Michael Mahlberg
(mm@consulting-guild.de)
ist Berater und Geschäftsführer bei der Consulting Guild GmbH in Köln. Abgesehen davon hilft er Unternehmen seit dem letzten Jahrtausend als beratender Methoden-Coach und Softwarearchitekt dabei, die Entwicklung und Nutzung von softwarebasierten Produkten zu verbessern. Seit 1985 ununterbrochen selbstständig, hat er viele Technologien kommen und gehen sehen, aber noch kein Jahr erlebt, in dem Architekturen und Entwicklungsprozesse nicht zentral in seiner Tätigkeit gewesen wären.

Heute noch wie gestern testen?

Tests in produktionsnahen Umgebungen sind nur schwer zu vermeiden, aber bisher waren sie oft aufwendig und fehleranfällig – nicht zuletzt aufgrund der großen Datenmengen, die oft für realistische Szenarien in definierte Form gebracht werden müssen. Mit der Verkettung von Images in Containern bietet Docker – ein derzeit sehr populärer, leichtgewichtiger Ansatz zur Virtualisierung einzelner Bestandteile von Rechnersystemen – zumindest theoretisch interessante Alternativen. Im nachfolgenden Beitrag wird beleuchtet, wann und warum es sich lohnen kann, in diese Alternativen zu investieren.

In den Bereichen Spezifikation, Test und Verifikation hat sich in den letzten Jahren viel getan. Aber wo ist der echte Innovationssprung seit SUnit, dem in und für Smalltalk geschriebenen Urvater der xUnit-Frameworks von JUnit bis Nunit der 80er-Jahre? Viele der Innovationen waren eher schleichend und sind evolutionär aus vorhandenen Dingen entstanden. Entwicklungen wie Mocking-Frameworks oder die Erweiterungen in Richtung eines verhaltensgetriebenen Softwareentwurfs (*Behaviour Driven Design, BDD*) sind Beispiele für diesen Trend.

Aber auch das Testen kann von den Ideen hinter Werkzeugen wie dem Virtualisierungsansatz *Docker* oder dem Dateisystem *btrfs* profitieren. Gerade, wenn es um Tests in produktionsnahen Umgebungen geht, lassen sich hier unter den richtigen Umständen Verbesserungen um mehrere Größenordnungen bei Geschwindigkeit und übertragenen Datenmengen erreichen.

Testen mit echten Daten?

Dateisysteme und automatisierte Tests in einem Satz zu nennen, hat vor wenigen Jahren noch hochgehobene Augenbrauen unter TDD-Anhängern hervorgerufen und tut das – meiner Ansicht nach zu Recht – in vielen Fällen auch heute noch. Aber spätestens seit David Heinemeier Hansson, der Initiator von Ruby on Rails, 2014 mit seiner Diskussion über den „Tod von TDD“ (*“TDD is dead. Long live testing.“*) das Vorhandensein und die Notwendigkeit von anderen als reinen Unit-Tests wieder einer breiteren Masse bewusst gemacht hat, scheinen Integrations- und Abnahmetests auch unter Softwarearchitekten und -entwicklern wieder erheblich sichtbarer zu sein.

Und gerade bei Integrations- und Abnahmetests, wie auch im Umfeld der akzeptanztestgetriebenen Softwareentwicklung (*ATDD, Acceptance test-driven development*), werden Dateisystemzustände auf einmal wichtig. Irgendwo zwischen der Verifikation kleiner Einheiten durch Unit-Tests und dem Betrieb des Systems in der realen Wirkumgebung scheint es angebracht, auszuprobieren, was im Wechselspiel der einzelnen Elemente passiert. Und dabei wird es auch irgendwann notwendig, „echte“ Persistenz und damit Dateisystemzugriffe zu testen – häufig im Zusammenhang mit Datenbanken.

