S&S

5.2019
Deutschland 9,80 €
Österreich 10,80 €
Schweiz 19,50 sFr

windows .developer

www.windowsdeveloper.de

Strategisches Design

Warum einfach? Es geht auch komplex!

Entwicklung von Microservices mit Microsoft .NET

Große Business-A...

Angular-Architekturen mit Nx ...

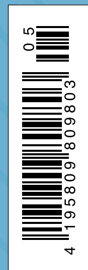
SONDERDRUCK FÜR

ACCSO
ACCELERATED SOLUTIONS

Wenn's etwas ... weniger sein darf

React und Redux kombiniert

© Plasteed/shutterstock.com, © ExpressVectors/shutterstock.com, © S&S Media



Je ne parle pas allemand

Wie Azure uns helfen kann, die Welt zu verstehen

▶ 28

Zurück zur Fachabteilung

Schnelle App-Erstellung mit Low-Code-Plattformen

▶ 72

WebAssembly und Sicherheit

Ist das sicher oder kann das weg?

▶ 92

© Dimj/Shutterstock.com

Entwicklung von Microservices mit Microsoft .NET

Warum einfach? Es geht auch komplex!

Wie zu erwarten zieht das heißdiskutierte Thema Microservices auch an Microsoft nicht vorüber. Grund genug, einmal näher zu betrachten, wie man als .NET-Entwickler bei der Entwicklung von Microservices mit Docker von technischer Seite unterstützt wird.

von Dr. Felix Nendzig

Mit „.NET Microservices: Architecture for Containerized .NET Applications" hat Microsoft nun ein mehr als dreihundertseitiges E-Book veröffentlicht, in dem das Unternehmen seine Sicht auf das Thema umfassend vorstellt und insbesondere die Benutzung von Docker bei der Entwicklung von Anwendungen mit Microservices-Architektur empfiehlt [1].

In diesem Beitrag soll Microsofts Sicht auf das Thema Microservices näher beleuchtet und mit Blick auf unsere eigene Erfahrung kommentiert werden. Weiterhin werden wir darauf eingehen, wie Docker bei der Entwicklung von Microservices genutzt werden kann.

Was ist eine Microservices-Architektur?

Das zentrale Erkennungsmerkmal einer Microservices-Architektur ist, dass sich die gesamte Anwendung aus einzelnen, voneinander unabhängigen Services zusammensetzt. Durch Dezentralisierung und Autonomiemaximierung soll auch bei komplexen Anwendungen eine gute Skalierbarkeit in der Entwicklung – gegenüber einer monolithischen Architektur – erreicht werden. Die Entwicklung kann in kleinen, unabhängigen Teams mit geringem Kommunikationsbedarf untereinander stattfinden. Um einen Vorteil gegenüber einer monolithischen Anwendung zu erreichen, sollten die Microservices jeweils folgende Bedingungen erfüllen:

- Ein Service erfüllt genau einen fachlichen Zweck und diesen vollständig
- Jeder Service umfasst sämtliche benötigte Schichten, Teams arbeiten unabhängig voneinander (vertikaler Schnitt)
- Asynchrone Kommunikation zwischen Services (keine zentral steuernde Einheit)

