Lab Orchestrator

Marco Schlicht Mohamed El Jemai

August 9, 2021

Abstract

This document is the project documentation and is intended, among other things, to describe and explain all aspects, such as tools and required knowledge, that are necessary to successfully complete the project.

The first chapter explains the motivation of the project and the goals we want to achive. There is also a division of the project into different project phases. In the second chapter the basics needed to understand this project are explained. There are different tools that are described and the key concepts of Kubernetes are explained. After that there are evaluations of which additional tools are required, for which further explanations are included. This contains a more detailed description of Kubernetes objects and KubeVirt, but also information about noVNC and ttyd, two tools which may be used to connect to the containers and VMs.

The project documentation accompanies the project and is continuously supplemented and expanded and should always reflect the current status of the project.

Contents

1 Introduction

1.1 Motivation

At the university, lecturers can simply provide their students with a VM in which the students can complete their assignments. In these VMs, software is pre-installed and pre-configured so that the students can, for example, directly start programming their microcontroller with the IDE provided. This has the advantage for the instructors that all students use the same system and therefore they only have to provide support for this system and do not have to worry about problems that vary from system to system. Also, the VM forms a sandbox and thus changes can be made to the system in this environment and if something breaks, a snapshot is taken or the original image is reinstalled.

However, the options here are limited to one VM and local deployment. It is not possible to simply start a whole network of VMs, nor is it possible to open these VMs in the browser.

1.2 Description

The Lab Orchestrator shall allow to start a network of virtualised systems (i.e. VMs and different types of containers) and make them accessible over the network. In the network of virtualised systems several virtualised systems shall run simultaneously and if a user has access to one of the systems it shall be possible to access others. Several such networks should be able to be started, so that users can work independently of each other. A user has a user session and in this session a network is started for this user, in which the user can work.

The access to the virtualised systems should be possible via an integration in the web. The user should be able to start a network on a web page and then get access to the graphical user interface or the terminal of the virtualised systems in a frame or HTML canvas on the web page.

Furthermore, in addition to the integration as a frame or canvas, it should be possible to optionally integrate a tutorial. These instructions should be able to contain several pages and several steps per page and teach the person working in the virtualised system. The instructions contain various features, such as steps that can be checked to see the progress and tutorial boxes that provide knowledge and explain certain parts. These tutorials are meant to extend the mere sandbox to be able to teach people something.

As virtualised systems, we want to support docker containers and classic VMs.

1.3 Target Groups

Target Groups:

- Universities
- Computer Science Clubs
- Companies
- IT Security Personal

- Learning Platforms
- Moodle

The software can be used in universities by lecturers to provide students with an environment in which they can learn and try out. On the one hand, it can be used parallel to lectures or practice sessions as a pure sandbox of a network in which students can do their exercises, and on the other hand, the tutorials can be integrated directly into the application.

Computer science clubs like the CCC often do Capture the Flag competitions. The program can make it easier to scale scenarios for competitions and learning.

Companies and private individuals can use the tool to map their internal IT environment and safely check their environment for security vulnerabilities in a sandbox. There is no need to consider any consequences for the live operation of the company.

IT security personnel also benefit from the sandbox environment and can be trained here or acquire their own knowledge. Although solutions such as Hack The Box already exist for this purpose, they cannot be hosted by the company itself and no instructions are available for them either.

Learning platforms can build on the program to create tutorials. Also an extension for Moodle, which is used by many universities, is conceivable, in which the courses of the application are integrated. One could use such a Moodle addon to work within Moodle in VMs and store tasks for students there.

1.4 Project Planning

The Lab Orchestrator is divided into several parts:

- Orchestrator of virtualised systems
- Accessible from the Web Base Images
- Lab Orchestrator Library
- REST-API

1.4.1 Orchestrator

Kubernetes is suitable as an orchestrator. Kubernetes allows you to launch a predefined set of Docker images. The images in a namespace can be connected to each other and ports can be opened to the outside. In Kubernetes' declarative YAML config, it is possible to define a set of Docker images, in addition to defining the ports opened to the outside and creating an internal network. This allows to access one container from another container. With such a config, it is easy to start and stop this set of containers as often as you want.

Kubernetes out of the box can only start containers. Here it is unclear whether the containers are sufficient to start graphical interfaces of Windows. The KubeVirt extension adds the function to use VMs instead of containers for this. KubeVirt can also be used to start a Windows VM with a graphical user interface. Kubernetes with KubeVirt therefore seems suitable as an orchestrator for the Lab Orchestrator .

1.4.2 Accessible from the Web Base Images

In order to be able to access the virtualised systems from the web and to make it as easy as possible to create your own virtualised systems, a technology must be found that makes it possible to access the terminal or the graphical interface of the virtualised system. This technology should then be provided in a template for example a docker base image or an virtual machine image both with the tool preinstalled.

For terminal access, there are already various tools, such as Gotty, wetty and ttyd. For desktop access, there is Apache Guacamole and noVNC. It is necessary to evaluate which of these tools are the most suitable and then install and configure these tools in the base image.

The goal of this step is to have an runnable image where the graphical interface and the terminal of the VM or Docker container can be accessed via the web.

Then, this image must be included in a Kubernetes template so that a network of such virtualised systems can be launched in Kubernetes.

With this template, it must also be tested how it is possible to access it with multiple users. This will probably require a proxy that authorizes certain requests and forwards them to the respective containers. It must be evaluated how the authentication and the routing works. One possibility would be to include a token in the URL. This may work with Kubernetes out of the box, but if that is not enough there are other possibilities. Traefik for example is a dynamic proxy that automatically detects new services in Docker and integrates them for routing. Consul is another tool for discovering services. Traefik can interact with Consul and Consul can report the new routes to Traefik. The best solution here would be one where Traefik directly detects the routes from Kubernetes, similar to how it already works with Docker in Traefik. If that is not possible we either need to be able to add new routes via Traefik's API or include Consul. Anyway, with Consul it is possible to insert a dynamic configuration of routes afterwards. The insertion of new routes should only be tested manually in this step and then automated in the Lab Orchestrator Library. If this concept does not work with Traefik and Consul, we have to find another proxy possibility, program a new one or extend an already existing one with this function.

1.4.3 Lab Orchestrator Library

The library is the core of the project and will be provided as a Python library. A network of virtualised systems base images is called a lab. The library should be able to manage the virtualised systems base images in Kubernetes and provide an interface to manage labs.

In the requirements list, "must" stands for that it has to be implemented, "should" is an optional requirement that should be implemented and "may" is an optional requirement that may be implemented.

Requirements for the library are:

- start and stop labs (must)
- pause and continue labs (should)

- add and remove labs (must)
- configuration of routing labs (must)
- authentication during routing (must)
- show authentication details in labs, e.g. login credentials (must)
- link users to their labs (must)
- add instructions (must)
- link labs and instructions (must)

Requirements for the instructions:

- Markdown or HTML syntax (should)
- pages with text (must)
- controller to select a virtualised system (must)
- steps per page (may)
- tick of steps (may)
- progress bar (may)
- progress bar for ticked steps (may)
- embed images and other media (should)
- present knowledge texts (must)
- interactions with virtualised systems, e.g. copy a text into the clipboard of a system (may)
- variables (may)

There is a Kubernetes client library for Python. This library can be used in the Lab Orchestrator to get access to the Kubernetes API and interact with Kubernetes.

Core functionalities of the library are start, stop, add and remove labs. To start and stop, the previously created template must be mapped into the Kubernetes Client Library and some settings such as the namespace must be kept variable. The Client Library can then be used to start and stop the templates. The configuration of the templates must be stored in a database and contain among other things the access token, the user ID and the specific template configuration like the namespace which is used.

For adding new labs, it is intended that one provides a path to the images of the VMs or, in the case of Docker, optionally a link to the image in a container registry. The specified VMs or containers are then added to the template and must have the respective terminal/desktop web solution integrated and properly configured. Additional Kubernetes configuration can also be entered here. The configuration is then stored in the database. To remove a lab, only the configuration then needs to be deleted from the database.

Pause and continue labs would be a useful extension, which, however, is not mandatory for the first version.

Depending on the routing and authentication solution from the previous step, the proxy must still be told how to use the ports of the VMs or containers when a lab is started. If the Kubernetes native solutions doesn't work, either the Traefik API or Consul must be used. These routing settings must be included in the response at startup so that users know which URL they can use to access it. There must also be a possibility to select the different containers in the URL.

An authorization who can start labs is not provided here and comes in the web interface with a proper user management. That means, everyone who uses this library can do everything in the code and must add an authorization layer, if certain labs are to be started only by certain users.

To link the users with their labs it is sufficient to store a user ID and optionally a name for the user, which can optionally be included in the instructions via variables.

The instructions are only texts, which have to be stored in the database. Several pages can be stored in different database entries or the complete instruction of a course can be stored in one database entry. These are then linked to the respective lab template. All but four features of the instructions are requirements to be implemented in the web interface and are simply different representations of the text. The controller for selecting the virtualised system can include the URLs from the response at startup, as links for the frame. So that individual steps can be checked off, another database table must be added under circumstances, which stores the status. Variables could be queried via an extra function and then also composed by the web interface. For the interaction with the virtualised system, existing solutions can be searched for or an interface for this must be integrated in the virtualised system. The simplest possibility for this would be to offer a service via an internal web interface in the virtualised system, which copies texts into the clipboard.

It must also be evaluated whether the library should write data to a database on its own or only return the data that needs to be saved as a response. The former would be more trivial to use and a SQLite database would be a good choice. The second would provide more flexibility and it would be easier to integrate into Django.

1.4.4 REST-API

The library alone offers the advantage that you can easily write programs that use this concept. For example, you can include the library in a Django app or in desktop software. Another use case for the library would be a web interface to control the library. This way you can include the library in a microservices system or you can access it from other programming languages and thus include it in many other non-Python projects. The web interface will be a REST-API and will be implemented with Django or Flask.

The library will not yet have authentication to start labs, only authorization when accessing the labs. In the web interface the library will be extended by a permission system and user management. Otherwise, the web interface only has to offer the functionalities of the library via REST.

1.5 Milestones

1.5.1 Proof of Concept and Prototype

First we need to understand the concepts of Kubernetes and find a way to to deploy labs. This is done in the proof of concept. After that we will include this in a prototype that should proof that the idea of labs is possible.

1. Install Kubernetes and KubeVirt

2. Understand basics of Kubernetes and Kubernetes templates
3. Understand how to start and stop Kubernetes templates, base images and VMs
4. Evaluation of web-terminal and web-vnc tools
5. Integration of web-terminal and web-vnc tools into base images and VMs
6. Integrate base images and VMs into Kubernetes templates
7. Evaluate and implement a routing solution
8. Add multi-user support to the routing solution

In this step Kubernetes and maybe KubeVirt will be installed and configured. We will take a look at Kubernetes templates and base images and how they can be started and stopped in Kubernetes. This is the basic knowledge we need to build the application.

After that, we will evaluate which web-terminal tool and which web-vnc tool is the most suitable for using in the base images. And afterwards this tools will be integrated into docker base images or VM base images. These will be the basis of the labs.

The base images will be integrated into a Kubernetes template and combined with a basic routing solution. After that works the routing will be extended to support multi-user labs.

If all that steps work, the prototype will be a success. This proves, that the orchestrator can be implemented with Kubernetes and the given web accessible base images.

1.5.2 Alpha Phase

Then in the alpha phase the Lab Orchestrator library will be implemented.

1. Start and stop labs
2. Automatically add routing and authorization
3. Add and remove labs
4. Add and remove instructions
5. Link users to labs
6. Link instructions to labs

At the end of the alpha phase we have a working solution as library, that fulfills the minimal needed set of requirements. This library can than be used in other project for example the REST-API.

1.5.3 Beta Phase

In the Beta Phase we will add the REST-API and add the remaining optional features.

1. Implement a REST-API that is able to use the library to start and stop labs
2. Add user-management and permission system to the REST-API
3. Add remaining features of the instructions
4. Pause and continue labs

After the beta phase has succeeded, the project is considered finished and can be released to the public.

2 Basics

The Lab Orchestrator application uses different tools that may be explained before the installation of the application. This chapter will give you an introduction into the tools that are used and required in this project, as well as an explanation about Kubernetes that is needed to understand how the Lab Orchestrator application is working on the inside.

2.1 Generating The Documentation

The documentation is written in markdown and converted to a pdf using pandoc. To generate the documentation pandoc and latex are used.

Installation: [3] - Download pandoc version 2.14.1 from git[1] and install it - Download pandoc-crossref[2] (v0.3.12.0b), extract the tar and copy the binary to `~/.local/bin` with the command `install pandoc-crossref` - Build and install pandoc-include-code[3] (version 1.5.0.0) and extend you `$PATH` with `~/.cabal/bin` - A latex environment - make

For the replacement of variables there is a lua script installed, so you need to install lua too. [4]

There is a make command to generate the docs: `make docs`.

2.2 Terminal Tools

2.2.1 Make

Make is used to resolve dependencies during a build process. In this project make is used to have some shortcuts for complex build commands. For example there is a make command to generate the documentation: `make docs`.

2.2.2 nohup

If a terminal is closed (for example if you logout), a HUP signal is send to all programs that are running in the terminal. [1] `nohup` is a command that executes a program, with intercepting the HUP signal. That results into the program doesn't exit when you logout. The output of the program is redirected into the file `nohup.out` `nohup` can be used with `&` to start a program in background that continues to run after logout. [2]

2.3 Kubernetes

Kubernetes is an open source container orchestration platform. With Kubernetes it's possible to automate deployments and easily scale containers. It has many features that make it useful for the project. Some of them are explained here. [5]

[1] https://github.com/jgm/pandoc/releases/tag/2.14.1
[2] https://github.com/lierdakil/pandoc-crossref/releases
[3] https://github.com/owickstrom/pandoc-include-code#build

2.3.1 Control Plane

The control plane controls the Kubernetes cluster. It also has an API that can be used with kubectl or REST calls to deploy stuff. [6]

2.3.2 Custom Resource

In Kubernetes it's possible to extend the Kubernetes API with so called custom resources (CR). A custom resource definition (CRD) defines the CR. [8]

2.3.3 Kubernetes Objects

Kubernetes Objects have specs and a status. The `spec` is the desired state the object should have. `status` is the current state of the object. You have to set the `spec` when you create an object.

Kubernetes objects are often described in yaml files. The required fields for Kubernetes objects are: [12] - `apiVersion`: Which version of the Kubernetes API you are using to create this object - `kind`: What kind of object you want to create - `metadata`: Helps to uniquely identify the object - `spec`: The desired state of the object

2.3.4 Pods

A pod is a group of one or more containers that are deployed to a single node. The containers in a pod share an ip address and a hostname.

2.3.5 Deployment

Deployments define the applications life cycle, for example which images to use, the number of pods and how to update them. [7]

2.3.6 Services

Services allows that service requests are automatically redirected to the correct pod. Services gets their own IP addresses that is used by the service proxy.

Services also allow to add multiple ports to one service. When using multiple ports, you must give all of them a name. For example you can add a port for http and another port for https.

Listing 1 Example of a Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

This service has the name `my-service` and listens on the port 80. It forwards the requests to the pods with the selector `app=MyApp` on the port 9376.

There is also the ability to publish services. To make use of this, the `ServiceType` must be changed. The default `ServiceType` is `ClusterIP`, which exposes the service on a cluster-internal IP, that makes this service only reachable from within the cluster. One other service type is `ExternalName`, that creates a CNAME record for this service. Other Types are `NodePort` and `LoadBalancer`. [9]

You should create a service before its corresponding deployments or pods. When Kubernetes starts a container, it provides environment variables pointing to all the services which were running when the container was started. These environment variables has the naming schema `servicename_SERVICE_HOST` and `servicename_SERVICE_PORT`, so for example if your service name is foo: [14]

```
FOO_SERVICE_HOST=<the host the Service is running on>
FOO_SERVICE_PORT=<the port the Service is running on>
```

You can also use Ingress to publish services.

2.3.7 Ingress

An ingress allows you to publish services. It acts as entrypoint for a cluster and allows to expose multiple services under the same IP address. [9]

With ingresses it's possible to route traffic from outside of the cluster into services within the cluster. It also provides externally-reachable URLs, load balancing and SSL termination.

Ingresses are made to expose http and https and no other ports. So exposing other than http or https should use services with a service type `NodePort` or `LoadBalancer`.

Ingresses allows to match specific hosts only and you can include multiple services in an ingress by separating them with a path in the URL. [10]

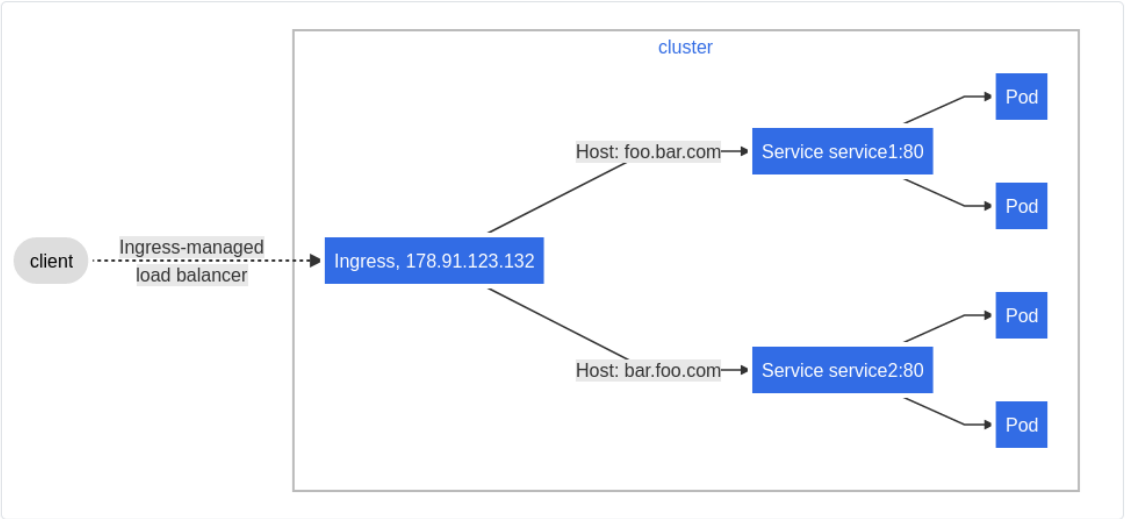

Figure 1: How Ingress interacts with Services and Pods [10]

Listing 2 Example of an Ingress

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: simple-fanout-example
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        pathType: Prefix
        backend:
          service:
            name: service1
            port:
              number: 4200
      - path: /bar
        pathType: Prefix
        backend:
          service:
            name: service2
            port:
              number: 8080
```

To use ingresses you need to have an ingress controller.

2.3.8 Ingress Controllers

Ingress controllers are responsible for fulfilling the ingress.

Examples of ingress controllers are: ingress-nginx[4] and Traefik Kubernetes Ingress provider[5].

2.3.9 Namespaces

Namespaces allows you to run multiple virtual clusters backed by the same physical cluster. They can be used when many users across multiple teams or projects use the same cluster.

Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. Namespaces are also a way to divide cluster resources between multiple users.

Namespaces may be useful to separate the networks of individual users.

2.3.10 Network Policies

With Network Policies it is possible to control the traffic flow at the ip address or port level. It allows you to specify how a pod is allowed to communicate with various network entities over the network. This can be useful to separate the networks of individual users. [11]

2.3.11 Config Maps and Secrets

A ConfigMap is an an API object to store configuration in key-value pairs. They can be used in pods as environment variables, command-line arguments or as a configuration file. [16]

Secrets does the same, but for sensitive information. They are by default unencrypted base64-encoded and can be retrieved as plain text by anyone with api access. But it's possible to enable encryption and RBAC (role based access control) rules. [15]

2.4 Kubernetes Tools

2.4.1 kubectl

`kubectl` is a command line tool that lets you control Kubernetes clusters. It can be used to deploy applications, inspect and manage cluster resources and view logs. [13]

2.4.2 kind and minikube

`kind` is used to deploy a local Kubernetes cluster in docker.

`minikube` is used to deploy a local Kubernetes cluster that only runs one node.

[4] https://kubernetes.github.io/ingress-nginx/deploy/
[5] https://doc.traefik.io/traefik/providers/kubernetes-ingress/

Both tools are used to get started with Kubernetes, to try out stuff and for daily development. To run Kubernetes in production you should install other solutions or use cloud infrastructure. [13]

In this project we use minikube for development.

2.4.3 Helm, Krew, KubeVirt Virtctl and Rancher

Helm is a package manager for Kubernetes. **Krew** is a package manager for kubectl plugins. **KubeVirt** enables Kubernetes to use virtual machines instead of containers. And **Virtctl** is a kubectl plugin to use KubeVirt with kubectl. Virtctl adds some commands for example to get access to a VMs console.

There is a tool called Containerized Data Importer (CDI) that is designed to import Virtual Machine images for use with KubeVirt. [18]

Rancher is an Web UI for Kubernetes, that can display all running resources and allows an admin to change them and create new. Maybe this is worth a look.

2.5 Web-Terminal Tools

There are several tools available to get access to a terminal over a website. Gotty, wetty and ttyd are examples of this. These tools start a terminal session and then allows a user to access this session over a website.

2.6 Web-VNC Tools

To connect to a VNC session of a virtualised system, there are also several tools. To name two of them, there are Apache Guacamole and noVNC. These tools start a VNC session and then allows a user to access this session over a website.

3 Installation

3.1 Prerequisites

TLDR:

- Install Minikube with kvm2[6]
- Install kubectl[7]
- `minikube start --driver kvm2 --memory=8192 --cpus=4 --disk-size=50g --cni=calico`
- Install Helm[8]
- Install Krew[9]
- Install KubeVirt[10]
- Install Virtctl with Krew
- Install CDI[11]

3.1.1 Kubernetes Development Installation

To run Lab Orchestrator you need an instance of Kubernetes. If you want to use VMs instead of containers you additionally need to install KubeVirt.

For development we use minikube. To install minikube install docker and kvm2[12] or some other driver for VMs and follow this guide[13]. Also install `kubectl` using this guide[14].

After the installation you should be able to start minikube with the command `minikube start --driver kvm2` and get access to the cluster with `kubectl get po -A`. The command `minikube dashboard` starts a dashboard, where you can inspect your cluster on a local website. If you like you can start it with this command in the background: `nohup minikube dashboard >/dev/null 2>/dev/null &`, but then it's only possible to stop the dashboard by stopping minikube with `minikube stop`.

You can start one cluster with docker `minikube start --driver=docker -p docker` and a second cluster with `minikube start -p kubevirt --driver=kvm2`. You should now see both profiles running with `minikube profile list`. This may be helpful for testing. [17]

Minikube creates a VM with 16GB or 20GB disk space. To prevent later errors in the step "Preparing desktop images with cloud-init" which occur due to insufficient space you should create a minikube instance with the parameter `--disk-size=XXGB`, where `XX` is the amount of space you want to use. You can also increase the memory and CPU amount with `--memory` and `--cpus`. We use `minikube start --vm-driver=kvm2`

[6] https://minikube.sigs.k8s.io/docs/start/
[7] https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/
[8] https://helm.sh/docs/intro/install/
[9] https://krew.sigs.k8s.io/docs/user-guide/setup/install/
[10] https://kubevirt.io/quickstart_minikube/
[11] https://kubevirt.io/user-guide/operations/containerized_data_importer/#install-cdi
[12] https://minikube.sigs.k8s.io/docs/drivers/kvm2/
[13] https://minikube.sigs.k8s.io/docs/start/
[14] https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/

`--memory=8192 --cpus=4 --disk-size=50GB`. In the next steps you should remember to add this parameter by yourself. [54] [56]

Also it's needed to use a network plugin that supports NetworkPolicy. The one we are using is called calico and can be configured during startup with `--cni=calico`. With the parameters from above this results into `minikube start --driver kvm2 --memory=8192 --cpus=4 --disk-size=50g --cni=calico`. [66] [65]

`kubectl` is now configured to use more than one cluster. There should be two contexts in `kubectl config view`: `docker` and `kubevirt`. You can use the minikube kubectl command like this to specify which cluster you would like to use: `minikube kubectl get pods -p docker` and `minikube kubectl get vms -p kubevirt`. Or you can specify the context in kubectl like this: `kubectl get pods --context docker` and `kubectl get vms --context kubevirt`.

You can stop them with `minikube stop -p docker` and `minikube stop -p kubevirt`. Deleting them works with the commands `minikube delete -p docker` and `minikube delete -p kubevirt`.

It is sufficient to only run one cluster with kvm2 driver, because this can execute docker as well.

3.1.2 Kubernetes Productive Installation

3.1.3 Helm, Krew, KubeVirt and Virtctl Installation

Start minikube: `minikube start --driver kvm2 --memory=8192 --cpus=4 --disk-size=50g --cni=calico`.

Install Helm using this guide[15].

Install Krew using this guide[16].

If you are running Minikube, use this installation guide to install KubeVirt and then Virtctl with Krew: KubeVirt quickstart with Minikube[17]. Verify the installation. This adds some commands to kubectl for example `kubectl get vms` instead of `kubectl get pods`.

Start kubevirt in the minikube cluster: `minikube addons enable kubevirt` or use the in-depth way. After that deploy a test VM using this guide: Use KubeVirt[18]

You also need to install CDI[19] which is an extension for KubeVirt.

3.2 Lab Orchestrator Installation

[15] https://helm.sh/docs/intro/install/
[16] https://krew.sigs.k8s.io/docs/user-guide/setup/install/
[17] https://kubevirt.io/quickstart_minikube/
[18] https://kubevirt.io/labs/kubernetes/lab1
[19] https://kubevirt.io/user-guide/operations/containerized_data_importer/#install-cdi

4 Proof of Concept

In one of the previous parts we described some basics of docker and Kubernetes. This chapter will extend the knowledge with KubeVirt, Virtual Machines, more in detail knowledge of Kubernetes and practical parts. We will proof that it is possible to deploy VMs and docker containers in Kubernetes, make their console and VNC accessible from the web and separate the namespaces where they are running. This concepts will represent the labs that we try to achieve in this project. Most of this is done via examples of Kubernetes resources in the yaml format and terminal commands, which will be abstracted into a prototype in the next chapter.

4.1 KubeVirt and Virtual Machines

You should have installed kubectl and minikube with activated kubevirt addon.

KubeVirt has a tool called Containerized Data Importer (CDI), which is designed to import Virtual Machine images for use with KubeVirt. This needs to be installed from here if you haven't already done this: Containerized Data Importer (CDI)[20].

The installation of KubeVirt and CDI adds several new CRs, which can be found in the documentation of kubevirt[21].

Resources from Kubernetes:

- PersistentVolume (PV)
  - already included in Kubernetes
- PersistentVolumeClaim (PVC)
  - already included in Kubernetes

CRs from KubeVirt:

- VirtualMachine (VM)
  - An image of an VM, e.g. Fedora 23
  - Can only be started once
- VirtualMachineInstance (VMI)
  - An instance of an VM, e.g. the Lab i'm currently using
- VirtualMachineSnapshot
- VirtualMachineSnapshotContent
- VirtualMachineRestore
- VirtualMachineInstanceMigration
- VirtualMachineInstanceReplicaSet
- VirtualMachineInstancePreset

CRs from CDI:

- StorageProfile
- Containerized Data Importer (CDI)
  - converts an VM image into the correct format to use it as VM in KubeVirt

[20] https://kubevirt.io/labs/kubernetes/lab2.html
[21] https://kubevirt.io/user-guide/

- CDIConfig
- DataVolume (DV)
- ObjectTransfer

4.1.1 KubeVirt Basics

There is an example vm config in the KubeVirt documentation. [19] Download the vm config `wget https://raw.githubusercontent.com/kubevirt/kubevirt.github.io/master/labs/manifests/vm.yaml` and apply it: `kubectl apply -f examples/vm.yaml`. Now you should see, that there is a new VM in `kubectl get vms` called testvm. You can start the VM with `kubectl virt start testvm`. This creates a new VM instance (VMI) that you can see in `kubectl get vmis`. You can then connect to the serial console using `kubectl virt console testvm`. Exit the console with `ctrl+]` and stop the VM with `kubectl virt stop testvm`. Stoping the VM deletes all changes made inside the VM and when you start it again, a new instance is created without the changes. You can start a VM only once.

When a VM gets started, its `status.created` attibutes becomes `true`. If the VM instance is ready, `status.ready` becomes `true` too. When the VM gets stopped, the attributes gets removed. A VM will never restart a VMI until the current instance is deleted. [20]

After starting the VM you can expose its ssh port with this command: `kubectl virt expose vm testvm --name vmiservice --port 27017 --target-port 22`. Then you can get the cluster-ip from `kubectl get svc`. The cluster ip can't be used directly to connect with ssh, but from inside minikube. So to connect to the ssh of the VM execute `minikube ssh`. This logs you in to the minikube environment. From there you can execute the corresponding ssh command, e.g. `ssh -p 27017 cirros@10.102.92.133`. [20]

VMIs can be paused and unpaused with the commands `kubectl virt pause vm testvm` or `kubectl virt pause vmi testvm` and the commands `kubectl virt unpause vm testvm` or `kubectl virt unpause vmi testvm`. This freezes the process of the VMI, that means that the VMI has no longer access to CPU and I/O but the memory will stay allocated. [21]

Listing 3 Example VM (poc/examples/vm.yaml)

