Red Hat OpenShift Development I: Introduction to Containers with Podman

DO188

DO188-RHOSCP4.14-en-1-20240307

Let’s meet each other!

Francesco Marchioni - fmarchio@redhat.com
Red Hat Certified Architect (RHCA)

Course Scheduling

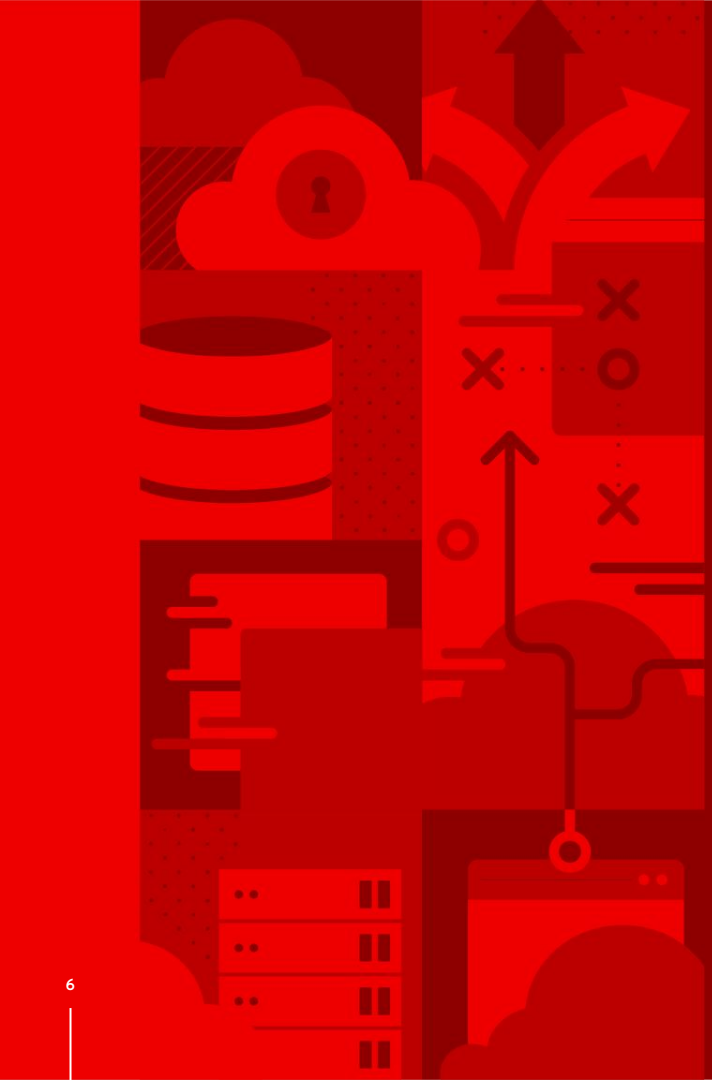
Tuesday to Friday

9:00 AM - 3 PM

Lunch time ??

Online training Attendance

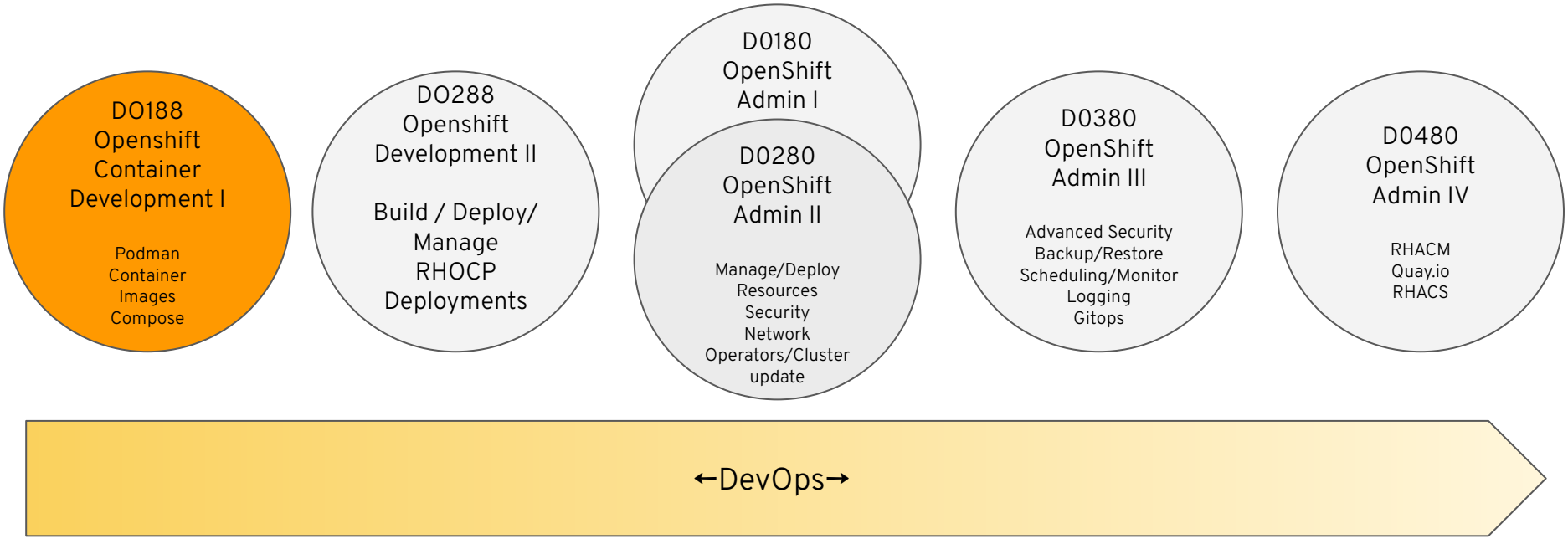
Please remain online during Course hours.

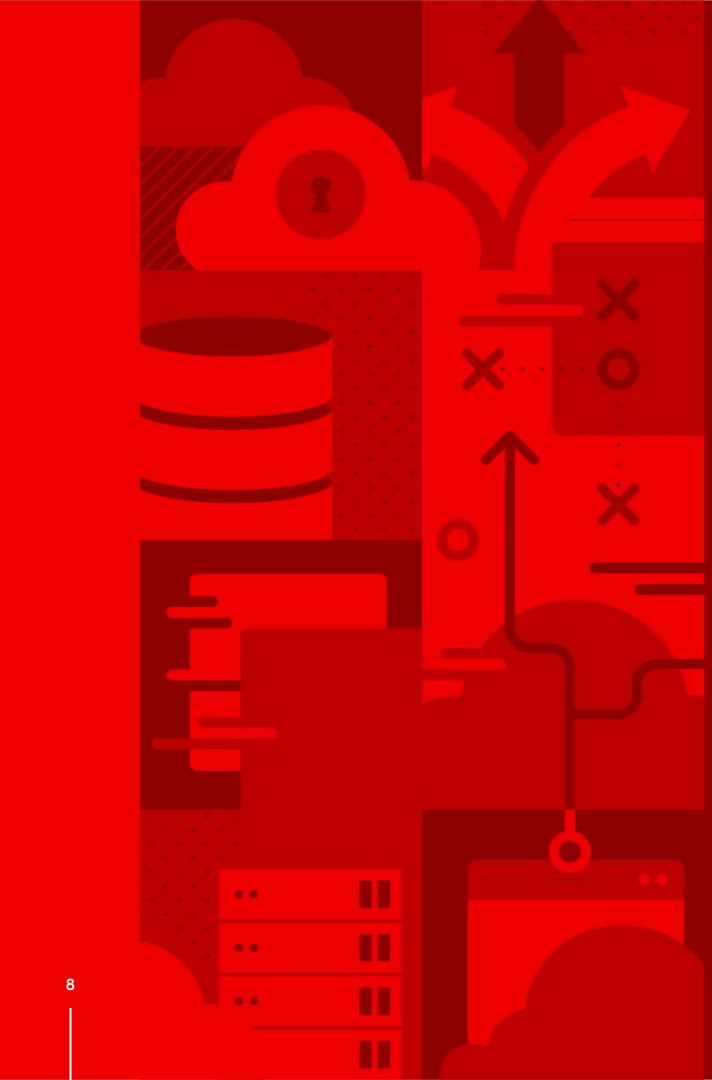


OpenShift Learning Path

Red Hat

OpenShift Training

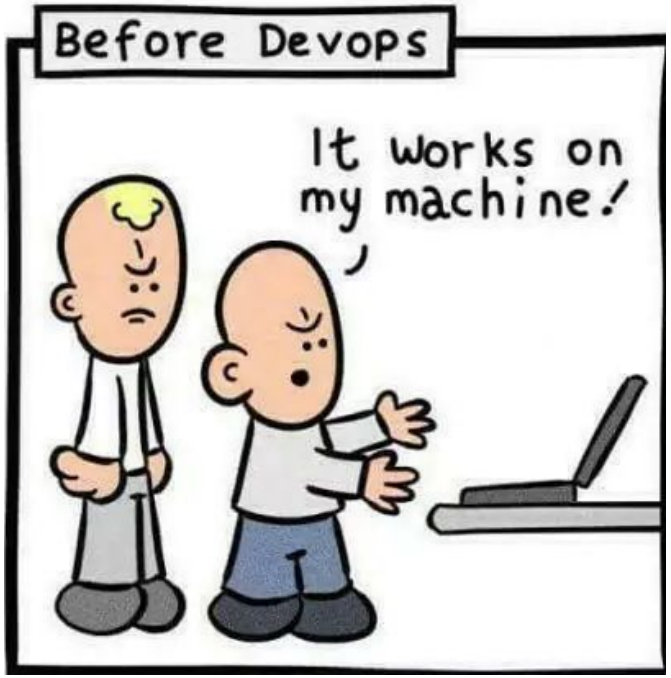




Welcome to World of Containers!

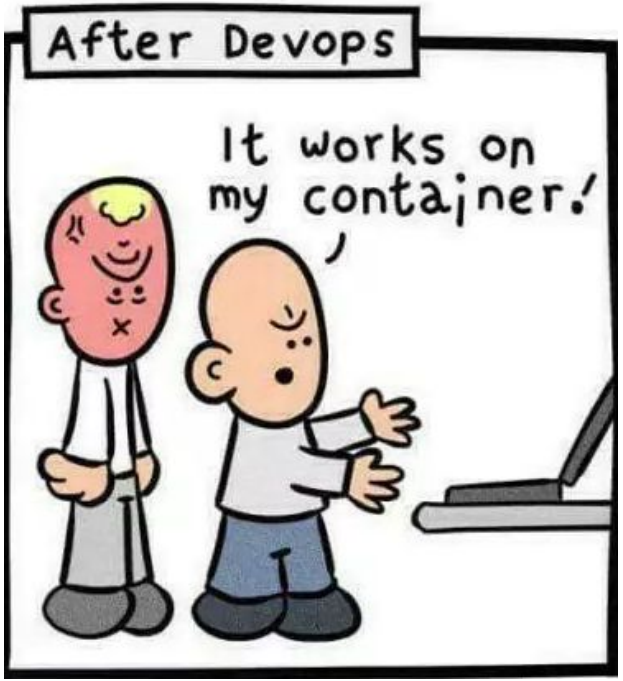
Red Hat

Problems a containers tries to solve…..