© Software & Support Media GmbH

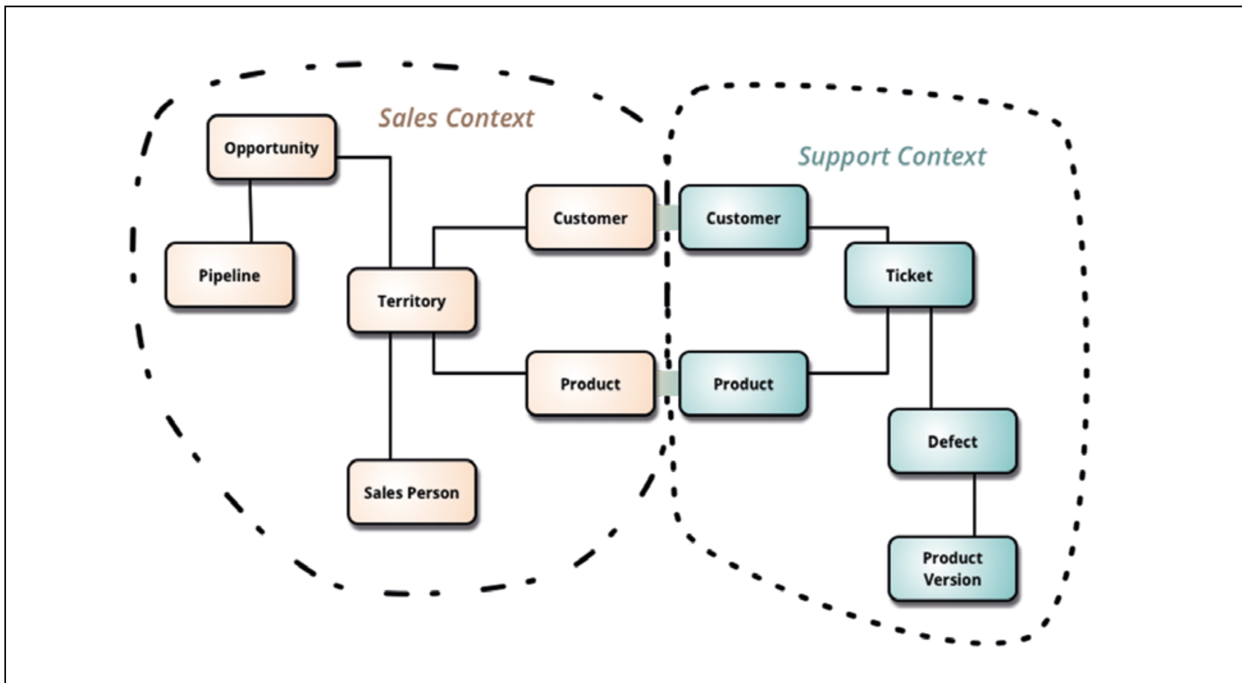


Abb. 1: Illustration zweier Bounded Contexts (Quelle: [2])

- Keine gemeinsamen Daten oder Code (shared Nothing)
- Jeder Service ist unabhängig deploybar
- Unabhängige Datenmodelle und minimale APIs zur Gewährleistung eigenständiger Entwicklungszyklen

Der Name „Microservices" impliziert, dass die einzelnen Services klein sein sollen. Diese Charakterisierung bezieht sich aber nicht auf die Lines of Code oder die Anzahl der Klassen je Service. Stattdessen bedeutet es, dass ein Service nur genau eine sinnvolle Funktion im Sinne der fachlichen Logik erfüllen soll. Diese Idee folgt dem Konzept des Bounded Context aus dem Domain-driven Design [2]. Danach wird ein großer fachlicher Themenbereich in kleinere Bereiche, die Bounded Contexts (**Abb. 1**), aufgeteilt, die jeweils ihre eigenen, eindeutigen Fachbegriffe benutzen (Ubiquitous Language). Ein Microservice überschreitet keinesfalls die Grenzen eines Bounded Contexts. Er implementiert maximal den gesamten Bounded Context, typischerweise aber nur einen Teilbereich.

Man sollte die kleinsten Microservices implementieren, die die oben genannten Bedingung erfüllen. Wird die optimale Größe unterschritten, wachsen die Abhängigkeiten der Services untereinander. Das verschlechtert die Performance der Anwendung zur Laufzeit und während der Entwicklung durch erhöhten Kommunikationsbedarf über Service- und Teamgrenzen hinweg. Der Schnitt der Microservices sollte deren innere Kohäsion maximieren und die Abhängigkeiten nach außen minimieren. Gemäß dem Gesetz von Conway [3] sollten sich die Service-Grenzen hierbei an der Organisationsstruktur des Unternehmens orientieren. Es kann sich auch lohnen, dieses Prinzip umzukehren, und die Organisation so anzupassen, dass man den aus Softwaresicht sinnvollsten Service-Schnitt wählen kann.

Bei richtiger Umsetzung erhält man ein skalierbares System lose gekoppelter Services, die unabhängig voneinander entwickelt und ausgeliefert werden können. Releasezyklus, Entwicklungs- und Betriebsumgebung können je Service passend gewählt werden. Der Testaufwand je Service sinkt, da er nicht übermäßig komplex ist und nur über Schnittstellen kommuniziert. Die Entwicklung in kleinen Teams erzeugt nur geringen organisatorischen Aufwand. Ausgehend von Amazons „2-Pizza-Regel" liegt die maximale Teamgröße bei fünf bis neun Personen [4].

Eine verteilte Anwendung bringt allerdings auch zusätzliche Schwierigkeiten mit sich, die bei falscher Vorgehensweise zu einer Verschlechterung gegenüber einem Monolith führen können: Es gibt keine zentrale Steuerung, sodass querschnittliche Aufgaben wie Service-übergreifendes Monitoring oder Logging zusätzlichen Entwicklungsaufwand mit sich bringen. Externe Tools wie der ELK-Stack [5], Zipkin [6], Jaeger [7] und AppDynamics [8] können hier Abhilfe schaffen. Die Durchführung von Service-übergreifenden Integrationstests oder Refactorings wird schwieriger. Transaktionen sind auf einzelne Microservices begrenzt. Das Prinzip des „shared Nothing" erzwingt, dass jeder Service seine benötigten Daten replizieren oder über Schnittstellen von anderen Services abfragen muss. Jedes Entwicklerteam hat zusätzlichen Aufwand durch erschwertes Debugging, eine eigene Fehlerbehandlung je Microservice, die Bereitstellung einer Delivery Pipeline etc. Ein ungünstiger Service-Schnitt führt zu ständigen Schnittstellenanpassungen, gemein-

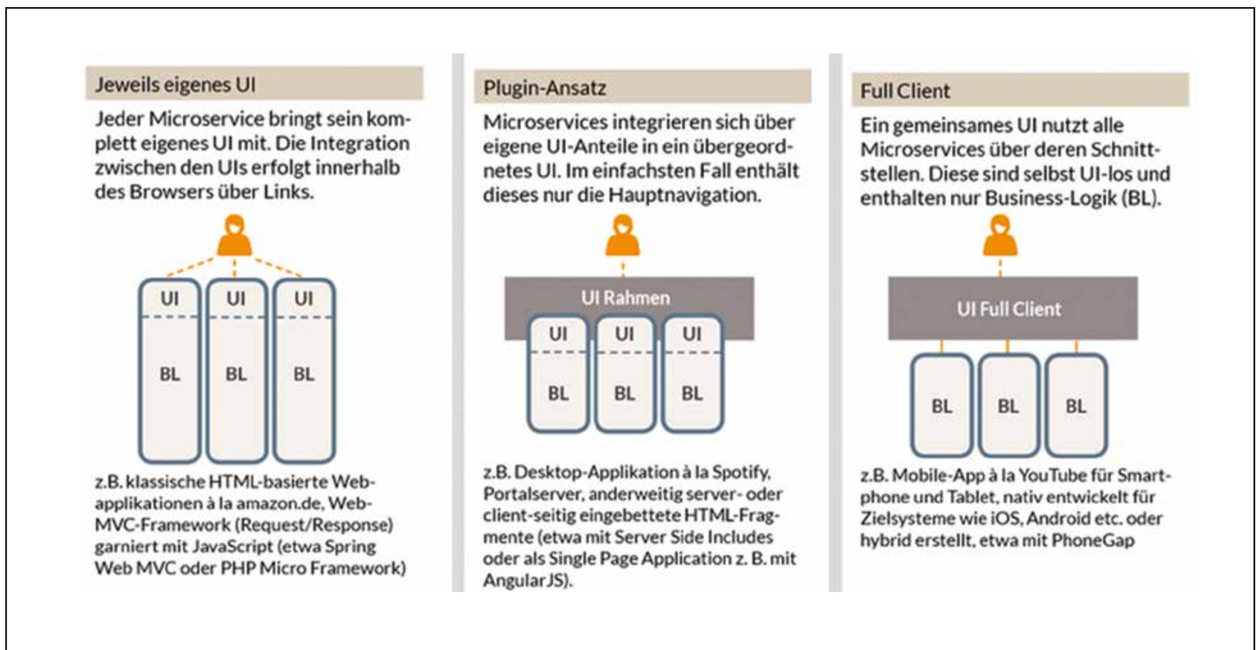


Abb. 2: Möglichkeiten zur Strukturierung des GUI in einer Microservices-Architektur (Quelle: [10])

samen Releases mehrerer Services oder der Auslastung mehrerer Services durch eine einzige Anfrage.

Stateful oder stateless Microservices?

In einer verteilten Anwendung kann eine variierende Anzahl von Service-Instanzen simultan laufen. Der Orchestrator (mehr dazu etwas später) kann jederzeit Instanzen auf einen anderen Knoten schieben oder die Anzahl laufender Instanzen an die momentane Auslastung anpassen. Man kann sich nicht darauf verlassen, dass eine spezielle Service-Instanz dauerhaft existiert. Das spricht gegen die Implementierung von stateful Services, deren Zustand im Arbeitsspeicher gehalten und jederzeit repliziert werden können muss. Für stateless Services stellt das kein Problem dar, da sie die Daten in eine externe Datenbank schreiben oder im Client oder einem Cachetool wie bspw. Redis cachen. Die externe Datenquelle wird je Service passend zu dessen Anforderungen gewählt.

Was ist aber mit Anwendungsfällen, in denen man Daten von verschiedenen Microservices sammeln oder anzeigen muss? Tritt ein solcher Fall auf, sollte man sich gut überlegen, ob der Service-Schnitt passend gewählt wurde und ob nicht eine Verschmelzung von Microservices die beste Lösung darstellt. Lässt es sich jedoch nicht vermeiden, muss man bei der Implementierung besonders darauf achten, dass die Autonomie der betroffen Microservices dadurch nicht eingeschränkt wird. Zudem kann sich die Aggregation von Daten verschiedener Services aufgrund erhöhter Kommunikation negativ auf die Performance des Systems auswirken.

Microsoft nennt als Alternative noch eine auf den Anwendungsfall zugeschnittene, denormalisierte Tabelle in einer separaten Datenbank, in die die Microservices jeweils ihre Daten schreiben. Aus Konsistenzgründen dürfen die abfragenden Services auf diese nur lesend zugreifen. Mit Blick auf die geforderte Autonomie ist diese Lösung allerdings kritisch zu sehen.

Inter-Service-Kommunikation

Wie jede verteilte Anwendung muss auch eine Microservices-Anwendung mit Teilausfällen des Systems umgehen können. Wie gut die Gesamtanwendung solche Ausfälle verkraftet, hängt maßgeblich von der Inter-Service-Kommunikation ab. Synchrone Kommunikation, bei der der Aufrufer abwarten muss, bis eine Antwort zurückgeliefert wird, kann bei lesendem Zugriff gegebenenfalls eine Lösung für die Kommunikation zwischen Front- und Backend sein. Innerhalb des Backends ist sie allerdings zu vermeiden, da sich Verzögerungen und Ausfälle des angefragten Service direkt auf aufrufende Services auswirken. Auf diese Weise wird schnell die Performance der Gesamtanwendung beeinträchtigt. Das Problem potenziert sich noch, wenn mehrere synchrone Aufrufe in Reihe stattfinden. Bei asynchroner Kommunikation wirkt sich der Ausfall eines einzelnen Microservice dagegen nicht so verheerend auf seine Nachbarn aus. Sie fördert zudem die Autonomie der einzelnen Microservices.

Da an der Verarbeitung eines Geschäftsprozesses im Allgemeinen mehrere asynchron kommunizierende, unabhängige Microservices beteiligt sind, ist es nicht möglich, stets Konsistenz im Sinne einer ACID-Transaktion sicherzustellen. Latenzen, Teilausfälle und unterschiedliche Verarbeitungsdauern können zwischenzeitlich Service-übergreifende, inkonsistente Zustände erzeugen (Eventual Consistency), deren Behandlung zusätzlichen Aufwand, z. B. in Form von Timeouts, Circuit Breakers und Bulkheads, erfordert [9].

© Software & Support Media GmbH

Ein Orchestrator ermöglicht eine einfache Steuerung der Anwendung inklusive Scheduling, Load Balancing und Gewährleistung von Verfügbarkeit und Robustheit.

Das GUI

Sollten Clients direkt mit den Microservices kommunizieren oder den Umweg über ein vorgeschaltetes API-Gateway nehmen? In der ersten Variante kann der Client über einen URL direkt den betreffenden Service (bzw. einen vorgeschalteten Load Balancer) anfragen. Jeder Microservice besitzt sein eigenes GUI oder ist in einem Plug-in-Ansatz für einen Teil des GUI innerhalb eines allgemeinen Rahmens verantwortlich.

Für kleine Anwendungen mit geringem Aufwand in der GUI-Entwicklung kann diese Variante ausreichen. Man bekommt jedoch schnell Probleme, wenn die Entwicklung der Benutzeroberfläche in mehreren Microservices gleichzeitig stattfindet und zudem noch für verschiedene Endgeräte entwickelt werden muss. Sind zum Aufbau einer Oberfläche mehrere unabhängige Abfragen nötig, erhöht das die Latenz. Ein besseres Ergebnis erhält man, wenn die Daten zuvor serverseitig aggregiert werden. Auch querschnittliche Anforderungen wie Sicherheit, Autorisierung, Logging und Caching möchte man nicht individuell für jeden Microservice, sondern gerne zentral implementieren.

