Alternativen zum Monolithen Kubernetes

Nils Bokermann

Moderne, verteilte Architektur löst Monolithen und große Systeme ab. Damit einhergehend ändert sich auch die Deployment-Architektur. Häufig wird die neue Welt direkt mit Kubernetes in Verbindung gebracht. Dass es neben Kubernetes andere Möglichkeiten gibt, die auch eine iterative Migration ermöglichen, wird in diesem Artikel gezeigt.

Als Erstes muss die Behauptung über den Monolithen Kubernetes klargestellt werden. Es sieht auf den ersten Blick doch so aus, als sei Kubernetes der Inbegriff einer Microservices-Architektur. Auf den zweiten Blick stellt man allerdings fest, dass die einzelnen Services derart verwoben sind, dass man hier eigentlich von einem Deployment-Monolithen sprechen kann.

Diese Architektur führt wiederum dazu, dass die Einführung von Kubernetes zu einer Big-Bang-Migration führt. Dabei wird nicht nur die Software-Architektur verändert, sondern auch viele betriebliche Belange wie Netzwerke, Storage und Deployment. Im klassischen Rechenzentrumsbetrieb werden diese Gewerke durch verschiedene Abteilungen oder Gruppen abgedeckt. Diese Gewerke müssen sich nun an ein gemeinsames Tooling und an Software-defined Networks anpassen.

Geht das vielleicht auch anders? Kann man nicht schrittweise im Rahmen einer iterativen Migration vorgehen? Um es vorwegzunehmen: Ja, man kann – und im Folgenden wird ein möglicher Weg aufgezeigt.

Um die folgenden Schritte in einem betrieblichen Kontext zu verstehen, werden zwei gegensätzliche Betriebskonzepte vorgestellt.