Red Hat

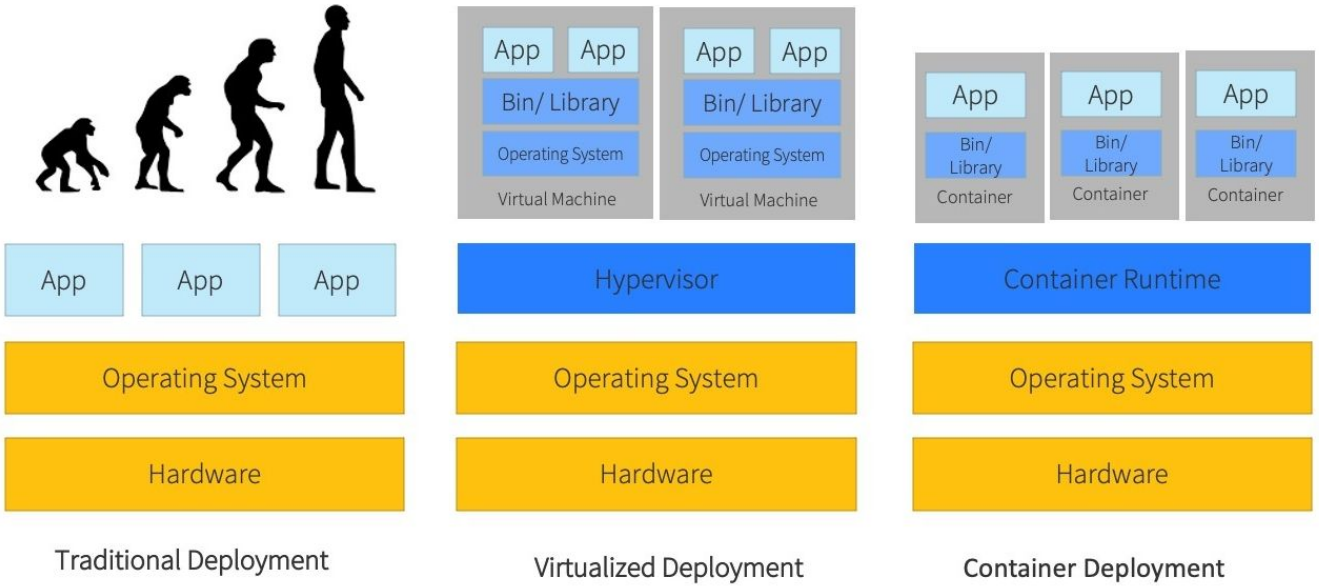
Problems a containers tries to solve…..



Red Hat

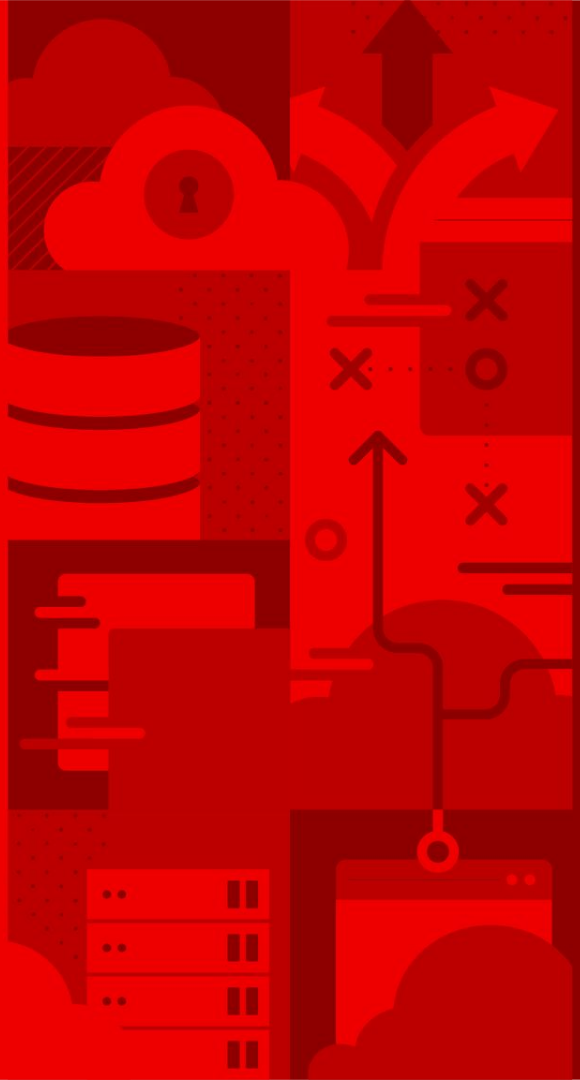
What is a Container ????

Evolution of Application Deployments



Benefits and Challenges of using Containers

Let’s explore this more in detail!



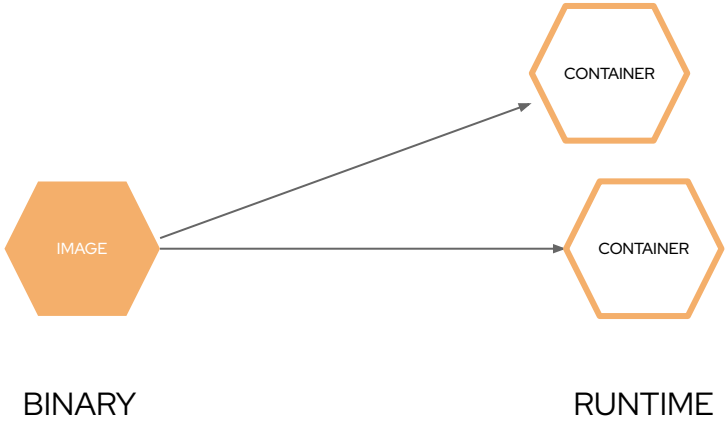
Container Concepts and Terminology

Chapter 1

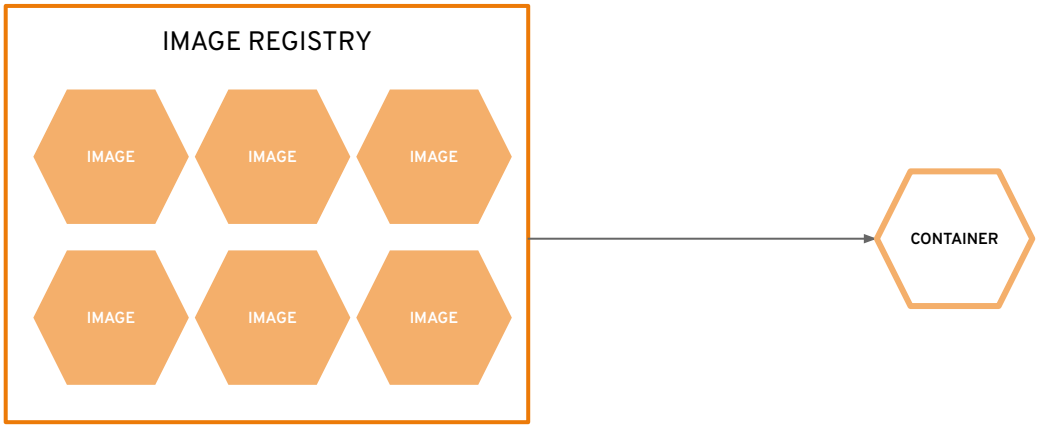
Red Hat

a container is the smallest compute unit

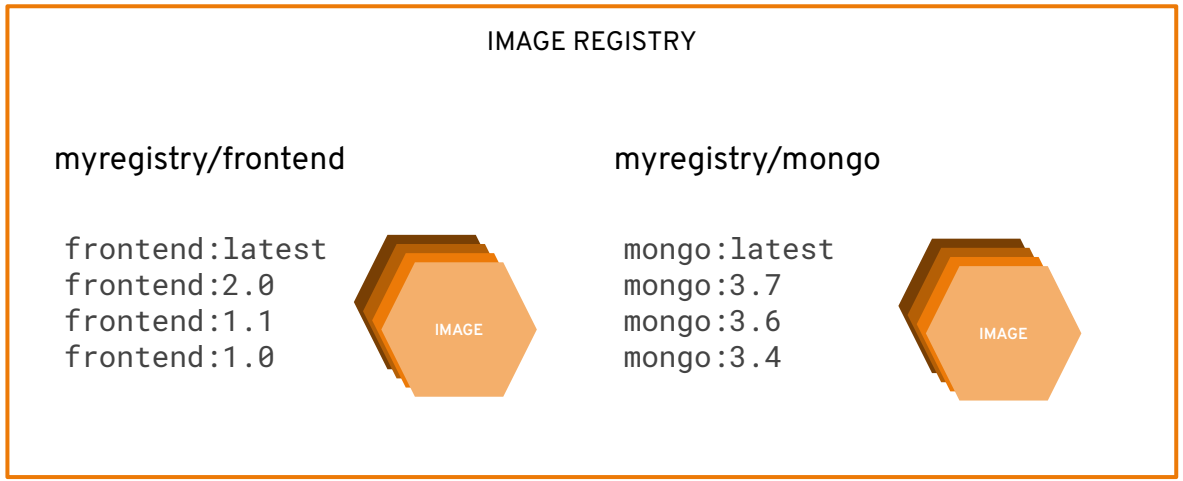
Containers are created from Container Images



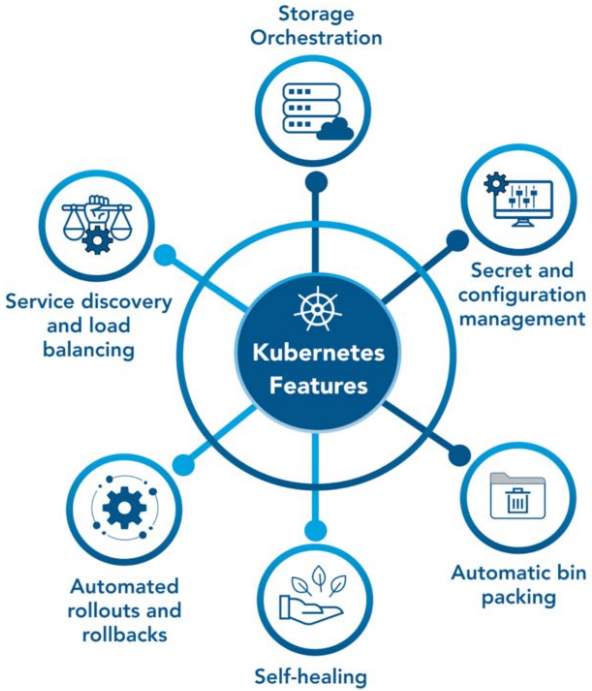
container images are stored in an image registry



