Es muss nicht immer Kubernetes sein

Microservice-Architekturen on-premise betreiben

Stephan Kaps

In vielen größeren Unternehmen existiert noch jede Menge Software, die eher monolithisch aufgebaut ist. Diese wird häufig in Applikationsservern auf dedizierten virtuellen Maschinen von einem eher klassisch aufgestellten und organisatorisch separierten IT-Betrieb betrieben. In Fachzeitschriften, Online-Artikeln und Konferenzen wird vorgeführt, wie einfach es doch ist, einen Spring Boot-Hello-World-Microservice mit mehreren Instanzen auf Kubernetes zu deployen. Doch zurück im Unternehmen wird klar: Sollte man es tatsächlich schaffen, alle notwendigen Personen davon zu überzeugen, ab sofort Kubernetes einzuführen, wird das für einen meist auch personell am Limit arbeitenden IT-Betrieb schnell zu einem Projekt mit vermutlich 1 bis 2 Jahren Laufzeit (je nach Erfahrung), mit möglichen Seiteneffekten, wie reduzierter Handlungsfähigkeit für das laufende Geschäft, und dem Zurückstellen anderer Modernisierungsmaßnahmen. In diesem Artikel wird die sich kontinuierlich entwickelnde (evolving) Architektur einer Anwendungslandschaft hin zu Cloud-native betrachtet und dabei werden Werkzeuge für die schrittweise Anpassung der On-premise-Infrastruktur vorgestellt, ganz ohne Kubernetes.

Domain-Driven Design

Für die Zerlegung eines Legacy-Systems in Microservices bietet sich als Vorgehensweise Domain-Driven Design (DDD) an. Bei Wikipedia findet man dazu folgende Definition:

Stephan Kaps leitet die Softwareentwicklung im Bundesversicherungsamt und ist Gründer der Java User Group Bonn. Als Softwarearchitekt und -entwickler hat er seit 2002 mit Java zu tun.
E-Mail: info@kitenco.de

„DDD ist eine Herangehensweise an die Modellierung komplexer [objektorientierter] Software. Die Modellierung der Software wird dabei maßgeblich von den umzusetzenden Fachlichkeiten der Anwendungsdomäne beeinflusst. Der Begriff ‚Domain-Driven Design' wurde 2003 von Eric Evans in seinem gleichnamigen Buch geprägt." [Wiki]

In diesem Artikel betrachten wir lediglich die Unterteilung einer Domäne in die drei Subdomänen generic, supporting und core. Für weiterführende Informationen zu DDD wird auf die bereits umfangreich existierende und gute Literatur verwiesen.

Die im Domain-Driven Design als *generic subdomains* bezeichneten Services repräsentieren bereits gelöste Probleme, die für diverse Systeme oder sogar unterschiedliche Unternehmen immer gleich umgesetzt werden. Sie gehören nicht zum Kerngeschäft (*core subdomains*) und können im einfachsten Fall hinzugekauft werden oder es existieren entsprechende Open-Source-Projekte. Ein Beispiel dafür wäre die Authentifizierung bzw. die Anbindung von Identity Providern. Hier könnte ein Produkt wie Keycloak [KeyCl] und Standards wie JWT oder OIDC zum Einsatz kommen.

In einer Fachanwendung existieren häufig zusätzlich mehrere unterstützende Funktionalitäten, die zwar mit der Business-Logik verknüpft sind, die jedoch leicht in separate Microservices ausgelagert werden könnten, um die Domäne von Code zu befreien, der nicht zum Kern gehört. Bei DDD spricht man dabei von *supporting subdomains*. Beispiele für supporting domains sind Referenz- oder Stammdatenverwaltungen, Protokollierungs- und Kommentarfunktionen.

Generic und supporting subdomain Services eignen sich demnach sehr gut als erste Microservice-Kandidaten. Dadurch können Makro-Architekturstandards eingeführt und das Altsystem von diversen cross-funktionalen oder Zusatz-Funktionalitäten befreit werden.

Warum eigentlich Microservices?

