HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN MÜNCHEN

Fakultät 07

Studiengang Bachelor Wirtschaftsinformatik

Studienarbeit in dem Fach Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen

Anforderungen an Cloud Native Applikationen

Verfasser:	Luzie Mertingk
Studiengruppe:	IB6A
Matrikelnummer:	14873615
Datum:	11. Mai 2018
Dozent:	Michael Theis

Eigenständigkeitserklärung:

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

Ort, Datum

München, 11. Mai 2018

Name, Vorname, Unterschrift

Mertingk, Luzie L. Mertingk

Inhaltsverzeichnis

Inhaltsverzeichnis ..... **I**

Abbildungsverzeichnis ..... **II**

Abkürzungsverzeichnis ..... **III**

Kurzfassung **1**

1. Einleitung **2**

2. Motivation für Cloud-Computing **3**

3. Was ist Cloud-Computing? **4**

3.1 Typen des Cloud-Computing 4

3.2 Varianten 5

4. Änderungen in der klassischen Softwareentwicklung **6**

5. Die Zwölf-Faktor-Apps Methode **9**

5.1 Codebase 9

5.2 Abhängigkeiten 10

5.3 Konfiguration 10

5.4 Unterstützende Dienste 11

5.5 Build, release, run 12

5.5.1 Build 12

5.5.2 Release 12

5.5.3 Run 13

5.6 Prozesse 13

5.7 Bindung an Ports 14

5.8 Nebenläufigkeit 14

5.9 Einweggebrauch 15

5.10 Dev-Prod-Vergleichbarkeit 16

5.10.1 Die Zeit-Lücke 16

5.10.2 Die Personal-Lücke 17

5.10.3 Die Werkzeug-Lücke 17

5.11 Logs 17

5.12 Admin-Prozesse 18

6. Beispiel für die wichtigsten Punkte der Zwölf-Faktor-Apps Methode **19**

7. Fazit **24**

Literaturverzeichnis ..... **IV**

Abbildungsverzeichnis

Abbildung 1: Cloud-Computing-Architektur 5

Abkürzungsverzeichnis

URL	Uniform Resource Locator
SMTP	Simple Mail Transfer Protocol
CI	Continuous Integration
IaaS	Infrastructure-as-a-Service
PaaS	Platform-as-a-Service
SaaS	Software-as-a-Service
RAM	Random Access Memory
CPU	Central Processing Unit
HTTP	Hypertext Transfer Protocol
CRM	Customer Relationship Management

Kurzfassung

In diesem Dokument werden die Auswirkungen des Cloud-Computing auf die Softwareentwicklung betrachtet. Dazu werden zunächst die Merkmale und Eigenschaften dieser Technologien analysiert, dann im Hinblick auf die Softwareentwicklung betrachtet und anhand eines Beispiels plausibilisiert.

1. Einleitung

Wenn man sich heutzutage mit der Entwicklung von Anwendungen beschäftigt, stößt man häufig auf das Thema der Cloud. Während Google und Amazon als Pioniere des Cloud-Computing den Anfang machten, basieren moderne Unternehmen wie Netflix oder airbnb heutzutage ausschließlich auf der Cloud. Aufgrund stetig wachsender Digitalisierung führt in den nächsten Jahren für viele Anwendungen kein Weg mehr an der Cloud vorbei. Durch die nahezu unbegrenzte Skalierbarkeit, unbegrenztem Datenvolumen und Hochverfügbarkeit sind die Cloud-Dienste bei den Anwendern sehr beliebt.

Doch um eine echte Cloud-Anwendung entwickeln und betreiben zu können, kann man diese Unternehmen nicht einfach blind nachahmen. Sich in der Welt der Cloud anzupassen und zurechtzufinden dauert für Unternehmen oft mehrere Jahre. Startups können ihr Unternehmen von Anfang an auf die Cloud-Architektur aufbauen, größere, bereits etablierte Softwareunternehmen müssen in ihrer Organisation und in ihrem Vorgehen einen erheblichen Umschwung vornehmen. Nur so sind sie in der Lage, die Potentiale der Welt der Cloud zu nutzen.

