Andreas Grabner
(andreas.grabner@dynatrace.com)
ist Performance Advocate bei Dynatrace. Er arbeitet seit mehr als 15 Jahren im Software-Performanceumfeld. Seine Schwerpunkte sind Java-, .NET- und Web-Performance.

Die (R)Evolution des Funktionalen Testens

Die Softwarewelt ist im Wandel. Anstatt einmal pro Jahr eine neue Version zu releasen, schaffen es Unternehmen wie Facebook, Etsy, Twitter und Co., mehrfach täglich neue Features auszurollen. Warum? Um schneller als die Konkurrenz zu sein. Und wir sprechen nicht nur von der bekannten Konkurrenz, sondern auch von neuen Start-ups, die mit Hilfe der Cloud, Docker, GitHub und Co. flexibler sind als so manche etablierte Softwareschmiede. Können die aktuellen Testansätze mit dem Wandel mithalten?

Nein, sie können es wohl nicht, denn auf der anderen Seite sehen wir vermehrt Schlagzeilen wie „Online Banking wieder mal nicht verfügbar“, „Fluglinie streicht Hunderte von Flügen wegen Softwareproblems“ oder „Paketzustellungen zu Weihnachten verzögern sich aufgrund von Softwarefehlern“.

Warum das? Was sind denn die Gründe, warum diese Systeme nicht funktionieren? Können wir mit unseren aktuellen Testansätzen nicht mehr mithalten? Können wir diese Probleme im Testbetrieb überhaupt finden? Wie werden denn mehr Feature Releases getestet als je zuvor? Features, die teilweise nur „on-demand“ oder für eine kleine Gruppe von Benutzern für A/B Tests released werden? Müssen wir einfach nur mehr Tester einstellen oder kaufen wir diese Leistung „as a service“ aus Billiglohnländern zu und werden somit unter dem Aspekt der Kostenkontrolle bald weg-automatisiert?

Die Antwort ist relativ einfach. Es besteht kein Grund zur Panik. Ganz im Gegenteil. Die Gründe, warum diese modernen Anwendungen fehlschlagen, sind durchaus bekannt, zum Beispiel: falsch entworfene Service-Architekturen, zu viele Abfragen und zu wenig Datencaching oder überladene Webseiten, die zum Kollaps der Content Delivery Pipeline führen. Wir haben die Chance, den nächsten (r)evolutionären Schritt für das Testen einzuleiten und die Software automatisiert, aber auch frühzeitig auf die thematisierten Probleme hin zu testen. Aber wie?

Evolution des Agilen Testens

In den letzten Jahren hat uns vor allem „Agiles Testen“ beschäftigt – und damit vor allem mehr Automatisierung und die Verschiebung von hauptsächlich UI-driven Tests hin zu Unit-, Komponenten- und Integrationstest.

Wie die Tendenz zeigt, steht nun ein technisches „Level-Up“ und ein „Shift-Left im Quality Focus“ vor der Tür: Anstatt rein die Funktionalität eines Features über Cucumber, Quick Test Pro oder Selenium zu testen und das Testresultat als „Rot/Grün“ in die Continuous Integrati-

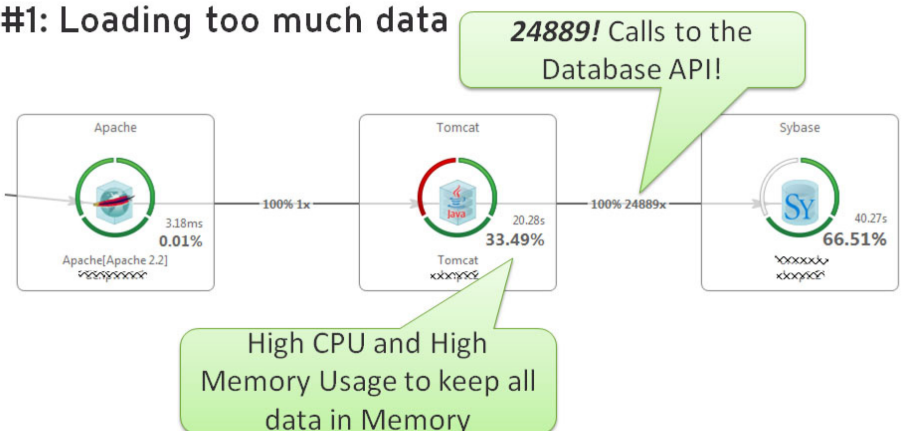

Abb. 1: Wenn uns ein „tested feature“ meldet, dass 24 889 (!) SQL-Abfragen ausgeführt werden, ist dieser Test als „FAILED“ zu betrachten, auch wenn er uns die richtigen funktionalen Resultate liefert!