Microservices bieten generell eine elegante Art der Modularisierung. Es wird möglich, die für die eigentliche Domäne primär nicht relevanten fachlichen Teile auszulagern, was Vorteile wie lose Kopplung, bessere Wartbarkeit und Testbarkeit sowie unabhängige Deployments ermöglicht.

Microservices bieten generell die Möglichkeit, unabhängige Technologie-Stacks zu verwenden. Transitive Abhängigkeiten eingesetzter Bibliotheken sind manchmal mit denen anderer Bibliotheken und Frameworks nicht vereinbar und erschweren dadurch die Einführung neuer Technologien. Durch die Auslagerung in einen separaten Service, der wiederum seine eigenen Abhängigkeiten mitbringen kann, können solche technischen Probleme elegant gelöst werden.

Bei der Anbindung von Drittsystemen, bei denen es sich häufig um zugekaufte Standardsoftware handelt und an denen Schnittstellenanpassungen nicht einfach möglich sind, empfiehlt es sich, einen eigenen Service zwischenzuschalten. Es handelt sich dabei um eine architektonische Schicht, die das Domänenmodell von anderen Modellen trennt und die es ermöglicht, dass auf das fremde Modell so zugegriffen werden kann, wie das eigene Domänenmodell es benötigt.

Dieses Vorgehen fördert Information Hiding und verbessert die Benutzbarkeit der Schnittstellen, da diese auf die fachlichen Bedürfnisse zugeschnitten sind. Diese Designtechnik wird auch als Anti-Corruption-Schicht bezeichnet [aim42].

Ob die Core Domain des Legacy-Systems in mehrere Bounded Contexts zerschnitten werden sollte, hängt von zwei Faktoren ab. Zum einen stellt sich die technisch organisatorische Frage, also ob man mit mehreren Teams an dem Produkt arbeitet und diese unabhängiger voneinander agieren lassen möchte. Zum anderen hängt es vor allem vom fachlichen Bedarf ab. Gibt es nur einen fachlich Verantwortlichen für das Verfahren und existieren somit keine fachlichen Schnittstellen oder getrennte Kontexte, ist fraglich, ob eine Zerschneidung nicht eher zu einem verteilten Monolithen führt. Sind jedoch mehrere Abteilungen am Prozess beteiligt und haben diese selbst interne Abläufe, für die sie verantwortlich sind, würde alles dafür sprechen, sowohl die Fachanwendung als auch die Prozesse in unabhängige Einheiten zu separieren. Dadurch wären lokale Anpassungen leichter umzusetzen, ohne Abstimmung, wann welche Funktionalität mit welcher Version released werden kann.

