MICROSERVICES UND IHRE ALTERNATIVEN

Mono? Micro? Macro? Modulo?

Microservice-Architekturen gelten als modernes Mittel, anpassbare Softwarestrukturen zu schaffen. Dabei haben auch sie Nachteile und punkten nicht in jeder Disziplin.

Spricht man heute über Microservices, rollt manch ein Entwickler schon entnervt mit den Augen. Zum einen, weil man in den letzten Jahren überraschend oft damit konfrontiert wurde. Zum anderen, weil das Architekturmuster lange Zeit als alternativlos konnotiert wurde und mittlerweile sogar in den Etagen der IT-Entscheider als das Nonplusultra der Softwarearchitekturen gilt.

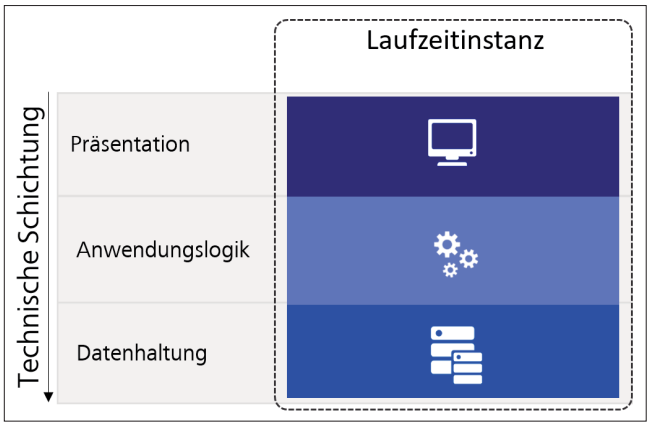
Zugegeben, diese Stellung kommt nicht von ungefähr, weisen Microservices doch eine Vielzahl unbestreitbarer Vorteile auf. Diese kommen aber auch zu einem Preis, und ob man bereit ist, den zu zahlen, kann man eben nur einschätzen, wenn man ihn auch kennt. Ziel dieses Artikels ist es daher, einen genaueren Blick auf das Architekturmuster und seine Nachteile zu werfen. Darüber hinaus sollen aber auch Alternativen aufgezeigt und einander gegenübergestellt werden, um eine Grundlage für zukünftige Architekturentscheidungen zu bieten.

Der Monolith

Bevor wir uns aber den Microservices im Detail zuwenden, betrachten wir zunächst ein anderes Muster, das in den vergangenen Jahren zunehmend wie ein Anti-Pattern behandelt wurde und mit seinen Nachteilen viele Argumentationshilfen pro Microservices geliefert hat. Die Rede ist vom Monolithen. Hierbei ist Monolith aber nicht gleich Monolith, und die Unterscheidung in Architekturmuster und Anti-Pattern ist durchaus angebracht, auch wenn der Übergang fließend ist.

Grundsätzlich beschreibt ein Monolith eine Softwarearchitektur, bei der das gesamte System nur als Einheit funktioniert und somit nicht aufgetrennt wird beziehungsweise aufgetrennt werden kann. Dabei ist es unerheblich, ob der Monolith intern zum Beispiel einer Schichtenarchitektur wie in Bild 1 folgt. Das Mono im Namen besagt, dass es nur „eines" gibt, und dieses „eine" kann nicht weiter zerlegt werden.

Je nachdem, was mit dem „einen" gemeint ist, können diverse Arten von Monolithen unterschieden werden. Deployment-Monolithen sind beispielsweise Softwaresysteme, bei denen alle Softwarebestandteile gemeinsam veröffentlicht werden müssen. Laufzeit-Monolithen hingegen sind Systeme, deren Bestandteile getrennt voneinander bereitgestellt werden und damit auch unterschiedlichen Releasezyklen folgen können. Für deren reibungslosen Betrieb müssen aber alle Bestandteile zur Laufzeit auch verfügbar und zueinander



Schichtenarchitektur eines Monolithen (Bild 1)

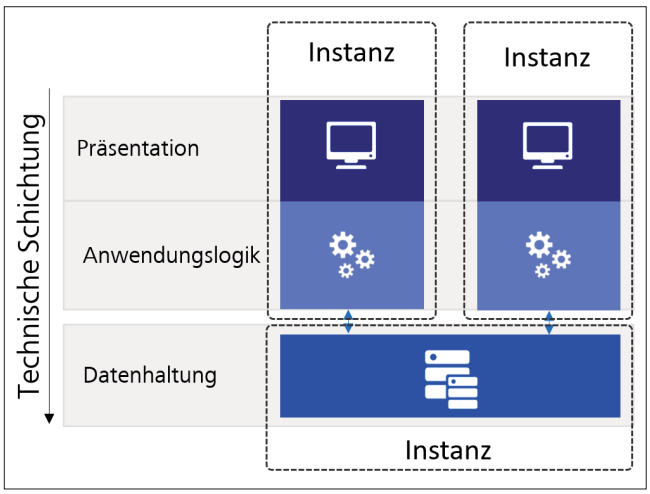
kompatibel sein. Dies bedeutet, dass die Bestandteile durchaus verteilt vorliegen, es aber eine so große Abhängigkeit zwischen ihnen gibt, dass sie immer nur als Ganzes betrieben werden können. Löst man also einen Teil heraus, funktioniert auch der Rest nicht mehr zuverlässig.

Nachteile des Monolithen

Je nachdem, ob es sich um einen Laufzeit- oder Deployment-Monolithen handelt, schränkt dies die Verwendung des Softwaresystems ein. Am deutlichsten wird dies bei der Skalierung. Wird die Software zur Laufzeit als ein einzelner Prozess auf einem einzelnen Rechner zur Verfügung gestellt, dann kann nur skaliert werden, indem entweder die Leistungsfähigkeit der Hardware erhöht wird (vertikale Skalierung) oder es muss eine exakte Kopie des Systems parallel aufgestellt werden. In letzterem Fall stellt sich dann aber die Frage, wie die beiden Instanzen miteinander synchronisiert werden können. In der Praxis geschieht dies nicht selten, indem eine gemeinsame Datenbank genutzt wird, auf die beide Monolith-Instanzen schreibend und lesend zugreifen. Hierbei ergibt sich dann aber mit der Datenbank selbst eine Art Monolith, der auf Dauer durchaus zum Flaschenhals werden kann. In dem Bestreben, die Skalierung zu verbessern, wurde ein Deployment-Monolith mit nur einer Laufzeitinstanz in einen Laufzeit-Monolithen gewandelt (Bild 2). Dessen Nachteile zeigen sich dann auch im Problemfeld der Ausfallsicherheit. Gibt es nur eine Instanz und fällt diese aus, so ist auch kein

weiteres Arbeiten möglich. Besitzt man mehrere Bestandteile, die aber stark voneinander abhängen, so ist es teils unerheblich, ob diese gemeinsam oder getrennt zur Verfügung gestellt werden können. Funktioniert einer der Bestandteile nicht, so funktionieren auch alle anderen nicht mehr.

