LINUX CONTAINER Introduction

For NOVALUG - March 13th, 2021

Peter Larsen
Staff Domain Architect / OpenShift
Red Hat

Some slides copied/borrowed from Scott McCarthy with appreciation for his great presentation skills

redhat.

AGENDA

Introduction - Linux Containers

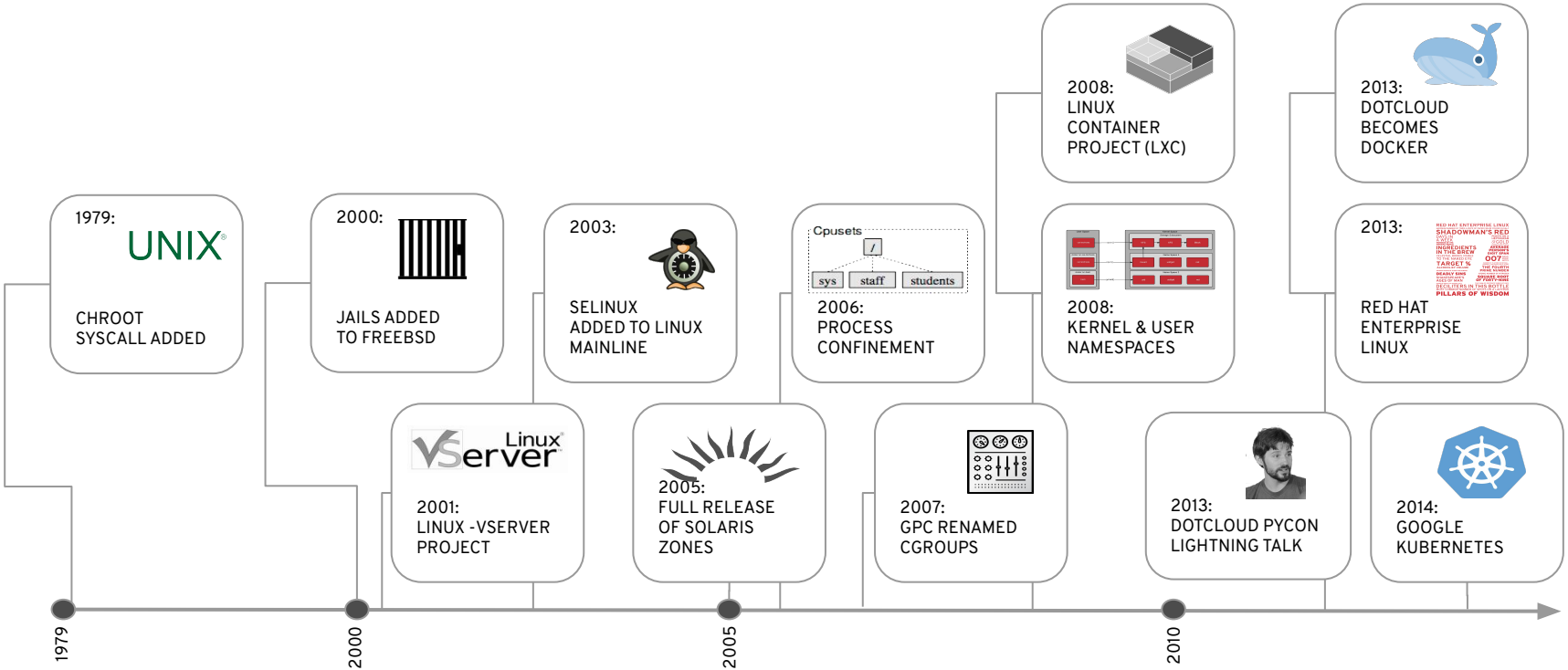
- What is a Linux Container / why use them / how to use them
- Using podman/buildah/skopeo on Fedora SilverBlue
- Container demonstrations

Homework! https://learn.openshift.com/subsystems

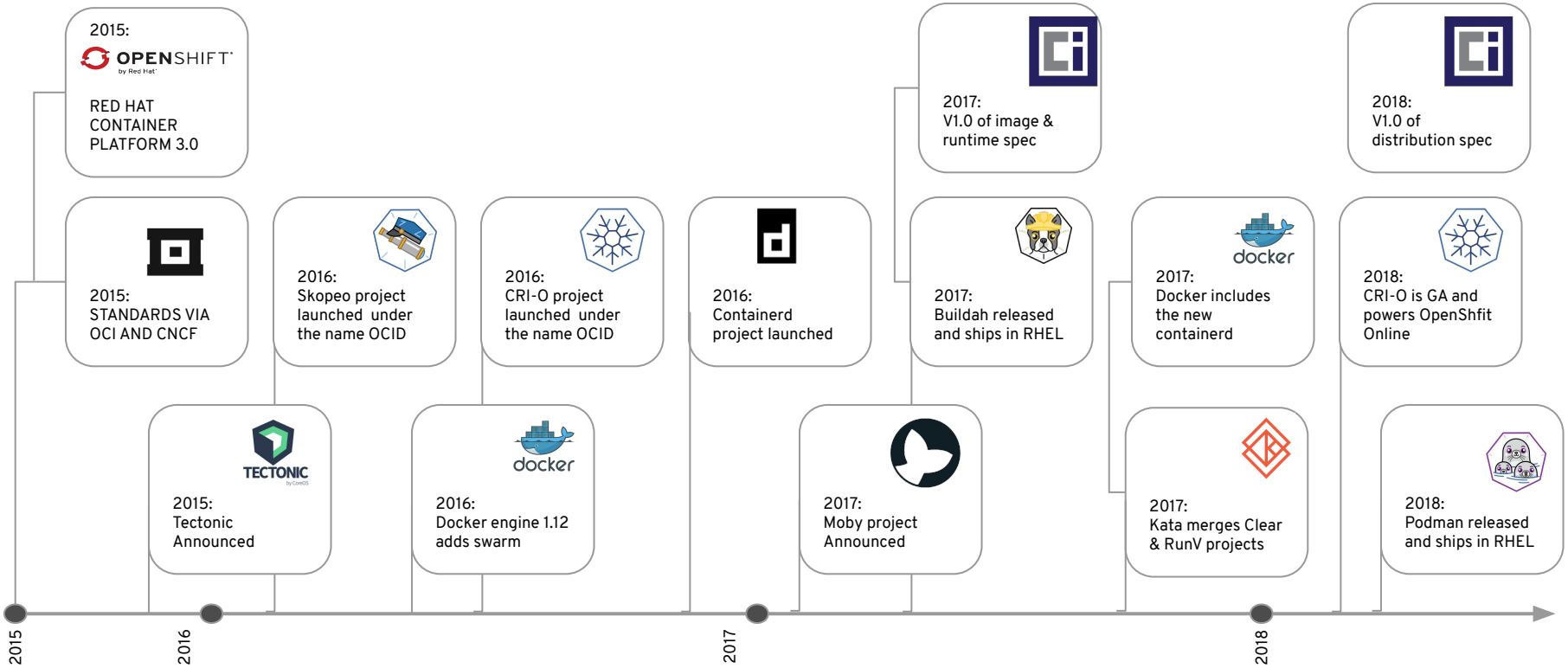
- You can find a completely hosted solution called Katacoda for your own experiments:
  - All you need is a web browser and Internet access
  - Instructions, code repositories, and terminal will be provided to a real, working virtual machine
  - All code is clickable, all you have to do is click on it and it will paste into the terminal
  - The environment can be reset at any time by refreshing (very nice)
  - Don't be intimidated by bash examples, there is a lot of gymnastics to make sure the lab can be run just by clicking. Feel free to ask me about bash stuff.


First a bit of HISTORY
Containers are not a new thing!

THE HISTORY OF CONTAINERS



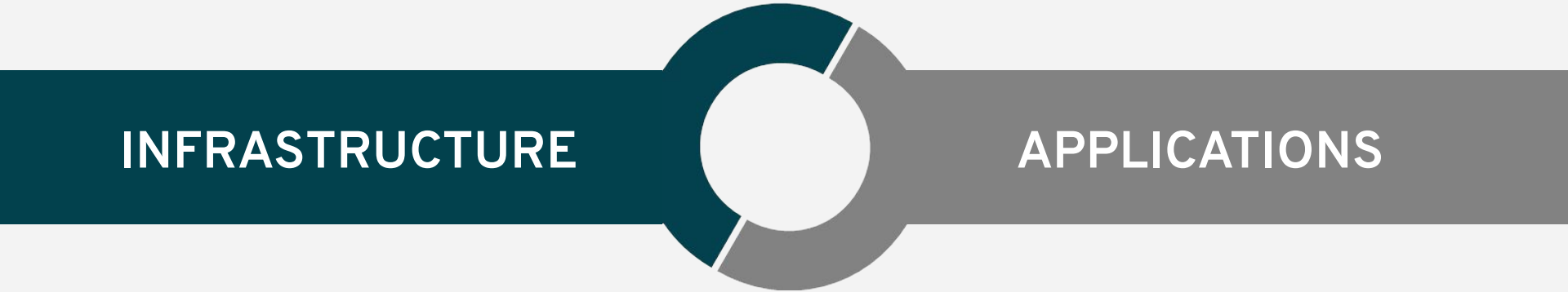
CONTAINER INNOVATION IS NOT FINISHED




INTRODUCTION
What is a container?
redhat.

WHAT ARE CONTAINERS?

It Depends Who You Ask

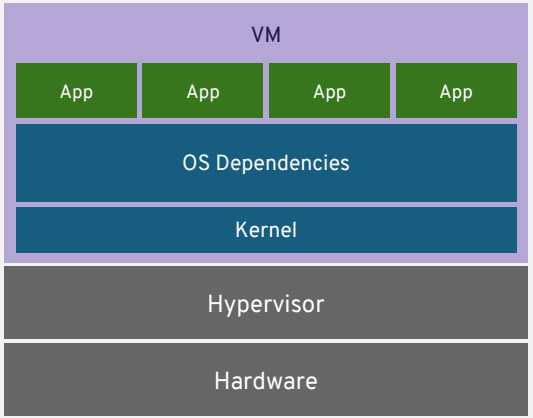


- Application processes on a shared kernel
- Simpler, lighter, and denser than VMs
- Portable across different environments

- Package apps with all dependencies
- Deploy to any environment in seconds
- Easily accessed and shared

