Keynote: Microservices Testen – Erfahrungsbericht und Umfrage

David Faragó (EclipseSource & QPR, dfarago [at] eclipsesource.com)
Dehla Sokenou (GEBIT Solutions GmbH, dehla.sokenou [at] gebit.de)

1 Einleitung

Ein System auf Basis einer Microservice-Architektur (MS-System) wird aus vielen kleinen, unabhängigen Prozessen komponiert, die Microservice (MS) genannt werden. Ein Microservice konzentriert sich darauf, genau eine Aufgabe möglichst gut zu lösen, ganz nach der Unix-Philosophie "Do one thing and do it well".

Microservices kommunizieren über eine sprachunabhängige Schnittstelle miteinander, um die umfangreicheren Businessaufgaben zu lösen. Somit ist die Microservice-Architektur eine Weiterentwicklung der Service-Oriented-Architecture [14].

Die Microservice-Architektur hat eine Reihe von Vorteilen, betrachtet man die bessere Kapselung und Modularisierung mit starker Entkopplung und strenger Orientierung an Verantwortlichkeiten. So ist ein Microservice im Idealfall genau für eine Aufgabe in allen Aspekten von Persistierung über Businesslogik bis hin zu Schnittstellen verantwortlich. Es ergibt sich nicht nur eine natürliche Architektur des Systems, sondern auch eine Zuordnung zu Entwicklungsteams, die jeweils einen oder mehrere Microservices in seiner Gesamtheit bereitstellen. Jeder Microservice kann unabhängig in der gewünschten Programmiersprache entwickelt und später im Einsatz individuell skaliert werden. Recoverability, Reliability, Continuous Delivery & Deployment sind Eigenschaften, welche durch eine Microservice-Architektur notwendiger, aber auch einfacher zu realisieren sind.

Gleichzeitig erkauft man mit einem MS-System eine höhere Komplexität, die neue Anforderungen an den Test des Gesamtsystems stellt, der ebenfalls komplexer ist [10, 2]. Die Teilnehmer der Umfrage [13] nennen die Cloud (49%), DevOps (38%) and Microservices (16%) als wichtige neue Technologien der Softwaretesting-Industrie in den nächsten 5 Jahren – Cloud-Testing (44%) sogar als stärksten Trend. Die Behauptung aus [10], dass viele Microservice-Projekte ein Defizit beim Thema Testen einräumen, lässt sich aus unserer Erfahrung noch heute beobachten. Manche verabschieden sich deswegen wieder von Microservice-Architekturen [15]. Natürlich testet jedes Team seine eigenen Microservices, mindestens mit Unit Tests, doch werden auch Aspekte wie das Zusammenspiel der Services im Gesamtsystem, Skalierung, Verteilung und Robustheit betrachtet?

> Umfrage: Kommen Sie in Berührung mit der Entwicklung von Microservices?

2 Fehlerursachen in MS-Systemen

Bevor Kapitel 3 Microservice-Teststufen erörtert, beschäftigt sich diese Kapitel mit der Klassifizierung von Software-Bugs in MS-Systemen. Hierzu ist uns leider nur sehr wenig Literatur und Statistik bekannt: So fehlt dieses Thema bspw. komplett in der Metastudie zu Microservices [1]. In [9] findet sich eine allgemeinere Klassifizierung von Fehlern in der Cloud, der typischen Ausführungsumgebung für MS-Systeme. Die Fehler sind nur in 18% der Fälle auf Scheduling und Softwarefehler zurückzuführen – ein Hinweis darauf, wie vielfältig die Fehlerursachen und wie komplex die Ausführung von MS-Systemen sein können (beispielsweise werden Resource-Utilization und zehn verschiedene Software-Technologien als Kriterien genannt). Die Komplexität erhöht sich auch durch die gestiegene Anzahl an beweglichen Teilen (genannt “ephemeral” oder “Moving Parts”). Dies ist Hardware und Software, die (in Analogie zu sich physisch bewegenden Teilen in Maschinen) zur Laufzeit neu verteilt werden kann (hinzukommen, wegfallen, umziehen). In MS-Systemen kann dies jederzeit mit Hardware, einzelnen Microservices und Infrastrukturprogrammen passieren, weswegen die Software reentrant sein sollte, d.h. damit umgehen können sollte.

