Container Optimization

© 2024 by Innovation In Software Corporation

Logistics

- Class Hours:
- Instructor will provide class start and end times.
- Breaks throughout class

- Telecommunication:
- Turn off or set electronic devices to vibrate
- Reading or attending to devices can be distracting to other students

© 2024 by Innovation In Software Corporation

Course Objectives

- Explain the benefits of using containers
- Understand Container images
- Optimize container images
- Container resource tuning
- Utilizing Open-Source tools

© 2024 by Innovation In Software Corporation

Instructor

Rod Davison

50 years experience

- Academia (math, linguistics, cognitive science)
- Artificial Intelligence R and D
- Software Development
- Data Analytics – Social Research
- Project Manager
- Quality and Testing
- Business Analysis
- Consulting and Training

© 2024 by Innovation In Software Corporation

Introductions

- Name
- Job Role
- Your experience with (scale 1 - 5)
  - Docker/Containers
- Expectations for the course (please be specific)

© 2024 by Innovation In Software Corporation

Data Center Evolution

© 2024 by Innovation In Software Corporation

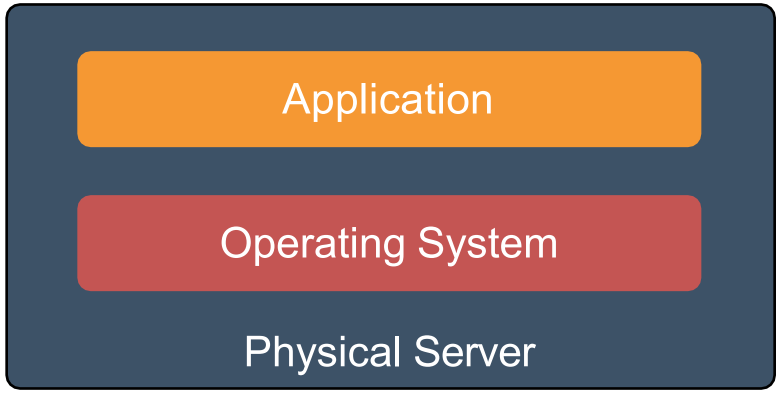
Monolithic

Monolithic

Virtualization

© 2024 by Innovation In Software Corporation

Monolithic Server Architecture

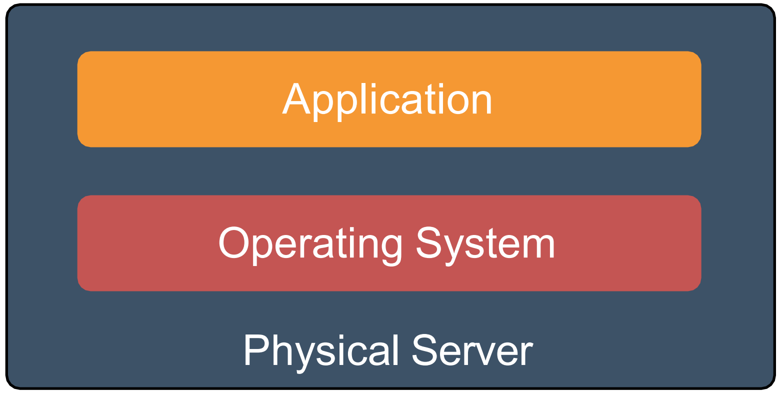


One physical server, one application

© 2024 by Innovation In Software Corporation

Monolithic Server Architecture

Monolithic → Virtualization



One physical server, one application

Problems

- Slow deployment times
- Cost
- Wasted resources
- Difficult to scale
- Difficult to migrate

© 2024 by Innovation In Software Corporation

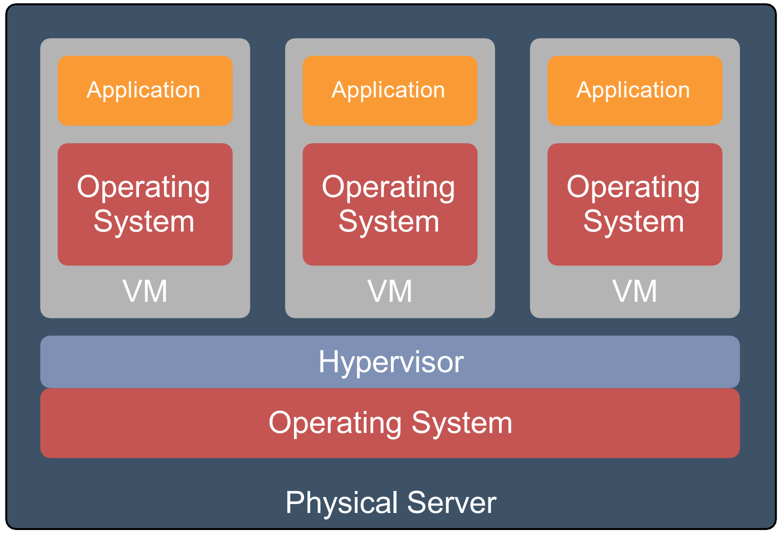
Virtualized

Monolithic

Virtualization

© 2024 by Innovation In Software Corporation

Virtualized Infrastructure



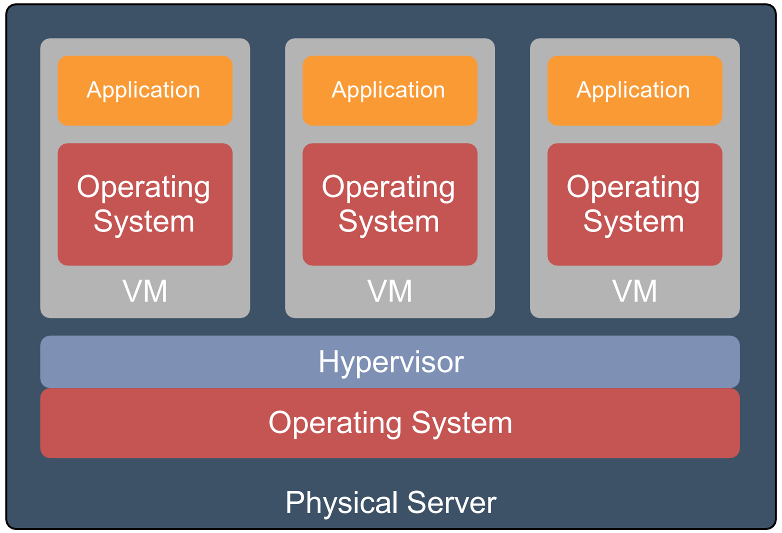
One physical server, multiple applications

© 2024 by Innovation In Software Corporation

Discussion

What are some of the advantages and disadvantages of Virtual Machines?

Virtualized Infrastructure - Advantages



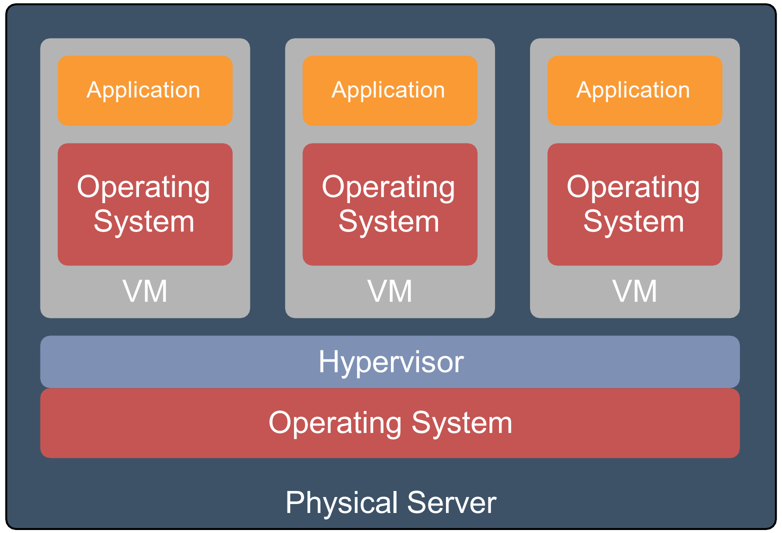
One physical server, multiple applications

Advantages

- Better resource pooling
- Easier to Scale
- Enables Cloud/IaaS
  - Rapid elasticity
  - Pay as you go model

© 2024 by Innovation In Software Corporation

Virtualized Infrastructure - Limitations



One physical server, multiple applications

Limitations

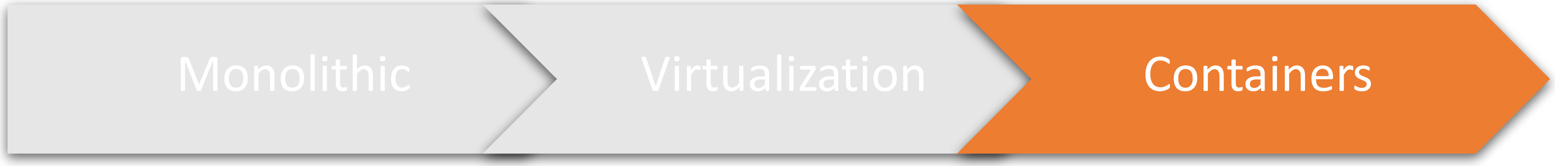
- Each VM requires:
  - CPU allocation
  - Storage
  - RAM
  - Guest Operating System
- More VMs, more wasted resources
- Application portability not guaranteed

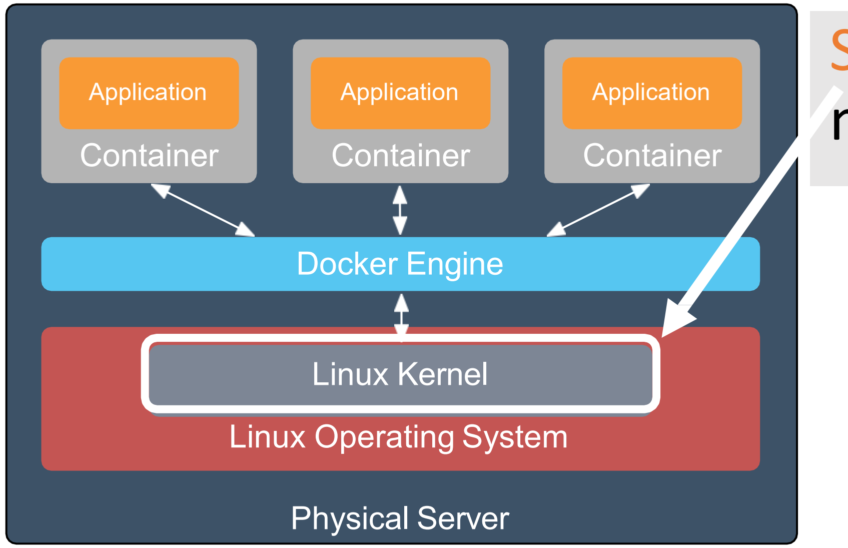
© 2024 by Innovation In Software Corporation

Containers

© 2024 by Innovation In Software Corporation

Containers



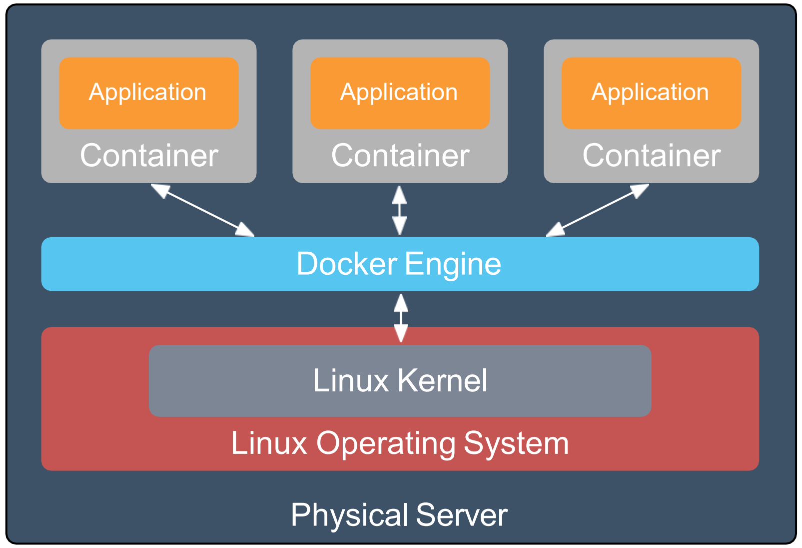


Shared kernel on the host to run multiple guest applications

© 2024 by Innovation In Software Corporation

Containers - Advantages

Monolithic → Virtualization → **Containers**



Shared kernel on the host to run multiple guest applications

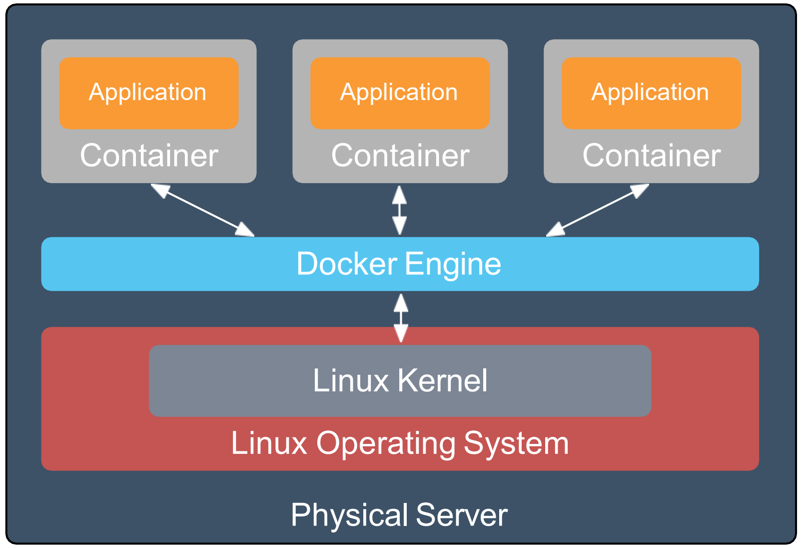
Advantages over VMs

- Containers are more lightweight
- No need to install a guest Operating System
- Less CPU, RAM, storage overhead
- More containers per machine
- Greater portability

© 2024 by Innovation In Software Corporation

Containers - Challenges

Monolithic → Virtualization → **Containers**



Shared kernel on the host to run multiple guest applications

Container Challenges

- Early Docker focused on single-node operations
- Up to user to cluster Docker hosts and manage deployment of containers on cluster
- User solves for automatic scale out of applications
- User solves for service discovery between application components (microservices)

© 2024 by Innovation In Software Corporation

Container – Concept of Operations

© 2024 by Innovation In Software Corporation

Container based virtualization

Uses the kernel on the host operating system to run multiple guest instances

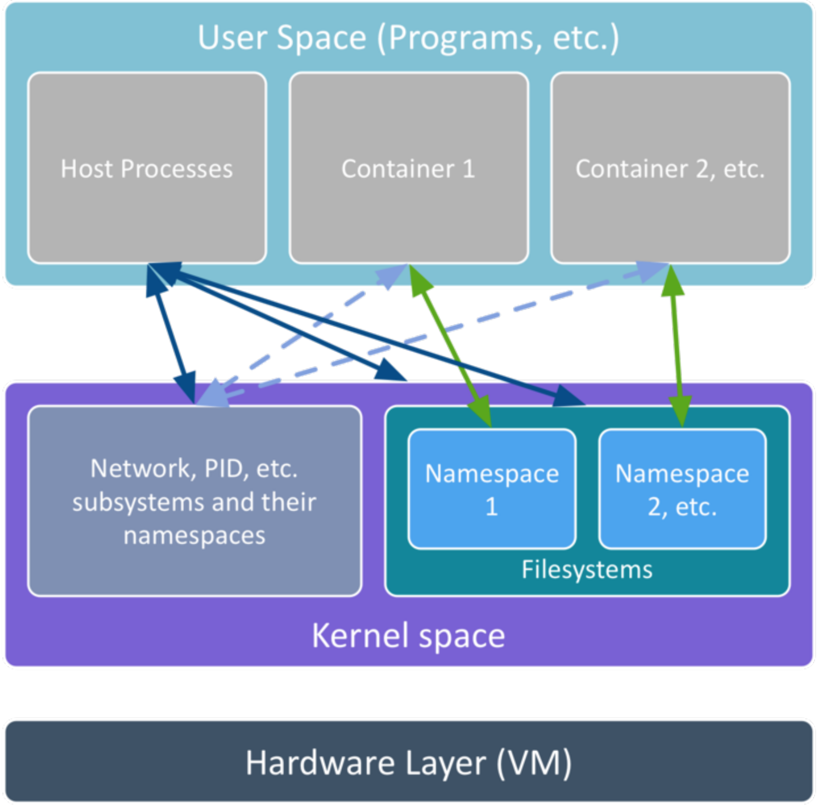
- Each guest instance is a container
- Each container has its own
  - Root filesystem
  - Processes
  - Memory
  - Devices
  - Network Ports

© 2024 by Innovation In Software Corporation

Isolation with Namespaces

Namespaces - Limits what a container can see (and therefore use)

- Namespace wrap a global system resource in an abstraction layer
- Processes running in that namespace think they have their own, isolated resource
- Isolation includes:
  - Network stack
  - Process space
  - Filesystem mount points
  - etc.



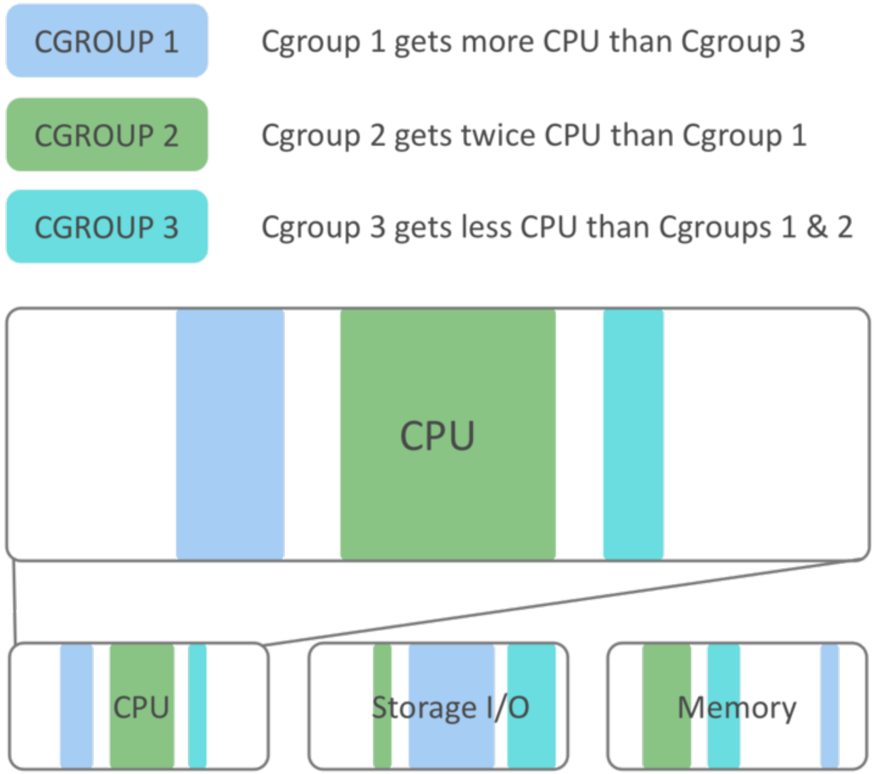
© 2024 by Innovation In Software Corporation

Isolation with Control group (Cgroups)

Cgroups - Limits what a container can use

- Resource metering and limiting
  - CPU
  - MEM
  - Block/IO
  - Network

- Device node (/dev/*) access control



© 2024 by Innovation In Software Corporation

Docker resource limits

```
play@lab19:~$ docker run --help |grep -E 'cpu|mem|blk|io'
Options:
      --blkio-weight uint16            Block IO (relative weight), between 10 and 1000, or 0 to disable (default 0)
      --blkio-weight-device list       Block IO weight (relative device weight) (default [])
      --cgroup-parent string           Optional parent cgroup for the container
      --cpu-period int                 Limit CPU CFS (Completely Fair Scheduler) period
      --cpu-quota int                  Limit CPU CFS (Completely Fair Scheduler) quota
      --cpu-rt-period int              Limit CPU real-time period in microseconds
      --cpu-rt-runtime int             Limit CPU real-time runtime in microseconds
  -c, --cpu-shares int                 CPU shares (relative weight)
      --cpus decimal                   Number of CPUs
      --cpuset-cpus string             CPUs in which to allow execution (0-3, 0,1)
      --cpuset-mems string             MEMs in which to allow execution (0-3, 0,1)
      --device-read-iops list          Limit read rate (IO per second) from a device (default [])
      --device-write-iops list         Limit write rate (IO per second) to a device (default [])
```

```
-m, --memory bytes                   Memory limit
    --memory-reservation bytes       Memory soft limit
    --memory-swap bytes              Swap limit equal to memory plus swap: '-1' to enable unlimited swap
    --memory-swappiness int          Tune container memory swappiness (0 to 100) (default -1)
```

© 2024 by Innovation In Software Corporation

Container Use Cases

© 2024 by Innovation In Software Corporation

DevOps

Developers

Focus on applications inside the container

Operations

Focus on orchestrating and maintaining containers in production

© 2024 by Innovation In Software Corporation

Container use-cases

Development

- Allows the ability to define the entire project configuration and tear-down/recreate it easily
- Supports multiple versions of application simultaneously

© 2024 by Innovation In Software Corporation

Container use-cases

Development

- Allows the ability to define the entire project configuration and tear-down/recreate it easily
- Supports multiple versions of application simultaneously

Test Environments

- Same container that developers run is the container that runs in test lab and production – includes all dependencies
- Well formed API allows for automated building and testing of new containers

© 2024 by Innovation In Software Corporation

Container use-cases

Development

- Allows the ability to define the entire project configuration and tear-down/recreate it easily
- Supports multiple versions of application simultaneously

Test Environments

- Same container that developers run is the container that runs in test lab and production – includes all dependencies
- Well formed API allows for automated building and testing of new containers

Microservices

- Design applications as suites of services, each written in the best language for the task
- Better resource allocation
- One container per microservice vs. one VM per microservice
- Can define all interdependencies of services with templates

© 2024 by Innovation In Software Corporation

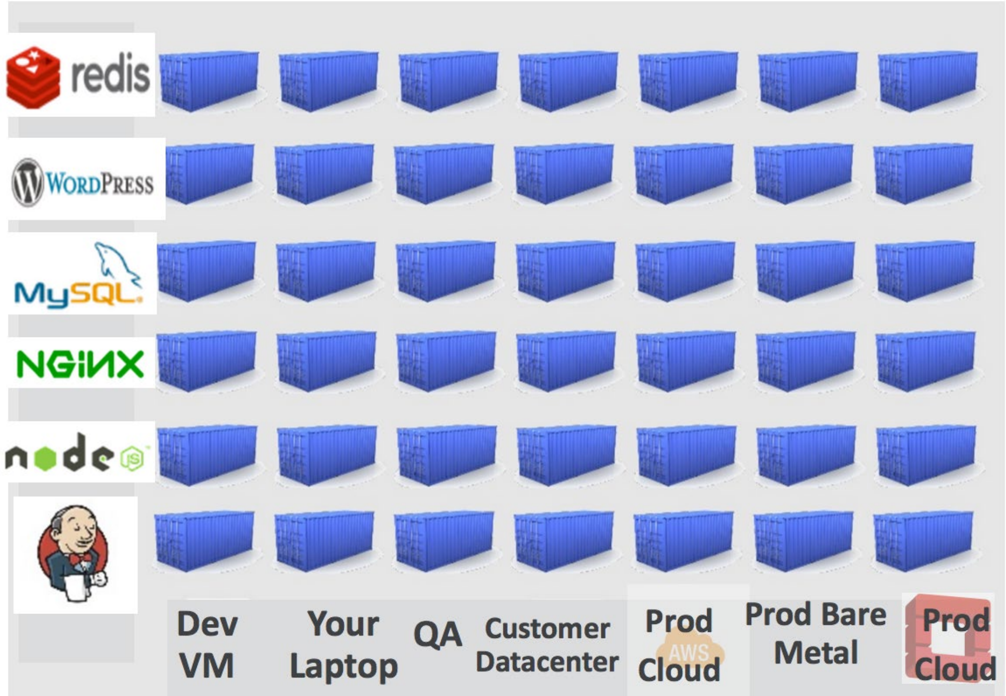
© 2024 by Innovation In Software Corporation

Container Overview

© 2024 by Innovation In Software Corporation

What is Docker?

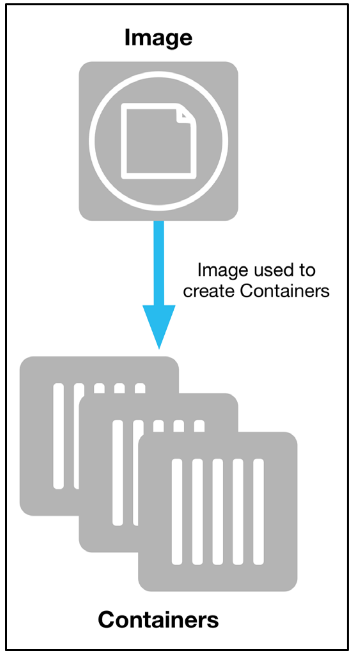
Docker allows you to package an application with all of its dependencies into a standardized unit for software development.



Terminology

Image

Read only template used to create containers
Built by you or other Docker users
Stored in Docker Hub, Docker Trusted Registry or your own Registry



© 2024 by Innovation In Software Corporation

Terminology

Image

Read only template used to create containers
Built by you or other Docker users
Stored in Docker Hub, Docker Trusted Registry or your own Registry

Container

- Isolated application platform
- Contains everything needed to run your application
- Based on one or more images

© 2024 by Innovation In Software Corporation

Container Images

© 2024 by Innovation In Software Corporation

Lab: Access VMs

© 2024 by Innovation In Software Corporation

Lab: Container resource limits

© 2024 by Innovation In Software Corporation

Docker Image Terminology

Image

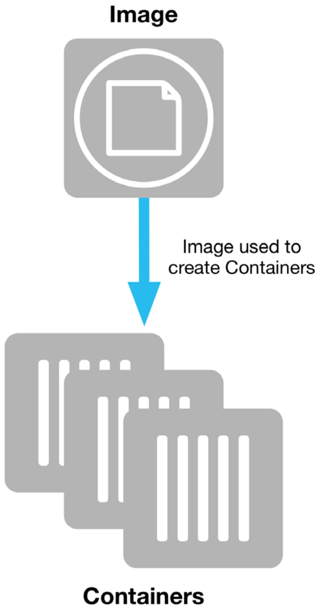
Union Filesystem

Dockerfile

Container

Hierarchy of files, with meta-data for how to create/run a container

- Read only template used to create containers
- Can be exported or modified to new images
- Created manually or through automated processes
- Stored in a Registry (Docker Hub, Docker Trusted Registry, etc.)



© 2024 by Innovation In Software Corporation

Docker Image Terminology

Image

Union Filesystem

Dockerfile

Container

UnionFS – Used by Docker to layer images

- Not a distributed File System

© 2024 by Innovation In Software Corporation

Docker Image Terminology

Image

Union Filesystem

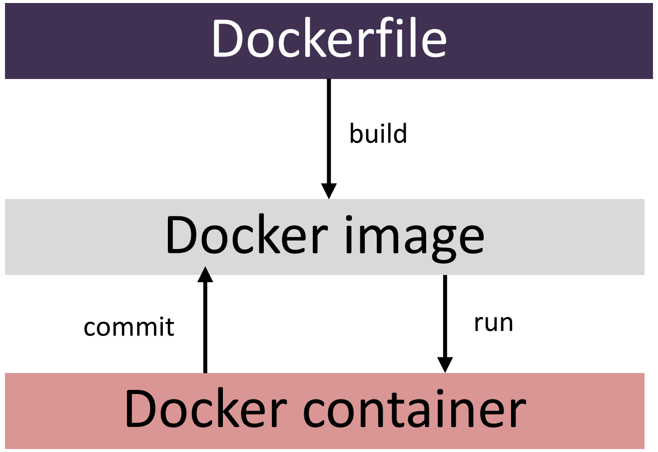
Dockerfile

Container

Configuration File (script) for creating images.

Defines:

- Existing image to be the starting point
- Set of instructions to augment that image (each of which results in a new layer of the file system)
- Meta-data such as ports exposed
- The command to execute when the image is run



© 2024 by Innovation In Software Corporation

Docker Image Terminology

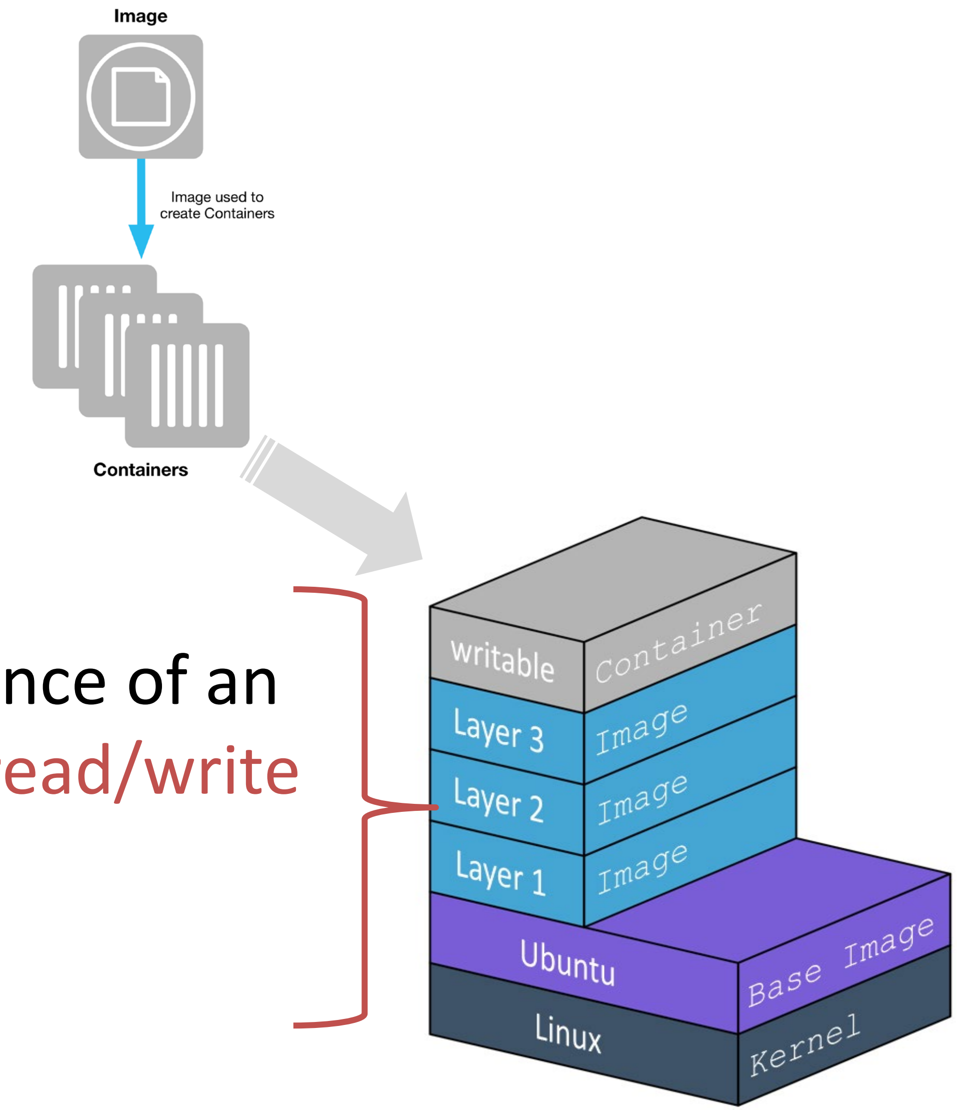
Image

Union Filesystem

Dockerfile

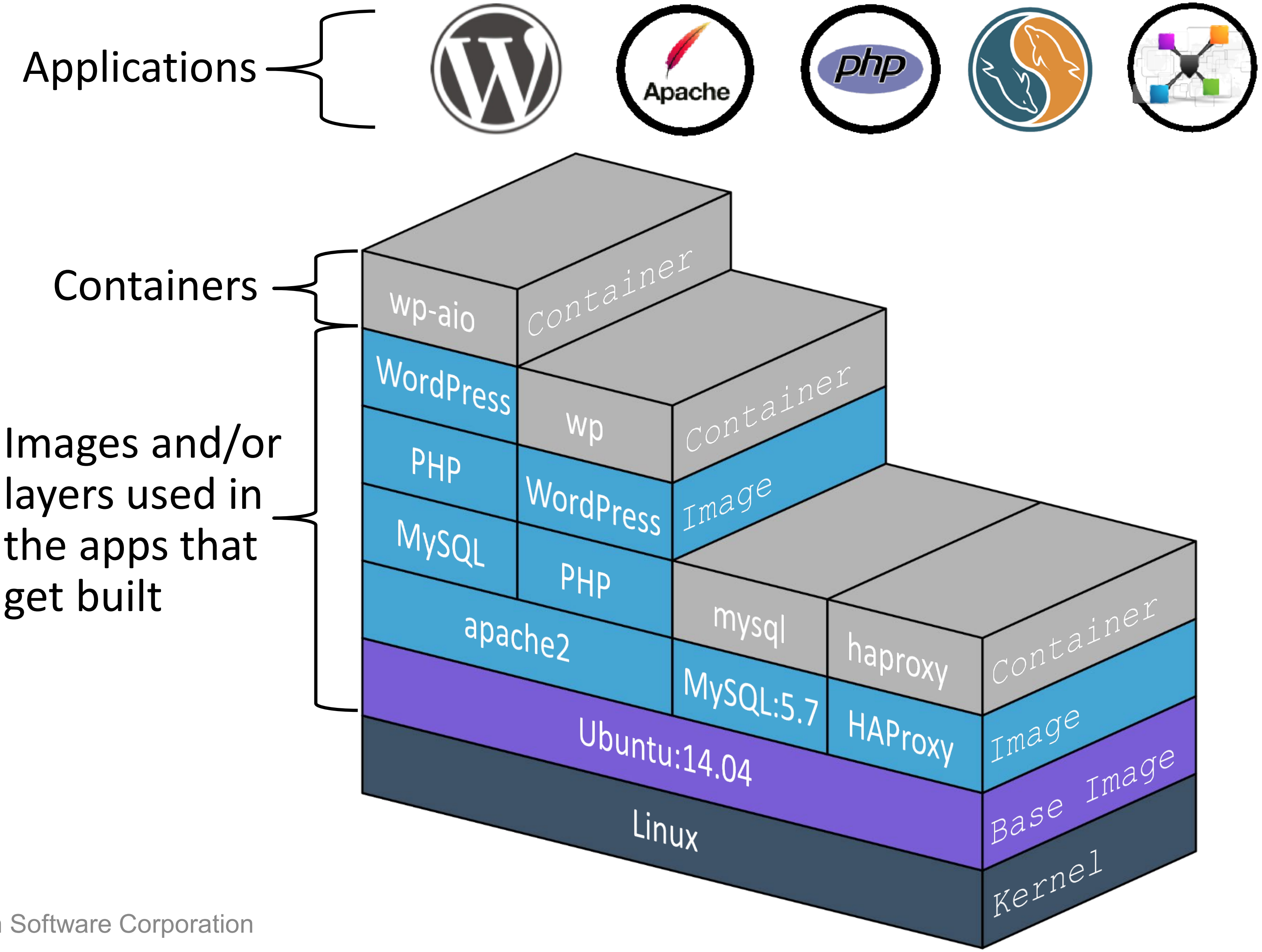
Container

Runtime instance of an image plus a read/write layer



© 2024 by Innovation In Software Corporation

Applications, Containers, and Images



© 2024 by Innovation In Software Corporation

Union Files System

© 2024 by Innovation In Software Corporation

Image/Container Interactions

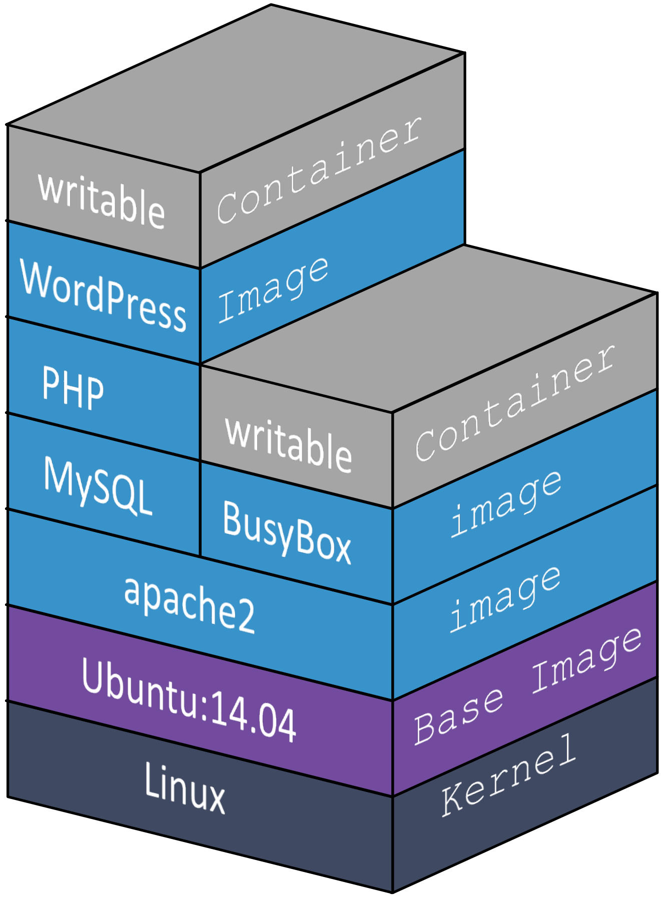
Pull an image from a Docker Registry

Run a container from an image

Add a file to a running container – Example, the image requires an additional file called index.html

Change to a running container – Example, the image requires an update of the index.php file

Delete files from a running container – All containers share the same host kernel

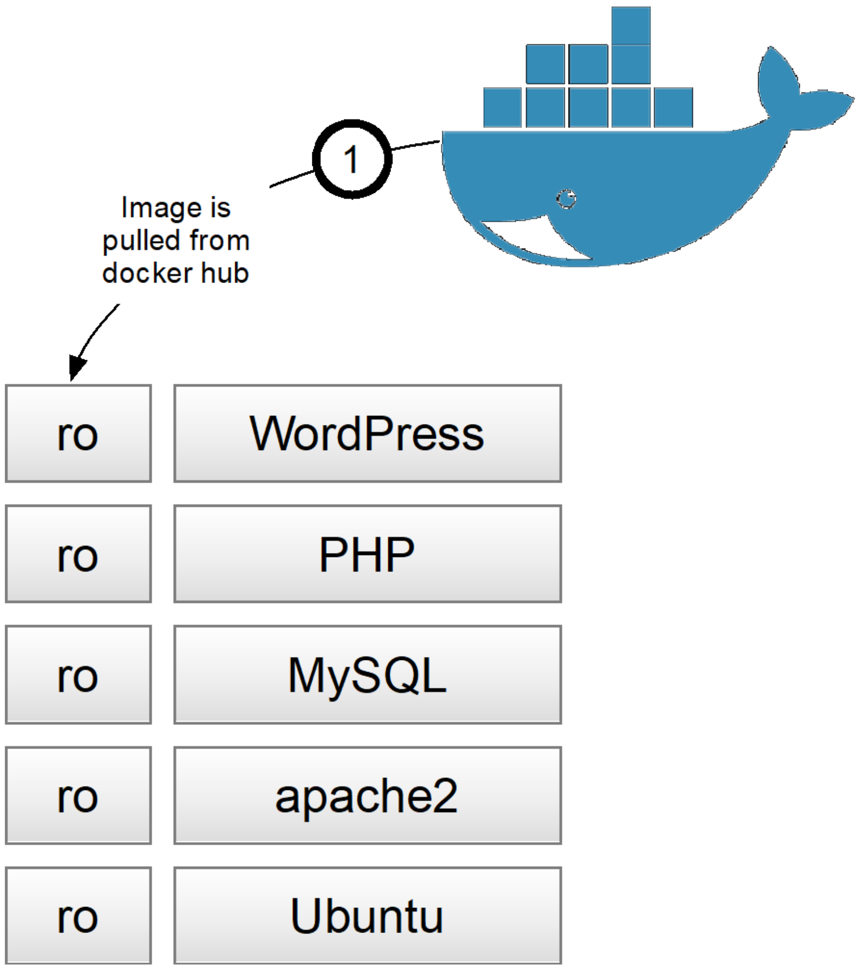


© 2024 by Innovation In Software Corporation

Pull the WordPress AIO

The WordPress All-In-One image is pulled from Docker Hub

```
$docker inspect wordpress-aio
```



Docker images are:

- Pulled or created
- Stored in a local image repo
- READ-ONLY
- The basis of all containers

© 2024 by Innovation In Software Corporation

Run the WordPress AIO

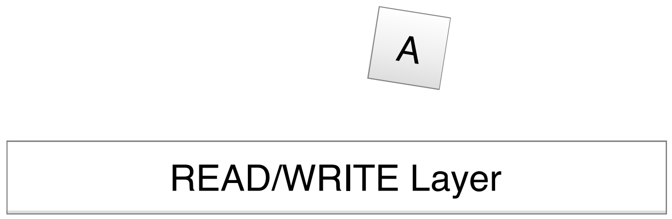
A container is made from the previously pulled WordPress AIO image

1. `$docker run wordpress-aio`
2. Container created
3. READ/WRITE layer created
4. All **ADD**, **CHANGE**, **DELETE** are committed to the READ/WRITE Layer

© 2024 by Innovation In Software Corporation

Key Takeaways

- **Images** are:
  - Highly portable and readily available
  - Reusable for many deployments
- **ADD** – ADD DATA to READ/WRITE Layer
- **CHANGE –** ADD DATA to READ/WRITE Layer
- **DELETE –** ADD DATA to READ/WRITE Layer
- **ALL** ADDs, CHANGEs, and DELETEs are copied to the READ/WRITE Layer



© 2024 by Innovation In Software Corporation

Dockerfile best practices

© 2024 by Innovation In Software Corporation

Advanced Docker builds

- A great docker image is small e.g. < 20MiB
- The ideal image contains only the application*
- A great image has very few layers*
- The ideal image has 1 layer*
- The ideal build is 100% reproducible
- The ideal build is as fast as possible

*except when it doesn’t

© 2024 by Innovation In Software Corporation

Q: When does performance matter?

© 2024 by Innovation In Software Corporation

Q: When does performance matter?

ALWAYS

© 2024 by Innovation In Software Corporation

Q: When does performance not matter?

© 2024 by Innovation In Software Corporation

Q: When does performance not matter?

NEVER

© 2024 by Innovation In Software Corporation

Typical order of operations

1. Correctness and Integrity
2. Readability and Usability
3. Performance

That said, let’s take performance where it’s free

© 2024 by Innovation In Software Corporation

Why care?

Smaller Images Mean:

- Faster builds
- Faster pulls
- Faster pushes
- Faster development
- Faster testing
- Faster deployment
- Faster feedback

© 2024 by Innovation In Software Corporation

Minimal base images

- scratch
- alpine (<5MiB)

Application Base Images

- redis:alpine
- mysql

Stack Base Images

- gcr.io/distroless (python, node, java, dotnet, cc)
- Python:alpine
- php:alpine
- golang

© 2024 by Innovation In Software Corporation

Layer review

- Images are stacks of layers
- Each Dockerfile line is a layer

```
#Layer n
FROM nixos/nix

#Layer n+1
ENV build=dev

#Layer n+2
CMD /app/bin/start

#Layer n+3
COPY app /app

#Layer n+4
RUN /app/bin/build.sh

```

© 2024 by Innovation In Software Corporation

Add small layers

Might be small, might be HUGE

```
FROM alpine

COPY . .

```

Probably smaller

```
FROM alpine

COPY app /app
```

© 2024 by Innovation In Software Corporation

Independent RUNs at top

RUNs are cacheable

```
FROM alpine

RUN dd if=/dev/random of=/tmp/seed bs=1 count=1

# More stuff ...
```

© 2024 by Innovation In Software Corporation

Dependent RUNs immediately after the file they depend on

```
#Layer n
FROM nixos/nix

#Layer n+1
ENV build=dev

#Layer n+2
CMD /app/bin/start

#Layer n+3
COPY app /app

#Layer n+4
RUN /app/bin/build.sh

```

© 2024 by Innovation In Software Corporation

Chain your RUNs

Chained commands happen in the same layer

```
FROM node:6-alpine

CMD ["node", "/www/src/index.js"]

COPY package.json /www/

RUN apk add --update git && \
    npm --prefix=/www install && \
    apk del git && \
    rm -rf /var/cache/apk/*

COPY src/ /www/src
```

© 2024 by Innovation In Software Corporation

Cleanup your RUNs

This build requires git, but the runtime doesn't. Let's cleanup after the build.

```
FROM node:6-alpine

CMD ["node", "/www/src/index.js"]

COPY package.json /www/

RUN apk add --update git && \
    npm --prefix=/www install && \
    apk del git && \
    rm -rf /var/cache/apk/*

COPY src/ /www/src
```

© 2024 by Innovation In Software Corporation

Add layers sparingly

Each line should have a good reason to exist

Common extraneous layers:
RUN followed by RUN
MAINTAINER <in+version+control@example.com>
EXPOSE 80
WORKDIR

Sometimes extraneous:
VOLUME /app/data

© 2024 by Innovation In Software Corporation

Order matters

Anti-Example:

```
COPY . /tmp/
RUN pip install --requirement /tmp/requirements.txt
```

Not bad?:

```
COPY requirements.txt /tmp/
RUN pip install --requirement /tmp/requirements.txt
COPY . /tmp/
```

© 2024 by Innovation In Software Corporation

Order matters

Good:

```
COPY LICENSE /tmp/
COPY requirements.txt /tmp/
RUN pip install --requirement /tmp/requirements.txt
COPY *.py /tmp/
```

Best:

```
COPY LICENSE /tmp/
COPY requirements.txt /tmp/
RUN pip install --requirement /tmp/requirements.txt
COPY constants.py /tmp/
COPY main.py /tmp/
```

© 2024 by Innovation In Software Corporation

.dockerignore - reduce build context

you can use .dockerignore in root of build context to exclude files from the build

.dockerignore makes these better:

```
COPY . .
COPY src/ /app/src
```

When simple, explicit is still preferable:

```
COPY foo /app/foo
COPY bar /app/bar
```

© 2024 by Innovation In Software Corporation

When to **not** minimize an image

When it impacts usability or readability:

- run a debugger inside the container (copy in the debugger dependencies)
- run a profiler inside the container (copy in the profiler dependencies)
- run a supporting application inside the container (e.g. redis-cli)
- open a shell inside a container

© 2024 by Innovation In Software Corporation

Multi-stage Docker builds

- Multiple FROM statements can be used to decompose a docker build into “stages”
- Conditionally build a single stage, or all stages

© 2024 by Innovation In Software Corporation

Common multi-stage patterns

- Create a “build” layer that performs the actual compilation of your application
- Create a “test” layer that contains canned dummy data for your application to consume or expose for testing
- Create a “production” layer that is stripped down to only a lightweight base image (like alpine) and your binary

© 2024 by Innovation In Software Corporation

Benefits of multi-stage builds

Portability

- Utilizing a ”build” layer in your Dockerfile allows a docker build to be ported to any build system (Jenkins, TravisCI, CircleCI, etc) that can build a container. No consideration has to be made as to whether the build system "supports" builds with your preferred runtime.

Security

- By only including explicitly what you want in the final image, you decrease your container attack surface

Simplicity

- Removes any requirement to have “test” and “production” Dockerfiles to facilitate testing.
- The number and size of layers in intermediate stages does not factor into the final deployable image.

© 2024 by Innovation In Software Corporation

Multi-stage build example

Create a named “build” stage

Build stage contains secrets that should not exist in the final image

Create an unnamed runtime stage

Copy artifacts from the “build” stage to the runtime stage

```
FROM node:8-alpine AS build
COPY .npmrc /home/user/.npmrc
COPY package.json /app/
RUN npm -prefix=/app install
COPY src /app/src

FROM node:8-alpine
CMD [“node”, ”/app/src/index.js”]
COPY --from=build /app /app
```

© 2024 by Innovation In Software Corporation

© 2024 by Innovation In Software Corporation

Lab: Multi-stage Dockerfiles

© 2024 by Innovation In Software Corporation

Lab: Analyzing images

© 2024 by Innovation In Software Corporation