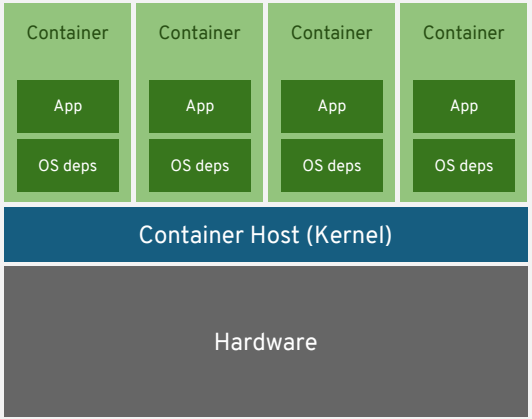
VIRTUAL MACHINES AND CONTAINERS

VIRTUAL MACHINES



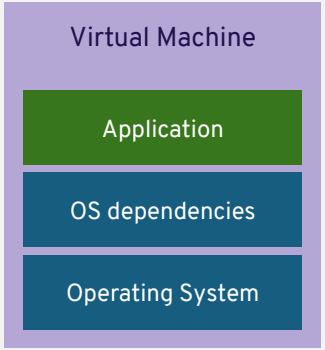
VM virtualizes the hardware

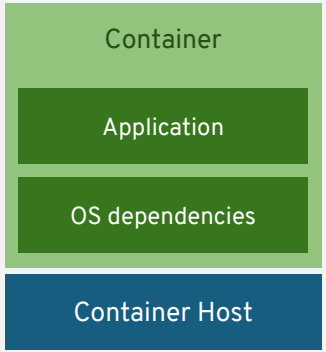
CONTAINERS



Container virtualizes the process

VIRTUAL MACHINES AND CONTAINERS

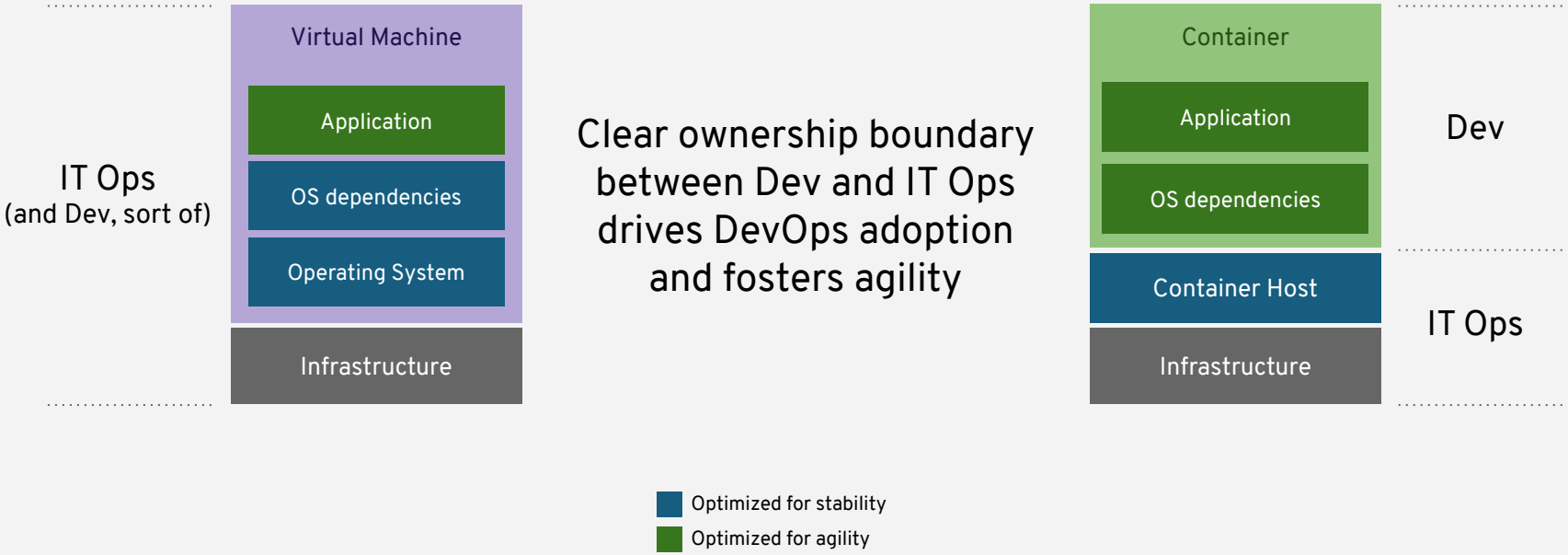




- ✚ VM Isolation
- ▬ Complete OS
- ▬ Static Compute
- ▬ Static Memory
- ▬ High Resource Usage

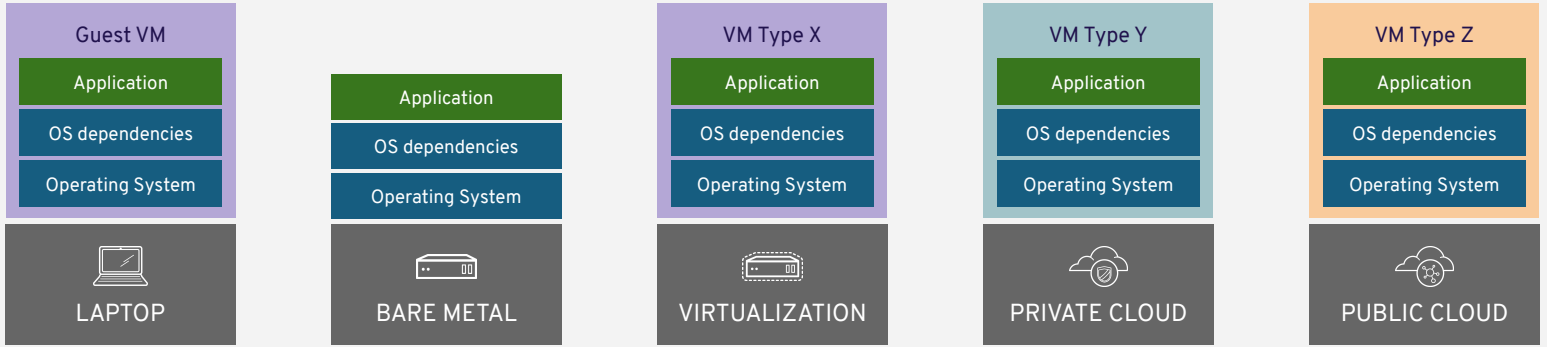
- ✚ Container Isolation
- ✚ Shared Kernel
- ✚ Burstable Compute
- ✚ Burstable Memory
- ✚ Low Resource Usage

VIRTUAL MACHINES AND CONTAINERS



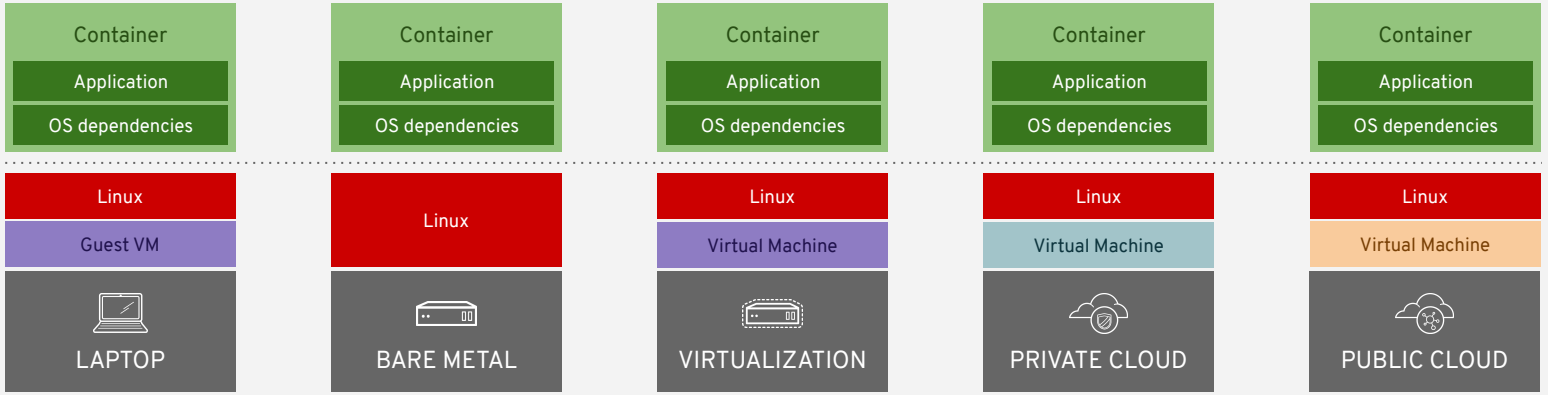
APPLICATION PORTABILITY WITH VM

Virtual machines are NOT portable across hypervisor and do NOT provide portable packaging for applications

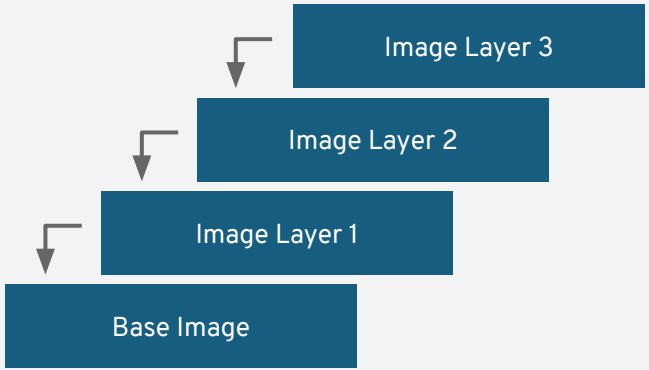


APPLICATION PORTABILITY WITH CONTAINERS

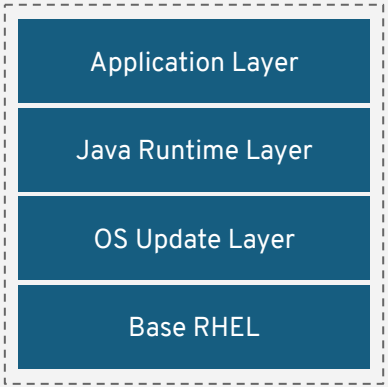
Containers + Container Host = Guaranteed Portability
Across Any Infrastructure



RAPID SECURITY PATCHING USING CONTAINER IMAGE LAYERING

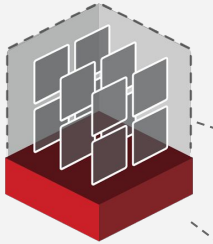


Container Image Layers



Example Container Image

DIGITAL WORKLOADS ARE MOVING TO CONTAINERS



LIFT & SHIFT

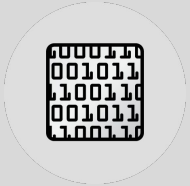
Migrate existing applications into more efficient container environments