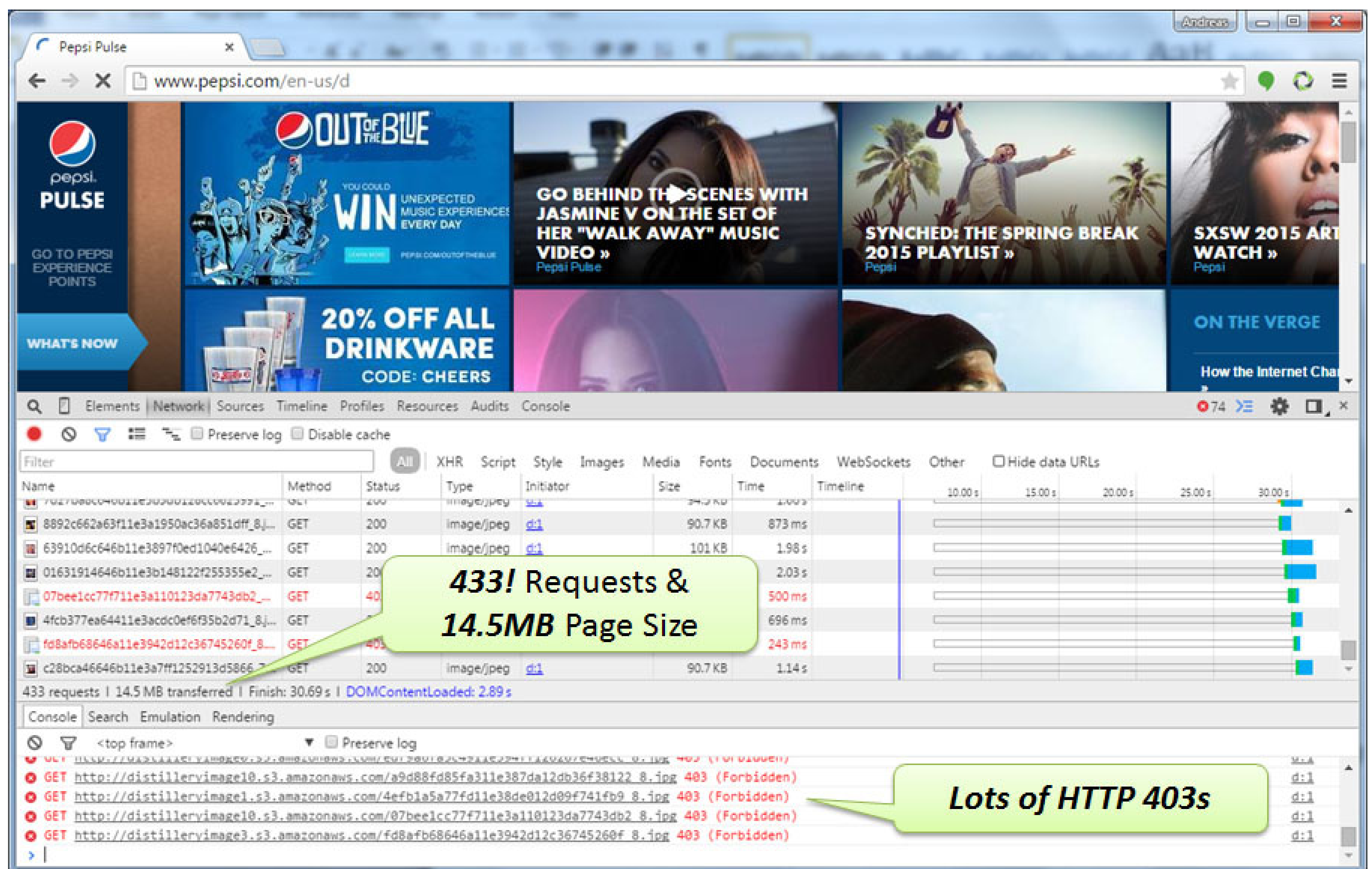

Abb. 2: Die Verwendung des DevTools von Chrome (Ctrl-Shift-I) enthüllt die Schwachpunkte – 433 Roundtrips, 14,5 MB Datenvolumen und viele HTTP-403-Fehler, nur um pepsi.com zu laden

on zu melden, sollten nach jedem Test zusätzlich Architektur-, Skalierungs- und Performance-Aspekte analysiert werden (Level-Up). Und das Ganze nicht nur im traditionell funktionalen UI-driven Test, sondern auch schon früher im Entwicklungszyklus (Shift-Left) bei beispielsweise Unit-, Komponenten- und Integrationstests.

