02-2015 | Sommer | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell

D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

JAVA IST SUPER STARK

Programmierung
JavaScript für Java-Entwickler

Cloud Computing
Software-Architekturen in wolkigen Zeiten

Applikationsserver
JBoss vs. WebLogic Server

JavaServer Faces
Interview mit Spec Lead Ed Burns

4 191978 304903 02

iJUG Verbund

In Xtend geschriebener Code ist meist kompakter und eleganter als sein Java-Äquivalent

Der Autor ist ein sehr großer Fan von Java, aber auch von C#. Er stellt einige interessante Seiten von C# vor

5 Das Java-Tagebuch
Andreas Badelt

8 Software-Architekturen in wolkigen Zeiten
Agim Emruli

12 „Ich glaube, dass die Expression Language der heimliche Held der gesamten Web-Ebene von Java EE ist …"
Interview mit Ed Burns

14 Einstieg in die Liferay-Portal-Entwicklung unter Verwendung von JSF
Frank Schlinkheider & Wilhelm Dück

19 JavaScript für Java-Entwickler
Niko Köbler

21 Besser Java programmieren mit Xtend
Moritz Eysholdt

27 Ich liebe Java und ich liebe C#
Rolf Borst

30 Gestensteuerung und die nächste Welle der 3D-Kameras
Martin Förtsch & Thomas Endres

35 Microservices und die Jagd nach mehr Konversion – das Heilmittel für erkrankte IT-Architekturen?
Bernd Zuther, codecentric AG

41 Alles klar? Von wegen! Von Glücksrädern, Bibliothekaren und schwierigen Fragen
Dr. Karl Kollischan

44 Greenfoot: Einstieg in die objektorientierte Programmierung
Dennis Nolte

47 jOOQ – ein alternativer Weg, mit Java und SQL zu arbeiten
Lukas Eder

53 JBoss vs. WebLogic Server – ein Duell auf Augenhöhe?
Manfred Huber

58 PDF-Dokumente automatisiert testen
Carsten Siedentop

62 Unbekannte Kostbarkeiten des SDK Heute: Bestimmung des Aufrufers
Bernd Müller, Ostfalia

64 Einstieg in Eclipse
Gelesen von Daniel Grycman

64 Java – Der Grundkurs
Gelesen von Oliver B. Fischer

65 Android-Apps entwickeln
Gelesen von Ulrich Cech

Die Korrektheit erzeugter PDF-Dokumente überprüfen – nicht manuell, sondern automatisiert

Microservices und die Jagd nach mehr Konversion – das Heilmittel für erkrankte IT-Architekturen?

Bernd Zuther, codecentric AG

Die Software-Entwicklung begibt sich derzeit auf eine neue Evolutionsstufe: Fachabteilungen haben gelernt, mithilfe agiler Methoden Änderungen in kurzen Zyklen produktiv umzusetzen. Damit haben sie die Möglichkeit, Änderungen an einem Produkt mit echtem Benutzer-Feedback vermessen zu können und durch empirische Untersuchungen des Benutzerverhaltens fundierte Entscheidungen über neue Software-Funktionalitäten zu treffen. Dazu müssen aber die meisten IT-Architekturen erst einmal so auf Vordermann gebracht werden, dass einzelne Teile einer Software flexibel ausgetauscht und unabhängig voneinander ausgeliefert werden können.

Im Marketing beschreibt die Konversionsrate eine Kennzahl, die den prozentualen Anteil zwischen den Besuchen einer Webseite und dem Erreichen eines bestimmten Ziels darstellt. *Abbildung 1* zeigt die Formel zur Berechnung der Konversionsrate. In einem Online-Shop kann das beispielsweise das Absenden einer Bestellung oder der Klick auf einen bestimmten Button sein. Mit dem Ziel, dass ein Benutzer eine Bestellung abschließt, beschreibt die Konversionsrate den prozentualen Anteil der Personen, die von einem Kauf-Interessenten zu einem zahlenden Kunden umgewandelt werden.

Die Konversionsrate stellt eine sehr gute Kennzahl zum Bewerten von neuen Software-Funktionalitäten dar, denn mit ihrer Hilfe kann man die potenzielle Umsatzsteigerung hochrechnen, die durch eine Änderung zu erwarten ist. Die Konversionsrate sollte man unter anderem heranziehen, wenn Software-Funktionalität „A" von Software-Funktionalität „B" abgelöst werden soll.

Nach der Entwicklung von Software-Funktionalität „B" werden die Funktionalitäten „A" und „B" an zwei unterschiedliche Gruppen ausgespielt und das Verhalten der Benutzer in diesen Gruppen gemessen. Durch dieses Vorgehen hat man die Möglichkeit, frühzeitig zu reagieren, wenn die Benutzer Software-Funktionalität „B" nicht annehmen und die Konversionsrate durch Software-Funktionalität „B" sinkt. In diesem Fall kann man immer noch bei Software-Funktionalität „A" bleiben.

Dieses Verfahren wird „A/B-Testing" genannt. Ein einfaches Beispiel dafür ist das Austauschen der Farbe eines „Call to action"-Buttons in einem Bestellprozess (etwa der „Jetzt kostenpflichtig kaufen"-Button zum Abschließen eines Bestellvorgangs).

Feature Toogle

Für einfache A/B-Tests, wie für die Farbänderung eines „Call to action"-Buttons, kann mit dem sogenannten „Feature Toggle Pattern" *[1, 2]* gearbeitet werden. Dies ist ein recht einfaches „if then else"-Statement *(siehe Listing 1)*.