MICROSERVICES

Better manage scalability and fast-moving application development cycles

MOBILE

Meet user demand, give them the ability to perform common tasks



ANALYTICS

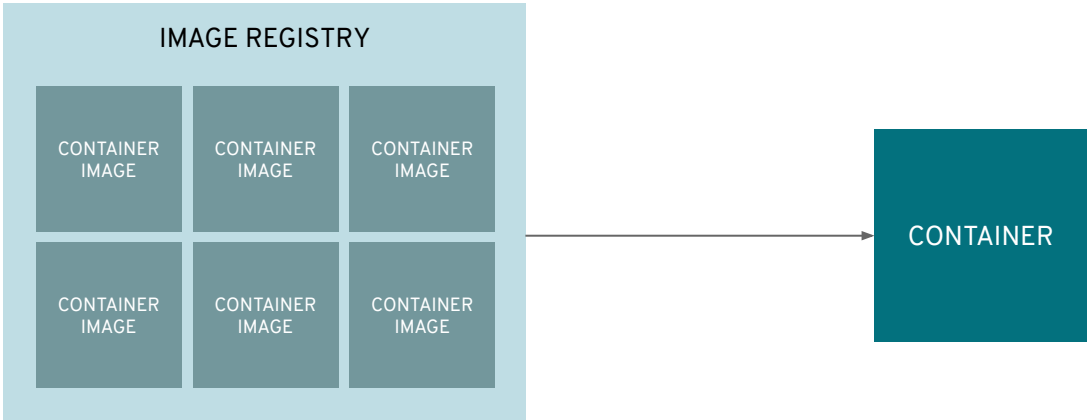
Move faster & find time for innovation, aligned to business needs

redhat

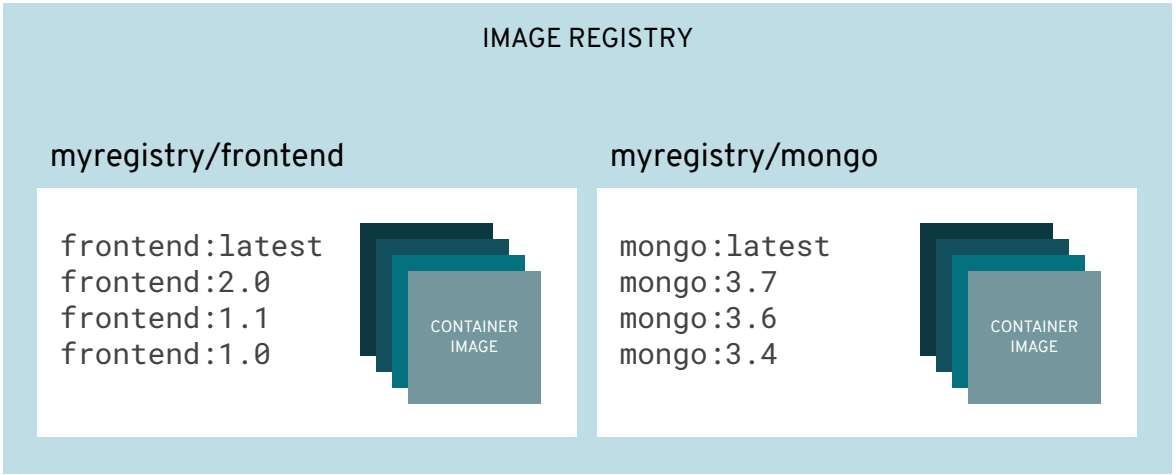
A container is the smallest compute unit

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



“Podman” and gang

We’ll be using podman for most of this talk (more details in a bit). Podman can easily be installed on all major distributions:

- https://podman.io/getting-started/installation

Notice the “Fedora SilverBlue” mention: No need to install! We’ll be using SilverBlue for this talk!

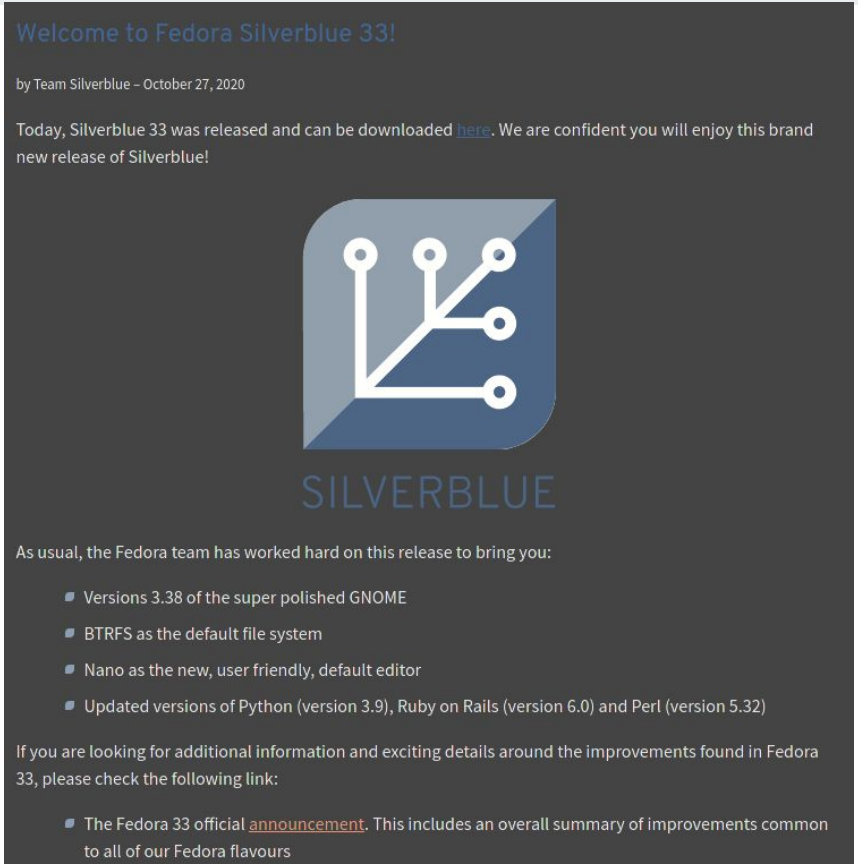
Why?

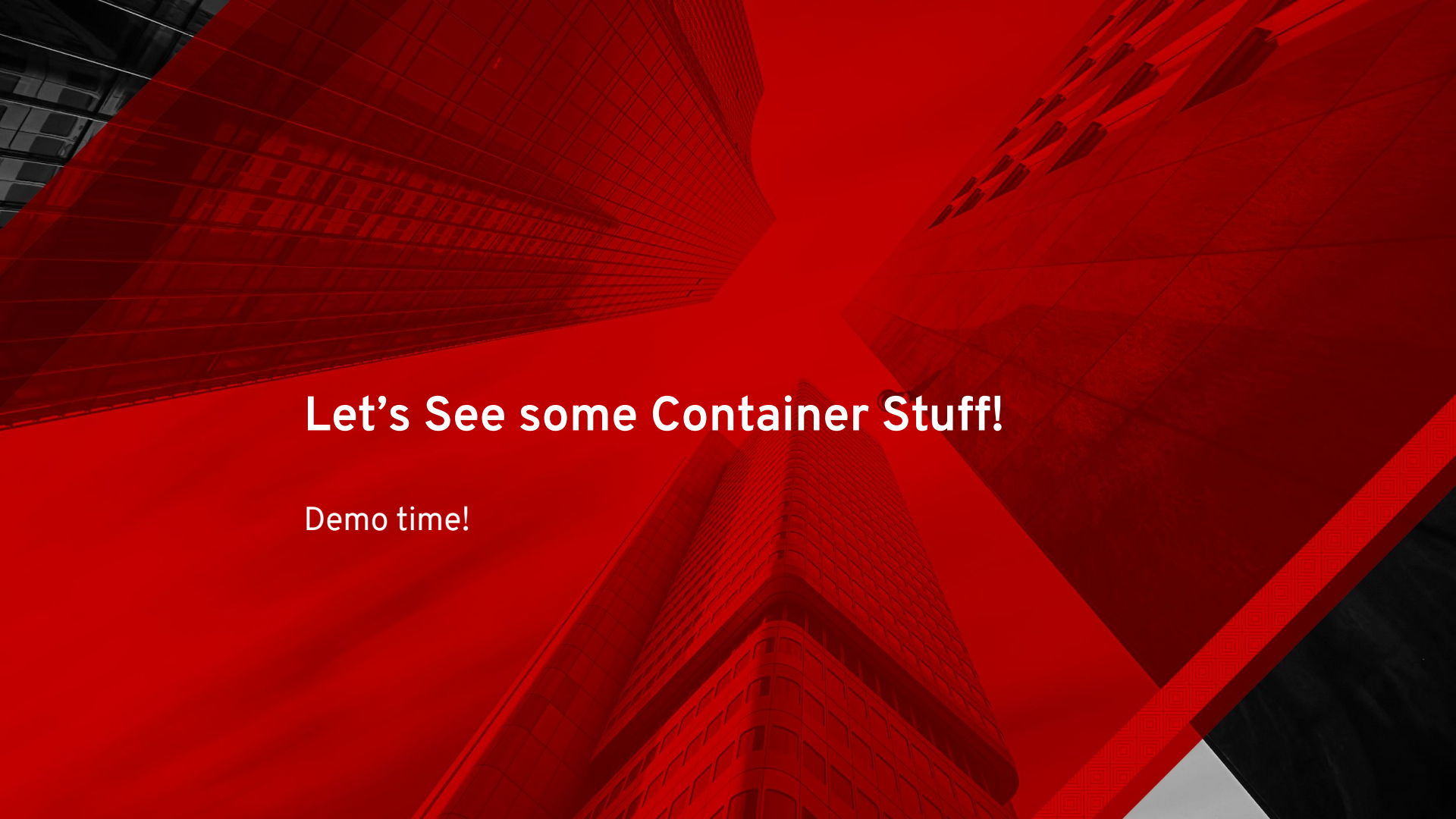
- “Podman” does not require root access! No daemon required.

What-ever we write starting with “podman” can be written using “docker” instead if you use docker!

Fedora SilverBlue

- https://silverblue.fedoraproject.org
- Container OS - ostree based
- Built to run container workloads
- (and flatpaks)





Let’s See some Container Stuff!

Demo time!

AGENDA

Introduction - Linux Container Internals

Introduction
Four new tools in your toolbelt

Container Images
The new standard in software packaging

Container Hosts
Container runtimes, engines, daemons

Container Registries
Sharing and collaboration

Container Orchestration
Distributed systems and containers

AGENDA

Advanced - Linux Container Internals

Container Standards
Understanding OCI, CRI, CNI, and more

Container Tools Ecosystem
Podman, Buildah, and Skopeo

Production Image Builds
Sharing and collaboration between specialists

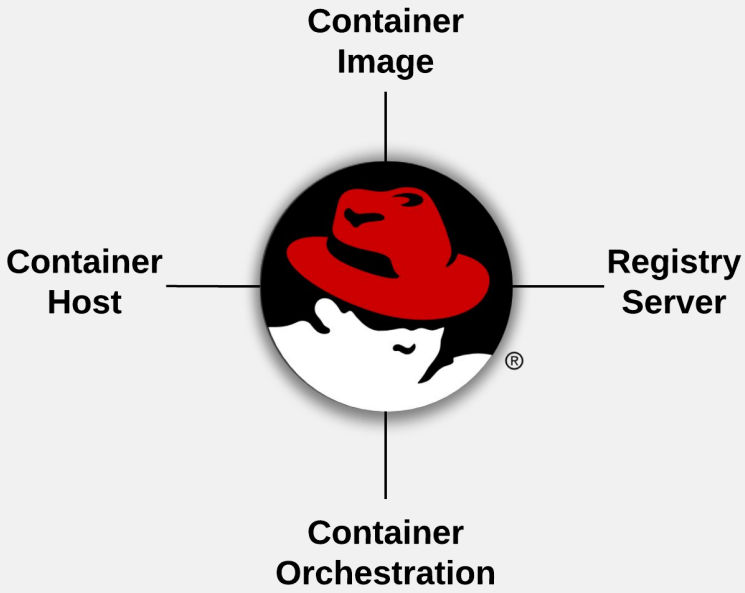
Intermediate Architecture
Production environments

Advanced Architecture
Building in resilience

Container History
Context for where we are today

Production-Ready Containers

What are the building blocks you need to think about?



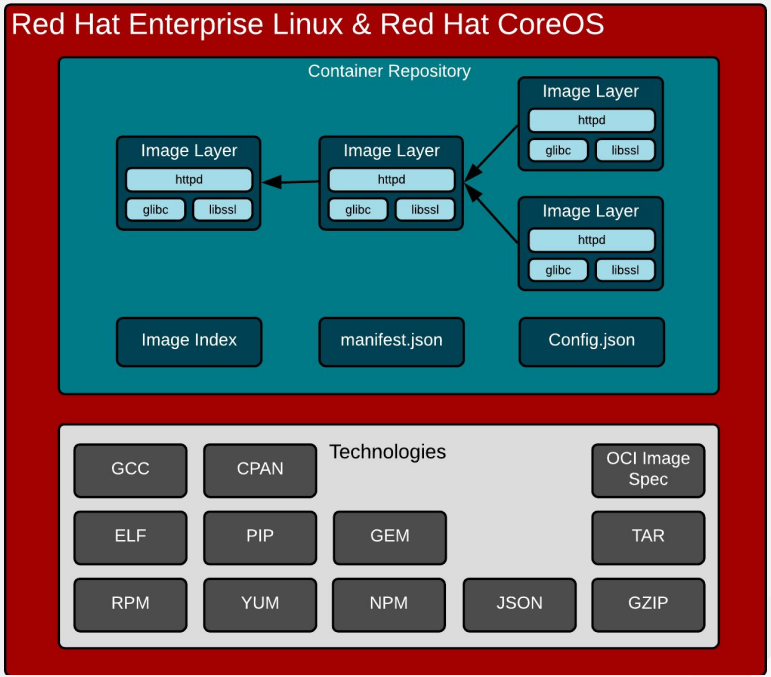

CONTAINER IMAGES
redhat.

CONTAINER IMAGE

Open source code/libraries, in a Linux distribution, in a tarball

Even base images are made up of layers:

- Libraries (glibc, libssl)
- Binaries (httpd)
- Packages (rpms)
- Dependency Management (yum)
- Repositories (rhel7)
- Image Layer & Tags (rhel7:7.5-404)
- At scale, across teams of developers and CI/CD systems, consider all of the necessary technology

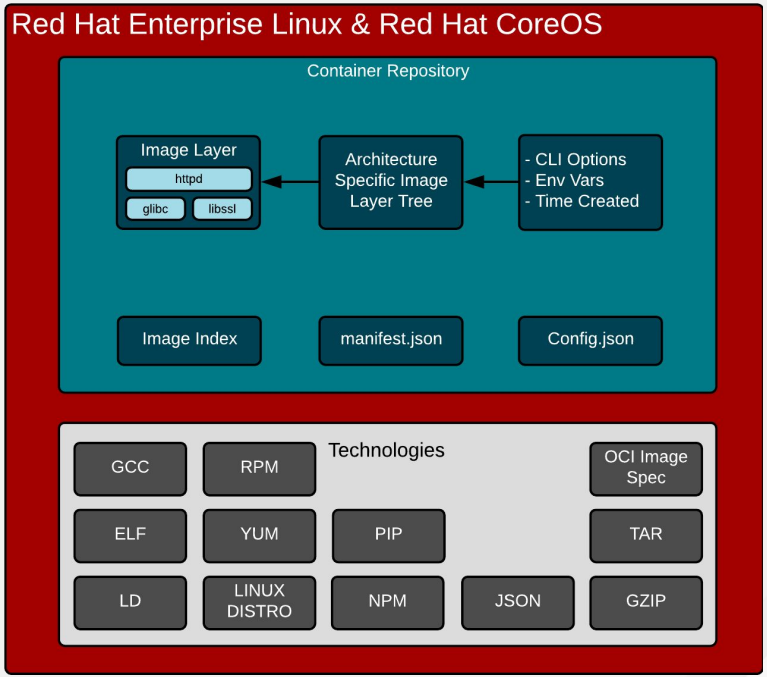


CONTAINER IMAGE PARTS

Governed by the OCI image specification standard

Lots of payload media types:

- Image Index/Manifest.json - provide index of image layers
- Image layers provide change sets - adds/deletes of files
- Config.json provides command line options, environment variables, time created, and much more
- Not actually single images, really repositories of image layers

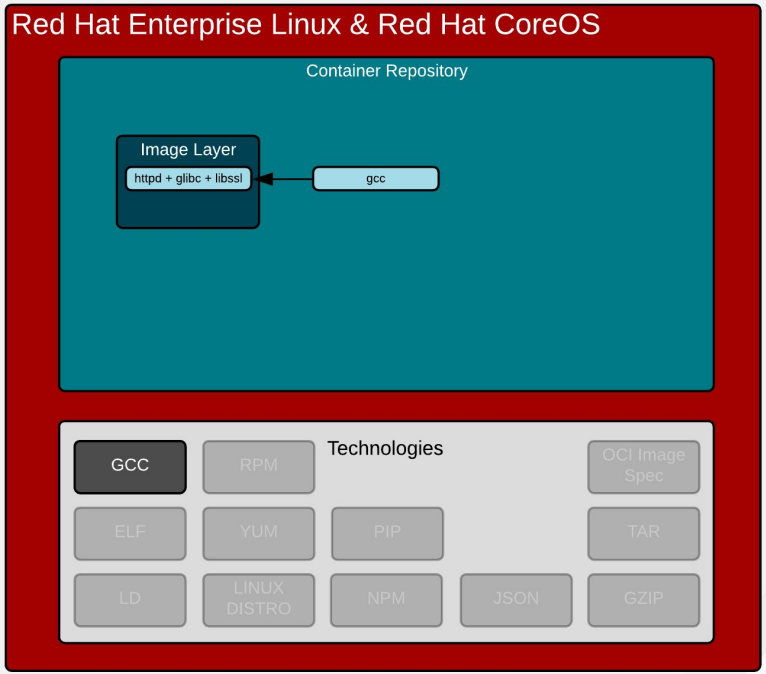


IT ALL STARTS WITH COMPILING

Statically linking everything into the binary

Starting with the basics:

- Programs rely on libraries
- Especially things like SSL - difficult to reimplement in for example PHP
- Math libraries are also common
- Libraries can be compiled into binaries - called static linking
- Example: C code + glibc + gcc = program

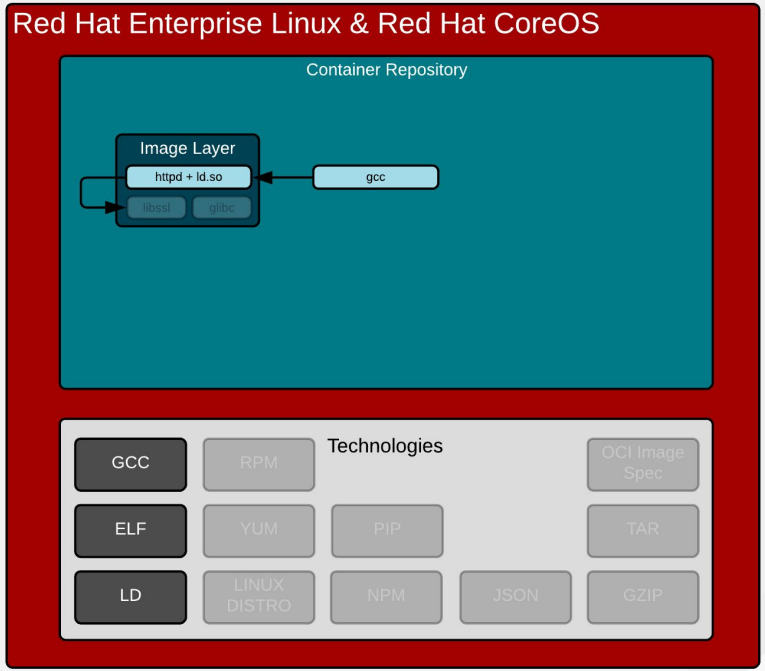


LEADS TO DEPENDENCIES

Dynamically linking libraries into the binary

Getting more advanced:

- This is convenient because programs can now share libraries
- Requires a dynamic linker
- Requires the kernel to understand where to find this linker at runtime
- Not terribly different than interpreters (hence the operating system is called an interpretive layer)

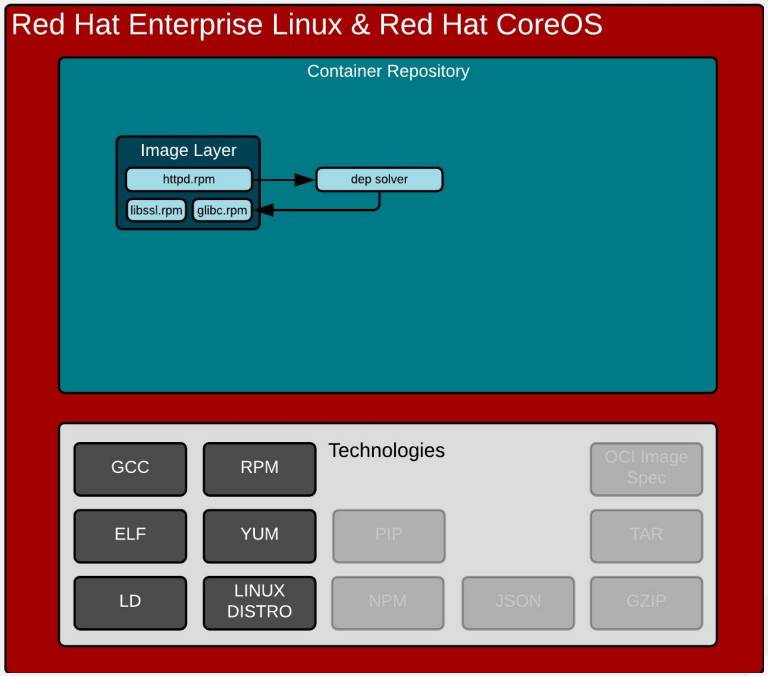


PACKAGING & DEPENDENCIES

RPM and Yum were invented a long time ago

Dependencies need resolvers:

- Humans have to create the dependency tree when packaging
- Computers have to resolve the dependency tree at install time (container image build)
- This is essentially what a Linux distribution does sans the installer (container image)

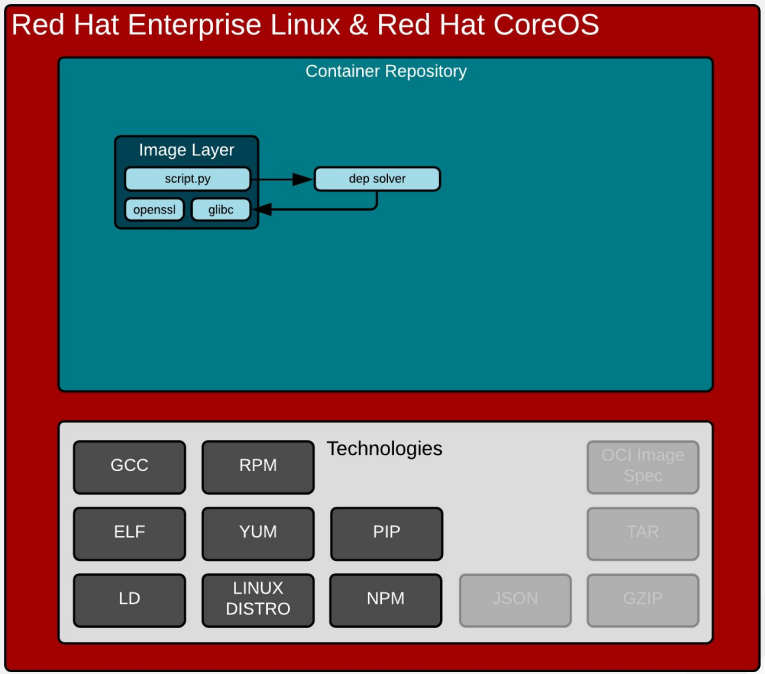


PACKAGING & DEPENDENCIES

Interpreters have to handle the same problems

Dependencies need resolvers:

- Humans have to create the dependency tree when packaging
- Computers have to resolve the dependency tree at install time (container image build)
- Python, Ruby, Node.js, and most other interpreted languages rely on C libraries for difficult tasks (ex. SSL)

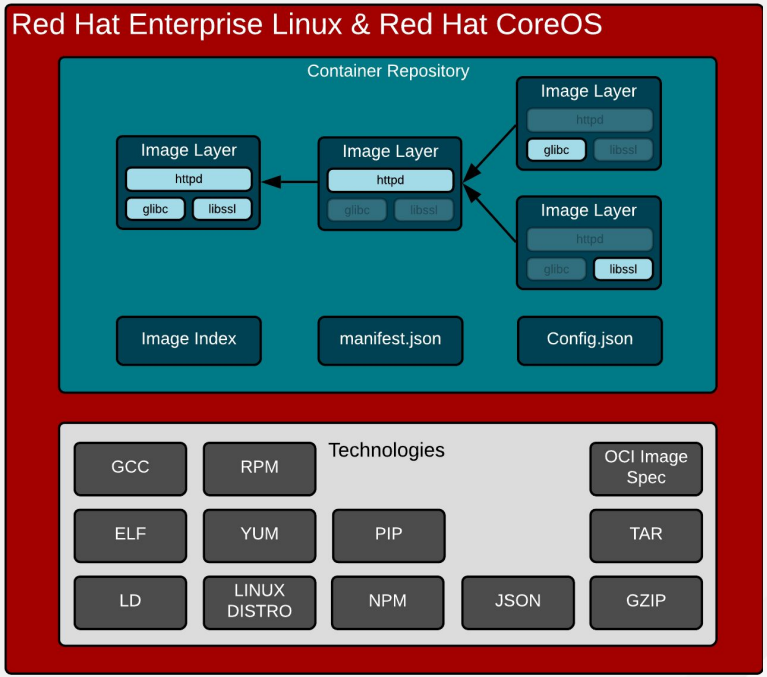


LAYERS ARE CHANGE SETS

Each layer has adds/deletes

Each image layer is a permutation in time:

- Different files can be added, updated or deleted with each change set
- Still relies on package management for dependency resolution
- Still relies on dynamic linking at runtime

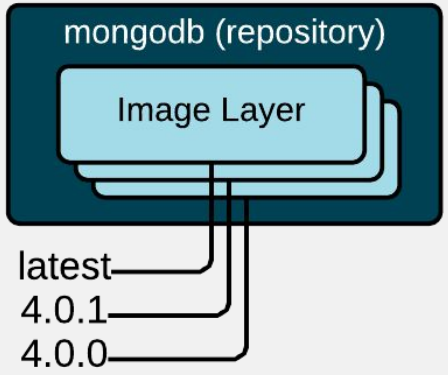


LAYERS ARE CHANGE SETS

Some layers are given a human readable name

Each image layer is a permutation in time:

- Different files can be added, updated or deleted with each change set
- Still relies on package management for dependency resolution
- Still relies on dynamic linking at runtime



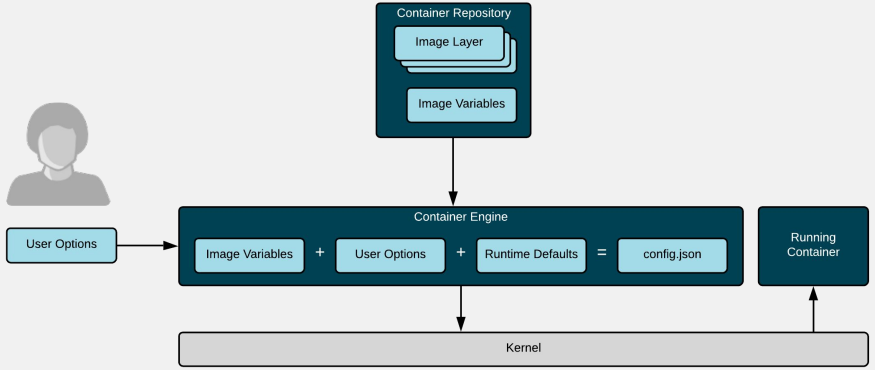
Layers and Tags

CONTAINER IMAGES & USER OPTIONS

Come with default binaries to start, environment variables, etc

Each image layer is a permutation in time:

- Different files can be added, updated or deleted with each change set
- Still relies on package management for dependency resolution
- Still relies on dynamic linking at runtime

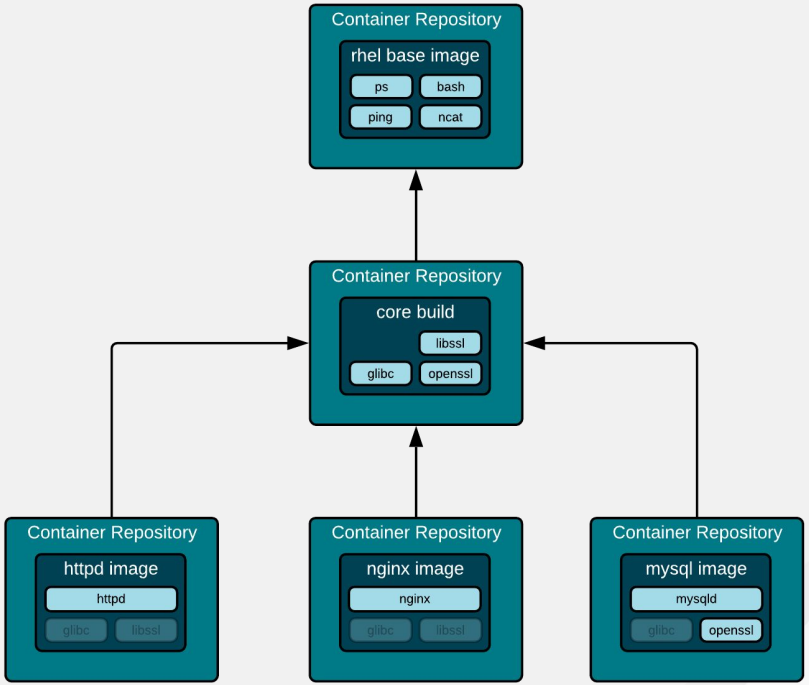


INTER REPOSITORY DEPENDENCIES

Think through this problem as well

You have to build this dependency tree yourself:

- DRY - Do not Repeat Yourself. Very similar to functions and coding
- OpenShift BuildConfigs and DeploymentConfigs can help
- Letting every development team embed their own libraries takes you back to the 90s

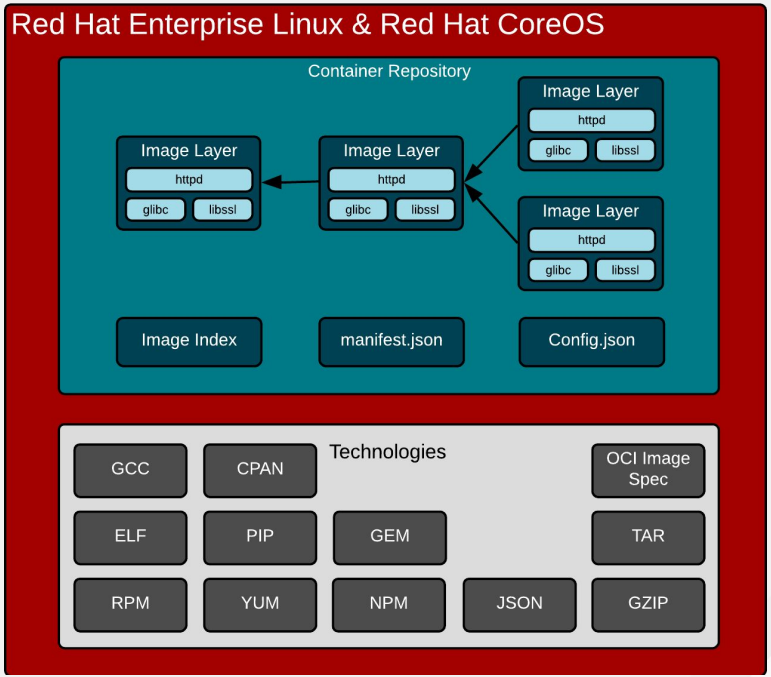


CONTAINER IMAGE

Open source code/libraries, in a Linux distribution, in a tarball

Even base images are made up of layers:

- Libraries (glibc, libssl)
- Binaries (httpd)
- Packages (rpms)
- Dependency Management (yum)
- Repositories (rhel7)
- Image Layer & Tags (rhel7:7.5-404)
- At scale, across teams of developers and CI/CD systems, consider all of the necessary technology



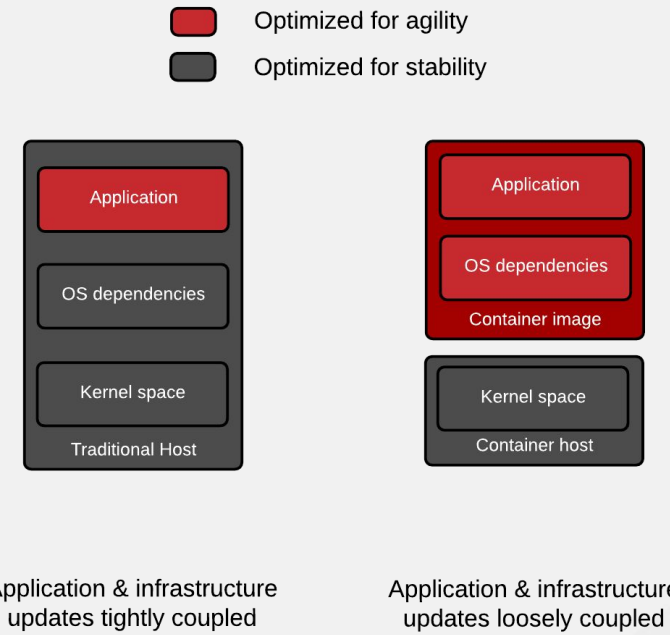

CONTAINER REGISTRIES

REGISTRY SERVERS

Better than virtual appliance market places :-)

Defines a standard way to:

- Find images
- Run images
- Build new images
- Share images
- Pull images
- Introspect images
- Shell into running container
- Etc, etc, etc

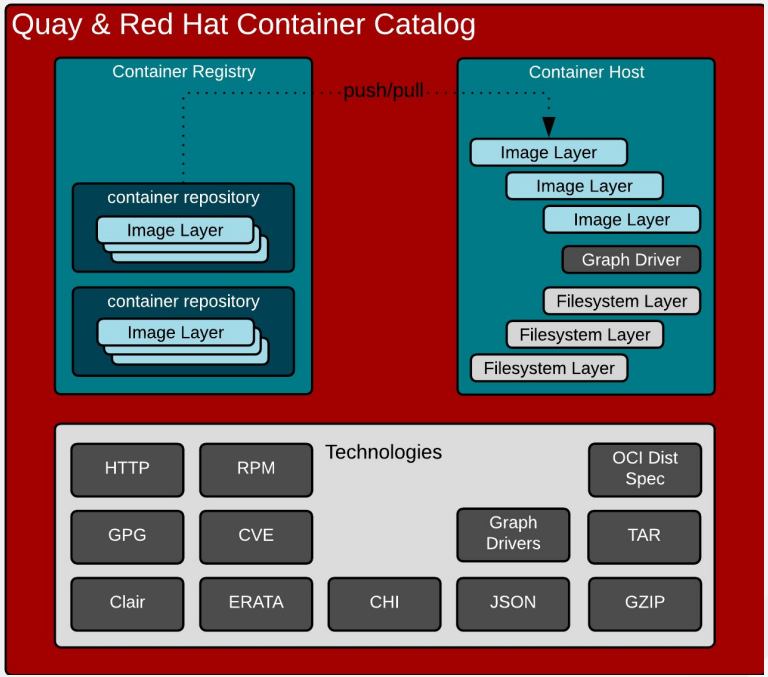


CONTAINER REGISTRY & STORAGE

Mapping image layers

Covering push, pull, and registry:

- Rest API (blobs, manifest, tags)
- Image Scanning (clair)
- CVE Tracking (errata)
- Scoring (Container Health Index)
- Graph Drivers (overlay2, dm)
- Responsible for maintaining chain of custody for secure images from registry to container host

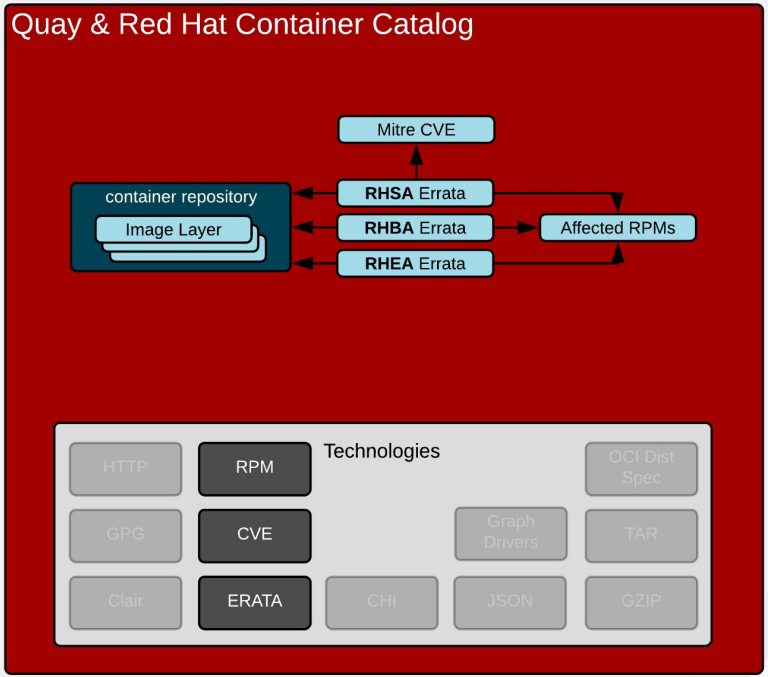


START WITH QUALITY REPOSITORIES

Repositories depend on good packages

Determining the quality of repository requires meta data:

- Errata is simple to explain, hard to build
  - Security Fixes
  - Bug Fixes
  - Enhancements
- Per container images layer (tag), often maps to multiple packages

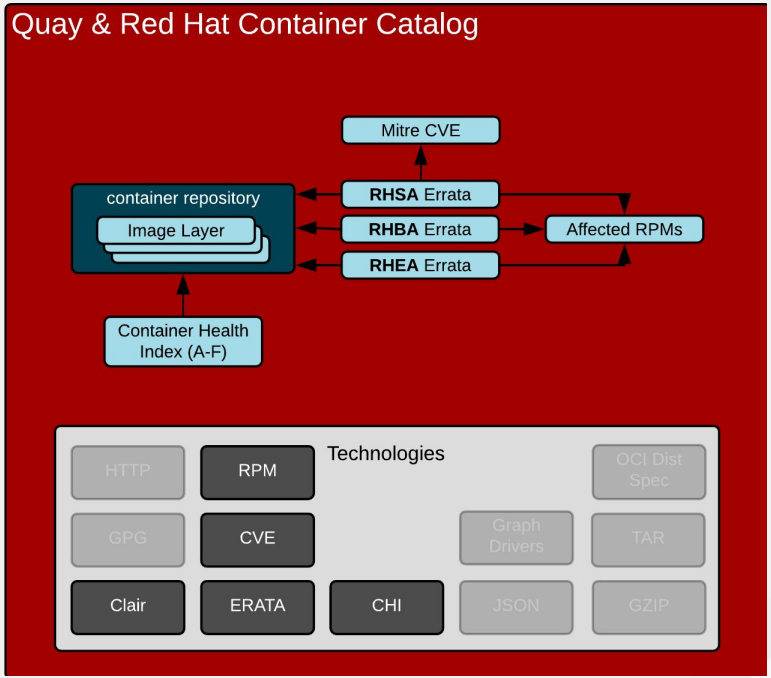


SCORING REPOSITORIES

Images age like cheese, not like wine

Based on severity and age of Security Errata:

- Trust is temporal
- Even good images go bad over time because the world changes around you

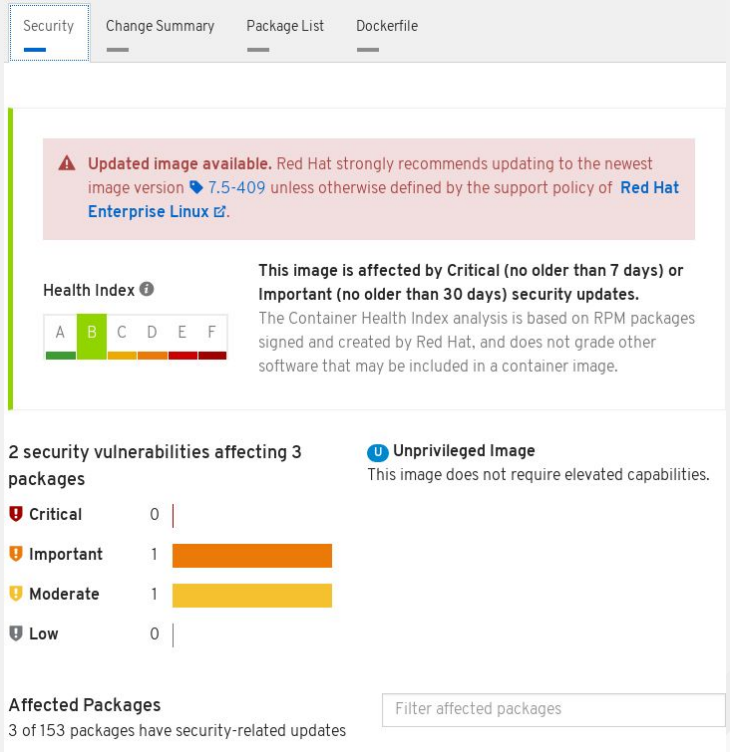


SCORING REPOSITORIES

Container Health Index

Based on severity and age of Security Errata:

- Trust is temporal
- Images must constantly be rebuilt to maintain score of “A”



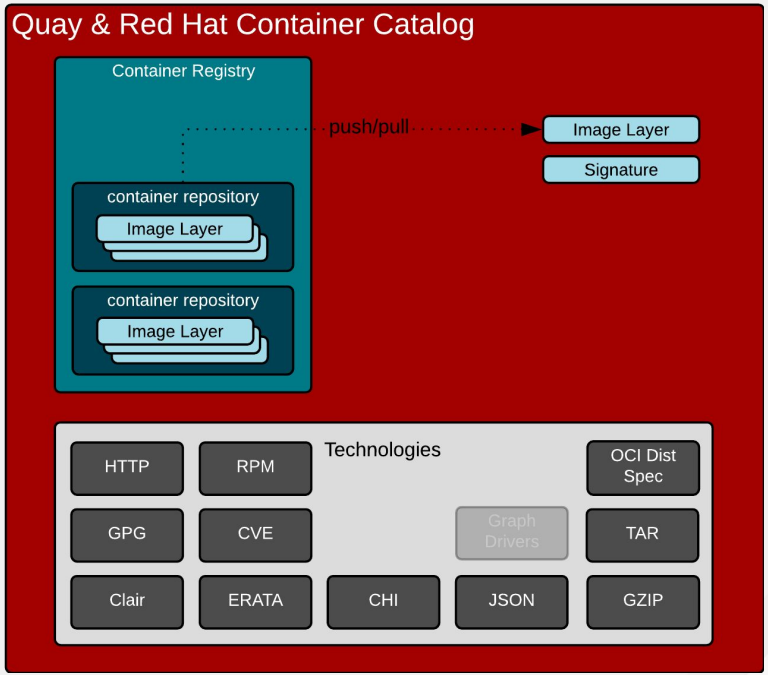
redhat.

