
Advertorial

**Wie clevere Testautomatisierung die Qualität steigert**

# Automatisches Testen von Microservices mittels Proxy-based Testing

*von Carsten Negrini*

Die Einführung einer Microservice-Architektur folgt aus dem Ansatz der agilen Softwareentwicklung, bietet aber auch noch andere Vorteile wie leichte Umsetzbarkeit in Cloud-Umgebungen, lose Koppelung der Services und damit geringe Abhängigkeiten. Der Testansatz muss der Architektur folgen, so dass sich Änderungen am Testaufbau ergeben. Hier hilft die proxy-basierte Testautomatisierung, um auf einer hohen Abstraktionsebene eine automatisierte Wiederholbarkeit von Tests bei geringem Aufwand zu erreichen.

## Was ist eine Microservice-Architektur?

Monolithische Software-Systeme haben den Nachteil, im Laufe der Zeit immer größer, schwerer wartbar und nur als Ganzes änderbar und ausrollbar zu sein. Dazu kommen sehr hohe Kosten, will man große Systeme ausfallsicher gestalten.

Die mit serviceorientierten Architekturen begonnene Umstellung von monolithischen Software-Systemen hin zu lose gekoppelten Services wird mit einer konsequenten Microservice-Architektur weiter vorangetrieben. Dabei wird die Aufteilung eines Systems in einzelne, übersichtliche und kleine Komponenten, eben Microservices, betrieben, die unabhängig voneinander betreibbar und über Standardkommunikationswege (zumeist http(s)-Aufrufe, REST-Calls) aufrufbar sind.

Dabei entstehen einige Vorteile gegenüber monolithischen Systemen: Man erhält deutlich leichter wartbare Codeblöcke, die durch unterschiedliche Teams zu unterschiedlichen Zeitpunkten entwickelt, getestet und in den produktiven Betrieb genommen werden können. Die so aufgebaute Softwarearchitektur entspricht auch mehr dem agilen Softwareentwicklungsprozess, da die einzelnen Microservices wenig Abhängigkeiten untereinander haben. Die Microservices können mit geringem Aufwand in einer Container-Umgebung verteilt und betrieben werden. Damit ist eine Skalierung leicht möglich. Service Meshes wie istio oder Linkerd können die so entstehenden Servicelandschaften gut beherrschen.

## Testaufwände in Microservice-Architekturen verschieben sich

Mit dieser veränderten Softwarearchitektur einhergehend kommen auf die Tester auch neue Herausforderungen zu: Während sich auf den unteren Ebenen der Testpyramide keine Änderungen ergeben (Unittests bleiben gleich), wird schon der Aufbau von Komponententests deutlich einfacher, da die Außengrenzen eines Microservice unabhängig von der verwendeten Programmiersprache durch alle Testwerkzeuge ansprechbar sind, die die verwendeten Standardkommunikationswege (zumeist

http(s)-Aufrufe, REST und JSON, manchmal auch Messaging) unterstützen. Zudem ist durch die prinzipbedingte Unabhängigkeit der Microservices der isolierte Test eines einzelnen Microservice mit viel weniger Aufwand möglich, als beispielsweise ein Modultest eines Java-Packages in monolithischer Softwarearchitektur.

Andererseits verändert sich die Herangehensweise bei der Testautomatisierung auf Integrationstest- bzw. Solutiontest-Ebene doch deutlich. Während monolithische Systeme im Verhältnis zur Microservice-Architektur weniger Schnittstellen haben, die getestet werden müssen, hat per Definition jeder Microservice selbst im Integrationstest zu testende Schnittstellen.
Zusätzlich erhöht sich der Aufwand für den Integrations- beziehungsweise Solutiontest auch deshalb, weil durch die Entkoppelung der Microservices gegenüber früher üblichen Softwarearchitekturen viel öfter Regressionstests durchgeführt werden können. Die Häufigkeit der Testdurchführung bedingt eine nahezu vollständige Automatisierung. In einer Microservicearchitektur hat man demnach auf der Integrationstest-Ebene mehr Schnittstellen, die zudem auch noch deutlich häufiger getestet werden müssen. Damit steigt die Notwendigkeit für eine gute aufgesetzte Testautomatisierung.

Die Automatisierung verursacht initial zwar einen höheren Aufwand als ein manueller Test, dieser ist jedoch gut investiert, da im Nachgang sehr schnell mit wenig Aufwand ganze Service-Landschaften getestet werden können. Gerade wenn viele Teams agil entwickeln, kommen die unterschiedlichen Services zu völlig unterschiedlichen Zeitpunkten in den Test, so dass eine Automatisierung hier ihre vollen Vorteile ausspielen kann.