Mit einem Feature Toggle lassen sich unter anderem Bedien- oder Darstellungselemente ein- beziehungsweise ausblenden, Styles ändern oder sogar Backend-Systeme zur Laufzeit umschalten. An seine Grenzen stößt dieses Verfahren allerdings, wenn man in einem Online-Shop beispielsweise einen kompletten Bestellvorgang für einen A/B Test austauschen möchte. Dies liegt sehr häufig daran, dass die Software-Architekturen nicht dafür ausgelegt sind, konsequent A/B-Tests zu unterstützen, da in einer monolithischen Software-Architektur funktionale Kontexte häufig nicht ausreichend voneinander abgegrenzt sind.

Monolithische Software-Architektur

Ein Ansatz, den man sehr häufig in der Praxis findet, ist die sogenannte „monolithische Software-Architektur". In einer monolithischen Software sind die Benutzer-Schnittstelle sowie die Datenzugriffs- und Logik-Schicht in einem einzelnen Artefakt vereint. Die monolithische Software-Architektur bringt allerdings eine Reihe von Nachteilen mit sich:

- Die Anwendung ist häufig nicht besonders gut wart- und erweiterbar
- Bei Modifikationen können nicht vorhersehbare Neben-Effekte auftreten
- Änderungen müssen meistens mehreren Teams bekanntgegeben werden, da es häufig Modul-Verantwortlichkeiten gibt

```
if (benutzer.is_in_testgroup_a)
  change_button_to_green
else
  change_button_to_pink
end
```

Listing 1

$$\text{Konversionsrate} = \frac{\text{Anzahl der Zielerreichungen}}{\text{Besuche}}$$

Abbildung 1: Formel zur Berechnung der Konversionsrate

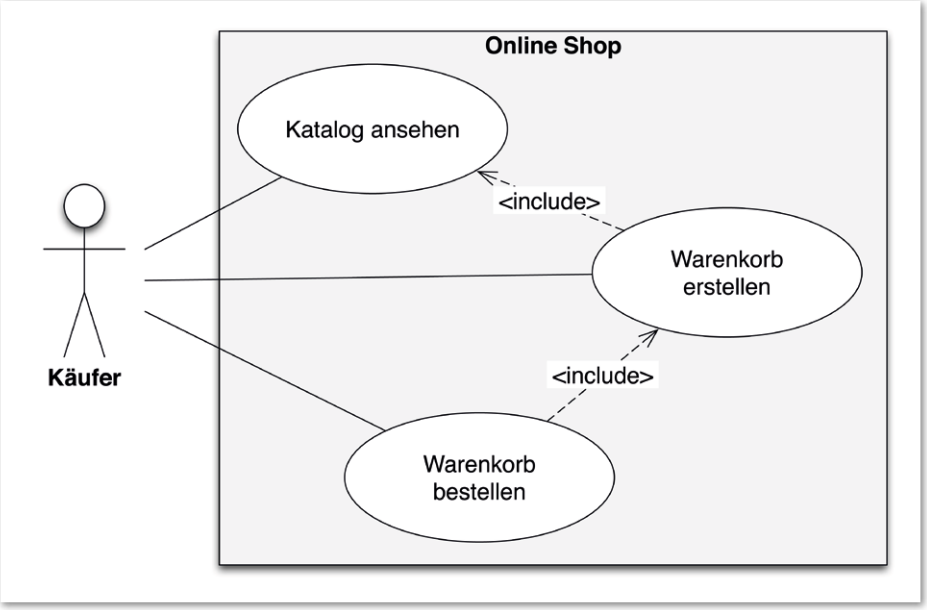

Abbildung 2: Anwendungsfalldiagramm eines Online Shop

und bei größeren Änderungen mehrere Module und damit Teams betroffen sind
- Teile des Software-Systems sind nur schlecht wiederverwendbar
- Eine Lastenverteilung von Teil-Komponenten ist nur schwer möglich
- Die Build-Laufzeiten können mehrere Stunden betragen

Natürlich besitzt diese Architektur-Form auch einige Vorteile, ansonsten würde man sie nicht so häufig vorfinden. Monolithische Anwendungen sind einfach zu übertragen und zu betreiben, da man sich um wenige Abhängigkeiten kümmern muss. Außerdem sind monolithische Anwendungen relativ einfach zu refaktorisieren, da sich der gesamte Quellcode meistens in einem Repository befindet. Auch in puncto Performance hat diese Architektur ihre Vorteile, da in der Regel wenige Remote Calls aufgerufen werden.

Man kann allerdings feststellen, dass monolithische Applikationen nur wenig wandelbar sind, was das konsequente Durchführen von A/B-Tests erschwert. Es ist aber wünschenswert, wenn man Anwendungen wie Lego-Bausteine ständig reorganisieren und mit neuen Bausteinen bestücken könnte. Eine Lösung für dieses Problem verspricht die sogenannte „Microservice-Architektur" [3].

Microservice-Architektur

Es gibt verschiedene Versuche, den Begriff „Microservice" zu erklären. Einen ausführlichen Definitionsversuch findet man in Ralf Westphals Blog [4]. Für diesen Artikel wird eine einfache Erklärung verwendet, die auf den Gedanken von Timmo Freudl-Gierke [5] basiert. Für das weitere Verständnis ist ein Microservice ein autonomer Lieferant einer bestimmten Funktionalität, der alle Daten verwaltet, die der Service zum Liefern dieser Funktionalität benötigt. Daraus ergibt sich, dass jeder Microservice eine bestimmte Funktionalität der Geschäftslogik implementiert. Es bleibt aber offen, mit welchen Mitteln (Programmiersprache, Datenbank, Protokolle) diese Funktionalität implementiert wird. Es ist außerdem nicht ausgeschlossen, dass ein Microservice auch eine Benutzerschnittstelle haben kann.

