Dr. Henning Sternkicker
(henning.sternkicker@de.ibm.com)
studierte Chemie an der RWTH Aachen. Nach dem Abschluss des Studiums und der Promotion begann er als technischer Berater bei Rational Software Deutschland. Seit der Übernahme von Rational durch IBM arbeitet er als Client Technical Professional vertriebsunterstützend für IBM Deutschland. Sein Tätigkeitsschwerpunkt im Bereich der IBM Entwicklungswerkzeuge ist, für Kunden Strategien und Architekturen für hybride Cloud-Lösungen zu entwickeln. Er ist spezialisiert auf Fragestellungen rund um IBM Bluemix als PaaS-Angebot und DevOps.

René Meyer
(Rene.Meyer@de.ibm.com)
arbeitet als Cloud-Architekt und DevOps-Spezialist bei der IBM Deutschland GmbH. Er interessiert sich besonders für Anwendungsentwicklung und deren Optimierung durch den Einsatz von hybriden Cloud-Konzepten. Der Schwerpunkt seiner Tätigkeit ist die Beratung von Unternehmen bei der Einführung von PaaS.

Microservices – Hype oder Segen?

Moderne mobile oder Web-Anwendungen stellen große Anforderungen an die verwendete Architektur. Die Anwendungen sollen in möglichst kurzen Zyklen produktiv gesetzt werden können, flexibel skalierbar sein und in den einzelnen Teilen unabhängig entwickelt werden können. Betrachtet man aus dem Blickwinkel dieser Anforderungen so manche Softwarearchitektur, so zeigt sich oft das genaue Gegenteil. Bestehende Anwendungen folgen zumeist einem monolithischen Architekturansatz und beinhalten eine komplexe verschachtelte Struktur, die viele verschiedene Geschäftsprozesse in einer einzelnen schwergewichtigen Komponente zu lösen versucht. So nehmen die ausgelieferten WAR- oder EAR-Dateien bei größeren Projekten im Java-Enterprise-Umfeld schnell eine Größenordnung von mehreren hundert Megabytes an. Selbst bei kleinsten Änderungen, die nur einen bestimmten Geschäftsvorgang betreffen, müssen diese Anwendungen in der Gesamtheit geändert bzw. neu entwickelt, gebaut, getestet und ausgeliefert werden. Die Komplexität einer solchen Anwendung in Verbindung mit den sich daraus ergebenden Abhängigkeiten zu Umgebungskonfigurationen, Infrastruktur- und Bereitstellungsprozessen macht Releaseverfahren – trotz der Einführung von Automationswerkzeugen in der Lieferkette – sehr aufwendig und nur schwer beherrschbar. Die Realität ist, dass solche komplexen Anwendungen oft nur zu festgelegten Release-Wochenenden (z. B. vierteljährlich) neu ausgeliefert werden. Eine Praxis, die sich mit den modernen Vorgehensweisen, wie agilen Softwareentwicklungsmethoden und notwendigen kurzen Releasezyklen, nur schwer vereinbaren lässt. Vor dem Hintergrund dieser Probleme haben sich in den letzten Jahren Microservices als neuer Trend in der Softwareentwicklung herausgebildet. Durch das Aufbrechen monolithischer Applikationen in geschäftsprozessorientierte und isolierte Microservices sollen die eben genannten Probleme bei der Entwicklung und dem Betrieb komplexer Applikationen adressiert und letztendlich gelöst werden.

Microservices – Grundlagen

Einige Publikationen bezeichnen Microservices auch als “fine-grained SOA“, also als eine fein granulare serviceorientierte Architektur (SOA) (siehe z. B. [Fow14], [Day15]). In einem solchen serviceorientierten Ansatz werden Applikationen aus einer Menge einzelner Services zusammengesetzt. Aber anders als beim SOA-Ansatz haben Microservices einen konkreten Bezug zu einem einzelnen Geschäftsprozess, bzw. setzen pro Service jeweils einen einzelnen Geschäftsprozess um.

