

Infrastructure As Code With Docker

Prelude

Welcome to this Software-Craftsmen workshop!

Martin Ahrer

Partner of Software Craftsmen GmbH & Co KG

Founder and president of [Enterprise Java User Group Austria](#)

Working with Java since 1998

Working with Docker as of early beta releases (pre 1.0)

Martin.Ahrer@software-craftsmen.at

twitter: [@MartinAhrer](#)

What is driving the need for containers?

- Continuous Delivery
- MicroService Architecture
- DevOps Culture

Continuous Delivery (CD)

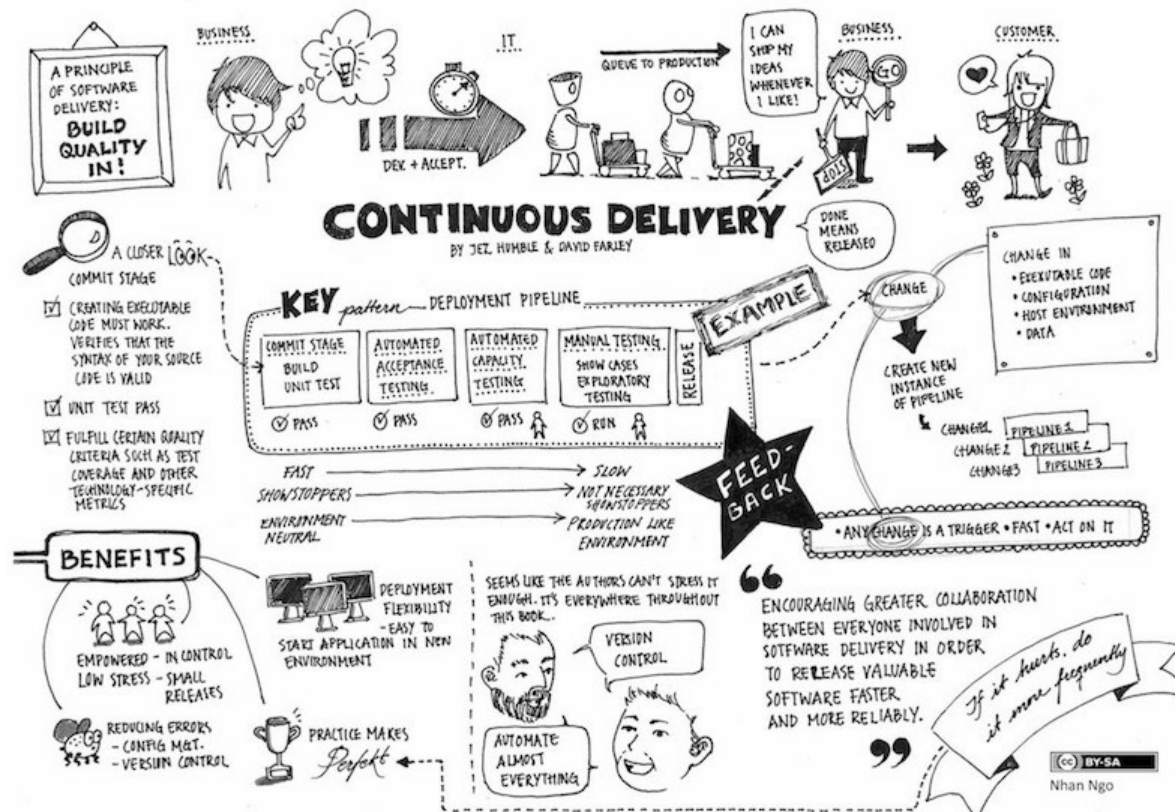


Figure 1. The Continuous Delivery Idea

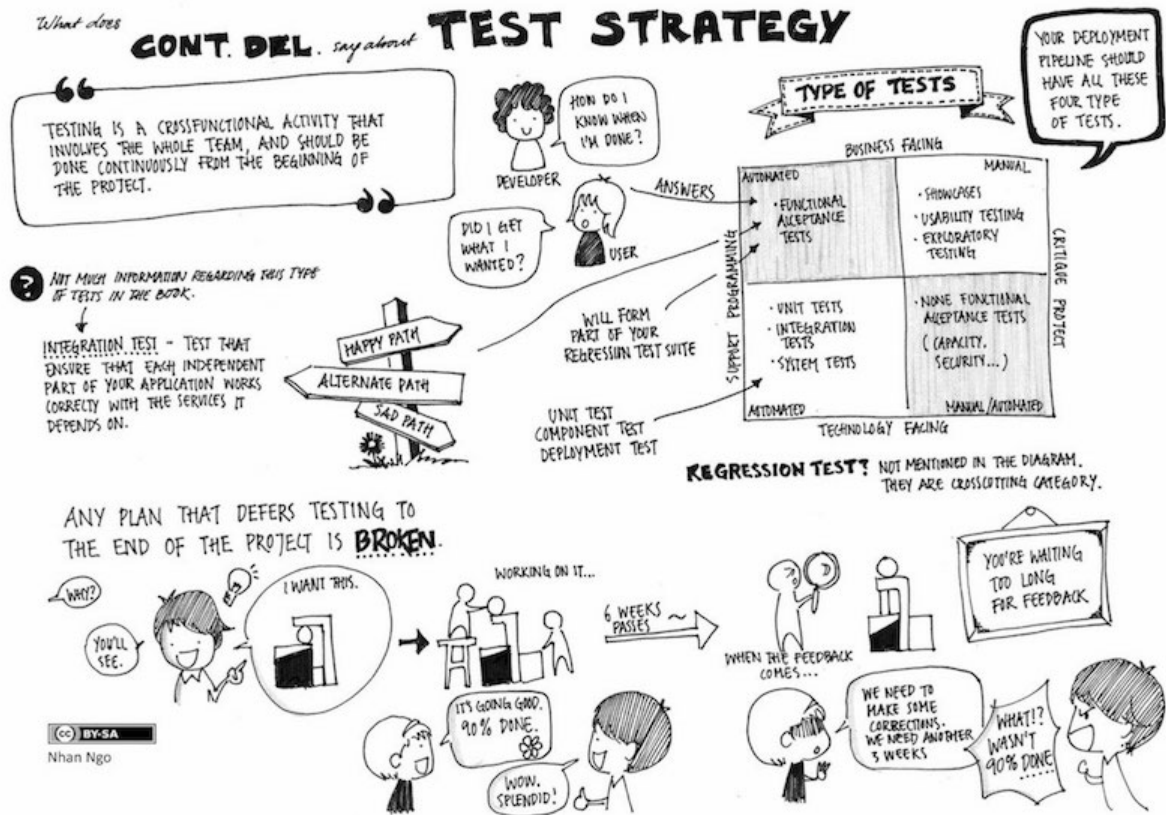


Figure 2. The Continuous Delivery Testing Strategy

What does **CONT. DEL** *sup about* **AUTOMATED ACCEPTANCE TESTING**

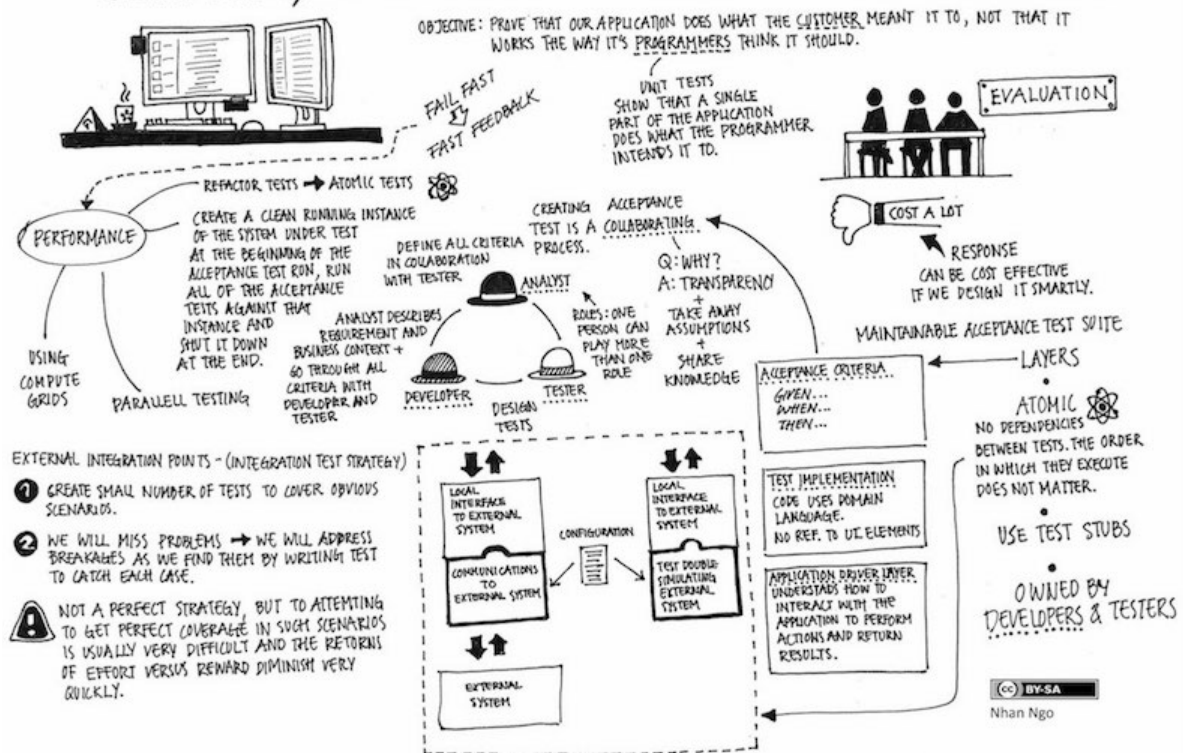


Figure 3. The Automated Acceptance Testing Strategy

A developer - operations scenario

Developer: “I need a host for performance testing 3.11”

Operations: “OPS is currently busy preparing a release”

Developer: “Can I support?”

Operations: “No, we have no docs”

Developer: “Is hardware or at least a virtual machine image available?”

Operations: “We have an old spare server somewhere!”

Developer: “WTF - when will I be able to start testing?”

Operations: “In approx. 6 hrs a colleague is free, then it will take a min. of 3hrs to setup the machine.”

Typical problems of test systems

- Multiple stakeholders are **competing for test systems** (DEV, QA, UAT, ...)
- Test systems are left in **undefined (messy) state** (experimental changes to configurations being the worst)
- Installations of test systems are **costly and time-consuming**

What is Continuous Delivery



Continuous Delivery is the natural extension of Continuous Integration: an approach in which teams ensure that every change to the system is releasable, and that we can release any version at the push of a button

— <http://www.thoughtworks.com/continuous-delivery>

Implications of Continuous Delivery

- **Release any version** ⇒ A CD system must be able to rebuild/release any version of a product.
- **Deploy any version** ⇒ A CD system must be able to **recreate any version of every required runtime environment** (UAT, Capacity, Performance, ...).
- Allowing for **multiple deployments a day**, setup of runtime environments must be **extremely fast**.

How do we version a runtime environment?

- Operating system
- Network configuration
- 3rd party software

How do we run multiple versions concurrently?

Provisioning 1/2



Server provisioning is a set of actions to prepare a server with appropriate systems, data and software, and make it ready for network operation. Typical tasks when provisioning a server are: select a server from a pool of available servers, load the appropriate software (operating system, device drivers, middleware, and applications), appropriately customize and configure the system...

— <https://en.wikipedia.org/wiki/Provisioning>

Provisioning 2/2

Provisioning of "bare metal" or Hypervisor virtualized machines is too slow and costly in general for a CD process.

Provisioning is mostly based on bare scripting, package management (such as apt) and configuration management tools (such as puppet or chef).

⇒ A CD environment asks for the fastest technology to enable build-time provisioning.

Enter container technology

Container technologies (such as Docker or Windows Server Container) are **enablers for efficient CD implementations**

Containers allow the creation of **ready-to-run packages** including

- Application binaries
- Operating system
- Network configuration
- ...

MicroService Architecture

What is a Micro Service?



The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

The old-school deployment monolith

Traditional web application architecture

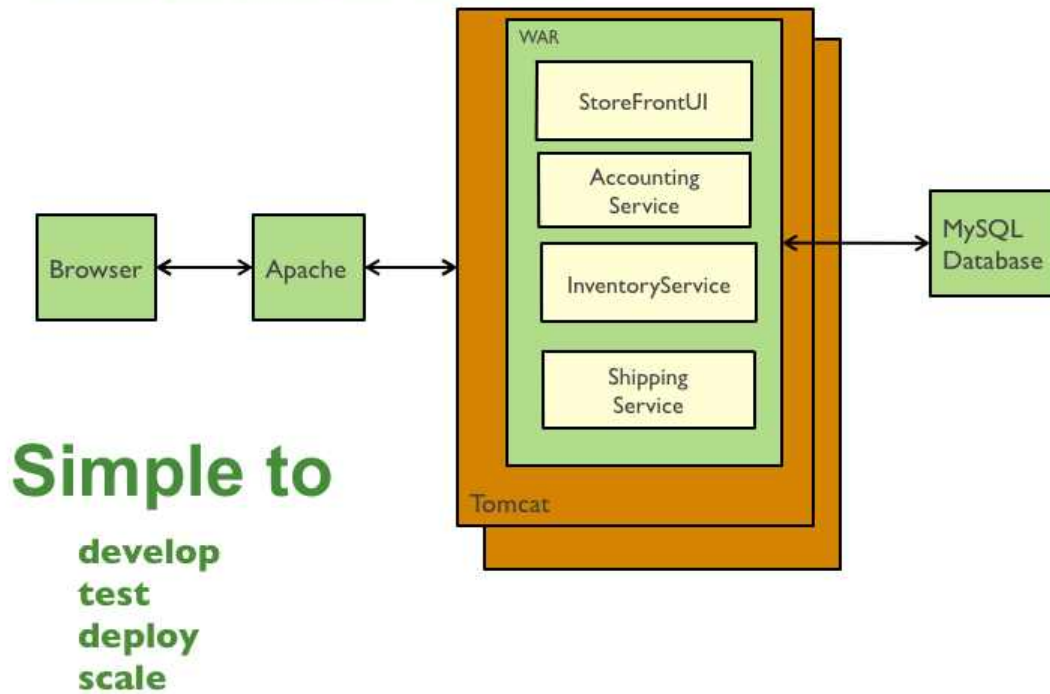
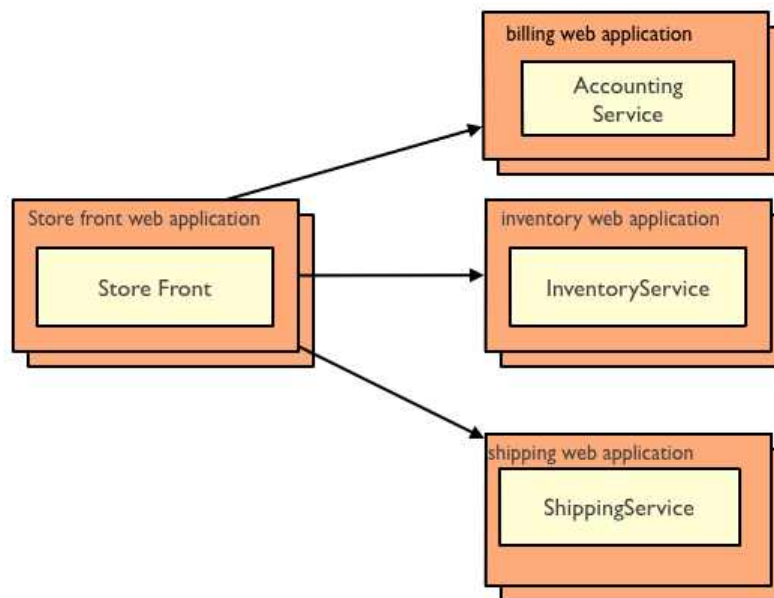


Figure 4. <http://microservices.io/index.html>

The micro services way

Y axis scaling - application level



Apply X axis cloning and/or Z axis partitioning to each service

Figure 5. <http://microservices.io/index.html>

The implication of micro services

With availability in mind a micro services architecture has to focus on stable communication channels across all services.

We need to spend much more time on topics such as resilience.

Therefore it is vital to have technologies that help cutting development efforts in other areas such as packaging applications, configure networks, etc.

DevOps

What is DevOps?



DevOps is a software development method that emphasizes communication, collaboration (information sharing and web service usage), integration, automation, and measurement of cooperation between software developers and other IT professionals. The method acknowledges the interdependence of software development, quality assurance (QA), and IT operations, and aims to help an organization rapidly produce software products and services and to improve operations performance.

— <https://en.wikipedia.org/wiki/DevOps>

Who is doing DevOps?



In October 2014, Rackspace commissioned independent technology market research specialist, Vanson Bourne, to survey 700 IT decision makers in companies with over 250 employees based in the UK, US, and Australia. They found that 55% of the respondents had implemented DevOps practices and 31% more planned to do so within the next three years.

— <https://www.managementstack.com/cfo-guide-q-improving-financial-results-using-devops/>

The ancient model of separated developers and operations

“

Divide and conquer, Caesar's strategy to break huge problems down into smaller parts, is an outdated model for structuring teams and organizations. Breaking teams apart by area like development, QA, operations, product management, etc, creates silo like divisions of labor. Unfortunately, these divisions create so many “walls of confusion” between the silos that your speed and agility is seriously hampered.

— <http://www.agileweboperations.com/devops-why-silos-suck-and-how-to-break-them>

Disadvantages of the ancient model

- Developers barely pay attention to operations' requirements
- Operations has little insight into application and has a hard time understanding an application that goes berserk.

Advantages of the DevOps model

- Developers get educated in minor operations tasks and learn to work towards a product that does not just "work on my machine".
- Operations is less busy with trivial tasks such as provisioning hosts but can focus on application specifics such as hardening security, adding monitoring, ...
- Operations can provide smart tools to developers in order to enable them to spin up new machines as the need them

DevOps synergy

Developers and Operations are cooperating on achieving a primary business goal: "deliver and deploy product features to the customer".

What is Docker?

Docker in short

- Docker containers **wrap a piece of software** in a complete filesystem that **contains everything needed to run**:
 - code, runtime, system tools, system libraries – anything that can be installed on a server.
- This guarantees that the software will **always run the same**, regardless of its environment.

Lightweight

- Containers running on a single machine **share the same OS kernel**
- They **start instantly** and **use less RAM**.
- Images are constructed from **layered filesystems** and share common files, making **disk usage** and image downloads much more **efficient**.

Lightweight

“

... Google starts over 2 billion containers per week, over 3000 started per second ...

— *<http://www.infoq.com/news/2014/06/everything-google-containers>*

Open

Open Container Initiative (OCI)

“

The OCI is a governance council responsible for standardizing the most fundamental components of container infrastructure such as image format and container runtime.

— Nigel Poulton. “*Docker Deep Dive.*”

Open

- Early Docker implementations were based on LXC (Linux Containers) allowing it to run on any Linux system.
- Today Docker runs on a variety of OS
 - Docker for Linux
 - Docker for Windows/Windows Server/Mac
 - Docker for (AWS, Azure, ..)

Open

“

As of Docker 1.11, the Docker Engine architecture conforms to the OCI runtime spec.

— *Nigel Poulton. “Docker Deep Dive”*

Secure

- Containers **isolate** applications from one another and the underlying infrastructure.
- Docker containers (on Linux) are using the **Linux process model** utilizing mature techniques such as **cgroups**, **namespaces**, ...

Secure



You can expect Intel to continue to enrich the Intel-VT_x instruction set and for the Linux kernel and containers to take advantage of those capabilities without the hypervisor as an intermediary.

Combined with removing most of the operating system wrapped arbitrarily around the application in a hypervisor VM, containers may actually already be more secure than the hypervisor model. But we can say for certain that given time this will certainly be true.

— <http://cloudscaling.com/blog/cloud-computing/will-containers-replace-hypervisors-almost-certainly/>

Docker "Mantra"

"Build, Ship, and Run Any App, Anywhere"

Docker Ecosystem

- Docker Engine
 - Docker Registry
 - Docker Hub
 - Docker Trusted Registry (commercial)
- Docker Machine
- Docker Compose
- Docker Swarm
- Docker Cloud (commercial)
- Docker Datacenter (commercial)

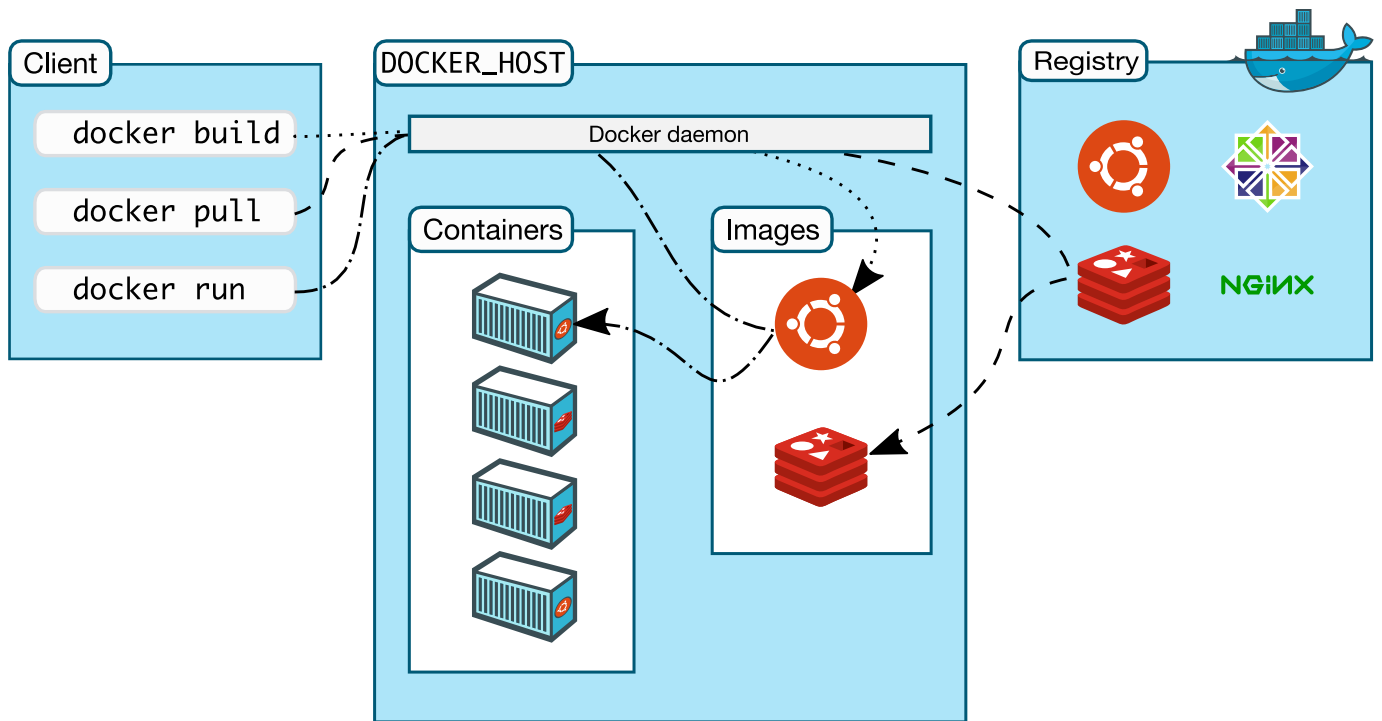
Understanding Docker

The Docker Platform

- Docker provides tooling and a platform to manage the lifecycle of containers
 - **Encapsulate** applications into Docker containers
 - **Distribute and ship** containers to teams for development and testing
 - **Deploy** application containers to production environment (local data center or the Cloud)

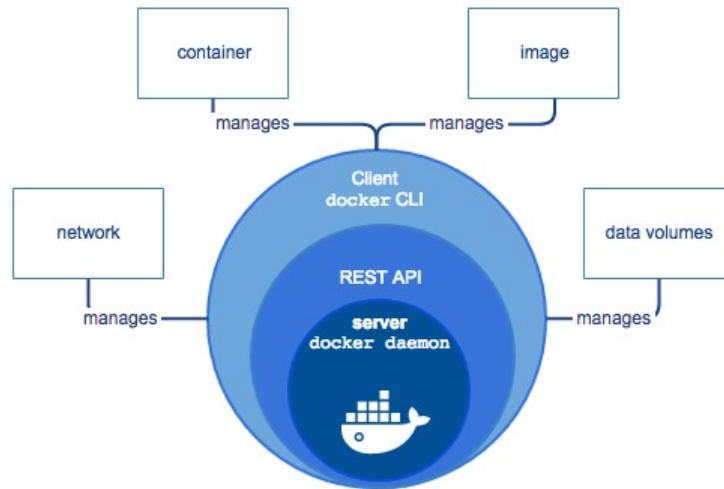
Docker Architecture

The most important components of Docker



Docker Architecture

Docker Daemon/Engine



Docker Architecture

client/server

- The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers.
- The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon.
- The Docker client and daemon communicate via sockets or through a REST API.

Docker Architecture

client/server

- The Docker daemon
 - The Docker daemon runs on a host machine.
 - The user uses the Docker client to interact with the daemon.
- The Docker client
 - The Docker client, in the form of the docker binary, is the primary user interface to Docker.
 - It accepts commands and configuration flags from the user and communicates with a Docker daemon.
 - One client can even communicate with multiple unrelated daemons.

Docker Architecture

Registry

- A docker registry is a server offering a **library of images**.
- A registry can be public or private
 - and can be on the same server as the Docker daemon or Docker client,
 - or on a totally separate server.

Service

- Switching engine into swarm mode enables a **service** abstraction
 - A service allows a swarm of nodes to work together, running a defined number of instances of a replica task.
 - You can specify the number of concurrent replica tasks to run, and the swarm manager ensures that the load is spread evenly across the worker nodes.
 - To the consumer, a service appears to be a single application.
- ❗ as of writing, other offerings such as **Kubernetes** might provide more powerful/popular abstraction of a service.

This is how it feels working with Docker

```
docker image pull alpine (1)
docker container run --name mycontainer alpine echo "Hello" (2)
docker container ls -a (3)
docker container rm mycontainer (4)
```

- 1 Pull the `alpine` image from the Docker Hub (a public registry).
- 2 Run a new container based on the image `alpine` and execute a command.
- 3 Show all Docker containers.
- 4 Delete Container

What can I use Docker for?

Fast, consistent delivery of applications

- Docker can **streamline the development lifecycle** by allowing developers to work in **standardized environments**
 - local containers provide applications and services.
 - **no local installation** of application and service runtime
- We can integrate Docker into continuous integration and continuous deployment (CI/CD) workflow.
 - CI/CD steps are run as container processes
 - **Simplified setup** of CI/CD infrastructure

Fast, consistent delivery of applications

~~Using the containerized AsciiDoctor tools~~

- AsciiDoctor is a popular toolset and and plain text markup language (based on Ruby on Rails) for producing high-quality documentation (PDF, epub, ...).
- Instead of installing the Ruby runtime and the AsciiDoctor tool we can choose to use a Docker image from Docker Hub and run AsciiDoctor as a container process.

Fast, consistent delivery of applications

Using the containerized AsciiDoctor tools

```
docker container run --rm \  
-v ${PWD}:/documents/ \  
asciidoctor/docker-asciidoctor \  
asciidoctor-pdf /documents/index.adoc (1)
```

- 1 Run the `asciidoctor-pdf` command provided by the `asciidoctor/docker-asciidoctor` image and convert the `index.adoc` file mounted as a container volume into a PDF file.

Fast, consistent delivery of applications

- Using the same approach and concepts we can run any tool from a chain of build tools as a container
 - Compiler
 - Code Analysis
 - Test software
 - Packaging
 - ..

Running more workloads on the same hardware

- Docker is lightweight and fast.
- It provides a viable, cost-effective alternative to hypervisor-based virtual machines
 - utilize more of an infrastructure's compute capacity.
 - However, containers and virtual machines are not competing technologies. Containers run well on virtual machines

Running more workloads on the same hardware

Setup swarm mode, add node

```
docker swarm init --advertise-addr <IP-address> (1)
docker swarm join --token <worker-or-manager-token> <IP-address>:2377 (2)
```

- 1 Run on a manager node
- 2 Run on a worker or manager node to add a worker or manager. Joining a worker requires a worker token, joining a manager requires a manager token.

! Due to complexity of a swarm setup the code above is pseudo-code. Docker swarm mode with Docker for Mac/Windows can only be used in single node mode (which does not really make sense).

Running more workloads on the same hardware

Create a service and scale it

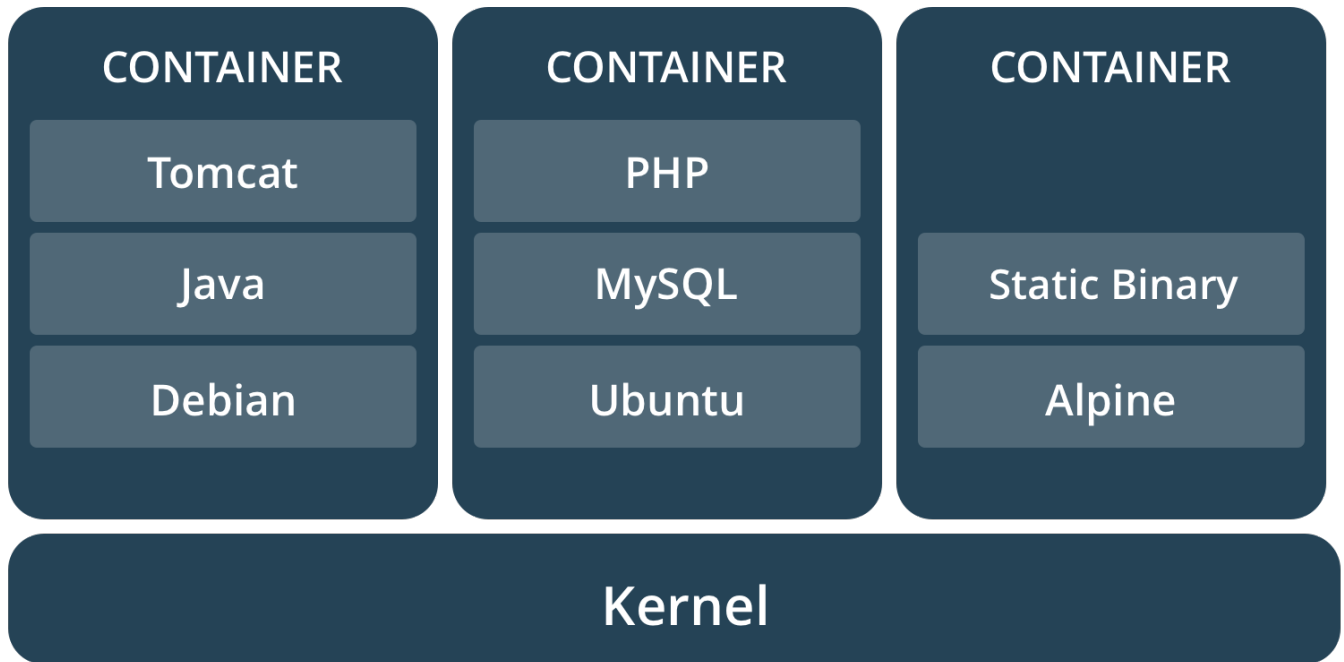
```
docker-machine start swarm-1 # Setup of a swarm not shown here
eval $(docker-machine env swarm-1)
docker node ls
docker service create --name hello alpine sh -c "while true; do echo Hello world"
docker service ps hello
docker service scale hello=3 (1)
docker service ps hello
docker service rm hello
```

Instruct the swarm to run 3 replicas of the service

1

How does a Docker image work?

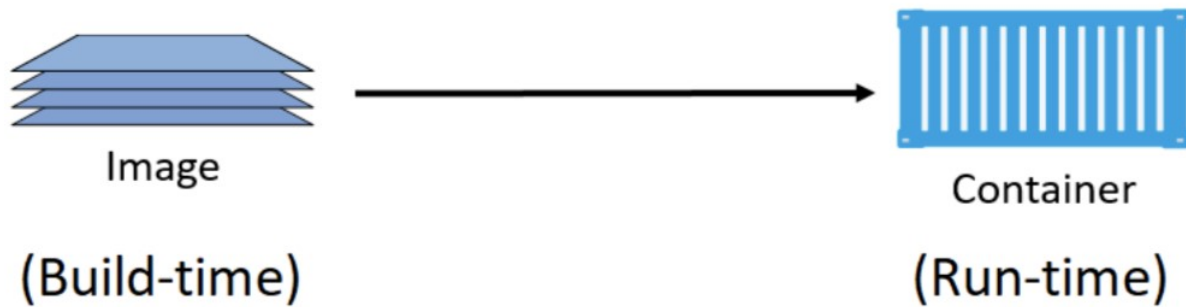
Application Package



What is a Docker image?

- If you're a former VM admin think of Docker images as being like VM templates.
- A VM template is like a stopped VM — a Docker image is like a stopped container.
- If you're a developer think of them as being similar to classes.

What is a Docker image?



An image is made up of multiple layers that get stacked on top of each other and represented as a single object.

Why are Docker images so lightweight?

It's the layers!

- When updating a Docker image, such as when updating an application,
 1. A new layer is built and replaces only the layer it updates.
 2. The other layers remain intact.
- To distribute the update, only updated layers are transferred.
 - Layering speeds up distribution of Docker images.
 - Docker determines which layers need to be updated at runtime.

How are Docker images created?

By committing a container

- We can create a container, install and configure software in an interactive shell
 - `sudo apt-get install apache` (on a Debian based system)
- The container state can be **committed** (persisted) to an image which includes its
 - Filesystem
 - Configuration (metadata)

 Hard to automate!

How are Docker images created?

`'Dockerfile'` script

- An image is defined through a **Dockerfile** containing a set of **instructions**.
- Instructions add
 - image **content** (files)
 - image **metadata**
- ❗ Enables **automated** (scripted) image build

Docker Repository

- A Docker image is **pulled** (downloaded) from a registry or **built** in a local repository
- Repository acts as **local cache** for image layers
 - Avoids repeated loading of an image from a Docker registry (e.g. Docker Hub)
- Repository can manage **multiple versions** of an image
 - Image tag

Demo: Find and pull an image

```
docker search alpine
docker image pull alpine
docker image ls
docker container run -ti --rm alpine sh -c "echo 'hello world'"
```

Demo: Update and commit an image

```
docker container run --name hello alpine sh -c "echo 'hello' > /var/tmp/hello.tx
docker container commit hello myimage
docker container rm hello
docker container run --name hello myimage sh -c "cat /var/tmp/hello.txt"
docker container rm hello
docker image rm myimage
```

Automating Docker images

What is a Dockerfile ?

- Dockerfile contains **instructions** for **assembling** an image.
 - Create files and directories
 - Install packages
 - Configure software
 - Add image metadata
 - ...

A sample Dockerfile

```
FROM openjdk:8-jre-alpine
LABEL maintainer='Martin Ahrer <this@martinahrer.at>'
EXPOSE 8080
CMD ["java", "-jar", "/app.jar"]
ADD app-latest-exec.jar /app.jar
```

1. **FROM**: Specify a **base image**
2. **LABEL**: Add some image metadata
3. **EXPOSE**: Configure networking
4. **CMD**: Configure container startup process
5. **ADD**: Copy content from local filesystem into image

Building an image from a Docker file

- `docker build` builds an image from a Dockerfile and a given build context
- build context is the files at a specified location
 - directory of the **local filesystem** (PATH)
 - location of a **Git repository** (URL)

```
docker build ./
```

```
Sending build context to Docker daemon 6.51 MB
```

```
...
```


Demo: Docker build context

```
ls ./  
./  
../  
Dockerfile  
app-latest-exec.jar
```

Demo: Building Dockerfile

```
docker build ./
```

```
Step 1/5 : FROM openjdk:8-jdk
---> 7c57090325cc
Step 2/5 : LABEL maintainer='Martin Ahrer <this@martinahrer.at>'
---> Running in 881396d1a4eb
Removing intermediate container 881396d1a4eb
---> d2e212116a68
Step 3/5 : EXPOSE 8080
---> Running in 8c7b4fe42e96
Removing intermediate container 8c7b4fe42e96
---> d092044791bf
Step 4/5 : CMD java -jar /app.jar
---> Running in e8e989d3ccc1
Removing intermediate container e8e989d3ccc1
---> cla3208abd2c
Step 5/5 : ADD app-latest-exec.jar /app.jar
```

Instructions

- Use a base image (**FROM**)
- Run a command (**RUN**)
- Add resources (**ADD**, **COPY**)
- Container startup command (**CMD**, **ENTRYPOINT**)
- Declare exposed ports (**EXPOSE**)
- Declare volumes (**VOLUME**)
- ...

For a full reference of Dockerfile instructions see
<https://docs.docker.com/engine/reference/builder/>

Environment

Environment variables (declared with the ENV or ARG statement) can also be used in certain instructions as variables to be interpreted by the Dockerfile.

Environment variables are notated in the Dockerfile either with `$variable_name` or `${variable_name}`.

```
FROM openjdk:8-jre-alpine
LABEL maintainer='Martin Ahrer <this@martinahrer.at>'
EXPOSE 8080
ARG VERSION=latest
ADD app-${VERSION}-exec.jar /app.jar
CMD java -jar /app.jar
```

Labels

- Using the `LABEL` instruction, a Dockerfile can add arbitrary metadata to an image

Example using recommendations from <http://label-schema.org>.

```
LABEL org.label-schema.vendor="Software Craftsmen Gmbh & Co KG" \  
      org.label-schema.name="Docker Training" \  
      org.label-schema.usage="${vcs_url}" \  
      org.label-schema.vcs-url=$vcs_url \  
      org.label-schema.vcs-branch=$vcs_branch \  
      org.label-schema.vcs-ref=$vcs_ref \  
      org.label-schema.version=$version \  
      org.label-schema.build-date=$build_date
```

Multistage build

Work in progress ...

Dockerfile Best Practice

A Container should be ephemeral



The container produced by the image defined by a Dockerfile defines should be as ephemeral as possible. By “ephemeral,” we mean that it can be stopped and destroyed and a new one built and put in place with an absolute minimum of set-up and configuration

— *https://docs.docker.com/develop/develop-images/dockerfile_best-practices/*

Run only one process per container

- Single responsibility principle
- In almost all cases, you should only run a single process in a single container.
- Decoupling applications into multiple containers makes it much easier to scale horizontally and reuse containers.

Immutable Server



We treat all our virtual servers as immutable. When we upgrade our system we create brand new servers and destroy the old ones, rather than upgrading them in-place. This is a logical extension of the phoenix server approach in which servers are regularly recreated from scratch.

— *<https://www.thoughtworks.com/insights/blog/rethinking-building-cloud-part-4-immutable-servers>*

Container Sizing

~~Keep image sizes as small as possible~~

- Use a `.dockerignore` file
- Avoid installing unnecessary packages.
- Run only one process per container → single responsibility

Container Sizing

Keep image sizes as small as possible

- Minimize the number of layers

Build cache

- During the process of building an image Docker will step through the instructions in your Dockerfile executing each in the order specified.
- As each instruction is examined Docker will look for an existing image in its cache that it can reuse, rather than creating a new (duplicate) image.
- If you do not want to use the cache at all you can use the `--no-cache=true` option on the docker build command.

Build cache

Example for Debian package manager

- Always combine `RUN apt-get update` with `apt-get install` in the same **RUN** statement.
- Using `apt-get update` alone in a `RUN` statement causes caching issues and subsequent `apt-get install` instructions fail.

```
RUN apt-get update && apt-get install -y \  
    package-bar \  
    package-baz \  
    package-foo
```

More Best Practice Suggestions

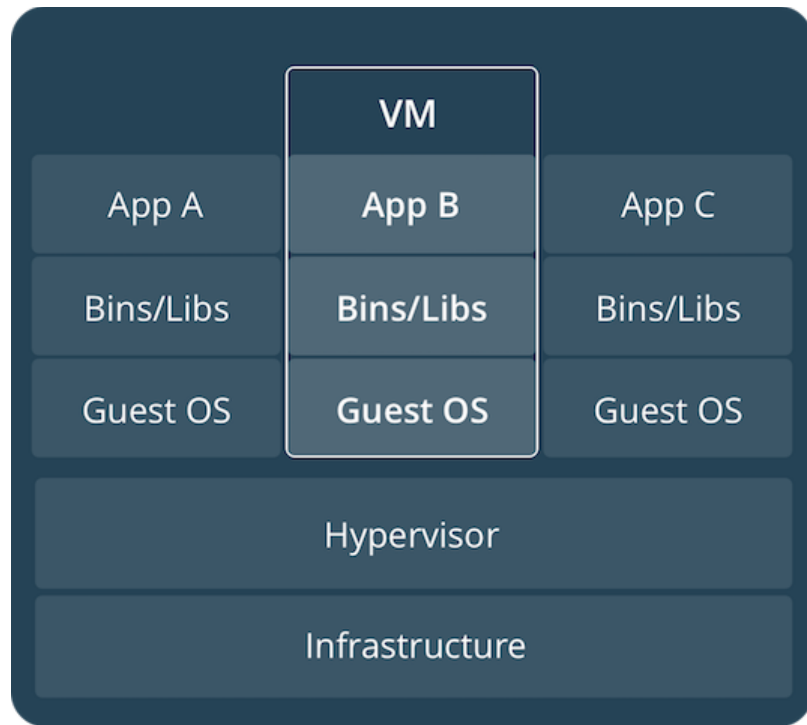
https://docs.docker.com/articles/dockerfile_best-practices/

Comparing Containers and Virtual Machines

Key differentiations

- Containers and virtual machines have similar resource isolation and allocation benefits, but function differently
 - Containers virtualize the operating system
 - Hypervisors virtualize hardware
- Containers are more portable and efficient.

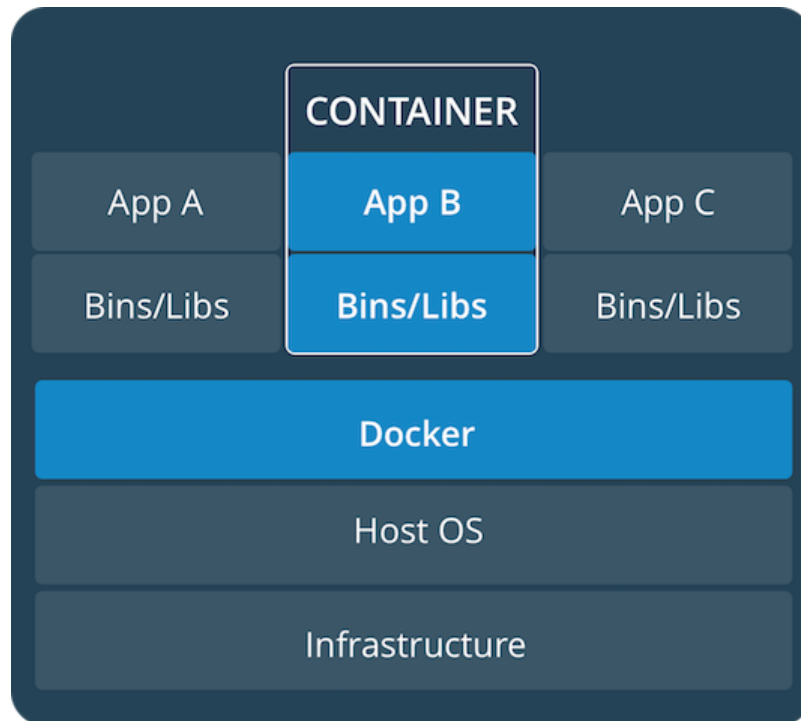
Virtual Machine



Virtual Machine

- Virtual machines (VMs) are an **abstraction of physical hardware** turning one server into many servers.
- The hypervisor allows multiple VMs to run on a single machine.
- Each VM includes a full copy of an operating system, one or more apps, necessary binaries and libraries - taking up tens of GBs.
- VMs can also be slow to boot.

Container

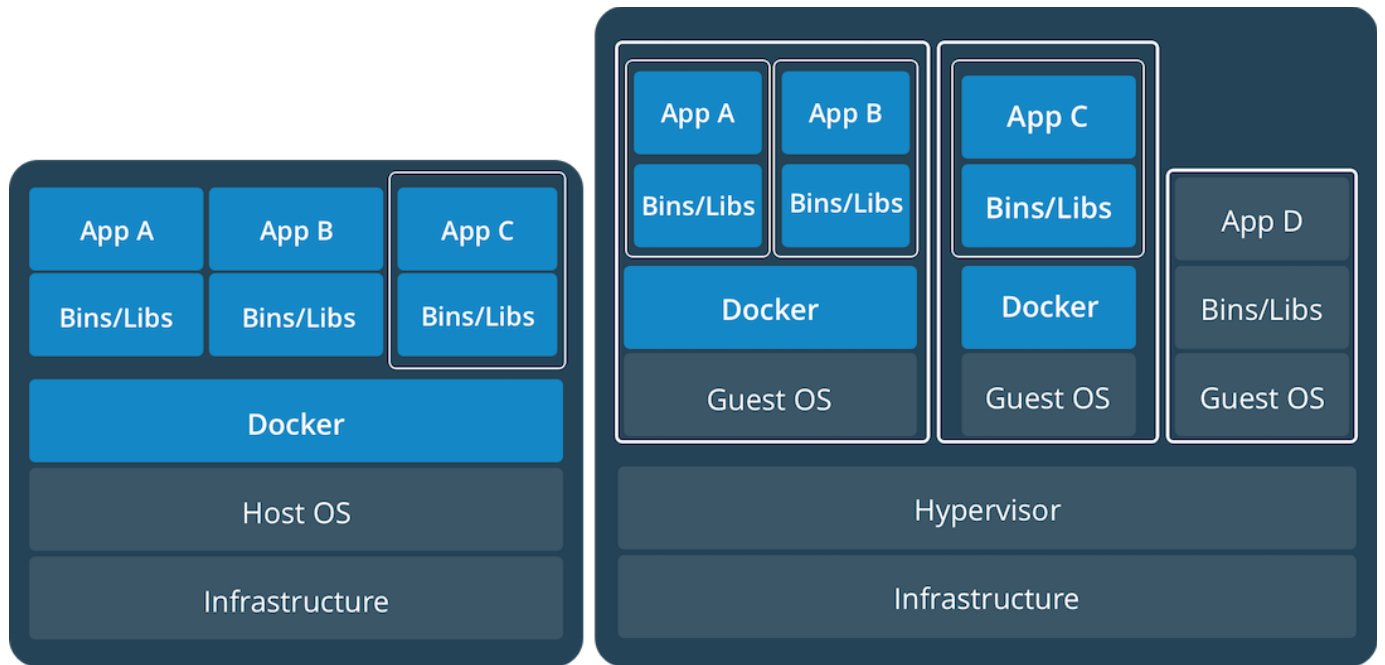


Container

- Containers are an **abstraction at the app layer** that packages code and dependencies together.
- Multiple containers can run on the same machine and **share the OS kernel** with other containers, each running as isolated processes in user space.
- Containers take up less space than VMs (container images are typically tens of MBs in size), and start almost instantly.

Containers and Virtual Machines

can coexist



How does a container work?

What is a Docker Container?

- A container is the runtime instance of an image.
- In the same way that we can start a virtual machine (VM) from a virtual machine template, we start one or more containers from a single image

What is a Docker Container?

- We can **run/create**, **start**, **stop**, or **remove** a container using the **Docker API** or **CLI commands**.
- When running/creating a container, we can add configuration such
 - networking
 - environment variables
 - sharing volumes

What is a Docker Container?

- Each container is an **isolated** and **secure** application platform, but can be given access to resources running in a different host or container, as well as persistent storage or databases.
- Docker on Linux leverages most of the common Linux security technologies.
 - namespaces,
 - control groups (cgroups),
 - capabilities,
 - mandatory access control (MAC) systems, and
 - seccomp (secure computing mode)

.Namespaces

- * pid: Process isolation (PID: Process ID).
- * net: Managing network interfaces (NET: Networking).
- * ipc: Managing access to IPC resources (IPC: InterProcess Communication).
- * mnt: Managing filesystem mount points (MNT: Mount).
- * uts: Isolating kernel and version identifiers. (UTS: Unix Timesharing System).

.cgroups

- * A cgroup limits an application to a specific set of resources.
- * E.g. limit the memory, CPU available to a container

Container filesystem

- When a container is created from an image, its filesystem is recreated from the layers of the image
- The filesystem composed from those layers is presented by a **UnionFS** such as AUFS, btrfs, overlay2, devicemapper, ..
- The layers constructed from the image are read only
- Each container has a read/write layer as top most layer
 - A container can have **persistent data**
 - Filesystem driver implements **copy-on-write (CoW)** which copies a file from an image layer to the r/w layer before attempting to write

Container commands

Running a container

run

```
docker container run -d alpine sh -c "while true; do echo Hello; sleep 10; done"
```

1. The Docker client sends the run command to the Docker daemon
 2. The Docker daemon searches for the image in the local cache
 3. If the image is not found it is pulled from the Docker Hub (or generally a registry)
 4. The Docker daemon creates a container and starts it
- ❗ A container is running until its container command terminates!

Running a container

run

```
CONTAINER=$(  
docker container run -d alpine sh -c "while true; do echo Hello; sleep 10; done"  
)
```

- ❗ The `docker run` command returns the **id** of the created container. We can use this to keep track of a container and control it.

Showing all containers

ls

```
CONTAINER=$(docker container run -d alpine sh -c "while true; do echo Hello; sleep 1; done")  
  
docker container ls
```


Managing life-cycle

`stop`, `restart`, `kill`, ...

```
docker container stop ${CONTAINER}
docker container start ${CONTAINER}
docker container restart ${CONTAINER}
docker container kill ${CONTAINER}
docker container rm ${CONTAINER}
```

- `stop` is allowing the container some time to shut down gracefully. A Linux/POSIX signal SIGTERM is sent to the PID 1 process inside the container.
- `kill` is sending a signal SIGTERM to the PID 1 process inside the container.

Following console output

logs

```
CONTAINER=$(docker run -d alpine sh -c "while true; do echo Hello; sleep 10; done")  
docker container logs ${CONTAINER}
```

Inspecting a container

inspect

```
#!/usr/bin/env bash
CONTAINER=$(docker container create alpine sh -c "echo Hello") (1)

docker container inspect ${CONTAINER} (2)
docker container inspect -f '{{ .NetworkSettings.IPAddress }}' ${CONTAINER} (3)
docker container rm ${CONTAINER}
```

- 1 `docker create` is used instead of `docker run` as we don't have to run it for inspection.
- 2 Inspect all of the container's configuration
- 3 Inspect all of the container's network configuration

Show the running processes of a container

top

```
CONTAINER=$(docker container run -d alpine sh -c "while true; do echo Hello; sleep 1; done")
docker container top ${CONTAINER}
```

Executing a command in a container

exec

```
CONTAINER=$(docker container run -d alpine sh -c "while true; do echo Hello; sleep 1; done")
docker container exec -ti ${CONTAINER} sh -c "ps -elf" (1)
```

- 1 Execute the command `sh -c ps -elf` inside the container. This will effectively list all processes inside the container

Self-healing containers with restart policies

- always
- unless-stopped
- on-failed:

```
docker run --restart always ...
```

Running a Web Application

Dockerfile

```
FROM openjdk:8-jre-alpine
LABEL maintainer='Martin Ahrer <this@martinahrer.at>'
EXPOSE 8080
ARG VERSION=latest
ADD app-${VERSION}-exec.jar /app.jar
CMD java -jar /app.jar
```

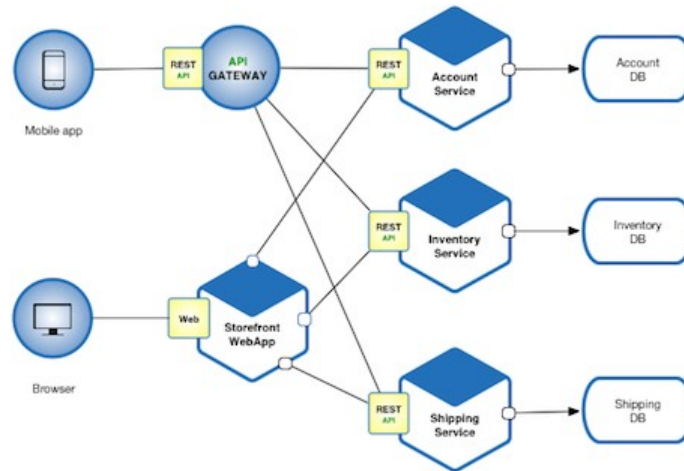

Building and running the container

```
docker image build --build-arg VERSION=latest -t continuousdelivery .  
#tag::run[]  
CONTAINER=$(docker container run -d -p 80:8080 continuousdelivery) (1)  
#end::run[]  
docker container logs $CONTAINER
```

Docker Container Networking


Current software architectures require flexible networking

- Cloud/micro services architectures lean towards a model of distributing services to larger numbers of hosts.
 - Horizontal scaling for work load handling multiplies hosts.



Management of IP Addresses and Ports

- A typical application consists of some HTTP server and one/multiple database server(s).
- Scaling services turns into massive multiplication of required IP addresses and ports
- By offering automatic IP address, port assignment and software defined networks (SDN), Docker greatly simplifies networking.

 In general, with cloud computing we try to avoid working with IP addresses and use service names.

Container Names

- Docker uses generated identifiers for identifying containers.
- These are a bit unhandy so Docker can assign nice names
 - explicitly (`--name` argument)
 - automatically by selecting names randomly

```
CONTAINER=$(docker container create alpine sh -c "echo 'Hello'")
docker ps -a --format "{{.Names}}"
docker rm -v $CONTAINER
```

Port Assignment

A Docker container is making ports visible only when they have been exposed before (by using the instruction `EXPOSE`).

Explicit port publishing

```
CONTAINER=$(docker container run -d -p 80:8080 continuousdelivery) (1)
```

1

`-p 80:8080` binds the local (Docker host) port `80` with the container port `8080`.

Automatic port publishing

We can instruct the docker engine to scan for a free port and publish it using the `run` option `-P` (no arguments) instead of `-p`.

The default bridge

- Docker creates a default bridge named `bridge` that is connected to bridge `docker0` (on Linux)
- Each container is connected to the bridge by default
- Containers are isolated from each other
- Containers are isolated from the host
- We can enable communication between containers by
 - binding containing ports to the host
 - using (deprecated) container links
 - using a software defined network (SDN)

Software Defined Network (SDN)

Network drivers

bridge

- The **default** network driver.
- If you don't specify a driver, this is the type of network you are creating.
- Bridge networks are usually used when your applications run in standalone containers that need to communicate.

Network drivers

host

- For standalone containers
- **Removes network isolation** between the container and the Docker host, and use the host's networking directly.

Network drivers

overlay

- Overlay networks **connect multiple Docker daemons (hosts)** together and enable swarm services to communicate with each other.
- You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons.
- This strategy removes the need to do OS-level routing between these containers.

Network drivers

macvlan

- Macvlan networks allow you to assign a MAC address to a container, making it **appear as a physical device** on your network.
- The Docker daemon routes traffic to containers by their MAC addresses.
- Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack.

Network drivers

none

- **Disable all networking.**
- Usually used in conjunction with a custom network driver.
- Not available for swarm services.

Docker embedded DNS server

- Docker daemon runs an embedded DNS server to provide automatic service discovery for containers connected to user defined networks.
- Name resolution requests from the containers are handled first by the embedded DNS server.
- If the embedded DNS server is unable to resolve the request it will be forwarded to any external DNS servers configured for the container.

Demo: Networking

```
docker network create -d bridge hello
CONTAINER=$(docker container run -d --name world alpine /bin/sh -c "while true;
docker network connect hello world
docker container run --rm --network hello alpine /bin/sh -c "ping -c 1 world"
docker container stop world
docker container rm world
docker network rm hello
```

Multi Container Management

docker-compose

The Problem

- Cloud/micro service based architectures lean towards a model of distributing services to larger numbers of hosts.
- For horizontal scaling of failover these hosts may be multiplied according to work load.
- Managing containers, their configuration and dependencies becomes hard.

Docker Compose

- Compose is a tool for defining and running multi-container Docker applications.
- With Compose, you use a Compose file to configure your application's services.
- Then, using a single command, you create and start all the services from your configuration

Demo: Jenkins Master

`docker-compose.yml`: Jenkins master

```
master:
  image: softwarecraftsmen/jenkins-master:2.60.1-3
  environment:
    - JAVA_OPTS = "-Djava.awt.headless=true"
    - JENKINS_URL
    - JENKINS_ADMIN_USERNAME
    - JENKINS_ADMIN_PASSWORD
  ports:
    - "${JENKINS_AGENT_PORT}:50000"
    - "${JENKINS_HTTP_PORT}:8080"
  volumes:
    - home:/var/jenkins_home/
```

Demo: Jenkins Agent

`docker-compose.yml`: Jenkins Agent

```
agent:
  image: softwarecraftsmen/jenkins-swarm-agent:0.4
  environment:
    - COMMAND_OPTIONS=-master http://master:8080 -username ${JENKINS_ADMIN_USERNAME}
    - JENKINS_AGENT_WORKSPACE
  depends_on:
    - master
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
    - ${JENKINS_AGENT_WORKSPACE}:/workspace
```

Demo: Running Jenkins

```
export JENKINS_URL=http://$(ipconfig getifaddr en1):8080 (1)
export JENKINS_AGENT_WORKSPACE=${PWD}/jenkins-agent (2)
docker-compose build
docker-compose up -d --scale agent=2 (3)
docker-compose logs
docker-compose down -v
```

- 1 Set environment required for the master service (see `docker-compose.yml`)
- 2 Set the environment required for the agent service (see `docker-compose.yml`)
- 3 Scaling the agent to run 2 instances

Managing development infrastructure

`docker-compose`

Key features for managing infrastructure

- Multiple isolated environments on a single host
- Preserve volume data when containers are created
- Only recreate containers that have changed
- Variables and moving a composition between environments

Multiple isolated environments on a single host

- Compose uses a project name to isolate environments from each other.
- You can make use of this project name in several different contexts
 - on a dev host, to create multiple copies of a single environment
 - on a CI server, to keep builds from interfering with each other, you can set the project name to a unique build number
 - on a shared host or dev host, to prevent different projects, which may use the same service names, from interfering with each other

Preserve volume data when containers are created

- Compose preserves all volumes used by your services.
- When docker-compose up runs, if it finds any containers from previous runs, it copies the volumes from the old container to the new container.
- This process ensures that any data you've created in volumes isn't lost.

Only recreate containers that have changed

- Compose caches the configuration used to create a container.
- When you restart a service that has not changed, Compose re-uses the existing containers.
- Reusing containers means that you can make changes to your environment very quickly.

Variables and moving a composition between environments

- Compose supports variables in the Compose file.
- You can use these variables to customize your composition for different environments, or different users.
- See Variable substitution for more details.

Common Use Cases for docker-compose

- Development environments
- Automated testing environments
- Single host deployments

Shipping Images

Docker Registry

- With Docker we can build images and test those locally by running containers off of them.
- In order to **share images** with other developers or for deployment we have to **publish** those images.
- Docker offers multiple options for publishing images through a **registry**
 - Docker Hub (partially commercial)
 - Docker Registry
 - Docker Trusted Registry (commercial only offering)

Docker Registry

- Many other (commercial) products also offer Docker image registry capabilities
 - Nexus
 - Artifactory
 - Harbor
 - ..

Interacting with a registry

pull

We use the `docker image pull` and `docker image push` commands to work with a registry.

Pulling from docker hub

```
docker image pull hello-world (1)
```

Pulling from a local registry

```
docker image pull localhost:5000/hello-world
```

- 1 A registry command (`push` / `pull`) directed at an image without host (and/or user information) will always address the Docker Hub.

Interacting with a registry

push

```
docker image tag hello-world localhost:5000/hello-world (1)
docker image push localhost:5000/hello-world
```

- 1 Registry location is coded into the image tag, therefore we have to assign an image name including the registry.

Interacting with a registry

Run a local registry

```
docker container run -d -p 5000:5000 --name registry registry:2 (1)
```

! This is not a recommended way of running a registry.

Image identifier

- A **fully qualified image identifier** is composed as `[registry-host/][user]/image[:tag]`.
- **Without** the `registry-host` component in the image identifier, a pull/push is **always targeting the public Docker Hub**.
- When **no tag** is included, the implicit tag `latest` is used.

Docker Hub

The Docker Hub is a public Docker registry hosted by the Docker organization with additional features such as automated builds.

TODO: multi-platform support:

<https://blog.docker.com/2017/09/docker-official-images-now-multi-platform/>

Data Management

Introduction

- A container's filesystem is built from its **read only image layers** using a Union File System.
- An **extra read/write layer** is put **on top** for allowing the container to make changes to the file system.
- When a **container is deleted this r/w layer is deleted**, container data is lost.

Data Volume

- A **data volume** is a specially-designated directory within one or more containers that **bypasses the Union File System**.
- Data volumes are designed to **persist data, independent of the container's life cycle**.
- Docker therefore never automatically deletes volumes when you remove a container, nor will it “garbage collect” volumes that are no longer referenced by a container.

Data Volume Characteristics

- Volumes are initialized when a container is created.
 - If the container's base image contains data at the specified mount point, that existing data is copied into the new volume upon volume initialization.
 - Later updates to the image will not be reflected in the volume
- Data volumes can be **shared and reused** among containers.
- A Data **volume persists** even if the container itself is deleted.
- Changes to a data volume are made directly.
- Changes to a data volume will not be included when you update an image.

Creating and using a volume

```
#!/usr/bin/env bash
docker container run --name hello -v /var/tmp alpine \
    sh -c "echo 'hello world' > /var/tmp/hello.txt" (1)
docker container stop hello 1> /dev/null
docker container run --rm --volumes-from hello alpine \
    sh -c "cat /var/tmp/hello.txt" (2)
docker container rm -v hello 1> /dev/null
```

- 1 `-v` mounts an *anonymous* volume at `/var/tmp`
- 2 `--volumes-from` mounts volumes from another container

Named volume

```
#!/usr/bin/env bash
docker container run --rm -v hello:/var/tmp alpine \
    sh -c "echo 'hello world' > /var/tmp/hello.txt" (1)
docker container run --rm -v hello:/var/tmp alpine \
    sh -c "cat /var/tmp/hello.txt" (2)
docker volume ls -f name=hello
docker volume rm hello
```

- 1 **-v** **creates and mounts** a *named* volume `hello` at `/var/tmp`. **Pay attention to the fact that the container is actually removed after termination.**
- 2 **-v** now **only mounts** the *named* volume `hello` at `/var/tmp` as it already exists.

Host directory (file) volume

```
#!/usr/bin/env bash
docker container run --rm -v /tmp:/var/tmp alpine \
    sh -c "echo 'hello world' > /var/tmp/hello.txt" (1)
cat /tmp/hello.txt
```

1 `-v` mounts a host directory volume at `/var/tmp`. We can also mount just a file.

! Mounting host files/directories when working with Docker for Windows/macOS requires extra attention.

Cleanup of volumes

- Docker will not warn you when removing a container without providing the `-v` option to delete its volumes.
- If you remove containers without using the `-v` option, you may end up with “dangling” volumes; volumes that are no longer referenced by a container.
- You can use `docker volume ls -f dangling=true` to find dangling volumes, and use `docker volume rm <volume name>` to remove a volume that’s no longer needed.

Shared Storage

- Volumes are local storage.
- In a cloud infrastructure where containers are moved around, shared storage is required.
- Some **Docker volume plugins** allow to provision and mount shared storage, such as iSCSI, NFS, or FC.

Shared Storage

Using the volume driver **flocker**

```
docker volume create -d flocker -o size=1GB hello
docker run --rm -v hello:/var/tmp buildpack-deps:latest \
    /bin/bash -c "echo 'hello world' > /var/tmp/hello.txt" (1)
docker run --rm -v hello:/var/tmp buildpack-deps:latest \
    /bin/bash -c "cat /var/tmp/hello.txt" (2)
docker volume ls -f name=/hello
docker volume rm hello
```

Writing to volume hello

1

Reading from volume hello

2

Machine Provisioning

Overview

- Creating a (virtual/remote) host and provisioning it with a docker engine requires a series of complex setup tasks
 - Install operating system
 - Install Docker engine
 - Setup SSH
 - Setup TLS certificates for securely connecting a Docker client



Docker Machine

- The Docker eco system provides tooling for **provisioning** Docker hosts
 - Local Docker (Oracle VirtualBox, Microsoft Hyper-V, local Docker Daemon)
 - Data center (OpenStack, VmWare, ...)
 - Cloud Docker (AWS, Digital Ocean, Google Compute Engine, ..)
 - Swarm clusters

Operations

- **Start**, inspect, **stop**, and restart a machine
- Perform a **software upgrade** of Docker client und daemon.
- **Connect a Docker client** with a Docker host (configures a Docker client)
- ..

Provisioning a machine

Oracle VirtualBox

```
machine=$(docker-machine ls --filter "name=swarm-1" --format "{{.Name}}")
if [ "${machine}" != "swarm-1" ];
then
    docker-machine create --driver virtualbox swarm-1 (1)
else
    docker-machine start swarm-1 (2)
fi
eval "$(docker-machine env swarm-1)" (3)
docker system info
docker-machine stop swarm-1 (4)
```

- 1 Create machine named `swarm-1` (if not exists)
- 2 Start machine named `swarm-1`
- 3 Set the Docker CLI to work with machine named `swarm-1`
- 4 Stop machine named `swarm-1`

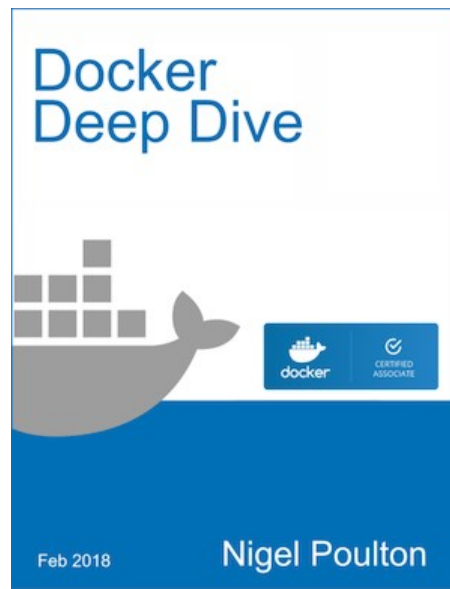
Provisioning a machine

Amazon Web Services (AWS) EC2

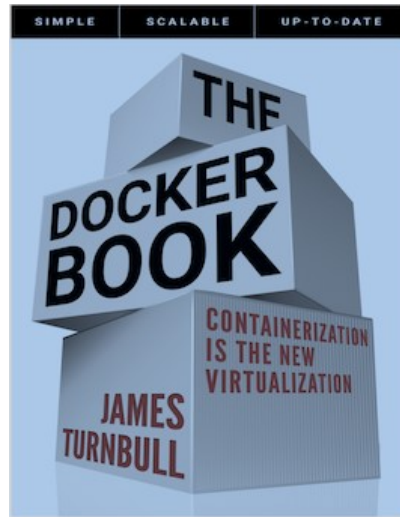
```
docker-machine -D create \  
  --driver amazec2 \  
  --amazec2-access-key $AWS_ACCESS_KEY_ID \  
  --amazec2-secret-key $AWS_SECRET_ACCESS_KEY \  
  --amazec2-vpc-id $AWS_VPC_ID \  
  --amazec2-zone b \  
  --amazec2-region eu-central-1 \  
  swarm-1  
eval "$(docker-machine env swarm-1)"  
docker system info
```

Resources

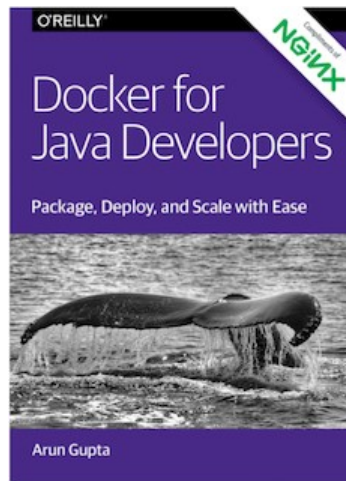
Books



Books



Books



Questions?

