THE CONTAINER UNIVERSE

Nicholas Dille

Docker Captain & Microsoft MVP

[Container Conf]

NICHOLAS DILLE

Husband, father, ops, automator

- Blogger since 2003
- Speaker since 2009
- Microsoft MVP since 2010
- Docker Captain since 2017
- Haufe.Group since 2016

AGENDA

All things containers

Docker

Kubernetes

Community projects

Opinionated selection of topics

Demo > slides

RELEVANCE OF DOCKER

With regard to...

Orchestration

- No, Kubernetes has won

Container runtime

- Often, but `containerd` is growing quickly

Image build

- Yes
- Other image builders are still niche players (e.g. `buildah`, `img`, `kaniko`)

BUILDKIT

concurrent, cache-efficient, and Dockerfile-agnostic builder toolkit

The project

- Initiated by Docker
- Community driven

Relevant features

- Multi-stage builds
- Parallel builds
- Remote build cache
- Build mounts
- Build secrets
- SSH forwarding

Working rootless implementation

DEMO: BUILDKIT

Standalone usage of buildkit:

Start buildkit daemon:

```
sudo buildkitd >buildkitd.log 2>&1 &
```

Run equivalent of docker build:

```
buildctl build \
    --frontend=dockerfile.v0 \
    --local context=. \
    --local dockerfile=. \
    --output type=image,name=localhost:5000/test,push=true
```

DEMO: BUILDKIT

List images in registry:

```
curl -s http://localhost:5000/v2/_catalog \
    | jq --raw-output '.repositories[]'
```

List tags of image:

```
curl -s http://localhost:5000/v2/test/tags/list \
    | jq --raw-output '.tags[]'
```

BUILDKIT CACHE WARMING

How it works

- Use remote images to warm cache
- Image layers will be downloaded as needed
- Same syntax using `--cache-from`
- Cache information must be embedded
- Prerequisites: Docker 19.03

Build with cache from remote image

```
export DOCKER_BUILDKIT=1

# Build image with cache information
docker build --tag myimage --build-arg BUILDKIT_INLINE_CACHE=1

# Build image from remote cache
docker build --cache-from myimage .
```

Demo: Buildkit Cache Warming

Build image with cache information:

```
export DOCKER_BUILDKIT=1
docker build \
    --tag localhost:5000/test:1 \
    --build-arg BUILDKIT_INLINE_CACHE=1 \
    .
docker push localhost:5000/test:1
```

Build with remote cache:

```
docker system prune --all
docker build \
    --cache-from localhost:5000/test:1 \
    .
```

DEMO: BUILDKIT CACHE INTERNALS

Check manifest for cache information:

```
curl -s \
    -H "Accept: application/vnd.docker.distribution.manifest.v2+json" \
    localhost:5000/v2/test/manifests/1 \
    | jq --raw-output '.config.digest' \
    | while read CONFIG_DIGEST; do \
        curl -s \
            -H "Accept: application/vnd.docker.container.image.v1+json" \
            localhost:5000/v2/test/blobs/${CONFIG_DIGEST} \
        | jq --raw-output '."moby.buildkit.cache.v0"' \
        | base64 -d \
        | jq; \
    done
```

BUILD SECRETS

Do not provide secrets using environment variables

`ENV` burns variables into image

Build arguments are only one option

Buildkit can mount secrets using `tmpfs`

Temporary files in `/run/secrets/`

DEMO: BUILD SECRETS

Use experimental syntax in `Dockerfile`:

```
# syntax=docker/dockerfile:1.0.0-experimental
FROM alpine
RUN --mount=type=secret,id=mysite.key ls -l /run/secrets
```

Build image with secret from `mysite.key`:

```
export DOCKER_BUILDKIT=1
docker build \
    --secret id=mysite.key,src=./mysite.key \
    --progress plain \
    .
```