Wenn wir diesen Schritt schaffen, können viele der Probleme, die zu den oben angeführten Schlagzeilen führen, bereits im Keim erstickt werden.

Testen in der Realität

Die Erfahrung zeigt, dass ein Großteil der Tester wenig Erfahrung mit einem durch Performance-Metriken gesteuerten Ansatz für Software-Engineering über den gesamten Lebenszyklus hat. Tester sollten es sich zur Aufgabe machen, sich bei ihren Tests nicht nur auf funktionelle Wirksamkeiten zu fokussieren, sondern auch einen Blick hinter die Kulissen zu werfen, um Dinge zu entdecken, die auch für sie von Interesse sein könnten, wie Anzahl der Elemente auf einer Seite, XHR Calls, SQL-Aufrufe, Speicherzuordnung, CPU-Hotspots oder mangelhaftes architektonisches Verhalten.

Abbildung 1 zeigt ein einleuchtendes Beispiel. Auch wenn ein Test die richtigen funktionalen Resultate liefert, kann er dennoch die Performance-Anforderungen verletzen.

Den meisten Testern ist durchaus bewusst, dass es an der Zeit ist, Lösungen für Leistungsengpässe zu finden, anstatt nur Bugs zu erzeugen. In einer perfekten DevOps-Welt werden Tests mehrheitlich automatisiert mittels Continuous Integration ausgeführt. Dabei werden nicht nur funktionale, sondern auch architekturelle und Implementierungsaspekte anhand von Performance-Metriken getestet. Diese Tests decken automatisch Verschlechterungen bestehender Funktionen auf und liefern viel schneller und technisch besser umsetzbares Feedback an die Entwickler.

Nur die Realität sieht leider ganz anders aus. In Wahrheit ist es nämlich so, dass noch immer ein Großteil der Tester Verfechter von manuellen Tests ist. Hinzu kommt, dass die meisten von ihnen Angst vor der Verwendung von „Werkzeugen, die Entwickler verwenden würden,“ haben und aus diesem Grunde einen großen Respekt vor diesen Metriken zeigen. Tester sind nicht mit der Terminologie in der Umgebung von JavaScript, CSS, XHR, Java, .NET, PHP, Nginx, node.js, Android oder iOS vertraut. Deshalb zweifeln sie auch die Seriosität der von Entwicklern präsentierten Auswertungen der Testergebnisse an, da sie nicht die gleiche Sprache sprechen.

Diese Angst ist durchaus verständlich. Aber jetzt ist es an der Zeit, auf ein höheres Niveau zu kommen!

Warum? Weil manuelle Tests von irgendjemandem in der Welt durchgeführt werden können. Die Arbeitskräfte sind günstig und Dienstleistungen von der Crowd wie Tests und andere machen es für Unternehmen noch leichter, diese Aufgaben auszulagern. Für Tester ist es unausweichlich, ein Teil dieser (R)Evolution zu werden, da nach immer mehr Releases mit besserer Qualität und nicht nur Bugs in Jira verlangt wird.

Vier Schritte zur (R)Evolution des Funktionalen Testens

Sind Sie bereit, Ihren aktuellen Testalltag hinter sich zu lassen und ein „Level-Up“ durchzuführen? Es ist ganz einfach, indem Sie die folgenden vier Schritte befolgen.

- Schritt 1: Machen Sie CTRL-SHIFT-I und F12 zu Ihren Lieblingen

Eines der größten Probleme bei Webseiten ist, dass sie überladen sind und keinen der seit Jahren bekannten „Web Performance Optimization Best Practices“ folgen. Das passiert auch Größen wie Pepsi, wo der Besucher der Webseite in Summe 14,5 MB (hauptsächlich Bilder) runterladen muss, bevor er in den „Genuss“ des Inhaltes kommt. Abbildung 2 zeigt einen Screenshot der Pepsi-Webseite mit Metriken, die genau auf diese Versäumnisse aufmerksam machen.