an Image Repository contains all versions of an image in the image registry

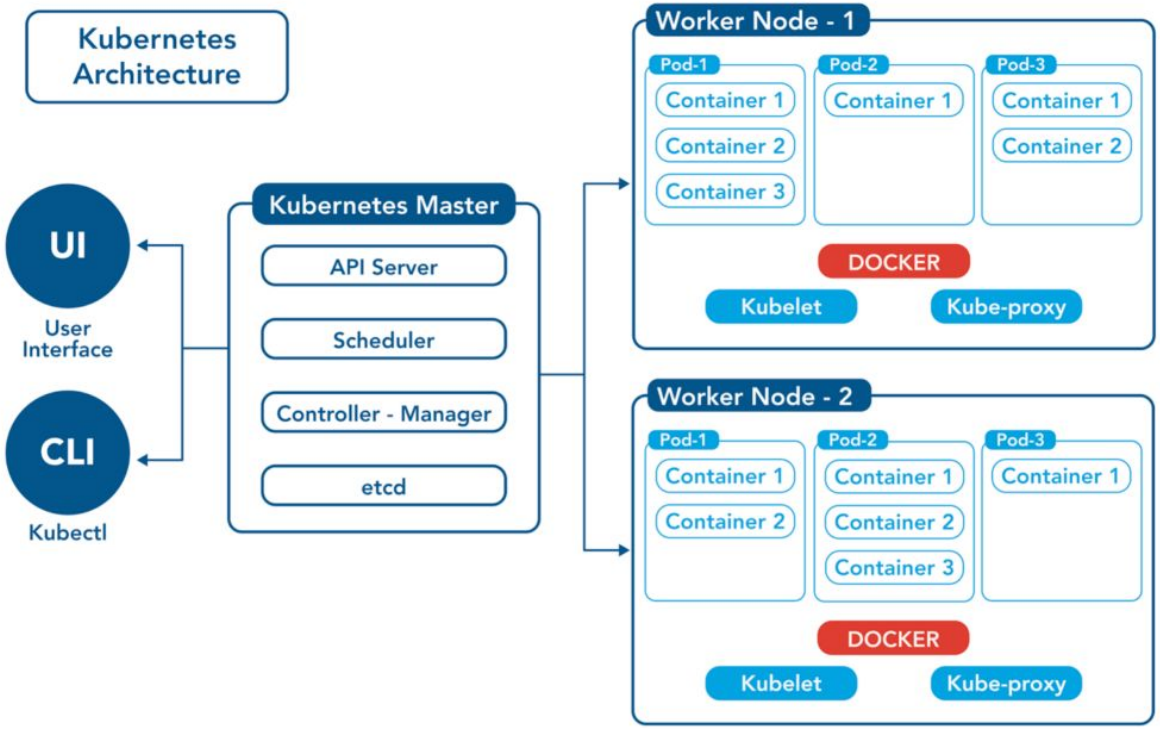


From Containers to Kubernetes

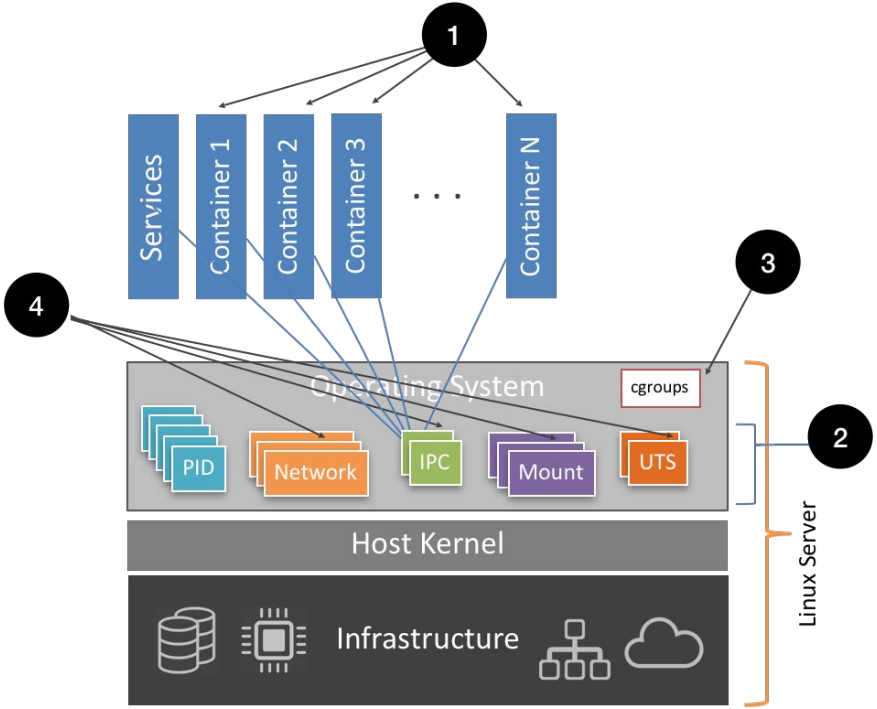


Red Hat

Kubernetes Architecture



Inside Containers



- **(#1)** Each Container has a unique PID namespace running a group of processes.
- **(#2)** Namespaces exist at Kernel level
- **(#3)** Control Groups (cgroups) are used to limit the access to system resources
- **(#4)** Containers will have other dedicated namespaces or will use the default namespace.

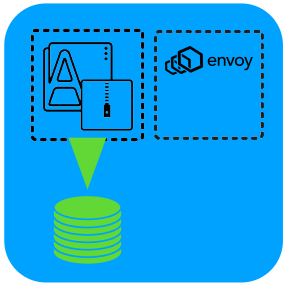
Kubernetes Pod

IP: 10.5.1.20

Pod with single App Container.
This is the most common and
preferred type of Pod.

IP: 10.5.1.73

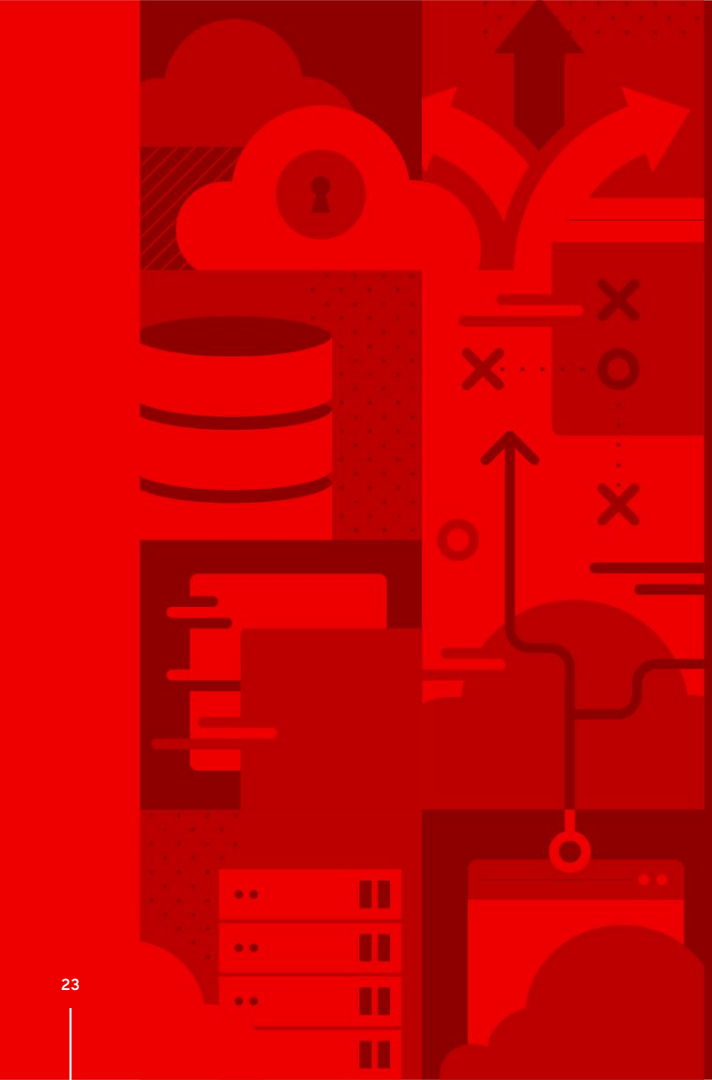
Pod with primary App Container
and helper sidecar Container
(i.e. using helper as data change watcher)



IP: 10.5.1.42

Pod with App Container
and Sidecar Container
(i.e. when using Envoy Proxy)

Container Volumes: The *Volumes* can be shared among *Containers* in the same *Pod*

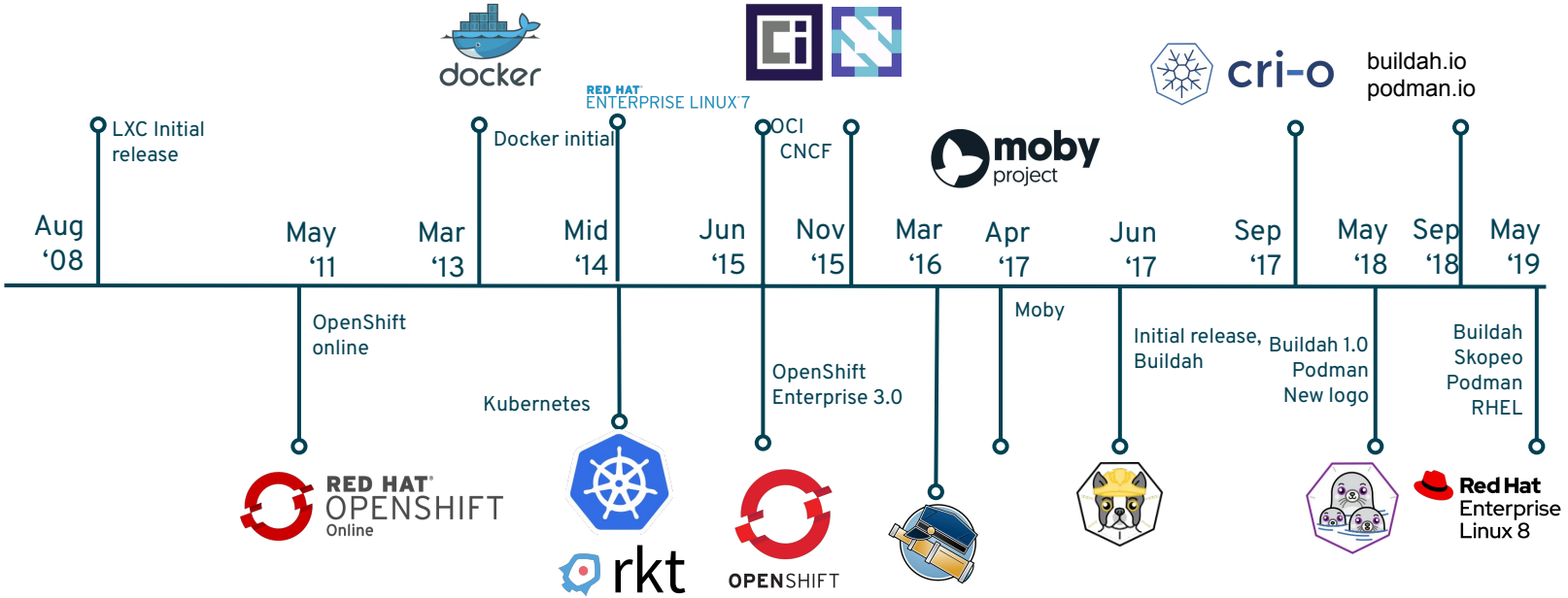


Creating Containers with Podman

Chapter 2

Red Hat

Container innovation continues



24

Red Hat

Introducing Podman and its companion tools

Podman, Buildah and Skopeo

Podman

- the POD MANager - is a tool for managing containers and images, volumes mounted into those containers, and pods made from groups of containers.

Buildah

is a tool that facilitates building Open Container Initiative (OCI) container images without a full container runtime or daemon installed

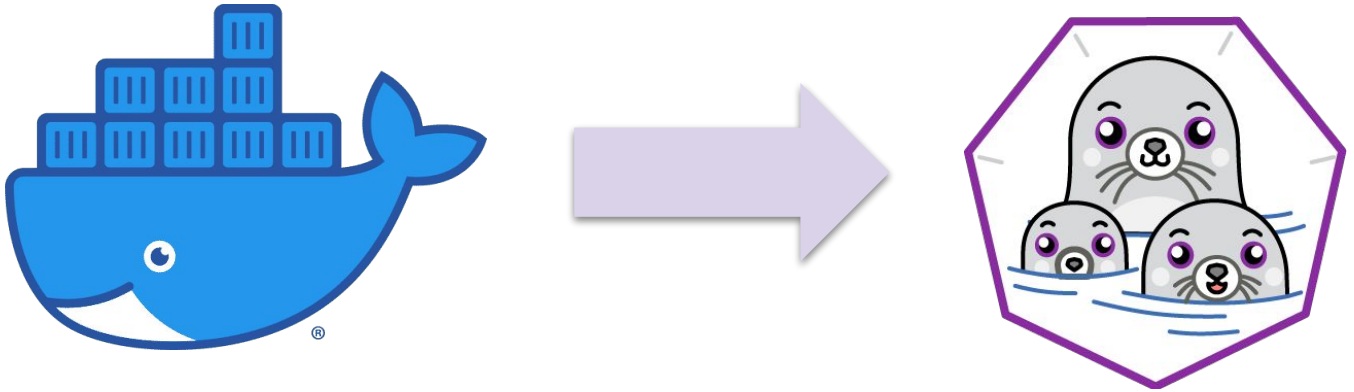
Skopeo

works with remote images registries - retrieving information, images, signing content

Source:
https://github.com/containers

From Docker to Podman

Helping the transition between technologies



Podman is designed to provide a smooth transition experience to Docker users thanks to CLI compatibility and OCI compliance.

However, knowing the technology foundations and main differences between the two engines is useful to provide a better migration experience.