SSH AGENT FORWARDING

Do not copy secrets into image layers

Buildkit can forward the SSH agent socket

Bad example:

```
FROM ubuntu
COPY id_rsa /root/.ssh/

RUN scp user@somewhere:/tmp/data .
RUN rm /root/.ssh/id_rsa
```

DEMO: SSH AGENT FORWARDING

Buildkit forwards the SSH agent socket

Prepare SSH agent:

```
ssh-keygen -f id_rsa_test
eval $(ssh-agent -s)
ssh-add id_rsa_test
ssh-add -l
```

Forward into build:

```
export DOCKER_BUILDKIT=1
docker build --ssh default --progress plain .
```

Compare local and build:

```
ssh-add -l
```

DOCKER CLI PLUGINS

Extend `docker` CLI with new sub commands

Located in `~/.docker/cli-plugins`

Executable file called `docker-<command>`

Command line parameters are passed as parameters

Plugin metdata via parameter `docker-cli-plugin-metadata`:

```
{
"SchemaVersion":"0.1.0",
"Vendor":"Nicholas Dille",
"Version":"0.0.1",
"ShortDescription":"Sample metadata",
"URL":"https://dille.name"
}
```

DEMO: DOCKER CLI PLUGINS

Place `docker-distribution` in `~/.docker/cli-plugins`:

```
mkdir -p ~/.docker/cli-plugins
cp docker-distribution ~/.docker/cli-plugins
chmod +x ~/.docker/cli-plugins/docker-distribution
```

Test integration into Docker CLI:

```
docker
```

Display help for sub commands:

```
docker distribution
```

DEMO: DOCKER CLI PLUGINS

Prepare local registry:

```
docker pull alpine
docker tag alpine localhost:5000/alpine
docker login localhost:5000
docker push localhost:5000/alpine
```

Use docker-distribution to determine tags:

```
docker distribution list-tags localhost:5000/alpine
```

DOCKER CLIENT PLUGINS MANAGER (CLIP)

CLIP created by Docker Captain Łukasz Lach

How it works

Framework for running containerized client plugins

Distributed using Docker registry

Plugin list

expose, publish, showcontext, microscan, dive, runlike, sh, hello

DEMO: DOCKER CLIENT PLUGINS MANAGER (CLIP)

Install plugin for `dive`:

```
docker clip add lukaszlach/clips:dive
docker clip ls
```

Run plugin dive:

```
docker dive localhost:5000/alpine
```

TROUBLESHOOTING `FROM scratch`

Missing tools

- Modern containers only contain a single binary
- No shell, no tools

Solution: Dump root partition content into container

DEMO: TROUBLESHOOTING `FROM scratch`

Container with image FROM scratch:

```
docker run -d --name traefik traefik:v1.7
```

Create local rootfs:

```
mkdir rootfs
docker create --name alpine alpine

docker cp alpine:/ rootfs
```

Copy into container:

```
rm rootfs/etc/hosts rootfs/etc/hostname rootfs/etc/resolv.conf
cd rootfs && docker cp . traefik:/ && cd ..
```

Enter container:

```
docker exec -it traefik /bin/sh
```

TROUBLESHOOTING NETWORKING

Missing tools

- Images are minimal
- Shell is available
- Containers are considered immutable

Share namespaces

- Create container image with required tools
- Run and share network/pid namespace
- Install tools as required

k8s 1.16 introduces ephemeral containers to troubleshoot pods

DEMO: NAMESPACE SHARING

Join namespaces with existing container:

```
docker run -d --name nginx nginx
docker run -it --rm \
    --net container:nginx \
    --pid container:nginx \
    alpine
```

COMMANDO: TOOLS ON DEMAND

Commando created by Docker Captain Łukasz Lach

Container images are created on demand:

```
docker run -it --rm cmd.cat/curl
```

Very powerful when troubleshooting networking:

```
docker run -it --rm \
    --net container:nginx \
    --pid container:nginx \
    cmd.cat/netstat/tcpdump/ip/ifconfig/ping
```

DEPENDENCIES

Container images are packaged runtime environments

Dependencies are included

Security issues will be discovered

You must build your images regularly

You must scan your dependencies

Tools include...