Nicht alles ist linear

Da Algorithmen, je nach Datenvolumen, mitunter sehr unterschiedliche Laufzeitverhalten an den Tag legen, wird es letztendlich unverzichtbar, mit realistischen Datenvolumina zu testen. Schließlich möchte man ja merken, ob sich der Algorithmus, dem man einen sanft ansteigenden Ressourcenbedarf zugeschrieben hat, nicht auf einmal, wenn z. B. die Datenbasis die Größe des Hauptspeichers überschreitet, zu einem extrem ineffizienten Algorithmus wird.

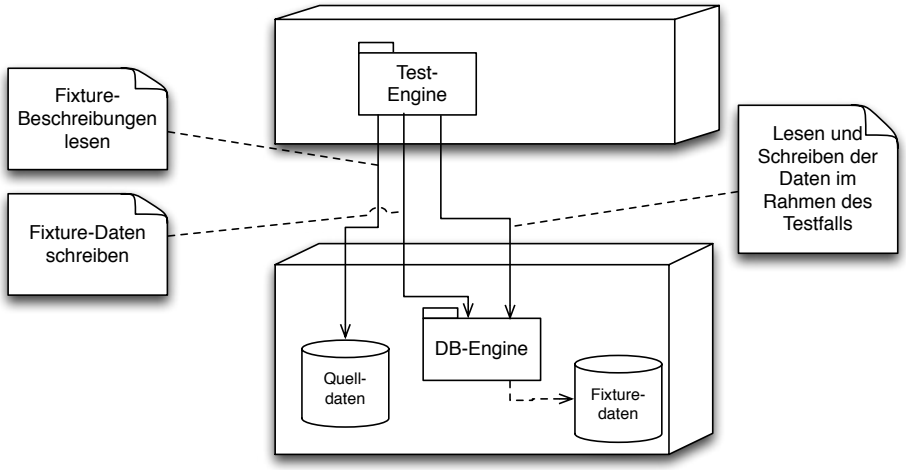

Abb. 1: Notwendige Operationen bei herkömmlicher Bereitstellung der Testdaten

Wie zum Beispiel in jenem nicht näher genannten J2EE-Projekt aus grauer Vergangenheit, bei dem sich herausstellte, dass aus einem angenommenen logarithmischen Laufzeitverhalten, also O(log n), bei mehr als 200 Kundendatensätzen leider O(2n) wurde, also ein quadratisch ansteigendes Laufzeitverhalten. Da deutlich mehr als 200 Kunden angestrebt wurden, war hier noch einmal deutliche Nacharbeit notwendig.

Aber zurück zu den Problemen bei derartigen Tests. In Wasserfall-Projekten, in denen die Testphase am Ende liegt, ist die Fragestellung überschaubar, denn hier kann man den Datenbestand anhand der hoffentlich in der Requirements-Analyse erhobenen Mengengerüste aufbauen und die ebenfalls in der Analysephase erhobenen Szenarien dagegen validieren. Aber wir schreiben das Jahr 2015 und die Zeit der reinen Wasserfall-Modelle, die nach den 1980er-Jahren das iterativ-inkrementelle Vorgehen für einige Zeit komplett zu verdrängen drohten, liegt hinter uns.

Heutzutage werden wieder mehr Projekte inkrementell und iterativ durchgeführt und die permanente Validierung der Annahmen ist zentraler Bestandteil dieses Vorgehens. Also werden auch Mechanismen benötigt, solche Massentests mit geringem Overhead und dennoch reproduzierbar, nachvollziehbar und realistisch durchzuführen.

Wo ist denn das Problem?

Abgesehen von der Schwierigkeit, große Testdatenbestände mit realistischen Inhalten herzustellen, stellt sich die Herausforderung, diese Datensätze vor den Tests oder Verifikationen in einen definierten Ausgangszustand zu versetzen. Würden wir für die Verifikationen und Tests in diesem Umfang die gleichen Verfahren anwenden, wie wir sie oft für Unit-Tests einsetzen, müssten die Daten im Rahmen der Testvorbereitung *jedes einzelnen Tests* erzeugt werden.