Main differences between Docker and Podman

Overview of the main technology pillars

Docker is a daemon-based container engine that consists of three fundamental pillars:

- Docker daemon, a service running in background that supervises the containers orchestration with a client/Server model
- Docker REST API, an interface to interact with the daemon
- Docker CLI, a CLI interface to manage containers and related resources

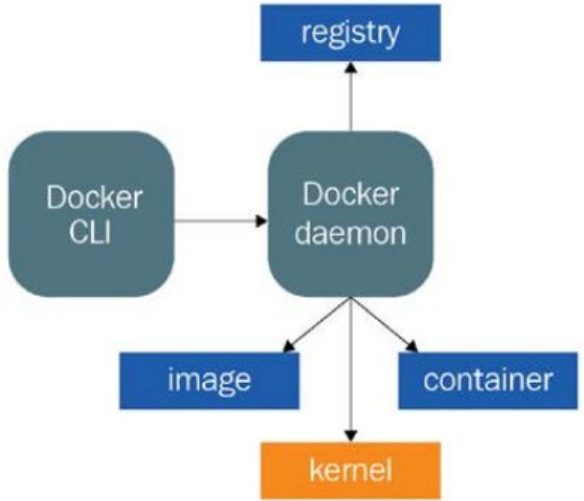
Podman is a container engine that enables users to manage containers, images, as well as storage and network resources:

- Daemonless architecture with fork/exec model
- Standard libpod library for container lifecycle
- Native support for rootless containers and Kubernetes pods
- Full support for OCI runtime/images/storage specs

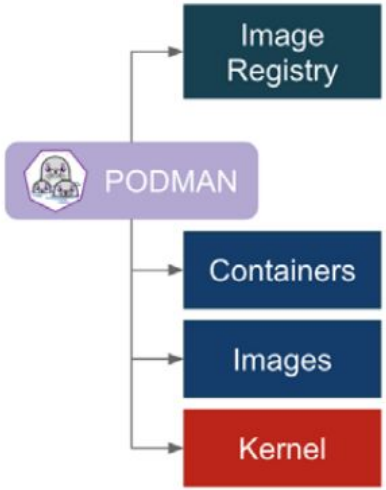
A 10000 ft view

An high level overview of the two different architectures

Docker



Podman



sed ‘s/docker/podman/g’

Different engines, same driving experience

Podman provides a compatible command line interface for most of Docker standardized commands.

Simply swap commands or use podman-docker default aliasing.

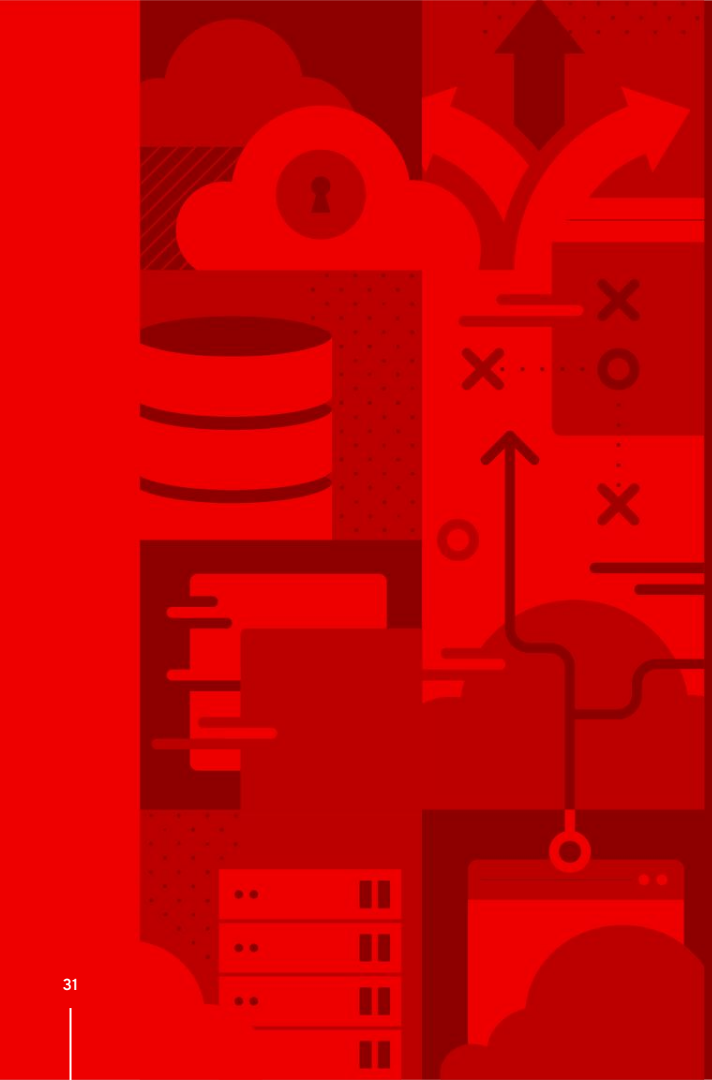
```
# Running an Nginx container with Docker
$ docker run -d --rm docker.io/library/nginx

# Running an Nginx container with Podman
$ podman run -d --rm docker.io/library/nginx
```

Running containers and pods

Pods as minimal execution units

Podman supports the execution of containers and pods. A pod is a single execution unit where multiple containers can share Linux namespaces.

```
# Initializing an empty pod
$ podman pod create --name wp-pod

# Creating containers inside the pod
$ podman create --pod wp-pod --name db -d docker.io/library/mysql
$ podman create --pod wp-pod --name wp -d
docker.io/library/wordpress

# Running the pod
$ podman pod start wp-pod
```



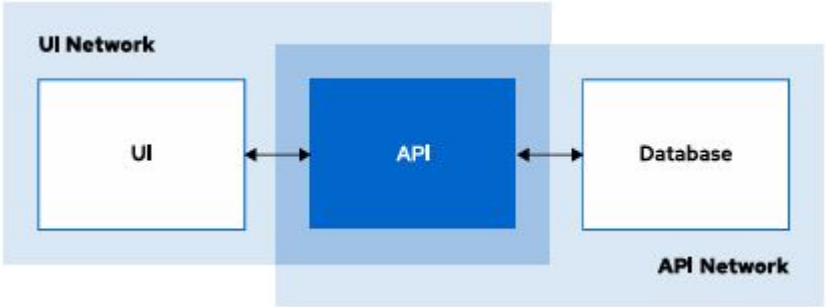
Container Networking

Chapter 2

Red Hat

Podman Networking: Introduction

- Each container runs in its own network namespace by default.
- Podman supports user-defined networks to allow communication between containers.



- Network management differs for rootful and rootless containers.
- Podman uses different network backends: CNI (legacy) and Netavark (default since v4).

Creating and Using Podman Networks

- Use `podman network create <name>` to define a custom bridge network.
- Attach a container to a network with `--net <network>` during `podman run`.
- Containers on the same network can communicate via IP or DNS (if enabled).
- Example:

```
$ podman network create mynet

$ podman run --net mynet --name web nginx
```

Effect of Joining a Network

- 🔗 Containers in a network (e.g., bridge) can reach each other
  Shared network namespace enables communication via IP or DNS.

- 🧠 DNS resolution works if enabled in the network settings
  Container names are resolvable within the same network.

- 🛑 Containers outside any defined network (e.g., using pasta) are isolated
  No communication between containers, even via IP.

- 🧭 Network connection affects access from host, container linking, and service discovery

Proper network setup is crucial for multi-container applications.

Bridge vs Pasta Networking

- 🌉 **Bridge**: Shared virtual network; supports DNS, IP access, and container communication. Ideal for multi-container apps that need to talk to each other.

- 🔒 **Pasta**: Rootless-only; per-container isolation, no inter-container communication. Designed for strong security and minimal configuration.

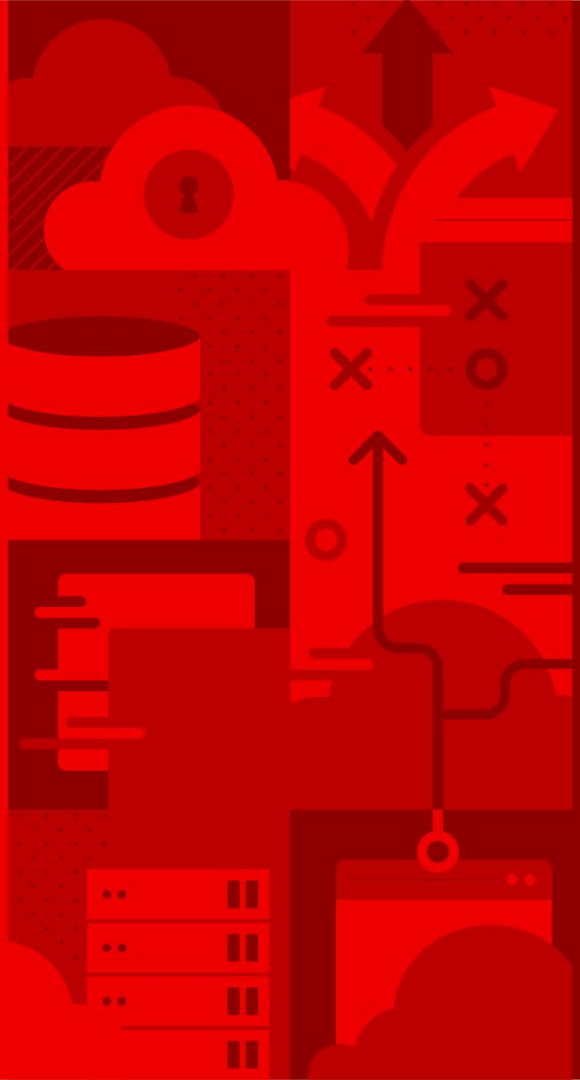
- ⚙️ Bridge is default for rootful containers; Pasta is default for rootless (if no `--net`). The default mode depends on user privileges.

- 🚫 Containers using pasta cannot be added to bridge networks later.
  Network modes are not compatible post-creation.

💡 Summary and Best Practices

- Use bridge networks for multi-container applications and service discovery.
- Use pasta for secure, isolated rootless containers with minimal config.
- *Always* define networks explicitly in scripts or production setups.
- Inspect networks with `*podman network inspect <name>*`.

Dashboard per lab basics-exposing



Container Registries

Chapter 3

Red Hat

Introduction to Registries

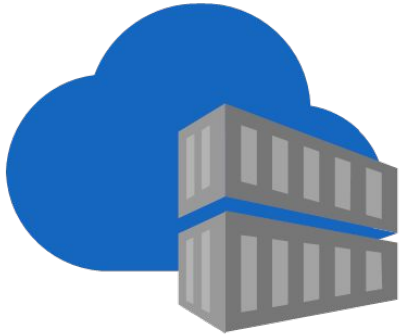
- ✅ Advantages of using Registries
- Container Registries and Podman
- Objectives:
  - 🧰 Understand how container images are packaged with all necessary dependencies.
  - 🌐 Learn how registries are used to store and share container images.
  - 🧭 Get an overview of managing images with Podman.

What is the Purpose of Registries?

Registries allow you to store and share container images in a controlled way.

Examples of Registries:

- Quay.io
- Red Hat Registry
- Docker Hub
- Amazon ECR



Example Command:

```
podman pull registry.redhat.io/ubi8/ubi:8.6
```

This pulls an image from the Red Hat Registry.

Red Hat Registries

Red Hat Registries:

- **registry.access.redhat.com**: No authentication
- **registry.redhat.io**: Requires authentication.

Red Hat Ecosystem Catalog:

- Visit catalog.redhat.com to search images and view technical details.

Useful Container Images:

- UBI 9: registry.access.redhat.com/ubi9 – Base image on RHEL 9.
- Python 3.9: registry.access.redhat.com/ubi9/python-39 – UBI-based image with Python runtime.
- Node.js 18: registry.access.redhat.com/ubi9/nodejs-18 – UBI-based image with Node.js runtime.
- Go Toolset: registry.access.redhat.com/ubi9/go-toolset – Contains Go runtime.

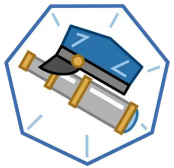
Managing Registries with Podman and Skopeo

Podman:

- Uses fully qualified image names:
- Format: Registry URL/User or organization/Image repository:Image tag
- If using an unqualified name (e.g., ubi8/python-39), Podman searches registries defined in /etc/containers/registries.conf.

Skopeo:

- A command-line tool to work with container images without local storage.
- Functions:
  - Inspect remote images.
  - Copy images between registries.
  - Sign images using OpenPGP keys.
  - Convert image formats (e.g., Docker to OCI).



Authentication and Credentials

Authentication Requirements:

- Some registries (e.g., registry.redhat.io) need user authentication.

Logging In:

- Use the command: `podman login registry.redhat.io`
- Then enter your username and password.

Credential Storage:

- Podman stores credentials in ${XDG_RUNTIME_DIR}/containers/auth.json.

Best Practices:

- Always use fully qualified image names to avoid pulling duplicate or unintended images. Do not use latest() images in production!

Managing Images

Managing Images with Podman

- Managing Images with Podman
- Objectives:
  - Learn how to pull, tag, build, push, inspect, and remove container images.
  - Understand best practices for image versioning and management.

