www.javaspektrum.de

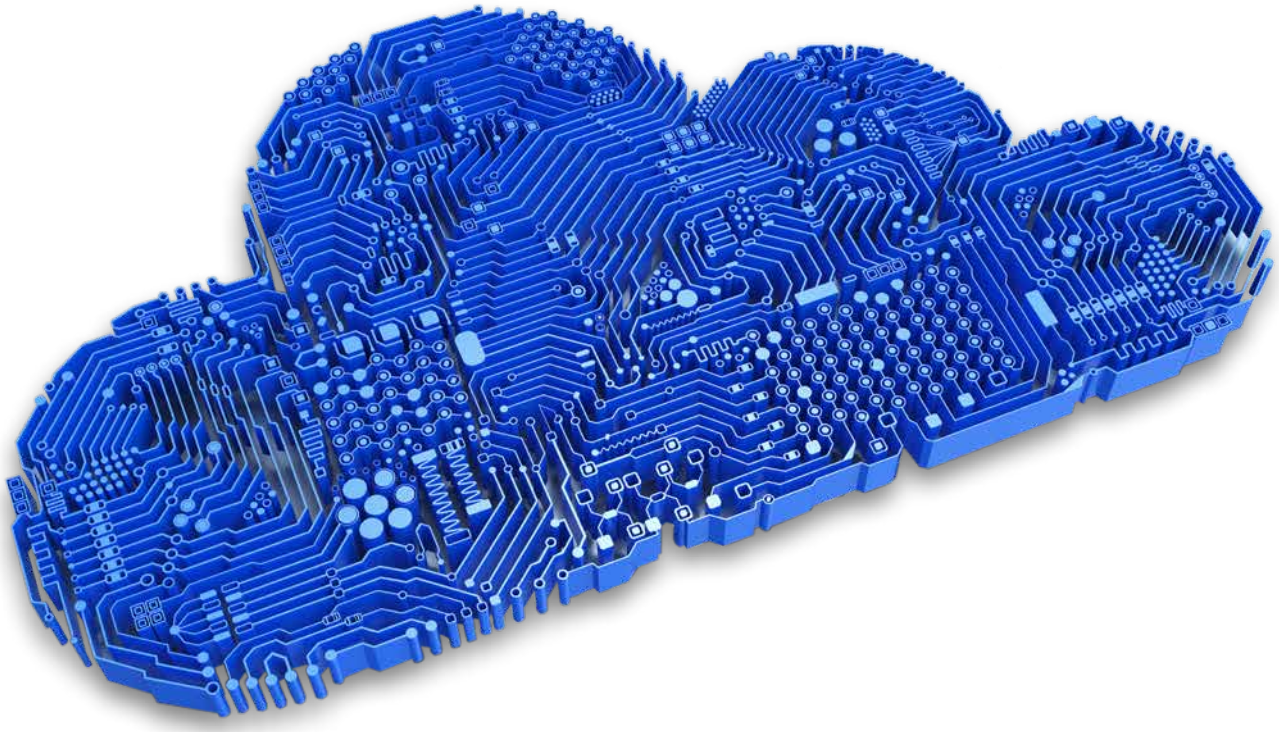
Ausgabe 1/2020
D € 9,90 · A € 10,90 · CHF 18,20

4 194156 109901 01

JavaSPEKTRUM

Magazin für professionelle Entwicklung und digitale Transformation

Cloud Native – Architektur & Integration



Cloud Native Enterprise Architecture

Johannes Weigend, Johannes Siedersleben,
Mario-Leander Reimer

Cloud Native – Architektur & Integration

G30325

Cluster One

Cloud Native Enterprise Architecture

Johannes Weigend, Johannes Siedersleben, Mario-Leander Reimer

Cloud-native Entwicklung hat den Zenit des Gartner-Hype-Zyklus überschritten. Die erste Welle der Ernüchterung ist eingetreten. Zwei Jahre junge Anwendungen sind schon heute technische Altlasten. Die Entwickelbarkeit von stark verteilten Anwendungen geht an die Grenze des Erträglichen. Der Aufwand für Betrieb und Wartung der Infrastruktur ist immens. Die Komplexität steigt an Stellen, die bisher von Outsourcing-Betriebsprovidern gerade noch bedient werden konnten. Der Kollaps ist programmiert. Wie kann Cloud-native Softwareentwicklung auf Unternehmensebene erfolgreich funktionieren und welche Rolle spielt die Softwarearchitektur dabei?

Daher stellen sich folgende Fragen:

- Was sind die übergreifenden Prinzipien für den Entwurf und Bau von Cloud-native Systemen auf Unternehmensebene?
- Wie sieht eine Referenzarchitektur für Cloud-native Systeme aus?