PUSH, PULL & SIGNING

Signing and verification before/after transit

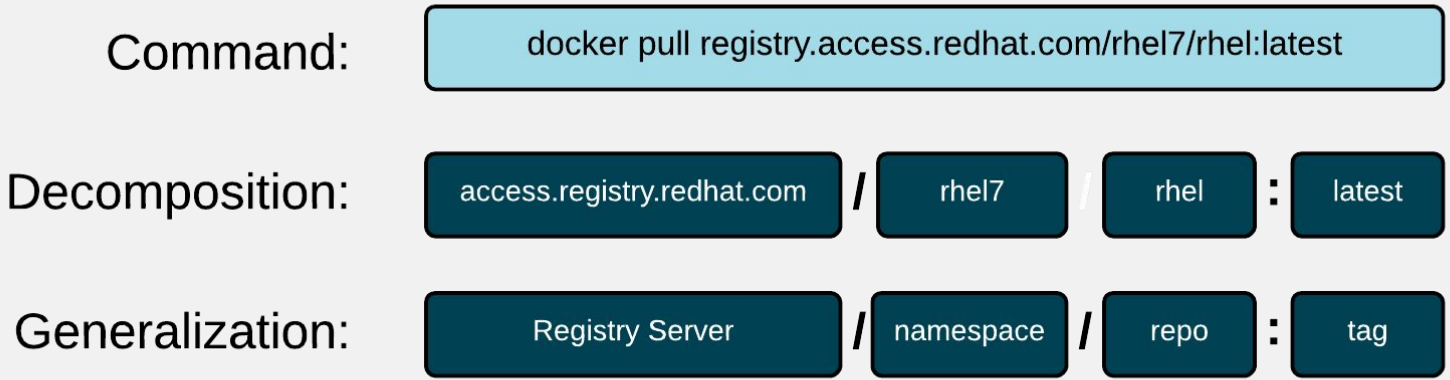
Registry has all of the image layers and can have the signatures as well:

- Download trusted thing
- Download from trusted source
- Neither is sufficient by itself



PUSH, PULL & SIGNING

Mapping image layers

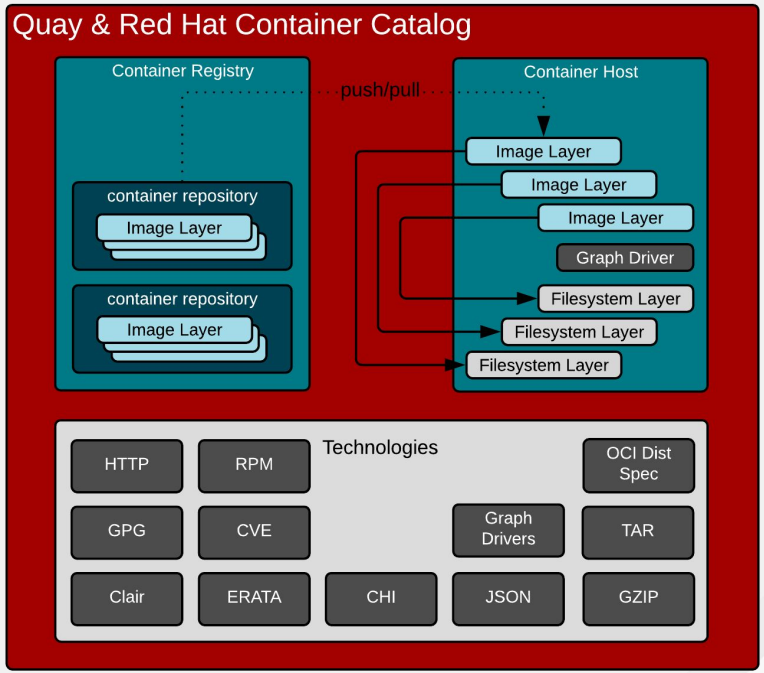


GRAPH DRIVERS

Mapping layers uses file system technology

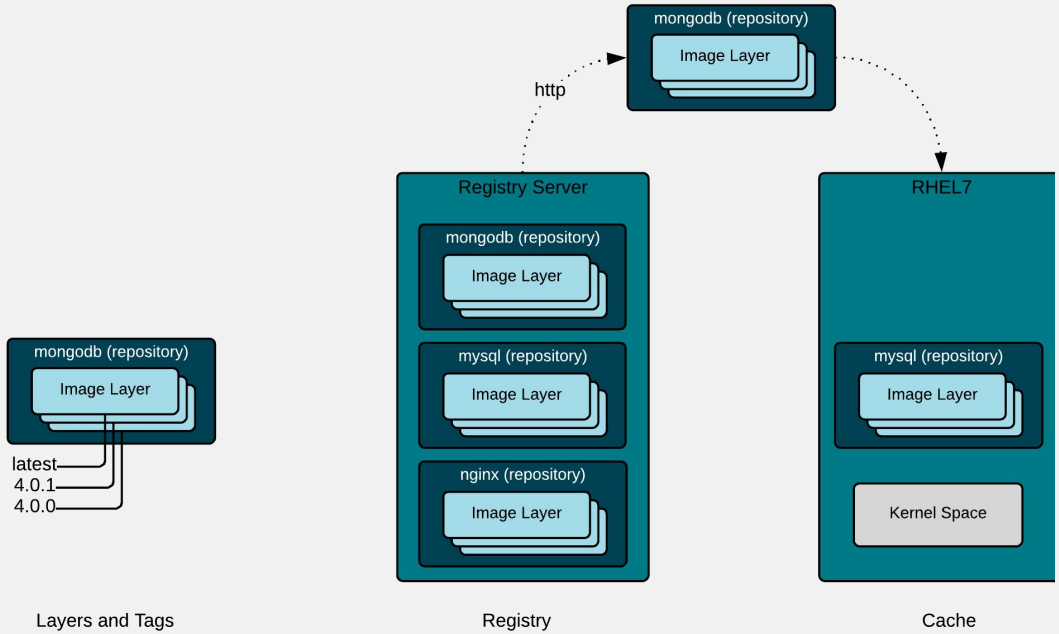
Local cache maps each layer to volume or filesystem layer:

- Overlay2 file system and container engine driver
- Device Mapper volumes and container engine driver



PUSH, PULL & SIGNING

Mapping image layers

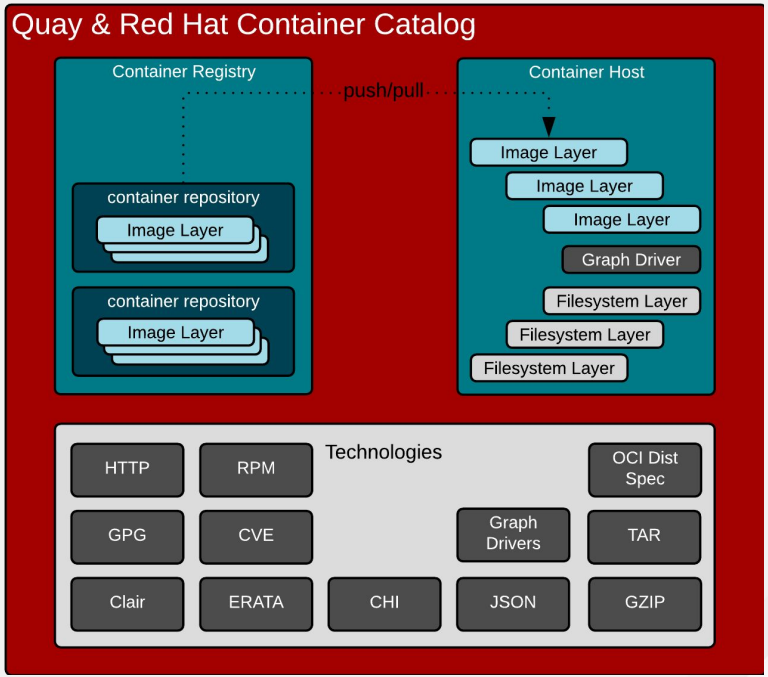


CONTAINER REGISTRY & STORAGE

Mapping image layers

Covering push, pull, and registry:

- Rest API (blobs, manifest, tags)
- Image Scanning (clair)
- CVE Tracking (errata)
- Scoring (Container Health Index)
- Graph Drivers (overlay2, dm)
- Responsible for maintaining chain of custody for secure images from registry to container host



DEMO TIME

How to use images/registries


CONTAINER HOSTS
redhat.

CONTAINER HOST BASICS

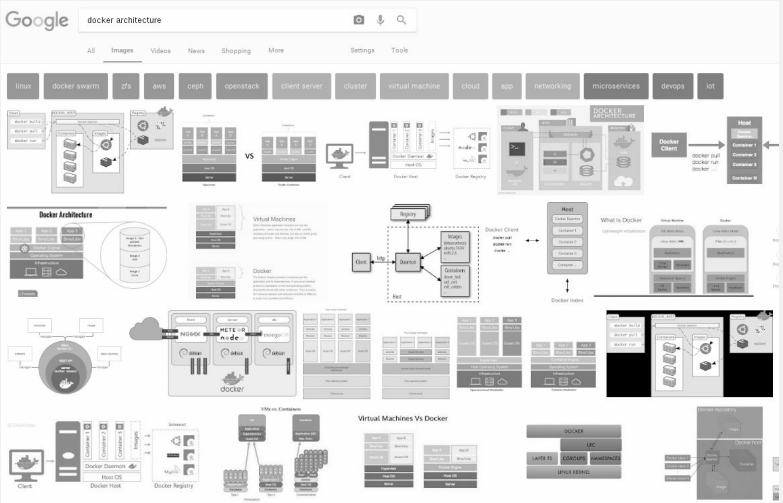
Container Engine, Runtime, and Kernel

CONTAINERS DON’T RUN ON DOCKER

The Internet is WRONG :-)

Important corrections

- Containers do not run ON docker. Containers are processes - they run on the Linux kernel. Containers are Linux processes (or Windows).
- The docker daemon is one of the many user space tools/libraries that talks to the kernel to set up containers

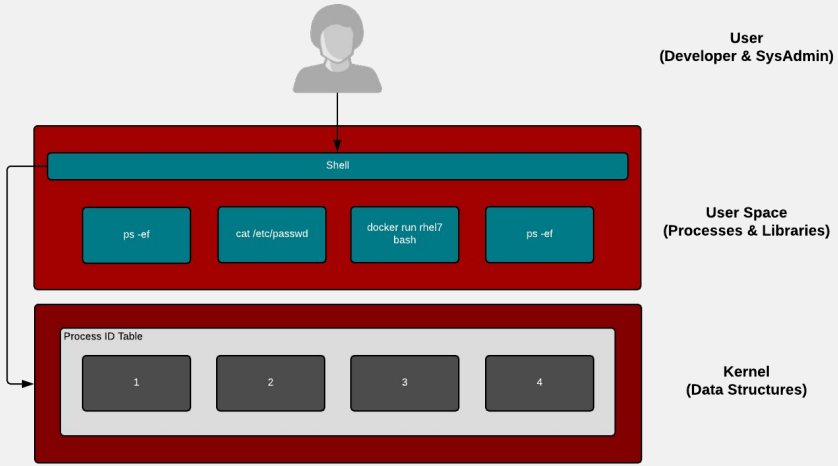


PROCESSES VS. CONTAINERS

Actually, there is no processes vs. containers in the kernel

User space and kernel work together

- There is only one process ID structure in the kernel
- There are multiple human and technical definitions for containers
- Container engines are one technical implementation which provides both a methodology and a definition for containers