Auch aus unserer Erfahrung spielen Fehler innerhalb eines einzelnen Microservice eine untergeordnete Rolle: Abb. 1 zeigt eine grob geschätzte Fehlerklassifizierung, basierend auf unserer Erfahrung aus zwei Unternehmen mit je einem Projekt, sowie mehreren Gesprächen mit weiteren Unternehmen. Wir haben in der Fehlerklassifizierung nicht unterschieden zwischen Erstentwicklung und DevOps, d.h. zwischen dem Zeitpunkt der reinen Entwicklung und dem der Wartung & Weiterentwicklung, denn der Personnel-Gap behindert die Dev/Prod-Parity (vgl. [18] und Kap. 4.1).

> Umfrage: In welche Klassen fallen Ihre Fehler?

10%	15%	20%	25%	30%
funktional in einem MS	weil (z.B. stateful, non-reentranter) MS unerwartet bewegt wurde	in der Interaktion mehrerer MSs	in der Cloud-Infrastruktur (z.B. Kubernetes, Docker, ELK-Stack)	in der Cloud-Konfiguration (falsches Probing, zu wenig oder schlecht verteilte HW-Ressourcen)

Abb. 1: Grob geschätzte Klassifizierung von Softwarebugs in MS-Systemen

3 Aufteilung der Tests in Teststufen

3.1 Bisherige Aufteilung

Vor 10 bis 20 Jahren war die Test-Pyramide aus Abb. 2 die gewünschte Teststruktur in der Testautomatisierung. Damals konnte man noch häufig die Test-Eistüte antreffen, welche die Test-Pyramide auf den Kopf gestellt hat (d.h. wenig Unit Tests und viele E2E Tests), heute ist die Test-Pyramide jedoch etabliert.

Ein **Unit Test** überprüft die Korrektheit einer kleinstmöglichen Einheit.Durch diesen Fokus kann ein Unit Test zur gleichen Zeit wie der Produktiv-Code geschrieben und sehr schnell ausgeführt werden, womit eine frühzeitige Fehlerfindung möglich ist. Sociable Unit Tests verwenden weiteren Produktiv-Code, deren Eigenschaften aber nicht explizit geprüft werden – die Korrektheitsprüfung (Orakel) konzentriert sich auf das Interface der eigentlichen Unit [6]. Solitary Unit Tests verwenden keinen weiteren Produktiv-Code, sondern ersetzen all diese Abhängigkeiten durch Test-Doubles [6]. Durch die Verwendung von Test Doubles deckt das Orakel auch die interne Interaktion der Unit mit seinen Abhängigkeiten ab. Durch die Isolation der Microservices und die meist wenigen, klar definierten vertragsbasierten Schnittstellen zu anderen Microservices, kommen die Solitary Unit Tests meist mit wenigen und einfachen Test-Doubles aus, weswegen Microservices in der Regel sehr gut durch Unit Tests (und weitergehend auch durch Contract Tests) abgedeckt sind. Dies ist neben der einfachen, isolierten Aufgabe eines Microservices ein weiterer Grund für den eher geringen Prozentsatz an funktionalen Fehlern in der Produktion (vgl. Kap. 2).
Tools: z.B. xunit, AssertJ, Mocking-Frameworks.

Ein **Integration Test** führt den Code mehrerer Units aus und überprüft, ob deren Interaktion ein korrektes übergeordnetes Verhalten erzeugt. Häufig wird er verwendet, um den External- und Persistence-Teil [4] eines Microservices zu testen und enthält Data-Stores, Caches und Teile aus unterschiedlichen Microservices. Dadurch ist die Definition und Ausführung eines Integration Tests für ein MS-System wesentlich schwieriger als für ein monolithisches System (vgl. auch [5]). Je mehr bewegliche Teile ein Test enthält und abdecken soll, desto wichtiger ist es, ihn möglichst in identischer Umgebung wie das Produktivsystem (Produktion) laufen zu lassen (vgl. Kap. 2, 4.1).
Tools: z.B. Hoverfly, Arquillian, testcontainers.org.