Besonders interessant ist an dieser Stelle, dass sich die tatsächliche Ausfallsicherheit hierbei am schwächsten Glied der Kette orientiert. Ist der Datenbankserver instabil, so ist auch das Gesamtsystem instabil und die Aufteilung in mehrere Bestandteile hat möglicherweise das System sogar geschwächt. Denn die Fehlersuche zur Laufzeit ist in drei Bestandteilen, so wie in Bild 2 dargestellt, viel schwieriger, als sie es in einer einzelnen Instanz wie in Bild 1 war.



Ein Laufzeit-Monolith mit zwei Instanzen (Bild 2)

Code-Monolithen

Das Ganze macht aber noch nicht unbedingt ein Anti-Pattern, sondern soll zunächst nur die Nachteile des Architekturmusters aufzeigen. Die Erklärungen deuten dabei aber bereits an, wie fließend der Übergang zwischen Pattern und Anti-Pattern sein kann, vergleiche den Kasten Pattern versus Anti-Pattern. Einen Fall, in dem deutlichere Zeichen eines Anti-Patterns zu erkennen sind, wollen wir nachträglich als Sourcecode-Monolithen bezeichnen. In diesem Fall findet sich der gesamte Code der Applikation in einer einzelnen Codebasis wieder, auf der auch alle Entwickler zeitgleich und ohne größere Abgrenzung arbeiten und die intern so stark verwoben ist, dass man Bestandteile nur sehr schwer aus ihr herauslösen kann. Zugegeben, sowohl Microsoft als auch Google folgen einem ähnlichen Vorgehen für sehr große Softwareprojekte [1], und der Begriff des Mono-Repos (Monolithic Repository) beschreibt ein sehr ähnliches Vorgehen, das gerade in der Webentwicklung einigen positiven Zuspruch erhält. Hierbei arbeiten dann verschiedene Teams an verschiedenen Bestandteilen eines verteilten Softwaresystems, teilen sich aber ein großes Code-Repository. Dies hat zum Beispiel den Vorteil, dass der gesamte Code zeitgleich zur Verfügung steht und man „mal eben nachschauen" kann, wie denn eine Methode konkret umgesetzt ist, die man verwenden will. ▶

Pattern versus Anti-Pattern

Das Verhältnis zwischen Pattern und Anti-Pattern ist scheinbar sehr eindeutig, die Klassifizierung aber teils sehr schwer. Eindeutig ist hierbei zunächst nur, dass Patterns etwas sind, das uns die Erläuterung komplexer Zusammenhänge dank einer eindeutigen Namensgebung erleichtert. Ein echtes Muster ergibt sich aber nicht nur anhand des Namens, sondern auch aufgrund der wiederkehrenden Natur der Zusammenhänge. So weiß man bei einem Singleton sofort, dass es sich um eine Klasse handelt, die nur eine Instanz haben kann. Dieser Umstand ist sehr oft anzutreffen und wird somit auch oft von Entwicklern wahrgenommen und verstanden. Indem man der Sache nun einen Namen und eine Beschreibung gibt, kann man außerdem mit ihnen Best Practices verbinden. Wie genau soll also ein Singleton umgesetzt werden, damit man Probleme vermeidet? Das Pattern an sich ist zunächst also wertungsfrei, auch wenn Patterns generell gern positiv konnotiert sind. So muss ein Singleton nicht in jedem Fall eine gute Idee sein, andernfalls würden wir ja nur noch mit dieser Art von Klassen arbeiten. Es wird erst aufgrund seines Kontexts zu einer guten Entscheidung, und Musterbeschreibungen wie das berühmte Entwurfsmusterbuch der Gang of Four [11] liefern deshalb den Kontext für Patterns auch mit.

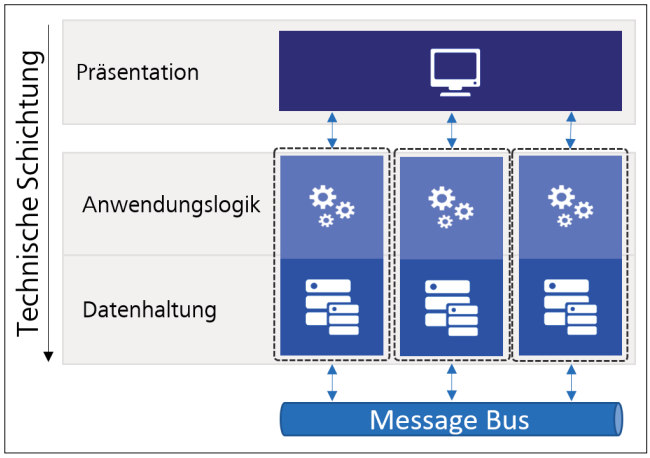
Bei einem Anti-Pattern ist es ganz ähnlich. Auch dies ist zunächst ein wiederkehrendes Muster mit einem eindeutigen Namen. Hierzu zählt zum Beispiel der berühmte Spaghetti-Code, den man nur schwer verstehen kann, weil keine eindeutigen Verantwortlichkeiten zu erkennen sind. Anti-Patterns sind dabei aber grundsätzlich negativ konnotiert. Etwas als Anti-Pattern zu bezeichnen ist also so, als würde man es mit einem großen Schild „Warnung" versehen.

Tatsächlich ergibt sich die Schadenswirkung eines Anti-Patterns aber ebenfalls erst aus seinem Kontext. Dies wird beispielsweise bei Code-Clones deutlich. Hierbei handelt es sich um Codeabschnitte, die sich sehr stark ähneln und zum Beispiel durch Copy-and-paste entstehen können. Sie sind schädlich und sollten vermieden werden, weil sie gegebenenfalls die gleiche Logik an verschiedenen Stellen in der Codebasis duplizieren. Nun ist aber nicht jede Zeile Code, die einer anderen ähnelt, auch gleich ein Klon. Vielmehr kann sich Code auch einfach so ähneln, weil er den gleichen Aufbau hat, nicht aber weil er die gleiche Logik abdeckt. Teilweise wird sogar bewusst Code kopiert, um keine unnötigen Abhängigkeiten herbeizuführen.

