Ibm Uvt Proiect Colectiv Devops

IBM UVT Proiect Colectiv DevOps

This repository holds course materials for "Podman" course, part of `DevOps UVT Proiect Colectiv`.

Course Structure

- **1 Introduction to containers-and-podman** (theoretical section)
  - What are containers?
  - Containers vs Virtual Machines
  - Podman
  - Dockerfile (definition) to Image (build) to Containers (running)
  - Writing Containerfiles (Dockerfiles)
  - Container Registries
  - Further Reading Materials
- **2 Hands on exercises** (practical section)
  - Install Podman (Rokcylinux)
  - Create an account on Dockerhub.com
  - Pull an image from a public container registry
  - Make changes to an images and save it
  - Build and tag an image
  - Push an image to a public container registry
- **3 Container orchestration** (OPTIONAL: If we have time left)
  - What/Why?
  - Kubernetes and Openshift

1 Introduction To Containers And Podman

Containers

What are containers?

Software applications typically depend on other libraries, configuration files, or services that are provided by the runtime environment.

Developers typically only share code or binaries (instead of sharing an entire virtual machine). Sharing an entire virtual machine with other developers can be inconvenient because of a virtual machine's size and format. Sharing only the code is advantageous in terms of size - the disadvantage being that extra steps are required when deploying the software into a production environment. Developers often build on code that requires underlying modules (e.g. Ruby, PHP PECL, Python, Perl, etc). Some of these modules are provided as packages in the operating system, but often they are pulled from an external repository -

some even need to be compiled from C/C++. Other developers, administrators, and architects are then left to figure out how to install these requirements. Tools like RPM, git, and RVM have been developed to make this easier... but the build and deployment steps can be complicated and often require some measure of expertise (outside the domain of the application being developed).

Neither the "code only" nor "full virtual machine" method of sharing is perfect.

One of the bigger pain points that has traditionally existed between development and operations teams is how to make changes rapidly enough to support effective development but without risking the stability of the production environment and infrastructure.

Containers make it easier for developers to develop an application in a container in their local environment and deploy that same container into production, minimizing risk and development overhead while also cutting down on the amount of deployment effort required of operations engineers.

Containers provide many of the same benefits as virtual machines, such as *security*, *storage*, and *network isolation*. Containers *require far fewer hardware resources and are quick to start and terminate*. They also *isolate the libraries and the runtime resources (such as CPU and storage) for an application* to minimize the impact of any OS update to the host OS.

Containers ares a set of one or more processes that are isolated from the rest of the system.

Containers help DevOps by enabling consistent and repeatable environments across development, testing, and production, which aligns development and operations teams by reducing configuration discrepancies and ensuring the application behaves the same way across all stages. This streamlines the CI/CD pipeline, accelerates deployment, and minimizes the risk of errors during transitions from development to production.

Containers vs Virtual Machines

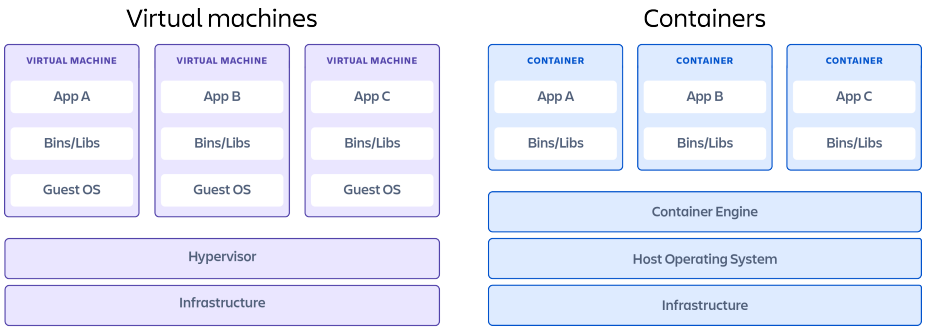

Figure 1: Containers vs VM illustration

In a **virtual machine (VM)**, we need to install an operating system with the appropriate device drivers; hence,the footprint or size of a virtual machine is huge. A normal VM with Tomcat and Java installed may take up to 10 GB of drive space: There's an overhead of memory management and device drivers. A VM has all the components a normal physical machine has in terms of operation.