Dieser Artikel präsentiert ein neues Konzept unter dem Namen *Cluster One* – eine Anlehnung an die Quasar-Standardarchitektur von 2004 [Sid04]. Die Bausteine von Cluster One sind:

- Domain-Driven Design (DDD) [DDD03],
- Microservices,
- DevOps auf Cloud-Plattformen,
- Software-Blutgruppen.
- Außerdem: Agilität und iteratives Vorgehen.

Zu jedem einzelnen Baustein gibt es viel Literatur. Entscheidend ist aber die Kombination: Wie hängt alles zusammen? Geht DDD nur in der Cloud? Funktioniert DevOps nur mit Microservices? Wie schneidet man Microservices und wie implementiert man sie? Was hat das Hexagon-Modell [Cock05] mit den Quasar-Blutgruppen zu tun? Wo liegt der DevOps-Code?

Dieser Artikel beschreibt Architekturprinzipien in einer Welt, in der agile Teams autarke Microservices von der Idee bis zum Betrieb verantworten. Sie planen ihr Produkt im Sinne von DDD, bauen es komponentenorientiert, bringen es auf die Cloud und schließlich in Produktion. Den klassischen Betrieb gibt es entweder gar nicht mehr oder in einer radikal geänderten Funktion. Unbewegliche Monolithen werden ersetzt durch Microservices, die nebeneinander wachsen und gedeihen.

Softwarearchitektur

Die Bedeutung von Softwarearchitektur wird immer wieder diskutiert: Braucht man so etwas überhaupt noch im Zeitalter von agilen Teams, Cloud-nativer Entwicklung und DevOps oder entsteht Softwarearchitektur agil und magisch-evolutionär von selbst?

Bevor wir darüber nachdenken, ist es wichtig, die Sichten zu justieren. Über welchen Teil der Architektur reden wir: über die konzeptionelle Seite einer Anwendung, die technische Umsetzung oder die Details einer Cloud-native Plattform wie Kubernetes? Jedes System hat drei Sichten:

- Die *A-Architektur* beschreibt die Anwendung.
- Die *T-Architektur* beschreibt die technische Umsetzung.
- Die *TI-Architektur* beschreibt die Infrastruktur und die Betriebskomponenten.

Die A-Architektur

Die A-Architektur oder Anwendungs-Architektur (engl. Domain Architecture) beschreibt fachlich, aus welchen Teilen ein System oder ein Systemverbund besteht. Hier ist es egal, ob man über Cloud-native oder konventionelle, aus Modulen oder Komponenten zusammengesetzte Systeme spricht. Allerdings können nicht-funktionale Eigenschaften wie Performanz und Skalierbarkeit den grundsätzlichen Schnitt beeinflussen. Es gibt viele Ansätze, die A-Architektur zu beschreiben: von Prosa zu formalen UML-Modellen hin zu Datenbankmodellierung. Ein Ansatz, der durch die Cloud-native Entwicklung auf Basis von Microservices in den Fokus gerückt ist, ist DDD.

Domain-Driven Design (DDD)

Domain-Driven Design enthält viele Elemente und Best Practices der letzten 20 Jahre für den Bau von modularen IT-Systemen. Ein Punkt ist aber zentral anders:

Die neue Botschaft von DDD ist *Autarkie*. Der zentrale Begriff ist der *Bounded Context*: eine fachliche Einheit mit einer bestimmten Wertschöpfung, einem eigenen Datenmodell und einer einheitlichen Terminologie (Ubiquitous Language), die sich auch im Code wiederfindet. Jeder Kontext ist über ein Interface zugänglich, dessen Services fachlich motiviert sind und sich weitgehend selbst erklären. Das Interface ist die Außensicht; der Kunde (also der Aufrufer) soll es möglichst einfach haben.

Eine Anti-Corruption-Schicht trennt das Domänenmodell von anderen Modellen und sorgt somit für *Autarkie*. DDD erklärt die *Autarkie* des Bounded Context als oberstes Design-Ziel und nimmt dafür Datenredundanz und Dateninkonsistenz in Kauf (s. Abb. 1). In unserer Sprache ist ein Bounded Context nichts anderes als eine Anwendungskomponente, in der Regel ziemlich groß.

Johannes Weigend ist technischer Geschäftsführer bei der QAware GmbH, München. Er entwirft und entwickelt seit mehr als 20 Jahren datenintensive Systeme mit Java, JavaScript, C/C++ und Go. Johannes Weigend ist Java Rockstar und Experte für Diagnose von Cloud-native Anwendungen. Er beschäftigt sich mit der Architektur Cloud-nativer Systeme und hält Vorlesungen an der TH Rosenheim. E-Mail: johannes.weigend@qaware.de

Prof. Dr. Johannes Siedersleben ist Beirat der QAware GmbH, München. Er hat mit seinen Publikationen das Thema Softwarearchitektur in Deutschland und darüber hinaus beeinflusst. Er berät die QAware in allen Gebieten des Software-Engineering. E-Mail: johannes.siedersleben@qaware.de