Ob ein Muster also hilfreich oder gar schädlich ist, ergibt sich aus seinem Kontext, und demzufolge sollte dieser Kontext vor dem Einsatz ganz klar geprüft werden. Dann kann sich auch schon mal herausstellen, dass ein offensichtlich scheinendes Anti-Pattern nur das geringste aller sonst möglichen Übel ist.

Sind in diesem Zusammenhang aber keine organisatorischen oder technischen Grenzen etabliert, hängt es nur noch von der Disziplin des Entwicklers ab, ob er jene Methode dann nicht auch noch gleich „anpasst". Das „mal eben etwas nachschauen" wird somit sehr leicht zum „mal eben etwas ändern". Solch ungeplante Spontanänderungen leisten der Architekturerosion aber Vorschub, indem sie Codebestandteile verbinden, die eigentlich entkoppelt sein sollten, was in der Praxis meist durch ein Umgehen der Schichtentrennung wahrgenommen werden kann. Sie sorgen außerdem für Bugs, da der Entwickler nicht immer wissen kann, in welchem Kontext der von ihm geänderte Code noch verwendet wird, und sie machen den Code insgesamt schwerer verständlich und lösen schnell eine Kaskade weiterer ungeplanter Änderungen aus.

Zugegeben, dies klingt, als würde der Autor versuchen, ein Horrorszenario zu konstruieren. Aber gerade in sehr lang laufenden Projekten, mit sehr großer Codebasis, ohne Code-Reviews und statische Codeanalyse, ergibt sich immer das gleiche Bild. Häufig kommt es zu einer Wucherung innerhalb des Quellcodes, die nachträglich nur sehr schwer zu beheben ist. Als Ergebnis kann die Codebasis dann nicht mehr aufgetrennt werden, weil die verschiedenen Bestandteile so stark miteinander verbunden sind, dass sie nur noch als Ganzes



Serviceorientierte Architekturen schematisch dargestellt (Bild 3)

funktionieren. Trennschichten haben sich damit also aufgelöst, und Schnittstellen sind nur noch Makulatur. Der ganze Prozess verläuft dabei so schleichend, dass er erst wahrgenommen wird, wenn es schon zu spät ist, vergleiche den Kasten Architekturerosion.

Um nun aber keinen falschen Eindruck zu erwecken: Ein Mono-Repo ist nicht zwangsläufig ein Anti-Pattern. Zum Code-Monolithen wird es, weil es unzureichende organisatorische und technische Sicherungsmechanismen gibt, um besagten Wildwuchs zu verhindern.

Warum macht man so was (nicht)?

Es stellt sich also die Frage, warum man nicht von Beginn an die Software entsprechend aufteilt und wartbar gestaltet. Bei neuer Software kann und sollte man dies auch tun. Bei bestehender Software hat dies meist den einfachen Grund, dass natürlich gewachsene Software nun einmal genau so entsteht. Mit „natürlich gewachsen" ist hierbei gemeint, dass die Software ohne größere Anpassungen der Gesamtarchitektur immer weiterentwickelt wird und die Architekturerosion somit ungehindert fortschreiten kann. Man beginnt mit einem kleinen Programm, das bestimmte Aufgaben erfüllt, und diese Aufgabenmenge steigt über die Jahre hinweg, womit die internen Strukturen des Softwaresystems selbst auch immer komplexer werden. Hierbei nachträglich eigenständige Module oder Komponenten herauszulösen, diese separiert bereitzustellen und zu pflegen ist mit einem erheblichen Restrukturierungsaufwand verbunden, der den Stakeholdern meist nur schwer vermittelt werden kann.

Die Nachteile von Monolithen liegen also auf der Hand: Sie können nur beschränkt skaliert werden und verleiten zu schwer wartbaren Strukturen. Somit wundert man sich nicht, dass eine Alternative, die all diese Nachteile adressiert, auf so offene Ohren gestoßen ist wie Microservices.

Vorteile von Microservices

Betrachtet man hierbei die gängigen Beschreibungen dessen, was als Microservice wahrgenommen wird, so erklärt sich die Begeisterung noch auf weitere Weisen. Als Microservices werden umgangssprachlich gern kleine Dienste verstanden, die zumindest zur Laufzeit voneinander entkoppelt sind und über ein bestimmtes Kommunikationsmedium miteinander interagieren (siehe Bild 3). Dieses Medium ist meist ein Netzwerk und wird über einen Service Bus oder über entsprechende Endpunkte zum Beispiel mit REST oder RPC umgesetzt. Dies verhindert, dass die Dienste über etwas anderes als ihre öffentlichen Schnittstellen angesprochen werden. Da sie physisch voneinander getrennt sind und in eigenen Prozessen, gegebenenfalls sogar auf völlig unterschiedlicher Hardware laufen, können sie zeitgleich auch leichter skaliert werden. Durch die Nutzung standardisierter Austauschformate kann darüber hinaus jedes Team seine Dienste technisch so umsetzen, wie es sie selbst umsetzen möchte. Dadurch können sich die Entwickler auf die Technologien konzentrieren, mit denen sie sich auskennen. Sollte ein Dienst außerdem einmal zu komplex oder zu schlecht wartbar sein, kann er aufgrund seiner geringen Größe vergleichsweise leicht ausgetauscht werden. Aus den gleichen Gründen sind die Dienste auch automatisch viel leichter verständlich und besser testbar als der Code innerhalb eines gewachsenen Monolithen. Dass sie aufgrund ihrer Natur auch ausfallsicherer sind, ist selbstverständlich.

Neben der rein technischen Skalierung erlauben Microservices aber auch eine organisatorische Skalierung. Neue Teams können sehr viel schneller produktiv arbeiten, als dies bei einem Monolithen der Fall ist. Diese neuen Teams erstellen eigene Dienste auf der grünen Wiese und müssen sich daher nicht so sehr um das kümmern, was bereits vorhanden ist. Insofern sie die Schnittstellen der anderen Dienste verstehen, können sie diese über Standardvorgehensweisen nutzen und brauchen sich nicht mit dem lästigen Klein-Klein von deren

Implementierung herumschlagen. Gerade Firmen wie Uber und Netflix konnten dank dieser Skalierbarkeit eine immense Entwicklungsgeschwindigkeit erreichen und binnen kurzer Zeit auf Marktveränderungen reagieren.