Schaltet man ein API-Gateway zwischen die Clients und die Microservices-Landschaft, kann es das Routing sowie weitere querschnittliche Aufgaben übernehmen. Der Aufbau einer Oberfläche erfordert keine mehrfachen Roundtrips und Sicherheitsthemen können zentral implementiert und verwaltet werden. Die Weiterentwicklung der Anwendung wird erleichtert, da die Clients nicht mehr über die Existenz bzw. Aufteilung der Microservices Bescheid wissen müssen.

Man sollte jedoch nicht einfach ein einzelnes „monolithisches“ Gateway implementieren, sondern muss auch hier sowohl die Separation nach Geschäftsprozessen als auch nach Client-Apps einhalten (**Abb. 2**). Anderenfalls erzeugt man ungewünschte Kopplungen zwischen den Microservices.

Im Falle eines Web-UI mit dem Browser als Integrationskomponente, dürfte der direkte Ansatz am einfachsten umzusetzen sein. Die Entwicklung nativer Apps für diverse Endgeräte erzwingt dagegen den API-Gateway-Ansatz, bei der die Microservices im Backend einer übergeordneten Präsentationsschicht arbeiten (Backend for Frontend). Dieses Muster ist auch dann zu bevorzugen, wenn man besonders viel ausgeklügelte Logik in der Oberfläche unterbringen will. Zwar sind die Microservices ohne eigenes GUI nicht mehr unbedingt als vollständige Anwendungen zu betrachten, dennoch stellt das API-Gateway-Pattern keine Verletzung der Microservices-Philosophie dar. Die Unabhängigkeit der Services untereinander, einschließlich ihrer Datenhoheit, bleibt bei richtiger Umsetzung erhalten [11].

Entwicklung mit Visual Studio und Docker

Microsoft empfiehlt, die Vorteile von Docker bei Entwicklung, Deployment und Betrieb von Anwendungen mit Microservices-Architektur zu nutzen. Für jeden Microservice wird ein eigenes Docker Image erstellt, sodass zur Laufzeit schnell beliebig viele Instanzen in Form von Docker-Containern erstellt werden können. Für diesen Ansatz mit allen damit verbundenen Fragestellungen existiert eine umfangreiche Dokumentation [1], [12]-[15]. Docker-Container passen gut in das Konzept der Microservices, da sie unabhängiges Deployment und Skalieren unterstützen.

Microsoft bietet in Visual Studio und Visual Studio Code integrierte Unterstützung für die Entwicklung von Anwendungen, die in Docker-Containern laufen sollen. Unterstützt wird die Entwicklung mit .NET Framework, .NET Core und Mono in den Sprachen C#, F# und VB. Mit .NET Core entwickelte Microservices sind auf unterschiedlichen Plattformen lauffähig. Das modulare .NET Core ist zudem sehr leichtgewichtig und damit besser geeignet, containerbasierte Anwendungen mit möglichst kleinen Microservices zu entwickeln. Das klassische .NET Framework ist vorzuziehen, wenn die bereits bestehende Anwendung oder andere Abhängigkeiten das erfordern. Darunter würden zum Beispiel dringend benötigte Pakete oder Technologien wie ASP.NET Web Forms oder WCF fallen, die nicht in .NET Core verfügbar sind. Ab Visual Studio 2017 ist die Unterstützung für Docker bereits standardmäßig enthalten. Für ältere Versionen können die Docker Tools nachinstalliert werden [16].

Beim Anlegen eines neuen Projekts mit Docker Support wird ein passendes Dockerfile erzeugt. Zusätzlich muss man noch „Container Orchestrator Support“ aktivieren, um für das Projekt ein Docker Compose anzulegen bzw. zu aktualisieren. Es enthält mit der Datei *docker-compose.yml* die Konfigurationsdaten der Container und Deployment-Umgebung. So kann später für jeden Service ein separates Docker Image erstellt werden. Der Entwicklungsworkflow in Visual Studio bleibt sonst größtenteils unverändert. Beim Starten der Anwendung werden die Docker Images automatisch gebaut und