Aus diesem Grund stellt sich die Frage: Was sind überhaupt Cloud-Anwendungen und welche Änderungen müssen in der herkömmlichen Softwareentwicklung vorgenommen werden, um echte Cloud-Native Applikationen entwickeln und betreiben zu können?

2. Motivation für Cloud-Computing

Zunächst sollen die Gründe erläutert werden, warum heutzutage immer mehr Unternehmen auf den Einsatz des Cloud-Computing setzen.

Ein großer Treiber des Cloud-Computing ist zunächst die vorhandene Elastizität. Hohe Lasten sind für Unternehmen nur schwer vorherzusehen, dazu kommen oft saisonale Schwankungen. Um mit solchen Lasten umzugehen, muss oft eine Vielzahl von Reserven bereitgehalten werden, die die meiste Zeit still liegen, also nicht genutzt werden und Kosten verursachen. Auch das Thema Big Data betrifft nicht mehr nur die großen Unternehmen, sondern auch kleinere Unternehmen müssen heutzutage mit großen Datenmengen umgehen, vor allem wenn es um die kontinuierliche und gezielte Kommunikation mit den Kunden geht.

Ein weiterer Treiber für den Einsatz von Cloud-Computing ist vor allem die Wirtschaftlichkeit. Bei den meisten klassischen On-Premise Rechenzentrumlösungen steigen die Kosten bei steigender Rechenleistung und steigendem Datenvolumen überproportional an. Bei Cloud-Anwendungen muss sichergestellt werden, dass die Kosten höchstens linear zu der Systemskalierung ansteigen.[1]

Neben dem Einsparungspotential bei den Infrastrukturkosten besteht weiterhin die Möglichkeit, die IT-Organisation zu verschlanken und damit Kosten für die Administration einzusparen. Die IT-Organisation größerer Unternehmen ist häufig sehr arbeitsteilig aufgebaut. Z.B. wird die Software-Entwicklung an Dienstleister vergeben, während der Aufbau der Infrastruktur und der Betrieb der Anwendungen intern oder durch andere Dienstleister erbracht wird. Selbst Letzteres ist teilweise arbeitsteilig organisiert, z.B. werden Netzwerk, Datenbanken, IT-Sicherheit etc. durch eigene Abteilungen verantwortet. Dies führt zu einem erheblichen Organisationsaufwand um alle Aspekte inhaltlich und zeitlich zu einem funktionierenden Ergebnis zusammen zu führen. Die Technologien des Cloud-Computing erlauben bzw. fördern u.a. durch einen hohen Automatisierungsgrad eine engere Verzahnung zwischen Entwicklung, Inbetriebnahme und Betrieb.

Ein weiterer großer Vorteil, den Cloud-Anwendungen bieten ist, dass sie nach dem Pay-per-Use-Prinzip abgerechnet werden. Das bedeutet, der Anwender bzw. sein Unternehmen bezahlt nur die tatsächliche Nutzung, z.B. der Infrastruktur oder Services. Dies wird realisiert durch zeitliche Abrechnungen oder einem Lizenzmanagement. Zusätzlich sind diese Services meist monatlich oder kürzer kündbar. Man verschwendet also keine Kosten durch Nichtnutzung.[2]

[1] Friedrichsen, U., & Dr. Kepser, S. (März 2011). Cloud Computing: Umdenken für Architekten in der Cloud. Abgerufen am 15. April 2018 von Codecentric: https://www.codecentric.de/publikation/rien-ne-va-plus-umdenken-fuer-architekten-in-der-cloud/ (Karlstetter, 2014)

[2] Karlstetter, F. (23. Mai 2014). Fünf gute Gründe für die Cloud. Abgerufen am 18. April 2018 von cloudcomputing-insider.de: https://www.cloudcomputing-insider.de/fuenf-gute-gruende-fuer-die-cloud-a-447047/

3. Was ist Cloud-Computing?

Nun muss zuerst die Frage geklärt werden: Was ist überhaupt Cloud-Computing und was bedeutet dies für Anwendungen bzw. die Anwendungsentwicklung?