Clair, JFrog Xray, GitHub Security Alerts ...any many more!

TRIVY: IMAGE SCANS IN CI/CD

trivy made by Aqua Security

- Covers OS (alpine, Ubuntu, Debian, CentOS, RedHat)
- Covers package managers (Ruby, Python, PHP, NPM, Rust)

Break build with high severity findings:

```
trivy \
    --skip-update \
    --ignore-unfixed \
    --exit-code 1 \
    --severity HIGH,CRITICAL \
    python:3.4-alpine3.9
```

Cache management

- Databases are cached locally
- Persist cache across builds

MULTI-ARCH IMAGE

Image only work on a single platform

But containers are supported on multiple architectures and operating systems

Virtual images to the rescue

Manifest links to multiple images for supported platforms

Now integrated in Docker CLI (docker manifest)

Based on manifest-tool (by Docker Captain Phil Estes)

Official images are already multi-arch

MULTI-ARCH IMAGE: OPENJDK

```
$ docker run mplatform/mquery openjdk:8-jdk
Image: openjdk:8-jdk
Dockerfile buildx-0.command buildx-1.command buildx-2.command
Dockerfile buildx-0.command buildx-1.command buildx-2.command
- linux/amd64
- windows/amd64:10.0.17763.805
- windows/amd64:10.0.17134.1069
- windows/amd64:10.0.14393.3274
$ docker run mplatform/mquery openjdk:8-jdk-nanoserver

Image: openjdk:8-jdk-nanoserver
Dockerfile buildx-0.command buildx-1.command buildx-2.command
Dockerfile buildx-0.command buildx-1.command buildx-2.command
- windows/amd64:10.0.17763.802
- windows/amd64:10.0.17134.1069
```

MULTI-ARCH IMAGE: HELLO-WORLD

```
$ docker run mplatform/mquery hello-world
Image: hello-world
Dockerfile buildx-0.command buildx-1.command buildx-2.command
Dockerfile buildx-0.command buildx-1.command buildx-2.command
- linux/amd64
- linux/arm/v5
- linux/arm/v7
- linux/arm64
- linux/386

- linux/ppc64le
- linux/s390x
- windows/amd64:10.0.17134.1069
- windows/amd64:10.0.17763.802
```

MULTI-ARCH IMAGE: DOCKER

```
$ docker run mplatform/mquery docker
Image: docker
Dockerfile buildx-0.command buildx-1.command buildx-2.command
Dockerfile buildx-0.command buildx-1.command buildx-2.command
- linux/amd64
- linux/arm/v6
- linux/arm/v7
- linux/arm64
```

DEMO: BUILDING FOR OTHER ARCHITECTURES

Prepare for new sub command `buildx`

Enable experimental mode for client and enable qemu:

```
export DOCKER_CLI_EXPERIMENTAL=enabled
docker run --rm --privileged \
    docker/binfmt:820fdd95a9972a5308930a2bdfb8573dd4447ad3
```

Create builder:

```
docker buildx create --name mybuilder --use
docker buildx inspect --bootstrap
```

DEMO: BUILDING FOR OTHER ARCHITECTURES

Build multi-arch:

```
docker buildx build \
    --platform linux/arm,linux/arm64,linux/amd64 \
    --tag localhost:5000/nicholasdille/hello \
    . \
    --push
```

Inspect result:

```
docker buildx imagetools inspect \
    localhost:5000/nicholasdille/hello
```

DEMO: BUILD MULTI-ARCH WITH PROPER TAGS (1)

