Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen

Continuous Deployment: Keine Angst vor Releases

Florian Tobusch – Sommersemester 2018

Inhaltsverzeichnis

Erklärung zur selbstständigen Verfassung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt, sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Ort, Datum

Unterschrift

1. Continuous Deployment – Wofür ist das gut?

1.1 Continuous Integration, Continuous Delivery, Continuous Deployment

Continuous Integration (CI) sollte in der heutigen Zeit der Standard in der Softwareentwicklung sein. Bei CI arbeiten die Entwickler in einem gemeinsamen Repository, dass sich unter Versionskontrolle befindet und auf einem CI-Server automatisch kompiliert, gebaut und getestet wird.
Mit Continuous Deployment wird nun der gesamte Prozess vom Commit bis zum Deployment in Produktion automatisiert. Jede Änderung im Code geht automatisch in Produktion, sobald sie alle Testphasen erfolgreich bestanden hat.
Continuous Delivery ist die »Vorstufe« von Continuous Deployment. Hier können die Entwickler per Knopfdruck selbst den Zeitpunkt bestimmen, wann eine neue Version in die Produktivumgebung deployt wird.
Zusammenfassend gesagt ist Continuous Integration die Voraussetzung für Continuous Delivery. Continuous Delivery ist wiederum die Voraussetzung für Continuous Deployment.

1.2 Einschneidende Veränderungen bei großen Technologie Firmen durch CI und CD

Auch die großen Technologiefirmen wie Google, Facebook oder Linkedin haben mal klein angefangen und ohne Continuous Delivery gearbeitet. An Continuous Deployment war zu diesem Zeitpunkt erst recht noch nicht zu denken. Während des rasanten Wachstums der Unternehmen gab es überall einen Punkt, ab dem es ohne Automatisierung des Deploymentprozesses nicht weiter gehen konnte.

Als Mitarbeiter des Google-Web-Server-(GWS)-Teams beschreibt Mike Bland den dortigen Zustand im Jahre 2005. Die Builds dauerten viel zu lange, die Tests dauerten ewig und teilweise wurde Code ungetestet in Produktion deployt. Die Entwickler checkten große Mengen Code in langen und unregelmäßigen Abständen ein. Dies führte häufig zu Konflikten. Bharat Mediratta, der Leiter des GWS-Teams, wollte das Problem durch automatische Tests lösen. Jeder Änderungen musste ab sofort durch automatische Tests überprüft werden (»No changes without tests«). In den nächsten Jahren entstand eine Suite aus tausenden automatisierten Tests.[1]
Durch die automatisierten Tests, Continuous Integration und weitere innovative Maßnahmen gelang es Google 2013, dass gleichzeitig 15.000 Entwickler an 4.000 Projekten arbeiteten. Der gesamte Code ist in nur einem Repository und wird alle paar Stunden neu gebaut.[2]

Google konnte vor allem durch Continuous Integration die Produktivität verbessern. Im nächsten Beispiel wird die Verbesserung durch Continuous Delivery aufgezeigt. CSG International, ein amerikanisches Unternehmen für Rechnungsdruck hat Probleme mit den halbjährlichen Releases. Die Testumgebungen entsprechen nicht der Produktionsumgebung, was zu Fehlern führt, die erst in Produktion erkannt werden.

1 Mike Bland (2015). Pain Is Over, If You Want It. DevOps Enterprise Summit 2015. *https://de.slideshare.net/ITRevolution/does15-mike-bland-pain-is-over-if-you-want-it-55236521* [20.05.2018]

2 Eran Messeri (2013). »What Goes Wrong When Thousands of Engineers Share the Same Continuous Build?«. GOTO Konferenz, Dänemark, 2013. Notizen zum Vortrag veröffentlicht von Gene Kim: *http://scribes.tweetscriber.com/realgenekim/206* [20.05.2018]

Durch die Einführung von täglichen Deployments und der automatisierten Erstellung von Umgebungen können nun die Probleme früher erkannt werden. Das Deployment wird dadurch zur Routine und das eigentliche Release in Produktion ist nichts besonderes mehr. Die Vorfälle in der Produktionsumgebung konnten dadurch innerhalb von zwei Jahren um 90% verringert werden.[3]

1.3 Vorteile

1.3.1 Kurze Durchlaufzeiten

Bevor es Continuous Delivery und Continuous Deployment gab, scheute man sich oft davor ein neues Release in Produktion zu deployen, denn der Deploymentprozess erforderte viele manuelle Schritte. Durch die Automatisierung wird den Entwicklern manuelle Arbeit abgenommen, wodurch die Zeit vom Commit bis zum Deployment deutlich reduziert werden kann.
Kürzere Durchlaufzeiten ermöglichen häufigere Releases, was dem Unternehmen neue Möglichkeiten bei der Vermarktung der Software bietet und die Time-To-Market verbessert. Time-To-Market beschreibt die Dauer, wie lange ein neues Produkt von der Entwicklung bis zur Platzierung am Markt benötigt. Gerade in der heutigen Zeit ist es wichtig schnell auf Veränderungen am Markt reagieren zu können. Mit Continuous Deployment können neue Features schnell auf dem Markt positioniert werden und bei Misserfolg auch schnell wieder vom Markt genommen werden.[4]

1.3.2 Risiko reduzieren und Qualität erhöhen

Durch die Automatisierung, die Continuous Deployment erfordert, entsteht eine Reduzierung des Risikos. Automatisierte Prozesse erhöhen die Qualität der Software, da die Prozesse für alle reproduzierbar und nachvollziehbar werden. Egal welcher Entwickler den Deploymentprozess anstößt, das Ergebnis ist bei jedem das gleiche. Vor dem eigentlichen Release wurde mit dem selben automatisierten Prozess bereits mehrmals erfolgreich in die Testumgebungen deployt. Fehler in der Produktivumgebung werden dadurch immer unwahrscheinlicher und seltener.
Eine starke Automatisierung ermöglicht den Entwicklern die Häufigkeit der Deployments in Produktion zu erhöhen. Der große Vorteil hierbei ist, dass der Umfang pro Deployment reduziert werden kann. Bei häufigen Deplyoments sind die Änderungen klein, überschaubar und nachvollziehbar, womit sich das Risiko pro Deployment reduziert.[5]