Eine der eindeutigsten und am weitesten anerkannten Definitionen des Cloud-Computing stammt von der US-amerikanischen Standardisierungsstelle NIST (National Institute of Standards and Technology). Folgende Übersetzung befindet sich auf der Website des Bundesamtes für Sicherheit in der Informationstechnik (BSI):

„Cloud-Computing ist ein Modell der Datenverarbeitung, mit dem bei Bedarf, jederzeit und überall bequem über ein Netz auf einen geteilten Pool von konfigurierbaren Rechnerressourcen (z.B. Netze, Server, Speichersysteme, Anwendungen und Dienste) zugegriffen werden kann. Diese können schnell und mit minimalem Verwaltungsaufwand bzw. geringer Serviceprovider-Interaktion zur Verfügung gestellt werden.“[3]

Grundvoraussetzung für den Einsatz einer Public Cloud und Software-as-a-Service ist, dass Anwendungen nicht mehr auf den lokalen Rechnern installiert, sondern via Internet bzw. Intranet, z.B. über einen Webbrowser, aufgerufen werden.

3.1 Typen des Cloud-Computing

Beim Cloud-Computing gibt es drei Haupttypen: Die Public Cloud, die Private Cloud und die Hybrid Cloud.

Bei der Public Cloud handelt es sich um den bekanntesten Typ des Cloud-Computing. „Diese basieren auf globalen Netzwerken aus Rechenzentren und stellen nutzungsbasierte Services für die Öffentlichkeit oder große Unternehmen bereit.“[4]

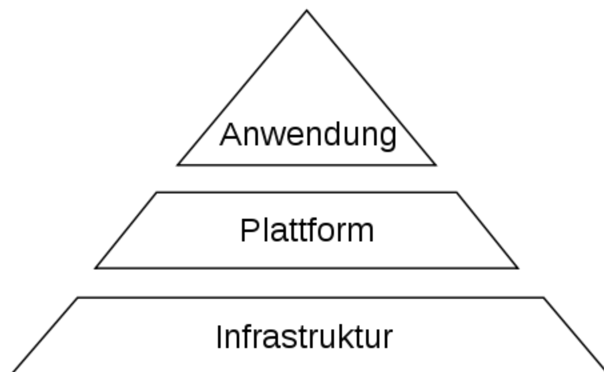
Bei einer Private Cloud handelt es sich um eine Cloud, die ein Unternehmen auf Basis ihrer eigenen Computing-Hardware bereitstellt. Diese betreibt das Unternehmen selbst (oder auch ein Dienstleister) und ausschließlich ihre eigenen Mitarbeiter oder dedizierte Partner bei der Bereitstellung von B2B-Services haben Zugriff darauf.[5] Diese sind gut geeignet, wenn ein Unternehmen vertrauliche Daten nicht außerhalb ihrer eigenen Systeme speichern möchte.

„Eine Mischform aus Public Cloud und Private Cloud stellt die sog. Hybrid Cloud dar. In dieser werden Computing- und Speicher-Services zwischen Public Cloud und Private Cloud verteilt. Somit können vertrauliche Daten intern und allgemeinere Daten extern auf preisgünstigeren Public-Cloud-Servern gespeichert werden.“[6]

[3] Bundesamt für Sicherheit in der Informationstechnik. Abgerufen am 11.April2018 von https://www.bsi.bund.de/DE/Themen/DigitaleGesellschaft/CloudComputing/Grundlagen/Grundlagen_node.html
[4] Dr. Alleweldt, F., Dr. Kara, et al. Cloud Computing. Abgerufen am 11. April 2018 von http://www.europarl.europa.eu/RegData/etudes/etudes/join/2012/475104/IPOL-IMCO_ET(2012)475104_DE.pdf
[5] Fraunhofer-Institut für Arbeitswirtschaft und Organisation. Abgerufen am 11. April 2018 von Fraunhofer Cloud: https://www.cloud.fraunhofer.de/de/faq/publicprivatehybrid.html
[6] ebd.

3.2 Varianten

Cloud-Computing lässt sich in verschiedene Arten unterteilen. Die Unterscheidung wird anhand der Cloud-Computing-Architektur deutlich, welche aus den drei Schichten Infrastruktur-, Plattform- und Anwendungsschicht besteht.