© Software & Support Media GmbH

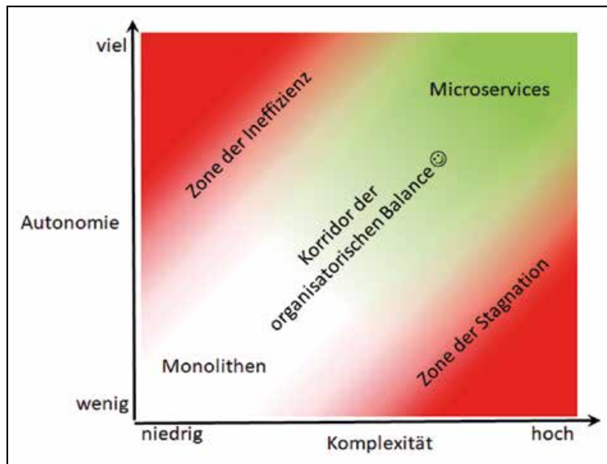


Abb. 3: Auswirkung von Autonomie und Komplexität bei der Entwicklung von Monolithen und Microservices (Quelle: [26])

direkt in Docker gestartet. Auch eine Multicontaineranwendung kann in Visual Studio debuggt werden. In Microsofts Visual-Studio-Dokumentation [17] wird erläutert, wie man eine Containeranwendung direkt in die Azure Container Registry deployen kann. Das Deployen und Testen sollte so früh wie möglich in Docker geschehen, da so die Produktivumgebung exakt reproduziert werden kann.

Um das System skalierbar zu machen, muss man eine Vielzahl Container gebündelt als Cluster managen und je nach Last die Anzahl der Service-Instanzen automatisch anpassen. Das wird erst durch einen Orchestrator ermöglicht, der die Komplexität eines Containerclusters abstrahiert und so das Management einer Multicontaineranwendung ermöglicht. Er erlaubt eine einfache Steuerung der Anwendung inklusive Scheduling, Load Balancing und Gewährleistung von Verfügbarkeit und Robustheit. Plattformen, die als Orchestrator agieren können, sind beispielsweise Azure Service Fabric, Kubernetes, Docker Swarm und Mesosphere DC/OS [18]-[22].

Wann verwendet man Microservices?

So lange die Prozesse eines Unternehmens oder eines Geschäftsbereichs durch eine Anwendung mit monolithischer Architektur abgebildet werden können, besteht kein Handlungsbedarf. Auch wenn die Anwendung schnell auf schwankende Last reagieren können muss, ist es nicht unbedingt nötig, den Flaschenhals im System als eigenen Service auszulagern. Nutzt man Docker, kann man einfach den gesamten Monolith in einem Container deployen und bei Bedarf zusätzliche Instanzen hochfahren. Wird das System aber zu komplex und steigt der Organisationsaufwand, dann wird es zunehmend schwerer, die Weiterentwickelung voranzutreiben, dabei das System wartbar zu halten und den Qualitätsanforderungen gerecht zu werden. Dann lohnt sich der Übergang zu einer Microservices-Architektur, um der Komplexität Herr zu werden.

Während komplizierte – aber nicht komplexe – Fachlichkeit gut durch monolithische Systeme abgebildet werden kann, lassen sich komplexe Systeme aufgrund einer Vielzahl wechselwirkender Teilsysteme nicht vollständig analysieren und nur teilweise steuern. Es ist daher nicht möglich, die optimale Architektur eines komplexen Systems vorab zu bestimmen. Diese muss viel mehr iterativ entwickelt werden, wobei praktische Erfahrungswerte einfließen können. Damit das möglichst unkompliziert vonstattengehen kann, sind z. B. kurze Releasezyklen, eine effektive Delivery Pipeline und geringer Kommunikationsbedarf durch kleine Teams nötig, wie es eine Microservices-Architektur ermöglicht [23].Daraus folgt allerdings auch, dass es gefährlich ist, Microservices auf der grünen Wiese zu entwickeln. Wählt man an kritischen Stellen den falschen Service-Schnitt, kann das folgenschwere Konsequenzen nach sich ziehen. Je besser man die Geschäftsprozesse und das Verhalten einer funktionierenden (aber schwer erweiterbaren) monolithischen Anwendung studieren kann, desto besser kann man einschätzen, wo der optimale Schnitt zu setzen ist. Es ist allerdings anzumerken, dass diese Frage in der Literatur [24], [25] kontrovers diskutiert wird (**Abb. 3**) .