In a VM, the hypervisor abstracts resources. Its package includes not only the application, but also the necessary binaries and libraries, and an entire guest operating system, for example, CentOS 6.7 and Windows 2003. Cloud service providers use a hypervisor to provide a standard runtime.

A **container** shares the operating system and device drivers of the host. Containers are created from images, and for a container with Tomcat installed, the size is less than 500 MB: Containers are small in size and hence effectively give faster and better performance. They abstract the operating system. A container runs as an isolated user space, with processes and filesystems in the user space on the host operating system itself, and it shares the kernel with other containers. Sharing and resource utilization are at their best in containers, and more resources are available due to less overhead. It works with very few required resources.

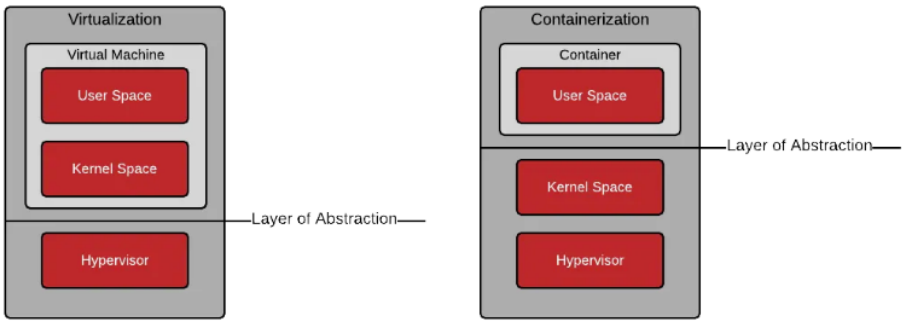

Figure 2: Virtualization vs Containerization

`Linux kernel` is the core part of the Linux operating system. It's what originally Linus wrote. `Linux OS` is a combination of the kernel and a user-land (libraries, GNU utilities, config files, etc). `Linux distribution` is a particular version of the Linux operating system like Debian or CentOS.

`User space` refers to all of the code in an operating system that lives outside of the kernel. Most Unix-like operating systems (including Linux) come pre-packaged with all kinds of utilities, programming languages, and graphical tools - these are user space applications. We often refer to this as "userland."

`Kernel space` is where the core of the operating system, the kernel, operates. The kernel is responsible for managing the system's resources, such as the CPU,

memory, and storage. It also provides system calls, which are interfaces that allow userspace applications to interact with the kernel.

A container is indeed just a `process` (or a bunch of processes) running on the Linux host. The container process `is isolated` (namespaces) from the rest of the system and `restricted from both the resource consumption` (cgroups) and security (capabilities, AppArmor, Seccomp) standpoints.

For those interested in the internals of the containers this blog with 3 articles from Scott McCarty is an extremely good read.

Podman

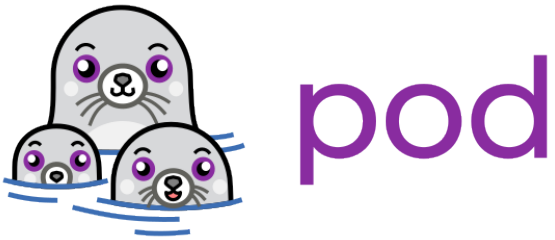

Figure 3: Podman logo

"Podman is a daemonless, open source, Linux native tool designed to make it easy to find, run, build, share and deploy applications using Open Containers Initiative (OCI) Containers and Container Images. Podman provides a command line interface (CLI) familiar to anyone who has used the Docker Container Engine. Most users can simply alias Docker to Podman (alias docker=podman) without any problems. Similar to other common Container Engines (Docker, CRI-O, containerd), Podman relies on an OCI compliant Container Runtime (runc, crun, runv, etc) to interface with the operating system and create the running containers. This makes the running containers created by Podman nearly indistinguishable from those created by any other common container engine." - Podman docs

Podman does bring some nicer features compared to Docker:

- Docker relies on a central daemon that manages containers, but Podman **doesn't require a background service running**
- **Rootless Containers**: Podman allows users to run containers as non-root users, enhancing security.
- **Docker Compatibility**: Podman aims to be a drop-in replacement for Docker, meaning it can often use the same commands and workflows.
- **Systemd integration**. Podman can generate systemd unit files that allow users to run containers as system services.