Da jeder Microservice über eigene Daten verfügt, muss er ein eigenes, explizites Repository besitzen, das ihn mit den benötigen Daten beliefert. Ein solches Repository kann eine relationale Datenbank, aber auch jede beliebige andere Datenquelle sein.

Wenn man Funktionalität und Daten als eine zusammengehörende Einheit betrachtet, ergeben sich interessante Implikationen für die Zusammenarbeit von verschiedenen Teams. Bei Optimierungen an der Datenbank oder für Schema-Änderungen ist nur noch das Team verantwortlich, dem die Funktionalität gehört, und Änderungen müssen nicht mit anderen Teams abgesprochen werden, solange sich dabei die Schnittstelle nach außen nicht ändert. Eine solche Schnittstelle sollte möglichst plattformneutral sein, denn dann können einzelne Software-Komponenten beliebig für einen A/B Test ausgetauscht werden. Abbildung 2 zeigt, welche minimale Funktionalität man von einem Online-Shop erwartet. In dem Diagramm sind die folgenden Anwendungsfälle zu sehen:

- Einem Benutzer muss ein Katalog von Produkten angezeigt werden
- Aus den angezeigten Produkten kann sich ein Benutzer einen Warenkorb zusammenstellen
- Diesen Warenkorb kann ein Benutzer bestellen

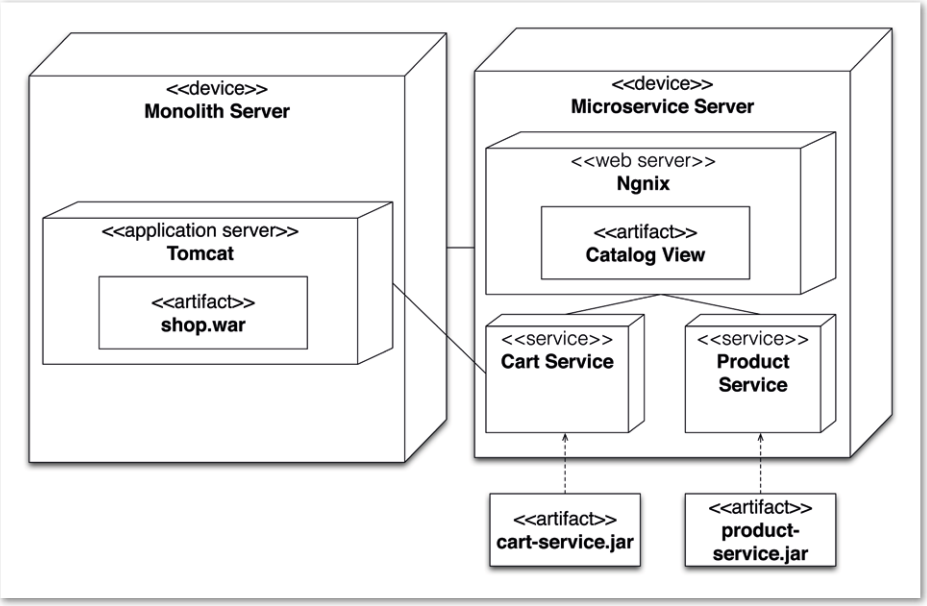

Abbildung 3: Verteilungsdiagramm einer Kombination aus monolithischer und Microservice-Architektur

Das Anwendungsfall-Diagramm zeigt also grundlegende Funktionalitäten. Im „Domain Driven Design" werden Funktionalitäten, die eine Fachlichkeit darstellen, als eigenständige zustandslose Services modelliert. Deshalb liegt es nahe, das Anwendungsfall-Diagramm als grobe Orientierungshilfe zu benutzen, um festzustellen, wie viele Microservices dieser Online Shop benötigen wird. Daraus ergeben sich die folgenden drei Microservices:

- Produkt-Service
- Warenkorb-Service
- Bestell-Service

Die Frage ist nun, ob diese Aufteilung schon ausreicht, wenn man das Ziel berücksichtigt, die Konversionsrate von verschiedenen Katalog-Ansichten oder Bestell-Prozessen vermessen und verbessern zu können. Die Antwort ist ein klares Nein. Um den Katalog oder den Bestellprozess optimieren zu können, sind zwei neue Microservices erforderlich. Diese stellen die Benutzeroberfläche bereit und man kann sie gegebenenfalls für einen A/B-Test durch andere Microservices austauschen. Daraus ergeben sich nun fünf Microservices, die man technisch mit Hilfe von drei REST-Services und zwei Single-Page-Web-Anwendung (SPA) zu einem Gesamtsystem zusammenfügen kann.

Bei diesen Kontextgrenzen kann man die fünf Dienste von fünf Teams parallel entwickeln lassen – vorausgesetzt, die Teams einigen sich auf gemeinsame Schnittstellen. Sie können dann autonom ihre Technologien festlegen. Das Produkt-Service-Team kann zum Beispiel MongoDB oder Elasticsearch als Datenbank für die Persistierung der Produkte benutzen und das Warenkorb-Service-Team alternativ eine Redis-Datenbank. Der Microservice, der die Katalog-Ansicht implementiert, steht in einer Kunden-Lieferanten-Beziehung zum Produkt-Service. Dafür kann dieser eine REST-Schnittstelle zur Verfügung stellen.