Overview of Image Management

Key Operations:

- Tagging: Map image versions to product updates.
- Pulling: Download images from registries (e.g., using podman pull).
- Building: Create images from a Containerfile with podman build.
- Pushing: Share images by pushing them to remote repositories.
- Inspecting: Retrieve image metadata using podman image inspect.
- Removing: Clean up unused or dangling images to free storage.

Image Versioning & Tagging

Deployment Artifacts:

- Container images package software and are versioned similar to product releases.

Semantic Versioning:

- Format: MAJOR.MINOR.PATCH
- MAJOR: Incompatible changes
- MINOR: Backward-compatible improvements
- PATCH: Bug fixes

Naming Format:

- [<repository>/<namespace>/]<image name>[:<tag>]

Pulling and Building Images

Pulling Images:

- Use: `podman pull IMAGE_NAME` (e.g., podman pull quay.io/argoproj/argocd)
- Note: Unqualified names are resolved using the registries defined in /etc/containers/registries.conf.

Building Images:

- Create images from a Containerfile with:

  `podman build --file Containerfile --tag quay.io/YOUR_QUAY_USER/IMAGE_NAME:TAG`

Storage Locations:

- Non-root: ~/.local/share/containers
- Root: /var/lib/containers

Pushing and Inspecting Images

Pushing Images:

- Authentication: Login with podman login REGISTRY
- Push using:

```
podman push quay.io/YOUR_QUAY_USER/IMAGE_NAME:TAG
```

Inspecting Images:

- Use: `podman image inspect IMAGE_NAME` to view metadata

Go Templating:

- Extract specific details with the --format option, e.g.:
- `podman image inspect IMAGE --format="{{.Config.Cmd}}"`

Removing Images

- Delete a single image:

  `podman image rm <IMAGE_NAME:TAG>`

- Force removal (stops dependent containers):

  `podman image rm -f IMAGE_NAME:TAG`

- Remove a Container

  `podman rm <CONTAINER>`

- Remove all dangling images [*]:

  `podman image prune -a`

  [*] = Untagged images generated during the build and not used any more

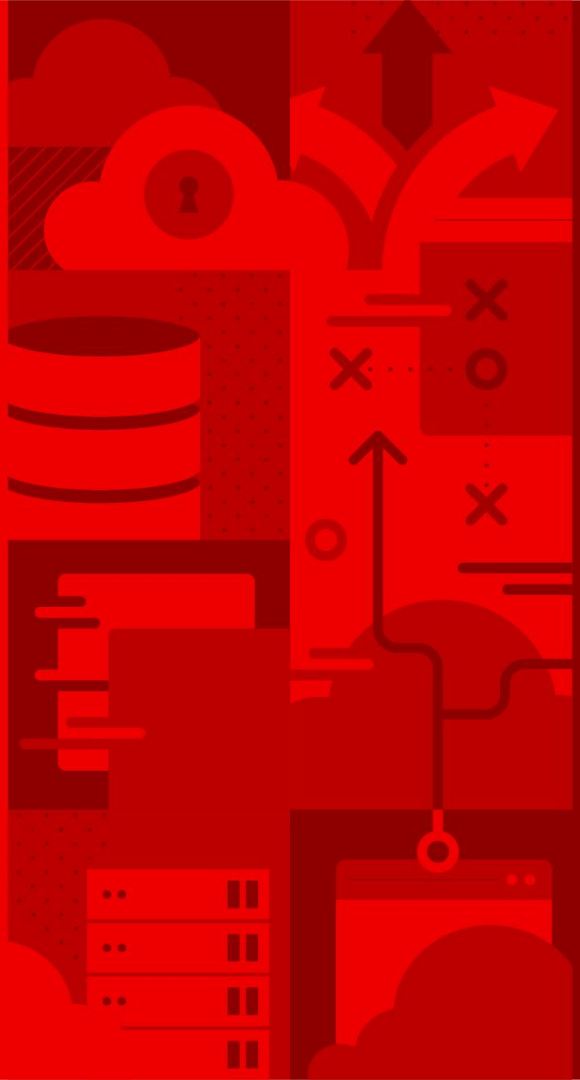
Exporting and Importing

- Export: Save a container's filesystem as a .tar file:

```
podman export -o mytarfile.tar CONTAINER_ID
```

- Import: Create an image from an exported .tar file:

```
podman import mytarfile.tar IMAGE_NAME:TAG
```



Custom Container Images

Chapter 4

Red Hat

Introduction

What is a Containerfile?

- A list of instructions for building a container image.
- Each instruction creates a new image layer.
- Similar to Dockerfiles but independent of any container runtime engine.
- Used to create multiple containers from the same image.

Why Use Containerfiles?

- Standardized way to define and build container images.
- Compatible with tools like Podman and Docker.
- Enables version control and reproducibility.

Exporting and Importing

What is a Base Image?

- The starting point for building a container image.
- Defines the Linux distribution, package manager, and preinstalled dependencies.

Types of Red Hat Universal Base Images (UBI):

- Standard – Includes DNF, systemd, gzip, and tar.
- Init – Uses systemd to manage multiple applications in one container.
- Minimal – A smaller image with the microdnf package manager.
- Micro – The smallest image with only essential components.

Where to Find UBIs?

- Available in the Red Hat Container Catalog.

Exporting and Importing

- Basic Instructions:
  - FROM – Defines the base image.
  - WORKDIR – Sets the working directory inside the container.
  - COPY / ADD – Copies files into the container image.
  - RUN – Executes commands during image build.
- Execution and Configuration:
  - ENTRYPOINT – Specifies the executable for the container.
  - CMD – Provides arguments for the ENTRYPOINT.
  - USER – Changes the user inside the container.
- Metadata and Networking:
  - LABEL – Adds metadata.
  - EXPOSE – Documents ports used by the container.
  - ENV – Defines environment variables.

Example Containerfile

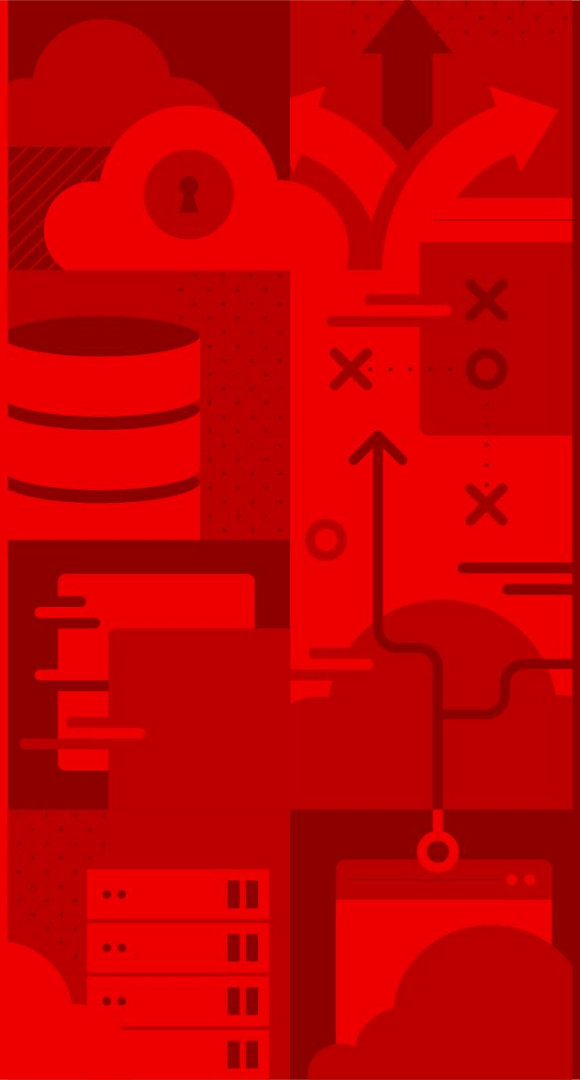
```
# Apache Web Server Containerfile

FROM registry.redhat.io/ubi8/ubi:8.6

LABEL description="Custom httpd container image"

RUN yum install -y httpd

EXPOSE 80

ENV LogLevel "info"

ADD http://someserver.com/filename.pdf /var/www/html

COPY ./src/ /var/www/html/

USER apache

ENTRYPOINT ["/usr/sbin/httpd"]

CMD ["-D", "FOREGROUND"]
```

Key Instructions in This Example:

- Uses ubi8/ubi:8.6 as base image.
- Installs Apache (httpd) using RUN.
- Defines EXPOSE for port 80.
- Adds files with ADD and COPY.
- Runs as user apache for security.
- Uses ENTRYPOINT and CMD to start the web server.

Images Tagging Best Practices

- Tagging Images:
  - Format: image-name:tag
  - Example: my-app:1.0
  - Default tag is latest if omitted.
- Best Practices:
  - Always specify a tag to prevent unintended updates.
  - Use meaningful labels (LABEL) for better organization.
  - Avoid running containers as root (USER).
  - Minimize image size by using lightweight base images.
- Cleaning Up Unused Images:
  - podman image prune – Removes unused images.
  - podman rmi <image> – Deletes a specific image.



Rootless Containers

Chapter 4

Red Hat

Introduction to Rootless Podman

What is Rootless Podman?

- A container runtime that allows running containers without requiring root privileges.
- Enhances security by isolating containerized applications from the host system.
- Uses user namespaces to map container users to non-root host users.

Why use Rootless Podman ?

- **Security Benefits:**
  - Reduces the risk of privilege escalation.
  - Limits potential attack surfaces in case of container escape.
- **Flexibility:**
  - Enables non-root users to run containers without admin intervention.
  - Works well in multi-user environments.
- **Compatibility:**
  - Supports standard container workloads while ensuring security.

How Rootless Podman works

- **Key Concepts:**
  - Podman starts containers as a user-space process without requiring root access.
  - Uses user namespaces to map container UIDs/GIDs to unprivileged host users.
  - The container runtime does not need elevated privileges.
- **Process Flow:**
  - Podman starts the container.
  - The containerized process attaches to the systemd parent process.

Prerequisites for Running Rootless Containers

- **Kernel Features Required:**
  - cgroup v2: Allows resource control without root privileges.
  - slirp4netns: Enables user-mode networking for rootless containers.
  - fuse-overlayfs: Manages the Copy-On-Write (COW) filesystem efficiently.
- **Setting Up Subordinate User IDs:**
  - /etc/subuid and /etc/subgid define UID/GID mappings.
  - Use usermod --add-subuids and podman system migrate to configure.

Changing the Container User

- Default Behavior:
  - Containers typically run as root inside, posing security risks.
  - Exploits can escalate privileges from the container to the host.
- Best Practice:
  - Use the USER directive in the Containerfile:

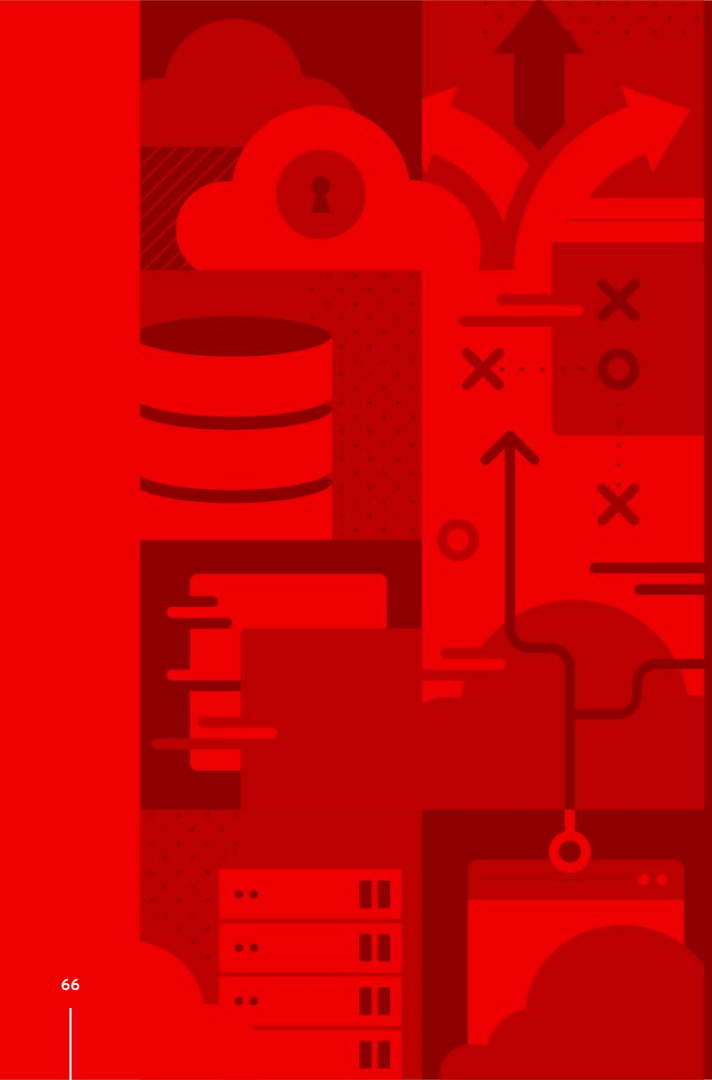
```
FROM registry.access.redhat.com/ubi9/ubi

RUN adduser --no-create-home --system --shell /usr/sbin/nologin appuser

USER appuser

CMD ["python3", "-m", "http.server"]
```

User Mapping in Rootless Containers

- How User Mapping Works:
  - Container users map to unprivileged host users.
  - Example mapping in /etc/subuid:
    student:100000:65536
  - podman top <container_id> huser user shows mapped users.
- Example:
  - Root inside the container maps to UID 1000 on the host.