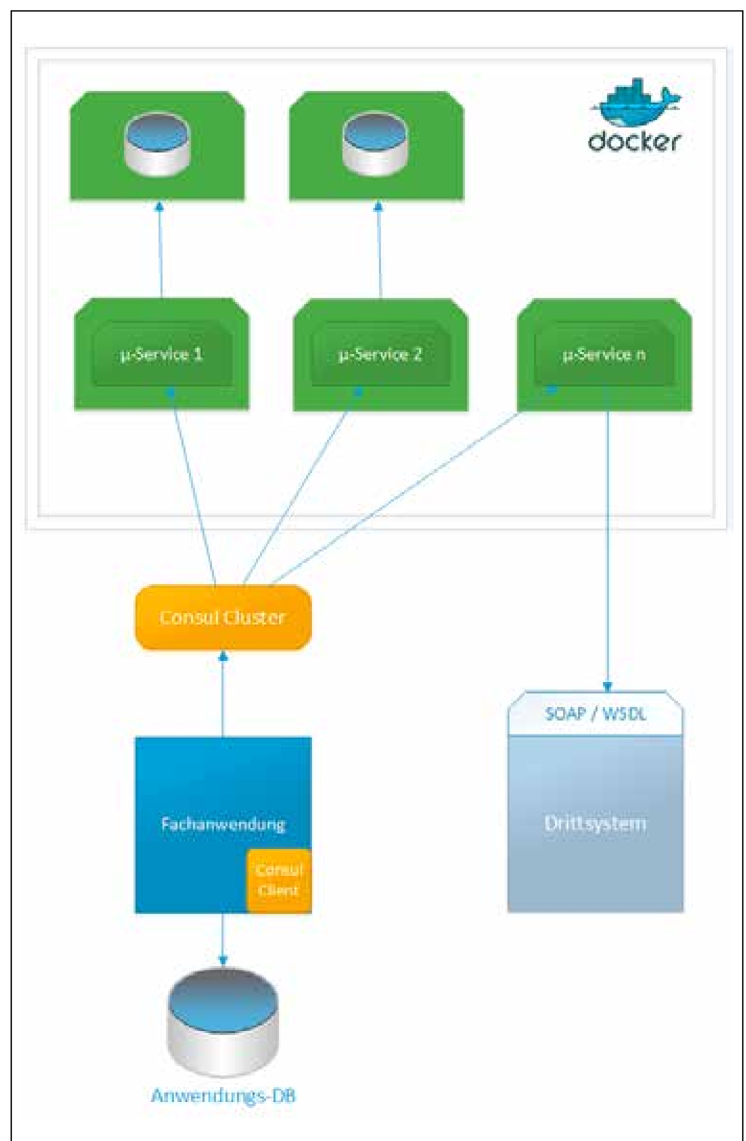

Abb. 1: Grober Überblick der Architektur in dieser Phase

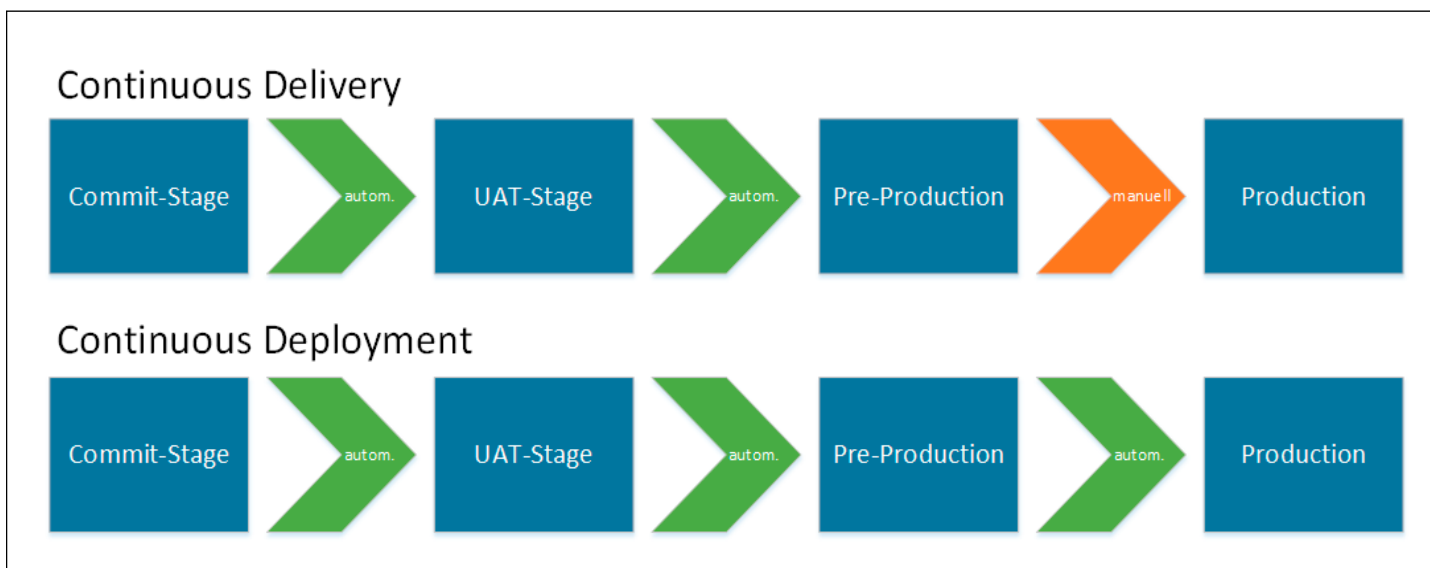

Abb. 2: Unterscheidung Continuous Delivery & Deployment

Neues Betriebsmodell

In direktem Zusammenhang mit Microservices steht deren Betrieb in Containern. Container-Technologien sind vor allem durch Docker populär geworden. Inzwischen gibt es jedoch sehr gute Alternativen für das Bauen und Deployen von Containern.