Ein **Ende-zu-Ende (E2E) Test** führt das Gesamtsystem aus und prüft, ob dessen Verhalten den Anforderungen entspricht, unabhängig vom Schnitt der einzelnen Microservices. Da hier alle beweglichen Teile ausgeführt werden, ist es für E2E Tests am wichtigsten, in der identischen Umgebung wie Produktion zu laufen. Da Integration und E2E Tests in Microservice-Umgebungen wesentlich komplexer sind und oft noch nicht viel Wissen und Erfahrung dazu vorhanden ist, werden diese häufig weniger intensiv und methodisch entwickelt. Diese stiefmütterliche Behandlung führt zu einer Test-Pyramide wie in Abb. 3, welche eher einem Test-Hut gleicht.
Tools: z.B. Cucumber, Cukes in Space!, Arquillian Cube, Selenium, REST-assured, Observatiliby-Tools.

3.2 Gewünschte Aufteilung

Abb. 4 zeigt die erweiterte Test-Pyramide mit der zusätzlichen Teststufe Component Tests, wie sie z.B. von [4] vorgeschlagen wird, sowie fünf Skalen zur Beurteilung von Vor- und Nachteilen der Teststufen.

Zur ursprünglichen Test-Pyramide kommen noch **Component Tests** hinzu: Tests auf der Ebene eines einzelnen, gesamten MS (Achtung: ISTQB definiert den Komponenten-Test hingegen synonym zum Unit Test). Im Vergleich zu niedrigeren Teststufen wechselt hier die Perspektive der Tests zum Konsumenten. Es wird entweder die vollständige API des MS getestet oder die Summe aller konkreten Verträge zwischen je einem konkreten Konsumenten und dem MS (sog. Consumer-driven Contract Tests, CDC). Da CDCs häufig nicht tiefgehenden, sondern nur die Parameter bestimmter Aufrufe testet, stufen manche CDCs als Integration Tests ein.
Tools: z.B. Arquillian, pact, Hoverfly, REST-assured, Moco, Pretender, mountebank.

