



WENN ICH DAS NUR VORHER GEWUSST  
HÄTTE!

# Kubernetes für Entwickler

Dr. Stefan Schlott (BeOne Stuttgart GmbH)

# ABOUT.TXT

Stefan Schlott, BeOne Stuttgart GmbH

Java-Entwickler, statische-Sprachen-Fan, Linux-  
Jünger

Seit jeher begeistert für Security und Privacy

Herr der Container (im Projekt)





# KUBERNETES!

Oh brave new world...

GUTEN TAG!

ICH BIN IHR NEUER  
ENTWICKLUNGSLEITER!

AB MORGEN MACHEN WIR ALLES  
MIT KUBERNETES!





## KEINE SORGE!

Sie müssen nur ihre bestehende  
Anwendung in einen Container  
packen, das ist alles!

# KUBERNETES IN A NUTSHELL

Ein kurzer Überblick

# ZUSTÄNDIGKEITS- MATRJOŠCHKA

Be1ne  
s u t t g a r t





# WAS IST WAS?

Container: Hoffentlich klar :-)

Pods: Ein oder mehrere Container. „Atomare Einheit“ in  
Kubernetes

Workload: Verwaltet Pods

Übliche Workloads: Deployment, StatefulSet

# WER MANAGT WAS?

Node → Container (Restarts, etc.)

Scheduler → Pods (Zuteilung auf Nodes)

Cluster Controller → Workloads (Skalierung, Rolling Updates,  
Neue Pods bei Ausfall von Node, etc.)

# TYPISCHE WORKLOADS

## DEPLOYMENT

Stateless

Einfacher zu handlen

## STATEFULSET

1:1-Verknüpfung mit Pod-individuellem Storage

# SERVICES

„Kleber“ zwischen (Gruppen von) Pods und einem  
(DNS-)Namen

Abstrahiert über „mehrere einer Sorte“ o.ä.

Ansprechbar als Hostname

Mapping: Hostname + Port + Protokoll → Port im Container

# LABELS

Kubernetes Object:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: foobar
  labels:
    app: foobar
    networking/ingress-allow: true
spec:
  ...
```

# SELECTORS

Selector wählt Ziel von Regeln

z.B. Service:

```
apiVersion: v1
kind: Service
metadata:
  name: foobar-service
spec:
  type: ClusterIP
  ports:
    - ...
  selector:
    app: foobar
```

Service zeigt auf alle Pods mit dem Label app=foobar



# PODS

Der große Container-Zirkus

# PODS

Kleinste von Kubernetes gemanagte Einheit

Ein oder mehrere Container

Laufen auf derselben Node

Selbe IP-Adresse (selber Network-Namespace)

Auch IPC / Shared Memory zwischen Containern möglich

Gemeinsames Nutzen von Volumes



# MODULBAUKASTEN

Jeder Pod: „Macht ein Ding“

→ nur eine Anwendung, etc.

→ Skalierung, Redundanz, ...: Anderer Mechanismus!

Also warum mehrere Container?

Teilfunktionen separat - und ggfs an mehreren Stellen  
wiederverwenden

# BEISPIEL: PROMETHEUS

„Hauptanwendung“: nginx

Gewünscht: Statistiken per Prometheus exportieren

„Klassischer Weg“: nginx-Container „ableiten“ und Prometheus-  
Exporter dazubauen

„The Pod way“: Zweiter App-Container mit dem Prometheus-  
Exporter

# INIT-CONTAINER

Naming: Init-Container vs App-Container

Container, die vor dem Start der App-Container ausgeführt werden

Der Reihe nach

Ohne Timeouts, erzwungene Restarts, etc.

Beispiel: Liquibase-Datenbank-Migrationen durchführen

# DEBUGGING

In Container verbinden: `kubectl exec -it podname /bin/sh`

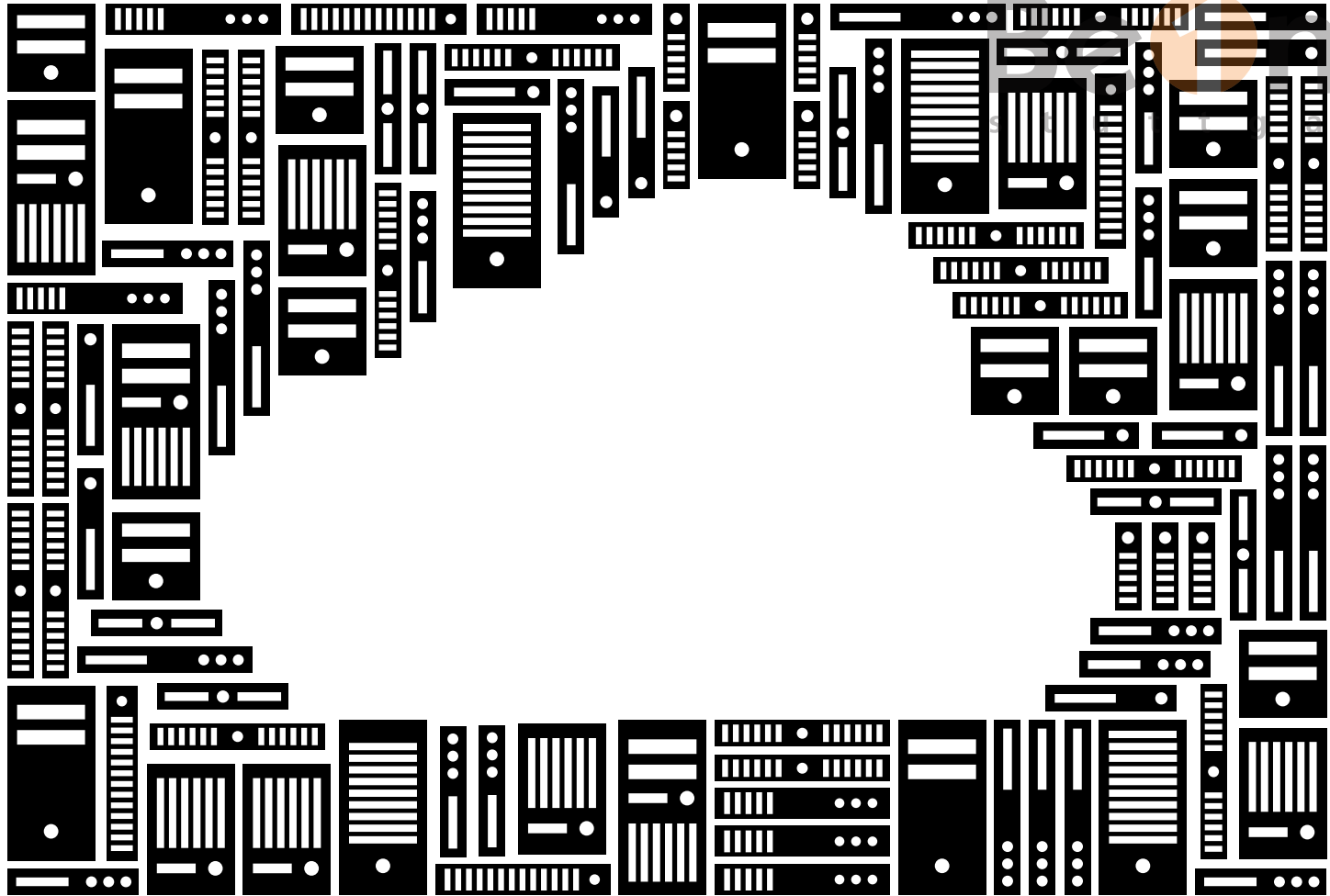
Port Forwarding: `kubectl port-forward podname localport:remoteport`

Ephemeral Containers ohne Namespace-Grenze  
(Kubernetes 1.23, beta)

`kubectl debug -it podname --image=busybox --target=containername`

# 1 RESSOURCEN- LIMITS

Was ist „normal“ in meiner Anwendung?



**There is NO CLOUD, just other people's computers**

# REQUESTS UND LIMITS

...für RAM und CPU

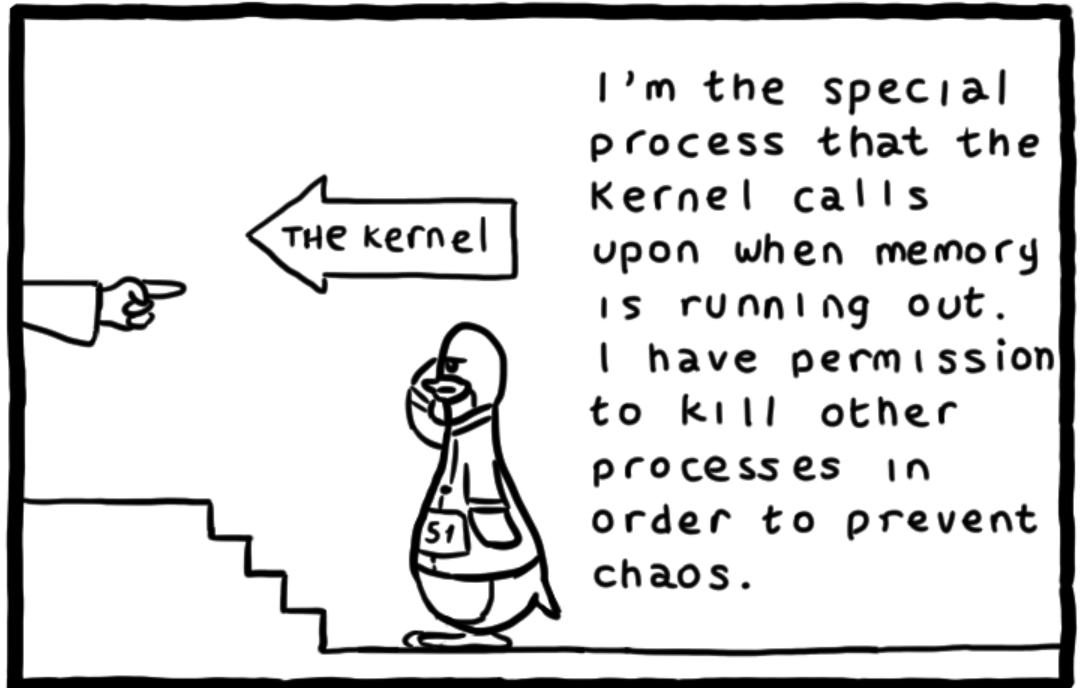
Requests: Mindestgarantie für einen Pod

Kapazitäts-Disposition für Admins

Indikator für Kubernetes (Scheduler, Autoscaler)

Limits: Maximalverbrauch

Schutz vor Amok-laufenden Anwendungen





# JAVA UND DER FREIE SPEICHER

Vor Java 8u181: Berücksichtigt cgroup nicht (JRE bestimmt Heapgröße auf Basis des gesamten RAMs der Node) ↴

Ohne -Xmx und -Xms: JRE bestimmt initiale und maximale Heapgröße selbst

*Aber:* Größen müssen auf Request/Limit abgestimmt sein!

# HEAPGRÖSSE $\neq$ LIMIT

-Xms / -Xmx → Initiale / Maximale Heapgröße

Kubernetes-Ressourcen: *Gesamter* RAM-Bedarf:

Heap, weiterer Speicherbedarf des JRE, aber auch z.B. tmpfs

→ Anwendung kennen, entsprechende Zugabe

# LIMIT $\neq$ GARANTIE

Eine Node kann weniger RAM als das Containerlimit haben!

→  $-Xms = -Xmx = \text{Request (minus Zuschlag)}$

→  $\text{Request} = \text{Limit}$

→ Anwendungsverhalten „kennen“, horizontal skalieren

# QUOTAS

Maximalwert für Summe aller Requests/Limits *in einem Namespace*

(z.B. vom Admin vorgegeben um „gierige Teams“ auszubremsen)

Vorsicht bei Rolling Updates:  
Nicht bis zum Anschlag ausreizen!

# LEBEN UND STERBEN IM CLUSTER



Die großen Fragen im Leben eines Containers

# PROBES

„Statusabfragen“ an den Pod  
(genauer: Die App-Container)

Verschiedene Möglichkeiten: Kommando ausführen, TCP-  
Connect, HTTP-Request, ...

Initial Delay und Interval

Success- und Failure Threshold

# READINESS PROBE

Liefert Information, ob Pod Daten verarbeiten kann

Wenn Failure Threshold überschritten: Keine Zustellung von Requests, etc.

→ Backpressure-Mechanismus, Überlastschutz

# LIVENESS PROBE

Liefert Information, ob Pod korrekt funktioniert

Pod-Neustart, wenn Failure Threshold überschritten

→ „Selbstheilung“ im Cluster



# STARTUP PROBE

„Initial Delay dynamisch“ für Liveness-Probe

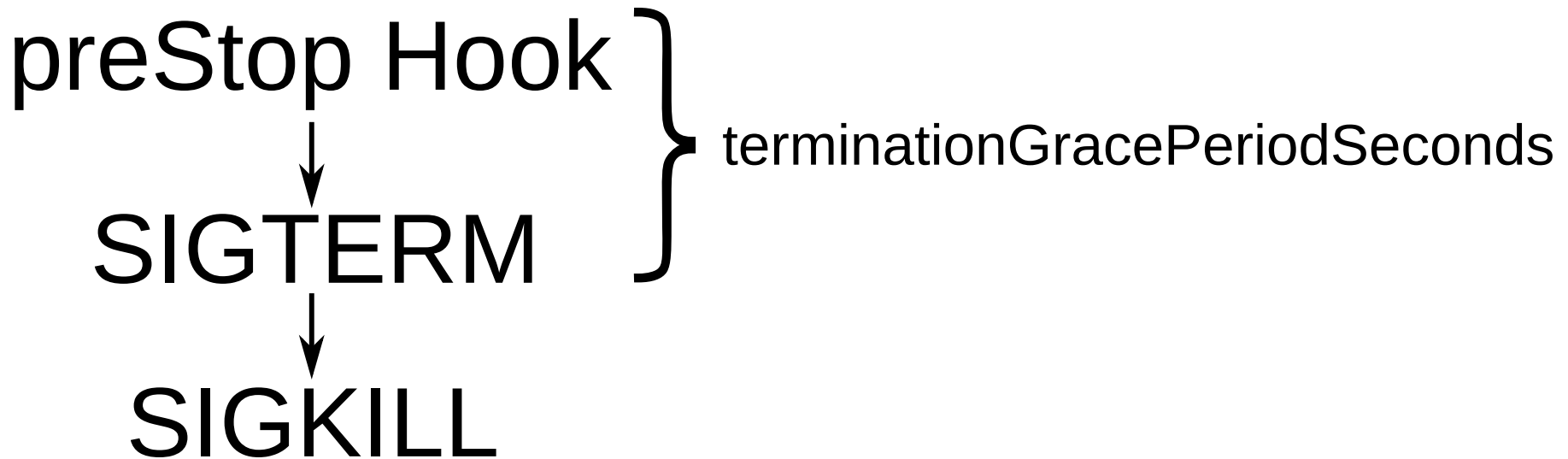