Migration eines Monolithen

Bisher kam der technische Schnitt eines Online-Shops zur Sprache, der sich in einem „Grüne Wiese"-Projekt anbietet. Wie soll man aber vorgehen, wenn man schon einen laufenden, monolithischen Shop betreibt, der im mittleren fünfstelligen Bereich Umsatz pro Stunde generiert? Ein Big-Bang-Umstieg wäre viel zu riskant, da die Gefahr groß ist, dass das neue System Fehler enthält oder vom Kunden nicht angenommen wird. Deshalb liegt die Frage nahe, ob man mithilfe von Microservices auch eine monolithische Anwendung im laufenden Betrieb migrieren kann.

Angenommen, man betreibt eine monolithische Anwendung, die die drei Anwendungsfälle aus dem vorangegangen Abschnitt implementiert. Technisch gesehen, handelt es sich dabei um eine Java-EE-Anwendung, die in Form einer WAR-Datei in den Servlet-Container Tomcat ausgeliefert wird.

Der Produktmanager möchte eine neue Katalog-Ansicht ausprobieren. Der neue Produktkatalog verfügt über sehr viele dynamische Bedienelement und die Frontend-Entwickler sind sich schnell einig, den neuen Katalog lieber mit einem neuen, hippen JavaScript-Framework zu programmieren, als mit ihrem heißgeliebten Web-Frontend-Framework Wicket, das in der monolitischen Anwendung zum Einsatz kommt. Natürlich möchte der Produktmanager den neuen Katalog erst mit 20 Prozent seiner Kunden testen, um zu validieren, ob der neue Katalog die Konversion wirklich erhöht, fachlich und technisch können also beide Systeme erst einmal nebeneinander existieren *(siehe Abbildung 3)*.

Aufgrund der vorangegangen Definition ist ein Microservice ein Lieferant einer bestimmten Funktionalität. Ein Monolith verfügt über verschiedene Funktionalitäten und für eine bestimmte Funktionalität kann dieser auch das führende System zur Bereitstellung der Funktionalität sein. Für unser Beispiel heißt das, dass die Java-EE-Anwendung weiter für den eigentlichen Bestellprozess verantwortlich bleibt.

JavaScript-Single-Page-Anwendung

Abbildung 3 zeigt ein Verteilungsdiagramm, das eine mögliche Kombination aus monolithischer und Microservice-Architektur enthält. In diesem Integrationsmodell wird eine JavaScript-Single-Page-Anwendung (SPA), die die Katalog-Funktionalität bereitstellt, vom Web-Server Ngnix ausgeliefert. Die SPA greift auf zwei REST-Schnittstellen zu, die in eigenen Prozessen auf dem Server laufen und über ein Fat Jar gestartet werden. Dies ist eine ausführbare Jar-Datei, in

