Android 4.2 Jelly Bean: Die Nutzerseite ▸107

S&S

JAVA Mag

Deutschland €9,80 Österreich €10,80 Schweiz sFr 19,50 Luxemburg €11,15 3.2013

Java magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

JavaScript-Tools

Automatisch, praktisch, gut ▸36

Android Development Tools

Ein Werkzeugkasten für Android ▸115

jax 2013

Alle Infos hier im Heft! ▸19

WEB-APPS MIT PLAY

Komplexe Java-Webanwendungen mit dem Play-Framework entwickeln ▸22

PaaS-Newcomer Jelastic

Die elastische JVM ▸94

Eclipse Gyrex

PaaS für OSGi-Anwendungen ▸98

Interview mit Doug Cutting

Dem Hadoop-Hype gerecht werden ▸16

4 194586 709801 03

Das Java-Monitoring-Tool MoSKito unter der Lupe

DevOps in der Praxis

Das Aufspüren von Fehlern in einer laufenden Applikation muss meistens schnell und unter hohem Druck erfolgen. Mit MoSKito betritt ein Open-Source-DevOps-Werkzeug die Bühne, das sich auf das Überwachen der Performance und Fehlerfreiheit der Anwendung zur Laufzeit konzentriert.

von Leon Rosenberg und Michael Schütz

Es ist geschafft. Die eigene Anwendung ist live. Doch damit ist die Entwicklung meist nicht abgeschlossen. Ab jetzt verändert sich die Software durch ständige Weiterentwicklung umso mehr. In diesem Wandel ist es entscheidend, das Laufzeitverhalten der eigenen Anwendung rechtzeitig zu verstehen. Hier kommen DevOps [1] ins Spiel. Ihr Aufgabenfeld umfasst sowohl die Entwicklung als auch den Betrieb. Zur Unterstützung stehen Ihnen etablierte serverbasierte Monitoring-Lösungen wie Nagios [2] zur Verfügung. Diese sind jedoch auf die Betriebssystemebene spezialisiert. Die Analyse auf Softwareseite ist meist mühselig.

Im Leben eines Entwicklerteams kommt irgendwann der Moment, in dem es am liebsten die Zeit anhalten möchte, die Anwendung Schraube für Schraube auseinandernehmen, verstehen, reparieren, ölen und wieder zusammensetzen möchte. Leider bleibt selten Zeit dazu. Selbst wenn die Anwendung nur noch langsam reagiert oder oft abstürzt – sie befindet sich im Livebetrieb und muss weiterhin ihre Dienste Tausenden von Nutzern zur Verfügung stellen. Dazu kommt, dass sich die Ursache der Probleme in Stresssituationen ohnehin schwer aufspüren lässt. Bühne frei für MoSKito!

MoSKito ist ein Open-Source-Projekt zum Messen, Analysieren und Optimieren der Geschwindigkeit und des Verhaltens einer Applikation. Ziel von MoSKito ist es, jederzeit Überblick über den Status der eigenen Applikationen zu haben. Wenn das Verständnis der Applikation zur Selbstverständlichkeit wird, schafft das Sicherheit und Vertrauen. So erhöht sich die Chance, Probleme im Ernstfall schneller lösen zu können.