Das ist bei drei Datensätzen sinnvoll, bei einem halben Terrabyte Musikdaten weder sinnvoll noch praktikabel. Und hier reden wir nicht mehr darüber, ob es schneller ist, die Datensätze per SQL in die Datenbank zu bekommen oder die darunter liegenden Dateisysteminhalte zu kopieren – der reine Transfer der Daten kostet schon so viel Zeit, dass Tests mit realistischen Datenmengen nur in niedriger Frequenz sinnvoll durchführbar sind.

Meistens noch mit dem zusätzlichen Nachteil, dass die Tests nicht so stark entkoppelt werden, wie dies wünschenswert wäre. Stattdessen wird in diesen Fällen in einigen Tests davon ausgegangen, dass andere Tests den Datenhaushalt in einem bestimmten Zustand hinterlassen haben und auch tatsächlich vor den jeweiligen Tests ausgeführt wurden.

In solchen Szenarien führt dann eine Änderung der Testreihenfolge zu einer Kaskade von fehlschlagenden Verifikationen und damit zu einer weiteren Erstarrung der Testlandschaft – oder dazu, dass fehlschlagende Tests einfach auskommentiert werden. Aber können diese Laufzeiten und Abhängigkeiten überhaupt vermieden werden?

Der lange Weg

Wenn 1,5 Terrabytes in der Datenbank sein sollen, müssen sie irgendwie hineinkommen. Zumindest über den Datenbus zur Platte müssen sie. Werden sie algorithmisch erzeugt, so bedeutet das auf jeden Fall das Schreiben von mehr als 1,5 Milliarden Bytes. Werden Ladebestände verwendet, zum Beispiel die anonymisierten Echtdaten der Vorversion der Software, so kommen noch mal mindestens 1,5 Milliarden Bytes dazu, die gelesen werden müssen.

Meistens sogar noch erheblich mehr, denn ein Datensatz der netto 5 Bytes enthält, besteht in den häufig verwendeten SQL-Ladeskripten zusätzlich noch aus solchen Worten wie INSERT und VALUES – ganz zu schweigen davon, dass der in der Datenbank mit zwei Bytes in einem Integer-Feld abgelegte Wert 12345 als Text mindesten 5 Bytes umfasst. In der Form „INSERT INTO T1 (F1) VALUES (12345);“ werden die 2 Bytes netto so schnell zu 35 Bytes brutto.

Und einige Ansätze exportieren die Dateninhalte in genau dieser Form, was bei geringem Datenumfang durchaus seine Vorteile hat – bei Millionen von Datensätzen aber zu vielen Problem führen kann. Der Ablauf für eine kontrollierte Bestückung der Datenbasis mit „Real-Daten“ als Testbasis (*Fixture*) besteht zwar je nach Szenario aus unterschiedlichen Schritten, aber im Wesentlichen bleibt es dabei, dass die Daten mehrfach in beide Richtungen transportiert werden müssen (siehe auch Abbildung 1).

Gestapelte Daten

Für diese Szenarien kann der Einsatz von Docker zu Änderungen im Bereich mehrerer Größenordnungen, also um das Zehn- oder Hundertfache, führen. Der Grundgedanke ist einfach, verbindet aber – was zu kontroversen Diskussionen führt – Testausführung und Infrastruktur eng.

Der Virtualisierungsansatz Docker bietet die Möglichkeit, einzelne sogenannte Images zu verketten. Dabei wird der jeweils obersten Zugriffsschicht nur vorgespiegelt, dass Änderungen an Daten tatsächlich zu Änderungen auf der Platte führen. Tatsächlich werden nur die Unterschiede zwischen dem aktuellen und dem verketteten Image abgespeichert – will man den Originalzustand wiederherstellen, reicht es aus, eine neue Instanz des obersten Images zu erzeugen (siehe auch Abbildung 2).