## Proxy-based Testing

In dem hier beschriebenen Szenario wird eine proxy-basierte Testautomatisierung vorgestellt. Der Grundgedanke ist, schon in einer sehr frühen Phase die Außenschnittstellen der vorhandenen Microservices anzusprechen, die Kommunikation mittels eines Proxys aufzuzeichnen und dann, eventuell nach Zeichenfolgenersetzungen, wiedergeben zu können. Im Laufe der Zeit werden immer mehr Microservices der gesamten Landschaft testfähig und können dann in die Prozessketten integriert werden, bis schließlich die Prozessketten als Ganzes sehr schnell regressionsgetestet werden können. Dabei wird in drei Schritten vorgegangen: Im ersten Schritt werden die Kommunikationsbeziehungen zwischen den Microservices untersucht (siehe **Abbildung 1**) und derart umkonfiguriert, dass zwischen jedem Kommunikationspartner ein Proxy etabliert wird.

Im zweiten Schritt werden die durch den Proxy geschriebenen Kommunikationsprotokolle um Automatismen angereichert, die bei der Wiederholung der Tests automatisch Testdaten einsetzen können (siehe **Abbildung 2**).

Abb. 1: Kommunikation zwischen Services

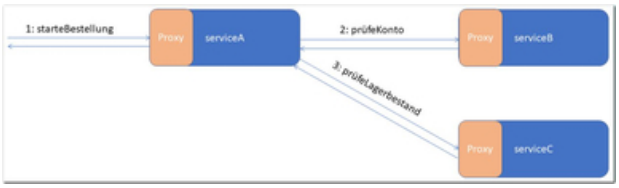

Abb. 2: Kommunikation unter Proxy-Verwendung

Im dritten Schritt wird der Proxy in den Modus Lastgenerator umgeschaltet. Dabei werden die in den Kommunikationsprotokollen aufgezeichneten Serviceaufrufe in Echtzeit mit Testdaten angereichert und die Antworten des angesprochenen Microservices mit den Soll-Antworten verglichen.

Im Detail wird eine Testumgebung mit zahlreichen Microservices aufgesetzt, wobei die Microservices sich nicht direkt ansprechen, sondern über einen Proxy. Dazu muss der verwendete Proxy die Kommunikationsbeziehungen aller Microservices untereinander kennen und diese in einer Weiterleitungstabelle verfügbar halten. Somit kann, abhängig von der aufgerufenen URL, ein Serviceaufruf durch den Proxy auf den tatsächlichen Service weitergeleitet werden.

Dabei zeichnet der Proxy den Datenverkehr der Serviceaufrufe auf. Die Aufrufe und Antworten der durch den Geschäftsprozess aufgerufenen Microservices werden in eine Protokoll-Datei geschrieben, inklusive der Metadaten wie Aufrufzeitpunkt, Antwortzeitpunkt, Inhalt der Nachrichten. Innerhalb dieser Datei kann man die gesamte Durchführung des kompletten Geschäftsprozesses für alle beteiligten Services nachverfolgen.

| Vor Variablenersetzung | Nach Variablenersetzung |
| --- | --- |
| {„bestellung“:<br>[„kundenId“: „{{testdaten.kdnr}}“,<br>„artikel“:<br>[{„artikelId“: „{{art1}}“, „anzahl“: „{{m1}}“},<br>{„artikelId“: „{{art2}}“, „anzahl“: „{{m2}}“}]<br>]<br>} | {„bestellung“:<br>[„kundenId“: „12345“,<br>„artikel“:<br>[{„artikelId“: „1733“, „anzahl“: „2“},<br>{„artikelId“: „927“, „anzahl“: „1“}]<br>]<br>} |

Abb. 3: Variablenersetzung

In der entstandenen Datei werden durch den Testautomatisierer die verwendeten Testdaten, welche bei jeder erneuten Testdurchführung geändert werden müssen, einmalig durch Variablenbezeichner ersetzt. Beispielsweise wird „vertragNr=12345“ ersetzt durch „vertragNr={dateiA.vertragNr}“ oder „vertragNr={dbA.tabelleA.vertragNr}“ (siehe **Abbildung 3**).

Nun wird der Proxy in den Modus Testplayer geschaltet. Damit werden die in der Protokoll-Datei enthaltenen Serviceaufrufe wiederholt, wobei vor dem Aufruf die Variablenbezeichnung durch Zeichenersetzungen mit den Testdaten ersetzt wird, die aus einer Datenbank oder einer Datei entnommen werden. Nach dem initialen Geschäftsprozessaufruf kann der Testplayer durch Vergleich der bei der Testdurchführung entstehenden neuen Protokoll-Datei mit der ursprünglichen Protokoll-Datei einen automatischen Soll-/Ist-Vergleich durchführen und damit sofort den Erfolg oder Misserfolg des Testlaufs bewerten.

Da ein so aufgebauter Testfall schnell und mit minimalem Aufwand reproduzierbar ist, kann nun spätestens bei jedem Deployment der Services mindestens einmal der Testfall durchgeführt werden. Im Idealfall kann sogar innerhalb der Sprints der Testfall durchgeführt werden, also können auf einer hohen Ebene der Testpyramide häufig komplette Serviceketten schon bei einzelnen Commits durchgetestet werden.
Der Aufwand für den Bearbeiter reduziert sich dabei auf die Kontrolle der im Wiedergabemodus des Proxys geschriebenen Logfiles, falls ein Fehler in der Testdurchführung aufgetreten ist.