Limitations of Rootless Containers

- Common Restrictions:
  - Cannot bind to privileged ports (e.g., 80, 443) unless configured (sysctl).
  - Some applications requiring root (e.g., ping) may need workarounds.
  - Not all applications can be containerized without root.
- Workarounds:
  - Use port forwarding instead of binding directly to privileged ports.
  - Adjust kernel parameters to allow specific capabilities (e.g., ping)



Managing Storage

Chapter 5

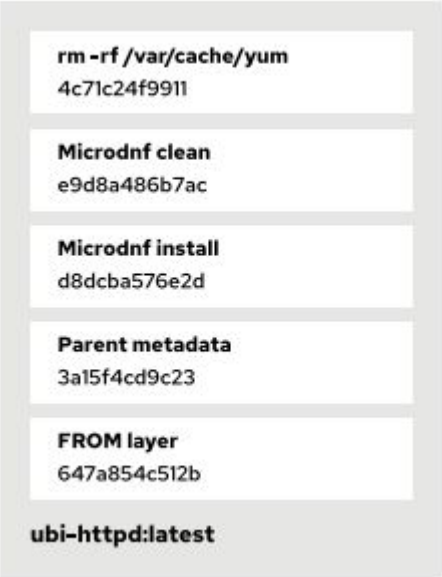
Red Hat

Introduction to Volume Mounting in Podman

- Objective: Understand the process of mounting volumes and its common use cases.
- **Why Use Volume Mounting?**
  - Persistent storage across container deletions
  - Efficient data sharing between containers
  - Performance improvements for write-heavy applications

Understanding the Copy-on-Write (COW) File System

- Container Image Layers
  - Each modification creates a new read-only layer.
  - The final image is a union of multiple layers.
- Example Image Structure (using podman image tree)
  - Multiple immutable layers form the base of a container.
- Implication: Containers share base layers, but each container has a separate read/write layer.



The Need for External Storage

- **Challenges with COW File System:**
  - Write operations are inefficient due to layer copying.
  - Data is lost when the container is removed.
- **Solution**: Store data outside of the container using bind mounts or volumes.

Types of External Mounts in Podman

1. **Bind Mounts**
   - Maps a host directory into the container.
   - Requires manual management of permissions.
2. **Volumes**
   - Managed by Podman.
   - More portable and persistent.
3. **Tmpfs Mounts**
   - Memory-backed, ephemeral storage.

Using Bind Mounts

Command Syntax:

```
podman run -p 8080:8080 --volume /www:/var/www/html:ro
registry.access.redhat.com/ubi8/httpd-24:latest
```

- Key Considerations:
  - Read-only (ro) prevents modifications.
  - Ensure correct file permissions.

Troubleshooting Bind Mounts

- **File Permission Issues:**
  - Use podman unshare ls -l /www/ to check permissions.
- **SELinux Issues:**
  - Check with ls -Zd /www

Fix using :z (shared access) or :Z (exclusive access):

```
podman run -p 8080:8080 --volume /www:/var/www/html:Z
registry.access.redhat.com/ubi8/httpd-24:latest
```

Using Podman Volumes

- Why Use Volumes?
  - More portable and easier to manage.
  - Ensures consistency across different hosts.
- Creating and Using Volumes:

```
podman volume create http-data

podman run -p 8080:8080 --volume http-data:/var/www/html \

registry.access.redhat.com/ubi8/httpd-24:latest
```

- How to change mount point volume path in podman?
  https://access.redhat.com/solutions/7020076

Advanced Volume Creation

- Podman volume create allows to use advanced options.
- For example, you can map an NFS volume by setting the appropriate type and device settings [1]

Creating and Using Volumes:

```
$ podman volume create --driver local --opt type=nfs --opt
o=addr=my-nfs-share.example.com,rw,context="system_u:object_r:container_file_t:s0"
--opt device=:/opt/container-nfs-storage nfs-volume-01
```


[1] https://access.redhat.com/solutions/6415411

Managing Volumes

- Inspect Volume Details:

```
podman volume inspect http-data
```

- Listing All Volumes:

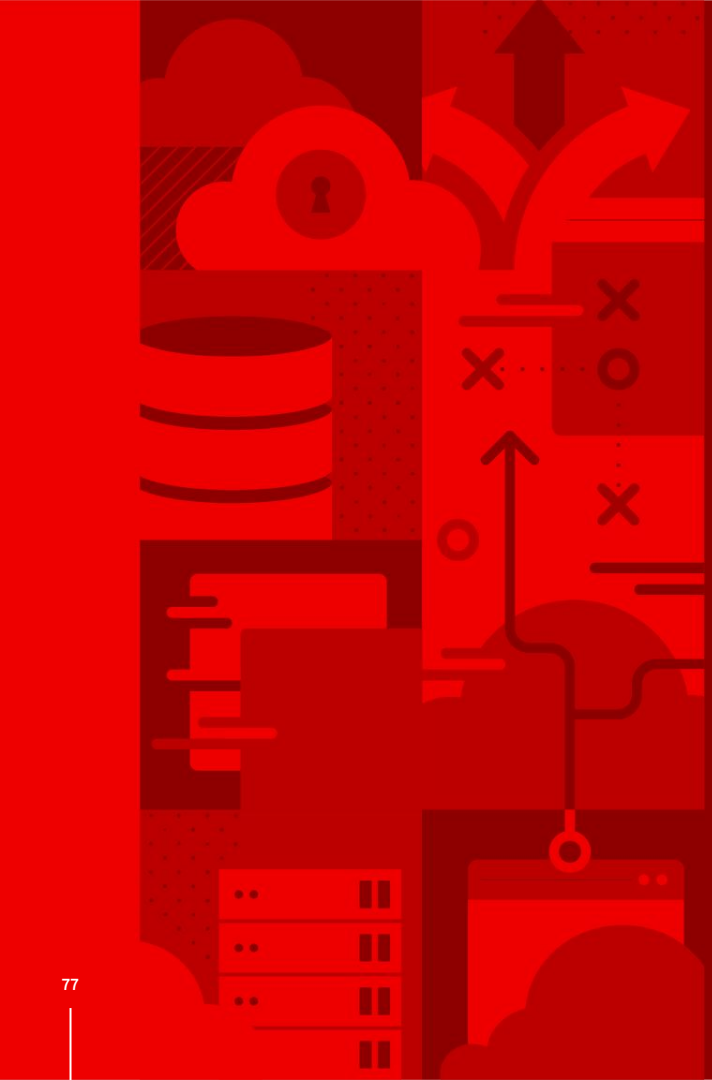
```
podman volume ls
```

- Removing Unused Volumes:

```
podman volume rm http-data
```

Summary & Best Practices

- Use Bind Mounts for:
  - Temporary data.
  - Environment-specific configurations.
- Use Volumes for:
  - Persistent data storage.
  - Portability across environments.
  - Better management by Podman.
- Ensure correct permissions when using bind mounts.



Troubleshooting Containers

Chapter 6

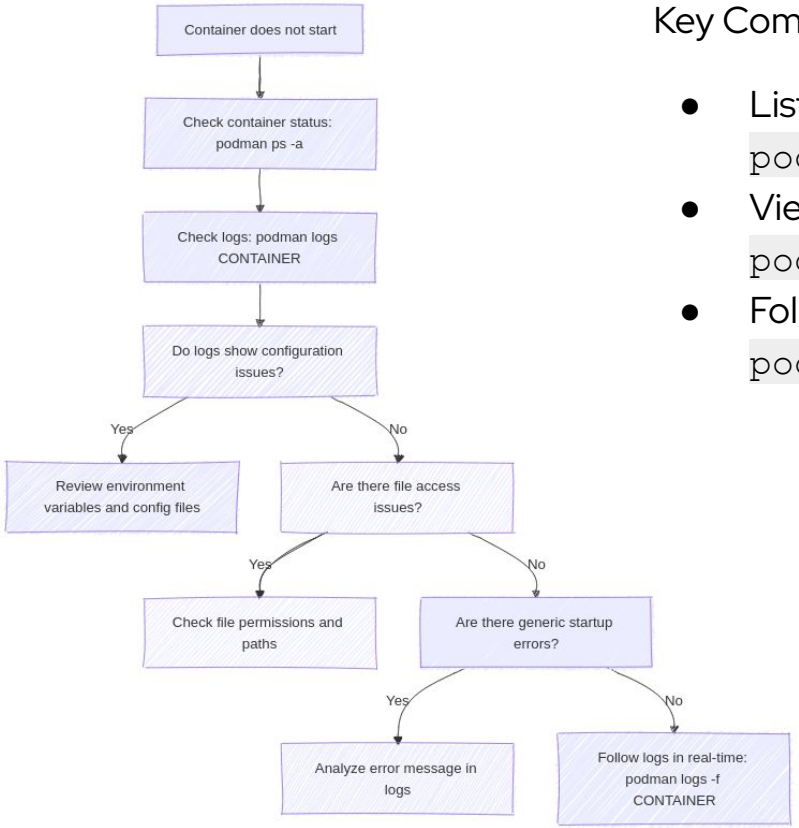
Red Hat

Summary & Best Practices

Objectives:

- Read container logs.
- Troubleshoot common container problems.
- Check Red Hat KCS for Podman Solutions

Troubleshoot common issues



Key Commands:

- List all containers: `podman ps -a`
- View container logs: `podman logs CONTAINER`
- Follow logs in real-time: `podman logs -f CONTAINER`

Podman internal Logs

- How to check Podman's own logs?

```
$ journalctl _COMM=podman

Nov 19 11:41:08 r95jbu01 podman[12609]: 2024-11-19 11:41:08.159570184 +0900 JST
m=+0.203488520 system refresh
Nov 19 11:41:21 r95jbu01 podman[12609]: 2024-11-19 11:41:21.880379337 +0900 JST
m=+13.924297681 image pull
02141a49ee4eaf2aa824472c45a281be14e3e85299e117891da1153a5eddc4c7 registry.access.r>
Nov 19 11:41:36 r95jbu01 podman[12699]: 2024-11-19 11:41:36.476350952 +0900 JST
m=+0.073530605 image pull
02141a49ee4eaf2aa824472c45a281be14e3e85299e117891da1153a5eddc4c7 ubi8
Nov 19 11:41:36 r95jbu01 podman[12699]: 2024-11-19 11:41:36.586357214 +0900 JST
m=+0.183536868 container create
```

- Use the --log-level <level> option to set the logging level.

Troubleshoot containers networking

Common Issues:

- Incorrect port mapping.
- No network access between containers.
- DNS resolution failures.