Microservices zeichnen sich durch folgende, grundlegende Eigenschaften aus:

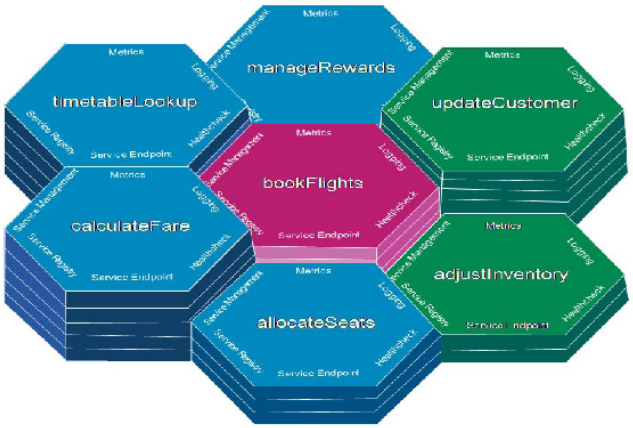

Abb. 1: Beispiel einer Microservice-Architektur für das Reservierungssystem einer Fluggesellschaft.

- Sie führen eine konkrete Aufgabe aus.
- Die Datenhaltung ist unter den Microservices klar abgegrenzt.
- Es gibt eine exakt definierte und dokumentierte Schnittstelle unter Ausnutzung von standardisierten Protokollen, wie z. B. REST.
- Die Teams sind multifunktional aus Entwicklung und Betrieb zusammengesetzt und haben eine überschaubare Größe.
- Sie werden als eine Einheit entwickelt, getestet und betrieben.

In Abbildung 1 ist ein Beispiel für eine Microservice-Architektur zu sehen. Eine Fluggesellschaft hat verschiedene Geschäftsprozesse, wie z. B. die Suchfunktion im Flugplan, das Buchen des Fluges, den Sitzplatz zuweisen, u. s. w. Jeder dieser einzelnen Vorgänge wird durch einen eigenen Microservice abgebildet. Die Höhe der unterschiedlichen „Service-Stapel“ deutet hier schon einen weiteren Vorteil einer solchen Architektur an: Je nach Last können neue Instanzen eines Dienstes gestartet werden.

Die Kommunikation der Services untereinander erfolgt durch standardisierte Schnittstellen und Protokolle. Dabei orientieren sich die Schnittstellen an dem Geschäftsvorfall im Sinne einer Business API. Dadurch und durch die klare Abgrenzung der Datenhaltung werden komplexe Abhängigkeiten vermieden, die sonst entstehen würden, wenn die Kommunikation über Datenstrukturen in einer Datenbank erfolgt.

Vor allem durch die klare Abgrenzung der Datenhaltung ist es möglich, für jeden Service die optimale Art der Datenspeicherung zu wählen. Waren früher relationale Datenbanken das Mittel der Wahl, so kann heute ganz nach dem konkreten Anwendungsfall der optimale Speicher ausgewählt werden, sodass entweder SQL- als auch NoSQL-Datenbanken, wie beispielsweise Key/Value-Stores, Document Stores, Graph-Datenbanken und andere zum Einsatz kommen [Edl10]. Für große Dateien kann alternativ die Nutzung eines Object Storages, wie beispielsweise OpenStack Swift oder Amazon S3, herangezogen werden.

Aber nicht nur in Bezug auf die Datenhaltung gibt es verschiedene Lösungen. Auch bei der Programmiersprache wird die für den jeweiligen Zweck am besten geeignete genommen. Hier spricht man davon, dass Microservices „language agnostic“ entwickelt werden.

In der genaueren Betrachtung des Beispiels des Reservierungssystems einer Fluggesellschaft wird deutlich, dass sich die einzelnen Services nur über die Schnittstellen aufrufen (siehe dazu auch Abbildung 2). Die Services an sich sind jedoch in Bezug auf die jeweilige Implementierung, Datenhaltung und Nutzung externer Services autark. Damit kann jedes Team, welches für einen Service verantwortlich ist, eigene Strategien wählen, um Aspekte wie Wartbarkeit, Skalierung und Kosten zu optimieren.