Für die Gestaltung von Tests mit Massendaten bedeutet das, dass es für die Vorbereitung der Fixture nicht mehr notwendig ist, Terrabytes von Daten zu trans-

portieren. Vielmehr reicht es aus, die Fixture-Daten in einem eigenen Docker-Image sauber zu verwalten. Danach kann man pro Testlauf das „oberste“ Image in dem Docker-Container, in dem die Testdaten verwaltet werden, durch eine leere Instanz ersetzen oder sogar eine von mehreren Instanzen auswählen, um bestimmte Datenkonstellationen zur Verfügung zu stellen.

Das Starten und Stoppen von Docker-Containern ist sehr leichtgewichtig und da der reine Netto-Datenverkehr im herkömmlichen Verfahren, ohne Steuerungsinformationen und Ähnliches zu betrachten, für den Transfer eines Terrabytes über eine 300 Mbit-Strecke ca. 60 Minuten beträgt, hätte man mit der Docker-Variante in diesem Szenario 6 Minuten Zeit für den Image-Austausch, um eine Größenordnung (10x) zu erreichen und immer noch 30 Sekunden, um einen Geschwindigkeitsgewinn von zwei Größenordnungen (100x) zu erzielen.

Gebotene Vorsicht

Das hier geschilderte Verfahren wird bereits an einigen Stellen eingesetzt, aber es gibt noch keine Referenzimplementierung mit unabhängigen Testberichten und keine Musterlösungen. Obwohl die Möglichkeiten einer solchen Docker-basierten Lösung sehr wichtige Probleme des Testens von Systemen mit großem Datenbedarf adressieren, sind noch längst nicht alle Fragen geklärt.

Insbesondere die Laufzeit-Aufschläge, die gestaffelte Images und Container mit sich bringen können, sind hier in jedem Einzelfall abzuwägen. Mein geschätzter Kollege Gaylord Aulke schlug vor, komplett auf die Docker-Schicht zu verzichten und statt der Container-Verkettung direkt mit btrfs (*B-tree file system*) zu arbeiten, ein Design-Ansatz, der auf jeden Fall noch weiter untersucht werden sollte!

Und nun?

Lohnt es sich, dieses Testverfahren einzusetzen? Die Antwort darauf kann nur ein klassisches „Das kommt darauf an“ sein. Da es noch keine Standardimplementierungen oder Frameworks für diese Aufgabenstellung gibt, ist hier auf jeden Fall viel eigene Arbeit und eigene Verantwortung gefragt.

Wenn das Umfeld aber von produktionsnäheren Tests mit größeren Datenmengen profitieren würde, dann ist das Vorgehen mit geschachtelten Images ein