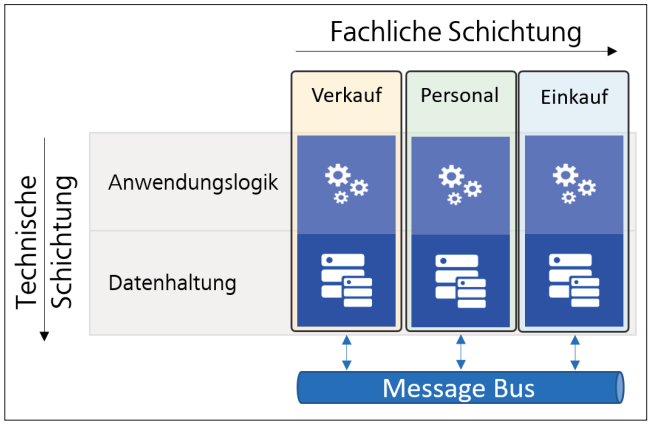
Was sind Microservices wirklich?

Das klingt fast zu gut, um wahr zu sein, und tatsächlich ist es das auch. Denn die zuvor genannte landläufige Beschreibung spart sehr viele der eigentlichen Kernkonzepte von Microservices aus und verschweigt die wichtigsten Problemfelder. Der Hauptfehler, der hierbei gern gemacht wird, ist, dass die Argumentation sehr stark von der technischen Seite getrieben wird. Das Kernkonzept von Microservices ist aber, dass sie gerade eben nicht technisch getrieben sein sollten, sondern fachlich.

Laut Sam Newman [2] sind Microservices Dienste, die unabhängig voneinander deployt werden können, um eine bestimmte Business-Domäne modelliert werden und über Netzwerke miteinander kommunizieren. Sie sind somit eine besondere Form der serviceorientierten Architektur, bei der speziell Rücksicht darauf genommen wird, wie die Grenzen der Dienste geschnitten werden. Genau dieser Schnitt der Dienste ist es indes, der in der Praxis zu einigen Schwierigkeiten führt.

Bild 1 zeigt beispielsweise eine Aufteilung rein auf Basis der technischen Schichten. Eine fachliche Trennung ist daher in Bild 1 nicht zu erkennen. Bild 2 wiederum nimmt eine Auftrennung mit dem Ziel einer Skalierung vor. Wobei Auftrennung hier nicht das richtige Wort ist: Tatsächlich werden nur mehrere Instanzen der gleichen Software bereitgestellt. Das gleiche Vorgehen wurde eigentlich auch bei Bild 3 verfolgt. Hier gibt es keine Unterscheidung, warum die Logik und die Datenhaltung konkret in eigenen Diensten vorliegen. Vielmehr wird alles in einem großen Frontend-Monolithen wieder zusammengeführt.

Ein solcher Fall liegt beispielsweise vor, wenn man einen typischen Desktop-Client betrachtet, der verschiedene Backend-Services anspricht, wie es bei einer serviceorientierten Architektur der Fall ist. Bild 3 zeigt demnach keine Microservice-Architektur, sondern eine serviceorientierte Architektur. Microservices werden die Dienste erst durch ihre fachliche Auftrennung und Entkopplung voneinander, wie sie beispielsweise in Bild 4 zu sehen ist. Hier wurde absichtlich auf die Darstellung der Präsentationsschicht verzichtet, um die Komplexität der Abbildung zu verringern. Die Auftrennung der Dienste erfolgte im Beispiel anhand der Fachabteilungen, für die sie entwickelt wurden, das heißt die Verkaufsabteilung, die Personalabteilung und die Einkaufsabteilung. Je nachdem, wie umfangreich die abzubildenden Prozesse sind, könnten solche Dienste aber noch weiter zerlegt werden. So wäre beispielsweise ein Dienst für die Rechnungslegung vorstellbar, oder ein Dienst für das Beantragen von Urlaub.



Auftrennung und Entkopplung der Dienste (Bild 4)

Architekturerosion

Architekturerosion beschreibt, inwieweit sich die bestehenden Strukturen eines Softwaresystems von den Strukturen unterscheiden, die tatsächlich gebraucht werden beziehungsweise geplant sind. Der Prozess der Architekturerosion geschieht dabei meist schleichend über einen längeren Zeitraum hinweg und wird beschleunigt durch unzureichendes Requirements Engineering, einen hohen Zeitdruck bei der Implementierung, Unerfahrenheit der Projektbeteiligten und ein unzureichendes Bewusstsein für die innere Qualität von Software.

Als Entwickler nimmt man Architekturerosion meist in Form von technischen Schulden wahr. Personen, die nicht direkt an der Entwicklung beteiligt sind, erleben sie durch wiederkehrende Fehler, erheblichen Mehraufwand bei Änderungen und sinkende Gesamtproduktivität des Entwicklungsteams.

Da Architekturerosion meist sehr langsam vonstattengeht, sich langfristig aber verheerend auswirken kann, stellen Personen außerhalb des Entwicklungsteams deren Schweregrad meist erst fest, wenn die Software mit erheblichen Wartungskosten verbunden ist und die Architektur nur über entsprechend hohe Investitionen in umfassende Restrukturierungen wiederhergestellt werden kann. Aus diesem Grund sollten die Strukturen von Software bereits von Entwicklungsbeginn an durch entsprechende Codeanalysen und automatisierte Tests gegen Erosion abgesichert werden, und es sollte kontinuierlich in ihren Werterhalt investiert werden.

Risiken ...

Genau an dieser Stelle bemerkt man erneut eine der größeren Herausforderungen beim Thema Microservices: Wie groß ist Micro eigentlich, und was ist der ideale Serviceschnitt? Je feiner die Granularität, desto einfacher ist es, die Logik wiederzuverwenden, die in den Diensten gebunden ist, und desto verständlicher sowie austauschbarer sind die Dienste. Das sind alles Vorteile. Nachteilig ist aber, dass mit steigender Granularität auch die externe Kommunikation steigt. Mit externer Kommunikation ist hierbei all der Informationsaustausch gemeint, der außerhalb des betrachteten Bestandteils beziehungsweise Dienstes geschieht. Diese Kommunikation ist aber vergleichsweise langsam und kann im Rahmen von Cloud-Software auch Traffic-Kosten verursachen. Wäh- ▶

rend man innerhalb eines Prozesses, wie es bei einem Monolithen üblich ist, keinerlei Kommunikationskosten zu tragen hat, entstehen bei der Kommunikation innerhalb eines Rechenzentrums oder über Grenzen von Rechenzentren hinweg entsprechende Traffic-Kosten. Neben diesen deutlich wahrnehmbaren Kosten fällt auch auf, dass es viel schwieriger ist, ein Gefühl für das Gesamtbild der Software zu erhalten. Darüber hinaus ist sie vergleichsweise schwer zu analysieren. Im Fehlerfall kann man also nicht unbedingt einfach einen Debugger anwerfen und dann den Status der interagierenden Dienste auslesen.