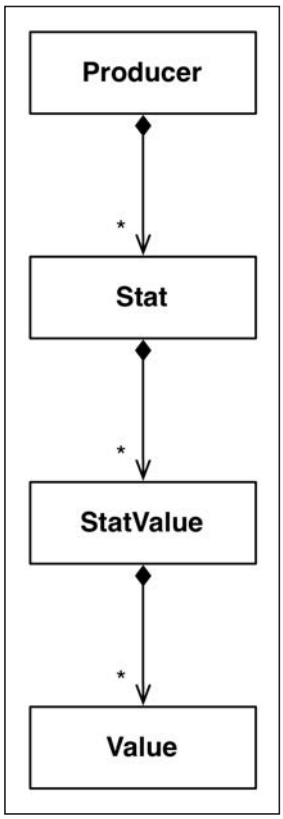


Abb. 1: Producer, Stat, StatValue, Value

Ein Open-Source-Monitoring-Framework für Java

Als Monitoring-Framework integriert sich MoSKito in die Applikation. Im Unterschied zu kostenpflichtigen alternativen Produkten wie AppDynamics [3] oder New Relic [4] erfolgt die Integration punktuell an den Stellen, an denen es darauf ankommt, dass sie überwacht werden. Idealerweise integriert man MoSKito an den Systemein- und ausgängen sowie an den Übergängen zwischen verschiedenen Schichten. Das können Komponenten wie Servlets, Filter, Actions, Services oder DAOs sein. Natürlich ist es dabei sinnvoll, den besonders rechen- oder fehlerintensiven Teilen der Applikation mehr Aufmerksamkeit zu schenken. Dabei sollten komplexe Komponenten wie z. B. eine Regel-Engine oder ein Tarifrechner genauso wenig fehlen wie der Payment-Provider, das Mail Gateway oder sonstige externe Systeme. Je mehr Systeme, Komponenten, Ein-, Aus- und Übergänge erfasst sind, desto klarer ist das Gesamtbild und desto geringer die Gefahr, durch weiße Flecken auf der Monitoring-Karte Probleme zu übersehen. Die Basiskonzepte von MoSKito bilden Producer, Stats und Values (**Abb. 1**), die im Folgenden vorgestellt werden.

Producer produzieren Statistiken. Ein Producer ist demnach etwas, was in irgendeiner Weise Arbeit innerhalb der Applikation verrichtet und Statistiken über diese Arbeit produzieren kann. Das kann ein Service sein, oder auch ein HTTP-Filter, Servlet, MVC Controller, eine Jersey Resource, ein eingebundenes API oder eine Datenbank. Jeder Producer kann mehrere Statistiken gleichen Typs produzieren, die *Stats* heißen. Im Falle eines Services wäre jede Methode ein Stat, dem ein Stat-Objekt entspricht, im Falle eines HTTP-Filters könnte jeder aufgerufene URL ein Stat-Objekt sein. Oft ist es nützlich und daher üblich, kumulierte Statistiken über den Gesamtaufwand eines Producers zu führen. Bei einem Service wären es alle Aufrufe zu dem Service, egal welche Methode dabei aufgerufen wurde. Die kumulierten Werte werden sowohl für Übersichten verwendet als auch dafür, schnell einen Überblick über die Arbeitslast verschiedener Producer zu erhalten.

Jede Statistik und somit jedes Stat-Objekt bestehen aus mehreren Werten, das könnten z. B. Anzahl der Requests, CPU-Zeit in der Methode, Anzahl der Fehler oder parallele Aufrufe sein. Ein solcher Wert heißt *StatValue*. Ein StatValue ist allerdings kein einfacher Wert, sondern ein multidimensionaler Zähler. Die Dimensionen werden dabei von Zeitintervallen gebildet, schematisch in **Abbildung 2** dargestellt. Zeitintervalle

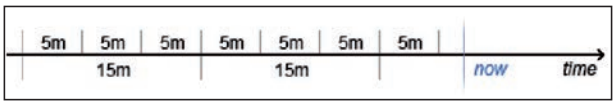


Abb. 2: MoSKito-Intervalle

in MoSKito dienen dazu, die Sichtbarkeit zu erhöhen und das Systemverhalten in kurzen Abschnitten zu betrachten.

Durch Intervalle Sichtbarkeit erhöhen

Das Problem mit dem Sammeln großer Mengen von Daten besteht darin, dass die Daten nach gewisser Zeit träge werden, d. h. durch ihre schiere Masse auf Veränderungen nur sehr langsam reagieren. Besonders die Durchschnittswerte stumpfen mit der Zeit ab. Folgendes Beispiel demonstriert das Verhalten:

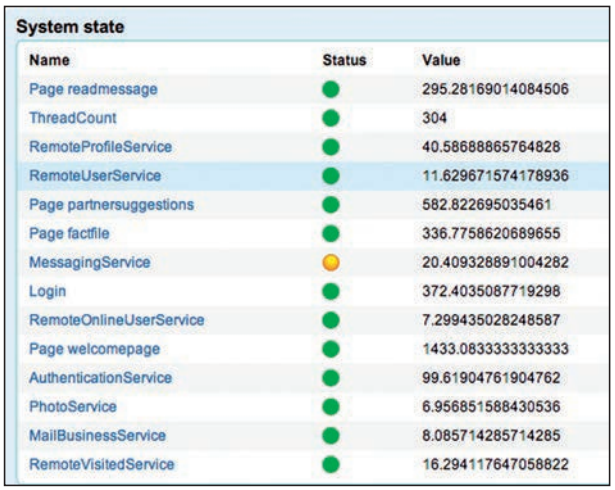
Wenn ein DAO normalerweise zehn Requests pro Minute bedient und 50 Millisekunden CPU-Zeit dabei verbraucht, ergibt das eine durchschnittliche Antwortzeit (Average) von 5 ms/Request. Wenn nach einer Minute ein langer Request 50 Sekunden dauert und die anderen 9 zusammen 45 Millisekunden, so wächst das Average über 2 Minuten auf 50,095 / 20 = 2,50475 Sekunden (!) an. Jeder, der diese Zahl sieht, ist sofort alarmiert!