Fazit

Der Einsatz von Microservices zielt darauf ab, komplexe Anwendungen, die als Monolith nur schwer zu managen wären, durch Dezentralisierung und Autonomiemaximierung skalierbar, wartbar und leicht erweiterbar zu machen. Jeder Microservice erfüllt dabei genau eine Funktion im Sinne der fachlichen Logik. Die wichtigsten Treiber in Richtung Microservices sind hohe Komplexität, hoher Innovationsbedarf sowie Leidensdruck bei der Skalierung. Microservices erben allerdings alle Probleme, die aus verteilten Anwendungen bekannt sind. Service-übergreifendes Monitoring und Logging aber auch Refactoring und Testen stellen eine Herausforderung dar. Zudem muss das System mit Eventual Consistency und Teilausfällen zurechtkommen.

Auf der technischen Seite bringt Docker bei Entwicklung und Betrieb von Microservices viele Vorteile wie etwa schnelles Deployment und einfache Skalierbarkeit mit sich. Die nahtlose Integration in den Entwicklungsworkflow stellt besonders für den .NET-Entwickler einen großen Vorteil dar.

Dr. Felix Nendzig studierte Physik am KIT und der Universität Heidelberg, wo er auch promovierte. Der Wechsel in die IT-Branche führte ihn zur Accso – Accelerated Solutions GmbH, wo er als erfahrener Senior Software Engineer tätig ist.

© Software & Support Media GmbH

Links & Literatur

[1] https://docs.microsoft.com/de-de/dotnet/standard/microservices-architecture/

[2] https://martinfowler.com/bliki/BoundedContext.html

[3] https://de.wikipedia.org/wiki/Gesetz_von_Conway

[4] https://www.informatik-aktuell.de/entwicklung/methoden/wie-gross-darf-ein-microservice-sein-und-ist-das-ueberhaupt-wichtig.html

[5] https://www.elastic.co/de/elk-stack

[6] https://zipkin.io

[7] https://www.jaegertracing.io

[8] https://www.appdynamics.com

[9] https://www.informatik-aktuell.de/entwicklung/methoden/microservices-sind-verteilte-systeme-asynchronitaet-und-eventual-consistency.html

[10] https://www.embarc.de/wp-content/uploads/2016/06/Architektur-Spicker3-Microservices.pdf

[11] https://www.informatik-aktuell.de/entwicklung/methoden/der-fachliche-schnitt-von-microservices-was-ist-das-was.html

[12] https://www.docker.com/community-edition

[13] https://docs.docker.com/docker-for-windows/

[14] de la Torre, Cesar; Microsoft: „Containerized Docker Application Lifecycle with Microsoft Platform and Tools“: https://azure.microsoft.com/de-de/resources/containerized-docker-application-lifecycle-with-microsoft-platform-and-tools/

[15] Lasker, Steve: „.NET Docker Development with Visual Studio 2017“: https://www.youtube.com/watch?v=hFCwfHo_Kbc

[16] https://docs.microsoft.com/aspnet/core/publishing/visual-studio-tools-for-docker/

[17] https://docs.microsoft.com/en-us/azure/vs-azure-tools-docker-hosting-web-apps-in-docker/

[18] https://azure.microsoft.com/documentation/articles/container-service-intro/

[19] https://docs.docker.com/swarm/overview/

[20] https://docs.docker.com/engine/swarm/

[21] https://docs.mesosphere.com/1.10/overview/

[22] http://kubernetes.io

[23] https://www.informatik-aktuell.de/entwicklung/methoden/microservices-nur-bei-ausreichender-komplexitaet.html

[24] https://martinfowler.com/bliki/MonolithFirst.html

[25] https://martinfowler.com/articles/dont-start-monolith.html

[26] https://www.informatik-aktuell.de/entwicklung/methoden/wann-und-fuer-wen-eignen-sich-microservices.html

Sie haben Fragen rund um Microservices mit .NET?
Accso ist Experte auf diesem Gebiet!
Kontaktieren Sie uns, wir helfen Ihnen gerne weiter.
www.accso.de/net

ACCSO
ACCELERATED SOLUTIONS

Accso - Accelerated Solutions GmbH

+49 6151 13029 0
info@accso.de
www.accso.de

Hilpertstraße 12 | 64295 Darmstadt
Im Mediapark 6a | 50670 Köln
Balanstraße 55 | 81541 München