Bei der schrittweisen Umgestaltung der Anwendungsarchitektur, sowie der dazugehörigen Infrastruktur, genügt es, zunächst zwei bis drei Docker Nodes bereitzustellen. Für die ersten Microservices werden Dockerfiles erstellt, mit fest definierten Ports und lediglich einer Instanz auf einem dieser Nodes. Diese Konfiguration kann im ersten Schritt der aufrufenden Anwendung fest mitgegeben werden.

Mit fortschreitender Umstellung steigt die Zahl der Microservices und der Wunsch nach mehr Resilienz wächst. Um mehr Resilienz zu erreichen, können mehrere Instanzen eines Service gestartet werden. Dies führt wiederum dazu, dass fest konfigurierte statische Serviceadressen durch die Einführung einer sogenannten Service-Discovery abgelöst werden müssen. Dabei handelt es sich quasi um einen Katalog der zur Verfügung stehenden Services. Eine Anwendung fragt diese Service-Discovery lediglich mit einem Servicenamen an und erhält die entsprechende Serviceadresse zurück. Als Werkzeug bietet sich beispielsweise Consul von Hashicorp [HashiC] an, welches leicht on-premise betrieben werden kann. Die Service-Discovery kann die von den Microservices bereitgestellten Health-Checks auswerten, um entscheiden zu können, welcher Service denn überhaupt bereit ist, Anfragen entgegen zu nehmen.

Bei einer Anwendung, die mit den Services kommunizieren möchte, läuft ein Client, der die Anfragen über localhost mit lediglich dem Servicenamen annimmt. Abbildung 1 zeigt einen groben Überblick, wie das System in dieser Phase aussieht.

Da durch die Zerlegung von Legacy-Systemen mit der Zeit die Zahl der Container in den diversen Umgebungen stark steigt, wächst der Bedarf nach einer einfachen Möglichkeit, diese managen zu können. Das Werkzeug Portainer [port] bietet eine aufgeräumte Oberfläche für die Verwaltung und das Management von Container-Umgebungen. Es können diverse Hosts sowie Registries angebunden, laufende Container gestartet, gestoppt, überwacht und analysiert werden. Portainer lässt sich sehr einfach in Betrieb nehmen und bietet sowohl für Entwickler als auch Betrieb nützliche Funktionen. Container können auch manuell gepullt, gebaut, Netzwerke, Volumes und Images verwaltet werden. Zudem wird der Docker Swarm Mode unterstützt, also das Betreiben diverser Nodes als Cluster.

Bei verteilten Architekturen und steigender Zahl Services entsteht die Notwendigkeit, diese geeignet überwachen zu können. Unter den Begriff Observability fallen die Aspekte Logging, Monitoring und Tracing.

Für die Herausforderung des *Log*-Managements in verteilten Architekturen gibt es seit Langem Lösungen, wie beispielsweise der Elastic Stack [elastic]. Dieser ermöglicht es, sämtliche Log-Nachrichten diverser Services, unabhängig ob in einem Container oder in einem Applikationsserver betrieben, an eine zentrale Stelle zu übermitteln, wo sie aufbereitet, zerlegt, indiziert und ausgewertet werden können.

Beim *Monitoring* geht es meist um die Überwachung von System-Ressourcen wie CPU oder Speicher. Häufig existieren jedoch auch Anforderungen, fachliche Metriken zu sammeln, um das Verfahren steuern zu können. Weit verbreitete Werkzeuge in diesem Umfeld sind Prometheus [Prom] und Grafana [Graf]. Die Anwendungen und Services bieten spezielle Endpunkte an (/metrics), die von Prometheus abgefragt und persistiert werden, um im Anschluss in Grafana auf konfigurierten Dashboards visuell ansprechend dargestellt zu werden.