Tester sind das nächste Sicherheitsnetz – aber nicht nur aus funktionaler Sicht, sondern auch aus Web-Performance-Sicht. Es gibt mittlerweile auch **keine Entschuldigung mehr**, nicht dieselben, verfügbaren Browserdiagnose-Werkzeuge von Firefox, Chrome, Safari und auch dem Internet Explorer zu verwenden, wie es Entwickler verwenden sollten, bevor sie ihre Änderungen einchecken. Warum? Weil sie bereits mit dem Browser mitkommen und sehr einfach genau die schon genannten Performance-Metriken liefern.

Wie geht das? In den neueren Versionen von Chrome und Firefox genügt ein Ctrl-Shift-I und die Entwicklungswerkzeuge stehen im Browser-Fenster zur Verfügung. Durch ein Klicken auf den Netzwerk-Tab kann der manuelle Test gestartet werden. Nach jedem Klick kann die Status-Bar überprüft und erkannt werden, wie viele Roundtrips mit welchen Downloadvolumen notwendig waren, um die Seite zu laden. Wenn mehr als 100 Ressourcen (Bilder, JavaScript- oder CSS-Dateien) und mehr als 5 MB geladen werden, ist der Test als gescheitert zu werten.

Warum? Wenn es dieser Build in die nächste Testphase oder in den Produktionsbetrieb schafft, ist die Wahrscheinlichkeit sehr hoch, dass diese Webapplikation oder die Infrastruktur unter der Last beziehungsweise den nötigen Datenvolumen zusammenbricht. Im Internet Explorer ist es ähnlich – mit dem Drücken von F12 öffnet sich das Entwicklungswerkzeug in einem separaten Fenster. Jetzt wird in der Regel auf „Netzwerk“ geklickt und die Aufzeichnung gestartet. Damit werden alle Aktivitäten in den Test aufgenommen und die Anzahl der Ressourcen und deren Volumen in der Status-Bar angezeigt.

■ Schritt 2: Grundlegende Software-Architektur-Überprüfungen ausführen

Dieser Schritt ist deutlich größer, aber bei Weitem nicht so aufwendig, wie es vielleicht aussieht. Ein Blick auf den in Abbildung 3 abgebildeten Dynatrace Transaction Flow zeigt dies in verständlicher Form. Es spielt keine Rolle, wenn jemand beim Testen mit den verwendeten Technologien wie Java, .NET oder PHP nicht im Detail vertraut ist. Das Lesen und das Verstehen dieser high-level Architektur-Metriken sind denkbar einfach.

Um solche Ansichten zu bekommen, können sogenannte APM (Application Performance Management Tools) wie Dynatrace, NewRelic oder AppDynamics verwendet werden, die als Gratis-Testversion zur Verfügung stehen und einfach zu installieren sind.

Ähnlich wie bei Schritt 1 gibt es auch hier wichtige Metriken, die je funktionalem Testfall genutzt werden sollten. Führt die Anwendung, wie im oben gezeigten Beispiel für eine einfache Suchanfrage, 171 Datenbankabfragen aus, welche durch 33 einzelne Webservice-Aufrufe getätigt werden, dann ist es wohl einleuchtend, dass diese Architektur unter realer Last schwer ins Schwitzen geraten wird. Der Tester steht hier in der Verantwortung, diese Metriken an die Entwicklung zu geben, sobald sie durch die Tests aufgedeckt werden. Das ist ein konstruktiver Teambeitrag, welcher die Gesamtqualität der Software verbessern wird.

■ Schritt 3: End-to-End-Plausibilitätsüberprüfungen ausführen

Bei einem manuellen oder automatisierten Test wird normalerweise nicht nur eine einzelne URL getestet, sondern es werden auch typischerweise mehrere Schritte innerhalb eines Tests durchgeführt. Es ist zu empfehlen, dass eine End-to-End-Plausibilitätsprüfung für jeden dieser Schritte durchgeführt wird und nicht nur für diejenigen, die fehlerhaft oder langsam sind. Warum? Weil sehr oft jene Schritte, die zu einem Problem führen, die eigentliche Fehlerursache sind. Hier zwei Beispiele:

- Ein Login hat die falsche Nutzerrolle zugewiesen und daher funktionieren viele Calls nicht mehr, weil sie als nicht autorisiert erkannt werden. Wenn aber bereits nachvollzogen werden kann, was beim Login passiert ist, wird die aktuelle Fehlerursache gefunden.
- Warenkorbaktualisierung: Wird etwas in den Warenkorb gelegt oder wieder entfernt, wird die Statusinformation des Session-Objektes des Nutzers nicht aktualisiert. Dies resultiert zunächst einmal daraus, dass die Summe des Warenkorbs falsch angezeigt wird. Da diese Speicherobjekte nicht freigegeben werden, kann dies zu einer überproportionalen Nutzung des Arbeitsspeichers führen. Hier ist es hilfreich und wichtig zu wissen, was genau bei diesen scheinbar korrekten Warenkorb-Transaktionen passiert, um die Fehlerursache zu erkennen und das Fehlverhalten von nachgelagerten Transaktionsschritten zu verstehen.