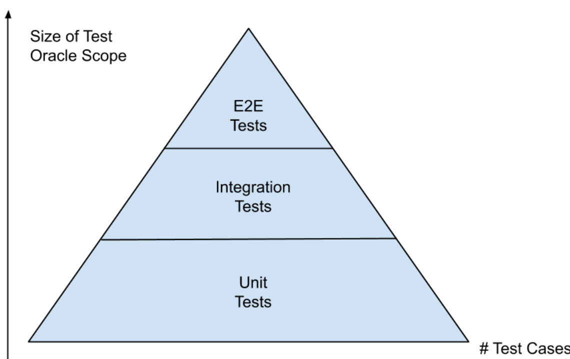

Abb. 2: Ursprüngliche Test-Pyramide

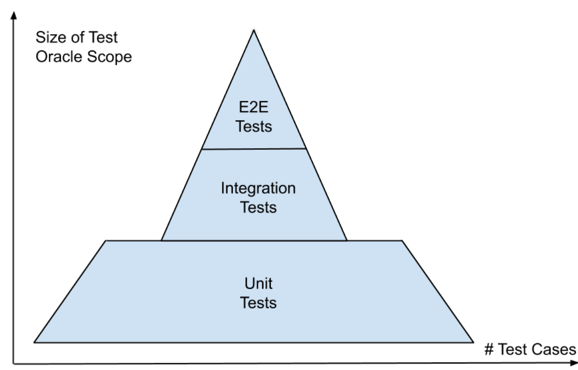

Abb. 3: Status quo für Microservices: der Test-Hut

Abb. 4 enthält folgende **Beurteilungsskalen**, um die Vor- und Nachteile der Teststufen zu beurteilen:
Oracle-Scope. Die Größe des Codes vom System-Under-Test, der vom Orakel direkt geprüft wird. Diese Skala kann als Definition für die jeweilige Teststufe angesehen werden, so dass sich die anderen Skalen daraus ableiten lassen.
Execution-Environment. Da ein Unit Test nur eine Funktion eines einzelnen MS testet, wird kein anderer MS benötigt und der Unit Test kann ohne Nachteile isoliert ausgeführt werden. Dies ist bei höheren Teststufen nicht mehr möglich (vgl. Kap. 3.1), mindestens bei E2E Tests sollte das Gesamtsystem in Produktion ausgeführt werden. Durch höheren Aufwand bei größerer Execution-Environment wirkt sich diese Skala direkt auf die folgenden aus.
Test-Runtime. Die Laufzeit eines Tests beeinflusst stark, wann im Entwicklungsprozess der Test ausgeführt wird: Unit Tests meist während dem Schreiben von Code, E2E Tests häufig nur am Ende eines Sprints. Diese Skala ist also ein wichtiger Faktor zur frühzeitigen Fehlerfindung.
Cost of finding & fixing bugs. Oracle-Scope, Execution-Environment und Ausführungszeitpunkt im Entwicklungsprozess beeinflussen den Zeitaufwand zur Fehlerfindung und Fehlerbehebung (sowie weitere Kosten, insbesondere wenn die Fehler bis zum Kunden durchdringen).
People involved. Da ein Team einen Microservice in seiner Gesamtheit bereitstellt, ist für dessen Unit Tests nur dieses Team involviert. Je größer der Oracle-Scope und die Execution-Environment, desto mehr Teams sind involviert.

Deswegen sollte ein Bug in der niedrigst-möglichen Teststufe entdeckt werden. Diese ist jedoch bei MS-Systemen meist höher als in monolithischen Systemen (vgl. Kap. 2 und 4.1), weswegen sich der Fokus auf höhere Teststufen verschiebt. Hinzu kommt noch, dass ein Microservice durch die starke Entkopplung und den Fokus auf eine einzige Aufgabe weniger komplexe Logik enthält als ein monolithisches System, weswegen innerhalb eines einzelnen Microservices weniger funktionale Fehler auftreten – und damit weniger Fehler von Unit Tests zu entdecken sind. Komplexe Business-Logik ergibt sich durch das Zusammenspiel mehrerer Services. Die Fehler im Zusammenspiel müssen durch höhere Teststufen entdeckt werden. Je mehr Microservices involviert sind im Zusammenspiel und je wichtiger die Ausführung der Tests im Gesamtsystem (vgl. Kap. 4.1), desto höher die Teststufe. Somit gibt es einen Trade-Off zwischen obigen Skalen einerseits und der Fehlerklassifizierung (Kap 2) und Dev/Prod-Parity (Kap. 4.1) andererseits. Dadurch lässt sich eine neue gewünschte Teststruktur in der Testautomatisierung von Microservices ableiten: das Test-Rechteck, wie in Abb. 5 dargestellt.

| Umfrage: Wieviel % testen Sie auf jeder Teststufe? |

4 Testen im Produktivsystem

Das vorherige Kapitel hat gezeigt, dass die höheren Teststufen sowohl in Produktion als auch leichtgewichtiger ausgeführt werden können: viele Tools (siehe [2]) ersetzen Produktion für eine schnellere, möglicherweise lokale Ausführung, mit einfacherem Test-Setup.