Zum Problem wird es jedoch, wenn die Verlangsamung erst nach drei Tagen eintrifft. In drei Tagen wurden bereits 4 320 „normale“ Minuten gemessen, in denen 43 200 Requests in 216 000 Millisekunden verarbeitet wurden, was einem Average von 216 000 / 43 200 = 5 ms entspricht. Sollte in der nächsten Minute ein Call kommen, der aufgrund von Datenbankproblemen 50 Sekunden dauert, so wird das neue Average *(216.000 + 45 + 50.000) / (43.200 + 10) = 6,18 Millisekunden* betragen, also dem Betrachter das falsche Gefühl der Sicherheit geben. Die folgenden „50 Sekunden Requests“ werden das Average erst langsam heben und den Ausfall sichtbar machen. Vergleicht man allerdings nur die Intervallwerte, also die letzte Minute oder die letzten fünf Minuten, so ist das Ausmaß des Schadens sofort sichtbar, und alle Thresholds springen an und schlagen Alarm.

Um das zu realisieren, hält jedes StatValue-Objekt ein *Value*-Objekt für jedes Intervall, das aus einem Zähler und dem letzten gezählten Stand besteht und jeweils am Ende des Intervalls aktualisiert wird.

Producer, Stats und Intervalle bilden die Grundlagen zur Sammlung von Daten und damit die Voraussetzung für die Auswertung. Hier kommen Thresholds und Akkumulatoren ins Spiel. Hat man das Gefühl dafür entwickelt, wie das System funktioniert, setzt man an den wichtigen Stellen Thresholds auf Basis von Vergleichsdaten an. Eine Seite mit Statuslämpchen ermöglicht die Systemübersicht auf einen Blick, wie in **Abbildung 3** dargestellt.

Ein weiteres Tool in dieser Phase sind die Akkumulatoren, die konfigurierte oder zusammengeklickte Werte über die Zeit aggregieren und visuell analysieren lassen (**Abb. 4**).



System state

Name	Status	Value
Page readmessage		295.28169014084506
ThreadCount		304
RemoteProfileService		40.58688865764828
RemoteUserService		11.629671574178936
Page partnersuggestions		582.822695035461
Page factfile		336.7758620689655
MessagingService		20.409328891004282
Login		372.4035087719298
RemoteOnlineUserService		7.299435028248587
Page welcomepage		1433.0833333333333
AuthenticationService		99.61904761904762
PhotoService		6.956851588430536
MailBusinessService		8.085714285714285
RemoteVisitedService		16.294117647058822

Abb. 3: MoSKito-Systemstatus anhand von Thresholds

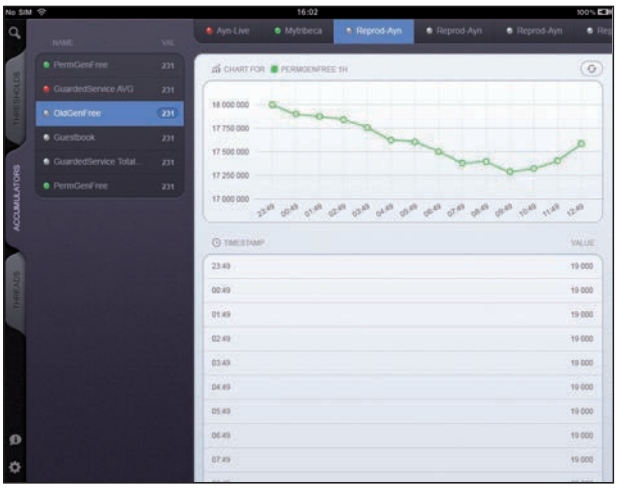


Abb. 4: Akkumulator nach Service-Crash

Das grundsätzliche Vorgehen ist also:

- System mit einem dichten Netz von Monitoring-Punkten durchleuchten
- Daten sammeln und grafisch und numerisch analysieren
- Ergebnisse der Analyse für Thresholds verwenden

Minimalinvasive Integration von MoSKito

Für die Integration von MoSKito stehen mehrere Wege zur Verfügung. Für den Einsatz mit Spring, AOP oder CDI dient die Annotation *@Monitor*, die ganze Klassen oder einzelne Methoden in das Monitoring aufnimmt. Man kann aber auch „selbst zählen", wie Listing 1 zeigt, oder mit Java-Proxies Interfaces überwachen – Listing 2 und 3.