1.3.3 Schnelles Feedback für die Entwickler

In den vorherigen Abschnitten haben wir bereits gesehen, dass der automatisierte Deployment-Prozess nicht nur für die Produktivumgebung, sondern auch zum deployen in Testumgebungen genutzt wird. In den Testumgebungen können unter produktionsnahen Bedingungen automatische Tests, wie z.B. Akzeptanz- oder Kapazitätstests ausgeführt werden.
Da mit CD neuer Code von Entwicklern nun viel öfters diverse Testumgebungen durchläuft, erhalten die Entwickler zeitnah Feedback, wie sich ihr neues Feature unter produktionsnahen Bedingungen verhält. Sie müssen sich nicht mehr wie früher erst wieder in den Code hinein denken, weil die Tests nur einmal im Monat oder vierteljährlich ausgeführt wurden.[6]

3 Scott Prugh & Erica Morrison (2015). Conway & Taylor Meet the Strangler (v2.0). DevOps Enterprise Summit 2015. *https://de.slideshare.net/ITRevolution/scott-prugh-scott-prughericamorrisonconwaytaylorstrangler-55018581* [20.05.2018]
4 [CDPE], S. 17-20
5 [CDPE], S. 20-23
6 [CDPE], S. 23f

1.3.4 Mehr Verantwortung für die Teams und Entwickler

Mit Continuous Deployment kann jede Änderung der Entwickler innerhalb kürzester Zeit automatisch über die Deployment-Pipeline in Produktion gelangen. Damit das Team keine fehlerhafte Software ausliefert, werden sie bemüht sein stetig die Pipeline zu verbessern und durch neue automatische Tests zu erweitern. So steigt die Qualität der Deployment-Pipeline kontinuierlich.[7]

2. Deployment Pipeline

Farley und Humble beschreiben »die Deployment Pipeline [als] eine automatisierte Manifestierung des Prozesses um Software von der Versionskontrolle in die Hände des Users zu bringen«[8]. Nach dem Einchecken von neuem Code wird die Software neu gebaut. Der Build durchläuft danach verschiedene Testphasen, bis er am Ende in Produktion deployt wird. Die Anordnung der Phasen (=Stages) sollte so gewählt werden, dass Fehler so früh wie möglich erkannt werden. Kann der Build eine Stage nicht erfolgreich passieren, wird er nicht zur nächsten Stage weitergegeben, sondern die Entwickler erhalten sofort Feedback über den fehlerhaften Build. Alle Builds müssen, um in Produktion zu gelangen, diese Pipeline durchlaufen.

Bei der Entwicklung der Pipeline sind folgende Dinge zu beachten[9]:

- Für die Software werden nur einmal am Anfang die Binaries erstellt. Diese durchlaufen danach alle Phasen der Pipeline. Die Binaries für die Tests werden auch in Produktion deployt.
- Über die gesamte Pipeline muss neben den selben Binaries auch für jede Phase der gleiche Deployment Prozess verwendet werden. Mit dem selben Prozess, wie bereits erfolgreich in den Testphasen deployt wurde, wird auch in Produktion deployt.
- Die Testumgebungen sollten nahezu der Produktionsumgebung entsprechen. Bei schlechterer Test-Hardware müssen eben die Anforderung der Tests an die Umgebung angepasst werden, um ein aussagekräftiges Ergebnis zu erhalten.

Für die Umsetzung einer Deployment Pipeline gibt es einige Tools zur Unterstützung. Die Tools steuern den Durchlauf durch die Pipeline und stellen die Ergebnisse für alle sichtbar dar. Bekannte Tools sind beispielsweise Jenkins, Bamboo oder TravisCI. Im folgenden werden nun die einzelnen Phasen der Pipeline genauer vorgestellt.