How to troubleshoot containers using nsenter:

https://access.redhat.com/solutions/1611883

Port Mapping Verification:

```
podman port CONTAINER
```

Check open ports in the container:

```
podman exec -it CONTAINER ss -pant
```

Use nsenter for network namespace troubleshooting:

```
podman inspect CONTAINER --format '{{.State.Pid}}'
```

```
sudo nsenter -n -t CONTAINER_PID ss -pant
```

Troubleshoot containers connectivity

Verify Network Assignment:

```
podman inspect CONTAINER --format='{{.NetworkSettings.Networks}}'
```

Check if DNS is enabled:

```
podman network inspect NETWORK
```

Podman Events for troubleshooting

- Monitor container events:

```
podman events --stream=false
```

- Filter events by type:

```
podman events --filter event=create --filter type=container
--stream=false
```

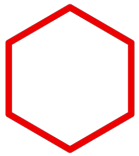
- View past events:

```
podman events --since 5m --stream=false
```

Red Hat OpenShift and Kubernetes Core Concepts

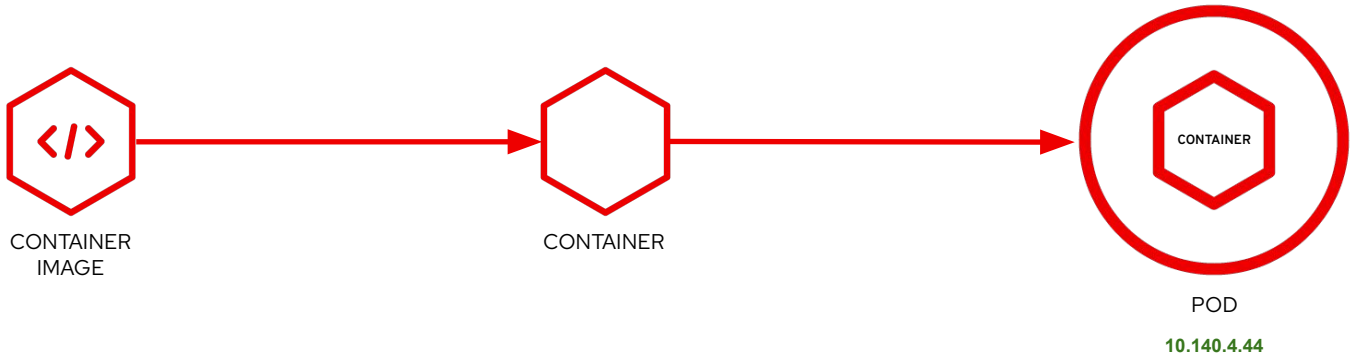
Red Hat

A container is the smallest compute unit



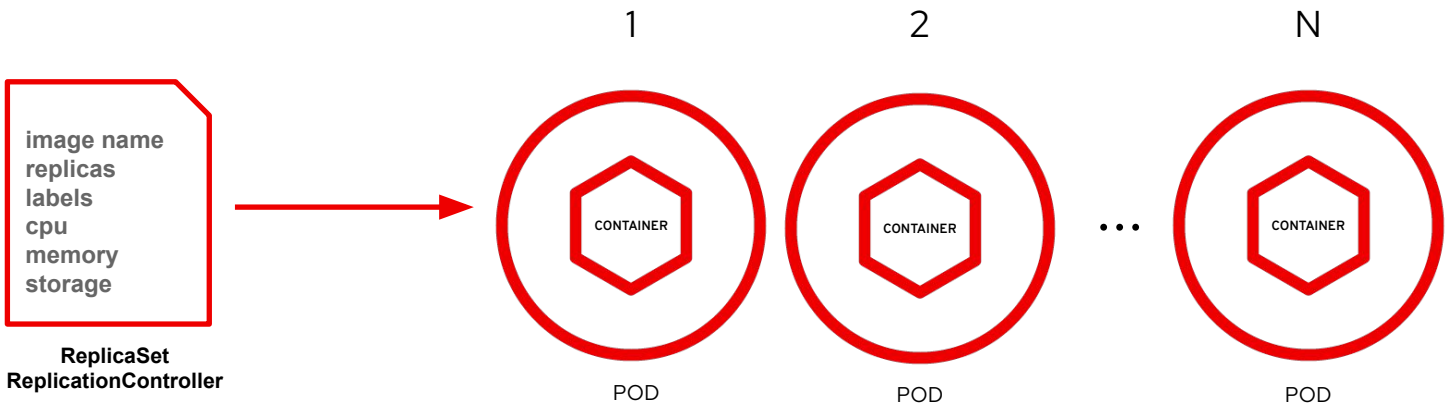
CONTAINER

Everything runs in pods

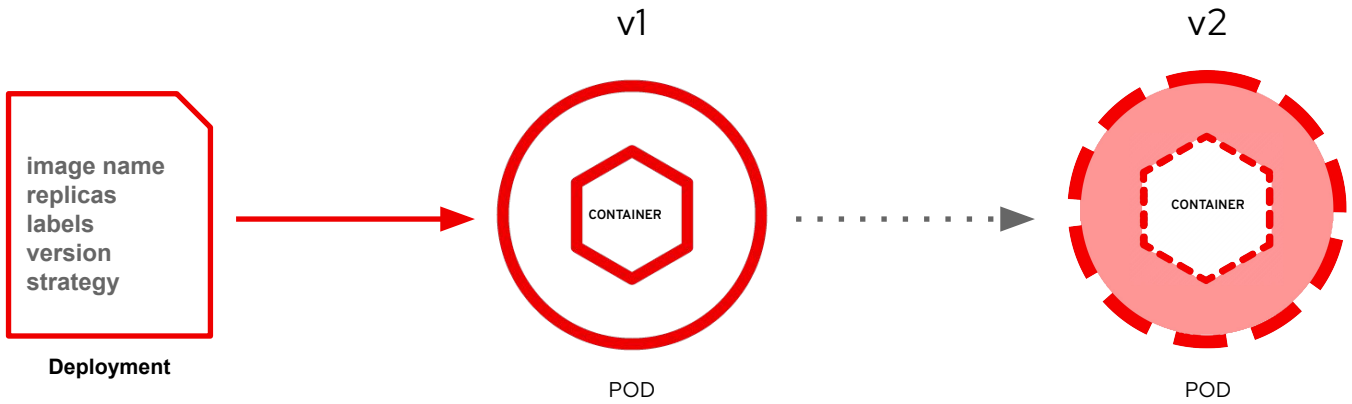


Containers are wrapped in pods which are units of deployment and management

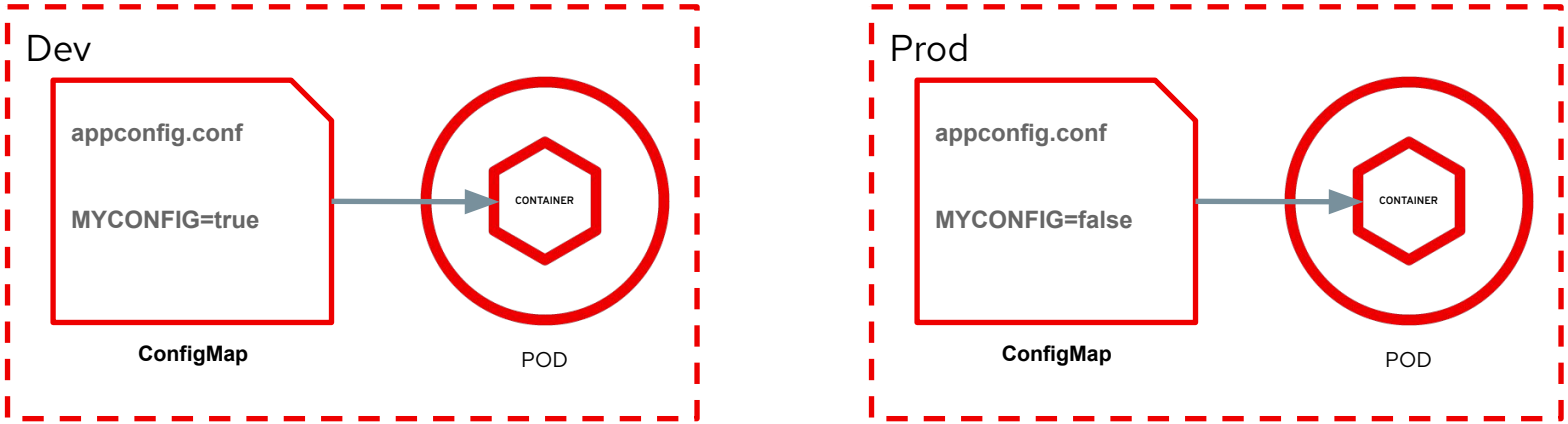
ReplicationControllers & ReplicaSets ensure a specified number of pods are running at any given time



Deployments define how to roll out new versions of Pods

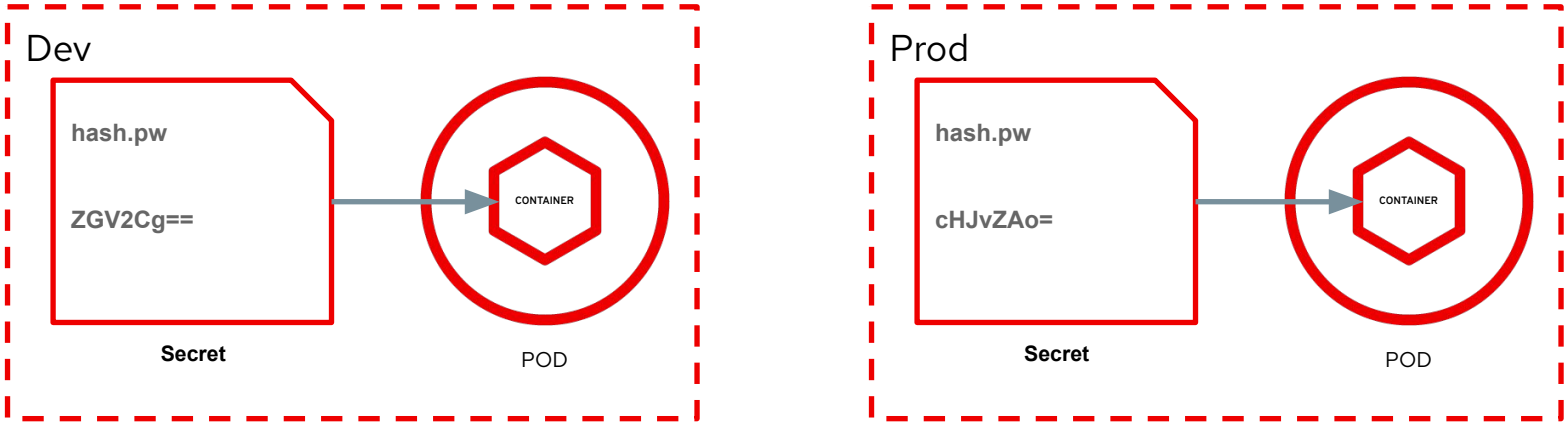


Configmaps allow you to decouple configuration artifacts from image content



Red Hat

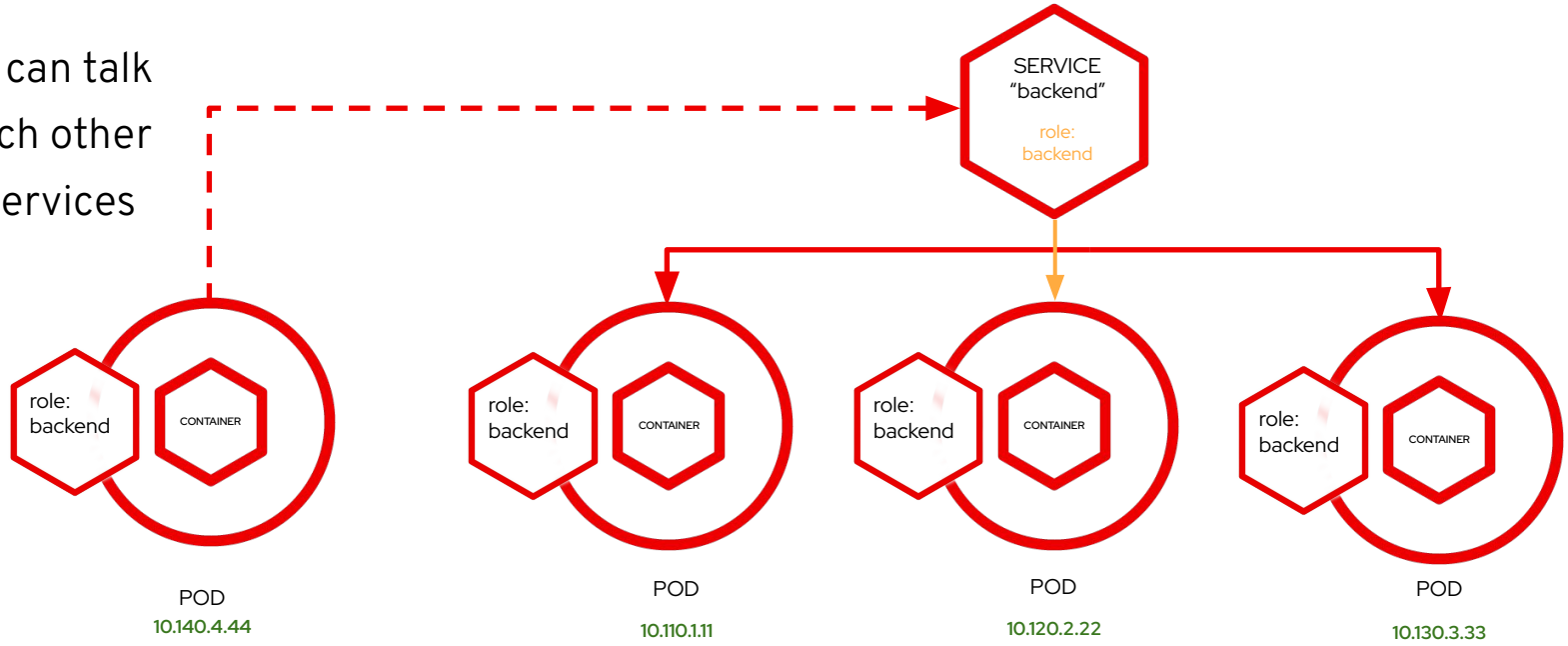
Secrets provide a mechanism to hold sensitive information such as passwords



