Alternative zum Prinzip Hoffnung

Kontinuierliche Performanztests mit Gatling und ELK

Thomas Jäger, Kristine Schaal, Renato Vinga-Martins

500 ms – viel länger darf die Antwortzeit nicht sein, bevor der Benutzer eine Webanwendung als träge oder langsam empfindet. Umso wichtiger ist es, Performanztests bei der Entwicklung parallel zu den Unittests aufzubauen und durchzuführen.

Performanzanforderungen – vom Test zur Analyse

Soll eine Anwendung für einen Benutzer fühlbar schnell sein, so ist eine Antwortzeit in der Größenordnung von 500 ms keine ungewöhnliche Anforderung. Es stellt sich die Frage, wann im Projektverlauf der richtige Zeitpunkt für Tests dieser Anforderungen ist. Zu jedem Sprint-Ende? Zum Abnahmezeitpunkt eines Releases?

Nichtfunktionale Eigenschaften haben oft einen Querschnittscharakter: Die Performanzanforderung soll für alle Funktionen der Anwendungen erfüllt werden. Eine Funktion kann also eigentlich erst als fertig implementiert gelten, wenn ihre zugehörigen nichtfunktionalen Eigenschaften ebenfalls erfüllt sind. Da ist es nur konsequent, wenn neben dem Unittest auch der Performanztest kontinuierlich (im nightly Build) durchgeführt wird.

Ein Build muss auch scheitern dürfen, wenn die Anwendung im Laufe der Entwicklung zu langsam geworden ist. Besser ist es jedoch, jederzeit ein Gefühl für die Performanz der eigenen Anwendung zu haben, bevor der Build bricht.

Der Aufbau eines solchen kontinuierlichen Testvorgehens kann überraschend einfach sein. Im Rahmen des Artikels wird gezeigt, wie man mit geringem Aufwand mit den Werkzeugen Gatling, AspectJ und ELK effizient Tests durchführen kann. Dabei unterstützt die vorgestellte Werkzeugkette nicht nur die Messung von außen, sondern insbesondere auch die Messung innerhalb der Anwendung, um Logikschicht, Persistenz und Zugriffe auf Fremdsysteme getrennt voneinander im Auge behalten zu können.

Demoprojekt Kundenverwaltung

Zur Demonstration der Werkzeugkette wurde eine kleine Anwendung mit Spring Boot [SpringBoot] implementiert, mit deren Hilfe die aufgezeigten Technologien nachvollzogen werden können. Die Anwendung definiert klassisch drei Anwendungsschichten:

- Präsentationsschicht mit Spring-MVC-Webcontrollern und Thymeleaf-Templates,
- Anwendungsschicht,
- Persistenz mit Spring-JPA-Data-Repositories.

Die Anwendung ist eine einfache CRUD-Anwendung: Über eine Weboberfläche lassen sich Kunden anlegen, auflisten und löschen. Die Beispielanwendung kann über [GitHub] heruntergeladen werden.

Performanztestfälle mit Gatling definieren und ausführen

Als Werkzeug zum Entwerfen und Ausführen der Testfälle wurde Gatling [Gatling] gewählt, ein Last- und Performanztest-Framework, welches auf den Technologien Scala, Akka und Netty basiert. Im Gegensatz zu vergleichbaren Tools wie etwa jMeter zeichnet sich Gatling insbesondere dadurch aus, dass Testfälle in Scala implementiert sind und nicht mithilfe von XML konfiguriert werden. Auch wer Scala nicht kennt, kann Gatling gut benutzen: Die Programmierschnittstellen von Gatling definiert eine sehr gut lesbare domänenspezifische Sprache (DSL), wie man im Folgenden sehen wird. Außerdem bringt Gatling seine eigenen Auswertungen und HTML-Reports mit.

In den folgenden Code-Beispielen wurde Gatling in der aktuellen Version 2.1.7 genutzt, um die Testszenarien auf dem Demoprojekt zu definieren. Im Demoprojekt wurde Gatling über Maven integriert. Ein Gatling-Test kann mit dem Maven-Befehl `mvn gatling:execute` aufgerufen werden.