4.1 Dev/Prod-Parity

Dev/Prod-Parity [18, 11] empfiehlt jedoch, auf Produktion zu testen, damit Entwicklungs-, Test- und Produktivsystem möglichst ähnlich sind, d.h. der Tools-Gap möglichst gering gehalten wird. Ein Test unter realen Bedingungen erleichtert u.a. die Aufdeckung der aus unserer Erfahrung vermehrt auf nicht funktionale Ursachen zurückzuführende Fehler (vgl. Kap. 2). Zwar erschwert die komplexe Umgebung die Fehlerfindung, allerdings gibt es moderne Techniken zur Unterstützung.

| Umfrage: Wie wichtig ist für Sie Dev/Prod-Parity? |

4.2 Stages

Um zu vermeiden, dass beim Testen auf Produktion die Kundensysteme in Mitleidenschaft gezogen werden, schlägt [8] das sogenannte Green-/Blue-Deployment vor. Dabei ist zunächst die blaue Umgebung für die Produktion und die grüne für den Test vorgesehen. Deployment und Test erfolgen grundsätzlich auf der grünen Umgebung. Nach dem Test werden die Stages vertauscht – geht etwas schief, kann schnell wieder zurück auf die blaue Umgebung gewechselt werden. Läuft alles stabil, zieht die blaue Umgebung nach und wird zur nächsten Testumgebung. Weiter-

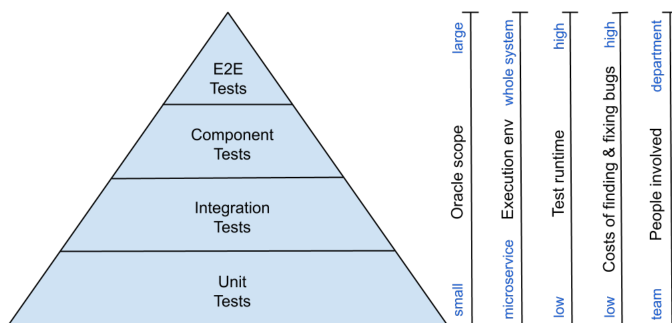

Abb. 4: Erweiterte Test-Pyramide für Microservices

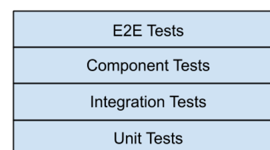

Abb. 5: Gewünschte Aufteilung: das Test-Rechteck

führende Techniken wie Canary-Deployment weichen den strengen Wechsel auf und sind damit flexibler.

4.3 Moderne Testtreiber

Da viele Bugs in MS-Systemen nicht funktional sind (vgl. Kap. 2), kommen auch alternative Testtreiber vermehrt zum Einsatz, wie Fuzzing, Fault Injection und Chaos Monkey Testing. Diese haben das Ziel, Fehlverhalten des Systems zu provozieren oder Fehler in das System zu induzieren, um die Fehlertoleranz zu prüfen. Ein Beispiel ist das gezielte Abschalten einiger Microservices, um die Reaktion des Gesamtsystems auf solch eine Situation zu beobachten [4].

4.4 Moderne Orakel und Observability

Durch die neue Herausforderung, das komplexe Zusammenspiel der Microservices bei der Fehlersuche und bei anderen Analysen nachzuvollziehen, fällt oft die Aussage, dass **Monitoring** das neue Testen sei [4]. Durch die Nutzung der gesammelten Informationen ergeben sich viele Vorteile: sie vereinfachen die Fehlerfindung, können aber auch für Recovery[1] verwendet werden. Die Monitoring-Aktivitäten (Messen, Einsammeln und Verarbeiten unterschiedlicher diagnostischer Signale wie z.B. Metriken, Traces, Logs, Events, Profile) für bestimmte Zwecke führt zur **Observability**: dies bedeutet, dass der interne Zustand eines Systems nur durch seine Ausgaben möglichst exakt inferiert wird. Dabei sollten beliebige, zur Entwicklerzeit noch nicht bekannte Fragestellungen zur Laufzeit beantwortbar sein [12]. Wichtig ist, dass im gesamten Code strukturiert geloggt wird, um nach den richtigen Dimensionen tracen zu können, insbesondere nach Business-Aspekten.