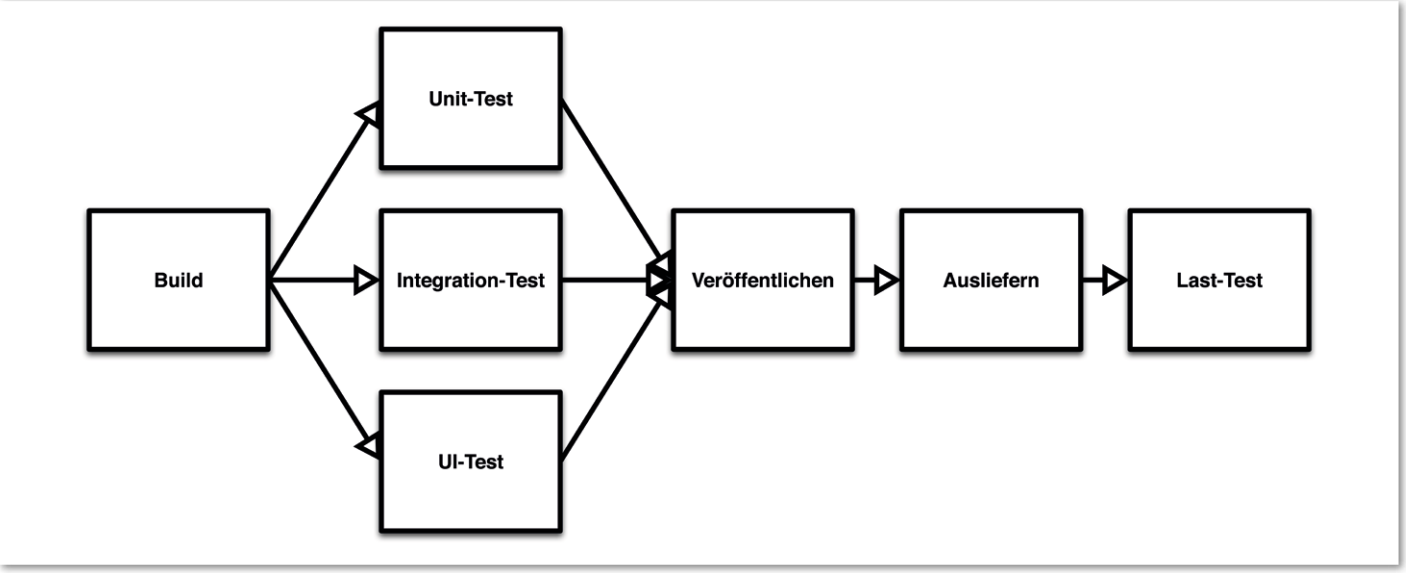

Abbildung 4: Deployment-Pipeline eines Monolithen

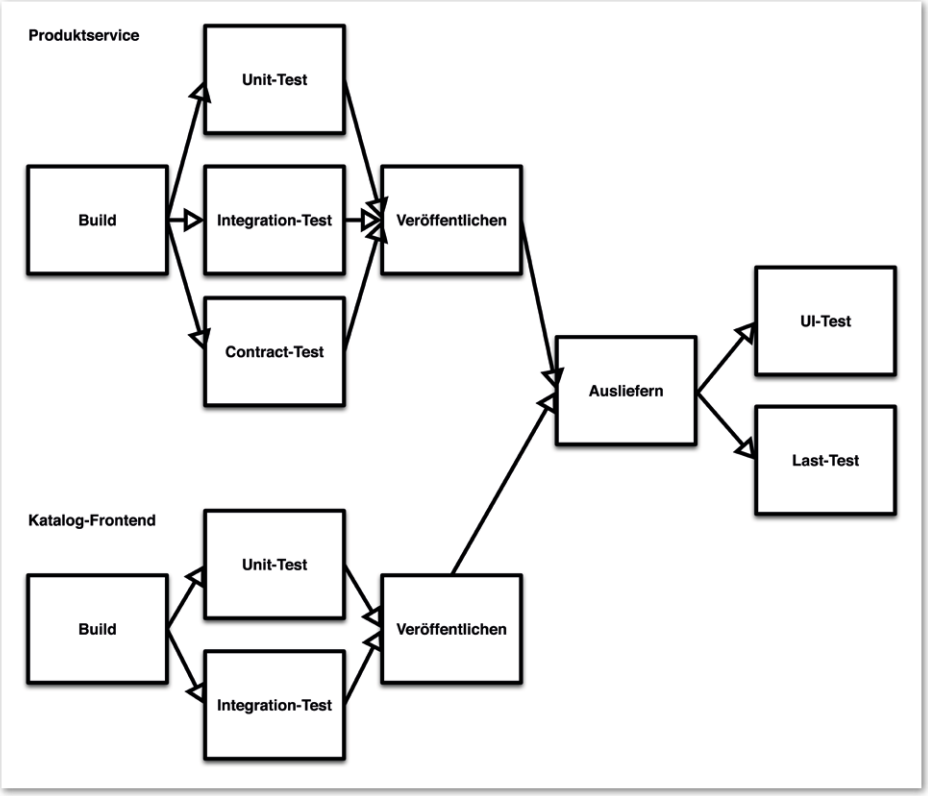

Abbildung 5: Multi-Deployment-Pipeline einer Microservice-Architektur

der alle Bibliotheken enthalten sind, die von einer Anwendung benötigt werden.

Ngnix sorgt neben der Auslieferung der SPA auch dafür, dass es beim Zugriff auf den Produkt- und Warenkorb-Service (Cart) nicht zu Problemen mit der „Same Origin"-Policy kommt, die in den meisten Browsern implementiert ist.

Um den alten, monolithischen Shop mit dem neuen Katalog zu verknüpfen, muss ein Benutzer die Möglichkeit haben, in den alten Shop einzuspringen und auf den Warenkorb-Service des neuen Katalogs zuzugreifen, um im Altsystem den Bestellprozess abzuschließen. Technisch kann das über einen Link *(etwa „https://checkout.microservice.io/shop/checkout?cart=b579b0b4-75c8-49c7-92e9-f8265c6e5d1c")* erreicht werden, man verlinkt also aus dem neuen Katalog in den alten Shop. Dieser reagiert auf den Link und sorgt dafür, dass über eine andere Strategie auf den Warenkorb-Service zugriffen wird.

So lässt sich Stück für Stück eine Anwendung in eine Microservice-Architektur migrieren. Außerdem hält man das Risiko bei der Migration sehr gering, denn wenn die neue Funktionalität nicht richtig funktioniert oder die Benutzer die neue Funktionalität nicht annehmen, kann man mit geringem Aufwand wieder die alte Implementierung verwenden. In *Abbildung 3* ist aber auch zu sehen, dass sich die Anzahl an Artefakten erhöht. Aus einem Artefakt (monolithische Architektur) sind nun vier Artefakte geworden. Continuous Delivery und Continuous Integration werden deshalb noch wichtiger. Denn die Flexibilität beim Zusammenstellen von einzelnen Fachlichkeiten zu einem Gesamtsystem erhält man nicht umsonst.

Multi-Deployment Pipeline

Wie beschrieben, erhöht sich mit einer Microservice-Architektur die Anzahl der Artefakte,