In unserem Beispiel ist in der Realität das Aufrufen des Flugplanes wahrscheinlich stärker nachgefragt als die Nutzung anderer Services. Das Team, das sich um die Umsetzung des Services für den Flugplan kümmern muss, wird deshalb z. B. Data Caches und andere Maßnahmen nutzen wollen und kann dies, unabhängig von den Entscheidungen anderer Teams, implementieren.

Wichtig ist jedoch, dass jeder Service für sich eine horizontale Skalierung ermöglicht. Erhöht sich die Last durch eine steigende Nutzung des Geschäftsprozesses, müssen umso mehr Instanzen des Dienstes automatisiert bereitgestellt werden. Dazu ist es erforderlich, die Microservices möglichst zustandslos zu entwickeln und Informationen über den Zustand des Vorgangs entweder auf dem Client oder durch die Nutzung von verteilten redundanten Memory Caches zu verwalten.

Durch die klare Abgrenzung der Microservices untereinander und die Konzentrierung auf einen Geschäftsprozess pro Service ergibt sich die Notwendigkeit für die unabhängige Auslieferungsfähigkeit der jeweiligen Komponente. Dem Entwicklungsteam wird die Verantwortung übertragen alle notwendigen Bestandteile in der Implementierung so zu paketieren, dass der jeweilige Microservice unabhängig und effizient in verschiedenste Umgebungen ausgeliefert werden kann.

Um dies zu ermöglichen, empfiehlt es sich, auf offene Containerformate zurückzugreifen, die auf möglichst vielen Plattformen zur Verfügung stehen und durch Standardisierung ein hohes Maß an Abstraktion von der konkreten Infrastruktur erzielen. Viele Unternehmen haben sich zur Bündelung der Kräfte in der Open Container Initiative (OCI), siehe auch [OCI], zusammengeschlossen.

Mit Docker-Containern und dem Cloud Foundry Platform as a Service (PaaS) Angebot, das ebenfalls auf einem Linux Containerformat beruht, seien an dieser Stelle nur zwei Vertreter beispielhaft genannt, die es durch einfach zu verwaltende Beschreibungen und Definitionen pro Applikation bzw. Microservices gestatten, die Laufzeitumgebung, Middleware und den Applikations-Code zu realisieren. Zusätzliche Aspekte im Betrieb wie z. B. Service-Orchestrierung, Workload-Management und Isolation verschiedener Umgebungen spielen bei der Auswahl einer solchen Plattform ebenfalls eine wichtige Rolle.

Von einer Microservice-Architektur erwartet man sich ebenfalls ein besseres Verhalten hinsichtlich Stabilität und Robustheit. Fällt ein Service aus oder hat ungewöhnlich lange Antwortzeiten, z. B. durch Probleme im Zusammenspiel mit externen Systemen, soll sich dieser Fehler möglichst nicht auf die gesamte Anwen-

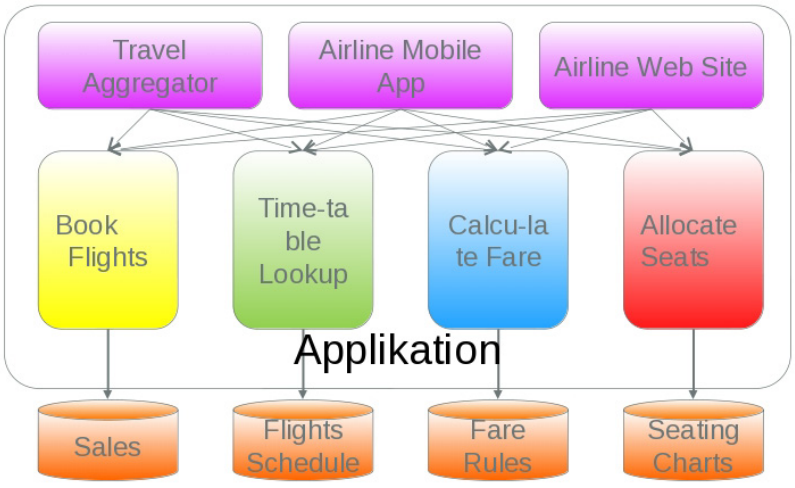

Abb. 2: Microservices mit unterschiedlicher Implementierung und Datenhaltung.

dung auswirken. Hier haben sich bestimmte Vorgehensweisen bzw. Muster etabliert.