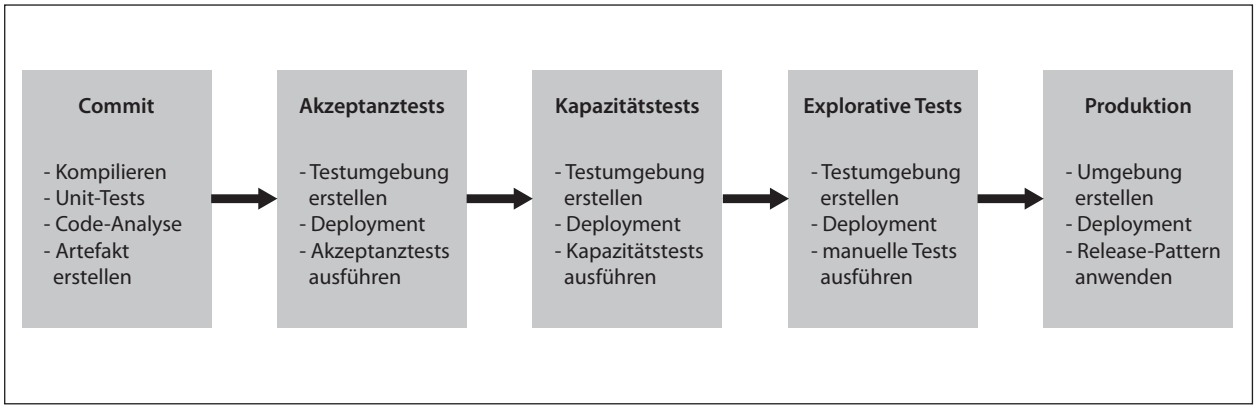

Abbildung 1: Deployment-Pipeline mit den einzelnen Phasen

7 [CDPE], S. 189f

8 [CD], S. 106

9 Jez Humble. Implementing continuous delivery. *https://continuousdelivery.com/implementing/patterns/* [21.05.2018]

2.1 Commit Stage

Ein Entwickler hat eine Änderung im Code vorgenommen und checkt diese nun in das Versionskontrollsystem ein. Das Einchecken in den Master ist der Trigger für den CI-Server. Er erstellt eine neue Instanz der Deployment-Pipeline und führt die Tasks der Commit Stage aus. Die Dauer der Commit Stage sollte so gewählt sein, dass der Entwickler darauf warten kann. Idealerweise laufen die Tasks parallel auf einem Build Grid (= Ein Pool von Rechnern). Die Commit Stage sollte nicht länger als 5 Minuten dauern.
Als erstes wird in der Commit Stage der Code kompiliert und darauf eine Suite von Tests ausgeführt. Die meisten dieser Tests sind Unit-Tests. Im Laufe der Zeit werden sich in den späteren Phasen eventuell Fehler herauskristallisieren, die häufiger auftreten. Damit für fehlerhafte Builds nicht unnötig Zeit in den anderen Phasen der Pipeline verschwendet wird, sollte versucht werden diese häufigen Fehler durch schnelle Tests in der Commit Stage abzufangen.
Nachdem alle Tests erfolgreich bestanden wurden, wird noch eine Codeanylse durchgeführt. Liegen die Messungen nicht innerhalb vordefinierter Werte, erreicht der Build nicht die nächste Phase. Metriken hierfür können beispielsweise sein:

- Testabdeckung (Test Coverage)
- Duplizierter Code
- Zyklomatische Komplexität
- Zahl der Abhängigkeiten zwischen Packages und Klassen

Wenn bis hier alle Tests problemlos absolviert werden konnten, dann wird die Commitphase mit der Erstellung eines deploybaren Artefakts abgeschlossen. Dies kann z. B. eine WAR- oder EAR-Datei sein.
Wird wegen Fehler durch den neuen Code die Commit-Stage abgebrochen, so muss dem Entwickler Feedback gegeben werden. Er sollte einen Bericht erhalten, welche Tests bereits erfolgreich absolviert wurden und welche fehlgeschlagen sind, bzw. warum diese Tests nicht funktioniert haben. Auch bei erfolgreicher Auswertung sollten die Ergebnisse für alle einsichtbar sein.[10]

2.1.1 Artefakte managen

Wie bereits eingangs erwähnt, durchläuft das in der Commit-Phase erstellte Artefakt (Binaries + Metadaten) alle nachfolgenden Phasen. Das Artefakt wird nicht für jede Phase neu erstellt. Um ein Artefakt allen zugänglich zu machen, gibt es ein Artefakt Repository. Nach der Commit-Phase wird das Artefakt im Artefakt Repository abgelegt, damit jede Phase der Deployment-Pipeline darauf zugreifen kann .[11, 12]

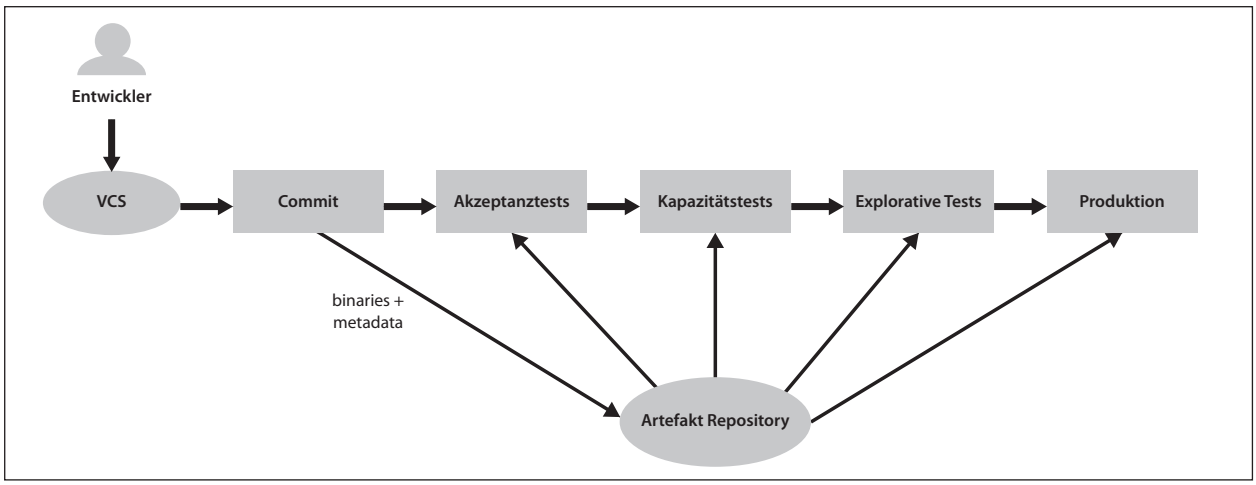

Abbildung 2: Ein Artefakt für alle Phasen der Deployment-Pipeline[13]

Für das Managen von Artefakten stehen Tools wie Nexus oder Artifactory zur Verfügung. Diese Repository Manager vereinfachen den firmenweiten Umgang mit Artefakten.

10 [CD], S. 120f
11 [CD], S. 174-177
12 [CDPE], S.123ff
13 [CD], S. 176

2.1.2 Build-Tools

Um für alle einheitliche und reproduzierbare Software-Builds zu erstellen, gibt es in der Javawelt Build-Tools wie Maven oder Gradle. Die Build-Tools enthalten, ähnlich wie die Deployment-Pipeline, verschiedene Phasen, die nach einer vordefinierten Reihenfolge ausgeführt werden. Beispiele für Phasen sind das Ausführen von Unit-Tests, das Managen von Abhängigkeiten oder das Ablegen des Artefakts in einem zentralen Repository. Mit den Build-Tools kann der gesamte »Lifecycle« vom Bauen, über Testen bis hin zum Deployen abgebildet werden.[14]