Beispiele aus der Praxis

So weit, so gut – die Basiskonzepte samt Integration von MoSKito sind beleuchtet. Allerdings möchten wir nicht die Illusion wecken, dass nach einmaliger Integration das Monitoring abgeschlossen ist. Die Zeiten, in denen eine Anwendung geschrieben wurde und dann einen langen Zeitraum unter unveränderlichen Bedingungen lief, sind lange vorbei. Dank agiler Methoden wie Scrum oder Kanban können neue Releases alle zwei bis drei Wochen bereitgestellt werden. Mit jedem Release ändern sich die Applikation und ihr Verhalten, durch neue Bugs oder Features fließen die Daten auf anderen Wegen durch die Applikation.

Doch wie genau kann MoSKito hier unterstützend wirken? Zum Beispiel mit Journeys. Eine Journey stellt eine Aufzeichnung der Requests dar, die ein User während einer bestimmten fachlichen Aktion durchläuft. Die *TracedCall* genannten Requests zeichnen den Weg durch die von MoSKito verwalteten Klassen und Services inklusive Ausführungszeiten und übergebener Parameter. Diese Funktionalität mag stark an Debugging erinnern, bietet jedoch den Vorteil, auch in einem Livesystem und unter Last zuverlässig zur Verfügung zu stehen.

In einem verteilten System sind die Requests übers Netz meist langsam im Vergleich zu den lokalen Requests und zahlen mit am meisten auf die Gesamt-Request-Dauer ein. Als wir bei einer großen Casual

Listing 1

```
//wir sind in der Methode und wollen jetzt monitoren
try{
  CallExecution execution =
                    producer.getStats("myMonitoredSectionName").createCallExecution();
  execution.startExecution();
  //Jetzt passiert die ganze überwachte Magie!
  execution.finishExecution();

}catch(OnDemandStatsProducerException e){
//Fehlerbehandlung
}
```

Listing 2

```
SimpleService unmonitoredInstance = new SimpleServiceImpl();
MoskitoInvokationProxy proxy = new MoskitoInvokationProxy(
  unmonitoredInstance,
  new ServiceStatsCallHandler(),
  new ServiceStatsFactory(),
  "SimpleService",
  "service",
  "my-sub-system",
  SimpleService.class
);
SimpleService monitoredInstance = (SimpleService)proxy.createProxy();
```

Listing 3

```
SimpleService monitoredInstance = ProxyUtils.createServiceInstance(
  new SimpleServiceImpl(), "my-sub-system", SimpleService.class);
```