- Podman **can generate Kubernetes YAML files from existing container configurations**, which makes it easy to use Podman containers as part of a Kubernetes-based orchestration setup.

Bellow are some of the basic Podman commands a begginer should know:

```
# Verifiy if Podman is installed and check the version
podman version

# Build an image using instructions from Containerfiles
podman build

# Run a command in a new container
podman run

# Save image(s) to an archive
podman save

# Load image(s) from a tar archive
podman load

# List images in local storage
podman images

# List containers
podman ps

# Run a process in a running container
podman exec

# Log in to a container registry
podman login
```

The `Open Container Initiative (OCI)` is a Linux Foundation project dedicated to managing specifications and projects related to the storage, distribution, and execution of container images. The OCI was formed in 2015 when developers recognized that the quickly growing container industry needed standards to ensure the portability of containers across systems and platforms.

The OCI currently manages three specifications: the `Runtime Specification`, the `Image Specification`, and the `Distribution Specification`. These specifications work together to ensure that any OCI-compliant image can be run on any OCI-compliant runtime, and that OCI-compliant registries (such as Docker, Amazon Elastic Container Registry, or Google Container Registry) are able to distribute OCI images according to OCI guidelines.

Containerfile (definition) to Image (build) to Containers (running)

A `Containerfile (Dockerfile)` is the Docker image's source code. A Containerfile (Dockerfile) is a text file containing various instructions and configurations. The `FROM` command in a Containerfile (Dockerfile) identifies the base image from which you are constructing. An `image` is an immutable file that contains the source code, libraries, dependencies, tools, and other files needed for an application to run. In a way images are templates used to build a container. A `container` is, ultimately, just a running image.

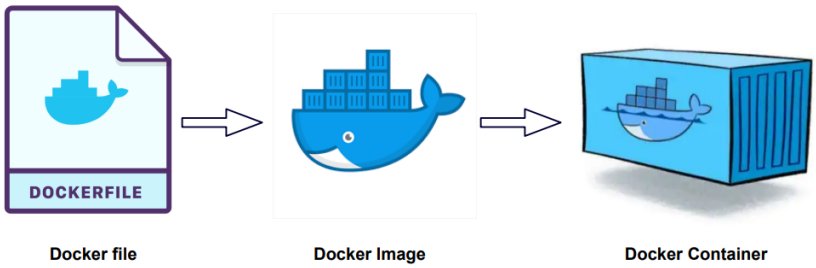

Figure 4: Dockerfile vs image vs container illustration

Writing Containerfiles (Dockerfiles)

A Containerfile (Dockerfile) is a text-based document that's used to create a container image. It provides instructions to the image builder on the commands to run, files to copy, startup command, and more.

Example:

```
FROM python:3.12

WORKDIR /usr/local/app

# Install the application dependencies
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

# Copy in the source code
COPY src ./src
EXPOSE 5000

# Setup an app user so the container doesn't run as the root user
RUN useradd app
```

```
USER app

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080"]
```

For a complete guide check Dockerfile Refference

Container Registries

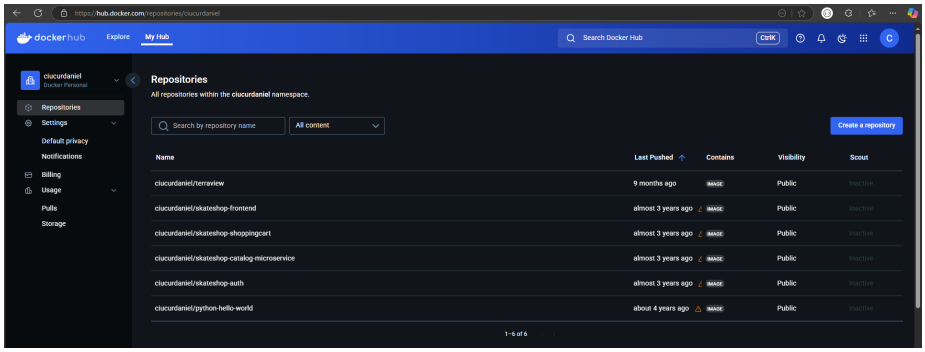

Figure 5: Screenshot from DockerHub

An **image registry** is a centralized location for **storing and sharing** your container images. It can be either **public** or **private**. Docker Hub is a public registry that anyone can use and is the default registry.