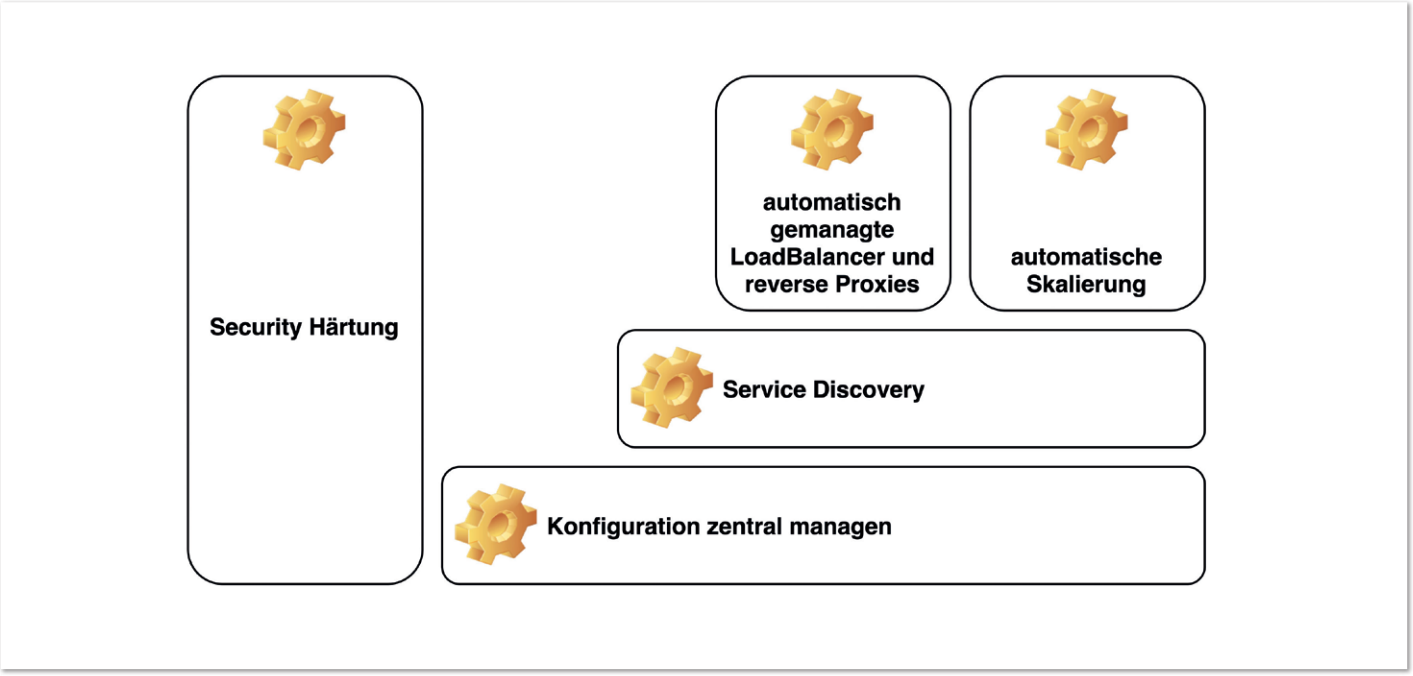

Abbildung 1: Modulübersicht (© Nils Bokermann)

Es soll im Weiteren zwischen einem manuell gemanagten Betrieb und einem automatisch gemanagten Betrieb unterschieden werden. Beim manuell gemanagten Betrieb stehen die Konfiguration und das Deployment-Ziel zum Deployment-Zeitpunkt fest. Damit kann die Konfiguration vor dem Deployment auf das entsprechende System gepusht werden. Im Gegensatz dazu wird bei einem automatisch gemanagten Betrieb das Deployment-Ziel erst während des Deployments festgelegt. Damit ist eine proaktive Verteilung von Konfiguration nicht mehr sinnvoll möglich. Der Service muss also beim Start seine Konfiguration aus einer Konfigurationsdatenbank ziehen. Die beiden Betriebsansätze unterscheiden sich also vornehmlich in der Art des Konfigurationsmanagements. Während der manuell gemanagte Betrieb nach dem Push-Prinzip verteilt, geschieht dies im automatisch gemanagten Betrieb nach dem Pull-Prinzip.

In *Abbildung 1* finden sich die Bausteine, die für eine Transition in einen automatisiert gemanagten Betrieb in Betracht kommen. Als Basis dient die zentrale Verteilung von Konfiguration (Konfiguration zentral managen). Damit kann ein Verteilen von Environment-Dateien auf verschiedene Systeme abgelöst werden. Eine weitere Vereinfachung der Konfiguration kann durch die Einführung einer Service-Discovery-Lösung geschaffen werden. Sind die Informationen über Services im Service Discovery vorhanden, kann mittels dieser Information auch Load-Balancing und Reverse-Proxy konfiguriert werden. Damit ist auch eine dynamische Verteilung von Services auf Systeme und eine automatische Skalierung möglich. Die Security-Härtung wird in diesem Artikel an den Schluss gestellt, obwohl sie von allen anderen Maßnahmen unabhängig ist und demnach auch zu jeder Zeit in der Transition eingeführt werden kann.

Beispiel-Code

Im Folgenden werden immer wieder Code-Snippets gezeigt, die zu einer einfachen Client-Server-Anwendung gehören. Diese Anwendung findet sich in den verschiedenen Transitionsstufen auf GitHub *[1]*.

Konfiguration zentral managen

Im ersten Schritt der Transition wird es darum gehen, die Shell-Environment-Verteilung loszuwerden. Damit sind Konfigurationswerte gemeint wie:

- URLs zu anderen Services,
- Flags,
- Feature-Switches oder
- Credentials

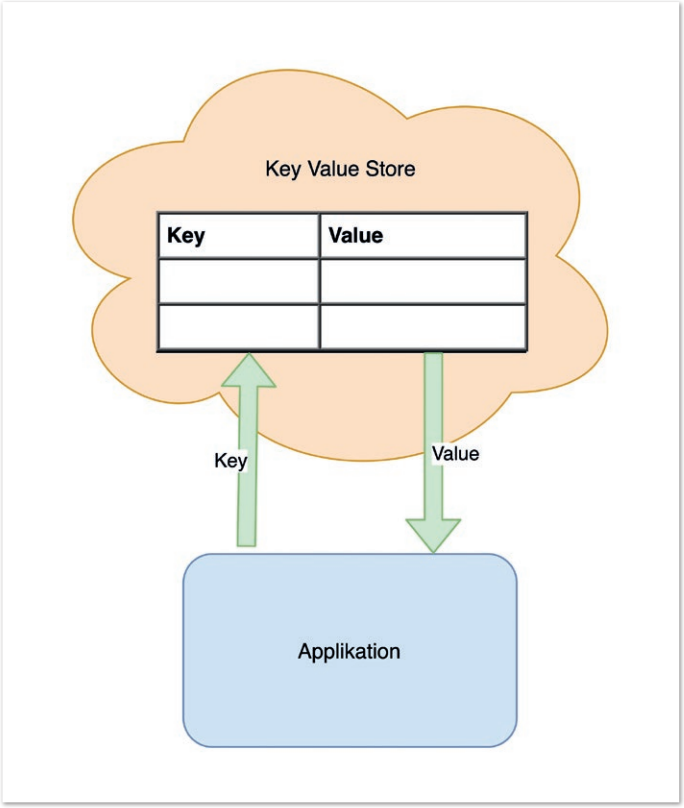

Abbildung 2: Key-Value Store (© Nils Bokermann)

```
public HeroServiceRestClient(
           @Value("${backend.url}") String backendUrl,
           @Value("${backend.username}") String username,
           @Value("${backend.password}") String password) {
     …
}
```

Listing 1: Klassische Value-Injection in Spring Boot

```
public HeroServiceRestClient(
              ConfigurableEnvironment ce) {
    String beUrl = ce.getRequiredProperty("backend.url");
    String username = ce.getProperty("backend.username");
    String password = ce.getProperty("backend.password");
    …
    }
```

Listing 2: Value-Injection mittels ConfigurableEnvironment

Zu diesem Zweck wird ein Key-Value-Store an die Services angeschlossen, in dem die ehemaligen Environment-Variablen zentral gespeichert werden. Zum Abruf wird der Value zu einem bekannten Key angefragt *(siehe Abbildung 2)*.

Als Beispiel für einen Key-Value-Store wird HashiCorp Consul *[2]* genutzt. An diesem Beispiel werden die Umstellungen an Software-Architektur vorgestellt und die Einflüsse auf den Betrieb aufgezeigt. Wenn in der bestehenden (Spring-Boot-)Software die Konfiguration via `application.properties` „injected" wurde, ist nur eine Erweiterung der Dependencies um `spring-cloud-starter-consul-config` notwendig. Es ist allerdings zu Debugging-Zwecken ratsam, zumindest an einer Stelle im Code die Änderungen von *Listing 1* zu *Listing 2* zu übernehmen. Durch die Injection des `ConfigurableEnvironment` kann überprüft werden, welche Datenquellen angezogen werden.

Auf der Seite des Betriebs ändern sich in diesem Schritt mehrere Dinge. Zum einen muss eine neue Software installiert und konfiguriert werden. Zum anderen wird der Verteilungsmechanismus für die Environment-Variablen umgestellt. Eine sinnvolle Verteilung der Daten nach Service-Gruppen und Schutzbedarf sollte in diesem Schritt bedacht oder zumindest begonnen werden.

Wie Ihnen sicher aufgefallen ist, sind die Credentials einfach so mit allen anderen Environment-Variablen umgezogen. Aus Security-Gesichtspunkten sollten diese natürlich einem besonderen Schutz unterliegen. Wenn die Credentials allerdings vorher genau wie alle anderen Variablen im Shell-Environment gelegen haben, ist die Überführung in den zentralen Key-Value-Store immerhin keine Verschlechterung. Existiert jedoch schon eine entsprechender Secret-Store, so sollte dieser auch weiter genutzt werden.

Service Discovery

Unter dem Begriff Service versteht man in diesen Zusammenhang jede Art von Dienst, die über Netzwerk angesprochen wird. Von der Datenbank über REST-Services, Docker-Container, Application-Server und FatJar-Anwendungen.

Um die Service Discovery nutzen zu können, muss sich der Service-Geber (Server) an der Service Registry anmelden (siehe *Abbildung 3*, Schritt 1). Danach kann der Service-Nehmer (Client) auf die Service Registry zugreifen und analog zum Key-Value-Store über den Service-Namen die aktuelle URL und gegebenenfalls Meta-Daten abfragen (siehe *Abbildung 3*, Schritte 2 und 3). Die Nutzung erfolgt dann analog zur Nutzung der statisch konfigurierten URL aus dem vorigen Beispiel.

Da HashiCorp Consul neben dem Key-Value-Store auch die Möglichkeit hat, als Service Registry zu fungieren, wird diese Funktionalität genutzt.

Die Software-Änderung im Bereich des Service-Gebers beschränkt sich, dank Spring-Boot, wiederum auf das Hinzufügen einer Dependency. Es wird einfach das Paket `spring-cloud-starter-consul-discovery` hinzugefügt. Zusätzlich sollte der Health-Check-Endpunkt für Consul gepflegt werden, damit nur Endpunkte verteilt werden, die gestartet und aktiv sind. In *Listing 3* wird diese Konfiguration beispielhaft für Spring-Boot-Actuator gezeigt.

Auf der Service-Nehmer-Seite wird etwas mehr Aufwand nötig. Die Services werden nicht durch das `ConfigurableEnvironment` oder `@Value`-Annotationen „injected", sondern müssen über den DiscoveryClient bezogen werden. In *Listing 4* wird die Service-URL für den Service „Backend" aus dem `DiscoveryClient` bezogen.

Auf der Betriebsseite ist für diese Stufe der Transition wenig zu tun. Der Consul-Service ist bereits im Rechenzentrum verteilt und liefert Daten aus. Auch der Rückbau der statisch konfigurierten URLs kann und sollte erst nach dem Ausrollen der Service Discovery durchgeführt werden. Solange die Services ihren Standort nicht verändern, bleibt die statische Information valide. Nach Umstellung der Clients sollten die nicht mehr benötigten URLs aus dem Key-Value-Store entfernt werden, um Seiteneffekte zu entdecken und Verwirrung zu vermeiden.

Reverse Proxy

Bis zu diesem Schritt wurden Reverse Proxies und Loadbalancer durch das Betriebsteam händisch provisioniert. Nachdem nun aber alle notwendigen Informationen in der Service Registry vorliegen, können diese auch genutzt werden, um zum Beispiel den Reverse Proxy zu konfigurieren.

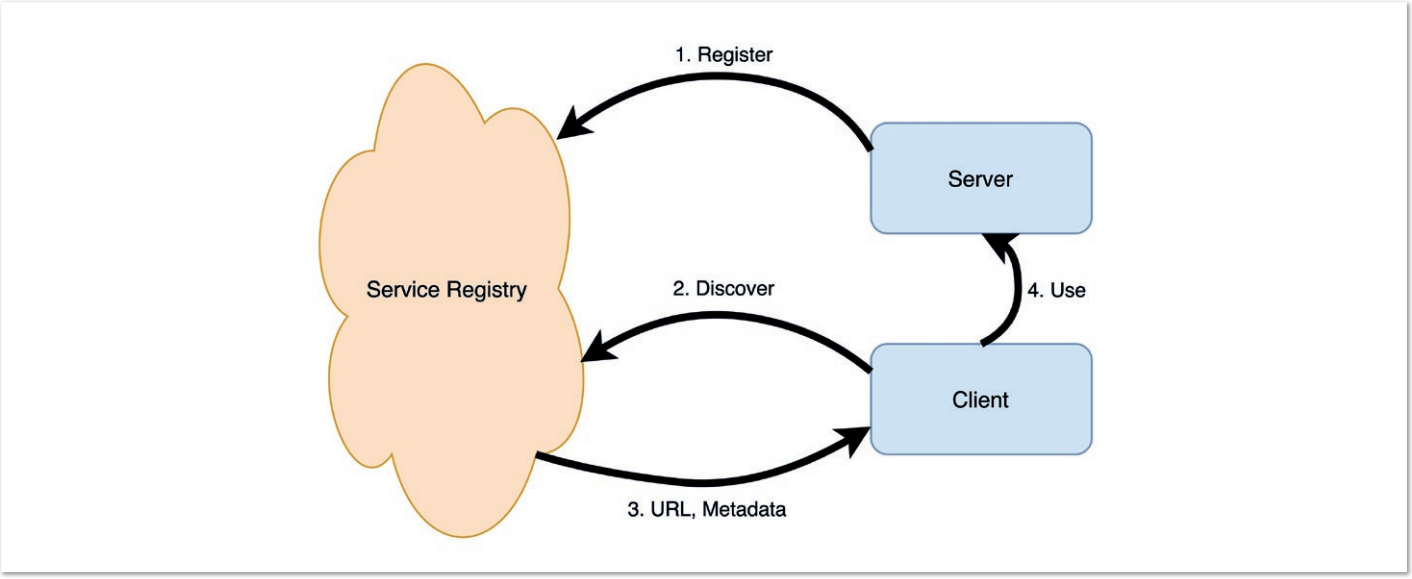

Abbildung 3: Service Discovery Flow (© Nils Bokermann)

```
spring.cloud.consul.discovery.healthCheckPath=/actuator/health
spring.cloud.consul.discovery.healthCheckInterval=5s
```

Listing 3: Konfiguration des Health-Endpunkts in den application.properties.

Am Beispiel von `traefik` *[3]* wird die Konfiguration eines Reverse Proxy vorgestellt. Die Vorbereitungen für diesen Schritt wurden auf der Software-Seite schon durch die Anbindung an die Service Discovery getroffen. Hier ist also die Betriebsabteilung gefragt, die Reverse Proxies umzustellen.

Eine minimale Konfiguration findet sich in *Listing 5*. Diese Konfiguration eignet sich dazu, in einer Entwicklungsumgebung erste Schritte zu gehen, ist jedoch nicht für einen produktiven Betrieb geeignet. Hierfür müssen weitere Regeln erstellt werden, da in dieser Konfiguration unter anderem auch der Consul-Service exponiert wird und damit das komplette Environment, also auch die Passwörter unserer Services.

Automatische Skalierung

Als nächster logischer Schritt steht die Deployment-Verteilung an. Auch zu diesem Zweck wird wiederum auf ein Tool aus dem Hause HashiCorp zurückgegriffen: HashiCorp Nomad *[4]*. Nomad ist ein

```
private String retrieveURL(
    DiscoveryClient discoveryClient) {
    Optional<ServiceInstance> serviceOptional =
       discoveryClient.getInstances("Backend")
            .stream().findAny();
    if (serviceOptional.isPresent()) {
        ServiceInstance serviceInstance =
             serviceOptional.get();
        URI uri = serviceInstance.getUri();
        return uri.toString();
    }
    return null;
}
```

Listing 4: Nutzung des DiscoveryClient, um einen dynamischen Endpunkt zu ermitteln

„Workload Orchestrator", es verteilt Applikationen auf zur Verfügung stehende Systeme. Dazu wird ein regelbasiertes System genutzt, das aufgrund von Last- und Speicher-Anforderungen der Applikation entscheidet, wohin die Applikation deployt wird.

Diese Form der Automatisierung hat diverse Auswirkungen auf die Software-Architektur. Durch die automatische Verteilung und Optimierung kann eine Applikation auf einem System beendet werden, um auf einem anderen System neu gestartet zu werden. Die Applikationen müssen also tolerant gegen einen Stopp per Unix-Signal sein. Es kann dabei genauso vorkommen, dass die Applikation auf zwei Systemen gleichzeitig läuft und Anfragen an beide Instanzen der Applikation verteilt werden. Lokale Datenhaltung oder Singletons können daher zu Problemen führen.

Für die Clients ergeben sich darüber hinaus noch weitere Anforderungen. Sie müssen in der Lage sein, auf fehlende Services fehlertolerant zu reagieren, zum Beispiel durch Retry-Mechanismen oder durch gecachte Default-Werte. Diese Lösungsansätze sind offensichtlich fachlich getrieben und daher kann es keine allgemeingültige (technische) Lösung geben.

Um einen möglichen Retry eines Requests zu gestatten, muss darauf geachtet werden, dass Schnittstellen idempotent gestaltet werden.

Die Nomad-Konfiguration *(siehe Listing 6)* bildet die Komplexität und Flexibilität einer Rechenzentrumsverwaltung ab. Nomad verwaltet eine dreistufige Hierarchie aus `jobs`, `groups` und `tasks`. Ein `task` ist das Deployment einer ausführbaren Einheit, also einer Applikation, eines Docker-Containers oder auch eines Skripts.

Innerhalb des `task` wird ein `service` definiert, der der Service Registry bekannt gemacht wird. Ein `check` ist optional, aber überaus sinnvoll,

```
# Consul als Provider anschliessen
[providers.consulCatalog]
  # Alle Service Exponieren (Nur für Entwicklungszwecke)
  exposedByDefault = true
  # Abfrage-Häufigkeit. Abwägung zwischen traffic und Ausfallsicherheit
  refreshInterval = "30s"
  # Für jeden Service einen Hostname mit seinem Namen
  defaultRule = "Host(`{{ .Name }}`)"
# Consul-Server
[providers.consulCatalog.endpoint]
    scheme = "http"
    address = "127.0.0.1:8500"
```

Listing 5: Erweiterung um den DiscoveryClient

```
# Job ist die toplevel Struktur-Einheit
job "frontend" {
  region = "global"
  datacenters = ["dc1"]

  type = "service"

  update {
    stagger      = "30s"
    max_parallel = 1
  }

# Group ist die zweite Stufe der Strukturierung
  group "webs" {
    count = 1

# Task ist die dritte Stufe der Strukturierung
    task "frontend" {
      # Nomad unterstützt verschiedene “driver”.
      driver = "java"

      # Die Konfiguration ist spezifisch für den „driver“
      config {
        jar_path    = "local/frontend-1.0.0-SNAPSHOT.jar"
        args        = ["--backend.instanceName=backend-docker-backend"]
      }

      artifact {
        source      = "https://github.com/sanddorn/InfrastructureAsMicroservice/releases/download/1.0.0-SNAPSHOT/
frontend-1.0.0-SNAPSHOT.jar"
        options {
          checksum = "md5:892e87bd35f7ca7c989b9ad38df85632"
        }
      }

      service {
        port = "http"

        check {
          type     = "http"
          path     = "/actuator/health"
          interval = "30s"
          timeout  = "2s"
        }
      }

      resources {
        cpu    = 500 # MHz
        memory = 512 # MB

        network {
          port "http" {
       static = 8080
          }
        }
      }
    }
  }
}
```

Listing 6: Nomad-Deployment-Konfiguration

um den Betrieb der Applikation automatisiert sicherzustellen. Soll der Service-Port dynamisch, und nicht wie im Beispiel statisch, konfiguriert werden, ist das möglich. Die gewählte Adresse und der Port liegen dann im Start-Environment der Applikation. Weiterführende Informationen sind auf der Nomad-Website zu finden *[5]*.

Security-Härtung

Es ist offensichtlich, dass Credentials nicht im Klartext gespeichert werden sollten. Zuvor, bei der Einführung des Key-Value-Stores, wurde dieses Sicherheitsrisiko bewusst in Kauf genommen. An dieser Stelle werden wir durch die Einführung einer Secret-Management-Lösung dieses Problem beheben. Die Einführung einer Secret-Management-Lösung lässt sich zu jeder Zeit während der Transition angehen, also auch als ersten Schritt.