2.2 Akzeptanztests

Das folgende Beispiel aus [CD] macht klar, warum Akzeptanztests benötigt werden. Für die Entwicklung eines neuen Systems wird bereits Continuous Integration verwendet. Nach einiger Zeit der Entwicklung wurde das System erneut deployt, aber diesmal funktionierte es nicht mehr. Alle Unit-Tests wurden allerdings zuvor erfolgreich abgeschlossen und besaßen auch eine sehr gute Testabdeckung. Im Deployment-Prozess selbst ließen sich ebenfalls keine Fehler finden. Es stellte sich schließlich heraus, dass die Version von vor 3 Wochen erfolgreich deployt werden konnte. Seit dieser Version hatte sich ein Bug eingeschlichen, welcher das System nicht richtig starten ließ. Die Entwickler hatten die ganze Zeit mit Unit-Tests nur einzelne Einheiten des Systems getestet, aber nicht das System selbst.[15]
Es sollte nun klar sein, dass Unit-Tests alleine nicht ausreichen. Akzeptanztests bieten die fehlende Überprüfung der Software auf einer höheren Ebene, wie sie im vorherigen Beispiel nicht statt gefunden hat. Die Anwendung wird als ganzes getestet. »Ein einzelner Akzeptanztest ist dazu vorgesehen, zu überprüfen, ob die Akzeptanzkriterien einer Story oder eines Requirements erfüllt sind.«[16] Bei der Erstellung der Akzeptanztests sollten neben den Entwicklern auch, die Tester, sowie die Kunden beteiligt sein. Alle Beteiligten sollten die Akzeptanztests verstehen und wissen welche Kriterien überprüft werden. Am Ende dienen die Tests schließlich dem Kunden als Basis zur Abnahme der Software. Zusätzlich wird mit diesen Tests gewährleistet, dass auch bereits funktionierende Teile der Software, trotz der Neuerungen weiterhin einwandfrei funktionieren. Dabei spricht man von Regressiontests.

2.2.1 Akzeptanztests erstellen

Die Kriterien für die Akzeptanztests sind schnell definiert. Das Schreiben des eigentlichen Tests ist allerdings etwas schwieriger, wie es auf den ersten Blick scheint. Denn es sollte darauf geachtet werden, Akzeptanztests nicht zu sehr von einer konkreten Implementierung abhängig zu machen. Sie sollten »in der Sprache des Business und nicht in der Sprache der Technologie der Applikation«[17] verfasst werden. Um eine Unabhängigkeit der Tests von konkreten Implementierungen und GUI-Versionen zu garantieren wird eine zusätzliche Schicht eingeführt. Bei Änderungen an der GUI müssen nicht alle Akzeptanztests angepasst werden, sondern nur die darunter liegende Schicht.[18]

14 [CDPE], S. 88-99
15 Vgl. [CD], S. 123
16 [CD], S. 188
17 [CD], S. 125
18 [CD], S. 190ff

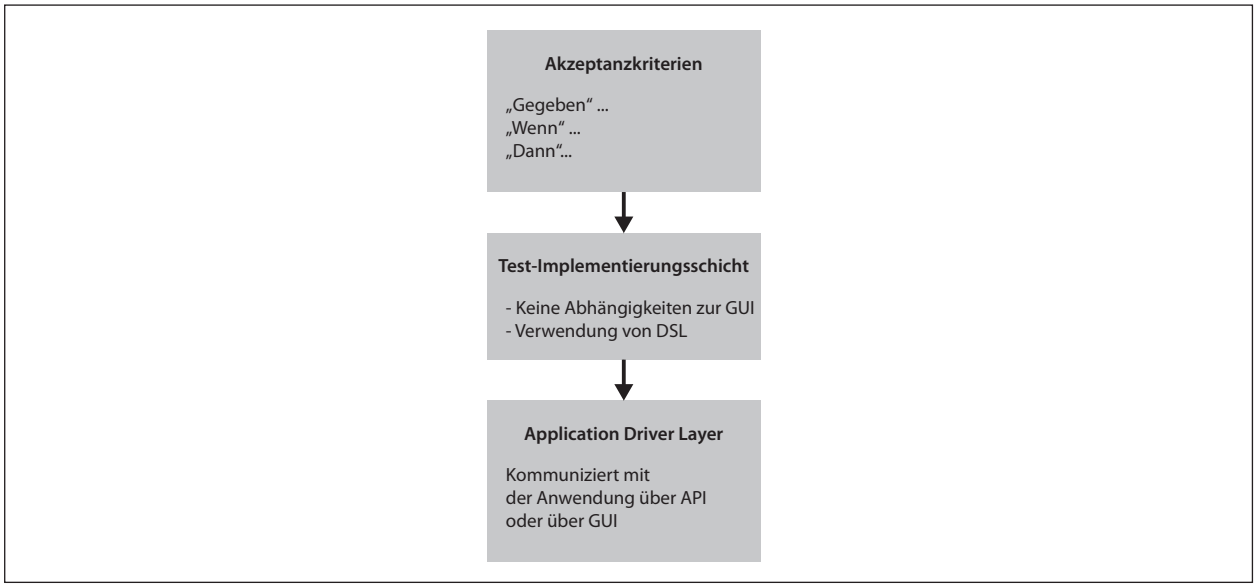

Abbildung 3: Die unterschiedlichen Schichten der Akzeptanztests mit Behavior Driven Development[19]

Für eine einfache und verständliche Definition der Akzeptanztests wäre es von Vorteil die Tests textuell zu beschreiben, da schließlich neben den Entwicklern auch die Kunden ein Verständnis dieser Tests haben sollten. Behavior Driven Development (BDD) verfolgt genau diesen Ansatz. Die Tests werden formalisiert, sodass sie automatisiert mit Tools wie JBehave oder Cucumber ausgeführt werden können.
Die Akzeptanzkriterien können über eine externe domänenspezifische Sprache (DSL) beschrieben werden. Ein Test wird über die drei Grundbestandteile »Gegeben«, »Wenn« und »Dann« wie folgt definiert:

```
Szenario: Kunde registriert sich erfolgreich
Gegeben ein neuer Kunde mit
E-Mail eberhard.wolff@gmail.com
Vorname Eberhard Name Wolff
Wenn der Kunde sich registriert
Dann sollte ein Kunde mit
der E-Mail eberhard.wolff@gmail.com existieren
Und es sollte kein Fehler gemeldet werden
```