Exemplarisch sei hier das „Circuit Breaker pattern“ erwähnt, bei dem es im Kern um das frühzeitige Erkennen fehlerhafter Services und den proaktiven Umgang damit geht [Rot14]. Der Grundgedanke ist, mögliche Servicefehler in der Architektur bereits im Vorhinein zu berücksichtigen, um das Fortpflanzen eines Fehlers in der gesamten Anwendung zu verhindern und dem Anwender einen alternativen Ablauf anzubieten („Design for Failure“-Prinzip).

Bewährte Verfahren für eine Microservice-Architektur

Im vorangegangenen Abschnitt sind einige Konzepte, die sich in der Umsetzung einer Microservice-Architektur ergeben, näher betrachtet worden. Diese, zusammen mit weiteren, haben sich in der Entwicklung moderner Web-Applikationen als besonders erfolgreich erwiesen und sind deshalb als bewährte „Faktoren“ anerkannt und in der sogenannten „12 Factor App“ als eine Art Best-Practice-Sammlung zusammengefasst worden [Wig12] [Kof14].

Die so zusammengetragenen Praktiken stehen somit auch exemplarisch für erfolgreiche Umsetzungen von Microservice-Architekuren. Dabei ist entscheidend, dass sich die Faktoren nicht nur um Vorgehensweisen kümmern, die die Architektur oder die Implementierung betreffen, hier wären z. B. die Faktoren “Codebase”, “Dependencies”, “Config”, “Backing Services” und “Port Binding” zu nennen, sondern dass auch explizit Faktoren aufgeführt werden, die den erfolgreichen Betrieb der Anwendungen beschreiben. Hier spielen vor allem “Concurrency”, “Disposability” und “Dev/Prod Parity” eine wichtige Rolle, weshalb wir uns im Folgenden auf diese Bereiche konzentrieren wollen.

Durch das Verfolgen des „Concurrency-Prinzips“, sprich das Behandeln des einzelnen Microservice als separaten Prozess, ist ein horizontales Skalieren einfach: Für Geschäftsprozesse, die gerade “gefragt” sind, wird eine entsprechende Anzahl an Instanzen des jeweiligen Microservices als neuer Prozess gestartet, um die Anfragen bedienen zu können.

Dies geht natürlich Hand in Hand mit dem “Disposability”-Faktor, der besagt, dass diese Prozesse zum einen schnell – sprich innerhalb von Sekunden – neu gestartet werden können, zum anderen aber auch schnell und sicher beendet werden können. Der Faktor “Dev/Prod-Parity” fügt dem Ganzen noch eine weitere Dimension hinzu. Dahinter steckt die Forderung, dass es für die Entwicklung möglich sein muss, Änderungen in kurzen Zyklen, eigenverantwortlich auch bis hin zur Produktion deployen zu können.

Für ein Operations-Team stehen diese Anforderungen in einem starken Gegensatz zu denen, die eine Anwendung mit einer monolithischen Architektur mit sich bringt. Durch die Komplexität eines solchen Monoliths ist die Produktionsstufe ebenfalls ein komplexes Infrastruktursystem, das als solches nicht einfach in frühen Test- oder gar Entwicklungsstufen repliziert werden kann.

Das Starten geht weit über “wenige Sekunden” hinaus. Eine Robustheit gegenüber geplanten oder plötzlichen Ausfällen ist daher nur über Hochverfügbarkeitslösungen gegeben. Mit anderen Worten, in einer monolithischen Architektur wird die Umgebung gehegt und gepflegt, fast so wie man sein Haustier zu Hause pflegt.

Der Vorteil liegt hier in erster Linie darin, dass man sehr gut über die Belange der Umgebung Bescheid weiß. Dies ist auch notwendig, denn so, wie man in der Regel nur ein Haustier hat, hat man als Operations-Team auch nur eine Produktionsumgebung, um die es sich zu kümmern gilt, um Ausfälle zu vermeiden. Sicherlich ein etwas überzogener Vergleich, aber er erleichtert uns in der Diskussion den nächsten Schritt.

Schauen wir nun als Operations-Team auf die Welt der Microservices mit den oben beschriebenen Anforderungen. **Abbildung 3** stellt schematisch dar, wie sich die Faktoren auf den Lebenszyklus von Anwendungen auswirken. Anstatt sich um eine oder wenige Umgebungen kümmern zu müssen, steht der Betrieb nun vor der Herausforderung, eine Menge von *x*-Microservices, von denen jeder eine fast beliebige Anzahl an Instanzen *y* haben kann, unter gleichen Bedingungen und Konfigurationen auf *z*-Umgebungen betreuen zu müssen.

Das ist der Punkt, an dem sich der “Segen” der Microservice-Architektur schnell in einen “Fluch” verwandeln kann. Aus betrieblicher Sicht ist es schlicht nicht möglich, die gleiche Aufmerksamkeit und manuelle Zuwendung aufzubringen wie bei einer klassischen Anwendung. Im Bereich der Infrastruktur wird in einigen

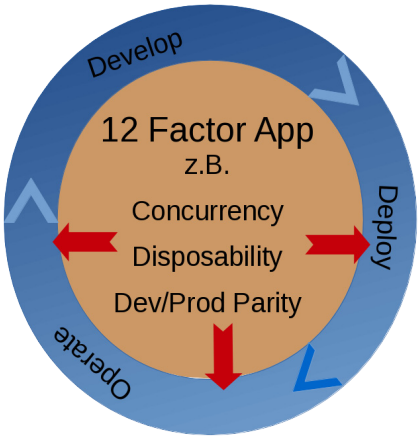

Abb. 3: Wirkungen ausgewählter Faktoren auf Auslieferung und Betrieb.

Veröffentlichungen [Sla14] gern von “Cattle vs. Pets” gesprochen, sprich nicht das einzelne liebgewonnene Haustier zu sehen und zu pflegen, sondern die große Herde als Ganzes zu hüten. Dabei muss man jedoch in Kauf nehmen, dass neue Tiere hinzukommen und andere in der gleichen Zeit sterben, was aber im Schnitt nichts an der Gesamtpopulation und an dem Zustand des Systems ändert.

Wenn also die große Menge an Services, Instanzen und Umgebungen nicht mehr manuell beherrscht werden kann, so muss versucht werden, die Situation durch Standardisierungen und Automatisierungen zu vereinfachen. In vielen Unternehmen und ihren Entwicklungsabteilungen wird mittels Automatisierung eine „Continuous Delivery Pipeline“ aufgebaut, in der es ausgehend von einem Build in der Entwicklung möglich ist, automatische Deployments in verschiedenste Umgebungen, ggf. bis hin zur Produktion, durchzuführen.

Hierbei ist zu berücksichtigen, dass dies nicht nur eine technische Umsetzung ist, in der neue Werkzeuge, wie z. B. zur „Deployment Automation“, eingeführt werden, sondern dass damit auch eine kulturelle Veränderung einsetzen muss. Hier kommt der Begriff “DevOps” auf, was letztendlich nichts anderes besagt als das Entwicklung und Betrieb viel enger zusammenarbeiten müssen. Gerade im Bereich der Microservices ist dieser Gedanke berücksichtigt, da das Team zur Entwicklung eines Microservices immer multifunktional zusammengesetzt ist und somit der Entwickler die Komponente bis hin zum Betrieb begleiten kann.

Aber eine solche Automation in den Deployments führt schließlich auch dazu,

Abb. 4: PaaS-Plattform für Microservices am Beispiel von IBM Bluemix.

dass immer öfter auch die notwendige Infrastruktur als Zielumgebung neu bereitgestellt werden muss. Gerade bei häufigen Deployments in Entwicklungs- und Teststufen zeigt sich sehr schnell, dass die klassische Server-Bereitstellung Grenzen hat. Selbst durch eine Virtualisierung stoßen auch große Unternehmen mit entsprechenden Rechenzentren an ihre Kapazitätsgrenzen.

Wie schon an anderer Stelle bemerkt, entscheiden sich Entwicklungsteams bei der Implementierung gern für Linux-Container-Technologien. Gerade in der Nutzung dieser Technologien im Zusammenhang mit Cloud-Lösungen unterstützen die 12 Faktoren in idealer Weise die Entwicklung und den Betrieb von Microservices.

Während bei virtuellen Maschinen die virtualisierte Hardware mit dem eigenen Betriebssystem gestartet werden muss bevor eine Anwendung wie ein Microservice deployed und gestartet werden kann, teilen sich bei der Containertechnologie, wie z. B. Docker, verschiedene Container die Ressourcen des Hosts, die aber trotzdem klar voneinander abgetrennt sind. Container sind leichtgewichtig und dadurch schnell zu starten und somit eine ideale Möglichkeit, um Microservices umzusetzen.

Aber Docker ist nur eine der Lösungen, die die Linux-Container-Technologie unterstützt. Bei Docker-Containern kommt in der Regel noch der Schritt der Definition und Erstellung des Images vor dem Deployment der eigentlichen Anwendung. Eine weitere Vereinfachung ergibt sich, wenn man als Nutzer auf PaaS-Angebote zurückgreift. Dort werden Zusammenstellungen von Laufzeitumgebungen, die gegebenenfalls auch mit weiteren Services komibiniert sind, als direkt einsetzbare Komponenten angeboten. Mit Cloud Foundry ist hier in den letzten Jahren die Basis für verschiedenste PaaS-Angebote entstanden.

Abbildung 4 zeigt ein Beispiel für ein solches PaaS-Angebot. IBM Bluemix [Blu1], das im Kern auf Cloud Foundry aufsetzt, bietet sogenannte "Instant Runtimes" für Java, Javascript, Python, Ruby, Swift und andere in einer Public Cloud an. Diese standardisierten Laufzeitumgebungen gestatten es, Web- oder mobile Anwendungen innerhalb weniger Sekunden in Produktion zu bringen.

Darüber hinaus bietet IBM Bluemix einen umfangreichen Katalog mit Services und API's aus den Bereichen Cognitive Computing, Data und Analytics, Internet of Things oder Security und Integration, mit denen Anwendungen jeglicher Größe bis hin zu Enterprise-Anwendungen erstellt und betrieben werden können. Zusätzlich bietet die Cloud-basierte Innovationsplattform durch das Einrichten und Betreiben von Docker-Containern (künftig auch virtuelle Server) mehr Flexibilität und Kontrolle.

Aus der Sicht des Operation-Teams ist ein PaaS-Angebot wie IBM Bluemix als zusätzliches Deployment-Ziel zu betrachten, das in den Aufbau einer automatisierten Delivery Pipeline mit einzubeziehen ist. Das gemeinsame Bestreben von Entwicklung und Betrieb, also dem DevOps-Team, sollte im Hinblick auf die bestmögliche Microservice-Architektur das Erreichen der größtmöglichen Standardisierung sein. Oder anders gesagt: Je weniger virtuelle Server oder selbst gepflegte Docker-Images verwendet werden, umso einfacher ist es, eine Automatisierung zu erreichen und umso größer wird die Anzahl an "deploybaren" Microservices.

Fazit:

Anwendungen in einer Microservice-Architektur zu entwickeln, kann ein Segen sein, wenn man sich an grundlegende Prinzipien hält. Entscheidend sind hier vor allem die Orientierung an einem Geschäftsprozess, die klare Abgrenzung der Datenhaltung, die kleinen, multifunktional zusammengesetzten Teams und die Deploybarkeit der einzelnen Services. Hierbei bieten die 12-Factor-App-Regeln ein gutes Rahmenwerk anhand dessen man die Tauglichkeit seiner Architektur immer wieder überprüfen kann.

Um zu vermeiden, dass aus den Vorteilen der Microservices auch ein Fluch werden kann, gilt es, die Entwicklung und den Betrieb unter dem DevOps-Ansatz zusammenzubringen. Die Möglichkeit, Microservice-Architekturen umzusetzen, ist durch die Schaffung agiler Entwicklungsteams sicherlich gestiegen. Wichtig ist jedoch auch diesen Teams die notwendigen Verantwortlichkeiten für agile Methoden konsequent zuzugestehen, sodass diese auch effizient eingesetzt werden können.