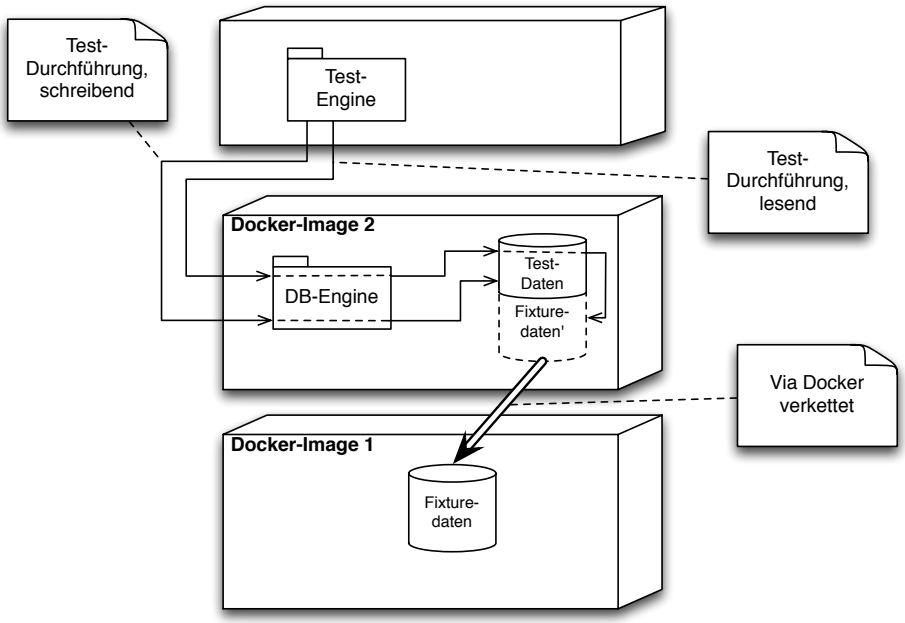

Abb. 2: Testablauf bei der Verwendung von Docker-Mechanismen

Verifikation vs. Test

In diesem Artikel wird ganz bewusst manchmal von Tests und manchmal von Verifikationen gesprochen. Ein Test, so wie er beispielsweise von Materialprüfern oder Testingenieuren verstanden wird, hinterlässt das Testobjekt normalerweise in einem mehr oder weniger stark beschädigten Zustand. Hier wird nicht überprüft, ob das Testobjekt sich innerhalb der Spezifikationen befindet, sondern, was mit ihm *außerhalb* der Spezifikationen passiert. Wenn wir Autos zum TÜV bringen, wird dort nur geprüft, ob sie den Spezifikationen entsprechen. Deshalb heißt die entsprechende Untersuchung auch Haupt*untersuchung*. Tester würden wahrscheinlich eher prüfen, bei wie viel Tonnen Dachlast zum Beispiel die B-Säule tatsächlich einknickt – ein Test, der wenig Sinn ergibt, wenn wir mit dem Auto hinterher noch nach Hause fahren wollen.

In der Softwareentwicklung ist die Situation ähnlich. Wie Edsger W. Dijkstra schon 1969 (!) schrieb, lässt sich durch Tests nur die Anwesenheit, nicht die Abwesenheit von Fehlern beweisen. Unit-*Tests* leisten, wenn die Software fertig ist, noch nicht mal das – sie beweisen „nur“, dass sich die Units noch so verhalten, wie es in den Tests spezifiziert wurde. Daher werden sie zunehmend auch nur noch als Verifikationen oder ausführbare Spezifikation bezeichnet.
Testen, wie es beispielsweise im explorativen Testen (*exploratory testing*) verstanden wird, versucht, die Dinge zu finden, an die während der Spezifikation und Entwicklung nicht gedacht wurde.

Die unter Testern von Anwendungen für mobile Geräte verbreitete Mnemonic “*I SLICED UP FUN*“, bei der jeder Buchstabe für eine Variante steht, einen unerwarteten Fehler in der Anwendung zu Tage zu bringen, zeigt die Unterschiede der Geisteshaltung echter Tester und reiner Verifikation sehr deutlich. Das erste I steht zum Beispiel für Inputs und alleine hier werden mehr als ein halbes Dutzend Möglichkeiten geschildert, die Anwendung aus dem Tritt zu bringen. Die bei solchen Tests gefundenen Fehler sollten natürlich wieder als automatisierte Verifikationen in den Spezifikations- und Entwicklungszyklus zurückfließen. Wo sie dann wahrscheinlich aufgrund des allgemeinen Sprachgebrauches wieder als Tests bezeichnet werden.

Ansatz, der zumindest die Datenbereitstellung in erheblichem Maße beschleunigen kann. Und wer weiß – vielleicht ist ja unter den Lesern dieses Artikels jemand, der das dazugehörige Open-Source-Projekt aus der Taufe hebt, innerhalb dessen die Standardfragen, die sich stellen, beantwortet werden.

Links

http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html
http://testobsessed.com/2008/12/acceptance-test-driven-development-atdd-an-overview/
https://btrfs.wiki.kernel.org/index.php/Main_Page
http://www.satisfice.com/articles/what_is_et.shtml
http://www.kohl.ca/2010/test-mobile-apps-with-i-sliced-up-fun/