In einer gemanagten Service-Mesh-Umgebung wie beispielsweise istio ist das Vorgehen ebenfalls gut umsetzbar. Hier wird der verwendete Proxy envoy entsprechend konfiguriert, um die Requests und Responses mitzuloggen. Der Automatisierungsaufwand und die Testfallwiedergabe entspricht der Lösung ohne Service-Mesh.

## Erfahrungen und Resultate

Bei den meisten der von uns begleiteten Projekte kam es in recht enger zeitlicher Abfolge zu Paradigmenwechseln „Wasserfall“ zu „Agile“ und dementsprechend auch zu einem Umbau der Architektur in Richtung „Microservices“. Ein agiler Softwareentwicklungsprozess bedingt, um seine Vorteile realisieren zu können, Änderungen in der Softwarearchitektur. Die damit einhergehenden Änderungen verursachen Veränderungen im Testvorgehen, um einen effizienten und effektiven Test zu ermöglichen.
Während die Vorteile einer Testautomatisierung schon in den bisher üblichen Softwareentwicklungsprozessen bei einigen oft zu testenden Geschäftsprozessen deutlich sichtbar waren, kommen diese Vorteile bei einer Microservice-Architektur noch stärker zum Vorschein.

In Softwareentwicklungsprozessen nach Wasserfall oder V-Modell beispielsweise werden die Tests auf einer der oberen Ebenen der Testpyramide typischerweise spät im Prozess durchgeführt, wenn alle Komponenten verfügbar und weitgehend fertig entwickelt sind. In großen bis sehr großen agilen Entwicklungsprojekten fertigen jedoch unterschiedliche Teams die Komponenten beziehungsweise Microservices eigenverantwortlich und agieren dabei auch unabhängig von den anderen Teams.

Damit gelangen die einzelnen Microservices zu unterschiedlichen Zeiten in den Integrations- beziehungsweise Solutiontest, wodurch eine häufige Wiederholung der Tests notwendig wird. So haben wir in einigen Projekten wichtige Geschäftsprozesse, die mehr als dreißig verschiedene Microservices von fünf verschiedenen agilen Teams beinhalteten, täglich in einem Ende-zu-Ende-Test überprüft.

Die zeitlich späte Durchführung des Integrationstests im agilen Prozess, also kurz vor der Fertigstellung des Gesamtsystems, ist

jedoch nicht sinnvoll, da im Falle notwendiger Bugfixes die entsprechenden Teams bereits in den nächsten Sprints und daher mit ihren Tätigkeiten in den Sprints bereits ausgelastet sind. Daher ist ein häufiger, früher Integrationstest mit hohem Automatisierungsgrad in diesen Projekten sehr zweckdienlich.

Wird eine Microservice-Architektur als Service-Mesh ausgerollt, erfüllt eine Control-Plane gleich mehrere Zwecke: Sie ermöglicht erst eine Übersicht über die diversen Microservices, hält die Konfiguration der Services und der Kommunikationsbeziehungen untereinander an einer zentralen Stelle sicht- sowie editierbar und ermöglicht durch Erweiterungen an den ohnehin vorhandenen Proxys die Testaufzeichnung und die Wiedergabe von Testfällen, ohne den speziellen Proxy der redbots GmbH zu verwenden. Zudem kann man durch geschickte Manipulation der Kommunikationsbeziehungen die in herkömmlichen Monolithen übliche Trennung zwischen Produktions- und Testumgebung logisch abbilden, so dass die Notwendigkeit zusätzlicher Hardware entfällt. Hier gibt es also eine Win-Win-Situation sowohl für den Betrieb wie auch für den Test.

## Tools und Frameworks

| Name | Zweck |
|---|---|
| redbots proxy | Proxy zur Aufzeichnung von http-Netzwerkverkehr und Wiedergabe von aufgezeichneten Kommunikationsprotokollen inklusive Echtzeitmanipulation an Requests und Responses (www.redbots.de ). |
| SoapUI | Werkzeug zur Erzeugung von Soap-Requests, womit Geschäftsprozesse in service-orientierten Architekturen angestartet werden können (www.soapui.org ). |
| JMeter | Lasterzeugungswerkzeug, ähnlich SoapUI, geeignet für höhere Lasten, skaliert sehr gut (jmeter.apache.org ). |
| Istio | Konfigurations- und Management-Werkzeug für Service-Meshes, enthält auch den Proxy Envoy (istio.io, bzw. www.envoyproxy.io ), der ebenfalls http-Netzwerkverkehr aufzeichnen kann. |
| Linkerd | Konfigurations- und Management-Werkzeug für Service-Meshes, ähnlich Istio (https://linkerd.io ), allerdings muss der integrierte Proxy für die Serviceaufzeichnung customized werden. |

---


## Carsten Negrini

ist Diplom-Informatiker und seit 1999 als Softwareentwickler tätig. Seit 2011 ist er Geschäftsführer der redbots GmbH (www.redbots.de), die Beratung von der agilen Projektsteuerung über Konzeption und Architekturcoaching bis hin zu Softwareentwicklung und Testautomatisierung anbietet.

E-Mail: carsten.negrini(at)redbots.de


---

**Bildnachweise:**

redbots GmbH