Im Bezug auf die Eigenverantwortlichkeit des Teams bedarf es des uneingeschränkten Rückhaltes durch das Management. Ist das Team mit allen notwendigen Rollen, begonnen mit der Entwicklung bis hin zum Betrieb, richtig zusammengesetzt, so sind schließlich auch alle Kompetenzen vorhanden, um kurze Releasezyklen bis hin zum Betrieb sicherzustellen.

Gerade diese zuletzt genannten Schritte stellen in vielen Unternehmen, zumindest aber in den Entwicklungs- und Betriebsabteilungen einen enormen Transformationsprozess dar. Hier können Erfahrungen von anderen, die diese Transformationen schon erfolgreich gemeistert haben, sehr hilfreich sein.

Eine solche Sammlung von agilen Praktiken ist unter dem Namen „Bluemix Garage Method" [Blu2] zu finden. Mit einer Vielzahl von Best Practices aus den Bereichen Agile und Lean Development, Design Thinking und DevOps bietet diese Methode einen idealen Einstieg für Entwicklungsteams, um die richtigen Baustei-

ne für das eigene Vorgehen zu finden. Rund um den Bereich der Änderung der Kultur sind hier die kleinen Bausteine zu setzen, die den Weg zum Ziel deutlich erleichtern.

Einfache, vordefinierte, sogenannte „Tracks“ [Tra], zum Beispiel für die Entwicklung von Cloud-native oder Web-Applikationen, geben einen guten Leitfaden, um die eigene Umsetzung zu finden. Um neben den kulturellen Herausforderungen solche multifunktionalen Teams erfolgreich zu machen, gilt es auch die technische Umsetzung zu meistern.

Kurze Releasezyklen gepaart mit einer großen Zahl von Microservices erfordern eine weitgehende Automatisierung und Standardisierung im Betrieb, wie auch die Verwirklichung des DevOps-Ansatzes im Team, die eine technische Umsetzung als konsequente Weiterführung erfordert. Es gilt eine kontinuierliche Lieferung durch eine „Delivery Pipeline“ aufzubauen, in der schon möglichst früh in der Entwicklung in produktionsähnlichen Umgebungen deployed und getestet werden kann.

Durch Automation des Lieferprozesses werden die Ergebnisse reproduzierbarer und die Tests effizienter und verlässlicher, sodass letztendlich die Forderung, qualitativ hochwertige Software in immer kürzeren Zyklen zu liefern, erfüllt werden kann. Dabei stellen Cloud-basierte Zielumgebungen, egal ob Private- oder Public Cloud-Angebote, die ideale Umgebung dar, um die „Dev/Prod-parity“ zu gewährleisten. Durch die Ausnutzung von schlanken und effizienten Linux-Container-Technologien, vor allem im Zusammenhang mit PaaS-Angeboten, sind die besten Voraussetzungen geschaffen, um vom „Segen“ einer Microservice-Architektur zu profitieren. ■

Literatur & Links

[Fow14] M. Fowler, J. Lewis, Microservices – a definition of this new architecture, siehe http://martinfowler.com/articles/microservices.html, 2014.

[Day15] Shahir Daya et. al., Microservices from Theory to Practice, IBM Redbooks, 2015.

[Edl10] S. Edlich et. al., NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken, Hanser-Verlag, 2010.

[Rot14] G. Roth, Stability Patterns applied in a RESTful architecture, siehe http://www.javaworld.com/article/2824163/application-performance/stability-patterns-applied-in-a-restful-architecture.html, 2014.

[Wig12] A. Wiggins, The Twelve-Factor App, siehe http://12factor.net/, 2012.

[Kof14] W. Koffel, 12-Factor Apps in plain english, siehe http://www.clearlytech.com/2014/01/04/12-factor-apps-plain-english/, 2014.

[Sla14] N. Slater, Pets vs. Cattle, https://blog.engineyard.com/2014/pets-vs-cattle, 2014.

[OCI] http://opencontainers.org.

[Blu1] ibm.biz/Bluemix-Microservices.

[Blu2] ibm.com/devops/method/.

[Tra] ibm.com/devops/method/category/tracks.