Am einfachsten ist der Einstieg über den Testfall-Recorder, den Gatling mitbringt. Bei dem Recorder handelt es sich um einen HTTP-Proxy, der in jedem Browser konfiguriert werden kann. Details hierzu sind in [GatlingQuickstart] ausgeführt. Ist der Proxy gesetzt, klickt man einen passenden Testfall direkt in der Anwendung zusammen und speichert dann die Simulation. Gatling hat damit eine erste Testfalldefinition generiert. Dieser generierte Code ist gut lesbar und daher eine ideale Basis für die weitere Arbeit.

```
1 import scala.concurrent.duration._
2 import io.gatling.core.Predef._
3 import io.gatling.http.Predef._
4 import io.gatling.jdbc.Predef._
5
6 class RecordedSimulation extends Simulation {
7   val scn = scenario("RecordedSimulation")
8     .exec(http("request_0")
9     .get("/customers"))
10  val httpProtocol = http.baseURL("http://localhost:8080")
11  setUp(scn.inject(atOnceUsers(1))).protocols(httpProtocol)
12 }
```

Listing 1: Mit dem Recorder erstellte Simulation (Klasse RecordedSimulation)

Im generierten Code (s. Listing 1) findet man bereits den Klassenrumpf sowie die wichtigsten Methoden:

- Zeile 1: Die benötigten Scala- und Gatling-Klassen werden importiert.
- Zeile 6: Jede Testfalldefinition muss von der Gatling-Simulation ableiten.
- Zeile 7: Der Kern des Testfalls ist das „Scenario“. Dieses definiert den Ablauf eines Tests. Man kann in einem Testfall mehrere Szenarien definieren und diese parallel ablaufen lassen.
- Zeile 8: Definition eines HTTP-GET-Requests auf „/customers“.
- Zeile 10: Definition des Protokolls und der Basis-Url.
- Zeile 11: Bisher wurde der Testfall nur definiert. Gestartet wird er über die Methode **setUp**, bei der man das Scenario mitgibt sowie weitere Anweisungen. Im Beispiel ist es der einfache Fall, nur jeweils einen Benutzer das Scenario ausführen zu lassen.

Im Folgenden werden die Testfälle für die Demoanwendung definiert: Es soll getestet werden, wie sich die Anwendung unter schreibenden und lesenden Zugriffen verhält. Zunächst werden einige Kunden angelegt. Zum Anlegen eines Kunden definiert man, ähnlich wie in Listing 1 gezeigt, einen HTTP-Request, jedoch diesmal mit einer **post**-Methode mit Form-Parametern (s. Listing 2).

```
exec(http("Neuen Kunden anlegen")
  .post("/customers")
  .formParam("firstName", "Vorname")
  .formParam("lastName", "Nachname"))
```

Listing 2: Kunden anlegen

Um mehrere Kunden anzulegen, könnte man den Aufruf aus Listing 2 mit einer Scala-for-Schleife wiederholen. Das ist aber sehr ungeschickt. Zur Erinnerung: Hier wird der Testfall nur definiert. Man würde damit quasi n-mal den Aufruf aus Listing 2 hintereinanderschreiben. Besser ist es, ein von Gatling angebotenes Konstrukt zu nutzen, das die Wiederholung erst zur Laufzeit ausführt. Listing 3 zeigt ein **repeat**.

```
scenario("CreateCustomers").repeat(100, "customerNo")(
  exec(http("Neuen Kunden anlegen")
    .post("/customers")
    .formParam("firstName", "Vorname_${customerNo}")
    .formParam("lastName", "Nachname_${customerNo}")
  )
)
```

Listing 3: Mit repeat mehrere Kunden anlegen

Neben der Zahl der Wiederholungen kann ein Zählername (hier **customerNo**) mitgegeben werden, um beispielsweise die Namen der Kunden zu variieren.

In dem gezeigten Beispiel wurden die Testdaten (die Namen der Kunden) direkt im Testfall generiert. Gatling bietet aber auch Unterstützung, um Testdaten von außen einzulesen. Dafür gibt es Feeder für verschiedene Quellen, beispielsweise für csv-Dateien oder (über eine JDBC-Connection) für eine Datenbank. Ein Beispiel findet sich im Code in [GitHub].