```yaml
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: testvm
spec:
  running: false
  template:
    metadata:
      labels:
        kubevirt.io/size: small
        kubevirt.io/domain: testvm
    spec:
      domain:
        devices:
          disks:
            - name: containerdisk
              disk:
                bus: virtio
            - name: cloudinitdisk
              disk:
                bus: virtio
          interfaces:
          - name: default
            bridge: {}
        resources:
          requests:
            memory: 64M
      networks:
      - name: default
        pod: {}
      volumes:
        - name: containerdisk
          containerDisk:
            image: quay.io/kubevirt/cirros-container-disk-demo
        - name: cloudinitdisk
          cloudInitNoCloud:
            userDataBase64: SGkuXG4=
```

The source of the example can be found in [19].

4.1.2 KubeVirt Run Strategies

VirtualMachines have different so called run strategies. If a VMI crashes it restarts if you set `spec.running: true`, but by defining a `spec.RunStrategy` this behaviour can

be changed. You can only use `spec.running` or `spec.RunStrategy` and not both at the same time. There are four run strategies: [22]

- Always: If the VMI crashes, a new one is created. It's the same as setting `spec.running: true`
- RerunOnFailure: VMI restarts, if the previous failed in an error state. It will not be re-created if the guest stopped it.
- Manual: It doesn't restart until someone starts it manually.
- Halted: This means, the VMI is stopped. It's the same as setting `spec.running: false`

4.1.3 KubeVirt Presets

`VirtualMachineInstancePreset` is a resource that can be used to create re-usable settings that can be applied to various machines. These presets work like the `PodPreset` resource from Kubernetes. They are namespaces, so if you need to add these presets to every namespace where you need it. Any domain structure can be added in the `spec` of a preset, for example memory, disks and network interfaces. The presets uses `Labels` and `Selectors` to determine which VMI is affected from the preset. If you don't add any selector, the preset will be applied to all VMIs in the namespace. [23]

You can use presets to define a set of specs with different values and give them labels and then customise VMIs with them. This abstracts some of the specs of VMIs and make it easily customisable to change the specs of a VMI. [23]

Listing 4 Example VirtualMachineInstancePreset

```
apiVersion: kubevirt.io/v1alpha3
kind: VirtualMachineInstancePreset
metadata:
  name: small-qemu
spec:
  selector:
    matchLabels:
      kubevirt.io/size: small
  domain:
    resources:
      requests:
        memory: 64M
```

Listing 5 Example VMI, that matches the correct labels

```yaml
apiVersion: kubevirt.io/v1alpha3
kind: VirtualMachineInstance
version: v1
metadata:
  name: myvmi
  labels:
    kubevirt.io/size: small
```

The source of the examples can be found in [23]. The example shows a preset, which applies 64M of memory to every VMI with the label `kubevirt.io/size: small`. [23]

When a preset and a VMI define the same specs but with different values there is a collision. Collisions are handled in the way that the VMI settings override the presets settings. If there are collisions between two presets that are applied to the same VMI an error occurs. [23]

If you change a preset it is only applied to new created VMIs. Old VMIs doesn't change. [23]

4.1.4 KubeVirt Disks and Volumes

4.1.4.1 Disks Disks are like virtual disks to the VM. They can for example be mounted from inside /dev. Disks are specified in `spec.domain.devices.disks` and need to reference the name of a volume. [25]

Possible disk types are: `lun`, `disk` and `cdrom`. `disk` is an ordinary disk to the VM. `lun` is a disk that uses iCSI commands. And `cdrom` is exposed as a cdrom drive and read-only by default. [25]

Disks have a bus type. A bus type indicates the type of disk device to emulate. Possible types are: `virtio`, `sata`, `scsi`, `ide`. [24]

4.1.4.2 Volumes Volumes are a Kubernetes Concept. They try to solve the problem of ephemeral disks. Without volumes, if a container restarts, it restarts with a clean state and it's not possible to save any state. Volumes allows to have a disk attached, that is persistent. There are ephemeral and persistent volumes. Ephemeral volumes have the same lifetime as a pod. Persistent volumes aren't deleted. For both of them in a given pod, data is preserved across container restarts. [26]

In the context of KubeVirt, volumes define the KubeVirts type of the disk. For example you can make them persistent in your cluster or even store them in a container image registry. [25]

Possible disk types are: `cloudInitNoCloud`, `cloudInitConfigDrive`, `persistentVolumeClaim`, `persistentVolumeClaim`, `dataVolume`, `ephemeral`, `containerDisk`, `emptyDisk`, `hostDisk`, `configMap`, `secret`, `serviceAccount`, `downwardMetrics`. [25]

4.1.4.3 cloudInitNoCloud

`cloudInitNoCloud` can be used to attach some user-data to the VM, if the VM contains a proper cloud-init setup. The NoCloud data will be added as a disk to the VMI. This can be used for example to automatically put an ssh key into `~/.ssh/authorized_keys`. For more information see the cloudinit nocloud documentation[22] or the KubeVirt cloudInitNoCloud documentation[23]. [25]

4.1.4.4 Persistent Volumes and Persistent Volume Claims

Kubernetes provides some resources for providing persistent storage. The first is a `PersistentVolume`. A `PersistentVolume` is a piece of storage in the cluster that is reserved from a cluster administrator or it is dynamically provisioned using Storage Classes. [28] A `StorageClass` is the second resource and it is a way for administrators to customize the types of storage the offer. [27]

Storage Classes use Provisioners to provision persistent volumes. Provisioners are independent programs and not Kubernetes resources. You can write your own provisioner following a specification from Kubernetes. There are already some internal provisioners provided by Kubernetes. This includes AzureFile, AzureDisk, StorageOS and other cloud providers. Minikube adds another provisioner called `storage-provisioner`. You can see it in `minikube addons list` and it should be enabled. A Storage Class binds a provisioner with a specific configuration to a storage class name. For example you can add two storage classes, one with slow long term storage and one for fast NVME storage and give them the names `slow` and `fast`. Now if you create a PVC you can specify which storage class you want to use. For example you can add a volume to your VM with one disk attached that uses a PVC with the slow storage class. The storage class then creates the PV automatically. Minikube also adds a default storage class with the addon `default-storageclass`, which you can see in `minikube addons list` too. You can see the storage class with `kubectl get storage class`. In the provisioner field of the response you can see the provisioner `k8s.io/minikube-hostpath`, which is the default provisioner in minikube. [28]

You can read more about `StorageClass` and `PersistentVolume` in the Kubernetes documentation about Storage Classes[24] and PersistentVolumes[25].

A `PersistentVolumeClaim` (PVC) is the third resource provided by Kubernetes. It is a request for storage by a user. In KubeVirt it is used, when the VMIs disk needs to persist after the VM terminates. This makes the VM data persistent between restarts. `PersistentVolumes` and `StorageClasses` can be used to customize the Storage that can be provided to PVCs. [25]

If you create your own persistent volumes you should know that in minikube only some folders are persistent, for example: `/data` and `/tmp/hostpath_pv`. If you save your PVs in other folders they will be removed on every reboot. [68]

[22] http://cloudinit.readthedocs.io/en/latest/topics/datasources/nocloud.html
[23] https://kubevirt.io/user-guide/virtual_machines/disks_and_volumes/#cloudinitnocloud
[24] https://kubernetes.io/docs/concepts/storage/storage-classes/#local
[25] https://kubernetes.io/docs/concepts/storage/persistent-volumes/#persistentvolumeclaims

Listing 6 Example of PV

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0001
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 10Gi
  hostPath:
    path: /mnt/vda1/hostpath_pv/pv0001
```

The example PV creates a PV with 10GB saving it in `/mnt/vda1/hostpath_pv/pv0001`. You can see the created PV in `minikube ssh` when executing `ls /mnt/vda1/hostpath_pv/` or with executing `kubectl get pv`. The example can be found in the minikube persistent volumes docs.

Listing 7 Example of PVC

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
  storageClassName: standard
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

The example PVC creates a PVC with 3GB using the storage class `standard`. You can see the created PV in `minikube ssh` when executing `ls /mnt/vda1/hostpath-provisioner/default` or with executing `kubectl get pv`. The example can be found in the kubernetes configure persistent volume storage docs.

Listing 8 Example of VMI with PVC

```yaml
metadata:
  name: testvmi-pvc
apiVersion: kubevirt.io/v1alpha3
kind: VirtualMachineInstance
spec:
  domain:
    resources:
      requests:
        memory: 64M
    devices:
      disks:
      - name: mypvcdisk
        lun: {}
  volumes:
    - name: mypvcdisk
      persistentVolumeClaim:
        claimName: mypvc
```

The source of the example can be found in [25]. This examples creates a VMI and attaches a PVC with the name `mypvc` as a lun disk.

4.1.4.5 Data Volumes

`dataVolume` are part of the Containerized Data Importer (CDI) which need to be installed separately. A data volume is used to automate importing VM disks onto PVCs. Without a `DataVolume`, users have to prepare a PVC with a disk image before assigning it to a VM. DataVolumes are defined in the VM spec by adding the attribute list `dataVolumeTemplates`. The specs of a data volume contain a `source` and `pvc` attribute. `source` describes where to find the disk image. `pvc` describes which specs the PVC that is created should have. An example can be found here[26]. When the VM is deleted, the PVC ist deleted as well. When a VM manifest is posted to the cluster (for example with a yaml config), the PVC is created directly before the VM is even started. That may be used for performance improvements when starting a VM. It is possible to attach a data volume while creating a VMI, but then the data volume is not tied to the life-cycle of the VMI. [25]

4.1.4.6 Container Disks

`containerDisk` is a volume that references a docker image. The disks are pulled from the container registry and reside on the local node. It is an ephemeral storage device and can be used by multiple VMIs. This makes them an ideal tool for users who want to replicate a large number of VMs that do not require persistent data. They are often used in `VirtualMachineInstanceReplicaSet`. They are not a good solution if you need persistent root disks across VM restarts. Container disks are file based and therefore cannot be attached as a lun device. [25]

To use container disks you need to create a docker image which contains the VMI disk.

[26] https://kubevirt.io/user-guide/virtual_machines/disks_and_volumes/#datavolume-vm-behavior

The disk must be placed into the `/disk` directory of the container and must be readable for the user with the UID 107 (qemu). The format of the VMI disk must be raw or qcow2. The base image of the docker image should be based on the docker `scratch` base image and no other content except the image is required. [25]

Listing 9 Dockerfile example with local qcow2 image

```
FROM scratch
ADD --chown=107:107 fedora25.qcow2 /disk/
```

Listing 10 Dockerfile example with remote qcow2 image

```
FROM scratch
ADD --chown=107:107 https://cloud.centos.org/centos/7/images/CentOS-7-
↪ x86_64-GenericCloud.qcow2
↪ /disk/
```

Listing 11 Example VMI with Container Disk

```
metadata:
  name: testvmi-containerdisk
apiVersion: kubevirt.io/v1alpha3
kind: VirtualMachineInstance
spec:
  domain:
    resources:
      requests:
        memory: 8G
    devices:
      disks:
      - name: containerdisk
        disk: {}
  volumes:
    - name: containerdisk
      containerDisk:
        image: vmidisks/fedora25:latest
```

The source of the examples can be found in [25]. The dockerfiles can then be build with `docker build -t example/example:latest .` and pushed to a remote docker container registry with `docker push example/example:latest`. [25]

4.1.4.7 Empty Disks and Ephemeral Disks

`emptyDisk` is a temporary disk which shares the VMIs lifecycle. The disk lifes as long as the VM, so it will persist between reboots and will be deleted when the VM is deleted. You need to specify the `capacity`. [25]

`ephemeral` is also a temporary disk, but it wraps around `PersistentVolumeClaims`. It is mounted as read-only network volume. An ephemeral volume is never mutated, instead all writes are stored on the ephemeral image which exists locally. The local image is created when a VM starts and it is deleted when the VM stops. They are useful when persistence is not needed. [25]

The difference between `ephemeral` and `emptyDisk` is, that `ephemeral` disks are read only and there is only a small space for application data. Also the application data is deleted, when the VM reboots. This can cause problem to some applications and then it's useful to use `emptyDisks`. [25]

4.1.4.8 Remaining Volumes

`hostDisk`, `configMap`, `secrets` and the other volumes are explained in the KubeVirt Disks and Volumes Documentation[27].

4.1.5 KubeVirt Interfaces and Networks

There are two parts needed to connect a VM to a network. First there is the interface that is a virtual network interface of a virtual machine and second there is the network which connects VMs to logical or physical devices.

Networks need unique names and a type. There are two fields in a network. The first field is `pod`. A pod network is the default `eth0` interface. [30] And the second field is Multus. Multus enables attaching a secondary interface that enables multiple network interfaces in Kubernetes. To be able to use multus it needs to be installed separately. [29]

Interfaces describe the properties of a virtual interface and are seen inside the quest instance. They are defined in `spec.domain.devices.interfaces`. You can specify its type by adding the type with curly brackets (`masquerade: {}`). Available types are `bridge`, `slirp`, `sriov` and `masquerade`. Other properties that you can change are `model`, `macAddress`, `ports` and `pciAddress`. Custom mac addresses are not always supported.

You can read more about the types here[28]

[27] https://kubevirt.io/user-guide/virtual_machines/disks_and_volumes/
[28] https://kubevirt.io/user-guide/virtual_machines/interfaces_and_networks/

Listing 12 Example Network and Interface

```yaml
kind: VM
spec:
  domain:
    devices:
      interfaces:
        - name: default
          masquerade: {}
          ports:
           - name: http
             port: 80
  networks:
  - name: default
    pod: {}
```

The ports field can be used to limit the ports the VM listens to.

If you would like to disable network connectivity, you can use the `autoattachPodInterface` field.

Listing 13 Example of autoattachPodInterface

```yaml
kind: VM
spec:
  domain:
    devices:
      autoattachPodInterface: false
```

4.1.6 KubeVirt Network Policy

By default, all VMIs in a namespace share a network and are accessible from other VMIs. To isolate them, you can create NetworkPolicy objects. NetworkPolicy objects entirely control the network isolation in a namespace. Examples on how to deny all traffic, only allow traffic in the same namespace or only allow HTTP and HTTPS access can be found here[29]. [37]

NetworkPolicy objects are included in Kubernetes and are used to separate networks of pods. But with KubeVirt installed, VMIs and pods are treated equally and NetworkPolicy objects can be used for VMIs too. We need to add NetworkPolicy objects to isolate the VMIs of different users, so that the users can't connect to the VMIs of other users. If we create a new namespace for every user, the default settings are sufficient, but creating NetworkPolicy objects gives us more flexibility, e.g. cross namespace connections or isolation of ports. [38]

To use network policies you need to install a network plugin, that supports network policies. So make sure your cluster fulfills this condition. [38]

[29] https://kubevirt.io/user-guide/virtual_machines/networkpolicy/

Network policies are additive, so if you add two policies the union of them is chosen. If the egress policy or the ingress policy on a pod denies the traffic, the traffic will not be possible even though the network policy would allow it. [38]

How to create and use a NetworkPolicy object is described in the kubernetes network policy documentation[30]. You need to define a name of the network policy in the `metadata.name` field and you can specify the namespace this network policy is running in in the `metadata.namespace` field. After that you can specify the policy in the `spec` field. The `spec` field contains a `podSelector`, `policyTypes`, `ingress` and `egress` fields. The `podSelector` field selects the pods the policy will be applied to by defining labels. If the selector is empty all pods in the namespace are selected. Available `policyTypes` are `Ingress` and `Egress`. They can be added to this field to include them. The `Ingress` type is used for incoming requests and the `Egress` type is used for outgoing requests. If you don't specify this field, `Ingress` is activated by default and `Egress` only if an `Egress` rule is added. To add `Ingress` and `Egress` rules there is also the `ingress` and `egress` field in `spec`. Each `ingress` rule allows traffic which matches both the `from` and `ports` sections. The `egress` rules matches both the `to` and `ports` sections. Inside the `from` or `to` sections you can specify for example a `podSelector`, an `ipBlock` or a `namespaceSelector`. The full list of available options can be found in the NetworkPolicy reference[31]. [38]

Listing 14 Example of NetworkPolicy

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: multi-port-egress
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Egress
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 32000
      endPort: 32768
```

The example shows a network policy with an `Egress` rule that allows all pods and VMIs with the label `role: db` to connect to all pods and VMIs within the IP range 10.0.0.0/24

[30] https://kubernetes.io/docs/concepts/services-networking/network-policies/#networkpolicy-resource

[31] https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#networkpolicy-v1-networking-k8s-io

over TCP with the ports between 32000 and 32778. The source of the example can be found here: [38].

4.1.7 KubeVirt ReplicaSets

`VirtualMachineInstanceReplicaSets` are similar like Kubernetes `ReplicaSets`. They are used to deploy multiple instances of the same VMI to guarantee uptime. There is no state in the instances of a `ReplicaSet` so you need to use read-only or internal writable tmpfs disks. Since our labs need a state we probably won't need `ReplicaSets`. [39]

4.1.8 KubeVirt Services

VMIs can be exposed with services. Services were explained earlier. This is needed to connect to a VMI for example over SSH. Services use labels to identify the VMI, so you need to add labels to the VMI you want to connect to. To create a new Service you can either create a File and load it with `kubectl -f file.yaml` or you can use the `virtctl` tool (remember it may also be used with `kubectl virt`): `virtctl expose virtualmachineinstance vmi-ephemeral --name lbsvc --type LoadBalancer --port 27017 --target-port 3389`. This command uses the type `LoadBalancer`, other types are `NodePort` and `ClusterIP`. [40]

Listing 15 Example VMI with Labels

```yaml
apiVersion: kubevirt.io/v1alpha3
kind: VirtualMachineInstance
metadata:
  name: vmi-ephemeral
  labels:
    special: key
spec:
  ...
```

Listing 16 Example VMI exposed as Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: vmiservice
spec:
  ports:
  - port: 27017
    protocol: TCP
    targetPort: 22
  selector:
    special: key
  type: ClusterIP
```

The examples are from KubeVirts Service Objects Documentation[32] and they show a VMI with a Label and a Service that exposes SSH on this VMI.

4.1.9 KubeVirt Other Features

There are several other features that we are not going into detail but recommend reading. The most interesting features are the following:

- Virtual Hardware[33], e.g. Resources like CPU, timezone, GPU and memory.
- Liveness and Readiness Probes[34]
- Startup Scripts[35]
- KubeVirt Snapshots, may be used to pause VMs.
- KubeVirt user interface options[36], there are different KubeVirt User Interfaces.

4.1.10 KubeVirt Containerized Data Importer (CDI)

The CDI is a separate project that can be added to KubeVirt. To use this you need to install it[37] if you haven't already done this.

If you use minikube assure that the addons `storage-provisioner` and `default-storageclass` are enabled. If you don't use minikube, you need to create a persistent volume or a storage class.

There is a nice guide on how to upload VM images to KubeVirt with CDI that you can find here[38]. Because we will be using `ContainerDisks` instead of `DataVolumes` this is not as interesting and just a marginal note that you can use this way as alternative. In short you need to create a service for the CDI upload proxy, then create a `DataVolume` and an upload token. After that you can upload your images into the data volume with curl.