Bei Infrastructure-as-a-Service (kurz: IaaS) handelt es sich um die Bereitstellung von Infrastruktur in einer Cloud. Der Nutzer verwaltet seine Infrastruktur nicht mehr selber lokal, sondern kann auf die Ressourcen in einer Cloud zugreifen. Der Vorteil hierbei ist die Flexibilität, da sich der Nutzer die benötigten Ressourcen selbst einteilen kann. Es muss also keine feste Größe an Infrastruktur lokal bereitstehen, sondern der Nutzer kann nach Bedarf auf die Infrastruktur aus der Cloud zugreifen.

Bei Platform-as-a-Service (kurz: PaaS) handelt es sich um die mittlere Schicht der Cloud-Architektur. Hier wird Unternehmen ermöglicht, auf Basis von Plattformfunktionalitäten eigene Anwendungen zu entwickeln und auszuführen. Dazu werden vom PaaS-Anbieter Entwicklungsumgebungen bereitgestellt.

Der Unterscheid zu IaaS ist hier, dass der Entwickler nicht auf die eingesetzte Infrastruktur zugreift.

Bei Software-as-a-Service (kurz: SaaS) wird dem End-Nutzer bzw. einem Unternehmen eine Anwendung als Dienstleistung gegen Entgelt zur Verfügung gestellt.

Auf diese Anwendung kann von verschiedenen Geräten via Internet, z.B. über einen Internetbrowser, zugegriffen werden.

„Software wird [also] nicht lizensiert und auf eigener Hardware installiert, sondern nur als Service „angemietet“ und kann überall mit jedem Internetbrowser genutzt werden.“[7]

[7] Henzel, T. (8. Januar 2016). Cloud Computing: SaaS, PaaS & IaaS einfach erklärt. Abgerufen am 16. April 2018 von https://www.datenschutzbeauftragter-info.de/die-cloud-saas-paas-und-iaas-einfach-erklaert/

4. Änderungen in der klassischen Softwareentwicklung

In diesem Kapitel wird sich auf den Artikel „Cloud-Computing: Umdenken für Architekten in der Cloud“ aus dem Magazin „Business Technology“ bezogen, welcher einen Überblick über die wesentlichen Merkmale des Cloud-Computing gibt.

Die Vorgehensweise der klassischen Softwareentwicklung für Rechenzentren und der Standalone Anwendungsentwicklung können nicht einfach auf die Entwicklung einer Cloud-Anwendung übertragen werden. Die Anwendungen müssen hier anderen Anforderungen genügen, v.a. Skalierbarkeit, Elastizität und Robustheit.

Das Design, die Infrastruktur, der Betrieb und die Architektur einer solchen Anwendung müssen an die veränderten Anforderungen angepasst werden, was also ein Umdenken bei der Entwicklung einer echten Cloud-Anwendung erfordert.

Einer der wichtigsten Anforderungen an eine Cloud-Anwendung ist die Verfügbarkeit. Hier gibt es einige Änderungen im Vergleich zu herkömmlichen Anwendungen. Ist in diesen hohe Verfügbarkeit gefordert, nutzt man häufig die Hochverfügbarkeitsangebote der Hardware- und Softwareanbieter.

Das Problem hierbei ist jedoch, dass sie sich zum einen nicht beliebig skalieren lassen und zum anderen die Kosten mit der Skalierung überproportional ansteigen.

„Stattdessen setzt man in Cloud-Umgebungen in der Regel auf einfache Standardhardware und auf Redundanz zur Sicherstellung der Verfügbarkeit. Das bedeutet im Gegenzug aber auch, dass man aus Anwendungssicht immer damit rechnen muss, dass eine Systemkomponente im nächsten Moment nicht mehr verfügbar ist. Ausfälle werden damit nicht mehr transparent für die Anwendungslogik behandelt, sondern müssen explizit beim Design der Anwendung betrachtet werden.“ [8]

Außerdem muss man bei der Entwicklung einer Cloud-Applikation das Thema der Datenkonsistenz beachten. Aus dem sog. „CAP-Theorem“ ergibt sich, dass die Konsistenz nicht uneingeschränkt aufrechterhalten werden kann. Gemäß diesem Theorem kann ein System nicht gleichzeitig die Anforderungen „Konsistenz (Consistency), Verfügbarkeit (Availability) und Toleranz gegenüber partiellen Systemausfällen (Partition Tolerance) erfüllen, sondern nur maximal zwei von ihnen.