Um die Lese-Performanz zu testen, soll die Anwendung mit verschiedenen Szenarien parallel unter Last gesetzt werden. In Listing 4 sind zwei Szenarien dargestellt, eines sendet Requests an die Übersichtsseite, das andere an die Kunden-Detailseite. Die Methode **randomSwitch** dient dazu, zufällig Kundendetails von Kunden 1 oder Kunde 2 zu lesen.

```
val readCustomersScn = scenario("ReadCustomers").repeat(200)(
  exec(http("Kundenübersicht lesen")
    .get("/customers"))
)
val readCustomerDetailsScn =
  scenario("readCustomerDetails").repeat(500)(
  exec(http("Kundendetails lesen")
    .get("/customer/1"))
    .randomSwitch(
      30.0 -> exec(http("Kundendetails lesen").get("/customer/1")),
      70.0 -> exec(http("Kundendetails lesen").get("/customer/2"))
    )
)
```

Listing 4: Lesende Testfälle

```
1 setUp(
2   readCustomersScn.inject(atOnceUsers(7)),
3   readCustomerDetailsScn.inject(
4       rampUsersPerSec(10) to 20 during(10 seconds) randomized))
5     .protocols(httpProtocol)
6     .assertions(
7       global.successfulRequests.percent.greaterThan(99),
8       forAll.responseTime.max.lessThan(500))
9         .maxDuration(1 minute)
```

Listing 5: Verschiedene Szenarien parallel

Diesmal soll eine große Zahl an Requests parallel die Anwendung unter Last setzen. Außerdem soll eine unterschiedlich intensive Belastung simuliert werden. Das vollständige Setup ist in Listing 5 dargestellt:

- Zeile 2: Im ersten Scenario werden sieben parallele Benutzer simuliert.
- Zeile 4: Das zweite Szenario startet mit 10 Benutzern, dann kommen weitere dazu, bis es 20 sind.
- Zeile 6: Mit Assertions wird geprüft, ob 99 Prozent der Requests erfolgreich waren und ob alle innerhalb von 500 ms beendet wurden.
- Zeile 9: Die gesamte Simulation soll nicht länger als eine Minute laufen. Dafür sorgt **maxDuration**.

Schon diese kleinen Beispiele zeigen die Mächtigkeit von Gatling zur Definition von Last- und Performanztestszenarien. Mit Scala steht die gesamte Mächtigkeit einer Programmiersprache zur Verfügung, um auch elaborierte Testszenarien zu schreiben. Gatling definiert eine sehr elegante DSL zum Erstellen von Testszenarien. Den Code und die meisten Sprachelemente versteht man ohne lange Erklärungen. Nicht zuletzt ist Scala besonders gut dazu geeignet, diese ausdrucksstarken Sprachmittel zu kreieren, erlaubt die Sprache doch so elegante Konstrukte wie das Argument „10 seconds“ als Argument von „during“.

Setzen und Protokollieren von Messpunkten in Java-Programmen

Während Gatling gut geeignet ist, die Anwendung unter Last zu setzen und von außen die Performanz zu messen, kann man damit natürlich nicht die Performanz innerhalb einer Anwendung erfassen. Für diese Messung wird also noch ein weiteres Werkzeug benötigt.

Doch wo sind Messpunkte überhaupt sinnvoll? In einer ersten Iteration zur Auswahl von Messpunkten können

- die Aufrufe der Schnittstellen einer Anwendungsschicht (Präsentation, Logik, Persistenz) und
- die Aufrufe von IO-lastigen Funktionen (Aufruf von Drittsystemen, Lesen oder Schreiben von Daten auf die Datenbank oder auf die Festplatte)

verwendet werden.

In der Demoanwendung wurde AspectJ zum Setzen der Messpunkte verwendet. AspectJ ermöglicht es, vor und nach einem Methodenaufruf Code auszuführen, ohne dass dafür eine Änderung an der bestehenden Anwendung vorgenommen werden muss. Dazu schreibt man den auszuführenden Code in einem Aspekt, welcher als Bytecode während des Ladens einer Klasse an die konfigurierte Stelle injiziert wird. Da diese Instrumentierung vor dem Laden der ersten Klasse der Anwendung erfolgen muss, wird AspectJ mithilfe eines Agenten in der Java-Runtime gestartet:

```
java -javaagent:lib/aspectjweaver-1.8.8.jar
```

Während des Starts der Anwendung benötigt AspectJ eine Konfiguration aus dem Klassenpfad META-INF/aop.xml. In dieser Konfiguration wird angegeben, welche Klassen generell instrumentiert werden und welche Aspekte in die Software integriert werden sollen. In der Demoanwendung wird genau ein Aspekt definiert, die Messung der Zeiten für einen Methodenaufruf: die Klasse `PerformanceLogger` (s. Listing 6).

Über die Annotation `@Around` innerhalb des `PerformanceLogger`s wird dem AspectJ-Framework mitgeteilt, dass der Aspekt vor der als Argument angegebenen Codestelle aufgerufen werden soll. Im Beispiel wird der Aufruf aller Methoden aus den Controller-Klassen adressiert.

```
@Around("execution(public * " +
    "de.accso.performancetesting.web.*Controller.*(..))")
public Object measureControllerCall(ProceedingJoinPoint thisJoinPoint)
    throws Throwable {
  return measureTime(thisJoinPoint);
}
private Object measureTime(ProceedingJoinPoint thisJoinPoint)
    throws Throwable {
  StopWatch sp = new StopWatch();
  sp.start();
  Object result = thisJoinPoint.proceed();
  sp.stop();
  logger.info(
  append("durationinmillis", sp.getTotalTimeMillis())
    .and(append("servicename", thisJoinPoint.getTarget()
    .getClass().getName()
    + "." + thisJoinPoint.getSignature().getName())),
    "Performance");
  return result;
}
```

Listing 6: Codebeispiel Klasse PerformanceLogger

Vor jedem Aufruf der Zielmethode wird eine Stoppuhr gestartet, nach dem Aufruf der Zielmethode gestoppt und die verbrauchte Zeit geloggt. Zusammenfassend wird mit dieser Implementierung also die Aufrufzeit aller Methoden der Webcontroller innerhalb der Anwendung gemessen und protokolliert.

Zum Protokollieren der verbrauchten Zeit kommt schließlich ein spezieller Logger zum Einsatz: ein Logger, der in eine Datei direkt im richtigen Zielformat (JSON) für die spätere Analyse mit ELK loggt. Damit kommt man zur dritten Phase des Performanztests: der Auswertung.

Auswertung der Tests mit ELK

In jüngerer Zeit integrieren zunehmend Unternehmen die Open-Source-Werkzeugkette ELK des Unternehmens Elastic in ihr Monitoring. ELK steht für die Hauptprodukte *E*lasticsearch, *L*ogstash und *K*ibana. Wie das vorliegende Beispiel zeigt, ist ELK auch für die Analyse von Performanztests sehr gut geeignet.

Die Suchmaschine Elasticsearch bildet das Kernstück der Infrastruktur. Logstash ist für die Datenvorverarbeitung und die Zulieferung an Elasticsearch zuständig, während Kibana seine vielfältigen Visualisierungen auf die Elasticsearch-Suchschnittstelle abbildet.

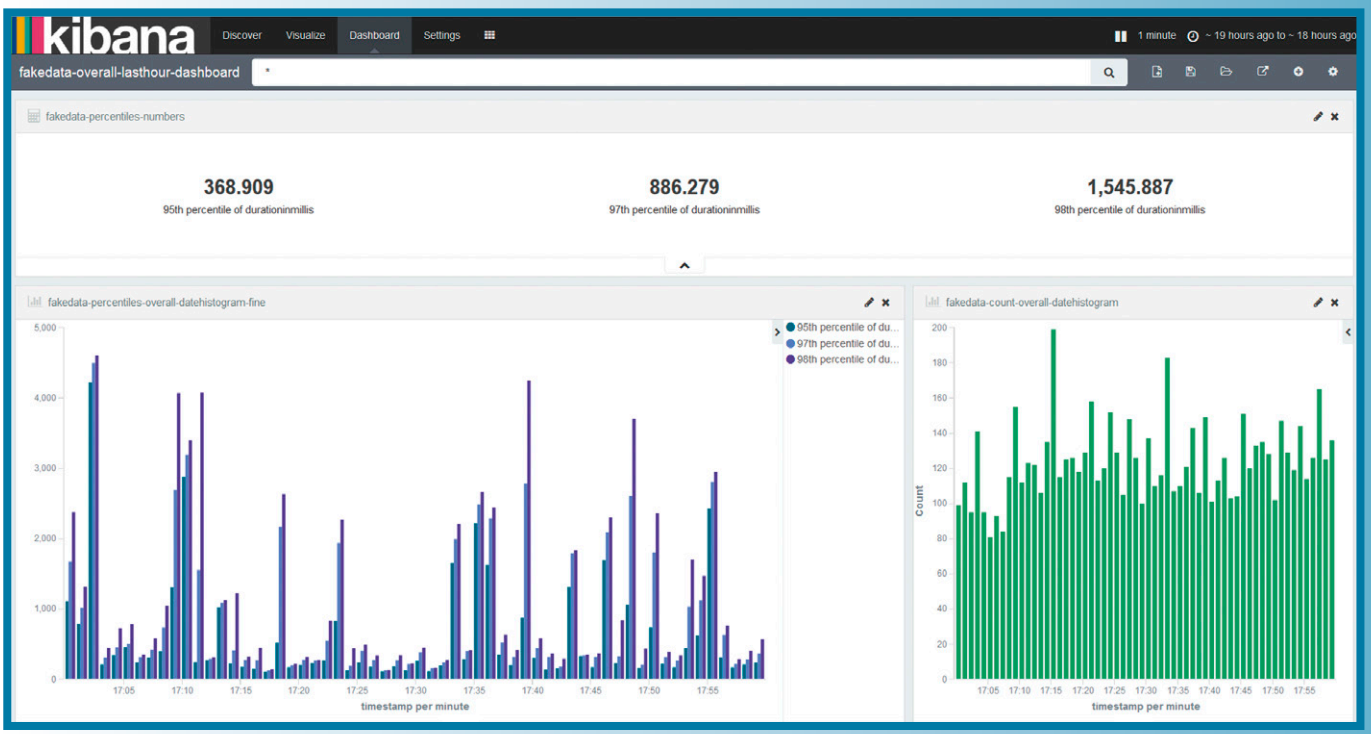

Abb. 1: Kibana-Darstellung während eines Performanztests

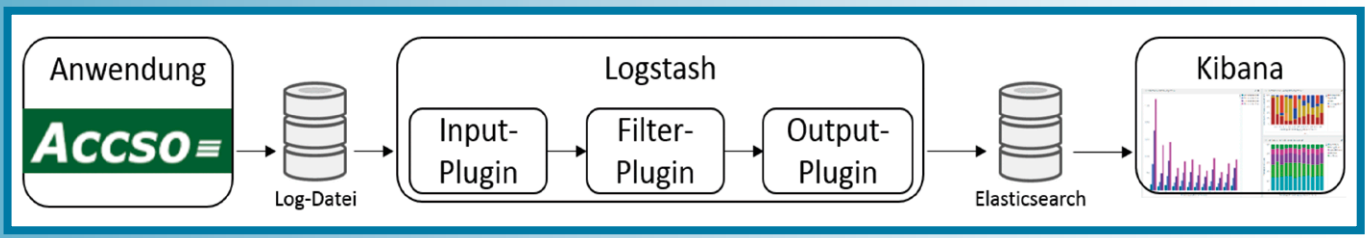

Abb. 2: Die ELK-Werkzeugkette in der Übersicht

Im vorgesehenen Performanztest-Szenario liegt das Augenmerk auf dem Einhalten von Performanzvorgaben. Erfahrungen zeigen, dass in den meisten Fällen harte Maximalvorgaben weder notwendig noch mit angemessenem Aufwand erreichbar sind. Stattdessen ist es zielführender, mit weichen Vorgaben in Form „mittlerer" Maximalwerte zu arbeiten. Dabei nimmt man bewusst in Kauf, nur einen bestimmten Anteil der Aufrufe innerhalb einer vorgegebenen Dauer zu bedienen. Typischerweise gibt es gestaffelt mehrere dieser Perzentilvorgaben, wie 95 Prozent aller Anfragen innerhalb von 0,5 s, 97 Prozent innerhalb von 1 s und 98 Prozent innerhalb von 1,5 s.

Während eines Performanztests könnte eine Kibana-Darstellung wie in Abbildung 1 entstehen: Sie zeigt in der Übersicht eine zurückliegende Teststunde, oben die drei genannten Perzentile als Gesamtwerte, unten links die Perzentile in einminütigen Intervallen und ergänzend unten rechts die Aufrufanzahl ebenfalls in einminütigen Intervallen.

Die Verlaufssichten ermöglichen es, Performanzschwankungen mit Testszenarien in Verbindung zu bringen, wobei die Aufrufanzahl als Indikator für die Last dient. Die Einstellung des Beobachtungszeitraums erfolgt zentral für alle Visualisierungen der Übersicht, das schnelle Hineinzoomen per Mausauswahl unterstützt Kibana ebenfalls. Die Beispielübersicht verdeutlicht schnell wiederkehrende Performanzspitzen, die mit Belastungsspitzen zusammenhängen könnten und dafür verantwortlich sind, dass die Anwendung die 98-Prozent-Vorgabe nicht einhält.

Eine Übersicht wie diese führt sofort zu Fragen über die Rahmenbedingungen: Eine geringe Request-Anzahl, hier indirekt als einzelnes Minutenintervall sichtbar, führt zu starken Perzentilschwankungen und schwächt die angestrebte Mittelung ab. Damit liegen einige Spitzen deutlich über den Stundenperzentilen. Diese engmaschige Verlaufsdarstellung ist aber fürs Team zur Engpassidentifizierung und als Frühwarnsystem wichtig.

Derartige Betrachtungen brauchen Zeit, um ein fundiertes Verständnis für sinnvolle Zielvorgaben zu entwickeln und zu vermitteln. Umso wichtiger ist es, in diese Diskussionen frühzeitig einzusteigen und den Stand entwicklungsbegleitend kontinuierlich zu überprüfen.

Abbildung 2 zeigt das Zusammenspiel der beteiligten Komponenten der ELK-Werkzeugkette, von der ausgemessenen Anwendung ausgehend bis zur Visualisierung in Kibana. Die Aufgaben der Komponenten sind:

- Die *Anwendung* legt ihre Messdaten im Dateisystem ab.
- *Logstash* ist eine Ruby-Anwendung, deren Verarbeitung in den drei Phasen Input, Filter und Output verläuft. Das gewünschte Verhalten entsteht durch die passende Plug-in-Wahl für die einzelnen Phasen. Das file-Input-Plug-in von Logstash überwacht die Messdaten-Dateien und liest kontinuierlich neue Datensätze ein. Andere Input-Plug-ins ermöglichen alternative Datenquellen. Die filter-Plug-ins transformieren die Eingabedaten in passende Datensätze und reichern sie gegebenenfalls um weitere Elemente an. So stehen filter-Plug-ins für Apache-Log-, CSV- oder JSON-Formate zur Verfügung. Das elasticsearch-Output-Plug-in übergibt die Datensätze zur Indexierung an Elasticsearch. Durch seinen Pipeline-artigen Aufbau ist Logstash über das hier gezeigte Beispiel hinaus wie ein ETL-Werkzeug (Extract, Transform, Load) einsetzbar, ermöglicht also beispielsweise auch den Datenexport aus Elasticsearch. Mithilfe der Performanzmessdaten-Datei sind Anwendung und Logstash voneinander betrieblich entkoppelt. Probleme in der Datenauswertung beeinträchtigen somit nicht die Datenerhebung, und die erhobenen Daten stehen nachträglich unabhängig von Elasticsearch zur Verfügung. Erweisen sich die filter-Transformationen als zu Ressourcen-intensiv, um sie auf dem Anwendungsknoten laufen zu lassen, kann man Logstash auf einen separaten Knoten verlagern. In dieser Konstellation läuft auf dem Anwendungsknoten nur die zusätzliche Komponente Filebeat mit, die als einfacher Log-Shipper die Messdaten 1:1 an die entfernte Logstash-Installation weiterreicht.
- *Elasticsearch* ist eine auf der Suchmaschine Lucene aufbauende Java-Anwendung. Die Elasticsearch-Installation indexiert die von Logstash gelieferten Daten. Elasticsearch kann den angelieferten Daten selbstständig die benötigten Schema-Informationen entnehmen, sodass im einfachsten Fall keinerlei Konfiguration notwendig ist. Elasticsearch organisiert logisch zusammengehörige Dokumente (Datensätze) in einem benannten Index, vergleichbar mit einer Datenbank. Logstash gibt den Indexnamen bei der Elasticsearch-Adressierung an. Elasticsearch selbst besitzt keine Nutzer- oder Administrationsoberfläche. Alle Funktionsblöcke, wie für Index- oder Dokumentverwaltung oder für Suchanfragen, stehen als REST-Schnittstellen mittels JSON-über-HTTP zur Verfügung. Der Zugriff ist also mithilfe von Kommandozeilenwerkzeugen wie „curl" oder Browser-REST-Plug-ins möglich. Wer Bedienoberflächen bevorzugt, dem sei das von Elasticsearch unabhängige Open-Source-Werkzeug Elastic HQ empfohlen.
- *Kibana* ist eine auf Node.js laufende JavaScript-Anwendung zur Visualisierung der Elasticsearch-Daten. Kibana fragt von Elasticsearch die komfortabel per Browser-Bedienoberfläche konfigurierbaren Aggregationen ab. Ausgewählte Visualisierungen gruppiert der Nutzer in sogenannten Dashboards und erhält so durch das Laden eines Dashboards eine prägnante Auswertungssicht. Kibana speichert seine Konfigurationen in einem eigenen Elasticsearch-Index.