Um dies zu verdeutlichen, wird in Bild 5 ein beispielhaftes Service-Mesh gezeigt. Diese Netze von Diensten zeigen an, welche Dienste mit welchen anderen kommunizieren, und verdeutlichen somit die Laufzeitabhängigkeiten der Dienste untereinander. Je mehr Dienste es gibt, desto komplexer ist das so entstehende Netz, und diese Netze können durchaus sensibel auf Störungen reagieren. Möchte in Bild 5 beispielsweise Service A eine Anfrage bearbeiten und benötigt er dazu Zuarbeiten von Service B, kann sich eine transitive Abhängigkeit bis hin zu Service X ergeben, je nachdem, wie die einzelnen Dienste umgesetzt sind. Diese Abhängigkeit wird aber erst ersichtlich, wenn man die Dienste gemeinsam testet. Prüft man jeden für sich und ersetzt die externen Abhängigkeiten durch Testdoubles, werden solche Zusammenhänge nicht auffallen.

Weiterhin kann man Bild 5 auch entnehmen, dass der Dienst B wie eine Spinne im Netz sitzt. In seine Richtung gehen viele Anfragen und von ihm werden viele Anfragen gestellt. Fällt dieser Dienst aus, könnte das gesamte Netz und damit auch das Softwaresystem nicht mehr korrekt arbeiten. In Summe betrachtet hat man also aufgrund einiger ungünstiger Entscheidungen ebenfalls eine stark gekoppelte Struktur, nur dass diese nun auch noch verteilt vorliegt.

Die Einfachheit der einzelnen Dienste wird somit durch die Komplexität ihrer Gesamtkomposition wettgemacht. Dem kann man entgegenwirken, indem man die Dienste im Betrieb kontinuierlich überwacht und möglichst schnell auf Fehler reagiert. Netflix geht hierbei sogar so weit, dass sie über ihr Tool Chaosmonkey [3] Server in der Produktionsumgebung vom Netz nehmen und prüfen, wie das Gesamtsystem darauf reagiert. Dies ist ein beeindruckendes Vorgehen, zugleich ist es aber auch mit erheblichen Betriebskosten verbunden, die von kleineren Unternehmen höchstwahrscheinlich nur schwer getragen werden können.

Macroservices

Neben den Abhängigkeiten zwischen den Diensten fällt auch eine Abhängigkeit zwischen den Entwicklerteams auf. Sind die Teams ungünstig geschnitten, wird dadurch automatisch auch der Kommunikationsaufwand zwischen den Teams erhöht. Dies spiegelt sich dann in regelmäßigen Synchronisierungsmeetings wider, bei denen Anforderungen geklärt und Spezifikationen ausgetauscht werden müssen. In solchen Fällen machen Anpassungen an einem Dienst in einem Team auch Änderungen an Diensten eines anderen Teams notwendig. Diese kooperativen Änderungen wirken auf Dauer arbeitsbehindernd für die Teams und können sie dementsprechend stark ausbremsen.

Aufgrund der Granularität hat man somit einen Leistungsverlust sowohl zur Laufzeit als auch in der Entwicklungszeit zu beklagen. Je feiner die Granularität der Dienste ist, desto

Deployment Unit versus Bounded Context

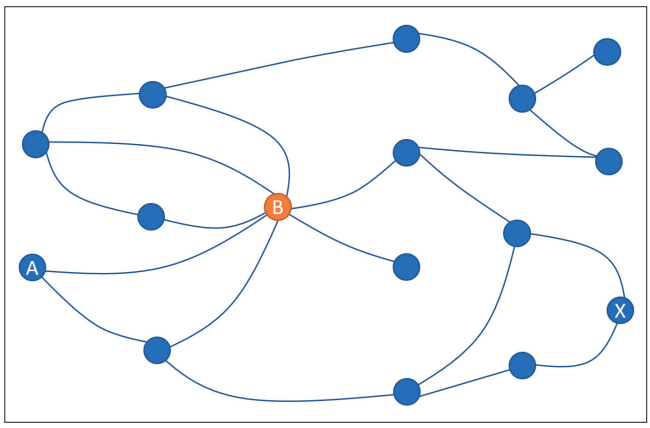
Microservices und Domain Driven Design (DDD) teilen sich viele gemeinsame Überzeugungen, wodurch einige Begriffe des einen gelegentlich im Kontext des anderen verwendet werden. Trotzdem sind sie nicht das Gleiche. DDD ist eine Methodik samt diverser Werkzeuge, die sich damit beschäftigt, Software auf Basis der tatsächlichen Bedarfe aus den Fachbereichen beziehungsweise Fachdomänen zu gestalten. Microservices als Architekturmuster können hierbei eine Möglichkeit sein, wie die Ziele erreicht werden, die durch DDD herausgearbeitet wurden.

Ein Begriff, der in diesem Zusammenhang oft falsch verstanden wird, ist der Bounded Context [12]. Er beschreibt einen fachlichen Zusammenhang, der in sich geschlossen ist und gegen andere fachliche Bereiche abgegrenzt werden kann. In der Umsetzung bedeutet dies meist, dass Bezeichnungen innerhalb des einen Kontexts ganz andere Bedeutungen haben als in einem anderen. So kann etwa eine Rechnung, die beim Einkauf eingeht, ganz andere Eigenschaften und Bedeutungen haben als eine Rechnung, die vom Verkauf ausgestellt wird.

Die Theorien hinter Bounded Contexts sprengen den Rahmen dieser Erklärung und sollen daher hier nicht weiter betrachtet werden. Es ist jedoch wichtig zu wissen, dass sie vor allem ein Mittel sind, um Software strategisch zu planen.

Bei der Umsetzung dieser Strategien kann dann auf Services oder Module zurückgegriffen werden. So können mehrere Microservices einen Bounded Context abbilden, damit den gleichen fachlichen Zielen dienen und das gleiche Daten-Model nutzen.