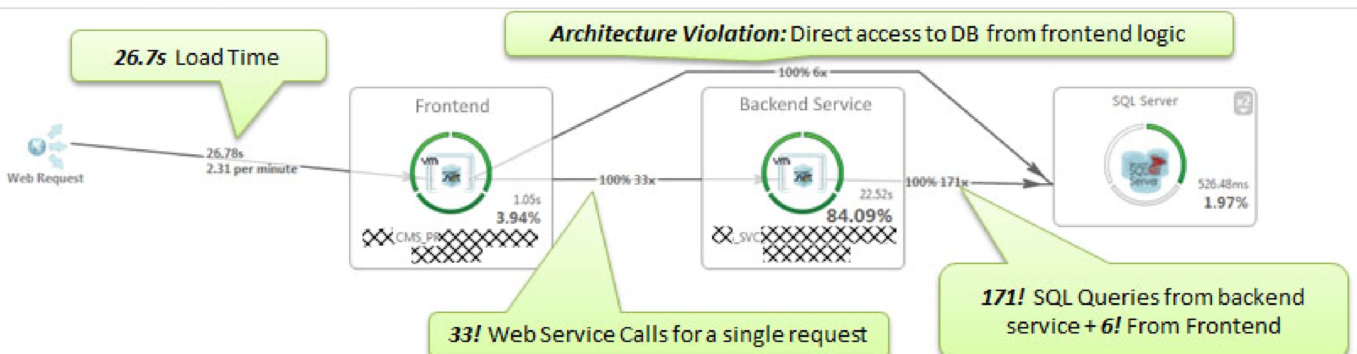

Abb. 3: Einfach zu lesende und zu verstehende Architektur-Metriken eines funktionalen Tests – 171 SQL- und 33 Webservice-Aufrufe sind klar zu erkennen