While Docker Hub is a popular option, there are many other available container registries available today, including Amazon Elastic Container Registry(ECR), Azure Container Registry (ACR), and Google Container Registry (GCR). You can even run your private registry on your local system or inside your organization. For example, Harbor, JFrog Artifactory, GitLab Container registry etc.

To address a registry artifact for push and pull operations with Docker or other client tools, combine the fully qualified registry name, repository name (including namespace path if applicable), and an artifact tag or manifest digest. See previous sections for explanations of these terms.

Address by tag: `[loginServerUrl]/[repository][:tag]`

Address by digest: `[loginServerUrl]/[repository@sha256][:digest]`

We can also have namespaces within the image name. Those are optional sometimes may represent the user or organization. For example, in Dockerhub official images will be under `library` namespace such as `docker.io/library/ubuntu:latest`. If you are pushing images to docker hub then you will need to add your username as a namespace: `docker.io/johndoe/ubuntu:latest`.

Reading: docker image tag (same applies to podman)

Further Reading Materials

- IBM Introduction to containerization
- Best practices for building containers
- A Practical Introduction to Docker Container Terminology
- Write your first Containerfile for Podman
- Base Images
- Developing inside a container via VsCode

2 Hands On Exercises

Hands on exercises

Install Podman (Rokcylinux)

Podman should come pre-installed in Rockylinux. If that is not the case, install it using:

```
dnf install podman
```

Verify installation:

```
podman version
```

> Hint: Rockylinux - Podman Guide Rockylinux - Podman Guide - 2

Create an account on Dockerhub.com

Go to https://hub.docker.com and sign up for an account.

Remember the credentials! We will need them to push images to DockerHub

Pull an image from a public container registry

```
# Pull an ubuntu image
podman pull docker.io/ubuntu:22.04

# list local images
podman image list

# Run your first container
podman run --rm ubuntu:22.04 cat /etc/os-release

# Run container interactively
podman run -it --name my-container ubuntu:22.04

# (inside container) install git
apt-get update && apt-get install -y git
```