Mario-Leander Reimer ist passionierter Entwickler, Architekt und #CloudNativeNerd. Als Cheftechnologe bei der QAware GmbH verantwortet er den technischen Erfolg von Projekten im Aftersales-Bereich beim Kunden BMW. Er beschäftigt sich intensiv mit Technologien rund um den Cloud-native Stack und dessen Einsatzmöglichkeiten im Unternehmensumfeld. Als Lehrbeauftragter unterrichtet er Cloud-Computing und Software-Qualitätssicherung an der TH Rosenheim. E-Mail: mario-leander.reimer@qaware.de

Mario-L. Reimer hält auf der OOP2020 den Vortrag Holistische Sicherheit für Microservice-Architekturen, 4.2.2020, 14:00 – 14:45

Autarkie vs. Autonomie

Die linke Seite von Abbildung 1 zeigt den klassischen Ansatz: Der Service S1 ist zuständig für Aktualität und Konsistenz der Daten D. Andere Services (hier S2), die D ganz oder teilweise lesend verwenden, holen sich die Daten über Services der Zugriffsschicht ZG1. Das hat schon oft funktioniert, schafft aber eine Abhängigkeit: S2 läuft nur, wenn S1 läuft.

Die rechte Seite von Abbildung 1 zeigt die Idee der *Autarkie*: Der Service S2 (und jeder andere, der Daten von D lesend benötigt) hat seine eigene Kopie D2 und seine eigene Zugriffsschicht ZG2. D2 ist etwas kleiner als D dargestellt, weil S2 vielleicht nur einen Bruchteil von D verwendet. S2 wird ereignisgesteuert versorgt: Jede relevante Änderung von D bewirkt einen Update von D2. Dieser Update ist verzögert; D und D2 sind für die Dauer des Updates inkonsistent (oft nur Bruchteile von Sekunden); wenn S1 steht oder die Datenversorgung unterbrochen ist, kann die Inkonsistenz länger dauern (Minuten oder Stunden). Dies alles setzt natürlich voraus, dass S2 *aus fachlicher Sicht* mit veralteten Daten umgehen kann – dann ist S2 autark und läuft auch ohne S1.

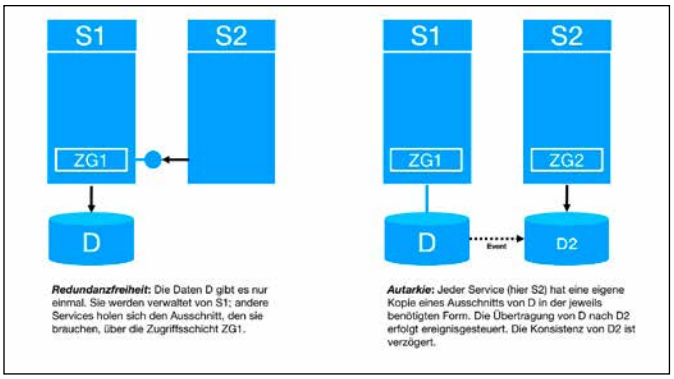


Abb. 1: Redundanzfreiheit vs. Autarkie

Auch bei DDD gilt der Grundsatz, dass für jeden Datenbereich genau eine Komponente das Schreibrecht hat und somit die Verantwortung für Aktualität und Konsistenz.

Natürlich ist das *Autarkie*-Pattern (rechte Seite von Abb. 1) kein Allheilmittel; der Einsatz erfordert Vorsicht. Redundanz und Inkonsistenz sind beherrschbar, das größte Problem ist die ereignisgesteuerte Datenübertragung: Datenflüsse ersetzen Aufruf-Beziehungen. Deshalb müssen sie genauso sorgfältig geplant werden. Es gibt nützliche Tools (z. B. Kafka), aber *there is no silver bullet*.

Die T-Architektur

Die A-Architektur lässt sich komplett ohne Wissen über Programmiersprache oder Plattform beschreiben und entwickeln. Das ist wichtig und notwendig, um die Komplexität zu beherrschen. Redet man über Java EE oder über einen Warenkorb? In der technischen Umsetzung spielen nicht-funktionale Anforderungen die zentrale Rolle: Auf welchem Cluster-Orchestrator läuft die Anwendung?

Microservices

Microservices sind die technische Umsetzung eines Bounded Context.

Weitgehende Einigkeit besteht über die folgenden Eigenschaften: Microservices sind Einheiten von Entwurf, Entwicklung, Integration, Deployment, Wartung und Betrieb. Jeder Microservice wird ganzheitlich von einem Team betreut; der Bruch zwischen Entwicklung und Betrieb verschwindet.

Weitgehende Uneinigkeit besteht über die Größe. Der Name *Micro* suggeriert kleine Einheiten; manche Autoren beschränken Microservices auf 1000 LoC.