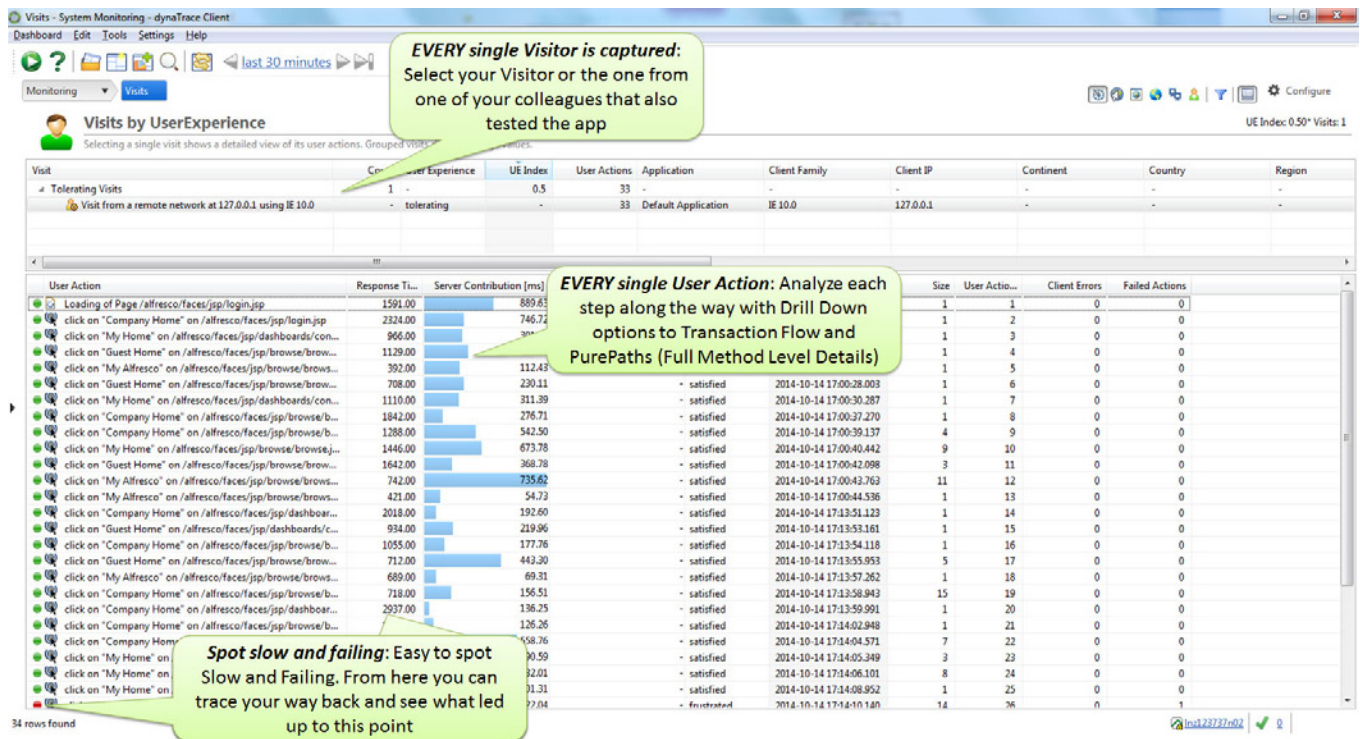

Abb. 4: Das Festhalten aller Visits und Nutzeraktionen ist eine einzigartige Fähigkeit und macht unser Leben als Tester leichter, da mit einem Drilldown von den high-level Ansichten zu den technischen Details gelangt wird

Wird ein Werkzeug verwendet, das während des Testens jede einzelne Interaktion aufzeichnet, dann kann der gesamte Visit samt allen Nutzeraktionen nachvollzogen werden. Auf diese Weise kann jeder Klick End-to-End verfolgt und Plausibilitätsprüfungen können anhand der zuvor genannten Metriken, wie Anzahl der allokierten Objekte, SQL Roundtrips, Warenkorbsummen usw., durchgeführt werden. Der Screenshot in Abbildung 4 zeigt einen Dynatrace Visit und beinhaltet alle aufgezeichneten Aktionen, welche der Nutzer gesetzt hat. Für jede Nutzeraktion zeichnet Dynatrace alle Details End-to-End auf und stellt auch den zuvor genannten Transaction Flow dar.

Für Tester ist es so viel leichter, gefundene Probleme dem Entwickler näher zu bringen, da jeder einzelne Schritt „protokolliert" ist. Die Ausrede: „Es funktioniert auf meiner Maschine" ist damit entkräftet, da nicht nur die Schritte, sondern auch technische Metriken, die das Problem selbst beschreiben, aufgezeichnet wurden.

Wiederum geht es hier um die Nutzung moderner APM-Tools, welche diese Art der Datenaufzeichnung automatisiert zur Verfügung stellen.

Schritt 4: Teilen, zusammenarbeiten und lernen

Wie bereits in den vorherigen Schritt erwähnt, ist es wichtig, dass Tester diese Erkenntnisse nicht für sich behalten, sondern die Daten mit den Kollegen beim nächsten Stand-Up oder Sprint-Review-Meeting teilen. Für den Fall, dass die Entwickler die Tester mit den falschen Schlussfolgerungen konfrontieren, sollte nicht der Mut verloren werden. Vielmehr sollte versucht werden, eine gemeinsame Erklärung zu finden. Dies hilft beiden Seiten, die Problemmuster zu verstehen, Ursachen schneller zu finden und – noch wichtiger – es hilft Entwicklern, Fehler präventiv zu vermeiden.

Mit Dynatrace besteht die Möglichkeit, alle aufgezeichneten Daten in eine einzige Datei zu packen. Diese Dynatrace Session (.dts) wird häufig von Nutzern einfach an ein Support-Ticket angehängt. Dies ist sehr viel einfacher, als zahlreiche Screenshots für den Testfall zu erstellen und anhand dieser das Problem zu beschreiben. Mit den während der Ausführung von Dynatrace gesammelten Fakten ist dies unnötig und es wird enorm Zeit gespart.

Fazit

Es ist Zeit, den Schritt in die nächste Evolutionsstufe zu wagen. Der Tester ist der „Gatekeeper" der Qualität der Software, die schlussendlich beim Endanwender zu Freud oder Leid führt. Viel wichtiger, als noch mehr Probleme zu finden, ist es doch, mit der eigenen Erfahrung seinen Entwickler-Kollegen eine qualitativ hochwertigere Entwicklung zu ermöglichen. Wie geht das? Indem ein ständiger Erfahrungsaustausch stattfindet und mit denselben Werkzeugen versucht wird, Probleme frühzeitiger zu finden und gleichzeitig an der Problemlösung zusammenzuarbeiten.