If you want to know more about CDI and uploading images directly to you cluster take a look at KubeVirts CDI docs[39].

4.1.11 KubeVirt Additional Plugins

The local persistence volume static provisioner[40] manages the PersistentVolume lifecycle for preallocated disks.

4.1.12 cloud-init

Cloud-init is a standard for cloud instance initialization. Cloud-init will read any provided metadata and initialize the system accordingly. This includes setting up network and

[32] https://kubevirt.io/user-guide/virtual_machines/service_objects/
[33] https://kubevirt.io/user-guide/virtual_machines/virtual_hardware/
[34] https://kubevirt.io/user-guide/virtual_machines/liveness_and_readiness_probes/
[35] https://kubevirt.io/user-guide/virtual_machines/startup_scripts/
[36] https://kubevirt.io/2019/KubeVirt_UI_options.html
[37] https://kubevirt.io/user-guide/operations/containerized_data_importer/#install-cdi
[38] https://github.com/kubevirt/containerized-data-importer/blob/main/doc/upload.md
[39] https://kubevirt.io/user-guide/operations/containerized_data_importer/
[40] https://github.com/kubernetes-sigs/sig-storage-local-static-provisioner

storage devices and configuring SSH. For example it is possible to provide an ssh key as metadata. [33]

Cloud-init supports Windows and all major Linux distributions like: Arch, Alpine, Debian, Fedora, RHEL and SLES. [32]

Cloud-init needs to be integrated into the boot of the VM. For example this can be done with systemd. [31] In addition cloud-init needs a datasource. There are many supported datasources for different cloud providers, but the most important for this project will be the NoCloud datasource, because this can be used in KubeVirt as we have already seen above. [34] NoCloud allows to provide meta-data to the VM via files on a mounted filesystem. [35]

It is modularized and there are many modules available to support many different system configurations and different tools. The most important will be the SSH module and maybe Apt Configure, Disk Setup and Mount. All modules can be found in the cloud-init Modules Documentation[41] and examples can be found in the cloud-init config examples documentaion[42].

[41] https://cloudinit.readthedocs.io/en/latest/topics/modules.html
[42] https://cloudinit.readthedocs.io/en/latest/topics/examples.html

Listing 17 Example VM with cloud-init NoCloud

```
apiVersion: kubevirt.io/v1alpha1
kind: VirtualMachine
metadata:
  name: myvm
spec:
  terminationGracePeriodSeconds: 5
  domain:
    resources:
      requests:
        memory: 64M
    devices:
      disks:
      - name: registrydisk
        volumeName: registryvolume
        disk:
          bus: virtio
      - name: cloudinitdisk
        volumeName: cloudinitvolume
        disk:
          bus: virtio
  volumes:
    - name: registryvolume
      registryDisk:
        image: kubevirt/cirros-registry-disk-demo:devel
    - name: cloudinitvolume
      cloudInitNoCloud:
        userData: |
          ssh-authorized-keys:
            - ssh-rsa AAAAB3NzaK8L93bWxnyp test@test.com
```

This is an example that shows how cloud-init NoCloud could be used in KubeVirt to add an ssh key. The created VM contains two disks, one for the image that should be used and another disk that is used by cloud-init. The source can be found here: [36].

4.1.13 KubeVirt Running Windows TODO

https://kubevirt.io/user-guide/virtual_machines/windows_virtio_drivers/

4.2 Building a custom VM

In the first step we try to get a cloud image from a Linux distribution running inside of KubeVirt and then excess it. The second step tries to build a custom image on top of the cloud image. This is needed to install software for labs. In the third and fourth step we will install ttyd and noVNC or alternatives of that and access them with a web browser.

4.2.1 Custom Base Image with Cloud-init Setup

In KubeVirt you need cloud-images in the format of `qcow2` or `raw`. You can obtain your preferred distro from the openstack image guide[43]. The list in the openstack image guide contains images that comes with cloud-init preinstalled. This is useful, because most of them doesn't have a default login and we need to add the login data with cloud-init. In this example we have used the Ubuntu Hirsute cloud-image[44], saved it in the folder `images` and we have a docker hub account.

Listing 18 Example Dockerfile for Custom Image

```
1 FROM scratch
2 ADD --chown=107:107 images/ubuntu-21.04-server-cloudimg-amd64.img /disk/
```

After downloading the image create a dockerfile that adds the image into `/disk/`. The listing 16 shows how to do this with the ubuntu image. Save this file in a file called `dockerfile`.

After that, build the dockerfile with `docker build -t dockerhubusername/reponame:ubuntu2104 -f dockerfile .`. Then login the docker client to docker hub with `docker login` and providing your login credentials. Then upload the build image to docker hub with `docker push dockerhubusername/reponame:ubuntu2104`. Now you have a docker image in docker hub that contains the ubuntu cloud-image in `/disk/`.

In the next step we will use this image with a container disk to run the ubuntu cloud-image. First create a file called `ubuntu_container_disk.yaml` and add a container disk setup.

[43] https://docs.openstack.org/image-guide/obtain-images.html
[44] https://cloud-images.ubuntu.com/releases/hirsute/release/

Listing 19 Example Container Disk for Custom Image

```yaml
metadata:
  name: testvmi-containerdisk
  labels:
    special: key
apiVersion: kubevirt.io/v1alpha3
kind: VirtualMachineInstance
spec:
  domain:
    resources:
      requests:
        memory: 500M
    devices:
      disks:
      - name: containerdisk
        disk: {}
      - name: cloudinitdisk
        disk:
          bus: virtio
  volumes:
    - name: containerdisk
      containerDisk:
        image: dockerhubusername/reponame:ubuntu2104
    - name: cloudinitdisk
      cloudInitNoCloud:
        userData: |-
          #cloud-config
          users:
            - name: root
              ssh-authorized-keys:
                - ssh-rsa AAAABSSHKEY
          ssh_pwauth: True
          password: toor
          chpasswd:
            expire: False
            list: |-
               root:toor
```

The listing 17 is an example of a container disk that uses the docker image we have created previously. Also there is a cloud-init NoCloud disk attached that adds login credentials that can be used to login to the VM via console and via ssh. In this example the username is root and the password toor. If you want to use ssh replace `ssh-rsa AAAABSSHKEY` with your ssh-key, else remove this part of the configuration. To disable password login set `ssh_pwauth: False`.

Then run this VMI with the command `kubectl apply -f ubuntu_container_-`

`disk.yaml` and wait until the VMI is started with `kubectl wait --for=condition=Ready vmis/testvmi-containerdisk` or `kubectl wait --for=condition=Ready -f ubuntu_container_disk.yaml`.

Now the VMI is running and you can access it over console: `kubectl virt console testvmi-containerdisk`. To access the ssh, you need to create a service and connect over `minikube ssh`. Create the service with `kubectl virt expose vmi testvmi-containerdisk --name vmiservice --port 27017 --target-port 22`. You can get the ip with `kubectl get svc`. To connect to ssh, you need to execute `minikube ssh`, then insert you ssh private key with:

Listing 20 Insert SSH Key in Minikube

```
cat <<<EOF > ~/.ssh/id_rsa
YOURSSHKEY
EOF
```

After that change the permissions of the file to 600 with `chmod 600 ~/.ssh/id_rsa` and connect to the ip from the service on the given port with `ssh -p PORT root@IP` and you should be connected to the VMI.

```
➜  prototype git:(master) ✗ kubectl virt console testvmi-containerdisk
Successfully connected to testvmi-containerdisk console. The escape sequence is ^]

testvmi-containerdisk login: root
Password:
Welcome to Ubuntu 21.04 (GNU/Linux 5.11.0-22-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Sat Jul  3 20:28:23 UTC 2021

  System load:  0.44              Processes:               113
  Usage of /:   65.1% of 1.96GB   Users logged in:         0
  Memory usage: 41%               IPv4 address for enp1s0: 172.17.0.17
  Swap usage:   0%

0 updates can be applied immediately.


The list of available updates is more than a week old.
To check for new updates run: sudo apt update


The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

root@testvmi-containerdisk:~# whoami
root
root@testvmi-containerdisk:~#
```

Figure 2: Example of Console Login

```
➜  prototype git:(master) ✗ kubectl get svc
NAME         TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)     AGE
kubernetes   ClusterIP   10.96.0.1     <none>        443/TCP     28d
vmiservice   ClusterIP   10.98.58.45   <none>        27017/TCP   3h12m
➜  prototype git:(master) ✗ minikube ssh

                         _                 _
            _           _ ( )           ( )
  ___ ___  (_)  ___  (_)| |/')  _   _ | |_      __
/' _ ` _ `\| |/' _ `\| || , <  ( ) ( )| '_`\  /'__`\
| ( ) ( ) || || ( ) || || |\`\ | (_) || |_) )(  ___/
(_) (_) (_)(_)(_) (_)(_)(_) (_)`\___/'(_,__/'`\____)

$ cat <<EOF > ~/.ssh/id_rsa
> -----BEGIN RSA PRIVATE KEY-----
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
> -----END RSA PRIVATE KEY-----
> EOF
$ chmod 600 ~/.ssh/id_rsa
```

Figure 3: Example of SSH

```
$ ssh -p 27017 root@10.98.58.45
Welcome to Ubuntu 21.04 (GNU/Linux 5.11.0-22-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Sat Jul  3 20:34:33 UTC 2021

  System load:  0.0               Processes:               111
  Usage of /:   65.7% of 1.96GB   Users logged in:         1
  Memory usage: 39%               IPv4 address for enp1s0: 172.17.0.17
  Swap usage:   0%


0 updates can be applied immediately.


The list of available updates is more than a week old.
To check for new updates run: sudo apt update

Last login: Sat Jul  3 20:32:27 2021 from 172.17.0.1
root@testvmi-containerdisk:~# whoami
root
root@testvmi-containerdisk:~#
```

Figure 4: Example of SSH Setup Login

4.2.2 Customize Image

4.2.2.1 Docker Notice Because we use container disks in this part to run our VMs in the Kubernetes cluster and container disks are using docker you need to know about the build context of docker to prevent errors. The build context of docker is a folder from where all recursive contents of files and directories will be sent to the docker daemon. This is used by docker to isolate the context where the image is build and to prevent errors in later use of the image. If one of your subdirectories contains all your VM images, all of them will be sent to the docker daemon and this may slow down the build process. Also if your build context is too big docker hub will not accept your images even if they are much smaller (e.g. your docker image contains one VM and is 5GB big, but the context contains 6 VMs and is 30GB big). To speedup your builds and to prevent docker hub errors you should always separate your dockerfiles and VM images into different folders or exclude all VM images that you are not adding to the docker image in the `.dockerignore`. [57] [58]

4.2.2.2 Non Cloud Images If you use other images than cloud images, you need to install cloud-init manually. To use such an image, open it with gnome boxes, then install your software and shutdown. After that your qcow2 image is saved in `~/.var/app/org.gnome.Boxes/data/gnome-boxes/images/` if you installed gnome boxes with snap or in `~/.local/share/gnome-boxes` if you installed it with apt. [41]

4.2.2.3 Startup, password and internet setup

First install virt-customize, for example by following this guide: Customize qcow2-raw-image templates with virt-customize[45]. [42]

virt-customize allows you to customize your cloud images. The command `sudo virt-customize -a your_image.img --root-password password:StrongRootPassword` for example changes the password of the root user. This is needed, because most of the cloud images doesn't have a default root password. [42]

Now you can start the image with gnome boxes by adding a new VM with this as image. This creates a new qcow2 image in `~/.var/app/org.gnome.Boxes/data/gnome-boxes/images/` if you installed gnome boxes with snap or in `~/.local/share/gnome-boxes` if you installed it with apt. [41] You can check the filetype with `file filename`. After every change on the original image you need to delete the VM and create a new or you directly change the new image. After starting the image login with the previously set root password. You need to connect to the internet with the command `dhclient`. This gets an ip-address.

4.2.2.4 Resize

To resize the image first stop it if you are running it. Check the image size with the command `qemu-img info your_image.img`. Then resize the image size with `sudo qemu-img resize your_image.img +4G` and check the new size again with `qemu-img info your_image.img`. [43] [44]

Now start it and check how big the partitions are with `df -h`. Then execute `fdisk /dev/sda`. Then use `p` command to print all partitions and search for `Linux filesystem`. In the Ubuntu cloud image this is `/dev/sda1`. Next you need to delete this partition with the `d` command and then input the partition number (`1`). Now add a new partition with `n` and the same number (`1`) and then take the default values of the next two questions. After that don't remove the signature (`N`). Last execute `w` to write the changes. Now the partition is resized and you need to resize the filesystem. This is done with `resize2fs /dev/sda1`. [43] [45] [46]

Now you have a resized image of your cloud image in the gnome boxes folder. Make a backup of it by copying the file.

In the following images, the host terminal has white background and the VM terminal has a black background.

```
➜  images git:(master) ✗ qemu-img info custom-ubuntu-21.04-server-cloudimg-amd64.img
image: custom-ubuntu-21.04-server-cloudimg-amd64.img
file format: qcow2
virtual size: 2.2 GiB (2361393152 bytes)
disk size: 551 MiB
cluster_size: 65536
Format specific information:
    compat: 0.10
    refcount bits: 16
```

Figure 5: Image size before change

[45] https://computingforgeeks.com/customize-qcow2-raw-image-templates-with-virt-customize/

```
→  images git:(master) ✗ sudo qemu-img resize custom-ubuntu-21.04-server-cloudimg-amd64.img +
4G
Image resized.
→  images git:(master) ✗ qemu-img info custom-ubuntu-21.04-server-cloudimg-amd64.img
image: custom-ubuntu-21.04-server-cloudimg-amd64.img
file format: qcow2
virtual size: 6.2 GiB (6656360448 bytes)
disk size: 551 MiB
cluster_size: 65536
Format specific information:
    compat: 0.10
    refcount bits: 16
```

Figure 6: Image size after change

```
root@ubuntu:~# df -h
Filesystem      Size  Used Avail Use% Mounted on
tmpfs           198M  964K  197M   1% /run
/dev/sda1       2.0G  1.3G  677M  67% /
tmpfs           989M     0  989M   0% /dev/shm
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs           4.0M     0  4.0M   0% /sys/fs/cgroup
/dev/sda15      105M  5.2M  100M   5% /boot/efi
tmpfs           198M  4.0K  198M   1% /run/user/0
root@ubuntu:~# fdisk /dev/sda

Welcome to fdisk (util-linux 2.36.1).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

GPT PMBR size mismatch (4612095 != 13000703) will be corrected by write.
The backup GPT table is not on the end of the device. This problem will be corre
cted by write.

Command (m for help):
```

Figure 7: Disk size before change

```
Command (m for help): p

Disk /dev/sda: 6.2 GiB, 6656360448 bytes, 13000704 sectors
Disk model: QEMU HARDDISK
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: gpt
Disk identifier: 65EEDECF-3EFC-47F0-A985-253F82CD250D

Device       Start     End Sectors  Size Type
/dev/sda1   227328 4612062 4384735  2.1G Linux filesystem
/dev/sda14    2048   10239    8192    4M BIOS boot
/dev/sda15   10240  227327  217088  106M EFI System

Partition table entries are not in disk order.

Command (m for help): _
```

Figure 8: fdisk partition size before change

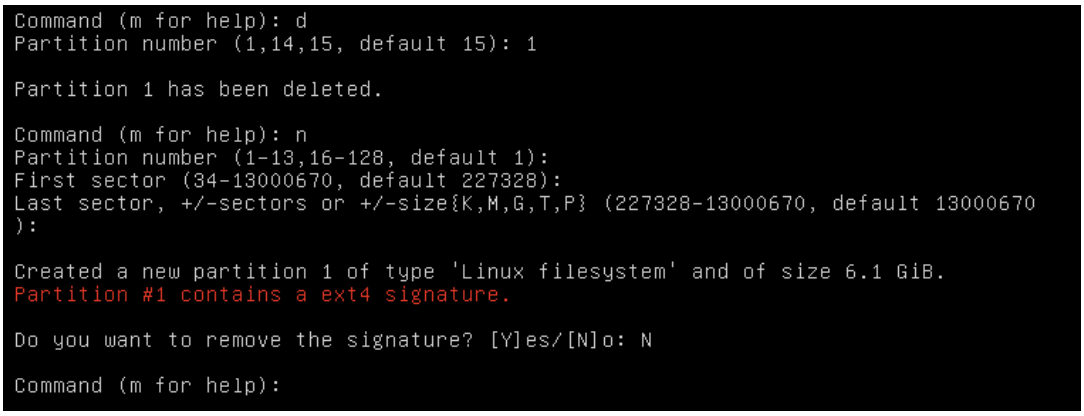

Figure 9: fdisk delete partition and create new

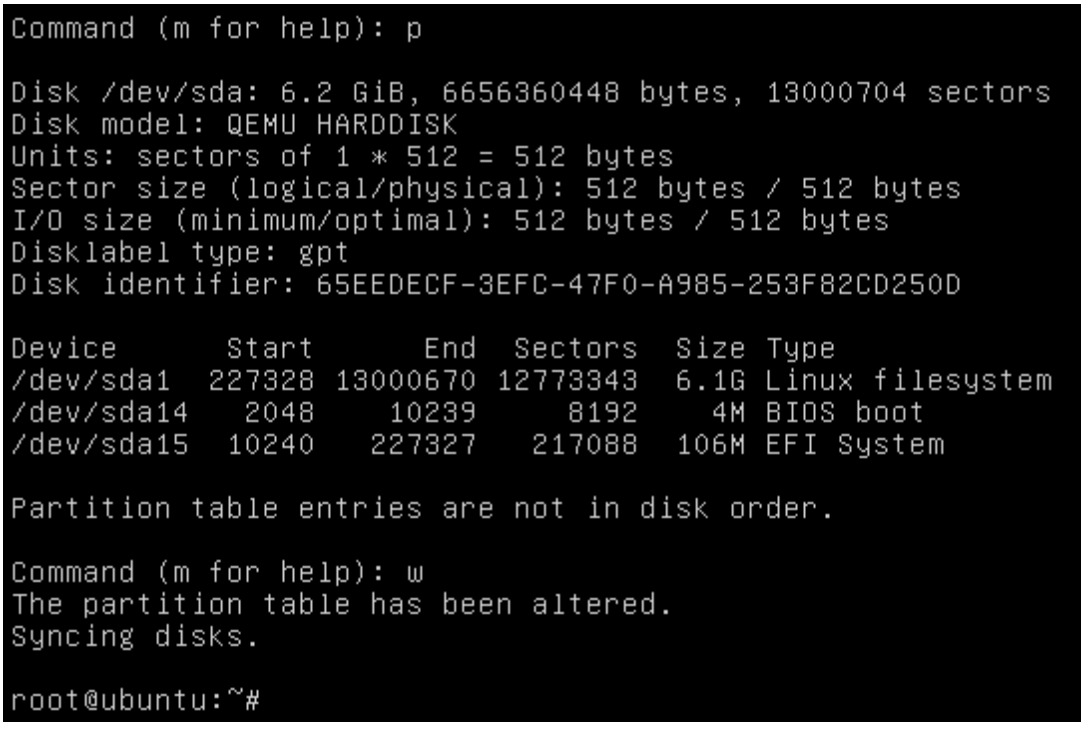

Figure 10: fdisk partition size after change and write changes

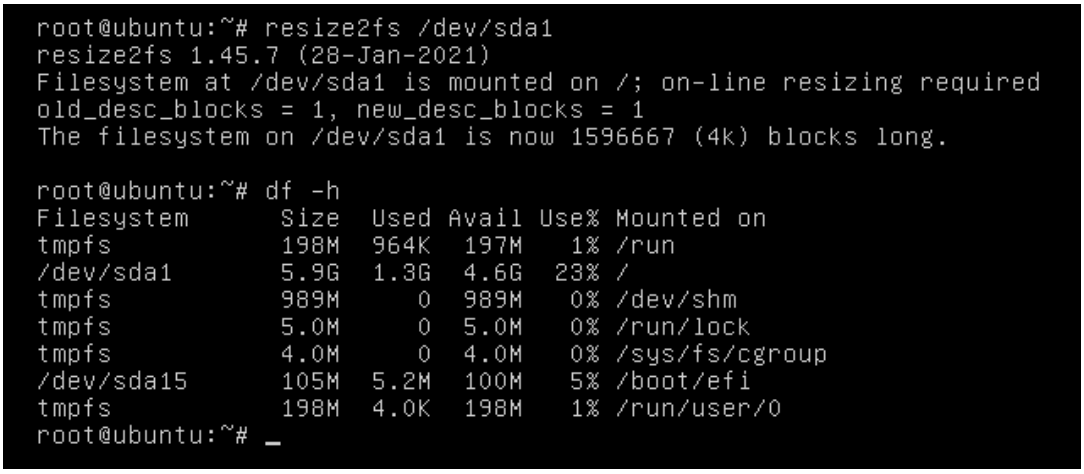

Figure 11: Resize filesystem and disk size after change

4.2.2.5 Installing software

There are two ways of installing software. The first uses `virt-customize` and the second uses gnome boxes.

To install software with `virt-customize` you can append the command `--install PackageName`, e.g. `virt-customize -a your_image.img --install firefox`. This uses the default package manager. [42]

If it's not possible to install software with the package manager, you can use the second way, gnome boxes. Start the resized image and connect to internet (`dhclient`). Then update the software with `apt update && apt upgrade` and install your software and shutdown. Remember that you are editing the image in the gnome boxes folder and not the original image.

4.3 Web Terminal Access

In this step we will try to get access to the terminal over a website. There are two ways of achieving this goal, the first is to install ttyd or a similar software inside the VM and the second one is to run ttyd outside of the VM and share `kubevirt virt console`.

4.3.1 ttyd inside VM

To achieve this, we need one of the VMs from before with enough space to install software.

Start the VM in boxes and install ttyd with this guide[46]. Then after the installation ttyd needs to be started automatically within systemstart. This can be done for example by adding a cronjob with `@reboot`. So execute `crontab -e` and add `@reboot ttyd bash` there. This will start `ttyd bash` when the system starts. It will automatically log in the user from which you run the cronjob. If you run `crontab -e` with the root user, the ttyd shell will have root permissions in the VM. If you run `crontab -e` with a custom user, the ttyd shell will have the permissions of this user. You can change `bash` with all other commands you want to be executed inside the webshell. `ttyd command` will execute the command and share it over http. In this scenario we like to have access to a bash console

[46] https://github.com/tsl0922/ttyd#installation

in the web browser, but you can also start a zsh or other shell or even nodejs, python interpreter or other software. [48] [49]

After installing ttyd and starting it automatically on system start with cron, we need to run this image in Kubernetes and expose the ttyd service. For this first stop the VM and copy the generated qcow2 image from `~/.var/app/org.gnome.Boxes/data/gnome-boxes/images/` or `~/.local/share/gnome-boxes` to your folder. Then build a new docker image that adds this qcow2 file, an example is given in listing 16. Push the image to your docker registry. Then start a new Kubernetes VM with an container disk attached that references this docker image like shown in listing 17. If you are using listing 17, the root password that you may have set before will be overwritten.