Microservices sind die Bausteine in einer Microservice-orientierten Architektur. Microservices sind in erster Linie ganz normale Komponenten: Jeder Microservice implementiert eine oder mehrere Schnittstellen, die von anderen Microservices aufgerufen werden, und er kann selbst andere Microservices aufrufen, natürlich nur über deren Schnittstellen. Jeder einzelne Microservice ist in der Innensicht nach allen Regeln der Kunst in Komponenten aufgeteilt.

Microservices besitzen aber zwei besondere Eigenschaften, die sie von gewöhnlichen Komponenten unterscheiden: Jeder Microservice ist sowohl eine Einheit des Deployments als auch der Teamorganisation; jeder Microservice hat seinen eigenen Entwicklungs- und Releasezyklus. Das funktioniert nur unter bestimmten technischen, fachlichen und organisatorischen Voraussetzungen:

Fachlich: Jeder Microservice ist aus fachlicher Sicht eine sinnvolle Einheit, ein abgeschlossenes Ganzes. Microservices enthalten oft (aber nicht immer) eine eigene GUI und eine eigene Datenhal-

tung. Die fachlichen Abhängigkeiten sind in der Weise minimiert, dass der Microservice für sich allein entwickelt und getestet werden kann.

Technisch: Jeder Microservice ist aus technischer Sicht eine sinnvolle Einheit, ein abgeschlossenes Ganzes. Die technischen Abhängigkeiten sind minimiert. Microservices machen minimale Annahmen über ihre Umgebung. Sie laufen lokal oder in einem Cluster, und man kann jeden Microservice beliebig vervielfältigen: Er läuft in einem, zehn oder hundert Exemplaren.

Organisatorisch: Jeder Microservice wird von einem Team betreut; jedes Team ist für die Microservices verantwortlich, die es im Lauf der Zeit gebaut hat. Natürlich gibt es manchmal auch Änderungen, die zwei oder mehr Microservices betreffen. In diesem Fall stimmen sich die betroffen Teams ab und bringen ihre Deployments falls nötig in eine sinnvolle Reihenfolge. Das passt zu agiler Entwicklung in Feature-Teams.

Alle Regeln der Komponentenbildung gelten auch für Microservices, zum Beispiel das Verbot von Zyklen. Aber man muss auch umdenken. Neu und ungewohnt ist die Handhabe von loser Kopplung, Redundanz und Heterogenität.

Lose Kopplung. Microservices misstrauen sich gegenseitig; Aufrufer und Gerufener rechnen mit dem Schlimmsten. Der Aufrufer stellt sich darauf ein, dass der Aufruf fehlschlägt, und agiert entsprechend defensiv. Der Gerufene stellt sich darauf ein, dass der Rufer Vorbedingungen missachtet, unsinnige Daten schickt oder gar einen Angriff startet. Er prüft also eine ganze Menge, bevor er den Aufruf überhaupt akzeptiert. Externe Aufrufe über Microservice-Grenzen sind schwergewichtig und langsam. Autarke Microservices kommunizieren deshalb bevorzugt asynchron über Ereignisse: Datensenken (der empfangende Microservice) registrieren sich an einer geeigneten Stelle; Datenquellen (der liefernde Microservice) melden alle relevanten Änderungen.

Redundanz. Jeder Microservice hat endogene und/oder exogene Daten. Endogene Daten entstehen zum Beispiel durch Messung, Berechnung oder Benutzereingaben lokal. Der zuständige Microservice ist der Single Point of Truth; er verantwortet Korrektheit und Konsistenz. Exogene Daten sind solche, die er von anderen Microservices bekommt. Die Grundidee ist folgende: Jeder Microservice speichert sowohl endogene als auch exogene Daten selbst. Die Synchronisation erfolgt ereignisgesteuert nach dem Prinzip der Eventual Consistency: Jeder Microservice meldet relevante Änderungen seiner endogenen Daten und wird benachrichtigt bei relevanten Änderungen seiner exogenen Daten.

Die Vorteile sind *Autarkie* und Performanz: Jeder Microservice kann alleine laufen, schlimmstenfalls mit veralteten Daten, und die Performanz ist hervorragend, weil jeder Microservice eine für seine Bedürfnisse optimierte Datenablage besitzt. Nachteile sind Short Term Inconsistency, Redundanz und eine Flut von Ereignissen, die zu kanalisieren sind.

Datensynchronisation über Servicegrenzen mittels Events ist auch eine Eigenschaft des CQRS-Patterns. Allerdings bezieht sich CQRS (Command query responsibility segregation) auf einen Bounded Context, während es bei DDD um die Synchronisation der Daten in unterschiedlichen Bounded Kontexten geht.