Build individual images to control tagging

Build for arm, arm64 and amd64:

```
docker buildx build --platform linux/arm \
    --tag localhost:5000/nicholasdille/hello:arm . --push
docker buildx build --platform linux/arm64 \
    --tag localhost:5000/nicholasdille/hello:arm64 . --push

docker buildx build --platform linux/amd64 \
    --tag localhost:5000/nicholasdille/hello:amd64 . --push
```

Test new images:

```
docker run localhost:5000/nicholasdille/hello:arm
docker run localhost:5000/nicholasdille/hello:arm64
docker run localhost:5000/nicholasdille/hello:amd64
```

This allows for proper versioning

DEMO: BUILD MULTI-ARCH WITH PROPER TAGS (2)

Create manifest list with all images:

```
docker manifest create --amend --insecure \
    localhost:5000/nicholasdille/hello \
    localhost:5000/nicholasdille/hello:arm \
    localhost:5000/nicholasdille/hello:arm64 \
    localhost:5000/nicholasdille/hello:amd64
docker manifest inspect localhost:5000/nicholasdille/hello
```

`docker context`

Manage connections to Docker instances

Like `docker-machine` without the deployment

Supports remoting via SSH

Check pre-defined context:

```
docker context ls
```

DEMO: `docker context`

Create and use new context:

```
docker context create docker-hcloud \
    --description 'Remote@Hetzner' \
    --docker 'host=ssh://020advanced-090dockercontext'
docker context use docker-hcloud
docker context ls
```

Check connectivity:

```
docker version
```

Check remote host:

```
docker run --uts host alpine hostname
```

docker context

Manage connections to Kubernetes clusters

```
k3d create --name context --worker 3
KUBECONFIG=$(k3d get-kubeconfig --name=context)
docker context create k3d --docker 'host=unix:///var/run/docker.sock'
```

CLOUD NATIVE STORAGE

OCI Registry As Storage

- Generalize artifact store
- Based on registry, e.g. Docker registry

Supports

- Multiple files per artifact
- Custom content types
- Config files per artifact - even with custom content type

Official guides

- Authoring OCI registry artifacts
- Annotations and Configurations in artifacts

DEMO: ORAS

Test command line tool oras

Create artifact:

```
echo blarg >artifact.txt
```

Upload:

```
oras push localhost:5000/test:latest artifact.txt
```

Download:

```
oras pull localhost:5000/test:latest --output out
```

Check:

```
cat out/artifact.txt
```

DEMO: ORAS INTERNALS

Check contents of registry

Check registry:

```
curl http://localhost:5000/v2/test/tags/list
curl -sH "Accept: application/vnd.oci.image.manifest.v1+json" \
    http://localhost:5000/v2/test/manifests/latest | jq
```

Get content:

```
DIGEST=$(curl -sH "Accept: application/vnd.oci.image.manifest.v1+json" \
    http://localhost:5000/v2/test/manifests/latest \
    | jq --raw-output '.layers[].digest')
curl -sH "Accept: application/vnd.oci.image.manifest.v1+json" \
    http://localhost:5000/v2/test/blobs/${DIGEST}
```

WINDOWS SUBSYSTEM FOR LINUX V2

Goals

- Improve file system performance
- Full system call compatibility

Design

- WSL 2 runs a Linux kernel maintained by Microsoft in a tiny VM
- Root file system is stored in `vhdx` file
- Runs in the same network as Windows

Availability

- Windows 10 Insider Fast Ring (>= 10.18917)

DEMO: WSL 2

Run Docker locally::

```
uname -a
sudo /etc/init.d/docker start
docker context ls
```

Test Docker:

```
docker run --uts host alpine hostname
```

Shutdown tiny VM::

```
wsl --shutdown
```

ROOTLESS DOCKER

Reduce attack surface

It is not...

- Running as non-root in container
- Enabling user namespace mapping

It is...