Da in einer Cloud-Anwendung immer damit gerechnet werden muss, dass einzelne Komponenten nicht hochverfügbar sind, muss die vollständige Datenkonsistenz aufgegeben werden. Daher sollte die Anwendung auf keinen Fall Mechanismen beinhalten, die auf die Konsistenz der verarbeiteten Daten absolut vertrauen.

Man muss die Daten also auf verschiedene Speicherorte verteilen. „Da verteilte Transaktionen nicht beliebig skalierbar sind (zumindest, wenn man die Systemverfügbarkeit hoch halten will), muss man zu einer weitestgehend unabhängigen Verteilung und Speicherung der Daten übergehen.“[9]

---

[8] Friedrichsen, U., & Dr. Kepser, S. (März 2011). Cloud Computing: Umdenken für Architekten in der Cloud. Abgerufen am 15. April 2018 von Codecentric: https://www.codecentric.de/publikation/rien-ne-va-plus-umdenken-fuer-architekten-in-der-cloud/

[9] ebd.

Dazu empfiehlt es sich, die Daten in Form von Entitäten zu organisieren. Diese können unabhängig voneinander verteilt werden. Unter einer Entität versteht man in diesem Zusammenhang einen Ordnungsbegriff inklusive aller zugeordneten Daten.

Daher basiert die Entwicklung einer Cloud-Anwendung auf dem sogenannten „Shared-Nothing"-Gedanken. Klassische Clusterarchitekturen lassen sich nicht beliebig skalieren, da sich bestimmte Komponenten einen Zustand teilen und der Kommunikationsaufwand bei steigender Skalierung für die Replikation zu groß wird. Der „Shared-Nothing"-Gedanke hilft, dieses Problem in Cloud-Anwendungen zu umgehen. Dieser besagt, dass die Komponenten auf einer Ebene keinen Zustand miteinander teilen. Damit kann man das Problem mit der Replikation lösen, die Komponenten kommunizieren nur noch in Form von Nachrichten.

Eine Voraussetzung für diesen Gedanken ist die Lose Kopplung auf der Anwendungsebene. Die Skalierbarkeit kann nur erreicht werden, wenn die Komponenten auf fachlicher und technischer Ebene möglichst unabhängig voneinander sind. Auf fachlicher Ebene geschieht dies anhand der zuvor beschriebenen Entitäten. Dies lässt sich ebenso auf die Anwendungsebene übertragen. Ein Block der Anwendungslogik lässt sich leichter skalieren, wenn sich dieser vom Rest der Anwendungslogik abkoppeln lässt. Durch die Entkopplung der einzelnen Bestandteile der Anwendung kann diese robuster gestaltet und die Verfügbarkeit damit erhöht werden. Auf technischer Ebene bedeutet die Entkopplung in Verbindung mit dem Shared-Nothing-Gedanken, dass die einzelnen Blöcke nur noch mittels Nachrichten miteinander kommunizieren.

Die beschriebenen Prinzipien sorgen für eine verstärkte Parallelisierbarkeit der Anwendung, beziehungsweise der Anwendungsteile. „Je mehr nicht parallelisierbare Anteile ein Programm enthält, desto geringer ist der Geschwindigkeitsgewinn bei einer Parallelisierung der Ausführung."[10]

Es muss also darauf geachtet werden, dass bei einer Cloud-Anwendung der Anteil der nicht-parallelisierbaren Anteile möglichst gering ist.

Wenn man das zuvor beschriebene Shared-Nothing-Prinzip betrachtet, sollte darauf verzichtet werden, dass sich parallel ausgeführte Bausteine einen Zustand teilen.

Da man in Cloud-Anwendungen im Sinne der wirtschaftlichen Realisierung von Elastizität einfache Hardware, u.a. auch dynamisch, einsetzt, ist es im Sinne der Verfügbarkeit wichtig, möglichst wenig Zustand einer Komponente im Hauptspeicher zu verwalten.