Es ist deshalb eine falsche Annahme, dass jeder Microservice selbst auch ein Bounded Context sei. Vielmehr wird die Bezeichnung einer Deployment-Einheit beziehungsweise einer Deployment Unit dem gerechter, was Microservices eigentlich sein sollen. Damit wird all das bezeichnet, was als Einheit gemeinsam veröffentlicht wird und alleinstehend veröffentlicht werden kann. Hierzu gehören beispielsweise ein Installer, Datenbankskripte und natürlich der kompilierte Quellcode. Ein Monolith ist aufgrund seiner Natur schon eine sehr große Deployment Unit. Microservices hingegen sind vergleichsweise kleine Units, die voneinander unabhängig deployt werden können. Sollten die Dienste nicht unabhängig deployt werden, sind es laut Definition keine Microservices mehr, und die Unit ist entsprechend größer zu betrachten, da sie mehrere Dienste umfasst.



Beispielhaftes Service-Mesh (Bild 5)

mehr verstärkt sich dieser Effekt. Aus diesem Grund wurde 2020 von Uber der Begriff der Macroservices geprägt [4]. Dies kam zustande, da Uber mehr als 4000 Microservices im Einsatz hat und sich die zuvor genannten Reibungsverluste in Teilen als so stark erwiesen, dass entschieden wurde, Dienste zusammenzufassen, die ohnehin sehr stark miteinander verbunden sind. Bezogen auf die Beispiele aus diesem Artikel bedeutet dies also, statt jeweils einen eigenen Dienst für die Urlaubsbeantragung, die Stundenerfassung und die Krankmeldungen bereitzustellen, nur jeweils einen Dienst für die Personalverwaltung aufzubauen und diesen über mehrere Endpunkte ansprechbar zu machen. Diese Dienste sind nach wie vor fachlich ausgerichtet und noch voneinander separat deployfähig, sie haben aber einen größeren Verantwortungsbereich.

Interessant zu wissen: Die Namensgebung Macroservices wurde von Uber deshalb genutzt, um den Unterschied zu sehr kleinteiligen Diensten hervorzuheben. Der Begriff selbst hat aber wenig positiven Nachhall erzeugt und wurde von weiten Teilen der Community eher so interpretiert, dass das eigentliche Konzept der Microservices von Uber nicht korrekt umgesetzt wurde. Inwiefern der Begriff in Zukunft überhaupt eine Bedeutung haben wird, bleibt abzuwarten.

... und Nebenwirkungen

Aber auch in anderen Bereichen lauern Herausforderungen. Da wäre zum Beispiel zu klären, wie die Daten zwischen den Services synchron gehalten werden können. Da jeder Service autark arbeiten können muss, verwaltet er auch seine eigenen Daten. Sind diese Daten nun aber im Netz verteilt, kämpft man als Entwickler möglicherweise mit Redundanzen und Inkonsistenzen. Auch bei der Gestaltung der Benutzerschnittstelle und dem Thema, wie die unterschiedlichen Bestandteile des UI in ein Gesamtkonzept eingebunden werden, stellen sich Fragen, die bei einem Monolithen meist schon implizit beantwortet sind. Wo ist das UI abzulegen? Wie wird es zusammengefügt? Wie kann es von unterschiedlichen Teams ohne Reibungsverluste bearbeitet werden? In Bild 3 ist dies gelöst, indem die Dienste von einem Client direkt verwendet werden. Alternativ dazu könnten aber auch die Dienste ihr eigenes UI mitliefern und als sogenanntes Micro-UI in einem entsprechenden Rahmen einbinden. Hierbei ist aber darauf zu achten, dass die Micro-UIs zueinander kompatibel sind, da der Nutzer sonst die Heterogenität der Technologien direkt vor sich sieht.

Damit sind wir auch bei einem anderen Thema, bei dem die optimistische Beschreibung der Microservices in der Realität an ihre Grenzen stößt. So klingt es natürlich für Entwickler verlockend, immer die Technologien einsetzen zu können, mit denen sie sich gut auskennen oder die aktuell besonders angesagt sind.

Die Summe der unterschiedlichen Technologien und des damit verbundenen Wissens macht das Gesamtsystem als solches aber schwerer beherrschbar, weil es zu einer Fragmentierung der Systemlandschaften innerhalb des Unternehmens führt. Das Geld, das während der Entwicklung durch eine höhere Produktivität der Entwickler eingespart wird, muss auf diese Weise in den entsprechenden Betrieb investiert werden, und diese Betriebskosten sind auf lange Sicht wesentlich höher als die anfänglichen Entwicklungskosten. Genau aus diesem Grund wird in vielen Unternehmen der Wahlfreiheit dann doch ein Riegel vorgeschoben.

Aufwände, die bei einem Monolithen somit über die reine Softwareentwicklung abgedeckt werden können, müssen in serviceorientierten Landschaften durch strategische Planung und unter Einbeziehung unterschiedlicher Stakeholder gelöst werden. Oder um es anders auszudrücken: In einem Monolithen kann Code sehr schnell und einfach geändert werden, mit allen Risiken und Nebenwirkungen. Bei verteilten Architekturen hat man als einzelner Entwickler nicht den Zugriff auf den gesamten Code und kann daher auch nicht einfach ohne Abstimmung etwas ändern. Für die Codequalität und -stabilität ist dies ein Vorteil, stellt aber an den organisatorischen Überbau des gesamten Softwareentwicklungsprozesses höhere Anforderungen, müssen doch Requirements, Softwareänderung und Releasepläne mit verschiedenen Stakeholdern abgestimmt werden, auch wenn das Vorgehen dies im Vorfeld so eventuell nicht vermuten lässt.

Vorsicht bei der Migration

Damit soll nicht gesagt sein, dass all diese Herausforderungen die Nutzung von Microservices unmöglich machen oder dazu führen, dass man sie meiden sollte. Es ist nur so, dass sie nicht bei den Optimalbeschreibungen auftauchen, mit denen man sich beim Thema immer wieder konfrontiert sieht. Gerade wenn man sich dem Thema der Microservices aus einem Monolithen heraus widmet, trifft man auf diverse Dinge, bei denen man eigentlich keine Probleme vermutet und dann mitten in der Umsetzung über ein komplett anderes Mindset stolpert. Erwähnenswert ist dies, weil das Thema der Microservices auch in den Etagen der Entscheider auf offene Ohren trifft. Ausgehend von den positiven Berichten von Uber, Netflix und anderen Großunternehmen werden vor allem die hohe Entwicklungsgeschwindigkeit und die Möglichkeit des autarken Arbeitens der unterschiedlichen Teams als große Vorteile wahrgenommen. Davon verspricht man sich kürzere Releasezyklen und somit eine schnellere Anpassbarkeit an neue Marktgegebenheiten. In der Praxis werden deshalb ▶

Entwicklungsteams damit konfrontiert, dass sie einen Monolithen in eine Microservice-Architektur wandeln müssen. Können in diesem Fall die Nachteile nicht eindeutig benannt und die damit verbundenen Risiken nicht adressiert werden, ist eine solche Migration fast sicher zum Scheitern verurteilt. Zumal sie möglicherweise gar nicht notwendig ist.