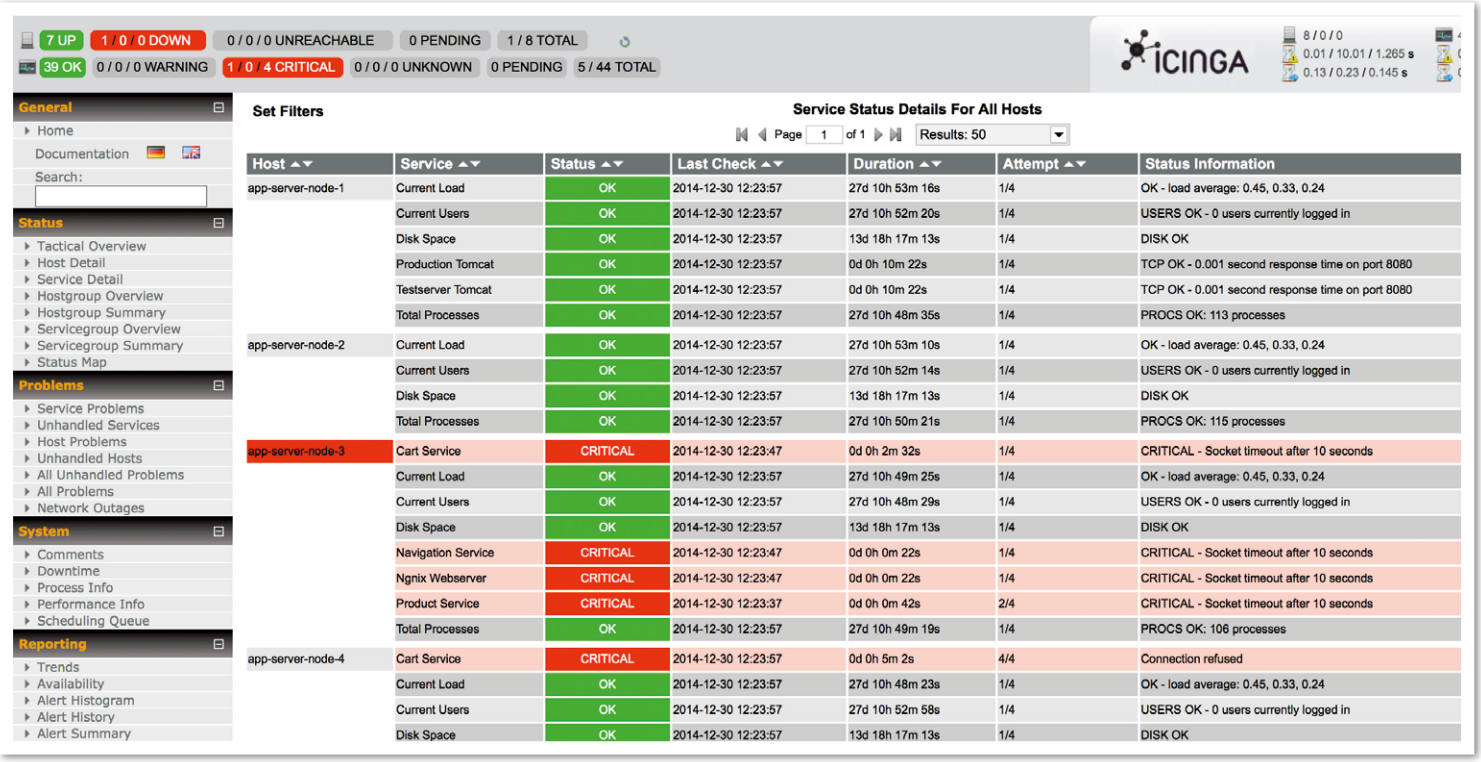

Abbildung 6: Icinga zur Überwachung einer Microservice-Architektur

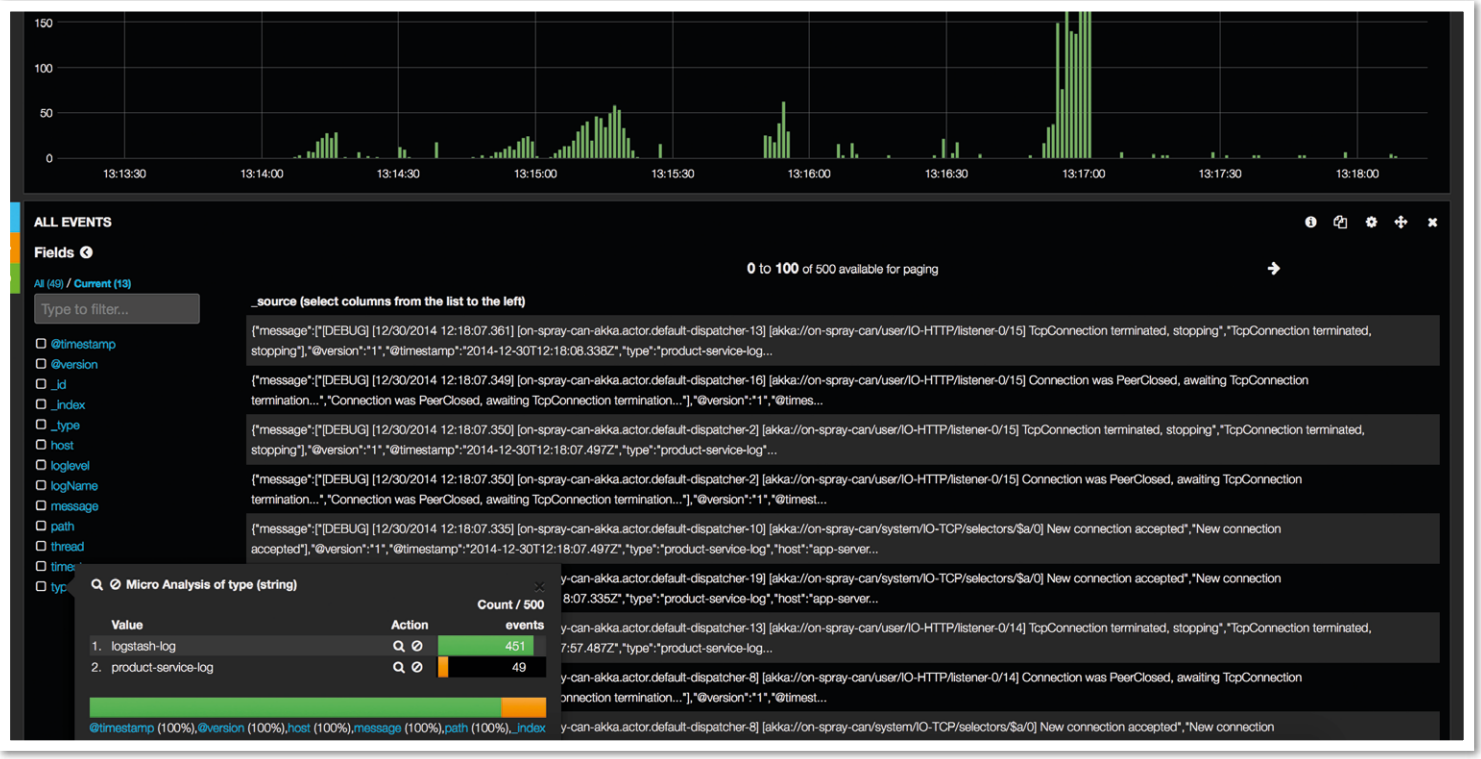

Abbildung 7: ELK-Stack für die Analyse verschiedener Log-Dateien

die für das Gesamtsystem erforderlich sind. *Abbildung 4* zeigt eine Deployment-Pipeline eines Monolithen, die mit dem Build der Software beginnt. Nachdem die Software gebaut ist, werden eine Reihe von Tests parallel ausgeführt und danach wird ein Artefakt veröffentlicht.

Dieses Artefakt kann, wie gesagt, eine WAR-Datei sein. Die Veröffentlichung erfolgt über einen Artefakt-Repository oder einen File-Server, von dem aus das Artefakt für die Auslieferung benutzt wird. Nach der Auslieferung können noch automatisierte Akzeptanz- und Lasttests laufen, um Integrationsprobleme festzustellen oder Performance-Regressionsfehler zu finden.

Abbildung 5 zeigt, wie sich die Deployment-Pipeline in einer Microservice-Architektur ändert. Exemplarisch sind zwei Komponenten dargestellt, die wir auch im Verteilungsdiagramm aus dem vorangegangen Abschnitt finden können: die Katalog-Darstellung („Catalog View") und der Produkt-Service. Sie sorgen dafür, dass ein Benutzer einen Katalog angezeigt bekommt. In *Abbildung 5* kann man sehen, dass diese Komponenten Zweige innerhalb der Deployment Pipeline darstellen, die nach dem Veröffentlichen der Artefakte wieder zusammenlaufen.

Nach dem Veröffentlichen werden die Artefakte zusammen ausgeliefert und das System kann integrativ getestet und dann durch anschließende UI- und Last-Tests überprüft werden. Auf Service-Ebene sollte man noch sogenannte „Contract-Tests" durchführen, die das Team schreiben sollte, das einen bestimmten Service benutzt. Diese Tests stellen sicher, dass die Schnittstellen eines Microservice nicht-abwärtskompatibel geändert werden.

Eine weitere Frage ist, welche Artefaktform man für eine möglichst einfache Auslieferung auswählt. Hier gibt es leider keine generelle Antwort. Container-Technologien wie Docker werden immer populärer und sind sicherlich ein weiterer Schritt nach vorn, um eine durchgängige Lieferkette vom Entwicklungssystem bis in Produktion gleich zu gestalten. Doch Container sind noch relativ jung und sicherlich nicht für alle Anwendungsfälle geeignet.

Um Software zu installieren, hat jedes Betriebssystem seine Eigenheiten: Windows hat Installationsroutinen, Linux eine Paketverwaltung und MacOS das sogenannte „Application Bundle". Auch diese Artefaktformen haben ihre Vorteile und fördern die Übertragbarkeit von Software *[6]*. Alle drei haben gemeinsam, dass auch Abhängigkeiten (Datenbanken, Webserver) ausgedrückt werden können, die benötigt werden, damit die Software fehlerfrei läuft.

Viele Wege führen damit zu einer durchgängigen Deployment-Pipeline und die einzige Bewertungsgrundlage sollte deshalb sein, ob die Software nach einem Commit automatisiert auf mindestens einer Umgebung ohne manuelle Schritte lauffähig ist. Falls ja, verwendet man bereits die richtige Technologie. Durch eine Multi-Deployment-Pipeline wird nun das gesamte System ausgeliefert. Doch wie überwacht man ein System, das aus mehreren Anwendungen besteht?

Monitoring

Zum Überwachen von komplexeren Microservice-Architekturen, kann man die freie Software Nagios benutzen *[8]*. Sie basiert auf einer objektorientierten Konfiguration. Bestandteil dieser Konfiguration sind Hosts, die über eine IP-Adresse definiert werden. Einem Host können die zu überwachenden Services zugeordnet werden. Die Überwachung erfolgt über sogenannte „Kommandos", die zur Alarmierung ausgeführt werden. Letztlich kann man noch definieren, wer bei bestimmten Unregelmäßigkeiten eine Nachricht erhalten soll. Mit Icinga *[9]* ist ein Nagios-Fork entstanden, der über eine modernere Web-Oberfläche verfügt und einen dynamischeren Entwicklungsprozess als Nagios besitzt *(siehe Abbildung 6)*.

Nagios oder Icinga bieten die Möglichkeit, die Verfügbarkeit der Services zu überwachen und SLA-Reports zu erstellen. Wenn man mit diesen Werkzeugen feststellt, dass

ein Service zu einer bestimmten Uhrzeit nicht verfügbar war, muss natürlich eine Analyse folgen, was die Ursache für das Problem ist. Dazu durchsucht man in der Regel die Logdateien einer bestimmten Applikation.

Erinnern wir uns an *Abbildung 3*, dann ergibt sich schon aus der Anzahl an Artefakten, dass das Gesamtsystem über mindestens vier Log-Dateien verfügt. Dazu kommen noch die Logdateien des Webservers und der Datenbanken. Damit müssen selbst in diesem einfachen Beispiel bis zu sieben Log-Dateien für die Ursachen-Analyse untersucht werden, die noch dazu auf mehreren Servern verteilt sind.