Beim *Tracing* geht es darum, einen Request von seinem Ursprung bis zum Ende verfolgen zu können. Das ist absolut sinnvoll und notwendig, wenn ein Request mehrere Services durchläuft, bis die eigentliche Response erfolgt und dabei zum Beispiel Latenz-Probleme zu analysieren sind. Technisch wird dies ermöglicht, indem eine Anwendung Aufrufe mit einem mit IDs angereicherten Header versieht, die im Anschluss von einer zentralen Instanz zusammengeführt werden können. Für diesen Zweck gibt es inzwischen mehrere Werkzeuge wie Zipkin [Zipk] oder Jaeger [Jaeger]. Diese bieten meist eine praktische Benutzeroberfläche, über die es möglich ist, die persistenten Daten abzufragen und die Aufrufketten grafisch ansprechend darzustellen.

Continuous Deployment

Aufgrund der Vervielfachung der Zahl der Komponenten eines Systems steigt nun wiederum der Bedarf, den Bereitstellungs-

prozess komplett zu automatisieren. Wir definieren, dass Continuous Delivery auch die manuelle Bereitstellung in der Produktion beinhaltet, während bei Continuous Deployment (s. Abb. 2) die Bereitstellung vollständig automatisch erfolgt.

Für Continuous Deployment existieren jedoch diverse Voraussetzungen, die zunächst erfüllt sein müssen. Dazu zählt die Etablierung von Continuous Integration und Continuous Delivery für alle Komponenten. Dafür wiederum existieren Anforderungen wie die Inbetriebnahme eines zentralen Artefakt-Repositorys, eines Build-Servers und einer Container-Registry.

Um Risiken zu minimieren, ist eine ausreichend hohe Testabdeckung durch automatisierte Tests und automatisiert reproduzierbare Pipelines notwendig sowie die Nutzung von Trunk-Based-Development sinnvoll. Um länger andauernde Implementierungs- oder Refactoring-Arbeiten zu ermöglichen, die ein System in einen nicht mehr auslieferungsfähigen Zustand versetzen können, bieten sich Konzepte wie Branch-by-Abstraction oder Feature-Toggles an.

Des Weiteren müssen Datenbank-Änderungen berücksichtigt und ebenfalls automatisiert werden. Werkzeuge dafür existieren [Flyw] und für Breaking-Changes am Schema bieten sich 2-Step-Migrations an. Zusätzlich sollten Prüfungen auf Verwundbarkeiten von Abhängigkeiten durchgeführt werden und im besten Fall auch dynamische Securtity-Tests (DAST), um Schwachstellen in der eigenen Anwendung frühzeitig zu entdecken.

Sollte eine Anforderung sein, dass es bei einem Update des Systems nicht zu Ausfallzeiten kommt (Zero-Downtime), bieten sich Blue-Green- oder Canary-Deployment-Strategien an. Diese lassen sich umsetzen, indem beim Deployment das alte System aus der Reverse-Proxy- beziehungsweise Load-Balancer-Konfiguration entfernt wird, während das neue hinzugefügt wird, oder durch den Einsatz von Feature-Toggles.

Bei einem klassischen Vorgehen mit geteilten Verantwortlichkeiten werden in der Regel von IT-Betriebsmitarbeitern vor dem Deployment die entsprechenden Zugangsdaten zu produktiven Datenbanken, Servern oder anderen Systemen manuell an die entsprechenden Stellen in den Konfigurationsdateien gesetzt. Um den Weg Richtung DevOps einschlagen zu können, muss das Ziel sein, Zugangsdaten zu verstecken, sowohl vor den Entwicklern, die jetzt Anwendungen produktiv setzen können, aber auch generell vor System- und Datenbankadministratoren aufgrund erhöhter Sicherheitsanforderungen.

Für Secrets-Management in dynamischen Infrastrukturen gibt es inzwischen sehr gute Werkzeugunterstützung von Hashicorp Vault [Vault]. Dabei handelt es sich um einen digitalen Schlüsselkasten geschrieben in Go mit vielen weiteren Funktionalitäten. Allgemein geht es um die geheime Aufbewahrung und den sicheren Zugriff auf Secrets. Dabei kann es sich um die Ablage von Schlüsselmaterial für die Inhaltsverschlüsselung von gesendeten und empfangenen Daten handeln, um vertrauliche Umgebungsvariablen, Zertifikate, Datenbankzugriffsdaten oder API-Keys.

Die Geheimnisse werden über ein API abgefragt und die Klartextversion verlässt dabei nie den Vault. Der Zugriff ist nur für autorisierte und authentifizierte User oder Applikationen möglich, was wiederum über Policies konfiguriert wird. Jeder Zugriff auf Geheimnisse wird zusätzlich aufgezeichnet (Audit). Die Schlüssel haben eine begrenzte Gültigkeit und können automatisch erneuert werden, sogenanntes Key Rolling. Für Datenbanken können beispielsweise dynamische Secrets erzeugt werden, die nur für den einen Zugriff gültig sind.

Durch den Einsatz von Vault ist es nun also möglich, Anwendungen produktiv zu setzen, ohne dass sensible Konfigurationen bekannt gemacht oder bereitgestellt werden müssen. Spring beispielsweise bietet mit dem Projekt Spring Cloud Vault eine direkte Integration an.

Service-Mesh

Alle Microservices sollen ausschließlich per *Mutual TLS (mTLS)* kommunizieren. Dafür müssen sie mit Zertifikaten versehen werden. Um die Zertifikate jedoch nicht in jeden Keystore sämtlicher Container integrieren zu müssen, bieten sich sogenannte Service-Proxys wie Traefik oder Envoy an. Diese bekommen mit, wenn ein neuer Container gestartet wurde, und können dann die Routen automatisch absichern.

Seit Version 1.2 stehen mit *Connect* Funktionalitäten in Consul bereit, die in Zukunft in Richtung Service-Mesh ausgebaut werden. Consul verwendet dabei zum Beispiel auch Envoy als Sidecar-Proxy und kann sich die Zertifikate für die mTLS-Absicherung der Services automatisch von Vault erzeugen lassen, wo die eigenen Root- und CA-Zertifikate hinterlegt sind. Diese Zertifikate können wiederum automatisch ausgetauscht werden, wodurch kurze Gültigkeiten festgelegt werden können, zum Beispiel 24 Stunden, was wiederum die Sicherheit erhöht (moving targets). Dieses Konzept mag den meisten bereits durch Let's Encrypt bekannt vorkommen. Jedoch ist es für viele Institutionen ausgeschlossen, dass für eine Let's Encrypt Challenge eine Kommunikation ins interne LAN ermöglicht wird. Durch die Lösung aus Consul/Vault wird es möglich, mithilfe der eigenen PKI interne Zertifikate automatisiert zu erzeugen und zu verteilen.

Die Services kommunizieren aber auch untereinander (Inter-Service-Communication), sodass wir nicht mehr nur noch eine Nord-Süd-, sondern auch eine Ost-West-Kommunikation absichern müssen. In einem einfachen Szenario, bestehend aus zwei Containern, wobei im ersten Container die Webanwendung läuft und im zweiten eine Datenbank, soll sichergestellt werden, dass der Container mit der Webanwendung, mit dem Container kommunizieren darf, in dem die Datenbank betrieben wird, aber nicht umgekehrt.

Consul Connect bietet seit Version 1.4 Intention Policies. Dadurch wird geregelt, dass beispielsweise Container der Gruppe webapp auf Container der Gruppe webdb zugreifen dürfen, jedoch nicht umgekehrt (s. Abb. 3).

In klassischen Anwendungslandschaften konnten IP-Adressen und Ports in Firewalls einmalig freigeschaltet werden. Loadbalancer-Konfigurationen und Routen im DNS waren statisch. Sobald wir uns in Cloud-Infrastrukturen bewegen, egal ob Private, Public oder on-premise, wird diese statische Infrastruktur zu einer dynamischen und eine manuelle Pflege schwierig. Die Zahl der Services steigt rasant, Services werden gestoppt und eventuell an anderer Stelle neu gestartet. Existiert der Bedarf der dynamischen Skalierung und kommt es dadurch zum Einsatz eines Orchestrators/Schedulers, ist das klassische manuelle Vorgehen nicht aufrecht zu erhalten.

Neue Herausforderungen

Bei der Zerlegung von monolithischen Systemen in diverse Microservices steigt vermutlich zunächst einmal der Ressourcen-Verbrauch. Ein sehr simpler Microservice für eine Validierung von PLZ/Ort-Kombinationen verbrauchte als Spring Boot-Anwen-

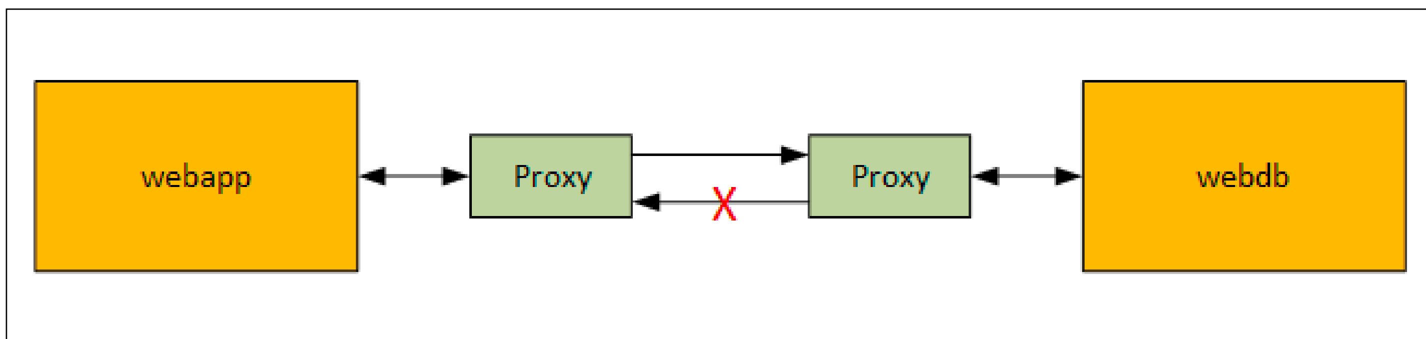

Abb. 3: Inter-Service-Communication Intention Policies

dung über 600 MB Arbeitsspeicher. Durch eine relativ einfache Umstellung auf Quarkus [Quark] konnte dieser Wert auf ca. 90 MB reduziert werden. Eine Umstellung der JVM von Hotspot auf OpenJ9 hat ebenfalls dazu beigetragen.

Klassische Webanwendungen werden häufig in Applikationsservern betrieben. Das funktioniert meist sehr gut und das notwendige Know-how ist bei den Entwicklern vorhanden. Alternativ geht die Entwicklung des JBoss Wildflys durch die Einführung von Galleon [Den19] auch mehr in die Cloud-native Richtung und ermöglicht dadurch einen schlanken Applikationsserver einfacher in Containern betreiben zu können. Sind die Startzeiten nicht so wichtig, ist es aber auch problemlos möglich, einen Wildfly mit dem Full-Profile im Container laufen zu lassen. Der RAM-Verbrauch bewegt sich dabei um die 500 MB.

Quarkus oder Spring Boot bieten eine direkte Integration von Vault, der Applikationsserver Wildfly leider noch nicht. Ein vorhandener Java-Client hilft dabei nicht weiter, da alle Zugangsdaten zum Beispiel zur Datenbank beim Start des Wildflys bereitstehen müssen. Diese Problematik ließe sich mit Werkzeugen wie envconsul oder consul-Template lösen, allerdings nicht mit vergleichbarem Sicherheitsniveau, denn entweder stehen im Anschluss die Zugangsdaten wieder im Klartext in der Konfiguration des App-Servers, oder eben in den Umgebungsvariablen.