Listing 1: BDD-Akzeptanztest[20]

»Gegeben« beschreibt den Zustand, bzw. Kontext, in dem der Test stattfindet. »Wenn« definiert das Ereignis, dass eintritt und mit »Dann« wird das gewünschte Ergebnis beschrieben.
Nach der Definition der obersten Schicht mit den Akzeptanzkriterien, wird noch die »Test-Implementierungsschicht« benötigt. Unter JBehave gibt es Annotationen, wie z.B. »@Given« oder »@When«, mit denen in dieser Schicht die Akzeptanzkriterien mit Code verknüpft werden können.

```
@Given(»ein neuer Kunde mit E-Mail $email Vorname $vorname Name $name«)
public void gegebenKunde(String email, String vorname, String name){
            kunde = new User(vorname, name, email);
}
```

Listing 2: Java-Methode mit JBehave-Annotation zum Ausführen des BDD-Akzeptanztests[21]

19 [CD], S. 191
20 [CDPE] Listing 5-2, S. 148
21 [CDPE] Listing 5-3, S. 148f

Die »Application Driver Layer« ist die Schicht, die schließlich weiß, wie mit der sich unter Test befindlichen Applikation kommuniziert werden kann. Diese Schicht kann die Software über eine API oder über die GUI testen. Wenn über die GUI getestet werden soll, wird nochmals eine neue Abstraktionsschicht eingezogen. Mit der neuen Schicht »Window Driver« kann somit unabhängig mit der GUI kommuniziert werden. Bei Änderungen an der GUI müssen nur die »Window Driver« angepasst werden.[22]

2.3 Kapazitätstests

Die nächste Phase nach den Akzeptanztests sind die Kapazitätstests. In der vorherigen Phase wurden die funktionalen Anforderungen ausgiebig geprüft. Jetzt werden die Nichtfunktionalen Anforderungen getestet. Was nützt es wenn das System ein korrektes Ergebnis ausgibt, aber die Verarbeitung der Anfrage viel zu lange dauert? Um bereits während des Entwicklungsprozesses spätere mögliche Performanceprobleme aufzudecken, werden die Kapazitätstests in die Deployment-Pipeline integriert. Somit können negative Veränderungen der Performance direkt auf die neuen Änderungen im Code zurückgeführt werden.
Eine große Schwierigkeit bei den Kapazitätstests ist die Erstellung einer möglich realitätsnahen Testumgebung. In Produktion gibt es riesige Datenbanken, Abhängigkeiten zu anderen System und leistungsstarke Server. Die Anforderungen an das Produktivsystem lassen sich nicht einfach linear skalieren, um diese an das Testsystem anzugleichen. Anstelle der Produktionsdaten werden nur Testdaten verwendet und das genaue Verhalten eines Benutzers im Produktivsystem lässt sich auch nur schwer vorhersagen. Die Kunst bei den Kapazitätests ist es nun ein geeignetes Modell für die Testumgebung zu finden.

2.3.1 Schwellwert für Erfolg und Misserfolg definieren

Die Messungen der Kapazitätstests liefern ein Ergebnis zurück, beispielsweise dauert die Verarbeitung einer Anfrage auf dem Server 2ms. Damit ist allerdings noch nicht definiert, ab wann der Test bestanden ist und ab wann nicht. Die Schwellwerte sollten realistisch sein. Bei einem zu aggressiven Schwellwert wird unnötige Zeit in die Optimierung der Performance gesteckt, welche im späteren Produktivsystem überhaupt nicht benötigt wird. Werden die Werte zu schwach gewählt laufen zwar die Tests problemlos, dafür treten später in Produktion Probleme auf.[23]

2.3.2 Kapazitätstests erstellen

Ähnlich wie bei den Akzeptanztests können die Kapazitätstests über eine API oder über die GUI ausgeführt werden. Tests über die GUI sind sehr anfällig gegenüber Änderungen an der grafischen Oberfläche. Allerdings entsprechen die Tests genau dem Vorgehen eines realen Benutzers. Werden die Tests über eine eigene API ausgeführt, entspricht dies nicht mehr genau dem späteren Nutzerverhalten. Testen über eine API vereinfacht die Tests und ermöglicht beispielsweise auch das einfache einspielen von Testdaten.[24]
Mit den Akzeptanztests als Grundlage können Szenarien wie »Produkte suchen«, »Bezahlen« oder »Registrierung eines neuen Kunden« abgebildet werden. Durch die Kombination mehrerer Szenarien lässt sich eine realistische Auslastung für die Kapazitätstests simulieren.[25]

2.4 Explorative Tests

Bisher ist die Software nur von automatischen Tests geprüft worden. Gerade bei sich immer wiederholenden Abläufen sollten automatische Tests den manuellen Tests vorgezogen werden. Manuelle Tests sind für häufige Wiederholungen einfach zu teuer und zu unsicher. Mit der explorativen Testphase werden nun erstmals in der Deployment-Pipeline manuelle Tests ausgeführt.

22 [CD], S. 201-204
23 [CDPE], S. 158-162
24 [CDPE], S. 163
25 [DOH], S. 131

Die manuellen Tests werden von Testern durchgeführt, die im Besitz von fachlichem Wissen über die Software sind. Sie kennen die Schwachstellen, sowie die neuen Features und können gezielt in diesen Bereichen testen. Somit können Fehler aufgedeckt werden, die von den automatischen Tests nicht gefunden wurden. Manuelle Tests ermöglichen es auch die Bedienbarkeit oder das Design der Software zu testen. Über automatische Tests lassen sich solche Anforderungen nur schwer überprüfen.
Der Grundgedanke von Continuous Deployment ist es, häufige und kleine Releasezyklen zu haben. Somit wird es schwer nach jedem Durchlauf der Pipeline auch noch manuelle Tests auszuführen. Grundsätzlich sind die explorativen Tests vor allem für neue Features gedacht. Sie sind nicht dazu gedacht die Anforderungen, die bereits von automatischen Tests getestet wurden, nochmals händisch zu überprüfen. Werden Fehler gefunden, sollte überlegt werden, wie diese mit automatischen Tests geprüft werden können. Dies bringt mehr Sicherheit und senkt die Kosten für die manuellen Tests.[26]