- Running `dockerd` as non-root
- Experimental preview in Docker 19.03
- See slides by Aikihiro Suda

DEMO: ROOTLESS DOCKER

Running Docker as unprivileged user

- Based on user namespaces (map UIDs to unprivileged range)
- Handling of namespaces is based on `rootlesskit` by Akihiro Suda
- Networking is based on `moby/vpnkit`

```
curl -sSL https://get.docker.com/rootless | sh
```

What does not work

- Only Ubuntu supports `overlay` fs
- Other distributions use `vfs` (not recommended for production)

Rootless Everything

Akihiro is also working on...

- Rootless kubernetes
- Rootless buildkit

IMAGE BUILDING

kaniko

- Kaniko created by Google
- Daemonless, unprivileged
- Uses Dockerfile

buildah

- Buildah created by RedHat
- Daemonless, unprivileged
- Script/command based (not using Dockerfile)

img

- img created by Jessie Frazelle
- Based on buildkit
- Daemonless, unprivileged
- Uses Dockerfile

DEMO: IMG

Building and pushing using `img`

Enter pod:

```
docker exec -it img bash
```

Build image:

```
img build --tag localhost:5000/test .
```

Push image:

```
img push --insecure-registry localhost:5000/test
```

PRIVILEGED CONTAINERS

Privileged containers have access to all syscalls

If able to start containers, just leave the isolation:

```
docker run -it \
--privileged \
--pid=host \


alpine \
nsenter -t 1 -m -u -n -i sh
```

Parameters of `nsenter`

- Work with namespaces (`nsenter`)
- Uses process tree of host (`--pid=host`)
- Get namespace from PID 1 (`-t 1`)
- Enter namespaces required for shell (`-m -u -n -i`)

TAMING PRIVILEGES CONTAINERS

For example

Isolates containers in a lightweight VM
Configured as a container runtime (instead of `runc`)

DOCKER ENGINE API FILTER

For example `sockguard`

- No privileged containers
- No host bind mounts
- No host network

```
/sockguard -upstream-socket /var/run/docker-raw.sock -filename
```

DOCKER-APP

What it is

De facto standard is using `docker-compose` for apps

Package and distribute using container registry

Standalone binary in version <= 0.6

Docker CLI plugin in version >= 0.7

Further reading

Workshop by Docker Captain Michael Irwin @ Docker Summit 2019

DOCKER-APP: PREPARATION

This is an experimental CLI feature:

```
cp ~/.docker/config.json ~/.docker/config.json.bak
cat ~/.docker/config.json.bak | jq '. + {"experimental": "enabled"}'
```

DOCKER-APP: CREATION

Create an app stack in hello.dockerapp:

```
docker app init --compose-file docker-compose.yml hello
```

Add parameters `port` and `text`

Push to registry and check resulting app:

```
docker app push --tag localhost:5000/hello:1.0 hello
docker app inspect localhost:5000/hello:1.0
```

DOCKER-APP: DEPLOY

Render app from registry:

```
docker app render localhost:5000/hello:1.0
```

Render app with custom parameter:

```
docker app render \
    --set text="hello containerconf" \
    localhost:5000/hello:1.0
```

DOCKER-APP: INTERNALS

Stored like an image

Check image manifest:

```
MANIFEST=$(curl -sH \
    "Accept: application/vnd.docker.distribution.manifest.v2+json" \
    http://localhost:5000/v2/hello/manifests/1.0)
```

Image configuration is boring:

```
CONFIG=$(echo "${MANIFEST}" | jq --raw-output '.config.digest')
curl -o - \
    http://localhost:5000/v2/hello/blobs/${CONFIG} | jq
```

Layer contains packaged app:

```
LAYER=$(echo "${MANIFEST}" | jq --raw-output '.layers[0].digest')
curl -o - \
    http://localhost:5000/v2/hello/blobs/${LAYER} | tar -tvz
```

CLOUD NATIVE APPLICATION BUNDLE

CNAB is a standard packaging format for multi-component distributed applications

Features

1. Single logical unit (app)
2. Lifecycle management (install, upgrade, uninstall)
3. Digital signatures for bundles
4. Digital signatures for lifecycle state
5. Support for air gapped environments
6. Distribution

Specification

- Core spec addresses features 1, 2, 6
- Registry spec address features 5, 6
- Security spec addresses features 3, 4

CNAB REFERENCE IMPLEMENTATION

Easily create bundles with Duffle

Containerized install script

Must accept parameters

`install|uninstall|upgrade|downgrade|status`

Directory layout:

```
.
├── bundle.json
├── cnab
│   ├── app
│   │   └── run
│   └── Dockerfile
└── duffle.json
```

DEMO: DUFFLE

Basic example:

Clone examples:

```
git clone https://github.com/deislabs/duffle
```

Build bundle:

```
duffle build ./examples/helloworld/
```

List existing bundles:

```
duffle bundle list
```

DEMO: DUFFLE

Install bundle:

```
duffle install helloworld-demo helloworld:0.1.1
```

List installed bundles:

```
duffle list
```

Uninstall bundle:

```
duffle uninstall helloworld-demo
```

Many example bundles are available

Start a new bundle:

```
duffle create foo
```

CNAB INSTALLER

Interface with Helm, Terraform etc. using Porter

What it is

Containerized installer

Executes commands defined in `porter.yaml`

Mixins provide templates for common tasks

Publish to OCI registry

Use CNAB compliant tool to install from registry

DEMO: PORTER

Play with a new package

Create and build package:

```
porter create
```

Check porter.yaml:

```
vim porter.yaml
```

Build package:

```
porter build
```

CNAB AND REGISTRIES

Uses OCI registries as storage

`cnab-to-oci` manages bundles in registries

Convert a CNAB bundle for upload:

```
# Clone examples
git clone https://github.com/docker/cnab-to-oci


# Push and pull
cnab-to-oci push cnab-to-oci/examples/helloworld-cnab/bundle.j
cnab-to-oci pull myhubusername/repo@sha256:6cabd752cb01d2efb94
```

Better experience with `duffle`, `porter` and `docker-app`

K3S: THE LIGHTWEIGHT K8S

k3s is made by Rancher

- Removes legacy, alpha and non-default features
- Removes most plugins
- Adds sqlite3 as default configuration store
- Adds simple launcher
- Packages dependencies like containerd, flannel, CoreDNS, CNI

Demo: SSH into fresh VM

```
# Run k3s server
k3s server &
k3s kubectl get nodes
```

K3D: K3S IN DOCKER

k3d is made by Rancher

- Lightweight k3s distribution
- Prerequisites: Docker

Demo: Run on fresh Docker host

```
# Deploy k3s cluster
k3d create --name k3d --workers 3
export KUBECONFIG="$(k3d get-kubeconfig --name='k3d')"
kubectl cluster-info
docker ps
```

K8S: DO NOT DO BARE-METAL

Bootstrapping is hard

- Use offerings in public clouds
- Otherwise, do not script yourself
- Use a project like kubespray

Mind the focus of KthW

- Focuses on installation of k8s
- But: Docker can interfere with iptables and break DinD
- But: Some CNI plugins do not configure `portmap` and break `hostPort`

You build it, you own it

- Your responsibility
- Load balancing is much harder
- DMZ, firewalls and company policies are in your way
- Allocatable host resources and pod eviction are yours to configure

SUMMARY

Docker has become one among many players

Docker is investing heavily in the community

The community is also investing heavily - not necessarily in Docker ;-)

Hard to keep up with development

Who to follow: Jessie Frazelle, Phil Estes, Kelsey Hightower, Jérôme Petazzoni, Bret Fisher, Tibor Vas, Tonis Tiigi, Akihiro Suda, Brandon Mitchell, Darren Shepherd

What to follow: Learn k8s, TGIK