Beim Betrieb von Containern in Produktion sind viele neue Sicherheitsaspekte zu betrachten. Es muss sichergestellt werden, dass nicht ungewollt fremde Container aus unbekannten Quellen oder manipulierte Images mit Malware den Weg in die Produktion schaffen. Des Weiteren sollen keine Container betrieben werden, deren Images bekannte Verwundbarkeiten beinhalten. Die Open-Source-Cloud-native-Container-Registry Harbor [Harb] integriert die bekannten Tools Clair für Vulnerability Scanning und Docker Notary für das Signieren von Images und ermöglicht es, Compliances zu erzwingen, sodass Images mit Verwundbarkeiten oder nicht signierte Images gar nicht erst gepullt werden können.

Einen guten Einstieg, was zu beachten ist beim Betrieb von Docker-Containern in Produktion, bietet der Container Security Verification Standard [CSVS], der seit Kurzem unter dem Dach der OWASP zu finden ist. Stand 11/2019 beinhaltet der Standard insgesamt 106 Empfehlungen unterteilt in 12 Kategorien und jeweils drei Stufen, die man als Beginner-, Fortgeschritten- und Expertenniveau beschreiben könnte.

Der Autor hat die deutsche Übersetzung zu diesem Open-Source-Projekt beigesteuert. Der aktuelle Technology Radar von ThoughtWorks [TWRadar] hat Container Security in den Bereich *adopt* aufgenommen, was die Bedeutsamkeit und Aktualität widerspiegelt.

Fazit

Wir haben gesehen, dass es möglich ist, Microservice-Architekturen im eigenen Rechenzentrum zu betreiben, ohne direkt zu Kubernetes greifen zu müssen. Die Anforderung an die Skalierung ist aus Sicht des Autors der entscheidende Faktor. Handelt es sich eventuell nur um intern genutzte Systeme oder um externe Systeme, mit einer begrenzten Zahl an Usern und berechenbarem Traffic, dann ist der Einsatz eines Orchestrators/Schedulers wie Kubernetes nicht erforderlich.

Herausforderungen wie Verfügbarkeit, Qualität, Schnelligkeit, Sicherheit und vor allem geteilte Verantwortlichkeiten können mit den vorgestellten Maßnahmen und Werkzeugen gemeistert werden. Dieses Vorgehen ermöglicht sowohl der Entwicklung als auch dem IT-Betrieb inkrementelle Anpassungen, kontinuierliches Lernen und eine evolvierende Architektur.

Im Vergleich dazu gleicht die Einführung von Kubernetes einer Big-Bang-Migration, die einen großen Vorlauf benötigt und im Ergebnis einem Monolithen ähnelt, nur auf Infrastrukturseite. Ob Microservice-Architekturen überhaupt erstrebenswert sind, ist abzuwägen, indem alle Trade-Offs berücksichtigt werden [Fow15].

Links

[aim42] http://aim42.github.io/#Anticorruption-Layer
[CSVS] https://www.owasp.org/index.php/OWASP_Container_Security_Verification_Standard_(CSVS)
[Den19] J.-F. Denise, Wildfly 16 and Galleon, towards a cloud native EE application server, 1.3.2019, https://wildfly.org/news/2019/03/01/Galleon_Openshift/
[elastic] elastic.co
[Flyw] flywaydb.org
[Fow15] M. Fowler, Microservice Trade-Offs, 1.7.2015, https://martinfowler.com/articles/microservice-trade-offs.html
[Graf] grafana.com
[Harb] goharbor.io
[HashiC] consul.io
[Jaeger] jaegertracing.io
[KeyCl] keycloak.org
[port] portainer.io
[Prom] prometheus.io
[Quark] quarkus.io
[TWRadar] https://www.thoughtworks.com/de/radar/techniques?blipid=1041
[Vault] vaultproject.io
[Wiki] https://de.wikipedia.org/wiki/Domain-driven_Design
[Zipk] zipkin.io