2.5 Produktion

Sind alle Phasen der Deployment-Pipeline erfolgreich durchlaufen, muss eigentlich »nur noch« in die Produktionsumgebung deployt werden. Leider ist dies nicht ganz so einfach. Nicht funktionierende Software in Produktion möchte niemand sehen, da dies sehr teuer werden kann. Für das sichere Deployen gibt es verschiedene Techniken, die es ermöglichen jederzeit wieder auf eine stabile Version zu wechseln. Während eine neue Version der Software in Produktion deployt wird, sollten die derzeit aktiven Nutzer nichts von diesem Vorgang mitbekommen. Die Software muss während des Deployments weiterhin wie gewohnt funktionieren.

2.5.1 Rollout, Rollback, Roll Forward

Beim Deployment in Produktion (=Rollout) ist es wichtig Prozesse zu etablieren, die ausgeführt werden können, wenn Fehler mit der neuen Softwareversion in Produktion auftreten. Bei Problemen kann ein Rollback durchgeführt werden. Damit wird wieder zurück auf die alte Version gewechselt. Anstelle eines Rollback besteht auch die Möglichkeit den Fehler sofort zu beheben und mit einem Roll Forward auf die nächste stabile Version zu schalten. Die Änderung für das Roll Forward durchläuft dabei natürlich auch wieder die gesamte Deployment-Pipeline. Roll Forwards sind von Vorteil, wenn mit dem fehlerhaften Rollout umfangreiche Änderungen an der Datenbank vorgenommen wurden. Die Datenbank kann somit erhalten bleiben. Beim Rollback müsste die Datenbank wieder auf den alten Stand zurückgesetzt werden.

2.5.2 Blue-Green-Deployment

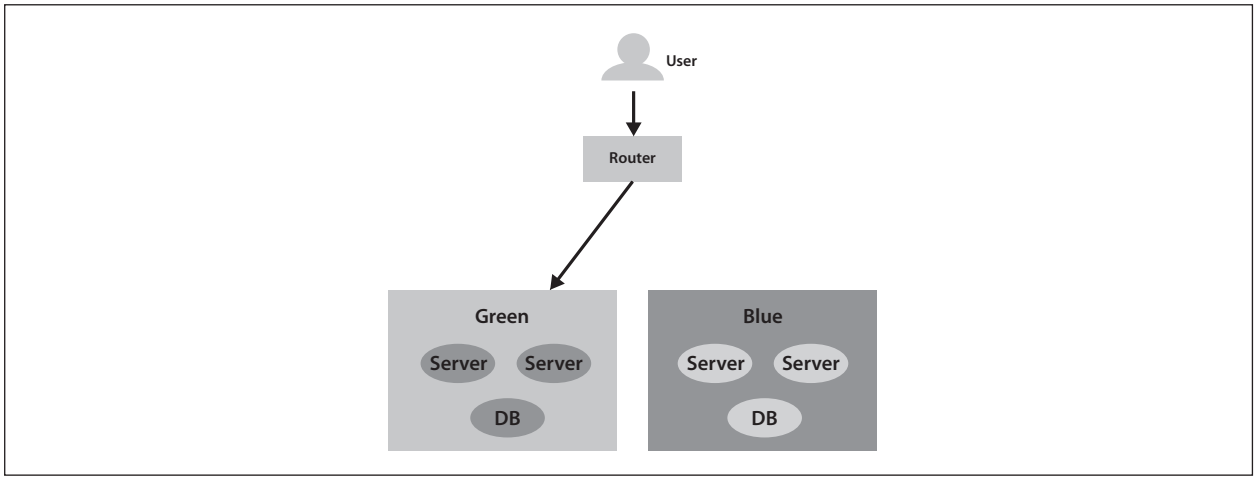

Abbildung 4: Blue-Green-Deployment

26 [CDPE], S. 173-179

Das Blue-Green-Deployment ist eine sehr einfache, aber wirkungsvolle Technik für das Release-Management. Ist gibt dabei zwei identische Produktivumgebungen, von denen nur die grüne Umgebung live ist. Eine neue Version der Software wird in die blaue Umgebung deployt. Anschließend können auch noch ein paar Smoketests ausgeführt werden, um sicherzustellen, dass die Anwendung auch läuft. Wenn alles wie gewünscht funktioniert, wird durch das Ändern der Routereinstellungen der Traffic auf die blaue Umgebung umgeleitet. Die blaue Umgebung mit der neuen Version wird zur grünen Umgebung und Grün wird zu Blau. Tritt ein Problem mit der neuen Version auf, kann der Traffic ganz einfach wieder umgeleitet werden. Die fehlerhafte Version kann nun in aller Ruhe auf der blauen Umgebung gedebuggt werden.[27]

Problematisch beim Blue-Green-Deployment ist der Umgang mit Datenbanken. Es gibt zwei mögliche Lösungsansätze:

- Die blaue und die grüne Umgebung verwenden jeweils ihre eigene Datenbank. Vor dem Wechsel der Umgebungen wird die grüne Datenbank auf read-only gestellt, ein Backup davon erstellt und in der blauen Datenbank eingespielt. Danach können Änderungen an der blauen Datenbank vorgenommen werden. Anschließend wird wieder zwischen Grün und Blau gewechselt und der schreibgeschützte Modus aufgehoben.[28]
- Die Änderungen an der Datenbank sind unabhängig von den Änderungen an der Software. Das Release der Datenbank wird damit vom Release der Anwendung entkoppelt und die Datenbank wird als eigenständige Komponente betrachtet. Ähnlich wie bei der Verwendung von APIs gibt es unterschiedliche Versionen, die abwärtskompatibel sind. Bei einer Datenbank gibt es eben unterschiedliche Versionen des Schemas. Das aktuelle Schema muss dabei auch das alte Schema unterstützen.[29]

2.5.3 Canary Releasing

Canary Releasing geht auf eine Idee der Bergmänner zurück. Unter Tage hatten sie Kanarienvögel dabei, die sie warnten, sobald es zu viel Kohlenmonoxid gab. Die Idee des vorzeitigen Warnsignals, wurde beim Canary Releasing übernommen. Zuerst wird die neue Version nur auf ein paar Server deployt, auf die dann nach und nach immer mehr Nutzer zugreifen können. Die Software wird somit langsam, aber sicher für alle Nutzer ausgerollt. Am Anfang kann die neue Software beispielsweise nur für Mitarbeiter oder für »power user« zur Verfügung stehen.