Mit HashiCorp Vault *[6]* steht eine Secret-Management-Lösung zur Verfügung, die sich ähnlich einfach, wie auch schon HashiCorp Consul, in das Spring-Boot-Universum einführen lässt. Neben der zusätzlichen Dependency (`spring-cloud-starter-vault-config`) muss hier allerdings noch eine Bootstrap-Konfiguration angelegt werden *(siehe Listing 7)*. Die Applikationskonfiguration (`application.properties`) reicht hierbei nicht, da auch Credentials verwaltet werden können, die schon im Bootstrap der Spring-Boot-Anwendung genutzt werden (beispielsweise Datenbank-Credentials oder TLS-Keys und Zertifikate).

Innerhalb der Demo-Anwendung wird ein Vault-Token benutzt, das ein vollständiges Credential ist. In einem sicheren Applikationsbetrieb sollte eine andere Art der Vault-Absicherung gewählt werden. Für die Applikationsentwicklung ist die Einbindung der Secret-Management-Lösung damit abgeschlossen; die kompliziertere Aufgabe ist es, das Rechte-Management sowie die Replikation innerhalb der Secret-Management-Lösung zu konfigurieren und zu implementieren.

Die neue „Silver Bullet"?

Die „Silver Bullet" ist diese Lösung nicht. Es gibt diverse andere Produkte auf dem Software-Markt, mit denen eine ähnliche Lösung aufzubauen ist. Da die HashiCorp-Module sich aber so gut in die Spring-Boot-Welt einbauen lassen, lag es nahe, diese auch für einen kurzen Abriss zu nutzen.

Kubernetes diskret?

War das jetzt Kubernetes in einzelnen Bausteinen? So halb. Auf der einen Seite ist die komplette Netzwerk-Schicht außen vor geblieben und kann nach bekannter Art mit Switch, Router und Firewall weiter betrieben werden. Auf der anderen Seite gibt es hier die Möglichkeit, nicht nur Container zu deployen, sondern auch andere Arten von Software, solange sie aus einer Shell heraus gestartet werden können.

```
spring:
  cloud:
    vault:
      token: ${VAULT_TOKEN}
      host: localhost
      port: 8200
      scheme: http
```

Listing 7: bootstrap.yml zur Konfiguration des Secret-Managements

Für den Fall, dass wir die Applikationen in Container verpacken können, ist die entstandene Landschaft darauf vorbereitet, auch innerhalb von Kubernetes-Clustern betrieben zu werden. Die Hürde, die jetzt noch durch die Umsetzung von Namespaces, Network-Policies und Pod-Security-Policies zu nehmen ist, wird deutlich leichter fallen als zu Beginn der Transition. Fraglich bleibt jetzt allerdings, ob ein Umstieg auf Kubernetes wirtschaftlich sinnvoll ist.

Wie geht's weiter?

In diesem Artikel wird jeder einzelne Transitionsschritt nur angerissen und die Folgen auf die Software-Architektur tiefer, die Folgen auf die Betriebsstruktur nur sehr oberflächlich behandelt. Die betrieblichen und Security-Aspekte für die Einführung der jeweiligen Services (Consul, Nomad, traefik und Vault) würden den Rahmen eines Überblick-Artikels bei Weitem sprengen.

Quellen

[1] Nils Bokermann, *https://github.com/sanddorn/InfrastructureAsMicroservice*
[2] *https://www.consul.io*
[3] *https://traefik.io/traefik/*
[4] *https://www.nomadproject.io*
[5] *https://www.nomadproject.io/docs/job-specification/service*
[6] *https://www.vaultproject.io*

Nils Bokermann
Freiberufler
nils.bokermann@bermuda.de

Nils Bokermann ist als freiberuflicher IT-Consultant unterwegs und berät seine Kunden in Architektur- und Methodik-Fragen. Er stellt gerne Hype-getriebene Entscheidungen infrage und ist daher gerne auch als „Advocatus Diaboli" bekannt.