin/bash
df -h
cat /transfer/wibble.txt
```

This should demonstrate that files within the volume are retained, even when you map the volume to a different type of container.