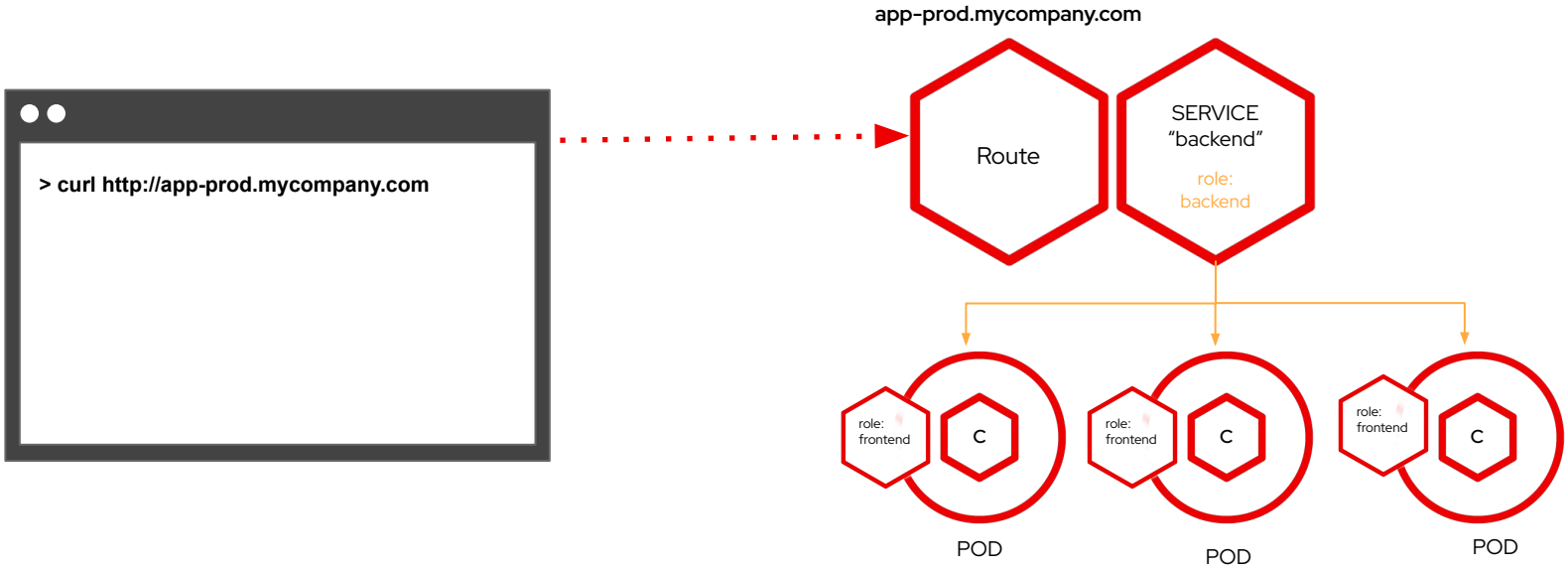
The etcd datastore can be encrypted for additional security
https://docs.openshift.com/container-platform/4.6/security/encrypting-etcd.html

Services provide internal load-balancing and service discovery across pods

Apps can talk to each other via services

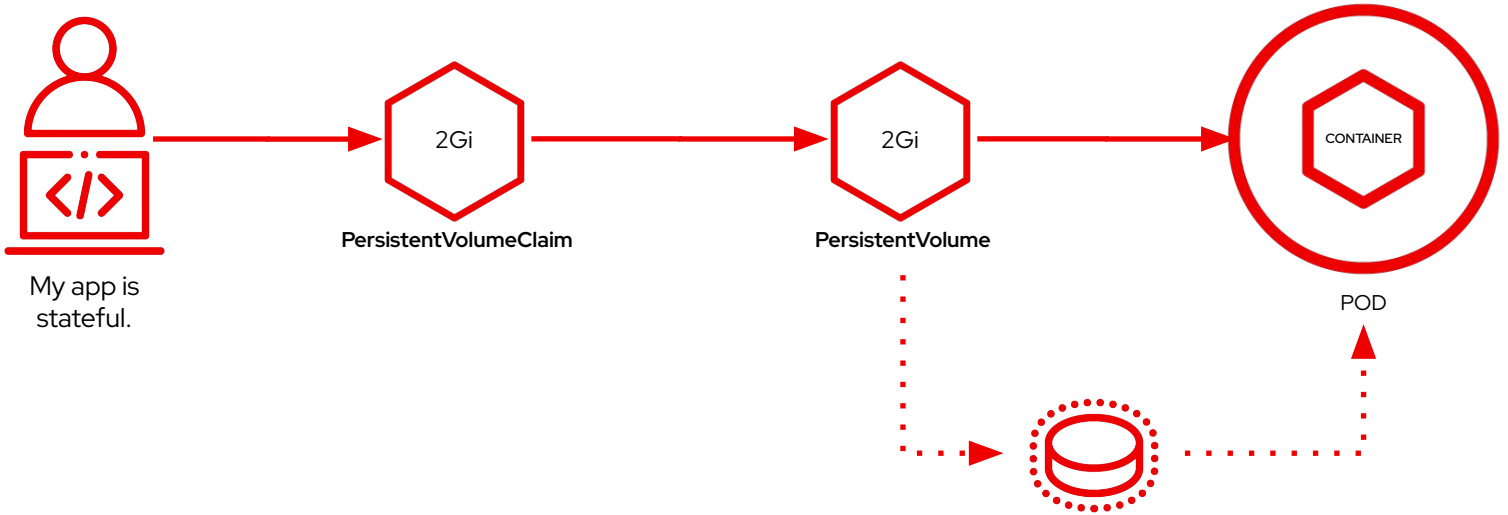


Routes make services accessible to clients outside the environment via real-world urls

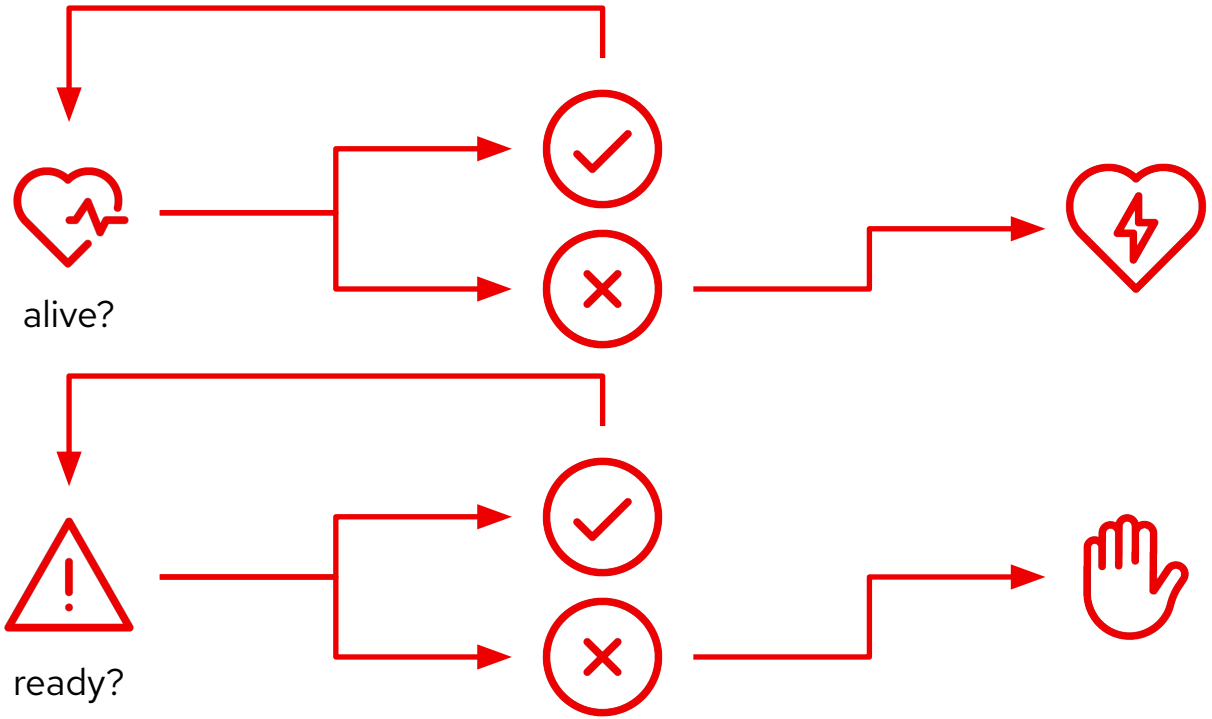


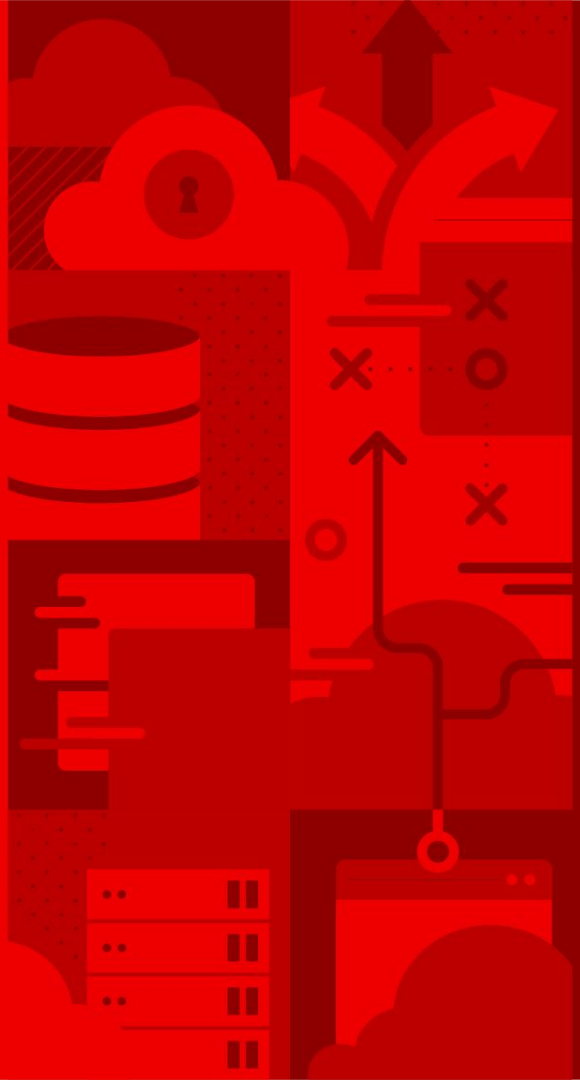
Red Hat

Persistent Volume and Claims



Liveness and Readiness





Final review

Chapter 9

Red Hat

Architecture Overview