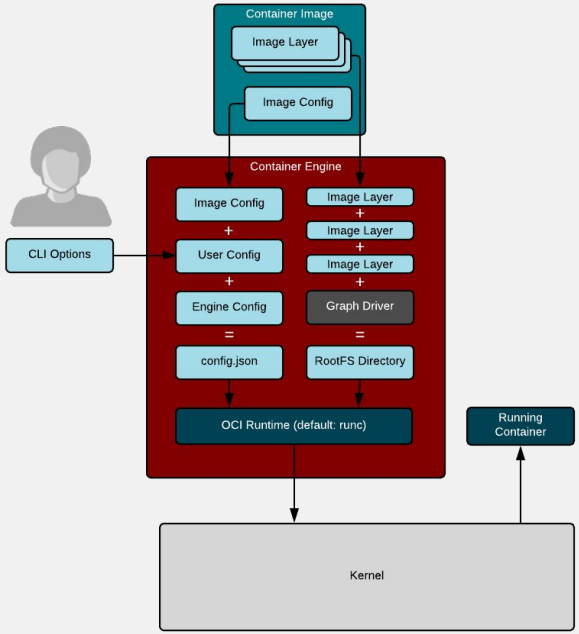


THE CONTAINER ENGINE IS BORN

This was a new concept introduced with Docker Engine and CLI

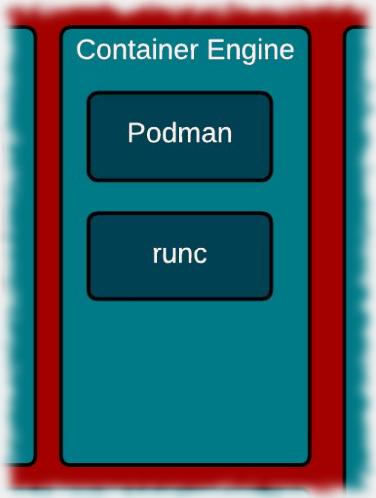
Think of the Docker Engine as a giant proof of concept - and it worked!

- Container images
- Registry Servers
- Ecosystem of pre-built images
- Container engine
- Container runtime (often confused)
- Container image builds
- API
- CLI
- A LOT of moving pieces

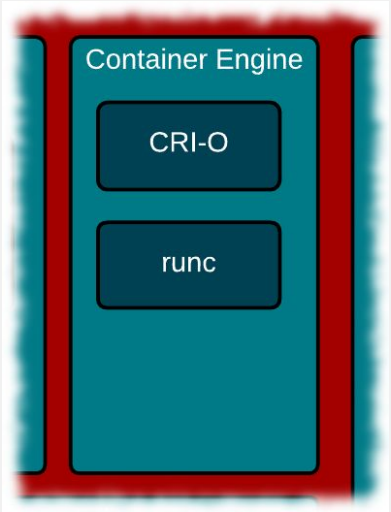


DIFFERENT ENGINES

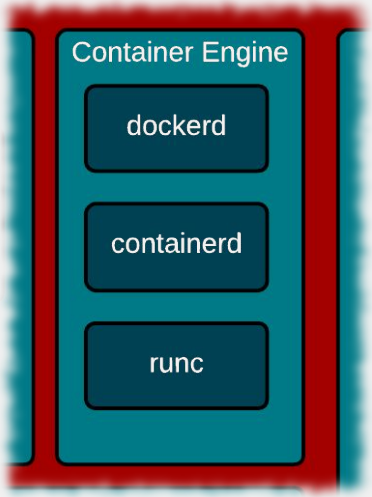
All of these container engines are OCI compliant



Podman



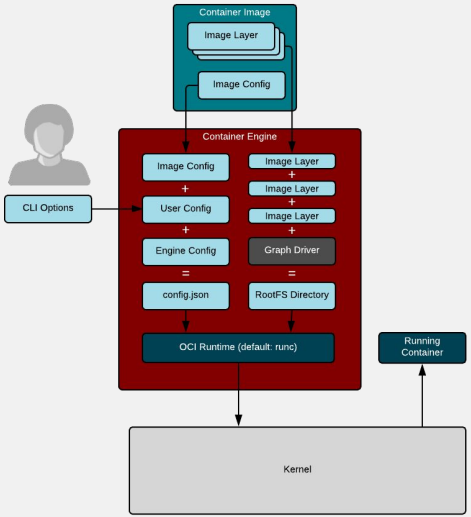
CRI-O



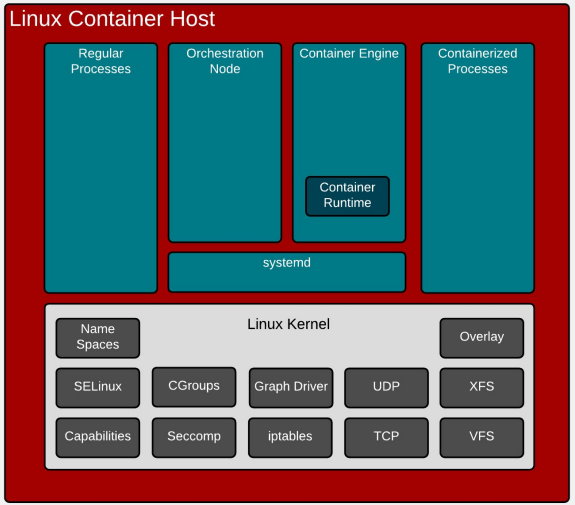
Docker

CONTAINER ENGINE VS. CONTAINER HOST

In reality the whole container host is the engine - like a Swiss watch



VS.

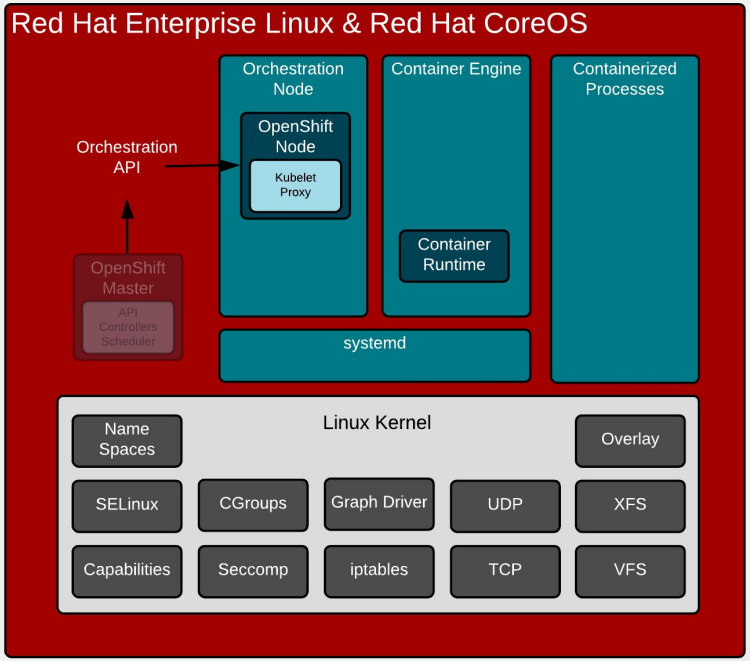


CONTAINER HOST

Released, patched, tested together

Tightly coupled communication through the kernel - all or nothing feature support:

- Operating System (kernel)
- Container Runtime (runc)
- Container Engine (Docker)
- Orchestration Node (Kubelet)
- Whole stack is responsible for running containers



Building and creating containers

The “Dockerfile”

- Traditional method to describe how to create a container
- Common keywords:
  - FROM
  - RUN
  - ENTRYPOINT
  - COPY
  - MAINTAINER
  - ENV
  - WORKDIR
  - EXPOSE

Dockerfile example

```
FROM nginx

ENV AUTHOR=Docker

WORKDIR /usr/share/nginx/html

COPY Hello_docker.html /usr/share/nginx/html

CMD cd /usr/share/nginx/html && sed -e s/Docker/"$AUTHOR"/ Hello_docker.html >
index.html ; nginx -g 'daemon off;'
```

Another Example

- Every RUN/COPY creates new image layer
- You must specify network port to expose
- You must specify a base image to start from or specify “scratch” and you must copy all artifacts in (from a tar ball from instance)
- Use your local directory to hold data you want inserted into the container image.

```
# our base image
FROM alpine:3.5

# Install python and pip
RUN apk add --update py2-pip

# upgrade pip
RUN pip install --upgrade pip

# install Python modules needed by the Python app
COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt

# copy files required for the app to run
COPY app.py /usr/src/app/
COPY templates/index.html /usr/src/app/templates/

# tell the port number the container should expose
EXPOSE 5000

# run the application
CMD ["python", "/usr/src/app/app.py"]
```

Introducing buildah

A more flexible way to build/manage containers

- Buildah allows you to interactively create a container image
- You can inspect the image, commit when you want and test things
- Buildah can use dockerfiles ! (buildah bud)
- Dockerfile commands are buildah commands:
  - Buildah copy
  - Buildah run
  - Buildah from
- Buildah config sets metadata like entrypoint, workingdir etc.
-

Buildah Example

- Easy debugging
- Easy inspection

```
# Create a container
container=$(buildah from fedora:28)

# Labels are part of the "buildah config" command
buildah config --label maintainer="Chris Collins <collins.christopher@gmail.com>'

# Grab the source code outside of the container
curl -sSL http://ftpmirror.gnu.org/hello/hello-2.10.tar.gz -o hello-2.10.tar.gz

buildah copy $container hello-2.10.tar.gz /tmp/hello-2.10.tar.gz

buildah run $container dnf install -y tar gzip gcc make
Buildah run $container dnf clean all
buildah run $container tar xvzf /tmp/hello-2.10.tar.gz -C /opt

# Workingdir is also a "buildah config" command
buildah config --workingdir /opt/hello-2.10 $container

buildah run $container ./configure
buildah run $container make
buildah run $container make install
buildah run $container hello -v

# Entrypoint, too, is a “buildah config” command
buildah config --entrypoint /usr/local/bin/hello $container

# Finally saves the running container to an image
buildah commit --format docker $container hello:latest
```

Buildah - the really cool stuff!

- Mountpoints!
  - Add your container image to your local file system to inspect the content!
  - Modify this local file system and commit - and the changes go straight to the container. Multiple commands result in a single image layer!
- Example
  - mountpoint=$(buildah mount ${container})
    sudo dnf install nginx --installroot $mountpoint
    chroot $mountpoint nginx -v
    nginx version:.....

Another way to make changes

- Containers are immutable
  - Every time a container image is run a new copy-on-write layer is created
  - When a container stops running and you're using the --rm parameter, this temporary layer is deleted. The container image is clean.
  - You can run a container without the --rm and use "podman commit" to write this temporary layer to a new version of the container image.
  - This includes temporary files and more - so be careful using it.
- Always assume your containers are immutable
- Data and configuration should be external to the container - in secrets or environment variables or a bind mount file system.

DEMO TIME

Let’s build some containers

Thanks for listening

Tune in next month for OpenShift
The Enterprise Kubernetes Platform