Auch beim Thema Fehlerbehandlung gibt es in der Cloud-Entwicklung einige Änderungen. Auf Grund der lose gekoppelten Komponenten kann man den Entwicklern nicht mehr freie Wahl lassen, wie sie einen Fehler behandeln, da sie die Folgen für die anderen Komponenten nicht abschätzen können. „Aus diesem Grunde verbietet es sich auch, die Fehlerbehandlung dem Container oder Framework zu überlassen. Die Applikation zu beenden oder den Fehler zu ignorieren, ist bei der Anforderung an die Verfügbarkeit ohnehin keine Option."[11]

---

[10] ebd S. 6

[11] ebd. S 6

Ein Ansatz für die Fehlerbehandlung ist der Einsatz von sog. Workern und Supervisorn. Der Worker erledigt hierbei eine Teilaufgabe entweder erfolgreich oder „stirbt“ in einem Fehlerfall. Wenn dies der Fall ist, erledigt der Supervisor die Fehlerbehandlung, welcher in einem eigenen Thread abläuft. Der Worker kümmert sich demnach nur um die Ausführung des Programms und muss sich nicht gleichzeitig um die Behebung von Fehlern kümmern.

Ein weiterer wichtiger Aspekt bei der Entwicklung einer Cloud-Anwendung ist die Automatisierung.

Sie ist die Voraussetzung für die Bereitstellung elastischer Anwendungen. Um eine Anwendung in kürzester Zeit skalieren zu können, ist es notwendig, ein automatisiertes Deployment und einen automatisierten Start der verschiedenen Anwendungsteile inkl. automatisierter Bereitstellung der benötigten Infrastruktur durchzuführen.
Das bedeutet, dass z.B. ein Monitorprogramm selbstständig bei Erkennen einer hohen Last eine weitere Instanz des benötigten Anwendungsteils aufsetzen und starten muss.

5. Die Zwölf-Faktor-Apps Methode

Wie man in dem Vorangegangen bereits erkennen konnte, müssen Cloud-Applikationen bestimmte Anforderungen erfüllen. Die Zwölf-Faktor-Apps Methode ist ein Manifest für die Entwicklung von SaaS-Anwendungen. Sie gibt einen Überblick darüber, welche Punkte dabei zu beachten sind.

Die Methode wurde von den Mitwirkenden bei der Entwicklung des PaaS-Dienstes Heroku entwickelt. In diesem Kapitel soll neben den von ihnen geschaffenen Grundlagen der Methode, welche der Webseite 12factor.net zu entnehmen sind[12], zusätzlich auf die von Kevin Hoffmann verfasste Erweiterung der Methode „Beyond the Twelve-Factor-App“ eingegangen werden.

5.1 Codebase

Der erste Faktor der Methode besagt: „Eine im Versionsmanagementsystem verwaltete Codebase, viele Deployments“

Bei der Verwaltung unzähliger Aspekte eines Entwicklungsteams wird die Organisation von Code oft als unwichtiges Details angesehen oder geradezu vernachlässigt. Die richtige Anwendung von Organisation ist bei der Entwicklung jedoch sehr wichtig. Cloud-Applikationen müssen immer aus einer einzigen Codebase bestehen, die in einem Versionsmanagementsystem verwaltet wird. Bei einem „Repository“ handelt es sich um eine Kopie der Versionsdatenbank.

„Eine Codebase ist jedes einzelne Repo (in einem zentralisierten Versionsmanagementsystem wie Subversion) oder jede Menge von Repos, die einen ursprünglichen Commit teilen (in dezentralisierten Versionsmanagementsystemen wie git).“[13]

Die Codebase wird verwendet, um eine beliebige Anzahl unveränderlicher Releases zu erzeugen, die für verschiedene Umgebungen bestimmt sind. Die Entwicklungsteams müssen den Zuschnitt ihrer Anwendung analysieren und potentielle Monolithen erkennen, welche in einzelne Services aufgeteilt werden müssen. Wenn es mehrere Codebases gibt, handelt es sich nicht mehr um eine App, sondern um ein verteiltes System. „Wenn mehrere Apps denselben Code teilen, verletzt dies die zwölf Faktoren. Lösen lässt sich dies, indem man den gemeinsamen Code in Bibliotheken auslagert und über die Abhängigkeitsverwaltung lädt.“[14]

Das einfachste Beispiel für die Verletzung dieser Regel ist, wenn eine Anwendung aus mehreren Quellcode-Repositories besteht. Dies macht es fast unmöglich, die Build- und Deploy-Phasen des Lebenszyklus der Anwendung zu automatisieren. Umgekehrt kann diese Regel verletzt werden, wenn eine Codebase zum Erstellen mehrerer Anwendungen verwendet wird.