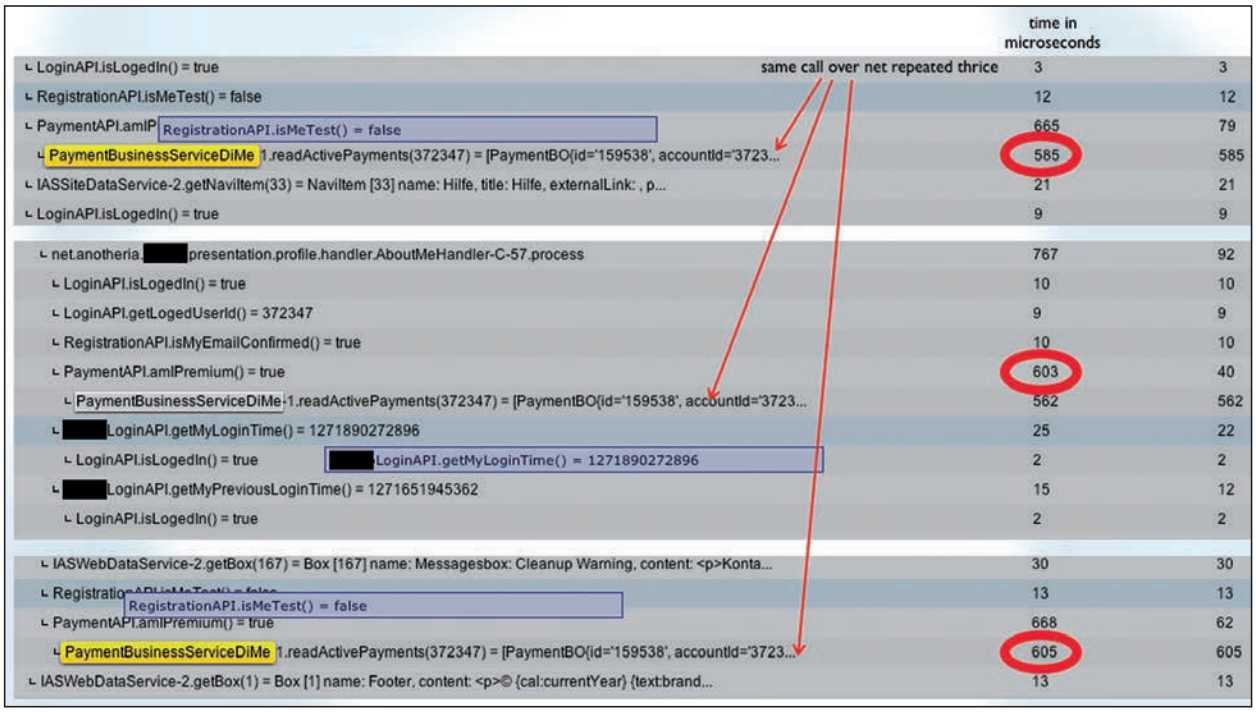


Abb. 5: Requests des PaymentService im Journey

Dating Platform in Europa im Zuge der Performanceoptimierung eine Journey-gestützte Analyse durchgeführt haben, stellten wir fest, dass innerhalb eines HTTP-Requests drei identische Requests zum Payment Service ausgeführt wurden (**Abb. 5**). Jeder einzelne Request dauerte 600 ms, zusammen also 1,8 Sekunden. Nachdem wir es in der Journey aufgedeckt hatten, konnten wir die Ergebnisse des ersten Requests lokal zwischenspeichern und so eine Sekunde bzw. 60 Prozent der Requestdauer einsparen.

Auch zum Live-Debuggen eignen sich die Journeys. Wer ein frequentiertes System betreibt, weiß, wie schwierig es ist, ein Livesystem zu debuggen. Die Log-

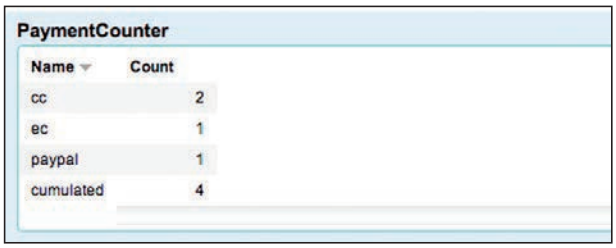


Abb. 6: PaymentCounter Producer in MoSKito

Ausgaben sind meist unzureichend, erst recht im Kontext eines Benutzers. Bei einem anderen MoSKito-User ist genau der Fall eingetreten, dass man einen Fehler hatte, der auf einem Testsystem nicht zu reproduzieren war und live ungenügend Logs hatte: Im Kundendienst ist ein Fehlerreport eingetroffen, dass Benutzer X die zweite Seite seines Postfachs nicht aufmachen konnte. Mit dem Einverständnis des Kunden haben wir eine Journey mit seinen Daten aufgenommen und Aufrufe der ersten und der zweiten Seite des Postfachs verglichen. Wir haben gesehen, dass beim Zusammenbau der Nachrichtenanzeige für eine bestimmte Nachricht eine nicht abgefangene *NullPointerException* auftrat. Der Grund dafür war, dass ein Administrator einen Nutzer direkt in der Datenbank gelöscht und der betroffene Kunde eine Nachricht von diesem gelöschten Nutzer hatte. Das führte zu einer Null auf einem *getUser* an einer bestimmten Stelle im Programm, einer anschließenden *NullPointerException* und, darauf folgend, zum Abbruch beim Rendern der Seite. Durch Identifizieren und Entfernen der Nachricht ließ sich der Fehler leicht reparieren.

Listing 4

```
@Count
public class PaymentCounter {

public void ec(){} // Electronic card payment
public void cc(){} // Credit card payment
public void paypal(){} // PayPal payment
}
```

Listing 5

```
PaymentCounter counter = new PaymentCounter();
//now make different payments
counter.ec();
//...
counter.cc();
//...
counter.paypal();
```

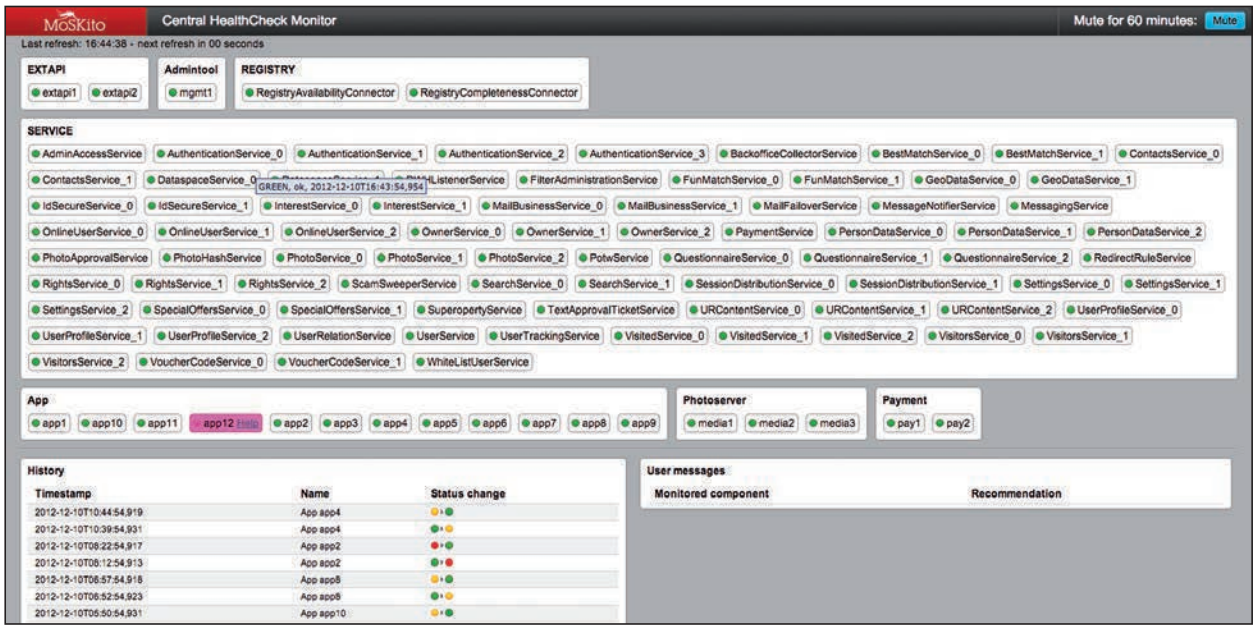


Abb. 7: MoSKito-Erweiterung: Health Check Monitor MoSKito Control

Integration mit JMX

Beim Monitoring mittels Journeys setzt MoSKito auf eigene Monitoring-Punkte. Zusätzlich stellt MoSKito eine direkte Integration nach JMX [5] bereit. Zum einen wird jeder MoSKito Producer als eine MBean registriert und macht die Daten somit für externe Werkzeuge und Anwendungen (z. B. Collectd, Nagios) verfügbar. Zum anderen fragt MoSKito auch die Standard JMX Beans der JVM ab. Auf die Weise kann z. B. Speicher in den verschiedenen Memory Pools überwacht werden. Anhand dieser Informationen lässt sich die Benutzer/Session-Anzahl direkt ins Verhältnis zu den erforderlichen Systemressourcen setzen und erleichtert das Planen und Anpassen der eigenen Serverlandschaft.

Unter den Informationen, die MoSKito aus JMX gewinnt, ist auch die Anzahl der aktuellen Threads sowie der Status, in dem sie sich befinden. MoSKito kann auch eine *ThreadHistory* aufbauen, d. h. verfolgen, welche Threads im System erstellt werden, was sie tun und wann sie sterben.

Wie die Thread-Überwachung der Problemfindung dienen kann, zeigt sich am Beispiel eines Kunden, bei dem sich MoSKito im produktiven Einsatz befindet. Im Livebetrieb und unter Last kam es zu Blockierungen der Threads. Letztere konnten nicht mehr abgebaut werden, der Threadcount stieg und die Anwendung stoppte schließlich. Mittels MoSKito konnte schnell die Thread-Liste nach blockierten Threads durchsucht werden. So ließ sich JSF als Übeltäter identifizieren. JSF erzeugte unnötige Objektinstanzen bei jedem Request. Das ist im Normalbetrieb nicht aufgefallen, dafür aber unter Last. Mit der Information bzgl. blockiertem Thread und JSF-Objekt war es schnell möglich, in Internetforen eine Lösung zu finden. Der Parameter *javax.faces.PROJECT_STAGE* war standardmäßig auf *DEBUG* gesetzt, was zu erneutem Erzeugen der JSF-Klassen führte. Nach einem Umsetzen des Parameters auf *javax.faces.PRODUCTION* in der *web.xml* tauchte das Problem nicht mehr auf.

Unternehmenskennzahlen im Fokus

Das Leben einer Plattform besteht aber nicht nur aus technischen Kennzahlen. Geschäftszahlen sind mindestens genauso wichtig. Grundsätzlich könnte zu jeder Businesskennzahl ein Producer mit entsprechenden Statistiken erstellt werden, der die Daten zur Laufzeit aufnimmt. Zur Erleichterung bietet MoSKito standardmäßig eine Counter-Funktionalität, die es erlaubt, Businesskennzahlen sehr leicht ins Monitoring und damit auch in die Threshold-Überwachung und Graphen mit einzubeziehen.

Dazu ein Beispiel: Der Onlineshop eines Unternehmens expandiert. Das Unternehmen möchte gern mehr Benutzerdaten tracken, um noch mehr auf die Bedürfnisse des Kunden eingehen zu können. Ziel ist es, die Ausfallsicherheit der Bezahlmethoden sicherzustellen. Dafür steht die MoSKito-Count-Funktionalität bereit. Das Framework unterstützt eine direkte Integration für AOP und CDI. Wir werden die CDI-Version kurz beleuchten. Generell ist das Vorgehen ähnlich:

1. Producer zur Sammlung von Daten
2. Akkumulator und Threshold zum Auswerten und Monitoren

Es steht die Annotation *@Count* bereit. Die entsprechend annotierte Klasse *PaymentCounter* in Listing 4 zeigt die Verwendung. Im Hintergrund arbeitet ein CDI Interceptor, um den entsprechenden Producer mit den Werten der einzelnen Methoden bzw. Stats zu befüllen. Die Methoden der Serviceschicht rufen entsprechend die jeweilige Methode des Counters auf. Daraufhin reagiert der Interceptor und inkrementiert den entsprechenden

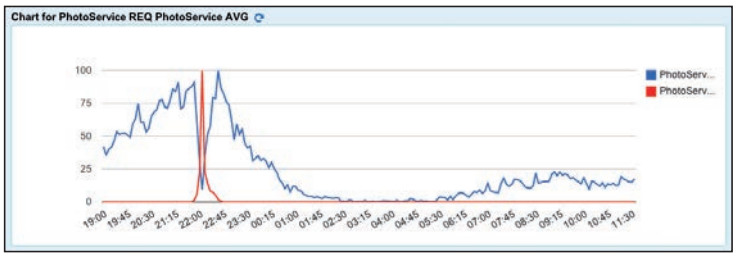


Abb. 8: MoSKito-iPad-Version

Stat des Producers (Listing 5). In MoSKito sieht das entsprechend aus wie in **Abbildung 6**.

Anhand des Producers können die Daten entsprechend gewünschter Intervalle aufbereitet werden, z. B. in Form eines Akkumulators oder Thresholds. Das Unternehmen ist so in der Lage, die Verfügbarkeit der einzelnen Bezahlkanäle zu überwachen und dadurch dem Kunden ein optimales Einkaufserlebnis und dem Unternehmen höchstmögliches SLA im Payment zu garantieren. Optional können zur Benachrichtigung Mails konfiguriert werden, sobald ein kritischer Wert über- oder unterschritten wird.

Komplexe und verteilte Systeme überwachen mittels MoSKito Control

Bisher haben wir primär darüber gesprochen, wie eine Komponente des eigenen Systems überwacht werden kann, sei es eine Web-App in einem Tomcat oder ein Service in einer Java VM. Aber was ist, wenn das System aus Hunderten von VMs besteht, deren Zustand sich auf den Gesamtzustand des Systems auswirkt? Die Suche nach einem konkreten Fehler erfordert das Eintauchen in die jeweilige Komponente. Aber wie genau zeigt sich der Fehler und wo sollte die Suche beginnen? Was ein DevOp braucht, ist eine Möglichkeit, den Zustand des Gesamtsystems zu überblicken und im Falle einer Auffälligkeit den Anhaltspunkt für weitere Untersuchungen zu haben.

Mit MoSKito Control, einer zuerst im Kundenauftrag entwickelten und dann zurückgespendeten Komponente, kann MoSKito auch komplexe Systeme auf einen Blick überschaubar machen. MoSKito Control ist grundsätzlich eine Seite mit vielen grünen Lämpchen (**Abb. 7**). Jede Komponente des Systems entspricht einem Lämpchen, das Grün, Gelb, Rot oder Lila sein kann. Dabei steht Grün für den optimalen Zustand, Gelb für ein kleines, nicht unmittelbar zu behebendes Problem, Rot für ein Problem, das die Komponente selbst festgestellt hat, und Lila für einen Totalausfall und eine Nichterreichbarkeit der Komponente. Die Zustände Grün bis Rot werden durch die Abfrage der MoSKito Thresholds in den jeweiligen Komponenten ermittelt. In regelmäßigen Abständen verbindet sich MoSKito Control zu den einzelnen Komponenten und fragt den Status ab. Hat sich der Status verändert, so sieht man das auf dem Bildschirm sofort. Und wenn nicht, dann wird man durch die E-Mail-Benachrichtigung unterrichtet. Mit einem Fernseher im Entwicklerraum hat das Team den Gesundheitszustand seiner Seite stets im Blick. Ein positiver Nebeneffekt ist dabei, dass sich Entwickler und Administratoren so auf ein gemeinsames System zum Ermitteln des Systemstatus einigen können.

Monitoring unterwegs

Neben MoSKito fürs Web steht auch eine mobile Anbindung bereit. Für iOS inklusive einer speziellen iPad-Version lässt sich MoSKito leicht aus dem App Shop installieren. In **Abbildung 8** wird die iPad-Version dargestellt.

Fazit

Im Rahmen des Artikels haben wir mit MoSKito ein DevOps-Werkzeug vorgestellt, um die eigene Anwendung gezielt überwachen und verstehen zu können. Zum Erscheinen des Artikels liegt MoSKito in Version 2.1.x vor. Neben der Konsolidierung der MoSKito-Funktionalitäten möchten wir in Zukunft weiter Qualität von Dokumentation und Beispielen steigern, um den Einstieg noch weiter zu erleichtern. Außerdem wird die CDI-Integration noch nahtloser erfolgen, und es werden bald neue Mobile-Versionen bereit stehen. Das große Ziel für 2013 ist jedoch das Redesign von MoSKito Central, einer Komponente, die gemessene Werte über längere Zeiträume kumuliert und mittels REST-API für weitere Analysen zur Verfügung stellt. Wir laden Sie ein, den Code selber auszuprobieren. Unter anderem steht ein fertiges Beispiel zur Integration von MoSkito in eine CDI-Umgebung bereit [6].

MosKito als Open-Source-Werkzeug lebt von seiner Community. Daher freuen wir uns über Feedback jeder Art. Dokumentation, Beispiele und Aktuelles finden Sie auf der MoSKito-Homepage [7].

Leon Rosenberg widmet sich als Softwarearchitekt seit über zehn Jahren der Entwicklung von hochskalierenden Portalen und ist seit 2007 MoSKito Founder und Commiter.

dvayanu **rosenberg.leon@gmail.com**

Michael Schütz ist als freiberuflicher Scrum Master und Softwarearchitekt tätig und beschäftigt sich seit Jahren mit Java-Enterprise-Architekturen und -Technologien. Seit 2012 ist er MoSKito Commiter. Gemeinsam mit seinem Team bereitet Michael momentan die offene Java-Konferenz BED-Con 2013 in Berlin vor.

michaelschuetz

Links & Literatur

[1] http://en.wikipedia.org/wiki/DevOps

[2] http://nagios.org

[3] http://www.appdynamics.com

[4] http://newrelic.com

[5] http://de.wikipedia.org/wiki/JMX

[6] https://github.com/anotheria

[7] http://moskito.anotheria.net