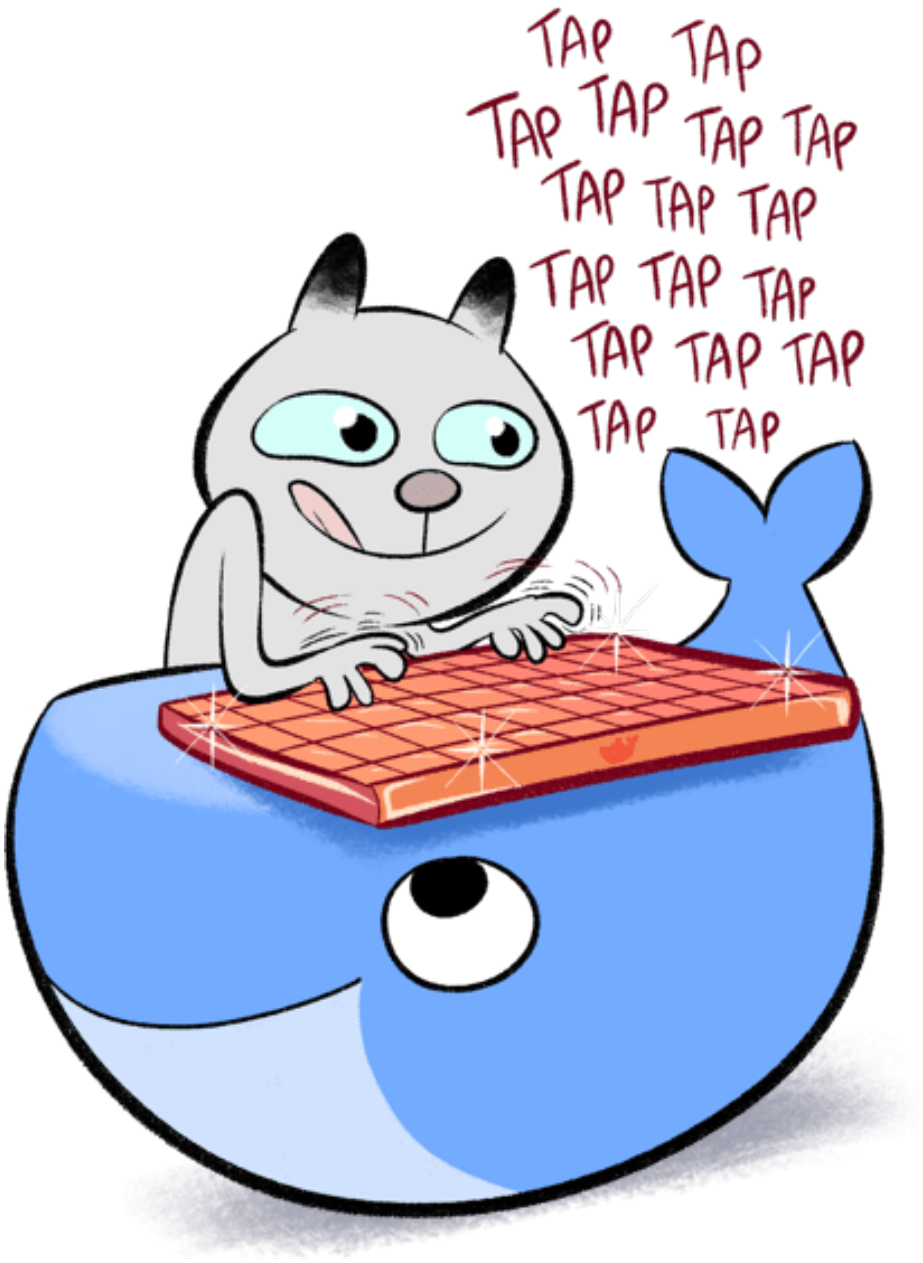

Figure 6: Cat with keyboard

HINT: Always use `--help` to know various flags you can use. Like –rm flag on podman run which "Automatically removes the container and its associated anonymous volumes when it exits"

Make changes to an images and save it

```
# Run container interactively
# -i flag -> Keeps STDIN (standard input) open, even if not attached
# -t flag -> Allocates a pseudo-TTY (a terminal). Makes the container look and feel like a t
podman run -it --name my-container ubuntu:22.04

# From inside ubuntu container install git
apt-get update && apt-get install -y git

# Commit your changes in a new container
podman commit my-container my-ubuntu-with-git

# Now enter you container and check is git is is installed
podman run -it my-ubuntu-with-git

git version
```

HINT: You will almost never do this. You always create a `Containerfile` such that all dependencies can be tracked effectively.

Build and tag an image

Create a directory and save the code bellow in a file called `main.go`.

```
cd
mkdir app && cd app
vi main.go # paste snipped from bellow
```

```
package main

import (
    "fmt"
    "net/http"
)

func helloHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello, world!")
}

func main() {
    http.HandleFunc("/hello", helloHandler)
    fmt.Println("Server is running on http://localhost:8080")
```

```
    http.ListenAndServe(":8080", nil)
}
```

Next create a Containerfile

```
vi Containerfile
```

```
FROM docker.io/golang:1.20

WORKDIR /app

COPY . .

RUN go mod init myapp || echo "go.mod already exists"

RUN go mod tidy

RUN go build -o hello-app .

EXPOSE 8080

CMD ["./hello-app"]
# ENTRYPOINT ["./hello-app"]
```

```
podman build -t go-hello-app .

podman run -d -p 8080:8080 go-hello-app

curl http://localhost:8080 # you need to install curl on rockylinux via: dnf install curl

podman stop <generated_name_for_container>
```

> HINT: Use `podman build --help` to check for important flags. In this case we are interested to gie the image a name and tag which is done via `-t, --tag Name and optionally a tag (format: "name:tag")` flag.
>
> HINT: Port forwarding in Docker/Podman allows you to map a port on your local machine to a port inside the container, enabling you to access the container's services from outside.
>
> HINT: The `-d` flag ensures that the container runs in detached mode, so your terminal remains free for other commands.