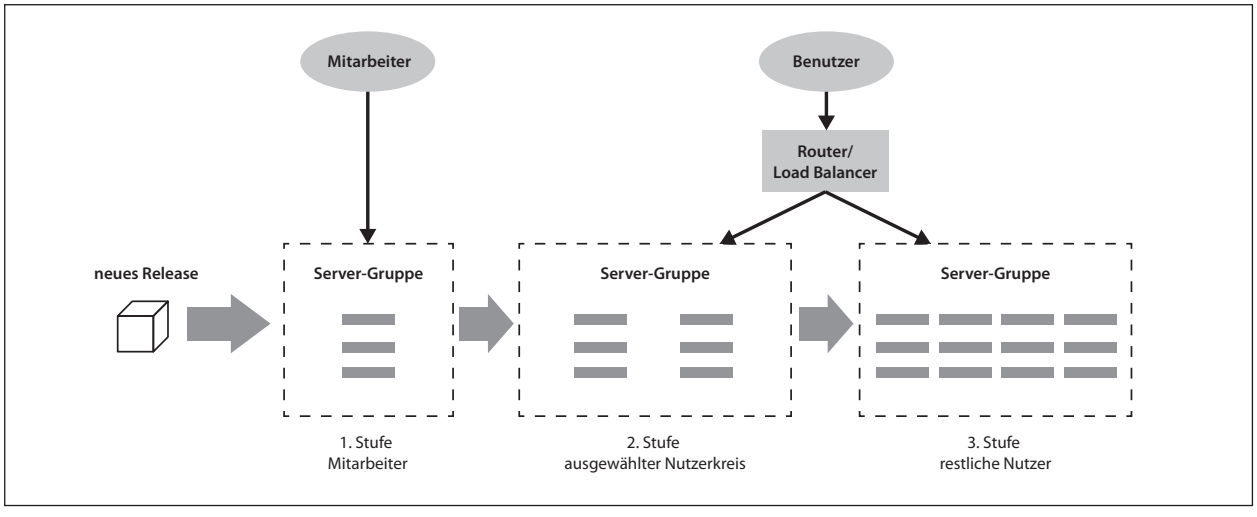

Abbildung 5: Canary-Release-Muster[30]

27 [CD], S. 261f
28 [CD], S. 262
29 [CDPE], S. 257
30 [DOH], S. 164

Neben dem sicheren ausrollen neuer Software kann die Produktionsumgebung auch als Testumgebung verwendet werden. Es kann überprüft werden, ob ein neues Feature bei den Nutzern gut ankommt oder ob die Performance akzeptabel ist. Sind die Kriterien bereits bei einem kleinen Nutzerkreis nicht erfüllt, ist dies das Warnsignal, die neue Version nicht weiter auszurollen.
Im Gegensatz zum Blue-Green-Deployment ist hier der Übergang zu einer neuen Version nicht so hart. Es sollte aber darauf geachtet werden, dass sich nicht zu viele unterschiedliche Versionen gleichzeitig in Produktion befinden.[31]

2.5.4 Feature-Toggle

Mit Feature-Toggles können bestimmte Features ein- und ausgeschaltet werden, ohne die Anwendung neu zu deployen. Wie bei den vorherigen Release-Techniken wird das Deployment vom Release entkoppelt. Die Entwickler können über den Feature-Toggle den Releasezeitpunkt selbst bestimmen. Ähnlich wie beim Canary-Releasing kann das Feature zuerst nur für einen bestimmten Personenkreis oder eine Region aktiviert werden. Läuft das Feature wie erwartet, wird es für weitere Nutzer freigeschalten.
Die Verwaltung der Feature-Toggles kann beispielsweise über eine Konfigurationsdatei oder sogar über einen eigenen Webservice erfolgen. Facebook nutzt intern den Service »Gatekeeper«[32], der Features für bestimmte Nutzergruppen oder Regionen aktivieren und deaktivieren kann.

2.5.5 Dark-Launch

Basis für Dark-Launches sind die Feature-Toggles. Feature-Toggles deaktivieren ein neues Feature komplett. Beim Dark-Launch ist das Feature aktiv, aber für die Benutzer nicht über die grafische Oberfläche sichtbar. Die neue Funktionalität kann somit unter realistischer Last z.B. auf Performance getestet werden. Es ist auch eine Kombination mit dem Canary-Releasing möglich, so dass die Last nach und nach größer wird und bei Problemen rechtzeitig eingegriffen werden kann. Mit den Dark-Launches können somit beispielsweise neue Such-Features getestet werden. Liefert die Suche in akzeptablen Zeiten das gewünschte Ergebnis, wird über einen Feature-Toggle die neue Funktionalität auch für die Nutzer sichtbar.[33]
Facebook nutzte die Dark-Launch-Methode für den Facebook-Chat. Mit dem Chat werden nicht nur Nachrichten versendet, sondern es wird auch der Online-/Offline-Status für alle meine Freunde angezeigt. Gerade die Ermittlung, welche Freunde online sind und welche offline sind ist sehr rechenintensiv. Während der Entwicklungsphase spendierte Facebook jedem Nutzer einen Chat-Client, der Nachrichten an den Server sendete. Die GUI des Chat-Client war für den Anwender allerdings nicht sichtbar. Jeder Facebook-Nutzer wurde somit zum Tester, wodurch Facebook realistische Testergebnisse erhielt.[34]