Fazit

Am Ende muss die Performanz stimmen! Performanz- und Lasttests dürfen daher nicht fehlen, wenn das Team die Anforderungen ernst nimmt. Solche Tests sind mit der fertigen Software für eine abschließende Optimierungsphase leichter zu organisieren. Aber was heißt da „fertig", wenn wir doch agil ent-

wickeln? Und wie wollen wir, wenn es schlecht läuft, ernsthaft so spät noch Architekturänderungen umsetzen?

Dieser Artikel zeigt, dass der kontinuierliche Performanztest inzwischen eine kostengünstige, professionelle Alternative zum Prinzip Hoffnung ist. Der Hauptaufwand liegt in der Testfalldefinition. Die notwendigen Werkzeuge Gatling, AspectJ und der ELK-Stack sind dagegen im Handumdrehen aufgesetzt. So wird das Performanz-Kibana-Dashboard zum normalen Bestandteil des täglichen Stand-up-Meetings, flankierend zu den Ergebnissen der bereits routinemäßig ablaufenden automatisierten Tests. Unangenehme Überraschungen bei der Performanz gehören damit der Vergangenheit an. Und das ist ein sehr gutes Gefühl.

Links

[ElasticHQ] `http://www.elastichq.org/`
[ELKArchitektur/Skalierung] `https://www.elastic.co/guide/en/logstash/current/deploying-and-scaling.html` und `http://ecmarchitect.com/archives/2015/07/27/4031`
[ELKDoku] `https://www.elastic.co/guide/index.html` und `https://www.timroes.de/2015/02/07/kibana-4-tutorial-part-1-introduction/`
[Gatling] `http://gatling.io`
[GatlingQuickstart] `http://gatling.io/docs/2.1.7/quickstart.html`
[GitHub] `https://github.com/accso/performancetesting-example`
[SpringBoot] `http://projects.spring.io/spring-boot/`

Thomas Jäger ist Cheftechnologe bei der Accso - Accelerated Solutions GmbH. Seine Schwerpunkte liegen in der technischen Architektur und Webanwendungen.
E-Mail: thomas.jaeger@accso.de

Dr. Kristine Schaal ist als Softwarearchitektin bei der Accso - Accelerated Solutions GmbH tätig. Sie arbeitet seit fast 20 Jahren in der Softwareentwicklung und ist in Projekten der Individualentwicklung für Kunden verschiedener Branchen unterwegs.
E-Mail: kristine.schaal@accso.de

Dr. Renato Vinga-Martins arbeitet bei der Accso – Accelerated Solutions GmbH mit technologischem Schwerpunkt als Architekt, Berater und Projektleiter in allen Phasen der Individualsoftwareentwicklung. Er begleitet seit über 20 Jahren Kunden in ihren Projekten.
E-Mail: renato.vinga-martins@accso.de