Bei **passivem Testen** beobachten die Orakel das Gesamtsystem und werten es zur Laufzeit für Testzwecke aus, um seinen aktuellen Zustand jederzeit zu beurteilen und ggf. Gegenmaßnahmen ergreifen zu können. Es werden Zielzustände definiert und deren Einhaltung anhand von Logs, Systeminformationen, versendeten Nachrichten u.ä. überwacht. In Produktion spielen die Endbenutzer die Rolle der Testtreiber ("every deploy is a test" [12]).

| Umfrage: Verwenden Sie Observability/Monitoring? |

5 Fazit

Wegen der Art und Häufigkeit der Fehler in Microservice-Architekturen sollten wir mehr Tests auf höherer Teststufe durchführen und diese im Gesamtsystem ausführen, woraus sich das Test-Rechteck ergibt. Dies entspricht der Dev/Prod-Parity. Andererseits entstehen im Microservice-Umfeld gerade viele Testwerkzeuge, die einen anderen Ansatz verfolgen: die Tests leichtgewichtiger auszuführen als im Gesamtsystem, und trotzdem möglichst viele Fehler finden. (Wann) wird es diesen Tools gelingen, den Tool-Gap so stark zu verkleinern, dass der Dev/Prod-Parity obsolet wird? Wenn die Tests technisch sauber in Testtreiber und Orakel aufgeteilt werden, können die Orakel für mehrere Testtreiber wiederverwendet werden, z.B. in leichtgewichtigeren Testwerkzeugen, aber auch für passives Testen wie Observability.

[1] Durch resilitente Systeme wie Kubernetes verschiebt sich der Fokus von der reinen Fehlerfindung hin zu Recoverability des Systems (MTTR wird wichtiger als MTBF).

Literatur

[1] N. Alshuqayran, N. Ali, R. Evans. *A Systematic Mapping Study in Microservice Architecture.* IEEE. 26.12.2016.

[2] A. S. Bueno, A. Gumbrecht, J. Porter. *Testing Java Microservices.* Manning. 2018.

[3] A. R. Cavalli, T. Higashino, M. Núñez. *A survey on formal active and passive testing with applications to the cloud.* Annals of Telecommunications, 70 (3-4). Springer Verlag. 2015.

[4] T. Clemson. *Testing Strategies in a Microservice Architectur.* 14.11.2014. `https://martinfowler.com/articles/microservice-testing/`

[5] A. Fachat. *Challenges and benefits of the microservice architectural style, Part 1.* 30.1.2019. `https://developer.ibm.com/articles/challenges-and-benefits-of-the-microservice-architectural-style-part-1/`

[6] M. Fowler. *Test Double; 17. Januar 2006.* `https://martinfowler.com/bliki/TestDouble.html`

[7] M. Fowler. *Mocks Aren't Stubs; 2. Januar 2007.* `https://martinfowler.com/articles/mocksArentStubs.html`

[8] M. Fowler. *BlueGreenDeployment; 1. März 2010.* `https://martinfowler.com/bliki/BlueGreenDeployment.html`

[9] S. Singh Gill, R. Buyya. *Failure Management for Reliable Cloud Computing: A Taxonomy, Model and Future Directions.* IEEE. 9.10.2018.

[10] H. Gnoyke. *Tests erst in Produktion? – Was wir von Tests bei Microservices lernen können.* Informatik Aktuell. 26.4.2016.

[11] K. Hoffman. *Beyond the Twelve-Factor App.* O'Reilly. 26.4.2016.

[12] honeycomb.io. *Observability For Developers.* 2019. `https://www.honeycomb.io/resources/guide-observability-for-developers/`

[13] ISTQB. *ISTQB Worldwide Software Testing Practices Report 2017-18.* 2017.

[14] S. Newman. *Building Microservices: Designing Fine-Grained Systems.* O'Reilly. 20.2.2015.

[15] A. Noonanon. *Goodbye Microservices: From 100s of problem children to 1 superstar.* 10.7.2018. `https://segment.com/blog/goodbye-microservices/`

[16] C. Richardson. *Microservice Patterns: With examples in Java.* Manning. 19.11.2018

[17] S. Marker. *Test Strategy for Microservices.* 8.5.2018. `https://www.gocd.org/2018/05/08/continuous-delivery-microservices-test-strategy/`

[18] A. Wiggins. *The twelve-factor app.* 2011. `https://12factor.net/`