31 [CD], S. 263ff
32 Andrew Bosworth (2012). Building and testing at Facebook. *https://code.facebook.com/posts/187489991429453/building-and-testing-at-facebook/* [23.05.2018]
33 [DOH], S. 167f
34 Eugene Letuchy (2008). Facebook Chat. *https://code.facebook.com/posts/150168455181595/facebook-chat/* [23.04.2018]

3. Optimierung der Tests

3.1 Testpyramide

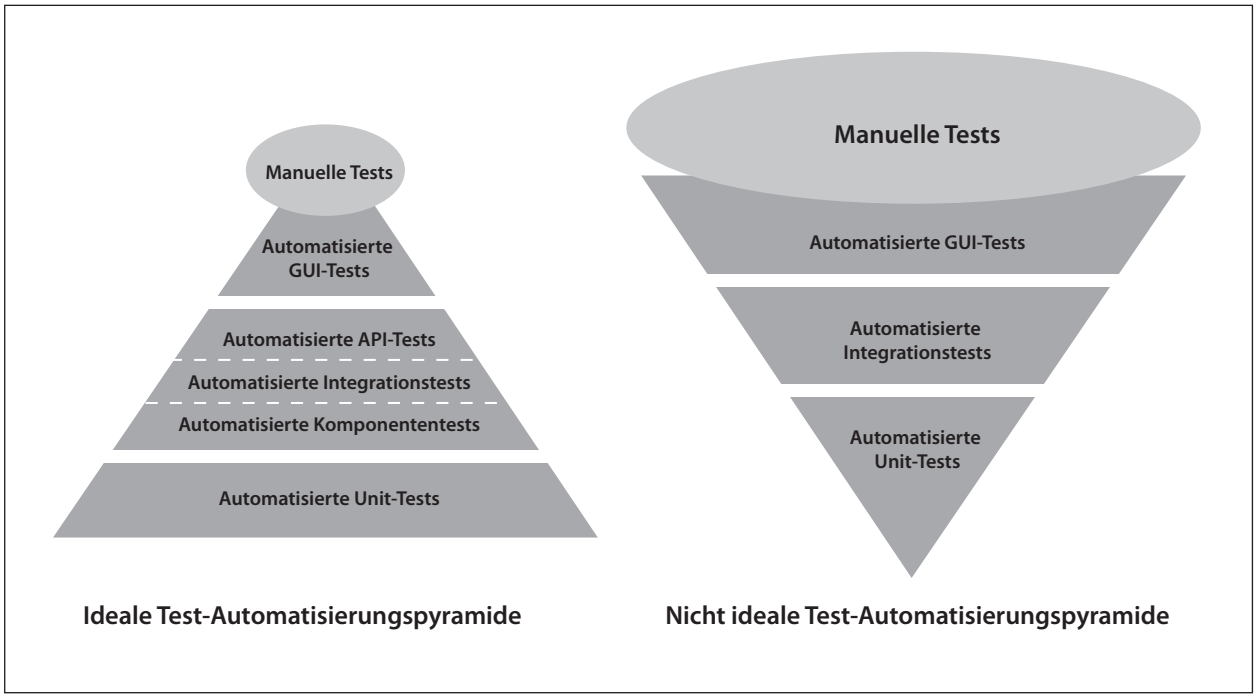

Abbildung 6: Ideale und nicht ideale Test-Automatisierungspyramide[35]

Die Abbildung zeigt, wie eine optimale Aufteilung der Tests aussieht. Bei Continuous Deployment gilt das Prinzip »Fail Fast«. Fehler sollten so früh wie möglich in der Pipeline entdeckt werden. Die meisten unserer Tests sollten Unittests sein. Sie befinden sich am Anfang der Pipeline und deren Ausführungszeit ist relativ kurz. Nach den schnellen Unittests kommen die längeren Tests, wie z.B. die Akzeptanz- oder Manuellen-Tests. Wenn ein Entwickler einen Bugfix in die Deployment-Pipeline bringt, sollte er so schnell wie möglich Feedback erhalten, ob der Fix das Bug beheben konnte. Wird der Fehler nur von Integrationstests überprüft, die lange Laufzeiten haben, kostet dies den Entwickler unnötige Zeit.[36]

35 [DOH], S. 127
36 [DOH], S. 126f

4. Fazit

Die Entwicklung einer Deployment-Pipeline ist ein aufwändiger Prozess, der nicht so schnell etabliert werden kann, wie es bei Continuous Integration noch der Fall war. Neben dem Aufbau der Prozesse und der Infrastrukturwn muss auch bei den Mitarbeitern eine Kultur entwickelt werden, in kleinen und regelmäßigen Abständen zu commiten.
Der in den vorherigen Abschnitten erklärte Aufbau der Pipeline dient als erste Grundlage. Basierend auf dieser Pipeline können weitere Stages hinzugefügt werden. Auch die Anordnung der Stages, kann individuell an die Gegebenheiten angepasst werden. Neben dem klassischen sequenziellen Ablauf ist auch eine Parallelisierung der Stages möglich.
Wenn die Deployment-Pipeline einmal aufgebaut ist, erleichtert sie den Softwareentwicklungsprozess ungemein. Neue Releases sind viel stressfreier und vor allem schneller möglich, was wiederum ganz neue betriebswirtschaftliche Möglichkeiten eröffnet. Es darf allerdings nicht vergessen werden, dass die Pipeline und die darin enthaltenen Tests stetig weiterentwickelt werden müssen.

Literaturverzeichnis

[DOH] Gene Kim, Jez Humble, Patrick Debois, John Willis (1. Auflage, 2017). Das DevOps Handbuch. Heidelberg: O'Reilly.

[CD] Jez Humble, David Farley (2011). Continuous Delivery. Boston: Pearson Education.

[CDPE] Eberhard Wolff (2. Auflage, 2016). Continuous Delivery – Der pragmatische Einstieg. Heidelberg: dpunkt.Verlag.