Push an image to a public container registry

```
podman login docker.io

podman image ls
```

```
podman tag go-hello-app docker.io/ciucurdaniel/go-hello-app:latest
```

```
podman push docker.io/ciucurdaniel/go-hello-app:latest
```

Go to container registry and check your image will now exist within a repository.

Example: https://hub.docker.com/r/ciucurdaniel/go-hello-app

Multi-stage container builds

First we create a seprate folder where we copy the golang code and the Dockerfile

```
cd
ls # should show the app/ directory with main.go and Dockerfile inside
mkdir multi-stage-app
cp app/* /multi-stage-app
```

```
# This will be stage 1 the build stage
FROM golang:1.20 as builder

WORKDIR /app

COPY . .

# Tell golang to produce a static binary, this is more relevant is we do multi-stage contai
ENV CGO_ENABLED=0 GOOS=linux

RUN go mod init myapp || echo "go.mod already exists"

RUN go mod tidy

RUN go build -o hello-app .

# This will be the second stage where we have a very minimal base image and we just add our
FROM scratch

COPY --from=builder /app/hello-app /app/hello-app

EXPOSE 8080

CMD ["/app/hello-app"]
# ENTRYPOINT ["./hello-app"]
```

```
# Build the image
podman build -t test-multi-stage .

# Run the image just to confirm it works correctly
```

```
podman run -d -p 8080:8080 test-multi-stage
```

```
curl http://localhost:8080
```

Compare the image sizes

```
podman images
```

The image build in a single stage which still contains the GO SDK will have around `943 MB` while the image build using multiple stages will have only about `6.24 MB`.

Cleanup commands

If you want to remove all containers and images run the commands:

```
podman system prune
podman container rm --all
podman image rm --all
```

3 Container Orchestration

Container orchestration

What/Why?

Today an organization might have hundreds or thousands of containers. An amount that would be nearly impossible for teams to manage manually.

Problem : as the number of containers managed by an organization grows, the work of manually starting them rises exponentially along with the need to quickly respond to external demands.

Enterprise needs:

- Easy communication between a large number of services
- Resources limits on applications regardless of the number fo containers running them
- To respond to application usage spikes to increase or decrease running containers
- Reacs to service deterioration with health checks
- Gradual roll out fo a new release to a set of users

This is where **container orchestration** comes in.

Kubernetes and Openshift

Kubernetes is an orchestration service that simplifies the deployment, management, and scaling of containerized applications.

Kubernetes is an orchestration service that simplifies the deployment, management, and scaling of containerized applications, the smallest unit if kunernetes is a pod that consist of one or more containers.

Kubernetes features of top of a container infra:

Service discovery and loading balancing : communication by a single DNS entry to each set of container, permits the load balancing across the pool of container. **Horizontal scaling** : Applications can scale up and down manually or automatically **Self-Healing**: user-defined health checks to monitor containers to restart in case of failure **Automated rollout and rollback** : roll updates out to application containers, if something goes wrong kubernetes can rollback to previous integration of the deployment **Secrets and configuration management** : can manage the config settings of application without rebuilding container **Operators** : use API to update the cluster state reacting to change in the app state

Red Hat OpenShift Container Plataform (RHOCP) is a set of modular components and services build on top of Kubernetes, adds the capabilities to provide PaaS platform.

OpenShift features to kubernetes cluster :

Integrated developer workflow : integrates a build in container registry, CI/CD pipeline and S2I, a tool to build artifacts from source repositories to container image *Routes* : expose service to the outside world **Metrics and logging** : Metric service and aggregated logging **Unified UI** : UI to manage the different capabilities

Kubernetes Arhitecture

A Kubernetes cluster consists of a control plane plus a set of worker machines, called nodes, that run containerized applications. Every cluster needs at least one worker node in order to run Pods.

Kubernetes objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster.

Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.

A Pod (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers.

There are many other object such as:

- **Deployments** – Manage stateless applications and ensure the desired number of Pods are running.
- **ReplicaSets** – Ensure a specified number of identical Pods are maintained.