Vorteile des Monolithen

Wenn wir den Monolithen noch einmal genauer betrachten, fällt auf, dass er eben nicht nur Nachteile, sondern auch eine ganze Reihe von Vorteilen mit sich bringt. Zu diesen Vorteilen gehört neben der Möglichkeit, das Gesamtsystem einfacher zu erfassen, auch eine bessere Nutzung der vorhandenen Ressourcen, als dies bei Microservices der Fall ist. Damit einher gehen durchschnittlich geringere Antwortzeiten, weil ein Großteil des Datenaustauschs bei einem Monolithen innerhalb des gleichen Prozesses stattfindet. Es ist eben doch ein Unterschied, ob Daten über ein Netzwerk übertragen werden müssen oder ob sie einfach nur von derselben CPU verarbeitet und im selben Arbeitsspeicher abgelegt werden. Im Rahmen von Webapplikationen ist das selbstverständlich eher selten ein massiver Vorteil. Webapplikationen leben davon, dass sie entsprechend verteilt sind, und stellen besondere Anforderungen an Ausfallsicherheit und Skalierung. Gerade im Rahmen von Microsoft-Technologie hat man es aber nicht selten auch mit größeren Desktopsystemen zu tun. Zu diesen Desktopsystemen gehören beispielsweise auch Hardwareansteuerung und Ähnliches. In diesem Bereich ist die Nutzung vieler unterschiedlicher Dienste möglicherweise behindernd, weil Daten, die von der Hardware kommen, nicht schnell genug verarbeitet werden können. Es empfiehlt sich daher, in diesen Bereichen auf einen entsprechenden Monolithen zu setzen, da die Skalierbarkeit keinen Mehrwert bietet, eine effektive Nutzung der vorhandenen Ressourcen aber durchaus einen entsprechenden Vorteil. Doch wenn nun der Monolith seinerseits zu einer schlechten Architektur führt und auf längere Sicht schwer zu überblicken ist? Was kann man tun, um dieser Komplexität Herr zu werden? Man geht den gleichen Weg wie bei Microservices: Man teilt die fachlichen Bestandteile auf und separiert sie voneinander.

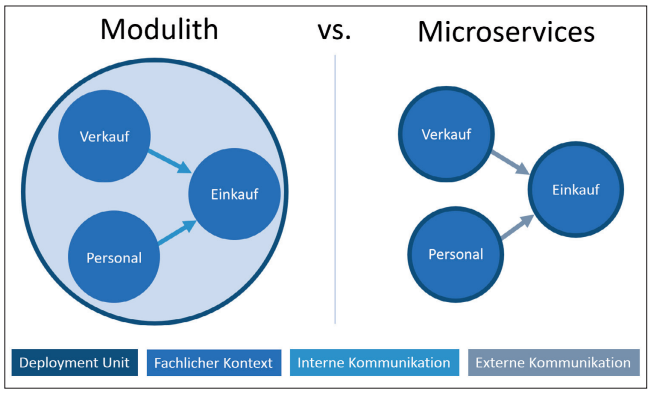


Bild: Oliver Drotbohm [7]

Vergleich von Modulith und Microservices (Bild 6)

Modulithen

Hinter dem Begriff des Modulithen verbirgt sich nichts anderes als ein modulorientierter Monolith. Diese werden in aller Regel als Deployment-Monolithen betrieben, in der Entwicklungszeit aber in stark separierte Softwaremodule zerlegt. Diese Module bilden geschlossene Fachkontexte ab und besitzen ihr eigenes Daten-Model, möglicherweise sogar eigene Datenbanken und Ähnliches. Sie kommunizieren nur über wohldefinierte Schnittstellen mit anderen Modulen und werden zur Laufzeit über einen Microkernel beziehungsweise ein entsprechendes Applikationsframework zur eigentlichen Applikation zusammengefügt. Typische Applikationsframeworks können dabei ASP.NET, aber auch das Prism Framework [5] sein, das im Zusammenhang mit WPF sehr gern verwendet wird.

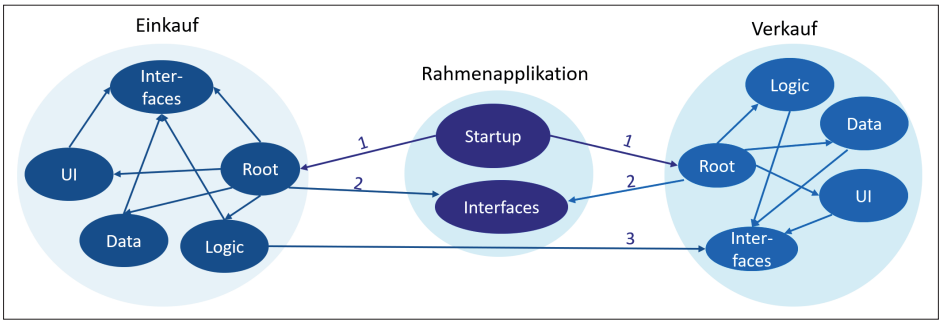
Interessant ist an dieser Stelle, dass es sich bei Modulithen nicht wirklich um etwas Neues handelt. Schon mit dem Composite Application Block [6] hat Microsofts Expertengruppe Patterns & Practices 2007 Unterstützung bei der Entkopplung der inneren Struktur von WinForms-Monolithen geboten. Die Erkenntnisse daraus hatten Einfluss auf das spätere WPF, welches mit dem Prism Framework ein eigenes Applikationsframework erhielt.

Bild 6 zeigt die Unterschiede zwischen Microservices und Modulithen noch einmal deutlicher [7]. Die Fachkontexte sind als Module umgesetzt, die einen unterschiedlich großen Umfang haben können. Alle Softwarebestandteile werden gemeinsam deployt. Dies bedeutet aber nicht, dass sie zur Laufzeit auch alle geladen werden. Vielmehr kann der Modulith anhand von Umgebungsparametern beim Start entscheiden, welche Module tatsächlich zu laden sind.

Die Kommunikation aller Module geschieht rein prozessintern, wobei sie selbst natürlich auch auf externe Ressourcen zugreifen können. Bei Microservices stellt jeder Dienst für sich ein fachliches Modul dar, das eigenständig deployt wird. Die Interaktion zwischen den Diensten geschieht als externe Kommunikation, da es sich bei den Diensten um getrennte Prozesse handelt.