Um Korrelationen in unterschiedlichen Log-Dateien zu finden und den Zugriff auf die unterschiedlichen Logdateien zu zentralisieren, gibt es den sogenannten „ELK-Stack" *[10]*. Er besteht aus den folgenden Bestandteilen *(siehe Abbildung 7)*:

- *Elasticsearch*
  Indexiert und macht die Informationen in der Log-Datei durchsuchbar
- *Logstash*
  Sammelt die Log-Dateien ein, transformiert und normalisiert dieses und extrahiert bestimmte Daten daraus
- *Kibana*
  Bietet eine grafische Oberfläche für die Log-Datei-Analyse

Der ELK-Stack bietet somit die Möglichkeit, die Log-Dateien zu zentralisieren und verschiedene Log-Formate normalisiert durchsuchbar zu machen.

Fazit

Auch wenn Microservices derzeit sehr als Hype gelten, sind sie kein Allheilmittel. Ein genaues Verständnis der Domäne ist weiterhin erforderlich *[11]*. Nur dann kann man die Kontextgrenzen zwischen Services genau definieren und damit fachliche Konzepte technisch trennen. Außerdem ist es wichtig, dass man das Problem genau versteht, das der Fachbereich lösen möchte. Nur dann kann man eine technische Lösung definieren, die die Anforderungen genau erfüllt und technisch möglichst einfach ist.

Wenn man für die Kommunikation zwischen den unterschiedlichen Microservices einen technologieneutralen Vertrag wählt, ist die konkrete Technologie irrelevant, die zur Implementierung des Microservice verwendet wird. Wichtig ist, dass sich sowohl die Auslieferung als auch das Testen der produzierten Artefakte automatisieren lässt.

Die Berücksichtigung dieser Punkte bietet die Möglichkeit, mithilfe von A/B-Tests ein Software-Produkt ständig zu optimieren, die Konversionsrate zu verbessern und den Umsatz zu steigern. Mit dem richtigen Software-Schnitt besteht die Möglichkeit, das Software-System ständig zu verändern und langfristig den Erfolg eines Produkts durch Innovation zu sichern.

Mit dem Schnitt einer Software in verschiedene funktionelle Teile bekommen die Themen „Continuous Delivery" und „Continuous Integration" eine noch größere Bedeutung. Denn nur dann, wenn die Auslieferung des Software-Systems möglichst einfach ist, hat man die Zeit, in einem solchen verteilten System das Benutzerverhalten zu analysieren und die Software ständig zu optimieren. Bei einer solchen Analyse hilft unter anderem ein zentralisiertes Logging, wie es der ELK-Stack oder kommerzielle Lösungen bieten.

Eine Referenz-Implementierung des Online-Shops aus diesem Artikel ist im GitHub-Repository des Autors *[12]*. In dem Projekt „Appstash" befindet sich eine Vagrant-Datei, die ein komplettes Entwicklungscluster erstellt, das über eine Lecacy-Online-Shop-Anwendung, eine Microservice-Implementierung eines Katalogs, einen Jenkins, der eine komplette Multi-Deployment-Pipeline mithilfe der Debian-Paketverwaltung zeigt, und über das beschriebene Nagios-Monitoring sowie den ELK-Stack verfügt.

Quellen

[1] http://www.heise.de/developer/artikel/Continuous-Delivery-mit-dem-FeatureToggle-Pattern-1825477.html
[2] http://martinfowler.com/bliki/FeatureToggle.html
[3] http://martinfowler.com/articles/microservices.html
[4] http://blog.ralfw.de/2014/08/warnung-vor-dem-microservice-versuch.html
[5] jaxenter.de/artikel/nie-wieder-monolithen-176652
[6] http://bernd-zuther.de/deployment-ganz-einfach
[7] http://12factor.net
[8] http://www.nagios.org
[9] https://www.icinga.org
[10] https://bernd-zuther.de/splunk-marke-eigenbau-mit-elasticsearch-logstash-und-kibana
[11] http://www.effectivetrainings.de/blog/2014/12/18/be-good-was-ein-guter-entwickler-konnen-muss
[12] https://github.com/zutherb/AppStash

Bernd Zuther
bernd.zuther@codecentric.de

Bernd Zuther ist Diplom-Informatiker und seit dem Jahr 2008 als Software-Entwickler tätig. Derzeit arbeitet er bei der codecentric AG als Berater im Bereich „Agile Software Factory", wo er sich vor allem mit Big Data, Continuous Delivery und agilen Software-Architekturen beschäftigt. Bernd Zuther verfügt über viel praktische Erfahrung im Bereich „E-Commerce" und beschäftigt sich seit mehreren Jahren mit der Entwicklung von hochverfügbaren, individuellen Online-Shop-Systemen.

http://ja.ijug.eu/15/2/10

Maven-Repository bald verfügbar

Oracle hatte auf der OpenWorld in San Francisco die baldige Veröffentlichung eines Maven-Repository angekündigt. Dafür plädierte die Entwickler-Community seit geraumer Zeit. Inzwischen sei der Konzern in der Endphase des Projekts.

Das Repository soll in Zukunft unter *https://maven.oracle.com* nach einer SSO-Anmeldung zu finden sein. Auf der Plattform finden Entwickler nur Artefakte, die von Oracle bereitgestellt und für Kompilierung, Paketierung, Tests und Verwendung von Client-APIs verwendet werden, zum Beispiel JAR-Dateien von WLS, ADF, SOA, OSB, BPM und WCP.

Da Patches einen Support-Vertrag erfordern, sollen nur Release-Versionen (also 12.1.2, 12.1.3) zur Verfügung stehen.