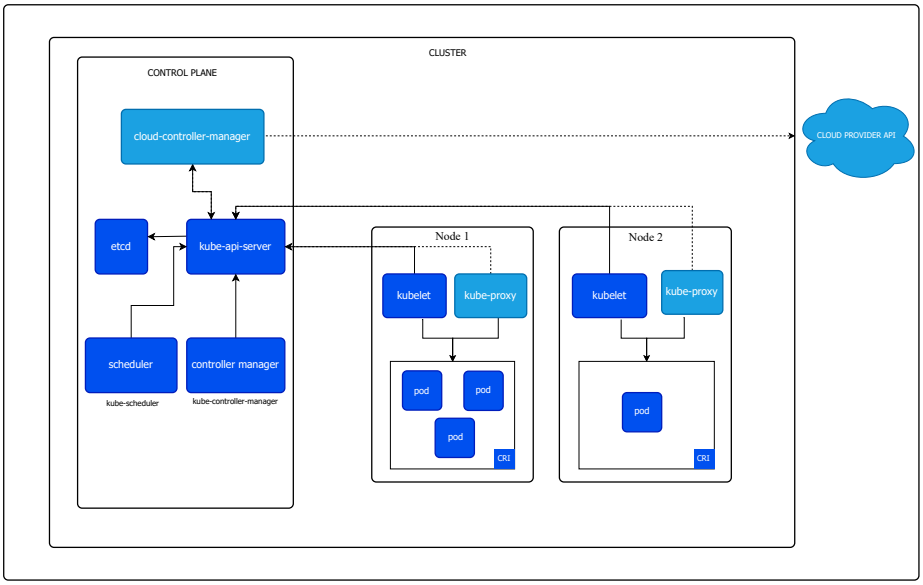

Figure 7: Kubernetes cluster arhitecture

- **StatefulSets** – Manage stateful applications that require persistent identity and storage.
- **DaemonSets** – Ensure a copy of a Pod runs on all (or some) nodes in the cluster.
- **Jobs** – Create Pods to run a task to completion.
- **CronJobs** – Run Jobs on a scheduled time (like cron).
- **Services** – Provide stable networking and load balancing for Pods.
- **Ingress** – Manage external access to Services, usually HTTP.
- **ConfigMaps** – Provide configuration data in key-value pairs.
- **Secrets** – Store sensitive data such as passwords or tokens.
- **Volumes** – Provide persistent or shared storage to Pods.
- **Namespaces** – Support multiple virtual clusters within the same physical cluster.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: go-hello-app
  labels:
    app: go-hello-app
spec:
  replicas: 2
  strategy:
    type: RollingUpdate
```

```
    rollingUpdate:
      maxSurge: 1           # Allows 1 extra pod during update
      maxUnavailable: 0     # Ensures no pods are taken down before new ones are ready
  selector:
    matchLabels:
      app: go-hello-app
  template:
    metadata:
      labels:
        app: go-hello-app
    spec:
      containers:
        - name: go-hello-app
          image: docker.io/ciucurdaniel/go-hello-app:latest
          ports:
            - containerPort: 8080
          livenessProbe:
            httpGet:
              path: /healthz
              port: 8080
            initialDelaySeconds: 10
            periodSeconds: 10
            timeoutSeconds: 2
            failureThreshold: 3
          readinessProbe:
            httpGet:
              path: /healthz
              port: 8080
            initialDelaySeconds: 5
            periodSeconds: 5
            timeoutSeconds: 2
            failureThreshold: 3

apiVersion: v1
kind: Service
metadata:
  name: go-hello-app
  labels:
    app: go-hello-app
spec:
  selector:
    app: go-hello-app
  ports:
    - name: http
      port: 80
      targetPort: 8080
```

```
  type: ClusterIP

apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: go-hello-app
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: go-hello-app
  minReplicas: 2
  maxReplicas: 5
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 70
```

How to get hands-on experience with Kubernetes

For those interested to get hands-on experience with Kubernetes without having to spend money on a cloud Kubernetes instance, there are various alterntives that can be used to simulate a Kubernetes cluster on your machine.

I personally recommend Minikube.

`minikube` is local Kubernetes, focusing on making it easy to learn and develop for Kubernetes.

All you need is Docker (or similarly compatible) container or a Virtual Machine environment, and Kubernetes is a single command away: `minikube start`.