Nach positivem Test der Startup-Probe: Wechsel zur Liveness-Probe

Anwendungsbeispiel: Selber Endpoint wie Liveness-Probe,  
aber deutlich höherer Failure Threshold

→ Init-Container vs. Startup-Probe

# GRACEFUL SHUTDOWN

Chance für Anwendung, sich sauber herunterzufahren:



# SHUTDOWN: SIGNALE

SIGTERM: „Höfliche Aufforderung“ an Anwendung

Kann von der Anwendung abgefangen und behandelt werden

Wird an ENTRYPOINT-Prozess geschickt

(Vorsicht bei Shellskripten - Weiterreichen des Signals?)

SIGKILL: Sofortiges Beenden des Prozess



# ROLLING UPDATES

Kingt cool, aber...

# ROLLING UPDATES

Idee: Zero Downtime

Tuning-Parameter: maxUnavailable und maxSurge

→ Update beginnt durch Stoppen von maxUnavailable alten Pods und Sarten von maxSurge neuen Pods

# FEHLERBILDER

Geht doch! vs Wo ist mein neues Feature!

*Unerschiedliche Stände laufen gleichzeitig*

Schemakompatibilität

(Datenbank-Schema, REST-APIs, etc)



# AUTOSCALING

Wenn die Hölle losbricht

# GRUNDFUNKTION

Controller passt Anzahl Replicas an Metriken an  
(innerhalb von gegebenem Minimum und Maximum)

Durchschnitt der Metrik vs gewünschter Zielwert =  
Skalierungsfaktor

Relative Zielwerte: Basierend auf Resource Request



# FEHLERBILD CPU- INTENSIVER STARTUP

Gut gemeint: Anwendung meldet sich „ready“, macht im Hintergrund noch Dinge

→ Autoscaler detektiert hohe CPU-Last  
Startet weitere Pods

→ Wieder CPU-intensiver Start, Durchschnitt steigt weiter,  
weitere Pods...



FAZIT

# FAZIT



Ganz ohne Rücksicht der Anwendung geht's nicht.

Architektur-Impact:

Splitten in Container (Init-Container!)

Endpoints für Probes

Graceful Shutdown

Informierte Entscheidung bzgl. Rolling Updates

 [stefan.schlott@beone-stuttgart.de](mailto:stefan.schlott@beone-stuttgart.de)

 @\_skyr  @skyr@chaos.social