Heterogenität. Microservices können von verschiedenen Teams in verschiedenen Programmiersprachen geschrieben sein. Der eine verwendet Solr, ein anderer MongoDB. Jeder Microservice hat seine eigenen Bibliotheken, verwendet seine eigenen Versionen und kümmert sich überhaupt recht wenig um andere Microservices. Technische Shared Libraries sind in Ordnung, wenn sie sich selten oder nie ändern, zum Beispiel eine Logging Library. Wenn nicht, schaffen sie Abhängigkeiten. Technische und fachliche Migrationen (ein neuer Suchalgorithmus) erfolgen pro Microservice und nicht mehr in der Breite. Früher waren Einheitlichkeit und Konsistenz ein Wert an sich. Das ist nicht mehr der Fall.

Autarkie vs. Autonomie

Microservices sind nicht autonom, sondern autark, jedenfalls soweit wie möglich. Autonomie bedeutet Selbstbestimmung, Souveränität. Autonom sind Staaten, Organisationen, Personen. Software ist nicht autonom, sondern mehr oder weniger autark. Microservices sind so autark wie möglich, das heißt, sie laufen weiter, auch wenn benachbarte Services nicht verfügbar sind. Autarkie bedeutet vollständige oder teilweise Selbstversorgung, Unabhängigkeit von externen Ressourcen (man vergleiche hierzu die entsprechenden Wikipedia-Einträge).

Wie groß sind Microservices?

Jeder Microservice bedeutet organisatorisch ein Team, ein Produkt und einen Releasezyklus. Schon die Aufteilung eines großen Systems in zwei Microservices ist ein riesiger Fortschritt: Man bekommt zwei unabhängige Teams, zwei Produkte und zwei Releasezyklen, die man mit mehr Redundanz, mehr Aufrufen und mehr Ereignissen bezahlt. Wenn dieser Preis zu hoch ist, dann schneidet man eben nicht. Die Menge der Microservices sollte überschaubar sein. Microservices können ziemlich groß sein. Aber das ist kein Problem, denn sie werden intern in eng gekoppelte Komponenten geschnitten. Alle Regeln der komponentenorientierten Softwarearchitektur gelten auch und gerade innerhalb von Microservices. Der Begriff *Micro* ist irreführend und hat zu vielen Missverständnissen geführt; Regeln wie *maximal 1000 LoC pro Microservice* sind schädlich.

Microservices in der Anwendungslandschaft

In einer Welt von Microservices rückt die Vorstellung eines Systems im herkömmlichen Sinn in den Hintergrund: Früher hatte man vielleicht 50 Systeme, heute sind es 500 Microservices. Wie verwaltet man die? Die Antwort lautet: Man betrachtet sie als Komponenten. Alles, was für Komponenten gilt, gilt auch für Microservices: Man kann sie auf einer oder mehreren Ebenen zu größeren Einheiten zusammenfügen (möglicher Name: Domäne, Domain), man kann Fassaden bauen und vieles mehr. Microservices sind die Komponenten der Anwendungslandschaft.

Microservices und Quasar-Blutgruppen

Microservices haben wie jede Komponente eine Innen- und eine Außensicht. Solange die Außensicht den genannten Regeln folgt, ist die Innensicht im Grunde zweitrangig: Verschiedene Teams verschiedener Glaubensrichtungen können im selben Großprojekt arbeiten, ohne sich in kleinlichen Diskussionen zu verlieren.

Trotzdem gibt es eine Standardarchitektur für Microservices: das alte Schichtenmodell in neuer Form. Früher hatte man die GUI oben, den Anwendungskern in der Mitte, die Datenbank unten und die Nachbarsystem irgendwie seitlich. Im neuen Modell (Hexagon oder Zwiebel, s. Abb. 2) befindet sich der Anwendungskern in der Mitte und kommuniziert mit verschiedenen Partnern, die sternförmig angeordnet sind. Das Hexagon, dem wir aus optischen Gründen den Vorzug geben, zeigt genau sechs externe Partner; im Allgemeinen sind es natürlich beliebig viele. Der Anwendungskern besteht aus drei Schichten oder besser Ringen:

- Die *fachliche Domäne* enthält die statischen Regeln, die zu jedem Zeitpunkt gelten.
- Der nächste Ring (*Prozesse*) implementiert die Geschäftsprozesse, und
- der äußere (*Services*) die Services so, wie sie den externen Partnern zur Verfügung gestellt werden.

Diese Ringe unterscheiden sich in ihrer Stabilität: Am häufigsten ändern sich die Services, am wenigsten häufig die statischen Regeln. Alle drei Ringe enthalten nur fachlichen Code (A-Code), sie sind frei von jeder Technik, egal ob REST, SQL oder NoSQL. Die Technik steckt in den Adaptern (grün in Abb. 2): Wenn ein Service in REST und SOAP angeboten wird, dann baut man zwei Adapter.

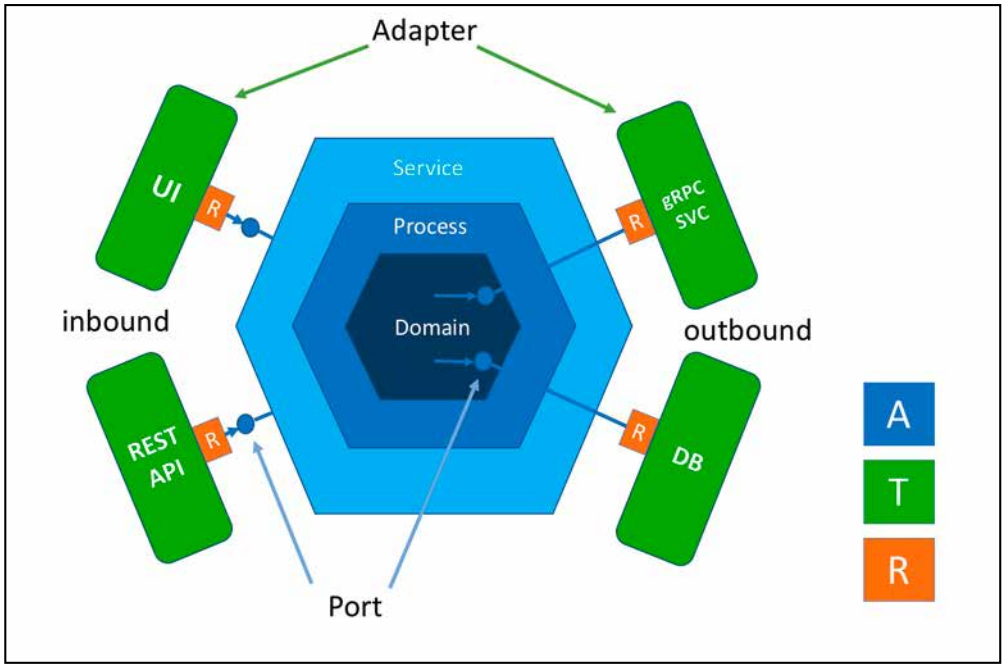


Abb. 2: Hexagonale Architektur & Quasar-Blutgruppen

Die Transformation von A- in T-Code erfolgt durch entsprechenden R-Code (oder Mapping Code). R-Code ist stereotyper, generierbarer Code ohne fachliches Wissen. Adapter und der zugehörige R-Code funktionieren in beide Richtungen: Bei Outbound-Schnittstellen transformieren sie von A nach T, bei Inbound-Schnittstellen von T nach A.

Das Konzept der Farben und der A/T/R-Klassifikation stammt aus dem Quasar-Blutgruppenmodell. Wenn man das richtig umsetzt, kann man die Domain- und Prozesslogik mit Mocks ideal testen, da die grünen Teile zum Beispiel per CDI ausgetauscht werden können. Der Service hat einen echten Anwendungskern, der nur die Fachlogik enthält und keinerlei spezifisches Wissen über Middleware (REST, gRPC ...) oder sonstige Technologie enthält.

Vom Hexagon zu Atomic Architecture

Das Hexagon-Modell ist eine Sonderform des allgemeineren Zwiebelmodells. Allerdings ist das Hexagon-Modell schärfer bei der Frage, wie die Ringe miteinander in Verbindung stehen. Wir bevorzugen eine runde Darstellung wie beim Zwiebelmodell, allerdings mit der Semantik des Hexagon-Modell. Wir nennen diese Notation das *Atomic Architecture*-Modell (s. Abb. 3).

Moderne Microservices verwenden zahllose Querschnittsfunktionen. Außerdem ermöglicht die Abstraktion eines Microservice als Atom auch den Bau von eng gekoppelten Molekülen und Verbindungen.

TI-Architektur

Bei der A- und T-Architektur hat sich eigentlich nicht viel geändert: Wir schneiden Systeme wie bisher in unabhängige Deployment-Einheiten, vermeiden Kopplung durch Redundanz und trennen die Anwendung von den technischen Details. An einer Stelle hat sich aber die Welt in der letzten Dekade komplett gedreht: die Automation des Betriebs durch die Cloud. Mit Cloud-Technologie lässt sich die TI-Architektur komplett automatisiert erstellen, ausführen und betreiben. Das gilt für Infrastruktur (Netzwerk, Rechner, Festplatten) und für Anwendung. Das funktioniert im Self-Service pro Entwicklungsteam. So verschmelzen Dev- und Ops zu DevOps.

DevOps und Cloud

Schon vor 20 Jahren hätte man Systeme nach dem hexagonalen Modell entwerfen und umsetzen können. Allerdings sprechen wir heute aufgrund der gewünschten *Autarkie* von vielen unabhängigen Services, die getrennt entwickelt und betrieben werden sollen. Im Kern hat man das mit EJB auch schon früher versucht (allerdings auf einer gemeinsamen Sprache und Entwicklungsplattform). Das war aber letztlich zu kompliziert, sodass die meisten EJB-Anwendungen monolithisch sind.

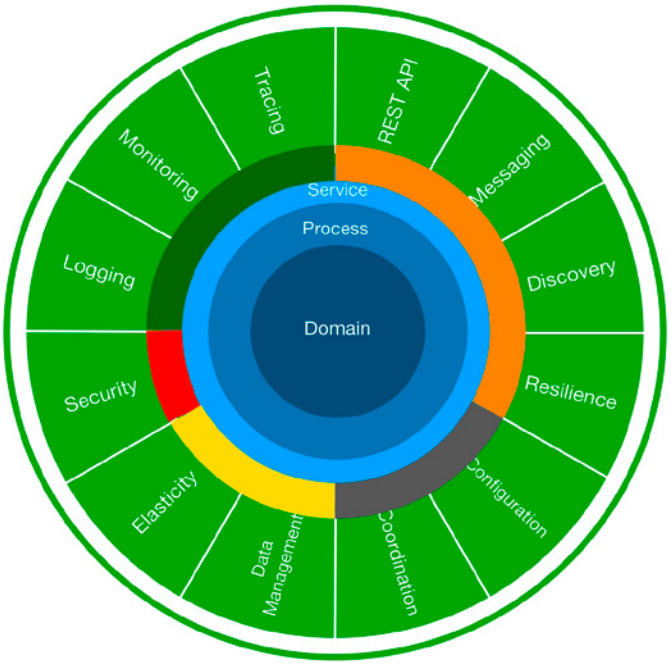


Abb. 3: Cluster One – Atomic Architecture

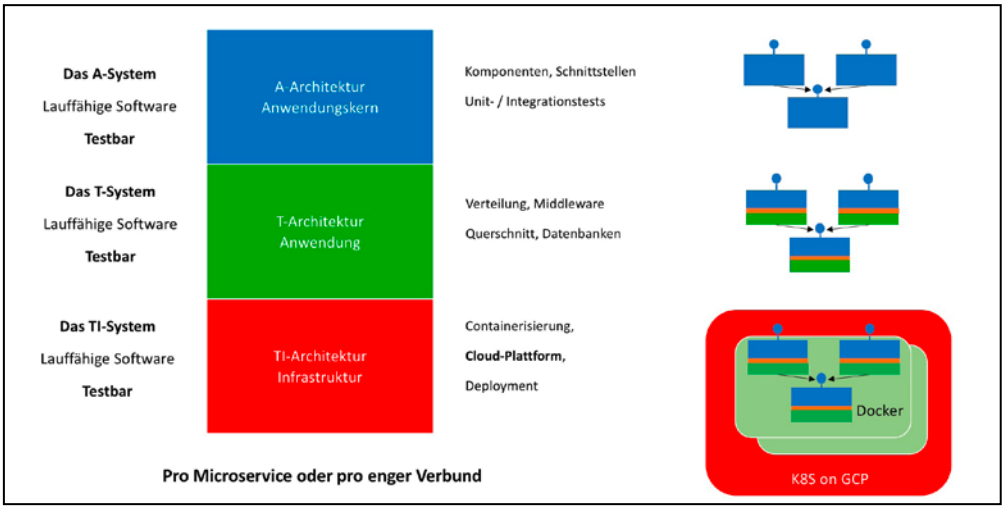


Abb. 4: Das A/T/TI-System als lauffähige Software

Es gab einige disruptive Entwicklungen, die damals kaum vorhersehbar waren und zum heutigen Stand des Cloud-Computing geführt haben:

- *IaaS* (Infrastructure as a Service): Virtualisierung von Rechenleistung und Automation. Mittels Virtualisierung und per Steuerung über Code ist es möglich, virtuelle Rechner in wenigen Sekunden zu erzeugen, zu starten und bei Bedarf zu löschen. Das gilt für Rechenleistung wie auch für Storage. Software implementiert heute die TI-Architektur. Das war früher undenkbar.
- *CaaS* (Cluster as a Service): Docker und Kubernetes spielen hier die zentrale Rolle. Mit Docker können alle Betriebskomponenten inklusive aller Abhängigkeiten auf einem System per Image ausgebracht und betrieben werden (Container). Kubernetes abstrahiert vom physikalischen Cluster und stellt eine logische Ausführungsumgebung zur Verfügung, auf dem die Container dann mehrfach ausgeführt werden können (skalierbar).
- *PaaS* (Platform as a Service): Plattformen als Ad-hoc-Entwicklungs- und -Betriebsplattform mit Fokus auf der einfachen Nutzung durch den Entwickler. Die Anwendung wird als Quellcode oder Applikationspaket übergeben, die Plattform kümmert sich automatisch um Deployment, Lifecycle-Management, Provisionierung, Service-Management und das Monitoring. Die Anwendung sieht dabei nur die Standard-APIs der Laufzeitumgebung.

Cluster One

Die Gefahr der Cloud ist, dass Anwendungen von Anfang an für eine spezielle Plattform entwickelt werden. Eine Austauschbarkeit ist dann nicht mehr gegeben, und jede noch so kleine Änderung der Basistechnologie wird zum aufreibenden Migrationsprojekt. Hier sind wir beim Kern von Cluster One: der Idee lauffähige Software auf jeder Ebene zu erreichen, Expertenwissen zu fokussieren und technologische Evolution zu ermöglichen.

Cluster One sieht drei Systeme (s. Abb. 4) vor, die unabhängig voneinander laufen, getestet werden können und die Ebenen entkoppeln. Das sind:

- Das A-System, welches die A-Architektur umsetzt. Das ist der Kern der hexagonalen Architektur.
- Das T-System, welches das A-System enthält und das aus technischen unabhängig aufgebauten Services auf jeder Cloud-Plattform oder auch Bare Metall läuft.
- Das TI-System, welches den DevOps-Infrastrukturcode enthält und welches die Anwendung mit einer konkreten Betriebsumgebung verbindet.

Den Kern bildet das A-System. Fügt man nun die technischen Adapter zum Service hinzu, erhält man das T-System; siehe Hexagon-Modell. Damit hat man ein lauffähiges System mit technischer Anbindung und technischen Schnittstellen (HTTPS, REST, DB).

Das Wesentliche hier: Sowohl A- als auch T-System sind Cloud-agnostisch. Beim T-System kann bereits Container-Technologie eingesetzt werden (z. B. für Tests). Aber verboten ist alles, was ein System an eine bestimmte Cloud, eine Plattform oder einen Provider bindet. Dies geschieht erst in der dritten Schicht, dem TI-System. Das TI-System kennt K8s, ISTO, AWS oder andere Technologien. Das TI-System kann auch Cloud-spezifische Adapter enthalten, die per Dependency Injection austauschbar sind.

Entscheidend ist: Alle drei Systeme (A alleine, A und T, A, T und TI) sind lauffähige Software. Außerdem sollte von oben nach unten entwickelt werden. Idealerweise ist der Code, der zum T-System und zum TI-System gehört, gering und kann mithilfe von Open-Source-Software minimiert werden. Ein System kann ein einzelner Microservice oder ein eng gekoppelter Verbund von mehreren Services sein. Alles, was bei der Entwicklung Agilität verspricht, ist erlaubt.

Warum der Aufwand? Das System ist verständlich. Das reduziert die Entwicklungs- und Wartungskosten. Das A- und T-System ist Cloud-agnostisch. Es war schon immer eine gute Idee, sich von konkreten Technologien zu entkoppeln. Die Entwicklung der OR-Mapper wie Hibernate oder Standards wie Java EE leben das vor. Damit verhindert man Vendor Lock und erreicht Zukunftssicherheit und Investitionsschutz.

Auch wenn ein Feature-Team auf allen Ebenen arbeiten muss, um ein fertiges Produkt auszuliefern, ist die saubere Aufteilung sinnvoll. Diese minimiert die Komplexität und schafft Fokus bei der Anwendungserstellung.

Damit bekommt die Cloud die richtige Bedeutung: Sie ist ein Betriebsmodell mit neuartigen Möglichkeiten, aber kein Selbstzweck. Cloud-Technologie verändert sich rasant und disruptiv. Anwendungen laufen aber über Jahre und Jahrzehnte. Es gibt also gute Gründe, sich von der Cloud zu entkoppeln, und dies ist möglich, ohne die Vorteile bei Deployment und Betrieb zu verlieren.

Literatur und Links

[Cock05] A. Cockburn, 2005, https://www.infoq.com/news/2014/10/exploring-hexagonal-architecture/

[DDD03] E. J. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2004

[Kamm19] Ch. Kamm, J. Weigend, J. Adersberger, Cloud-Native Enterprises, in: OBJEKTspektrum, OTS, 2019, https://www.sigs-datacom.de/ots/2019/cloud-computing/2-cloud-native-enterprises.html

[Sid04] J. Siedersleben, Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar, dpunkt.verlag, 2004

IT-Probleme lösen.
Digitale Zukunft gestalten.

Mit Erfindergeist und Handwerksstolz.

Great Place To Work. 2019
Beste Arbeitgeber™
Information & Telekommunikation

qaware.de/karriere
kununu.de/qaware

QAWARE
SOFTWARE ENGINEERING

IT-Probleme lösen.
Digitale Zukunft gestalten.

QAware GmbH München
Aschauer Straße 32
81549 München
Tel.: +49 89 232315-0
Fax: +49 89 232315-129
E-Mail: info@qaware.de

QAware GmbH Mainz
Rheinstraße 4 C
55116 Mainz
Tel.: +49 6131 21569-0
Fax: +49 6131 21569-68
E-Mail: info@qaware.de

QAWARE
SOFTWARE ENGINEERING