Now there should be our custom VM running in Kubernetes. To access the ttyd service we need to expose the port. The default ttyd port is 7681, so execute the command `kubectl virt expose vmi your_vmi_name --type=NodePort --name ttydservice --port 27017 --target-port 7681`. This creates a Kubernetes NodePort service, that makes it possible for us to access the port 7681 of the VM over the ip of our node with the port 27017. `minikube service ttydservice` will let us connect to the service and open it in our default browser. [47]

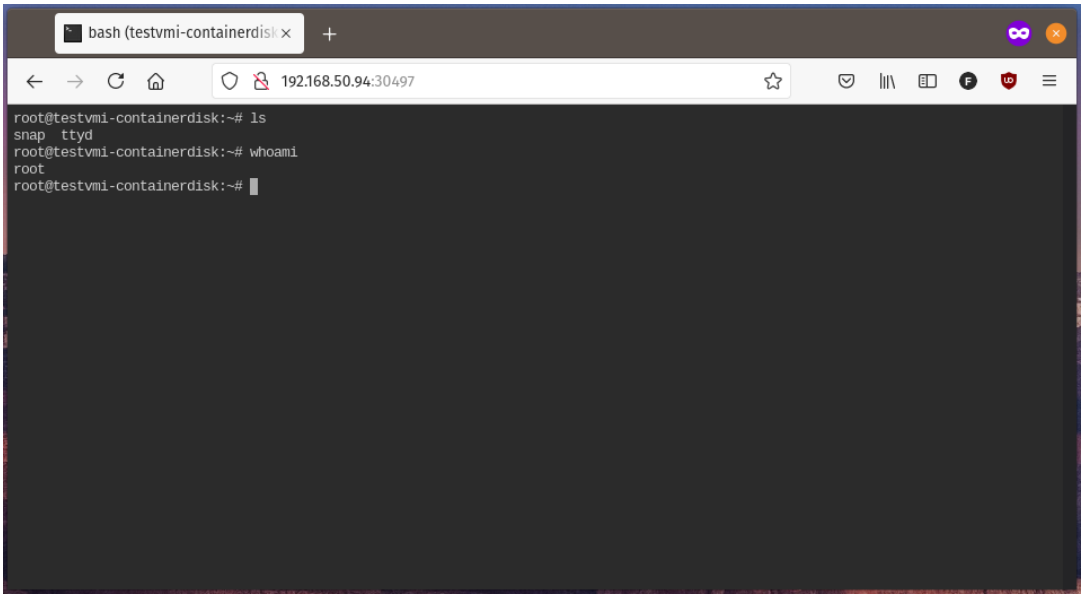

Figure 12: ttyd running inside the container

This allows us to create custom images and access them with any software, for example bash, zsh, python, nodejs. This solution is very customizable, but it's not possible to share the system console which shows e.g. the boot process.

4.3.2 ttyd outside VM

The second way is to run ttyd outside of the container and run `ttyd kubectl virt console your_vmi_name`. This allows to share the console of the VM with ttyd and this includes the boot process of the VM. VM developers can't customize which command is executed here, because this is run on the host machine. Also you aren't logged in

automatically. It may be possible to run this in a second container that maintains the VMs but that for another time.

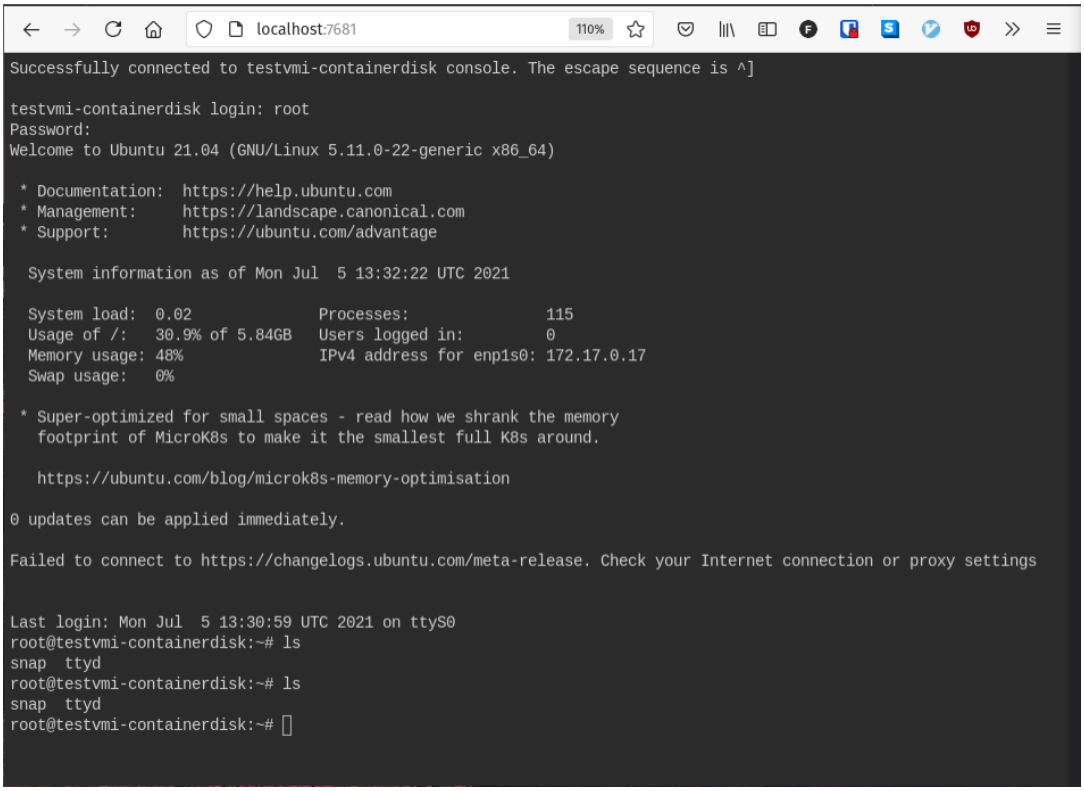

Figure 13: ttyd running outside the container

4.4 Web VNC Access

4.4.1 Tools

noVNC is a VNC viewer that runs in the browser. Usually VNC uses TCP sockets, but noVNC needs websockets. There is a tool called websockify which converts TCP sockets to websockets. This can be used to connect noVNC to any VNC server. [50]

KubeVirt has a command that creates a VNC server and opens a VNC client for any VMI: `kubectl virt vnc your_vmi_name`. If you want to connect your own VNC client there is the command kubectl `virt vnc your_vmi_name --proxy-only`. This creates a VNC server for the VMI with an TCP proxy. [51]

There is also a tool called virtVNC[47]. This tool can be used to access the VMIs graphical console using noVNC and combines the above two tools.

4.4.2 Preparing images

VNC enables us to use the desktop of the system. So we need to have images with a desktop environment installed. There are two ways of achieving this: 1. install desktop environment in cloud-images or 2. use images that already have a desktop environment installed.

4.4.3 Preparing desktop images with cloud-init

We will use an ubuntu 20.04 desktop image in this step with gnome installed. You can download the image from here[48] or use other images. Open the image in gnome boxes and install it. After the installation and configuration of the system start it and install cloud-init. An example installation tutorial for ubuntu can be found here[49]. After the installation of ubuntu and cloud-init stop the VM and copy the qcow2 image to your working folder, add it to a docker image and push it to docker hub like described earlier.

While starting the image in minikube an error occurred in my system: "no space left on device". This is because minikube runs in a VM that has a fixed size of 16GB by default, only 2GB left free and the image has 10GB. You can check how much space is available inside minikube by executing `minikube ssh` and then `df -h`. To fix this issue, you need to stop and delete your minikube instance and then create a new with the parameter `--disk-size=XXGB` where `XX` is the size you want. [54] [55] Commands:

1. `minikube stop`
2. `minikube delete`
3. `minikube start --vm-driver=kvm2 --disk-size=50GB`
4. Then reinstall everything: KubeVirt, virtVNC, etc.

Now start this image in Kubernetes. This may take a while because you need to download the image which might be very big. You can check if VNC is working by executing `kubectl virt vnc your_vmi_name`. If you can see the desktop it's working. [53]

[47] https://github.com/wavezhang/virtVNC
[48] https://ubuntu.com/download/desktop/thank-you?version=20.04.2.0&architecture=amd64
[49] https://zoomadmin.com/HowToInstall/UbuntuPackage/cloud-init

After connecting to the VNC of the Ubuntu machine you are able to login. The login and loading of Gnome takes some time. If it is very slow you may have insufficient RAM, then you need to delete the cluster, go back to the installation and reinstall the cluster with increased RAM.

4.4.4 Preparing cloud images with desktop environment

In this step we use the previously build ubuntu cloud image, that has be resized. To install gnome-desktop you need at least 2.2GB free space on the VM. To install gnome-desktop in ubuntu cloud image start it in boxes and then execute `apt install ubuntu-gnome-desktop`. After that shutdown the VM, copy the qcow2 image to your working folder, add it to a docker image and push it to docker hub like described earlier. Now start this image in Kubernetes with enough memory (e.g. 2GB). You can check if VNC is working by executing `kubectl virt vnc your_vmi_name`. If you can see the desktop and login it's working. [52]

To install other desktop environments you can follow the steps in this guide: Ubuntu cloud desktop adding gui to your cloud server instance[50].

4.4.5 Connect noVNC to kubectl vnc

Start the VNC proxy with `kubectl virt vnc your_vmi_name --proxy-only`. In the output of the command the port where the VNC server is reachable is shown. Now you can access the VNC with your VNC viewer. If you connect with an RDP program, or if you disconnect the VNC viewer, the kubectl proxy will break. This may be a problem, because it seems to be easy to break this solution. Also if you restart, a new random port will be used if you don't specify a fixed port with `--port`.

Install noVNC from github.com/novnc/novnc. You can run noVNC with `sudo novnc --listen 6081 --vnc localhost:40753`, where 6081 is the port where noVNC will be reachable and 40753 is the port where the VNC server is reachable. When starting this and the VNC server doesn't support websockets, websockify will automatically be started by noVNC. Now you can open this in your browser and connect to your VM.

`kubectl` is only a tool that wraps around the Kubernetes API. `kubevirt vnc` should executes some API commands to get a VNC connection. Maybe we can use this to directly get the VNC connection without kubevirt. This can then be used in our library.

noVNC can also be run as a service and listen on multiple ports and connecting to multiple VNC servers. It's also possible to embed noVNC into our own application[51].

4.4.6 virtVNC

`virtVNC` can be installed by just applying their yaml file: `kubectl apply -f https://github.com/wavezhang/virtVNC/raw/master/k8s/virtvnc.yaml`. After this is deployed to you minikube cluster, you can see there is a new service installed with

[50] https://www.suhendro.com/2019/04/ubuntu-cloud-desktop-adding-gui-to-your-cloud-server-instance/
[51] https://github.com/novnc/noVNC/blob/master/docs/EMBEDDING.md

```
→  Documentation git:(master) ✗ kubectl virt vnc ubuntu-desktop --proxy-only --port 40753
{"port":40753}
{"component":"","level":"info","msg":"connection timeout: 1m0s","pos":"vnc.go:144","timestamp
":"2021-07-09T09:57:31.963647Z"}

→  Documentation git:(master) ✗ sudo novnc --listen 6081 --vnc localhost:40753
[sudo] password for marco:
Warning: could not find self.pem
Using installed websockify at /snap/novnc/6/bin/websockify
Starting webserver and WebSockets proxy on port 6081
WebSocket server settings:
  - Listen on :6081
  - Web server. Web root: /snap/novnc/6
  - No SSL/TLS support (no cert file)
  - proxying from :6081 to localhost:40753


Navigate to this URL:

    http://pop-os:6081/vnc.html?host=pop-os&port=6081

Press Ctrl-C to exit
```

Figure 14: noVNC and vnc proxy

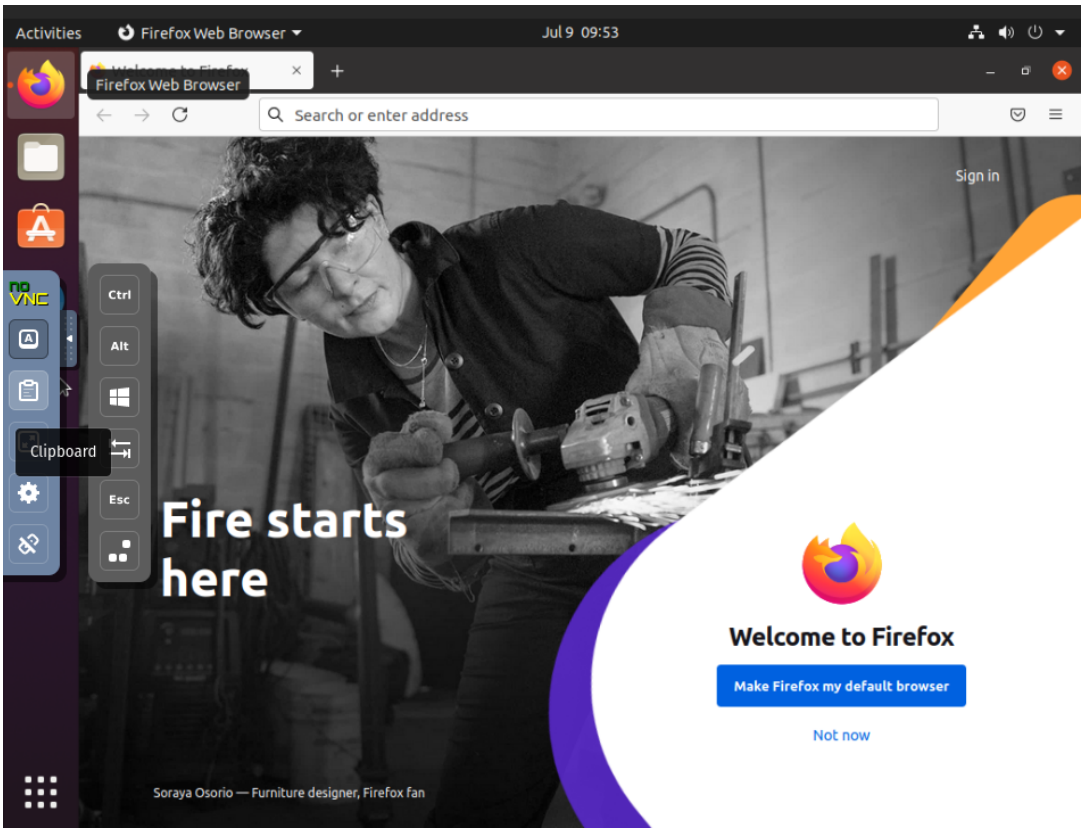

Figure 15: noVNC in browser

`kubectl get svc -n kubevirt virtvnc`. Execute `minikube service virtvnc -n kubevirt` to open `virtVNC` in your browser. [51]

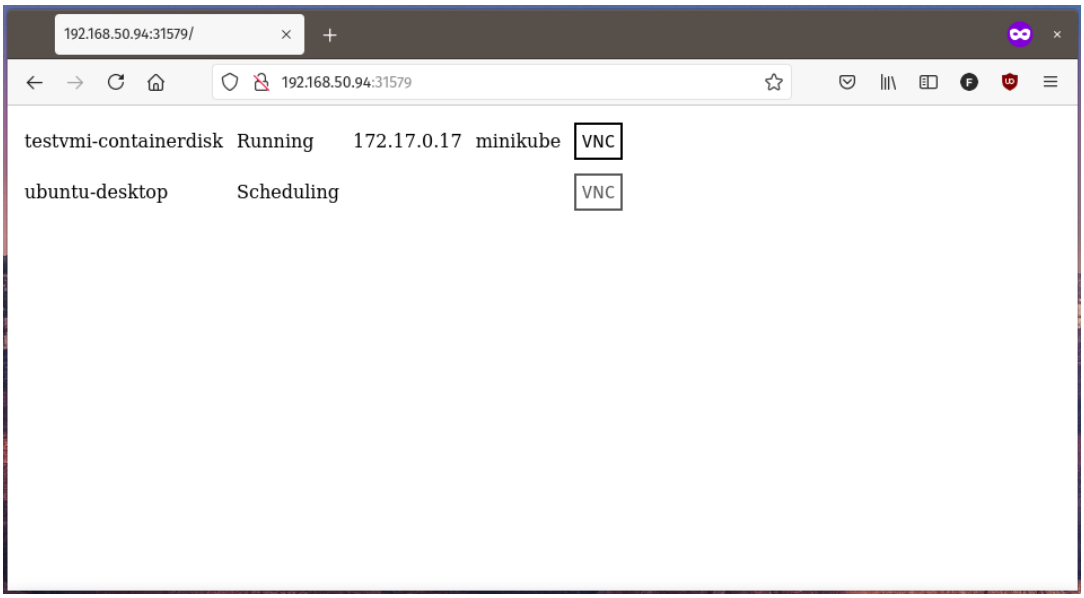

Figure 16: virtVNC list of VMs

`virtVNC` enables you to have a minimalistic dashboard, where you can see all running VMs and access their desktop over noVNC. You can filter the namespace by appending `?namespace=your_namespace` to the url. If VMs are currently not running but starting, for example if the image is downloading, they are shown with message `Scheduling`. An example of this is shown in Figure 16. [51]

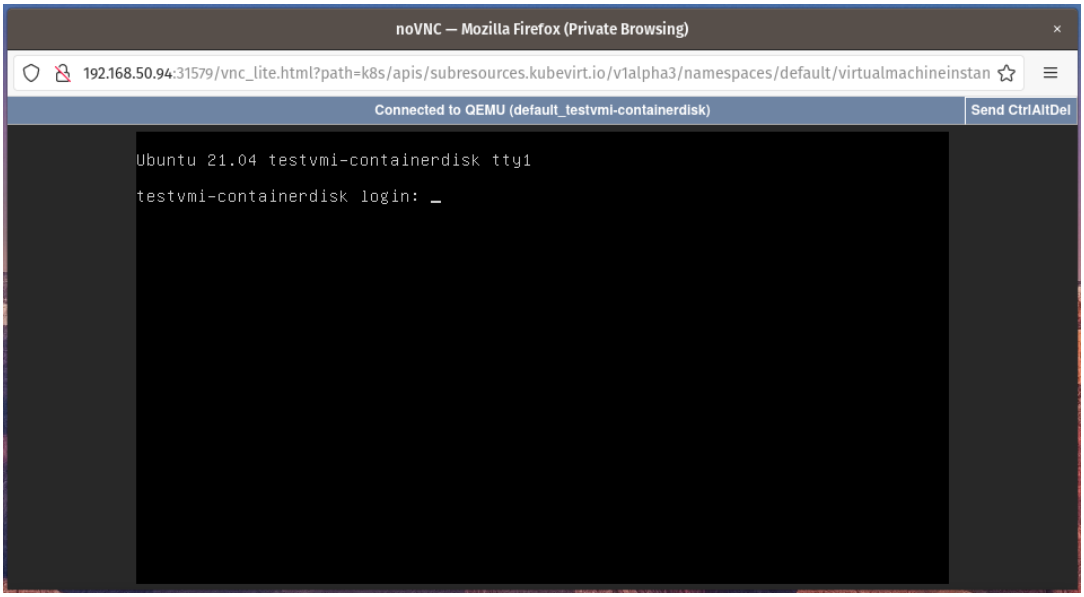

Figure 17: virtVNC showing a console

If you don't have a desktop environment installed, virtVNC gives you access to the console. This is seen in Figure 17.

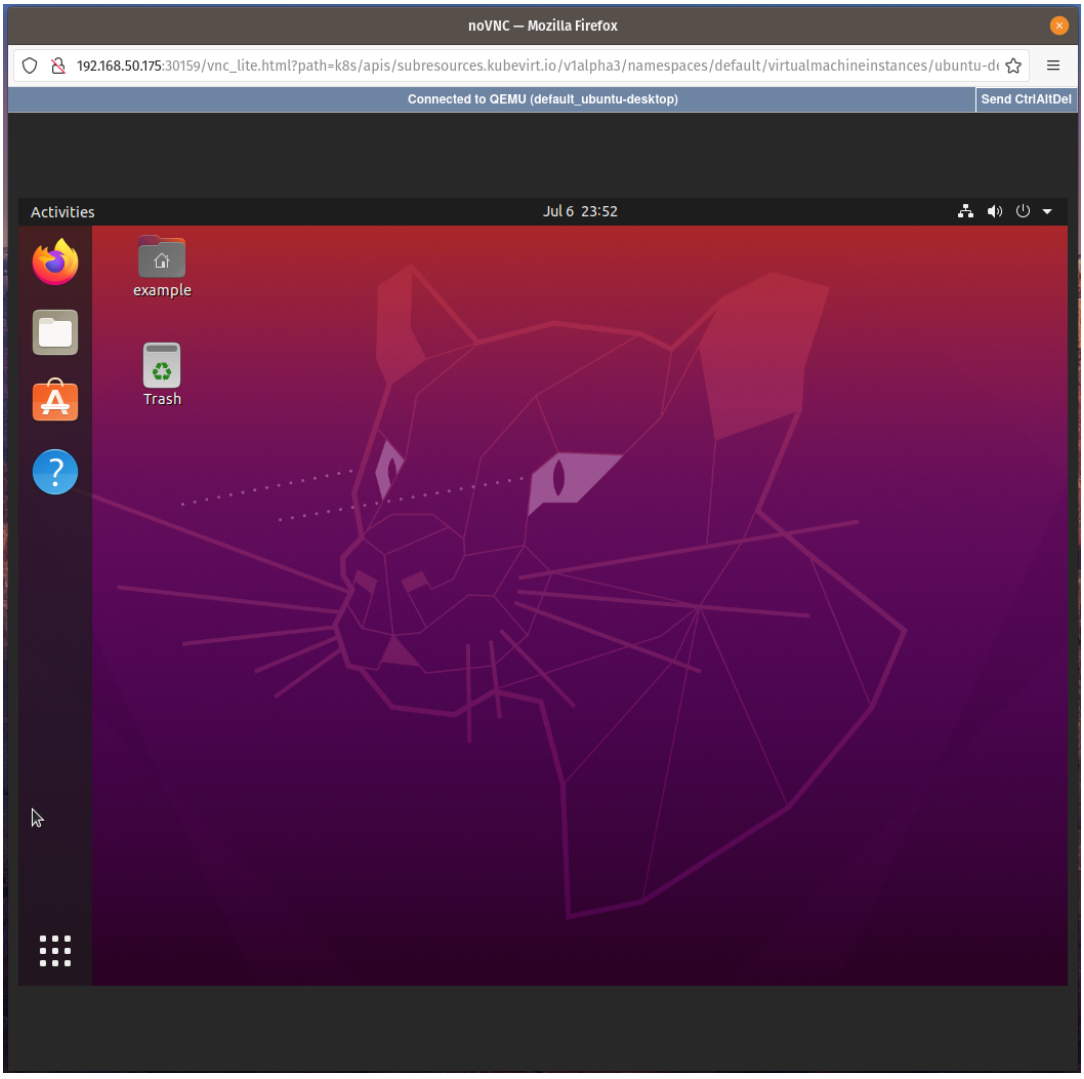

Figure 18: virtVNC showing Gnome

In the Figure 18 you can see the noVNC connection to the Ubuntu VM.

virtVNC doesn't have a permission system out of the box and it's possible to access any VNC console from any VM. In the lab orchestrator we need to be able to restrict users from accessing VMs that aren't theirs. Maybe we can extend virtVNC with a permission system or user authentication to restrict accessing every VM or build our own solution on top of the same principles like virtVNC. Maybe it will be enough to change the RBAC rules. Nevertheless, it is worth taking a look at how virtVNC works.

4.4.7 Directly accessing the API

To understand what's happening in the virtVNC pod we will take a look at the script that is running there. You can see it in the kubevirt docs about noVNC[52].

First of all there is a hint that KubeVirt provides Websocket api access for VNC under: `APISERVER:/apis/subresources.kubevirt.io/v1alpha3/namespaces/NAMESPACE/virtualmachineinstances/VM/vnc`. [59]

Kubernetes has an API and if you run Kubernetes with Minikube you can access the API over the ip of minikube: `minikube ip`. This API needs some authentication. To bypass the authentication you can run `kubectl proxy`. This gives you an ip and port where you can access the api without authentication.

What's also needed is a python program called SimpleHTTPServer. You can start this server with `python -m SimpleHTTPServer`. Now the server will run on `localhost:8000` and serve all files in the current directory. [60]

Download noVNC with `git clone https://github.com/novnc/noVNC`. Then `cd` into the folder and run `python -m SimpleHTTPServer`. Now you are serving the noVNC files over your localhost. Check it by opening `localhost:8000/vnc.html` in your browser. If you see the noVNC client, it's working. [59]

`vnc.html` has three parameters that are needed by us: `host`, `port` and `path`. `host` and `port` refers to the host and port the websocket has to connect to. `path` is the path that should be used. noVNC will build the websocket url as `host:port/path`. Other parameters can be found here: Embedding and Deploying noVNC Application[53]. [61]

Now you have a running noVNC server and can use the Kubernetes API unauthenticated. In my case `kubectl proxy` runs on port 8001 and noVNC (with SimpleHTTPServer) on 8000.

The Kubernetes API server lets you query and manipulate the state of API objects in Kubernetes. Every resource we have used is available over the Kubernetes API and can be created and viewed over the API. For example pods, namespaces, VMs and volumes are all API objects. [62]

The Kubernetes API is a self describing API. And there are some URLs that explain the sub-URLs. For example some of the API methods/resources that we can use are listed in the following two URLs:

[52] https://kubevirt.io/2019/Access-Virtual-Machines-graphic-console-using-noVNC.html
[53] https://github.com/novnc/noVNC/blob/master/docs/EMBEDDING.md

- `http://localhost:8001/apis/kubevirt.io/v1alpha3/`
- `http://localhost:8001/apis/subresources.kubevirt.io/v1alpha3`

The first contains KubeVirt resources like VMs and VMIs and the second contains KubeVirt subresources like the VNC and console. Notice: The second URL does not work if you append a trailing / to the URL.

As we already know some resources are available within namespaces, e.g. pods, VMs and VMIs. The API is structured like `<host>:<port>/apis/<resource_api>/<api_version>`. This will list all methods and resources available in this resource api. Then, when you want to open a resource and the resource is namespaced, it's structured like `<host>:<port>/apis/<resource_api>/<api_version>/namespaces/<namespace>/<resource>` and if you call a subresource `<host>:<port>/apis/<resource_api>/<api_version>/namespaces/<namespace>/<resource>/<subresource>`.

To get all VMIs from namespace default you can make a GET request to the URL: `http://localhost:8001/apis/kubevirt.io/v1alpha3/namespaces/default/virtualmachineinstances/`. VirtVNC uses this in the script to list all VMIs with status and name of the VMI. The result of the request contains information about the VMI. This contains the name (`metadata.name`) of the VMI, the phase (`status.phase`; e.g. "Running", "Failed"), the nodeName where the VMI is running (`status.nodeName`), the internal ip of the VMI (`status.interfaces[0].ipAddress`) and many other information for example the cloud-init configuration that contains our root password. This information maybe needs to be hidden from the user. The VMI name is used to identify the VMI in other calls. For example you can make a GET request to `http://localhost:8001/apis/kubevirt.io/v1alpha3/namespaces/default/virtualmachineinstances/<your_vmi_name>`. This gives you only the details to the specific VMI and not a list of all VMIs.

VirtVNC uses the above information to display all VMIs in the select page. Now when you click on one of the VMIs in VirtVNC it opens a new window with a link to the `vnc_lite.html` and the parameter `path` set to `apis/subresources.kubevirt.io/v1alpha3/namespaces/default/virtualmachineinstances/<your_vmi_name>/vnc`. This API URL is the one pointed out at the beginning that contains the Websocket api access for VNC.

To test if your WebSocket works, you first need to assure that the VMI is running. Then you can execute `curl --header "Connection: Upgrade" --header "Upgrade: websocket" --header "Sec-WebSocket-Version: 13" --header "Sec-WebSocket-Key: SomeKey" --include --no-buffer localhost:8001/apis/subresources.kubevirt.io/v1alpha3/namespaces/default/virtualmachineinstances/<your_vmi_name>/vnc`. If curl doesn't throw an exception and remains in a running connection, the Websocket works. [63]

Figure 19 shows the `kubevirt proxy` in the first window, the SimpleHTTPServer serving the noVNC client in the second window and a working curl-Websocket test in the third window.

Now you can open you own noVNC Url in the browser (`http://localhost:8000/vnc.html`) with the parameters `host=localhost`, `port=8001` and `path=apis/subresources.kube-`

```
➜  Documentation git:(master) ✗ kubectl proxy
Starting to serve on 127.0.0.1:8001


➜  noVNC git:(master) python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...


➜  Documentation git:(master) ✗ curl --header "Connection: Upgrade" --header "Upgrade: websoc
ket" --header "Sec-WebSocket-Version: 13" --header "Sec-WebSocket-Key: SomeKey" --include --n
o-buffer localhost:8001/apis/subresources.kubevirt.io/v1alpha3/namespaces/default/virtualmach
ineinstances/ubuntu-cloud-gnome/vnc
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: 50e5G6pmEIJWlEuAeY7ROmNH1r0=


RFB 003.008
```

Figure 19: Testing of Websockets with curl

`virt.io/v1alpha3/namespaces/default/virtualmachineinstances/<your_vmi_name>/vnc`. In my case this results in the URL: `http://localhost:8000/vnc.html?host=localhost&port=8001&path=apis/subresources.kubevirt.io/v1alpha3/namespaces/default/virtualmachineinstances/ubuntu-cloud-gnome/vnc`.

Figure 20: Selfhostet noVNC interface

The figures 20-22 shows our own custom noVNC instance connected to the VNC of the KubeVirt VMIs over the Kubernetes API.

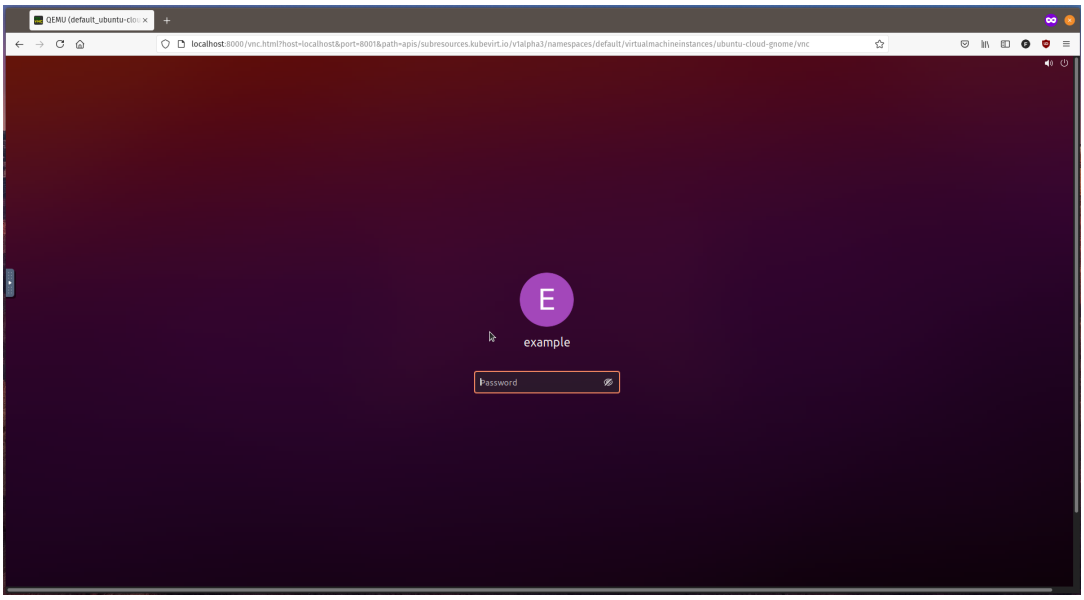

Figure 21: Selfhostet noVNC connected to VMI with login screen

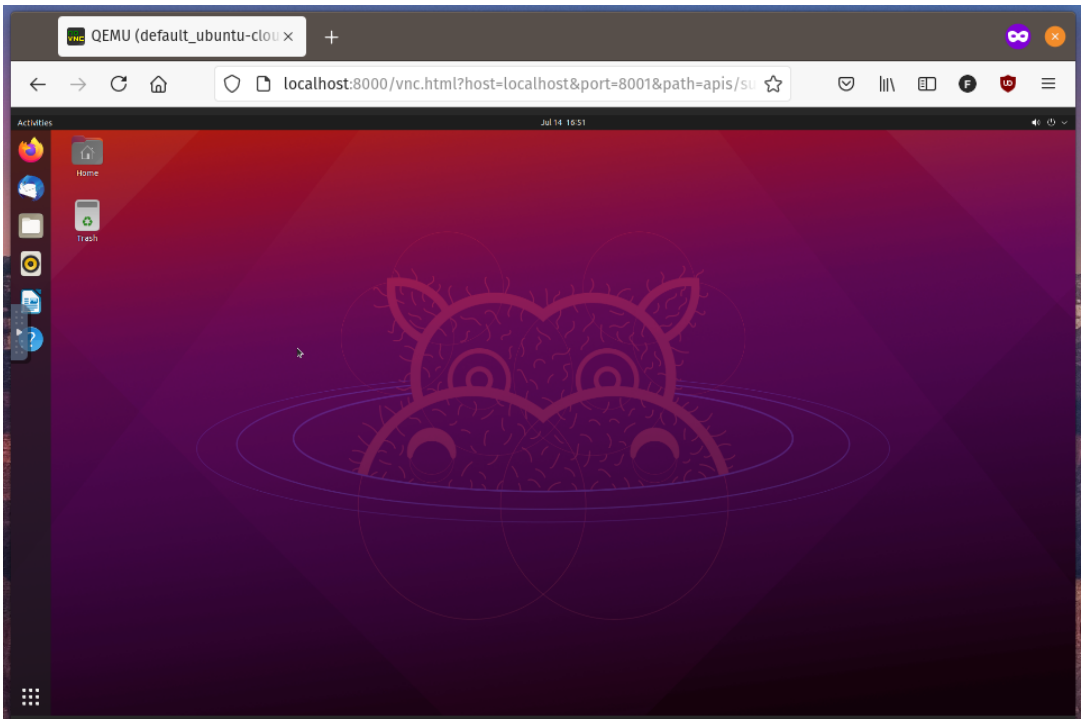

Figure 22: Selfhostet noVNC connected to VMI logged in

4.5 Separation of labs

In this step we want to achieve a separation of labs. That means that if you are connected to one lab, you can't access another lab from there. For example you are connected to a VMI in the lab 1 and the VMI in lab 2 has the ip 10.244.120.80 and you try to ssh into the machine it should not work.

To achieve this NetworkPolicy is needed. NetworkPolicy only work if you are running a network plugin that supports network policies for example calico. If you are using Minikube you can start the cluster with the network plugin calico with the following parameter: `--cni=calico`. If you haven't done this during the first start of Minikube, you need to delete your cluster and create a new one. If you don't do this, Kubernetes won't throw any errors in the next steps, but in the last step it will not fail when you ping the VMI in namespace `lab1`. [65] [66] [67]

To check if we can reach other namespaces create a new namespace using `kubectl create namespace testing`. Then create a second VMI yaml file and add `namespace: testing` to `metadata`. Then start the second VMI. Now you should have two VMIs: You can see the first one when you execute `kubectl get vmi` and the second when you execute `kubectl get vmi -n testing`. The command also shows you the ips of the VMIs. For reference my VMIs are called `ubuntu-cloud-gnome` and `ubuntu-cloud-gnome2`. Also the images have a default password set for the user `example` with the gnome-boxes way which was explained earlier.

Now connect to the console of one VMI: `kubectl virt console ubuntu-cloud-gnome2 -n testing` or `kubectl virt console ubuntu-cloud-gnome`. Then connect to the ip of the other machine over ssh: `ssh example@172.17.0.18`. If that works, the connection between namespaces is allowed. With this method we can check if the network policies work. Now delete the namespace `testing` and the two VMIs.

In the Lab Orchestrator every lab gets its own namespace and in this namespace there can be multiple VMs that you can connect to. To depict this create two namespaces called `lab1` and `lab2`.

Network policies are bound to a namespace and only apply to this namespace. So if you create a network policy in the namespace `lab1` that denies all traffic to other namespaces, this only affects the namespace `lab1`. Because of this behavior we need to create one of these network policies in every namespace that we use as a lab. The network policy configuration that denies traffic to other namespaces can be found in the examples of KubeVirts NetworkPolicy Documentation[54]. Create two of them, one for namespace `lab1` and one for `lab2`. You can display them with the command `kubectl get networkpolicy --all-namespaces` like in Figure 23. [67]

[54] https://kubevirt.io/user-guide/virtual_machines/networkpolicy/

```
→ prototype git:(master) ✗ kubectl get networkpolicy --all-namespaces
NAMESPACE   NAME                    POD-SELECTOR   AGE
lab1        allow-same-namespace    <none>         13m
lab2        allow-same-namespace    <none>         13m
→ prototype git:(master) ✗
```

Figure 23: NetworkPolicies in Namespaces lab1 and lab2

Listing 21 Create two namespaces (poc/examples/namespaces.yaml)

```yaml
kind: Namespace
apiVersion: v1
metadata:
  name: lab1
---
kind: Namespace
apiVersion: v1
metadata:
  name: lab2
```

Listing 22 NetworkPolicy allow same namespace (poc/examples/network_policy_allow_-same_namespace.yaml)

```yaml
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  namespace: lab1
  name: allow-same-namespace
spec:
  podSelector:
    matchLabels:
  ingress:
  - from:
    - podSelector: {}
---
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  namespace: lab2
  name: allow-same-namespace
spec:
  podSelector:
    matchLabels:
  ingress:
  - from:
    - podSelector: {}
```

Listing 19 creates two namespaces: `lab1` and `lab2`. Listing 20 creates two network policies that denies all traffic to other namespaces for the namespaces `lab1` and `lab2`. [64]

After that you can start some VMIs in these namespaces. For example we run one VMI in `lab1` and two in `lab2`. Now get the ip addresses of them with the command `kubectl get vmi --all-namespaces` like in Figure 24.

```
Every 2,0s: kubectl get vmi --all-namespaces    pop-os: Thu Jul 15 17:07:36 2021

NAMESPACE   NAME                  AGE     PHASE     IP              NODENAME
lab1        ubuntu-cloud-gnome    12m     Running   10.244.120.77   minikube
lab2        ubuntu-cloud-gnome    2m27s   Running   10.244.120.79   minikube
lab2        ubuntu-cloud-gnome2   2m27s   Running   10.244.120.80   minikube
```

Figure 24: Multiple VMIs in different namespaces

To check if the network policies work connect to one VMI in `lab2` with the command `kubectl virt console ubuntu-cloud-gnome -n lab2`. Then ping the ip of the VMI in `lab1`. If that doesn't work, the policy is working. After that ping the ip of the second VMI in `lab2`. That should work. Figure 25 shows an example of this step.

```
root@ubuntu-cloud-gnome:~# ip a | grep "inet 10"
    inet 10.244.120.79/32 scope global dynamic enp1s0
root@ubuntu-cloud-gnome:~# ping 10.244.120.77
PING 10.244.120.77 (10.244.120.77) 56(84) bytes of data.
^C
--- 10.244.120.77 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4095ms

root@ubuntu-cloud-gnome:~# ping 10.244.120.80
PING 10.244.120.80 (10.244.120.80) 56(84) bytes of data.
64 bytes from 10.244.120.80: icmp_seq=1 ttl=63 time=1.05 ms
64 bytes from 10.244.120.80: icmp_seq=2 ttl=63 time=1.83 ms
64 bytes from 10.244.120.80: icmp_seq=3 ttl=63 time=0.366 ms
64 bytes from 10.244.120.80: icmp_seq=4 ttl=63 time=0.767 ms
64 bytes from 10.244.120.80: icmp_seq=5 ttl=63 time=0.758 ms
64 bytes from 10.244.120.80: icmp_seq=6 ttl=63 time=0.345 ms
^C
--- 10.244.120.80 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5010ms
rtt min/avg/max/mdev = 0.345/0.852/1.830/0.500 ms
root@ubuntu-cloud-gnome:~#
```

Figure 25: Check if network policies are working

Now we are able to deploy multiple VMIs in different namespaces and separate these namespaces from each other, so that if you are connected to one namespace you can only navigate inside this namespace. For every user we need to create one namespace, one network policy and any amount of VMIs.

4.6 Authorization, Multi-user support and Routing

Because Kubernetes can be accessed through an API, we can wrap all API methods in an proxy application and add authorization in a different layer. This will be done in the prototype.

Multi-user support means that a user is only able to get access to its own labs. For this the separation of labs is a requirement. Also what's needed is authentication, which needs to be done in the application too. The multi-user support and routing can also be added in the proxy application and will also be done in the prototype.

These three parts may include Kubernetes resources, e.g. the routing can make use of Kubernetes ingresses. But it doesn't need to and can all be included in the application so it will not be part of the proof of concept but from the prototype.

4.7 Docker TODO

Because everything we want to achieve can already be done with VMs this step is just a bonus so that you can choose what you want to use.

4.7.1 Docker Basics

4.7.2 Building a custom Docker Image

4.7.3 Web access to terminal

4.7.4 Web access to graphical user interface

4.7.5 Separation of labs and authorization

Separation of labs works the same as with KubeVirt, because we use Kubernetes features for this. To check this start two docker images in different namespaces and create the same network policies as before. Then connect into one docker container and ping the other. If this doesn't work everything is fine.

Authorization was also just Kubernetes features and we don't need to change something for docker.

4.8 Conclusion TODO

4.8.1 Web access to terminal

For VMs we have two solutions to access the terminal: ttyd inside a VM and ttyd outside a VM. The first one …

4.8.2 Web access to graphical user interface

4.8.3 Separation of labs, authorization and routing

The separation of labs is done with network policies. We use one namespace for every lab and in this namespace a NetworkPolicy is created that isolates the VMIs and docker containers in this namespace from other namespaces.

TODO

5 Prototype

In this chapter we will abstract the concepts of the last chapter and include them in a prototype. The prototype will add additional concepts like authentication and multi-user support. The prototype should be used to deploy labs and should on the one hand prove that this project is feasible and on the other hand serve as a template for the alpha phase.

5.1 Deploying an API to the cluster TODO

The prototype will have an API and will be deployed with Kubernetes, so we first need an example API that will be deployed. For this we will use the Flask Quickstart Example[55]:

Listing 23 api.py

```
from flask import Flask

app = Flask(__name__)


@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"


app.run(host='0.0.0.0')
```

Then we need to install all dependencies. For this we use the `requirements.txt`:

Listing 24 requirements.txt

```
flask
```

The dependencies can be installed with the command `pip3 install -r requirements.txt`. Now this can be run with `python2 api.py`. Now open another terminal an make a request to the api: `curl localhost:5000`. The response should be `Hello, World!`.

After we have an API, we need to create a Docker Image that includes the API. For this we create a dockerfile. [69]

[55] https://flask.palletsprojects.com/en/2.0.x/quickstart/

Listing 25 dockerfile

```
FROM python:3

WORKDIR /app

COPY requirements.txt /app
RUN pip install --no-cache-dir -r requirements.txt


COPY api.py /app

CMD ["python", "api.py"]
```

Then build and push it to docker hub: `docker build -t username/repo:version .` and `docker push username/repo:version`.

Now we have an API in a docker container in docker hub, that needs to be integrated in a pod. We will do this with a deployment and take the virtVNC deployment as base. We also need a service to connect to the API and we will run this in a different namespace called `lab-controller`. This will also be included in the yaml. [59] [70] [9]

Listing 26 api-deploy.yaml

```yaml
kind: Namespace
apiVersion: v1
metadata:
  name: lab-controller
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: helloworldapi-deployment
  namespace: lab-controller
spec:
  replicas: 1
  selector:
    matchLabels:
      app: helloworldapi
  template:
    metadata:
      labels:
        app: helloworldapi
    spec:
      containers:
        - name: helloworldapi
          image: USERNAME/REPO:VERSION
          imagePullPolicy: Always
          ports:
          - containerPort: 5000
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: helloworldapi
  name: helloworldapi
  namespace: lab-controller
spec:
  selector:
    app: helloworldapi
  ports:
    - port: 80 # incoming
      protocol: TCP
      targetPort: 5000 # port in the pod
      nodePort: 30001
  type: NodePort
```

Now we have our API running in Kubernetes. To test it, we can execute `minikube service --url helloworldapi -n lab-controller` and open the link in our browser. If the browser shows `Hello World` it has worked.

5.2 Using the Kubernetes API TODO

So for now we have an hello world API running in our cluster that is accessible through a `NodePort`. Next we need to get access to the Kubernetes API.

5.2.1 Authorization

To get access to the Kubernetes API from within a pod we need to create a `ServiceAccount` and to get access to different API resources we need to make use of the RBAC authorization. First move the namespace resource from `api-deploy.yaml` to its own file:

Listing 27 Example Namespace

```
1 kind: Namespace
2 apiVersion: v1
3 metadata:
4   name: lab-controller
```

Then add a yaml-file for the service account and the RBAC rules:

Listing 28 Example Service Account with RBAC

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: lab-controller-account
  namespace: lab-controller
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: lab-controller-account
  namespace: lab-controller
rules:
  - apiGroups:
      - subresources.kubevirt.io
    resources:
      - virtualmachineinstances/console
      - virtualmachineinstances/vnc
    verbs:
      - get
  - apiGroups:
      - kubevirt.io
    resources:
      - virtualmachines
      - virtualmachineinstances
      - virtualmachineinstancepresets
      - virtualmachineinstancereplicasets
      - virtualmachineinstancemigrations
    verbs:
      - get
      - list
      - watch
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: lab-controller-account
  namespace: lab-controller
subjects:
  - kind: ServiceAccount
    name: lab-controller-account
    namespace: lab-controller
roleRef:
  kind: ClusterRole
  name: lab-controller-account
  apiGroup: rbac.authorization.k8s.io
```

The Listing Example Service Account with RBAC first creates a `ServiceAccount` called `lab-controller-account`. This is the `ServiceAccount` we will use to connect to the API. Then it creates a `ClusterRole`, which contains permissions to access some API resources. This includes listing all KubeVirt VMs. Last it adds a `ClusterRoleBinding`, which binds the `ClusterRole` to the `ServiceAccount`. [75] [76] [59] [77]

Next we need to update our deployment to use the `ServiceAccount` `lab-controller-account`. First we remove the Namespace creation if you not already have done this. Then we will add `serviceAccountName: lab-controller-account` to `spec.template.spec` as you see in Line 18. Adding the service account with this way adds a folder to the pod `/var/run/secrets/kubernetes.io/serviceaccount/` which contains the file `token` which is the token we need to use for making requests to the api which authenticates us, the file `namespace` which is the namespace we are in and the file `ca.crt` which is a certificate we need to use to make sure the connection to the Kubernetes API is secure. Notice that we have also changed the name of the deployment and pod to `serviceaccountapi`. [74]

Listing 29 Example deployment with service account

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: serviceaccountapi-deployment
  namespace: lab-controller
spec:
  replicas: 1
  selector:
    matchLabels:
      app: serviceaccountapi
  template:
    metadata:
      labels:
        app: serviceaccountapi
    spec:
      serviceAccountName: lab-controller-account
      containers:
        - name: serviceaccountapi
          image: USERNAME/REPO:VERSION
          imagePullPolicy: Always
          ports:
          - containerPort: 5000
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: serviceaccountapi
  name: serviceaccountapi
  namespace: lab-controller
spec:
  selector:
    app: serviceaccountapi
  ports:
    - port: 80 # incoming
      protocol: TCP
      targetPort: 5000 # port in the pod
      nodePort: 30001
  type: NodePort
```

Apply both yaml files to your Cluster. Then list you pods in the namespace `lab-controller` with `kubectl get pods -n lab-controller` and connect to the bash of the pod we have created with `kubectl exec --stdin --tty serviceaccountapi-deployment-RANDOM -- /bin/bash`. Now we are connected to our pod and try to access

the Kubernetes API. [73]

In the pod there are some environment variables that we need. The first is `$KUBERNETES_SERVICE_HOST` which gives us the IP address of the Kubernetes API and the second is `$KUBERNETES_SERVICE_PORT` which contains the port. This needs to be combined with the files from the service account folder. The following `curl` command will list all running VMIs: [71] [72] [74]

Listing 30 Getting all VMIs

```
curl \
    --cacert /var/run/secrets/kubernetes.io/serviceaccount/ca.crt \
    --header "Authorization: Bearer $(cat
     ↪  /var/run/secrets/kubernetes.io/serviceaccount/token)" \
    https://$KUBERNETES_SERVICE_HOST:$KUBERNETES_SERVICE_PORT/apis/kube-
 ↪  virt.io/v1alpha3/namespaces/default/virtualmachineinstances/ |
 ↪  \
    python -m json.tool
```

When you execute the command you should receive a list of VMIs if you have added any. If you don't have any running VMIs it looks like this:

```
root@serviceaccountapi-deployment-58bc75d4dd-pj759:/app# curl \
>     --cacert /var/run/secrets/kubernetes.io/serviceaccount/ca.crt \
>     --header "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" \
>     https://$KUBERNETES_SERVICE_HOST:$KUBERNETES_SERVICE_PORT/apis/kubevirt.io/v1alpha3/namespaces/default/virtualmachinein
stances/ | \
>     python -m json.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   138  100   138    0     0  13800      0 --:--:-- --:--:-- --:--:-- 15333
{
    "apiVersion": "kubevirt.io/v1alpha3",
    "items": [],
    "kind": "VirtualMachineInstanceList",
    "metadata": {
        "continue": "",
        "resourceVersion": "303772"
    }
}
root@serviceaccountapi-deployment-58bc75d4dd-pj759:/app#
```

Figure 26: API access listing empty VMI list

This can now be used in our `api.py`.

5.3 Access the Kubernetes API in the Application

For now we have a simple hello world API application and we are able to access the Kubernetes API. This chapter is split up into three parts, will extend the prototype to first list all VMIs, then create new VMIs and last give us access to the console and VNC. In the first step we prepare the base that is needed to communicate with the Kubernetes API, i.e. reading the environment variables needed for configuration, reading the files containing the key, applying the ca cert. The second step extends the application with POST requests to create new resources. The challenge of the last step is that console and VNC uses websockets that we need to pass through the application and we need to serve the noVNC application.

The following chapters will explain what we have done to achieve the chapters goals and will split the source code of some files. You can read the full files on github^(https://github.com/LernmodulController/LernmodulController-Documentation/tree/master/prototype/examples/accessing_api).

5.3.1 Access API to list VMIs

For step one basic knowledge about flask and requests is required. You can get a quickstart into requests in the requests quickstart[56] and a quickstart into flask in the flask quickstart[57].

[56] https://docs.python-requests.org/en/latest/user/quickstart/
[57] https://flask.palletsprojects.com/en/2.0.x/quickstart/

Listing 31 api.py step 1 part 1

```
from flask import Flask, make_response
import requests
import os
import logging

app = Flask(__name__)


class KubernetesAPI:
    def __init__(self, kubernetes_service_host, kubernetes_service_port,
                 service_account_token=None, cacert=None,
    ↪ insecure_ssl=False):
        if service_account_token is None:
            logging.warning("No service account token.")
        if cacert is None:
            logging.warning("No cacert.")
        self.service_host = kubernetes_service_host
        self.service_port = kubernetes_service_port
        self.service_account_token = service_account_token
        self.cacert = cacert
        self.insecure_ssl = insecure_ssl

    def get(self, address):
        base_uri = f"https://{self.service_host}:{self.service_port}"
        headers = {"Authorization": f"Bearer
    ↪ {self.service_account_token}"}
        if self.insecure_ssl:
            verify = False
        elif self.cacert is None:
            verify = True
        else:
            verify = self.cacert
        response = requests.get(base_uri + address, headers=headers,
    ↪ verify=verify)
        return response.text

    def get_vmis(self, namespace=None):
        if namespace is None:
            namespace = "default"
        address = f"/apis/kubevirt.io/v1alpha3/namespaces/{names-
    ↪ pace}/virtualmachineinstances/"
        return self.get(address)
```

The Listing api.py step 1 part 1 shows us the first part of the `api.py` after we implemented

the goals of step 1. First we have added a class `KubernetesAPI` that wraps the Kubernetes API. It takes the host and port, the token, ca cert and a boolean to disable verification of ssl. This class contains a generic method `get` that makes requests to the api and by automatically setting the base URL to the right host and port, setting the authentication method to bearer token and using the ca cert file if provided. Based on this generic method the `KubernetesAPI` class will be extended with methods that access resources. An example is `get_vmis`, which uses the `get` method to get all VMIs. [81] [82] [83] [84]

Listing 32 api.py step 1 part 2

```
41 def create_kubernetes_api_default():
42     kubernetes_service_host = os.environ["KUBERNETES_SERVICE_HOST"]
43     kubernetes_service_port = os.environ["KUBERNETES_SERVICE_PORT"]
44     service_account_token = None
45     with open('/var/run/secrets/kubernetes.io/serviceaccount/token',
       ↪  'r') as reader:
46         service_account_token = reader.read()
47     cacert = '/var/run/secrets/kubernetes.io/serviceaccount/ca.crt'
48     k8s_api = KubernetesAPI(kubernetes_service_host,
    ↪  kubernetes_service_port,
49                             service_account_token, cacert)
50     return k8s_api
51
52
53 kubernetes_api = create_kubernetes_api_default()
```

In Listing api.py step 1 part 2 the second part of the `api.py` is shown. The method `create_kubernetes_api_default` reads the host and port from environment variables and the file that contains the token and creates a variable with the location of the ca cert file. Then it creates a default instance of the `KubernetesAPI` that we can use in the flask routes.

Listing 33 api.py step 1 part 3

```
61 @app.route("/vmis")
62 def get_vmis():
63     r = kubernetes_api.get_vmis()
64     ret = make_response(r)
65     ret.mimetype = 'application/json'
66     return ret
67
68
69 app.run(host='0.0.0.0')
```

In Listing api.py step 1 part 3 the third part of the `api.py` is shown. Here we add another route to flask called `/vmis`. This route will return all VMIs of one namespace. For now we will only have access to VMIs in namespace default, but this will be changed in the

next chapter: User Support. In this method the mimetype of the response is changed to `application/json`. In the response we return json and if we change the mimetype to json browsers, e.g. Firefox, will display them in comfortable way. [79] [80]

After the changes on the prototype you can rebuild and push the docker image and redeploy it to Kubernetes. You can get its URL with the command `minikube service --url serviceaccountapi -n lab-controller`. Now you can see a list of all VMIs that are running in the default namespace under `http://YOUR_URL:30001/vmis`. If you have no VMI running, you should start one to see an effect. To access the logs of your pod run the command `kubectl logs serviceaccountapi-deployment-RANDOM -n lab-controller`. [78]

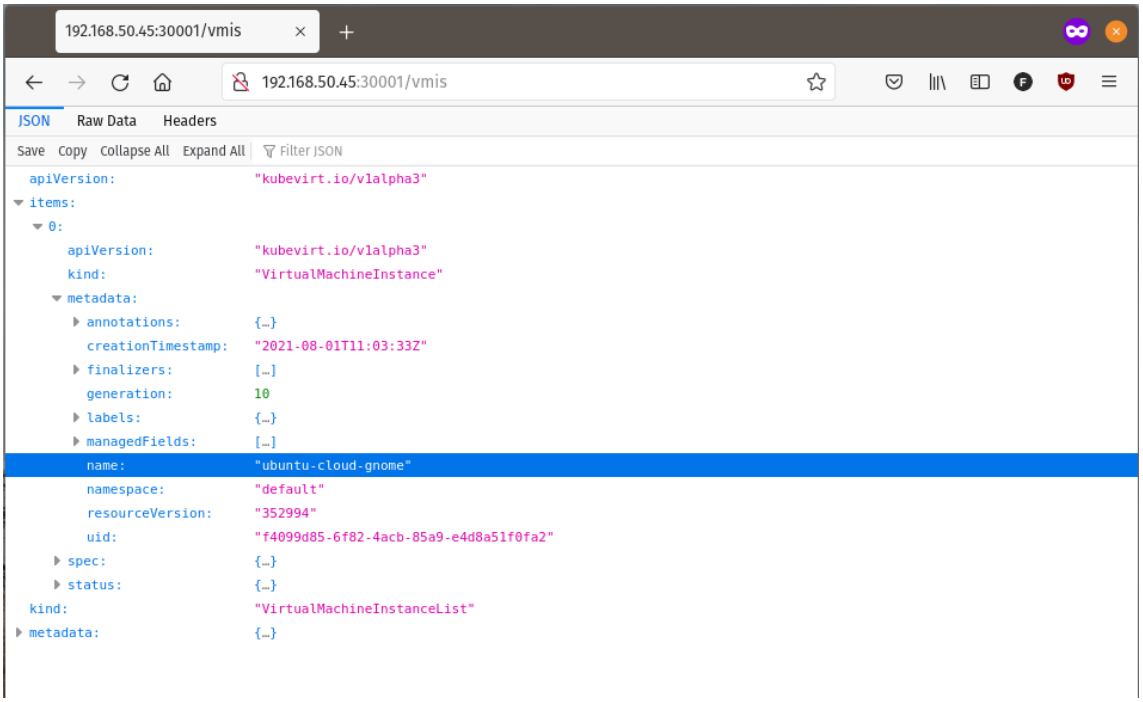

Figure 27: Application listing VMIs

The Figure Application listing VMIs shows the representation of `/vmis` in Firefox. This lists all VMIs currently running in the namespace default. As we see, there is one VMI running, the previously build `ubuntu-cloud-gnome`.

5.3.2 Access the API to run new VMIs

First we need to know how to create and delete VMIs over the Kubernetes API. The Kubernetes API Overview[58] gives us a good overview about how to get, create, replace and delete Kubernetes objects. What's missing there are the KubeVirt resources, but the concepts are the same. You get all objects of a resource with a GET request to the resource URI as we already know. To get a specific object you need to make a GET request to the object URI (resource uri + /name-of-object). To create new objects the request must make a POST request to the resource URI. The object data needs to be attached in the body and

[58] https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.19/

the header `Content-Type` must describe the format of the body. Accepted formats are yaml, json and protobuf. To delete an object you need to get the URI to the object and make a DELETE request. [85]

To test this, we start `kubectl proxy` and try creating and deleting a VMI with curl.

Listing 34 curl create vmi

```
curl \
    -X POST \
    --data-binary @"ubuntu_cloud_gnome.yaml" \
    -H "Content-Type: application/yaml" \
    http://localhost:8001/apis/kubevirt.io/v1alpha3/namespaces/de-
↪   fault/virtualmachineinstances/
```

Listing 35 curl create vmi 2

```
curl \
    -X POST \
    --data "$(cat ubuntu_cloud_gnome.yaml)" \
    -H "Content-Type: application/yaml" \
    http://localhost:8001/apis/kubevirt.io/v1alpha3/namespaces/de-
↪   fault/virtualmachineinstances/
```

Listing 36 ubuntu_cloud_gnome.yaml

```
metadata:
  namespace: default
  name: ubuntu-cloud-gnome
  labels:
    special: key
apiVersion: kubevirt.io/v1alpha3
kind: VirtualMachineInstance
spec:
  domain:
    cpu:
      cores: 3
    resources:
      requests:
        memory: 3G
    devices:
      disks:
      - name: containerdisk
        disk: {}
  volumes:
    - name: containerdisk
      containerDisk:
        image: USERNAME/REPO:VERSION
```

Listing curl create vmi shows a curl command that creates a VMI. It attaches the content of the file in Listing ubuntu_cloud_gnome.yaml to the body and sets the header `Content-Type` to `application/yaml`. When we run `kubectl get vmi` a new VMI is shown. Listing curl create vmi 2 does the same but it doesn't attach the file but attaches the file contents as string. This will have a benefit when implementing it in python, because we can generate the yaml string in python without saving it to a file. [86]

Listing 37 curl delete vmi

```
curl \
    -X DELETE \
    http://localhost:8001/apis/kubevirt.io/v1alpha3/namespaces/de-
    ↪ fault/virtualmachineinstances/ubuntu-cloud-gnome
```

Listing curl delete vmi shows a curl command that deletes the VMI that we have created before. When we run `kubectl get vmi` the VMI is succeeded or not displayed anymore.

Now we add this to the application, but first we need to extend the permission of our service user, because for now it is not allowed to create VMIs. In the next step we want to create and delete VMIs so we need to add `create` and `delete` to the ClusterRole of our ServiceAccount.

Listing 38 serviceaccount_version2.yaml

```yaml
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: lab-controller-account
  namespace: lab-controller
rules:
  - apiGroups:
      - subresources.kubevirt.io
    resources:
      - virtualmachineinstances/console
      - virtualmachineinstances/vnc
    verbs:
      - get
  - apiGroups:
      - kubevirt.io
    resources:
      - virtualmachines
      - virtualmachineinstances
      - virtualmachineinstancepresets
      - virtualmachineinstancereplicasets
      - virtualmachineinstancemigrations
    verbs:
      - get
      - list
      - watch
      - create
      - delete
```

Listing serviceaccount_version2.yaml shows the changed ClusterRole that contains create and delete permissions. This needs to be applied. [76]

Now we will add a VMI template to the docker image. This is used to deploy VMIs later.

Listing 39 vmi_template.yaml

```
metadata:
  namespace: ${namespace}
  name: ${vmi_name}
  labels:
    special: key
apiVersion: kubevirt.io/v1alpha3
kind: VirtualMachineInstance
spec:
  domain:
    cpu:
      cores: ${cores} #3
    resources:
      requests:
        memory: ${memory} #3G
    devices:
      disks:
      - name: containerdisk
        disk: {}
  volumes:
    - name: containerdisk
      containerDisk:
        image: ${vm_image}
```

The template in Listing vmi_template.yaml contains some variables: `${namespace}`, `${vmi_name}`, `${cores}`, `${memory}` and `${vm_image}`. `namespace` is the namespace this VMI is deployed to. `vmi_name` is its name. `cores` and `memory` are used to specify how much memory and cores the machine can use. `cores` needs to be an integer otherwise the api will throw an error. `vm_image` is the image location of our docker hub image in the format `USERNAME/REPO:VERSION`.

This needs to be added to the dockerfile if you don't copy all files from this directory in it. Also add `pyyaml` to the `requests.txt`.

After that we extend our `KubernetesAPI` class with delete and create possibilities, add new routes and a yaml template engine.

Listing 40 api-step2.py part 1

```
42      def post(self, address, data):
43          base_uri = f"https://{self.service_host}:{self.service_port}"
44          headers = {"Authorization": f"Bearer
    ↪ {self.service_account_token}",
45                     "Content-Type": "application/yaml"}
46          if self.insecure_ssl:
47              verify = False
48          elif self.cacert is None:
49              verify = True
50          else:
51              verify = self.cacert
52          response = requests.post(base_uri + address, data=data,
    ↪ headers=headers, verify=verify)
53          return response.text
54
55      def create_vmi(self, data, namespace=None):
56          if namespace is None:
57              namespace = "default"
58          address = f"/apis/kubevirt.io/v1alpha3/namespaces/{names-
    ↪ pace}/virtualmachineinstances/"
59          return self.post(address, data)
60
61      def delete(self, address):
62          base_uri = f"https://{self.service_host}:{self.service_port}"
63          headers = {"Authorization": f"Bearer
    ↪ {self.service_account_token}"}
64          if self.insecure_ssl:
65              verify = False
66          elif self.cacert is None:
67              verify = True
68          else:
69              verify = self.cacert
70          response = requests.delete(base_uri + address, headers=headers,
    ↪ verify=verify)
71          return response.text
72
73      def delete_vmi(self, vmi_name, namespace=None):
74          if namespace is None:
75              namespace = "default"
76          address = f"/apis/kubevirt.io/v1alpha3/namespaces/{names-
    ↪ pace}/virtualmachineinstances/{vmi_name}"
77          return self.delete(address)
```

The Listing api-step2.py part 1 shows us the part of the KubernetesAPI that we have

added. It contains four new methods: `post`, `create_vmi`, `delete` and `delete_vmi`. `post` and `delete` are two more generic methods that can be used to create and delete objects. To use the `post` method you need to add the data of the object in the yaml format. The `delete` method can delete resources. Due to currently missing permissions it's only possible to delete single objects and no collections, but the method will delete both when you add the correct permissions to the service account. The methods `create_vmi` and `delete_vmi` just prepares the URLs and then calls the generic methods.

Listing 41 api-step2.py part 2

```
95  class TemplateEngine:
96      def __init__(self, data):
97          self.path_matcher = re.compile(r'\$\{([^}^{]+)\}')
98          self.data = data
99
100     def path_constructor(self, loader, node):
101         value = node.value
102         match = self.path_matcher.match(value)
103         var = match.group()[2:-1]
104         val = self.data.get(var)
105         # needed to prevent converting integers to strings
106         if value[match.end():] == "":
107             return val
108         else:
109             return str(val) + value[match.end():]
110
111     def load_yaml(self, filename):
112         yaml.add_implicit_resolver('!path', self.path_matcher)
113         yaml.add_constructor('!path', self.path_constructor)
114
115         cont = open(filename)
116         p = yaml.load(cont, Loader=yaml.FullLoader)
117         return p
118
119     def replace(self, filename):
120         y = self.load_yaml(filename)
121         return yaml.dump(y, Dumper=yaml.Dumper)
```

The Listing api-step2.py part 2 shows the template engine. The template engine can read a yaml file and replace yaml variables with values from python variables. For this you need to create an object of the `TemplateEngine` and pass a data object into the class. This data object needs to be a dictionary that contains all variable names and its values. Then you can call the `load_yaml` method with the filename of the yaml file which gives you a yaml object with replaced variables. Alternatively you can directly call the `replace` method which will return the yaml as string instead of object. This is what we need for our creation of Kubernetes objects. [87] [88] [89]

Listing 42 api-step2.py part 3

```
137 @app.route("/create_vmi")
138 def create_vmi():
139     vm_image = request.args.get('vm_image', type=str)
140     vmi_name = request.args.get('vmi_name', type=str)
141     namespace = "default"
142     template_data = {"cores": 3, "memory": "3G",
143                      "vm_image": vm_image, "vmi_name": vmi_name,
144                      "namespace": namespace}
145     template_engine = TemplateEngine(template_data)
146     data = template_engine.replace('vmi_template.yaml')
147     r = kubernetes_api.create_vmi(data, namespace)
148     ret = make_response(r)
149     ret.mimetype = 'application/json'
150     return ret
151
152
153 @app.route("/delete_vmi")
154 def delete_vmi():
155     vmi_name = request.args.get('vmi_name', type=str)
156     namespace = "default"
157     r = kubernetes_api.delete_vmi(vmi_name, namespace)
158     ret = make_response(r)
159     ret.mimetype = 'application/json'
160     return ret
```

Last but not least in Listing api-step2.py part3 we can see the new routes. The first route is `/create_vmi`. In this method `request.args` is used to get the VMI name and the VM image location in docker hub from URL arguments. Then a dictionary is created that contains all key-value pairs that are needed for our template and a TemplateEngine object is initialized with this dictionary. Next the `replace` method gives us a string of the yaml template with all variables replaced with the values from the dictionary. This string is used to create a VMI with the `create_vmi` method. The `/delete_vmi` route is not that spectacular, it just gets an URL argument and calls the `delete_vmi` method. [90]

Build the image, push it and recreate the pod. Now we have implemented the create and delete feature for VMIs. We are able to create VMIs with opening this URL in the browser: `http://192.168.50.45:30001/create_vmi?vmi_name=ubuntu-cloud-gnome2&vm_image=USERNAME/REPO:VERSION` (replace the CAPSCASE with your VM image). This creates a VMI with the name `ubuntu-cloud-gnome2` and the VM image you specified. The VMI will be deployed to the default namespace. With `kubectl get vmi` you can see another VMI starting in our cluster. After it has started we can delete it with opening: `http://192.168.50.45:30001/delete_vmi?vmi_name=ubuntu-cloud-gnome2`. All in all this is a bad API design, but it's enough for the prototype.

5.3.3 Access VNC

First we need to create a websocket proxy in python, that creates one websocket connection to the Kubernetes api and another to the noVNC client that is accessible over the api. In the middle of this we can add our authentication. For the proxy we will use an example from github[59] and modify it. [92] [91]

Listing 43 ws_proxy-step3.py part 1

```
import argparse
import ssl
import sys
import threading
import asyncio
import websockets
import logging


__TOKEN_DB = []


def add_token(token, user, vmi_name):
    matches = [x for x in __TOKEN_DB if x["user"] == user and x["vmi"]
    ↪ == vmi_name]
    if len(matches) >= 1:
        return False
    __TOKEN_DB.append({"user": user, "vmi": vmi_name, "token": token})
    return True


def check_token(token, vmi_name):
    matches = [x for x in __TOKEN_DB if x["token"] == token]
    if len(matches) < 1:
        return False
    for match in matches:
        if match["vmi"] == vmi_name:
            return True
    return False
```

`TOKEN_DB` and the methods `add_token` and `check_token` are used as simple authentication mechanism. If a token is included in this list and associated with the VMI name, then the user is allowed to access the VNC. This may be changed in the next chapter. `add_token` is used to add credentials to the database.

[59] https://gist.github.com/bsergean/bad452fa543ec7df6b7fd496696b2cd8

Listing 44 ws_proxy-step3.py part 2

```
class WebsocketProxy:
    def __init__(self, remote_url, api_path, local_dev_mode):
        self.remote_url = remote_url
        self.api_path = api_path
        self.thread = None
        self.local_dev_mode = local_dev_mode

    def run(self, host, port):
        start_server = websockets.serve(self.proxy, host, port)
        asyncio.get_event_loop().run_until_complete(start_server)
        asyncio.get_event_loop().run_forever()

    def run_in_thread(self, host, port):
        start_server = websockets.serve(self.proxy, host, port)
        self.loop = asyncio.get_event_loop()
        self.loop.run_until_complete(start_server)
        self.thread = threading.Thread(target=self.loop.run_forever)
        self.thread.start()

    def stop_thread(self):
        if self.thread is not None:
            self.loop.call_soon_threadsafe(self.loop.stop)
            logging.info("Stopped loop")
            self.thread.join()
            logging.info("Stopped thread")
```

We have moved the code from the above example into a class called `WebsocketProxy`. During initialization you need to pass a `remote_url` and an `api_path`. The `api_path` needs to have `"{vmi_name}"` included because with `str.format` this will be inserted. With our Kubernetes API the `api_path` needs to be `"/apis/subresources.kubevirt.io/v1alpha3/namespaces/default/virtualmachineinstances/{vmi_name}/vnc"`. The method `run` will run the `ws_proxy` in foreground and block the python main thread. `run_in_thread` will run the `ws_proxy` in another thread so it won't block the main thread. This is needed because our main thread is used by flask. `stop_thread` is needed to stop the thread if the program is to be terminated.

Listing 45 ws_proxy-step3.py part 3

```
57      async def proxy(self, websocket, path):
58          '''Called whenever a new connection is made to the server'''
59          # split path to get vmi name and token
60          splitted = path.split("/")
61          if len(splitted) != 3:
62              logging.warning("Invalid URL")
63              await websocket.close(reason="invalid url")
64              return
65          vmi_name = splitted[1]
66          token = splitted[2]
67          # check if user has permissions to access this vmi
68          if not check_token(token, vmi_name):
69              logging.warning("Invalid token")
70              await websocket.close(reason="invalid token")
71              return
72          # build websocket url and connect to
73          url = self.remote_url + self.api_path.format(vmi_name=vmi_name)
74          if not self.local_dev_mode:
75              # adding selfsigned cert
76              ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
77              ssl_context.load_verify_-
    ↪ locations('/var/run/secrets/kubernetes.io/serviceaccount/ca.crt')
78              # adding bearer authorization
79              with open('/var/run/secrets/kubernetes.io/serviceaccount/to-
                ↪ ken', 'r') as
                ↪ reader:
80                  service_account_token = reader.read()
81                  header = {"Authorization": f"Bearer
    ↪ {service_account_token}"}
82              async with websockets.connect(url, ssl=ssl_context,
                ↪ extra_headers=header) as ws:
83                  taskA =
    ↪ asyncio.create_task(WebsocketProxy.clientToServer(ws, websocket))
84                  taskB =
    ↪ asyncio.create_task(WebsocketProxy.serverToClient(ws, websocket))
85                  await taskA
86                  await taskB
87          else:
88              async with websockets.connect(url) as ws:
89                  taskA =
    ↪ asyncio.create_task(WebsocketProxy.clientToServer(ws, websocket))
90                  taskB =
    ↪ asyncio.create_task(WebsocketProxy.serverToClient(ws, websocket))
91                  await taskA
92                  await taskB
```

The method `proxy` is called whenever a new connection to the `ws_proxy` is made. This method first checks the authentication. This is done by splitting the path by a divider and taking the first argument as VMI name and the second as token. This is a useful authentication, because in noVNC we can only specify the path and not for example special headers or other authentication mechanisms. So because of noVNC we are limited to make authentication with this trick. After authentication was successful the method opens a new websocket to the Kubernetes API. There are two ways for this, one is with SSL, where also the self signed certificate is included and the bearer token is attached. And a second way for local development without this. After that every message that is send to the `ws_proxy` within this websocket connection is redirected to the Kubernetes API and the other way around as well. So at this point we have two websocket connections: first client to `ws_proxy` and second `ws_proxy` to Kubernetes. These connections are kept alive and only the messages are redirected. [92] [91] [96] [95] [93] [94] [97] [99] [98] [98]

Listing 46 ws_proxy-step3.py part 3

```
94      async def clientToServer(ws, websocket):
95          async for message in ws:
96              await websocket.send(message)
97
98      async def serverToClient(ws, websocket):
99          async for message in websocket:
100             await ws.send(message)
```

The next two methods are just methods that redirect the traffic between the websockets. [92] [91]

Listing 47 ws_proxy-step3.py part 4

```
103 def main():
104     add_token("supersecret", "admin", "ubuntu-cloud-gnome")
105     remote_url = "ws://localhost:8001"
106     api_path = "/apis/subresources.kubevirt.io/v1alpha3/namespaces/de-
    ↪   fault/virtualmachineinstances/{vmi_name}/vnc"
107     wp = WebsocketProxy(remote_url, api_path, True)
108     wp.run("localhost", 8765)
109
110
111 if __name__ == '__main__':
112     main()
```

The line `if __name__ == '__main__':` checks if the file is executed directly or if it is included in another code. So this part only is executed if you run `python3 ws_proxy.py` and not if you include it in `api.py` and run `python3 api.py`. This is useful for development and testing of the module.

We will save our websocket proxy in an extra file `ws_proxy.py` for more modularity. We can later include it into the `api.py`. Now we can start the proxy with `python ws_proxy.py`. Now if we start a connection to `localhost:8765/VMI_NAME/TOKEN` this proxy creates a connection to the Kubernetes api to the VNC websocket from the VMI with name `VMI_NAME`. The token is then checked if you have the correct rights to connect to this VMI.

Now if we start the noVNC server with `python -m SimpleHTTPServer` we can access the proxied websocket at: `http://localhost:8000/vnc_lite.html?host=localhost&port=8765&path=ubuntu-cloud-gnome/supersecret`. If you change the token you won't get access. So as we see our websocket proxy is working and now we need to include it in the `api.py`.

First we will move the `TemplateEngine` into its own file `template_engine.py`. And then make some changes to the `api.py` so that we can run it with `kubectl proxy` which will simplify development. I will not go into details here, you can check the difference of the steps by yourself with `diff api-step1.py api-step2.py`.

Listing 48 api-step3.py part 1

```
123 def add_credentials():
124     user = request.args.get('user', type=str)
125     token = request.args.get('token', type=str)
126     vmi_name = request.args.get('vmi_name', type=str)
127     valid = add_token(token, user, vmi_name)
128     if valid:
129         return "Added credentials."
130     else:
131         return "Adding credentials not possible.", 400
```

This new route is used to add credentials that we need to connect to the VNC.

Listing 49 api-step3.py part 2

```
conf = {
    "websocket_remote_url": "ws://localhost:8001",
    "websocket_api_path": "/apis/subresources.kubevirt.io/v1alpha3/namespaces/default/virtualmachineinstances/{vmi_name}/vnc",
    "ws_proxy_host": "0.0.0.0",
    "ws_proxy_port": 5001,
    "flask_host": "0.0.0.0",
    "flask_port": 5000,
    "local_dev_mode": False,
}


def run(conf):
    wp = WebsocketProxy(conf["websocket_remote_url"], conf["websocket_api_path"], conf["local_dev_mode"])
    wp.run_in_thread(conf["ws_proxy_host"], conf["ws_proxy_port"])
    app.run(host=conf["flask_host"], port=conf["flask_port"], debug=False)
    wp.stop_thread()
```

First we have added a dictionary that contains some configuration for example the ports that should be used for `ws_proxy` and flask. This will later be changed to use environment variables. After that there is the method `run`, which will start the `ws_proxy` and flask and shuts the `ws_proxy` down after flask stopped.

We have (the last time) renamed the deployment and added the new port 5001.

Now add `websockets` to the `requirements.txt`. Then add the new files to the dockerfile and rebuild and push the docker image. Then update it in Kubernetes and open it.

Then we can call `minikube service --url lab-controller-api -n lab-controller` to get the URLs. We need to add our credentials with this URL: `192.168.50.45:30001/add_credentials?user=marco&token=geheim&vmi_name=ubuntu-cloud-gnome`

After that we can access the VNC in: `http://localhost:8000/vnc_lite.html?host=192.168.50.45&port=30002&path=ubuntu-cloud-gnome/geheim`

5.4 User Support

https://blog.miguelgrinberg.com/post/restful-authentication-with-flask https://github.com/miguelgrinberg/REST-auth

`curl -H "Content-Type: application/json" -X POST -d '{"username":"marco","password":"geheim"}' localhost:5000/api/users` [https://stackoverflow.com/questions/7172784/how-do-i-post-json-data-with-curl]

```
curl -u marco:geheim localhost:5000/api/resource

curl -u marco:geheim localhost:5000/api/token

{
  "duration": 600,
  "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6MSwiZXhwIjoxNjI4MjYyNDE5LjY4NDU
}

curl -u "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6MSwiZXhwIjoxNjI4MjYyNDE5LjY4NDU3ND
used localhost:5000/api/resource
```

https://flask-sqlalchemy.palletsprojects.com/en/2.x/models/ https://flask-sqlalchemy.palletsprojects.com/en/2.x/queries/ https://stackoverflow.com/questions/6750017/how-to-query-database-by-id-using-sqlalchemy https://flask-sqlalchemy.palletsprojects.com/en/2.x/quickstart/ https://stackoverflow.com/questions/41222412/sqlalchemy-init-takes-1-positional-argument-but-2-were-given-many-to-man https://stackoverflow.com/questions/10434599/get-the-data-received-in-a-flask-request https://en.wikipedia.org/wiki/List_of_HTTP_status_codes https://flask.palletsprojects.com/en/2.0.x/config/ https://blog.miguelgrinberg.com/post/restful-authentication-with-flask

Das Ergebnis

During this step we included a refactoring of the old code.

The KubernetesAPI was refactored and put into its own module called `kubernetes`. Before the refactoring, every Kubernetes API endpoint needed to be added into the KubernetesAPI class. Now you can write extensions and only need to add a class with the api URLs and register the class with an decorator. After the class is registered you can access it in the APIRegistry as a property. For example `APIRegistry(...).namespace` or `APIRegistry(...).virtual_machine_instances`.

Listing 50 kubernets/api.py Proxy

```
class Proxy:
    def __init__(self, base_uri, service_account_token=None,
                 cacert=None, insecure_ssl=False):
        if service_account_token is None:
            logging.warning("No service account token.")
        if cacert is None:
            logging.warning("No cacert.")
        self.base_uri = base_uri
        self.service_account_token = service_account_token
        if insecure_ssl:
            self.verify = False
        elif cacert is None:
            self.verify = True
        else:
            self.verify = cacert

    def get(self, address):
        headers = {"Authorization": f"Bearer
    ↪ {self.service_account_token}"}
        response = requests.get(self.base_uri + address,
                                headers=headers, verify=self.verify)
        return response.text

    def post(self, address, data):
        headers = {"Authorization": f"Bearer
    ↪ {self.service_account_token}",
                   "Content-Type": "application/yaml"}
        response = requests.post(self.base_uri + address,
                                 data=data, headers=headers,
    ↪ verify=self.verify)
        return response.text

    def delete(self, address):
        headers = {"Authorization": f"Bearer
    ↪ {self.service_account_token}"}
        response = requests.delete(self.base_uri + address,
                                   headers=headers, verify=self.verify)
        return response.text
```

The KubernetesAPI class is renamed into Proxy. This class sends request to the Kubernetes api and adds required headers and verifies the right certificate.

Listing 51 kubernets/api.py ApiExtension

```
73  class ApiExtension(ABC):
74      list_url = None
75      detail_url = None
76
77      def __init__(self, proxy: Proxy):
78          self.proxy = proxy
79
80
81  class NamespacedApi(ApiExtension):
82      def get_list(self, namespace):
83          return self.proxy.get(self.list_url.format(namespace=namespace))
84
85      def create(self, namespace, data):
86          return
            ↪  self.proxy.post(self.list_url.format(namespace=namespace),
            ↪  data)
87
88      def get(self, namespace, identifier):
89          return
            ↪  self.proxy.get(self.detail_url.format(namespace=namespace,
            ↪  identifier=identifier))
90
91      def delete(self, namespace, identifier):
92          return self.proxy.delete(self.detail_-
            ↪  url.format(namespace=namespace,
            ↪  identifier=identifier))
93
94
95  class NotNamespacedApi(ApiExtension):
96      def get_list(self):
97          return self.proxy.get(self.list_url)
98
99      def create(self, data):
100         return self.proxy.post(self.list_url, data)
101
102     def get(self, identifier):
103         return self.proxy.get(self.detail_-
            ↪  url.format(identifier=identifier))
104
105     def delete(self, identifier):
106         return self.proxy.delete(self.detail_-
            ↪  url.format(identifier=identifier))
```

The api has two different types of API endpoints. The ones that only work with names-

paces and the other that doesn't have a namespace. For example a namespace is a not namespaced resource and an VMI is a namespaced resource. The difference between these two types is how you build the URL. Every type has an identifier, but namespaced URLs has an namespace too. The ApiExtension class contains the basics for all api extensions and the NamespacedApi and NotNamespacedApi extends this to provide the two types of API endpoints. With this two abstract classes we are able to add any Kubernetes API endpoint to our library. [100]

Listing 52 kubernets/api.py Extensions

```
109 @add_api_not_namespaced("namespace")
110 class Namespace(NotNamespacedApi):
111     list_url = "/api/v1/namespaces"
112     detail_url = "/api/v1/namespaces/{identifier}"
113
114
115 @add_api_namespaced("virtual_machine_instance")
116 class VirtualMachineInstance(NamespacedApi):
117     list_url = "/apis/kubevirt.io/v1alpha3/namespaces/{namespace}/vir-
    ↪ tualmachineinstances/"
118     detail_url = "/apis/kubevirt.io/v1alpha3/namespaces/{namespace}/vir-
    ↪ tualmachineinstances/{identifier}"
119
120
121 @add_api_namespaced("network_policy")
122 class NetworkPolicy(NamespacedApi):
123     list_url =
    ↪ "/apis/networking.k8s.io/v1/namespaces/{namespace}/networkpolicies"
124     detail_url = "/apis/networking.k8s.io/v1/namespaces/{namespace}/net-
    ↪ workpolicies/{identifier}"
```

The listing kubernets/api.py Extensions shows three extensions we have added. One for the namespace resource, one for the VMIs and one for network policies. With this three extensions we are able to create, delete and get any of these resources. The extensions are registered with the decorators `add_api_not_namespaced` and `add_api_namespaced`. Without adding these decorators we are not able to use this extensions.

Listing 53 kubernetes/api.py decorators

```
API_EXTENSIONS_NAMESPACED: Dict[str, Type['NamespacedApi']] = {}
API_EXTENSIONS_NOT_NAMESPACED: Dict[str, Type['NotNamespacedApi']] = {}


def add_api_namespaced(name: str) -> Callable[[Type['NamespacedApi'],],
    Type['NamespacedApi']]:
    def inner(cls: Type[NamespacedApi]) -> Type[NamespacedApi]:
        API_EXTENSIONS_NAMESPACED[name] = cls
        return cls
    return inner


def add_api_not_namespaced(name: str) ->
    Callable[[Type['NotNamespacedApi'],], Type['NotNamespacedApi']]:
    def inner(cls: Type[NotNamespacedApi]) -> Type[NotNamespacedApi]:
        API_EXTENSIONS_NOT_NAMESPACED[name] = cls
        return cls
    return inner
```

The methods `add_api_namespaced` and `add_api_not_namespaced` return the decorators. Decorators are methods that get a function or class passed as argument. The return value of the decorator will replace the decorated function or class. So with decorators you are able to replace a function or class with another function. We use decorators here to add the passed class to a dictionary. The key of the dictionary is the string passed into the outer function, i.e. `namespace` in the Namespace Extension and network_policy in the NetworkPolicy Extension. The value is a reference to the class. We have two dictionaries here: one for the namespaced extensions and one for the not-namespaced extensions. The decorators return the same as they got passed into so that the function or class will not be replaced.

Listing 54 kubernetes/api.py APIRegistry

```
class APIRegistry:
    def __init__(self, proxy: Proxy):
        self.proxy = proxy

    def __getattr__(self, name) -> Union['NamespacedApi',
        'NotNamespacedApi']:
        if cls := API_EXTENSIONS_NAMESPACED.get(name):
            return cls(self.proxy)
        if cls := API_EXTENSIONS_NOT_NAMESPACED.get(name):
            return cls(self.proxy)
        raise AttributeError(f'{name} not found')
```

The `APIRegistry` can be initialized with an object of the `Proxy`. This class makes use of

the magic method `__getattr__`. In python when you call a method or get an attribute of an object, python executes the method `__getattribute__` with the name of the method or attribute as parameter if this method is defined. If `__getattribute__` is not defined python will look if the class has this attribute or method itself. If that is not the case, python will execute the `__getattr__` method if it is defined. With defining one of this methods you can dynamically process attributes. In the `APIRegistry` this is used to add new attributes to the class for every extension class that is in the dictionaries. Every `add_api_namespaced` decorator will add an attribute to this class with an instance of the decorated class. So if you want to create a namespace you can simply call `APIRegistry(...).namespace.create(...)` or if you want to get all VMIs you can call `APIRegistry(...).virtual_machine_instances.get_list(...)`.

Listing 55 model.py

```
6  class DockerImage(db.Model):
7      __tablename__ = 'docker_image'
8      id = sql.Column(sql.Integer, primary_key=True)
9      name = sql.Column(sql.String(32), unique=True)
10     description = sql.Column(sql.String(128))
11     url = sql.Column(sql.String(256))
12
13
14 class Lab(db.Model):
15     __tablename__ = 'lab'
16     id = sql.Column(sql.Integer, primary_key=True)
17     name = sql.Column(sql.String(32), unique=True)
18     namespace_prefix = sql.Column(sql.String(32), unique=True)
19     description = sql.Column(sql.String(128))
20     docker_image_id = sql.Column(sql.Integer,
    ↪  sql.ForeignKey('docker_image.id'))
21     docker_image_name = sql.Column(sql.String(32))
22
23
24 class LabInstance(db.Model):
25     __tablename__ = 'lab_instance'
26     id = sql.Column(sql.Integer, primary_key=True)
27     lab_id = sql.Column(sql.Integer, sql.ForeignKey('lab.id'))
28     user_id = sql.Column(sql.Integer, sql.ForeignKey('users.id'))
```

We are using SQLAlchemy as ORM and added some database classes. The first class is `DockerImage`. This class contains a name, a description and a URL. This can be used to add docker images to the lab orchestrator which can later be injected into a VMI template. This makes creating labs easy, because you only need to have the URL to your docker image. The second class is `Lab` which contains a name, a namespace prefix, a description, a reference to a docker image and a name for the docker image. The namespace prefix is used to create namespaces when launching a lab and to separate this from other labs.

That's the reason this needs to be unique. If you add a new lab you need to make sure your namespace prefix doesn't include characters that are not allowed in Kubernetes namespaces. The docker image is a reference to the first class and the idea ist that you can use a docker image for many labs if you don't need a custom image. For example you can create many labs that just use the default ubuntu image. The name of the docker image is used as VMI name and when adding new labs you need to make sure this doesn't include characters that are not allowed in Kubernetes VMI names. The third class is `LabInstance`. A lab instance is a lab that was started by a user, tho this class only references the user and the lab. [101]

Next we come to the controllers module. The controllers are used to group some services together and provide them in a central interface. They are used in the routes to access objects in the database model and the Kubernetes API. There are two types of main-controllers: `ModelController` and `KubernetesController`. The `KubernetesController` is further divided into two types: `NamespacedController` and `NotNamespacedController` so there is a total of three base classes that will be used.

Listing 56 kubernetes/controller.py ModelController

```
class ModelController(ABC):
    def _model(self) -> Type[db.Model]:
        raise NotImplementedError()

    def get_list(self):
        return self._model().query.all()

    def _create(self, *args, **kwargs) -> db.Model:
        obj = self._model()(*args, **kwargs)
        db.session.add(obj)
        db.session.commit()
        return obj

    def get(self, id) -> db.Model:
        obj = self._model().query.get(id)
        if obj is None:
            raise KeyError(f"Key error: {id}")
        return obj

    def delete(self, obj: db.Model):
        db.session.delete(obj)
        db.session.commit()

    def _serialize(self, obj):
        raise NotImplementedError()

    def make_response(self, inp: Union[db.Model, List[db.Model]]):
        if isinstance(inp, list):
            return jsonify([self._serialize(obj) for obj in inp])
        return jsonify(self._serialize(inp))
```

The `ModelController` is a controller that adds methods for database models. When extending this class you need to implement the methods `_model` and `_serialize`. `_model` needs to return the model class and `_serialize` needs to return a dictionary that can be used to serialize the objects and return them as JSON in the API. When extending this class you are automatically able to get a list of all items in the database table, you can get a specific object by its identifier and you can delete objects. There is also a create method that can be used to create new objects. The last method provided in this base class is `make_response`, which is used in the routes and returns a jsonified version of the object or a list of objects.

Listing 57 kubernetes/controller.py KubernetesController

```
46 class KubernetesController(ABC):
47     template_file = None
48 
49     def __init__(self, registry: APIRegistry):
50         self.registry = registry
51 
52     def _get_template(self, template_data):
53         template_engine = TemplateEngine(template_data)
54         return template_engine.replace(self.template_file)
55 
56     def make_response(self, inp: Union[db.Model, List[db.Model]]):
57         ret = make_response(inp)
58         ret.mimetype = 'application/json'
59         return ret
```

The `KubernetesController` class makes use of the `TemplateEngine` and the `APIRegistry` to provide methods for Kubernetes api resources. The `make_response` method turns the string from the Kubernetes API into a response and adds the `application/json` mimetype.

Listing 58 kubernetes/controller.py Namespaced- and NotNamespacedController

```
62 class NamespacedController(KubernetesController):
63     def _api(self) -> NamespacedApi:
64         raise NotImplementedError()
65 
66     def get_list(self, namespace):
67         return self._api().get_list(namespace)
68 
69     def get(self, namespace, identifier):
70         return self._api().get(namespace, identifier)
71 
72     def delete(self, namespace, identifier):
73         return self._api().delete(namespace, identifier)
74 
75 
76 class NotNamespacedController(KubernetesController):
77     def _api(self) -> NotNamespacedApi:
78         raise NotImplementedError()
79 
80     def get_list(self):
81         return self._api().get_list()
82 
83     def get(self, identifier):
84         return self._api().get(identifier)
85 
86     def delete(self, identifier):
87         return self._api().delete(identifier)
```

The `NamespacedController` and `NotNamespacedController` extend the `KubernetesController` to provide methods for the two types of Kubernetes API endpoints. This classes adds methods to get a list of all objects, get a specific object by its identifier and delete an object by its identifier. If you extend these classes you need to implement the `_api` method. This method needs to return the API extension class this controller should work onto.

Listing 59 kubernetes/controller.py NamespaceController

```
class NamespaceController(NotNamespacedController):
    template_file = 'templates/namespace_template.yaml'

    def _api(self) -> NotNamespacedApi:
        return self.registry.namespace

    def create(self, namespace):
        template_data = {'namespace': namespace}
        data = self._get_template(template_data)
        return self._api().create(data)
```

Listing 60 templates/namespace_template.yaml

```
kind: Namespace
apiVersion: v1
metadata:
  name: ${namespace}
```

The `NamespaceController` implements the `NotNamespacedController` and adds the template for namespaces and a create method.

Listing 61 kubernetes/controller.py NetworkPolicyController

```
class NetworkPolicyController(NamespacedController):
    template_file = 'templates/network_policy_template.yaml'

    def _api(self) -> NamespacedApi:
        return self.registry.network_policy

    def __init__(self, registry: APIRegistry):
        super().__init__(registry)
        self.default_name = "allow-same-namespace"

    def create(self, namespace):
        template_data = {'namespace': namespace, 'network_policy_name':
↪   self.default_name}
        data = self._get_template(template_data)
        return self._api().create(namespace, data)
```

Listing 62 templates/network_policy_template.yaml

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  namespace: ${namespace}
  name: ${network_policy_name}
spec:
  podSelector:
    matchLabels: {}
  ingress:
  - from:
    - podSelector: {}
```

The `NetworkPolicyController` implements the `NamespacedController` and adds the template for network policies and a create method. Because you only add one of these network policies to a namespace the network policy name has a default value and can't be changed.

Listing 63 kubernetes/controller.py DockerImageController

```
class DockerImageController(ModelController):
    def _model(self) -> Type[db.Model]:
        return DockerImage

    def _serialize(self, obj):
        return {'id': obj.id, 'name': obj.name, 'description':
         ↪  obj.description, 'url': obj.url}

    def create(self, name, description, url):
        return self._create(name=name, description=description, url=url)
```

The `DockerImageController` implements the `ModelController` and adds a create method.

Listing 64 kubernetes/controller.py VirtualMachineInstanceController

```
129 class VirtualMachineInstanceController(NamespacedController):
130     template_file = 'templates/vmi_template.yaml'
131 
132     def __init__(self, registry: APIRegistry, namespace_ctrl:
        ↪ NamespaceController,
133                  docker_image_ctrl: DockerImageController):
134         super().__init__(registry)
135         self.namespace_ctrl = namespace_ctrl
136         self.docker_image_ctrl = docker_image_ctrl
137 
138     def _api(self) -> NamespacedApi:
139         return self.registry.virtual_machine_instance
140 
141     def create(self, namespace, lab: Lab):
142         docker_image = self.docker_image_ctrl.get(lab.docker_image_id)
143         template_data = {"cores": 3, "memory": "3G",
144                          "vm_image": docker_image.url, "vmi_name":
                          ↪ lab.docker_image_name,
145                          "namespace": namespace}
146         data = self._get_template(template_data)
147         return self._api().create(namespace, data)
148 
149     def get_list_of_lab_instance(self, lab_instance: LabInstance):
150         namespace_name =
    ↪ LabInstanceController.get_namespace_name(lab_instance)
151         namespace = self.namespace_ctrl.get(namespace_name)
152         return self.get_list(namespace_name)
```

Listing 65 templates/vmi_template.yaml

```
metadata:
  namespace: ${namespace}
  name: ${vmi_name}
  labels:
    special: key
apiVersion: kubevirt.io/v1alpha3
kind: VirtualMachineInstance
spec:
  domain:
    cpu:
      cores: ${cores} #3
    resources:
      requests:
        memory: ${memory} #3G
    devices:
      disks:
      - name: containerdisk
        disk: {}
  volumes:
    - name: containerdisk
      containerDisk:
        image: ${vm_image}
```

The `VirtualMachineInstanceController` implements the `NamespacedController` and adds a create method and some special methods. The create method adds some preconfigured variables to the template data like amount of cores and size of the memory. For now this can't be changed, but it would be easy to make it customizable. The name of the VMI is taken from the `docker_image_name` attribute of the lab object from which the VMI will be generated. The URL from where the docker image is taken is taken from the referenced docker image of the lab. The method `get_list_of_lab_instances` is used to find all instances of a given lab instance. This is useful to find the VMIs of your currently started Lab. The method `get_of_lab_instance` does the same, but only returns the specified VMI if it is contained in the lab instance. This is used for the details page in the API.

Listing 66 kubernetes/controller.py LabController

```
160 class LabController(ModelController):
161     def _model(self) -> Type[db.Model]:
162         return Lab
163 
164     def _serialize(self, obj):
165         return {'id': obj.id, 'name': obj.name, 'namespace_prefix':
        ↪  obj.namespace_prefix,
166                 'description': obj.description, 'docker_image':
                ↪  obj.docker_image_id,
167                 'docker_image_name': obj.docker_image_name}
168 
169     def create(self, name, namespace_prefix, description, docker_image:
     ↪  DockerImage, docker_image_name) -> db.Model:
170         return self._create(name=name,
         ↪  namespace_prefix=namespace_prefix, description=description,
171                             docker_image=docker_image.id,
↪  docker_image_name=docker_image_name)
```

The `LabController` implements the `ModelController` and adds a create method.

Listing 67 kubernetes/controller.py LabInstanceController 1

```
174 class LabInstanceController(ModelController):
175     def _model(self) -> Type[db.Model]:
176         return LabInstance
177 
178     def _serialize(self, obj):
179         return {'id': obj.id, 'lab_id': obj.lab_id, 'user_id':
            ↪  obj.user_id}
180 
181     def __init__(self, virtual_machine_instance_ctrl:
        ↪  VirtualMachineInstanceController,
182                  namespace_ctrl: NamespaceController, lab_ctrl:
    ↪  LabController,
183                  network_policy_ctrl: NetworkPolicyController):
184         super().__init__()
185         self.virtual_machine_instance_ctrl =
            ↪  virtual_machine_instance_ctrl
186         self.namespace_ctrl = namespace_ctrl
187         self.lab_ctrl = lab_ctrl
188         self.network_policy_ctrl = network_policy_ctrl
189 
190     @staticmethod
191     def get_namespace_name(lab_instance: LabInstance):
192         lab = Lab.query.get(lab_instance.lab_id)
193         return LabInstanceController.gen_namespace_name(lab,
            ↪  lab_instance.user_id, lab_instance.id)
194 
195     @staticmethod
196     def gen_namespace_name(lab: Lab, user_id, lab_instance_id):
197         return f"{lab.namespace_prefix}-{user_id}-{lab_instance_id}"
```

Listing 68 kubernetes/controller.py LabInstanceController 2

```
199     def create(self, lab: Lab, user: User):
200         lab_instance = self._create(lab_id=lab.id, user_id=user.id)
201         # create namespace
202         namespace_name = LabInstanceController.gen_namespace_name(lab,
    ↪ user.id, lab_instance.id)
203         namespace = self.namespace_ctrl.create(namespace_name)
204         # create network policy
205         network_policy = self.network_policy_ctrl.create(namespace_name)
206         # create vmi
207         vmi = self.virtual_machine_instance_ctrl.create(namespace_name,
    ↪ lab)
208         return lab_instance
209 
210     def delete(self, lab_instance: LabInstance):
211         super().delete(lab_instance)
212         lab = self.lab_ctrl.get(lab_instance.lab_id)
213         namespace_name = LabInstanceController.gen_namespace_name(lab,
    ↪ lab_instance.user_id, lab_instance.id)
214         self.namespace_ctrl.delete(namespace_name)
215         # this also deletes VMIs and all other resources in the
            ↪ namespace
216 
217     def get_list_of_user(self, user: User):
218         lab_instances =
    ↪ LabInstance.query.filter_by(user_id=user.id).all()
219         return lab_instances
```

The `LabInstanceController` implements the `ModelController` and adds many methods. There are two static methods that can be used to generate the namespace name the lab is running into. Then a create method is added. The create method first creates the lab instance in the database. Then it generates the namespace name and creates the new namespace. After that the network policy and the VMI are created in this namespace. This method contains no error handling so it may not work without notifying you if you for example configured a wrong namespace prefix. After the create method the delete method is overwritten. It deletes the database object and then deletes the namespace where the VMI and network policy is running in. With deleting the namespace every resource in this namespace gets deleted too. The last method is `get_list_of_user` which returns a list of lab instances that belong to the given user.

Listing 69 kubernetes/controller.py ControllerCollection

```
@dataclass
class ControllerCollection:
    namespace_ctrl: NamespaceController
    network_policy_ctrl: NetworkPolicyController
    docker_image_ctrl: DockerImageController
    virtual_machine_instance_ctrl: VirtualMachineInstanceController
    lab_ctrl: LabController
    lab_instance_ctrl: LabInstanceController
```

The `ControllerCollection` doesn't implement any controller base class. This class is only used to have a collection with every controller. An object of this class is used in the routes to get access to the controllers.

The module `routes` contains the API routes. We have removed the old routes and added new routes. The routes are based on the Rest API design. You have the following URLs:

- `/lab_instance`: GET, POST
  - Users can see their lab instances
  - Users can create new lab instances
- `/lab_instance/<int:lab_instance_id>`: GET, DELETE
  - Users can see details to their lab instances
  - Users can delete their instance
- `/lab_instance/<int:lab_instance_id>/virtual_machine_instances`: GET
  - Users can see their VMIs
- `/lab_instance/<int:lab_instance_id>/virtual_machine_instances/<string:virtual_machine_instance_id>`: GET
  - Users can see details to their VMIs
- `/docker_image`: GET, POST
  - Everyone can see the docker images
  - Admins can create new docker images
- `/docker_image/<int:docker_image_id>`: GET, DELETE
  - Everyone can see details to the docker images
  - Admins can delete docker images
- `/lab`: GET, POST
  - Everyone can see the labs
  - Admins can add new labs
- `/lab/<int:lab_id>`: GET, DELETE
  - Everyone can see details to the lab
  - Admins can delete labs

The methods in the routes uses the services in the Controllers in the `ControllerCollection` object and adds permissions. The last method in this module is only needed to load this module. Every route is added to the app with decorators and they are only executed if this module is loaded.

The module `app` contains the flask app, the `SQLAlchemy` database object, the authentication

objects, some basic configuration and a singleton class for the `ControllerCollection` object that is used in the routes.

Listing 70 kubernetes/ws_proxy.py 1

```
13 def check_token(token, user_id):
14     user = User.verify_auth_token(token)
15     return user.id == user_id
```

Listing 71 kubernetes/ws_proxy.py 2

```
44     async def proxy(self, websocket, path):
45         '''Called whenever a new connection is made to the server'''
46         # split path to get vmi name and token
47         splitted = path.split("/")
48         if len(splitted) != 3:
49             logging.warning("Invalid URL")
50             await websocket.close(reason="invalid url")
51             return
52         lab_instance_id = splitted[1]
53         cc = CC.get()
54         lab_instance = cc.lab_instance_ctrl.get(lab_instance_id)
55         lab = cc.lab_ctrl.get(lab_instance.lab_id)
56         namespace_name =
    ↪ LabInstanceController.get_namespace_name(lab_instance)
57         vmi_name = lab.docker_image_name
58         token = splitted[2]
59         # check if user has permissions to access this vmi
60         if not check_token(token, lab_instance.user_id):
61             logging.warning("Invalid token")
62             await websocket.close(reason="invalid token")
63             return
64         # build websocket url and connect to
65         url = self.remote_url +
    ↪ self.api_path.format(namespace=namespace_name, vmi_name=vmi_name)
```

In the `ws_proxy` module we first removed the old authentication methods and replaced it with the JWT token authentication that we have added in the `user_management` module which we will explain in a moment. The `proxy` method has also some changes. The path now contains the id of the lab instance and a JWT token. The id is used to get the lab instance and the lab. This is needed to get the namespace name where the VMI is running and the VMI name. Both are needed to generate the VNC remote URL.

Last part of the refactoring is the `api` module. Here we load the config from environment variables and setup the `Proxy` and `APIRegistry` object and the Controllers. After that the `ws_proxy` and flask app are started.

Starting a lab and accessing VNC:

1. Start a lab instance: `curl -u admin:changeme -X POST -d lab_id=1 localhost:5000/lab_instance`
2. Show the VMI: `curl -u admin:changeme localhost:5000/lab_instance/1/virtual_machine_instances/`
3. Get JWT token: `curl -u admin:changeme localhost:5000/api/token`
4. Show lab instances with token: `curl -u eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6MSwiZXhwIjoxNjI4NDM1MzQ2LjUwMzcyMzZ9.BofQxA6gUFCiZfg1oSNp-296jPmg7OdLcWA0HKLrPBI:unused localhost:5000/lab_instance`
5. Open noVNC: `http://localhost:8000/vnc_lite.html?host=localhost&port=5001&path=1/eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6MSwiZXhwIjoxNjI4NDM1MzQ2LjUwMzcyMzZ9.BofQxA6gUFCiZfg1oSNp-296jPmg7OdLcWA0HKLrPBI`

KubeVirt Authorization[60]

Because Kubernetes can be accessed through an API, we can wrap all methods in an application and add authorization in a different layer. This will be shown in the prototype.

Hier wird nach einer Lösung gesucht, womit es möglich ist mehrere user zu haben und dass ein user nur auf sein lab zugreifen kann.

Hier werden die vnc und ttyd dienste, welche wir nutzen über ingresses oder services nach außen erreichbar gemacht. Für vnc werden wir vermutlich einen proxy bauen müssen oder ähnliches um die k8 api zu wrappen.

[60] https://kubevirt.io/user-guide/operations/authorization/

List of Figures

List of Listings

6 Bibliography

1. Signale | ubuntuusers. Retrieved June 1, 2021 from https://wiki.ubuntuusers.de/Signale/
2. Nohup | ubuntuusers. Retrieved June 1, 2021 from https://wiki.ubuntuusers.de/nohup/
3. Wissenschaftliche texte schreiben mit markdown und pandoc. Retrieved June 1, 2021 from https://vijual.de/2019/03/11/artikel-mit-markdown-und-pandoc-schreiben/
4. Replacing placeholders with their metadata value. Retrieved June 1, 2021 from https://pandoc.org/lua-filters.html#replacing-placeholders-with-their-metadata-value
5. What is kubernetes? Retrieved June 2, 2021 from https://www.redhat.com/en/topics/containers/what-is-kubernetes
6. Introduction to kubernetes architecture. Retrieved June 2, 2021 from https://www.redhat.com/en/topics/containers/kubernetes-architecture
7. What is a kubernetes deployment? Retrieved June 2, 2021 from https://www.redhat.com/en/topics/containers/what-is-kubernetes-deployment
8. What is a kubernetes operator? Retrieved June 2, 2021 from https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-operator
9. Service | kubernetes. Retrieved June 3, 2021 from https://kubernetes.io/docs/concepts/services-networking/service/
10. Ingress | kubernetes. Retrieved June 3, 2021 from https://kubernetes.io/docs/concepts/services-networking/ingress/
11. Network policies | kubernetes. Retrieved June 3, 2021 from https://kubernetes.io/docs/concepts/services-networking/network-policies/
12. Understanding kubernetes objects | kubernetes. Retrieved June 3, 2021 from https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/
13. Install tools | kubernetes. Retrieved June 3, 2021 from https://kubernetes.io/docs/tasks/tools/
14. Configuration best practices | kubernetes. Retrieved June 3, 2021 from https://kubernetes.io/docs/concepts/configuration/overview/
15. Secrets | kubernetes. Retrieved June 3, 2021 from https://kubernetes.io/docs/concepts/configuration/secret/
16. ConfigMaps | kubernetes. Retrieved June 3, 2021 from https://kubernetes.io/docs/concepts/configuration/configmap/
17. How to use KubeVirt with minikube. Retrieved June 4, 2021 from https://minikube.sigs.k8s.io/docs/tutorials/kubevirt/
18. Experiment with CDI | KubeVirt. Retrieved June 4, 2021 from https://kubevirt.io/labs/kubernetes/lab2.html
19. Use KubeVirt | KubeVirt. Retrieved June 5, 2021 from https://kubevirt.io/labs/kubernetes/lab1.html
20. Architecture | KubeVirt. Retrieved June 5, 2021 from https://kubevirt.io/user-guide/architecture/
21. Lifecycle | KubeVirt. Retrieved June 5, 2021 from https://kubevirt.io/user-guide/virtual_machines/lifecycle/
22. Run strategies | KubeVirt. Retrieved June 5, 2021 from https://kubevirt.io/user-guide/virtual_machines/run_strategies/
23. Presets | KubeVirt. Retrieved June 6, 2021 from https://kubevirt.io/user-guide/virtual_machines/presets/
24. Top level API objects | KubeVirt. Retrieved June 7, 2021 from https://kubevirt.io/api-reference/v0.6.4/definitions.html
25. Disks and volumes | KubeVirt. Retrieved June 7, 2021 from https://kubevirt.io/user-guide/virtual_machines/disks_and_volumes/

26. Volumes | kubernetes. Retrieved June 10, 2021 from https://kubernetes.io/docs/concepts/storage/volumes/
27. Persistent volumes | kubernetes. Retrieved June 10, 2021 from https://kubernetes.io/docs/concepts/storage/persistent-volumes/#persistentvolumeclaims
28. Storage classes | kubernetes. Retrieved June 10, 2021 from https://kubernetes.io/docs/concepts/storage/storage-classes/#local
29. Multus. Retrieved June 10, 2021 from https://github.com/k8snetworkplumbingwg/multus-cni
30. Interfaces and networks | KubeVirt. Retrieved June 10, 2021 from https://kubevirt.io/user-guide/virtual_machines/interfaces_and_networks/
31. Boot stages | cloud-init. Retrieved June 14, 2021 from https://cloudinit.readthedocs.io/en/latest/topics/boot.html
32. Availability | cloud-init. Retrieved June 14, 2021 from https://cloudinit.readthedocs.io/en/latest/topics/availability.html
33. Cloud-init documentation | cloud-init. Retrieved June 14, 2021 from https://cloudinit.readthedocs.io/en/latest/index.html
34. Datasources | cloud-init. Retrieved June 14, 2021 from https://cloudinit.readthedocs.io/en/latest/topics/datasources.html
35. NoCloud | cloud-init. Retrieved June 14, 2021 from https://cloudinit.readthedocs.io/en/latest/topics/datasources/nocloud.html
36. Getting to know kubevirt | kubernetes. Retrieved June 14, 2021 from https://kubernetes.io/blog/2018/05/22/getting-to-know-kubevirt/
37. NetworkPolicy | KubeVirt. Retrieved June 14, 2021 from https://kubevirt.io/user-guide/virtual_machines/networkpolicy/
38. Network policies | kubernetes. Retrieved June 14, 2021 from https://kubernetes.io/docs/concepts/services-networking/network-policies/
39. VirtualMachineInstanceReplicaSet | KubeVirt. Retrieved July 3, 2021 from https://kubevirt.io/user-guide/virtual_machines/replicaset/
40. Service objects | KubeVirt. Retrieved July 3, 2021 from https://kubevirt.io/user-guide/virtual_machines/service_objects/
41. Where does boxes store disk images? | gnome help. Retrieved July 4, 2021 from https://help.gnome.org/users/gnome-boxes/stable/disk-images.html.en
42. How to customize Qcow2/raw linux OS disk image with virt-customize | computing for geeks. Retrieved July 4, 2021 from https://computingforgeeks.com/customize-qcow2-raw-image-templates-with-virt-customize/
43. How to extend/increase KVM virtual machine (VM) disk size | computing for geeks. Retrieved July 4, 2021 from https://computingforgeeks.com/how-to-extend-increase-kvm-virtual-machine-disk-size/
44. Use ubuntu cloud image with KVM | medium. Retrieved July 4, 2021 from https://medium.com/@art.vasilyev/use-ubuntu-cloud-image-with-kvm-1f28c19f82f8
45. How do you increase a KVM guest's disk space? | server fault. Retrieved July 4, 2021 from https://serverfault.com/questions/324281/how-do-you-increase-a-kvm-guests-disk-space
46. Resizing a filesystem using qemu-img and fdisk | github gist. Retrieved July 4, 2021 from https://gist.github.com/larsks/3933980
47. Use a service to access an application in a cluster | kubernetes. Retrieved July 5, 2021 from https://kubernetes.io/docs/tasks/access-application-cluster/service-access-application-cluster/
48. Ttyd - share your terminal over the web | kubernetes. Retrieved July 5, 2021 from https://github.com/tsl0922/ttyd
49. Ttyd - share your terminal over the web | kubernetes. Retrieved July 5, 2021 from https://github.com/tsl0922/ttyd

50. Connect to QEMU/KVM via noVNC and websockify | programmer sought. Retrieved July 6, 2021 from https://www.programmersought.com/article/19755825712/
51. Accessing virtual machines. Retrieved July 6, 2021 from https://kubevirt.io/user-guide/virtual_machines/accessing_virtual_machines/
52. How to create desktop cloud image of ubuntu on OpenStack environment? | github gist. Retrieved July 6, 2021 from https://gist.github.com/sasukeh/0a50ca570722df3a4a65
53. How to install cloud-init package on ubuntu. Retrieved July 6, 2021 from https://zoomadmin.com/HowToInstall/UbuntuPackage/cloud-init
54. Minikube increase disk size. Retrieved July 6, 2021 from https://www.maxoberberger.net/blog/2018/09/minikube-increase-disk-size.html
55. I getting always“ no space left on device” issue in minikube even system have free space , how to resolve? | stackoverflow. Retrieved July 6, 2021 from https://stackoverflow.com/questions/62998012/i-getting-always-no-space-left-on-device-issue-in-minikube-even-system-have-fr
56. Build your kubernetes armory with minikube, kail, and kubens. Retrieved July 6, 2021 from https://developers.redhat.com/blog/2019/04/16/build-your-kubernetes-armory-with-minikube-kail-and-kubens#__minikube
57. Build context for docker image very large | stackoverflow. Retrieved July 12, 2021 from https://stackoverflow.com/a/26604307
58. Best practices for writing dockerfiles | docker docs. Retrieved July 12, 2021 from https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#understand-build-context
59. Access virtual machines' graphic console using noVNC. Retrieved July 14, 2021 from https://kubevirt.io/2019/Access-Virtual-Machines-graphic-console-using-noVNC.html
60. How to quickly serve files and folders over HTTP in linux. Retrieved July 14, 2021 from https://ostechnix.com/how-to-quickly-serve-files-and-folders-over-http-in-linux/
61. Embedding and deploying noVNC application | github. Retrieved July 14, 2021 from https://github.com/novnc/noVNC/blob/master/docs/EMBEDDING.md
62. The kubernetes API | kubernetes. Retrieved July 14, 2021 from https://kubernetes.io/docs/concepts/overview/kubernetes-api/
63. Test a WebSocket using curl. | github gist. Retrieved July 14, 2021 from https://gist.github.com/htp/fbce19069187ec1cc486b594104f01d0
64. Namespaces walkthrough | kubernetes. Retrieved July 15, 2021 from https://kubernetes.io/docs/tasks/administer-cluster/namespaces-walkthrough/
65. Networking und network policy | kubernetes. Retrieved July 15, 2021 from https://kubernetes.io/de/docs/concepts/cluster-administration/addons/
66. Alternative containerlaufzeitumgebungen | kubernetes. Retrieved July 15, 2021 from https://kubernetes.io/de/docs/setup/minikube/#alternative-containerlaufzeitumgebungen
67. Network plugins | kubernetes. Retrieved July 15, 2021 from https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/
68. Persistent volumes | minikube. Retrieved July 26, 2021 from https://minikube.sigs.k8s.io/docs/handbook/persistent_volumes/
69. Python | dockerhub. Retrieved July 31, 2021 from https://hub.docker.com/_/python
70. Deployments | kubernetes. Retrieved July 31, 2021 from https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
71. Add self signed certificate to ubuntu for use with curl | stackoverflow. Retrieved July 31, 2021 from https://stackoverflow.com/questions/5109661/add-self-signed-certificate-to-ubuntu-for-use-with-curl
72. Accessing the kubernetes API from a pod | kubernetes. Retrieved July 31, 2021 from https://kubernetes.io/docs/tasks/run-application/access-api-from-pod/

73. Get a shell to a running container | kubernetes. Retrieved July 31, 2021 from https://kubernetes.io/docs/tasks/debug-application-cluster/get-shell-running-container/
74. Accessing clusters | kubernetes. Retrieved July 31, 2021 from https://kubernetes.io/docs/tasks/access-application-cluster/access-cluster/
75. Configure service accounts for pods | kubernetes. Retrieved July 31, 2021 from https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/
76. Using RBAC authorization | kubernetes. Retrieved July 31, 2021 from https://kubernetes.io/docs/reference/access-authn-authz/rbac/#rolebinding-and-clusterrolebinding
77. Authenticating | kubernetes. Retrieved July 31, 2021 from https://kubernetes.io/docs/reference/access-authn-authz/authentication/
78. Interaktion mit laufenden pods | kubernetes. Retrieved August 1, 2021 from https://kubernetes.io/de/docs/reference/kubectl/cheatsheet/#interaktion-mit-laufenden-pods
79. Forcing application/json MIME type in a view (flask) | stackoverflow. Retrieved August 1, 2021 from https://stackoverflow.com/questions/11945523/forcing-application-json-mime-type-in-a-view-flask
80. Python flask, how to set content type | stackoverflow. Retrieved August 1, 2021 from https://stackoverflow.com/questions/11773348/python-flask-how-to-set-content-type
81. Advanced usage | python requests. Retrieved August 1, 2021 from https://docs.python-requests.org/en/master/user/advanced/
82. How do i disable the security certificate check in python requests | stackoverflow. Retrieved August 1, 2021 from https://stackoverflow.com/questions/15445981/how-do-i-disable-the-security-certificate-check-in-python-requests
83. Python requests: .pem -> .crt + key | stackoverflow. Retrieved August 1, 2021 from https://stackoverflow.com/questions/23705770/python-requests-pem-crt-key
84. Convert with curl to python requests with multiple ssl params | stackoverflow. Retrieved August 1, 2021 from https://stackoverflow.com/questions/51747092/convert-with-curl-to-python-requests-with-multiple-ssl-params
85. API OVERVIEW | kubernetes. Retrieved August 1, 2021 from https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.19/
86. Using cURL to upload POST data with files | stackoverflow. Retrieved August 1, 2021 from https://stackoverflow.com/questions/12667797/using-curl-to-upload-post-data-with-files
87. How to replace environment variable value in yaml file to be parsed using python script. Retrieved August 2, 2021 from https://stackoverflow.com/questions/52412297/how-to-replace-environment-variable-value-in-yaml-file-to-be-parsed-using-python
88. PyYAML documentation. Retrieved August 2, 2021 from https://pyyaml.org/wiki/PyYAMLDocumentation
89. Pyyaml - using different styles for keys and integers and strings | stackoverflow. Retrieved August 2, 2021 from https://stackoverflow.com/questions/30198481/pyyaml-using-different-styles-for-keys-and-integers-and-strings
90. How can i get the named parameters from a URL using flask? | stackoverflow. Retrieved August 2, 2021 from https://stackoverflow.com/questions/24892035/how-can-i-get-the-named-parameters-from-a-url-using-flask
91. Getting started | websockets. Retrieved August 4, 2021 from https://websockets.readthedocs.io/en/stable/intro.html
92. Simple websocket proxy written in python | github gist. Retrieved August 4, 2021 from https://gist.github.com/bsergean/bad452fa543ec7df6b7fd496696b2cd8
93. Python: Concurrently pending on async coroutine and synchronous function | stackoverflow. Retrieved August 4, 2021 from https://stackoverflow.com/questions/62373522/python-concurrently-pending-on-async-coroutine-and-synchronous-function

94. Python asyncio: Event loop does not seem to stop when stop method is called | stackoverflow. Retrieved August 4, 2021 from https://stackoverflow.com/questions/46093238/python-asyncio-event-loop-does-not-seem-to-stop-when-stop-method-is-called/46097639

95. Event loop | python docs. Retrieved August 4, 2021 from https://docs.python.org/3/library/asyncio-eventloop.html

96. Can an asyncio event loop run in the background without suspending the python interpreter? | stackoverflow. Retrieved August 4, 2021 from https://stackoverflow.com/questions/26270681/can-an-asyncio-event-loop-run-in-the-background-without-suspending-the-python-in

97. Start a flask application in separate thread | stackoverflow. Retrieved August 4, 2021 from https://stackoverflow.com/questions/31264826/start-a-flask-application-in-separate-thread

98. WebSocket JWT token connection authorization | stackoverflow. Retrieved August 4, 2021 from https://stackoverflow.com/questions/28129139/websocket-jwt-token-connection-authorization

99. WebSockets | readthedocs. Retrieved August 4, 2021 from https://websockets.readthedocs.io/en/2.1/

100. Kubernetes API concepts | kubernetes. Retrieved August 8, 2021 from https://kubernetes.io/docs/reference/using-api/api-concepts/

101. Basic relationship patterns | SQLAlchemy. Retrieved August 8, 2021 from https://docs.sqlalchemy.org/en/14/orm/basic_relationships.html