[12] The Twelve-Factor App. (2017). Abgerufen am 18. April 2018 von https://12factor.net/de/

[13] Codebase: The Twelve-Factor App. (2017). Abgerufen am 17. April 2018 von https://12factor.net/de/codebase

[14] ebd.

5.2 Abhängigkeiten

Bei dem zweiten Faktor der Methode handelt es sich um Abhängigkeiten.

Aus klassischen Enterprise-Umgebungen sind wir an das Konzept des „Mommy-Servers" gewohnt. Dies ist ein Server, der alles bereitstellt, was eine Anwendung benötigt: von der Auflösung der Abhängigkeiten bis zur Bereitstellung eines Servers, auf dem die App gehostet wird.

„Die meisten Programmiersprachen bieten ein System an, um unterstützende Bibliotheken zu verbreiten […]. Aus einem Paketsystem stammende Bibliotheken können systemweit installiert werden […] oder in ein Verzeichnis der App beschränkt werden […]."[15]

Eine Zwölf-Faktor-App kann jedoch nicht davon ausgehen, dass ein Server alles bereitstellt, was sie benötigt. In Java bedeutet dies, dass die Anwendung nicht davon ausgehen kann, dass ein Container den Klassenpfad auf dem Server verwaltet. Unabhängig von der Programmiersprache kann sich der Code einer Cloud-Anwendung nicht auf die Existenz von Abhängigkeiten verlassen.

Abhängigkeiten nicht ordnungsgemäß zu isolieren kann zu unzähligen Problemen führen. Beispielweise könnte ein Entwickler an der Version X einer abhängigen Bibliothek arbeiten, stattdessen wird aber Version X+1 dieser Bibliothek an einem zentralen Ort der Produktion installiert. Dies kann u.a. zu Laufzeitfehlern führen, die, wenn sie nicht behandelt werden, einen ganzen Server zum Absturz bringen können.

„[Um dies zu vermeiden] deklariert [eine Zwölf-Faktor-App] alle Abhängigkeiten vollständig und korrekt über eine Abhängigkeitsdeklaration. Weiter benutzt sie zur Laufzeit ein Werkzeug zur Isolation von Abhängigkeiten um sicherzustellen, dass keine impliziten Abhängigkeiten aus dem umgebenden System "hereinsickern". Die vollständige und explizite Spezifikation der Abhängigkeiten wird gleichermaßen in Produktion und Entwicklung angewandt."[16]

5.3 Konfiguration

Der dritte Faktor fordert die strikte Trennung der Konfiguration vom Code. Das bedeutet, dass Apps die Konfiguration auf keinen Fall als Konstanten im Code speichern dürfen. Dies ist wichtig, da sich die Konfigurationen im Gegensatz zum Code in den Deploys unterscheiden.

Zunächst sollte die Definition einer Konfiguration geklärt werden.

Unter Konfigurationen fallen u.a. URLs zu Web-Services und SMTP-Servern, Informationen zu einer Datenbankverbindung oder Anmeldeinformationen zu Drittanbieter-Diensten. Unter Konfigurationen fallen jedoch nicht die internen Konfigurationen, die Teil der Anwendung selbst sind, da sich diese nicht in den verschiedenen Deploys unterscheiden. Für die Konfiguration könnte man beispielsweise Dateien verwenden, die nicht ins Versionsmanagement eingecheckt werden. Es kann jedoch passieren, versehentlich eine solche Konfigurationsdatei ins Repository einzuchecken.

---

[15] Abhängigkeiten: The Twelve-Factor App. (2017). Abgerufen am 17. April 2018 von https://12factor.net/de/dependencies

[16] ebd.

„Die Zwölf-Faktor-App speichert die Konfiguration in Umgebungsvariablen (kurz auch env vars oder env).“[17] Dies hat den Vorteil, dass es im Gegensatz zu den oben genannten Konfigurationsdateien eher unwahrscheinlich ist, dass diese versehentlich ins Repository eingecheckt werden. