Container Zen: 12 Steps to Enlightenment

Nils Magnus
System Architect and Community Engineer
LinuxTag Association and Open Telekom Cloud

magnus@linuxtag.org
Container

One term, several concepts in our minds
Powerful tools ...

... but communication is key.

Objective to create a shared, agreed terms to talk about similar concepts.
Practice is the Path to Enlightenment

Gautam Buddha

L16 Oxycryocrypt
Nils

- Security engineer, system architect
- tech journalist, and conference organizer
- Slightly more ops than dev
- Volunteer for LinuxTag and German Unix Users Group
- Working as community liason for Open Telekom Cloud
- Researching containers since 2013
- PoCs, articles, training
- Designing enterprise architecture components
Goal to achieve Level 1

Starting your first container makes you appreciate seemingly cheap virtualization.
Get started

- Install and run Docker in 42 seconds!
- That was pretty easy
- But it is actually just the start
Goal to achieve Level 2

Eagerly studying, you dive into fundamental knowledge like namespaces, capabilities, or process isolation.
Core concepts

- Fundamentals of the process model of Linux/Unix
- Chroot environments: Lock in a process into a subfolder file system
- Namespaces: A clever way to “virtualize” important system resources:
  - process and user IDs
  - file system
  - network
  - clock, hostname, stuff
- Cgroups (if you need them)
- Seccomp and Apparmour for security context separation
- Containers share a single kernel: Separation vs. economic resource utilization
Goal to achieve Level 3

You discover the challenges of networking and feel the pain of storage mapping.
Increasing hosts from a network perspective

**Old world**
- Several subsystems on a single host
- One or several programming frameworks / languages
- Application server + code artifacts
- Mail server etc.
- Interactive user logins

**New challenges**
- More IPs needed (internal view)
- Discovery of services

**Containerized world**
- One container, one single purpose
- Just the application
- No need for infrastructure inside
- No longer necessary, to actually log in

**New Tasks**
- Mapping public to private IP addresses
- Don’t reimplement the old world!
Goal to achieve Level 4
You learn how to automate image creation with Dockerfiles.
Automate all the things

- It is possible to start with a plain containers (Ubuntu, Red Hat, CoreOS, Alpine, …)
- ... then install the software you need ...
- … then configure your setup ...
- and finally make a snapshot:
  
  docker commit -a 'Nils Magnus' d692399ae00b my-better-ubuntu

Configuration as Code: Dockerfile

- Basically a shell script of steps how to build your image
- A bit of metadata (including networking hints for the runtime)
- Very easy-to-learn DSL that is neither YAML nor JSON
Goal to achieve Level 5

Then you realize there is no need to write Dockerfiles, as images can be pulled from registries.
Why bother?

- Writing Dockerfiles is not complicated, but tedious
- There are several repositories ("registries") that have all in place
- 20,000+ images available in the public Docker Hub

Concern: security

- Who made the images, who wrote the Dockerfiles?
- What is the code/configuration quality?
- How to access registries through corporate proxy cascades?

- Vendors start providing solutions: Security scanners, own registries
Goal to achieve Level 6

You move all your services to containers.
Graduating from apprenticeship

- It’s actually fun to work with containers
  - They allow for quick success
  - They are disposable, so if you make a mistake, just start from the beginning
  - They don’t need many resources
  - … so why not just moving your complete IT into containers?

- Some services have special hardware requirements (databases)
- Some services need special operating systems (Windows, other strange stuff)
- Singleton services (wikis, mailservers, pads, field trip voting servers, you name it) don’t benefit overly from containers, but migrating them in does no harm, either
Goal to achieve Level 7

You acknowledge the need for orchestration, since your zoo of scripts easily loses control over your containers.
Starting, Scaling, and Discovering

- Application consisting of a single container with no dependencies are easy to manage manually on the command line.

- But most services depend on each other:
  - they need an order to start,
  - there is more than a single instance involved,
  - IP addresses are no longer fixed and thus move.

- It’s tempting to write little helper scripts that automate these tasks.

- It’s easy to underestimate the long tail.
Orchestration

- Selecting and using a framework is a major step for container adoption!
Goal to achieve Level 8

You spread your containers over many servers to make your services less dependent on hardware.
Scaling of runtime environments

- Since containers share a kernel, many of them can run on the same host.
- Several dozens of concurrent container instances are possible.
- However, vertical scaling of the host has limits.

- Usually the container platform takes care of this link to the IaaS management.
Goal to achieve Level 9

You negotiate consensus with your containers.
Agreement in a distributed world

- Consensus about a global state is crucial for many components
  - On which host can I find IP a.b.c.d?
  - Where should I ask for service X? How many instances are associated with it?
  - Which node in a cluster is authoritative to assign new resources?

- In an ideal world with reliable networks, no outages and data corruption this is easy.
- In reality it isn’t. Change is not the exception, but rather the rule.

- Servers implement RAFT or Paxos protocols
- Usually included in platform
Goal to achieve Level 10

You begin discovering services and working with dynamic management of containers fleets to ensure high availability.
Scale

- Adopt the number of container instances of a service according actual need
- Referencing services by name, not by IP or server name (**service discovery**)

**Loadbalancing**
- determining which metric to watch
- launching new instances if necessary, destroying idle instances no longer needed
- routing client requests to participating nodes in the cluster

**High availability, resilience**
- keeping a certain degree of redundancy
- restarting failed nodes

**Updates:** A/B testing, rolling updates, user acceptance tests, ...
Goal to achieve Level 11

You turn your software into stateless, ephemeral containers.
Failure as Normal Condition

- Building better components is desirable, but very expensive
  - two servers that **sync on hardware level** cost a fortune
  - two **commodity servers** with proper fail-over might provide a similar service

- Availability is important at the edge, not at the components

- The main enemy in clusters is **state**, because it’s difficult to replicate
  - applications with as much state removed are called **ephemeral**, since they are able to come and go
  - Applications without any state at all are kind of “boring”. Thus the necessary business data is **outsourced to explicit data stores** that have better strategies

- No need to monitor services inside your container. If they fail terminate the instance
Goal to achieve Level 12

You refactor your monolithic applications into microservices.
Microservices

- Most of the previous steps are more or less engineering tasks
- The last steps requires creative architectural and domain knowledge

- The granularity of a design is a non-trivial task
  - with too “small” services the overhead for management, accounting, authentication becomes too big
  - with too “big” services you approach the monolithic anti-pattern again

- The twelve-factor app is a good methodology to start with.
- Stepwise refinement is an agile way to start while keeping a working product increment.
Checklist and Wrap-Up

- I know how to install and start a container.
- I know how the basic blocks of container technology work.
- I can write a Dockerfile.
- I know how to pull images from a registry and also know about its threats.
- I moved almost all of my services into containers and now wonder if it was worth it.
- I acknowledged that running my services with scripts doesn’t scale. I want orchestration.
- One host is no longer enough. I made provisions to scale over server boundaries.
- I understand that consensus is important in a distributed system.
- I have implemented scaling measures to deal with varying workloads.
- I removed as much state from my containers and moved it into databases or other persistent storage.
- I embraced the microservice pattern and envisioned the future architecture of my application.

Welcome to the Enlightened!
Thank You!

Any questions?

Nils Magnus
magnus@linuxtag.org