Aufbau und Interaktion von Modulen

Es stellt sich noch die Frage, wie die Module aufgebaut sein sollten. Dies stellt Bild 7 dar, bei dem die Module Einkauf und Verkauf über eine Rahmenapplikation aggregiert und somit zur eigentlichen Applikation zusammengefügt werden. Das Vorgehen orientiert sich hierbei an umfangreichen Rich Clients und kann für Webapplikationen variieren. Dort gibt es aber dank ASP.NET sehr ähnliche Konzepte. Die Rahmenapplikation muss somit nicht nur über die beiden Bestandteile Startup und Interfaces verfügen, es sind auch die einzigen, die für die Module von Bedeutung sind. Denn beim Startup wird die Rahmenapplikation nach den Root-Elementen jedes Moduls suchen und diese initialisieren (Nummer 1). Die Wurzelelemente kennen alle Bestandteile des Moduls und können diese während des Starts bei den zentralen Diensten der Rahmenapplikation registrieren. Ein typischer Zentraldienst ist hierbei der IoC- beziehungsweise DI-Container. Dafür nutzen sie die Schnittstellen, die von der Rahmenapplikation bereitgestellt



Interner Aufbau eines Modulithen (Bild 7)

werden (Nummer 2). Meist bietet die Rahmenapplikation hier schon für das Wurzelelement eine Schnittstelle, mit der sich die Initialisierung steuern lässt und nach deren Implementierung während des Starts gesucht werden kann.

Intern sind die Module weiter zerlegt. Dabei kann die Zerlegung im einfachsten Fall über eigenständige Namensräume realisiert werden. Es kann sich bei komplexeren Modulen auch um völlig eigenständige Assemblies handeln. Die Zerlegung muss dabei nicht zwangsläufig wie in Bild 7 geschehen. Hierbei wurden die Geschäftslogik (Logic), die Datenzugriffsschicht (Data) und die Benutzerschnittstellen (UI) jeweils separiert, um eine technische Schichtung zu erzwingen, denn all diese Bestandteile haben keine direkten Abhängigkeiten zueinander, sondern nutzen nur Interfaces und fordern deren Implementierung über Dependency Injection an.

Sollte ein Modul Informationen oder Funktionalität eines anderen Moduls benötigen (Nummer 3), wird es ebenfalls nicht direkt auf dessen Logik zugreifen. Vielmehr nutzt es auch hier die öffentlich verfügbaren Schnittstellen und fragt deren Implementierung über die zentralen Dienste ab. Dabei entsteht auch eine gewisse Herausforderung, kann es doch gerade bei der Initialisierung zu Deadlocks kommen, falls Module Kreisabhängigkeiten zueinander aufweisen. Solche sind in jedem Fall zu vermeiden und sollten über statische Codeanalyse verhindert werden. Hierfür können, je nach Werkzeug, die Zugriffe aus bestimmten Namensräumen auf andere als schädlich markiert werden, wodurch dann beispielsweise der Build fehlschlägt. Als Werkzeuge können dabei beispielsweise NDepend [8], SonarQube [9] oder ArchUnitNet [10] dienen.

Fazit

Große Herausforderungen ergeben sich, wenn man versucht, von einer monolithischen Architektur zu einer Microservice-Architektur zu migrieren und dabei außer Acht lässt, dass Microservices nicht automatisch deshalb besser sind, weil die Dienste physisch voneinander getrennt sind. Damit das Vorgehen sein wahres Potenzial ausspielen kann, müssen der organisatorische Überbau und eine erhebliche Menge an Wissen für den Betrieb der Software aufgebaut werden. Nimmt das Unternehmen diese organisatorische Umstrukturierung nicht zeitgleich in Angriff, wird es nicht von den Vorteilen profitieren können. Ganz im Gegenteil, organisatorische Schwächen können sich in dieser Architekturform noch schwerwiegender auswirken, als es bei Monolithen der Fall ist. Ob sich der Aufwand also lohnt, ist nicht zuletzt davon abhängig, wie skalierbar die Software sein muss und wie viele Entwickler zeitgleich daran arbeiten sollen.

Mit dem Modulithen gibt es eine Architekturform, die einen Wechsel zwischen beiden Systemvarianten erlaubt und häufig auch als Zwischenstufe bei der Migration von der einen zur anderen genutzt wird. Die lose gekoppelten Strukturen eines Modulithen und die damit notwendige Infrastruktur verursachen aber einen gewissen Mehraufwand, der gerade bei kleineren Softwareprojekten als zu groß empfunden wird. Es sollte daher bereits vor der Entwicklung einer Software geprüft werden, wie umfangreich diese in Zukunft werden könnte, um dann von Beginn an entscheiden zu können, wie stark sie modularisiert werden sollte. Entscheidet man sich zu spät für eine Modularisierung, ist dies mit einem sehr hohen Restrukturierungsaufwand verbunden.

In jedem Fall muss aber noch einmal festgehalten werden, dass jede Form von Architekturmustern zu schlechten Ergebnissen führt, wenn sie falsch angewendet wird. ■

[1] Microsoft DevBlog, www.dotnetpro.de/SL2105Modulith1
[2] Sam Newman, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith, O'Reilly, 2019, ISBN 978-1-4920-4784-1
[3] Chaosmonkey, www.dotnetpro.de/SL2105Modulith2
[4] Macroservices, www.dotnetpro.de/SL2105Modulith3
[5] Prism Framework, https://prismlibrary.com/docs
[6] Composite Application Block, www.dotnetpro.de/SL2105Modulith4
[7] Unterschiede zwischen Microservices und Modulithen, www.dotnetpro.de/SL2105Modulith5
[8] NDepend, https://www.ndepend.com
[9] SonarQube, https://www.sonarqube.org
[10] ArchUnitNet, www.dotnetpro.de/SL2105Modulith6
[11] Entwurfsmusterbuch der Gang of Four bei Wikipedia, www.dotnetpro.de/SL2105Modulith7
[12] Bounded Context bei Martin Fowler, www.dotnetpro.de/SL2105Modulith8

Hendrik Lösch
ist Consultant und Architekt der ZEISS Digital Innovation. Sein Schwerpunkt liegt auf der Entwicklung und Bewertung von Software auf Basis von Microsoft-Technologien. Darüber hinaus ist er Sprecher und Autor.
hendrik.loesch@zeiss.com

dnpCode A2105Modulith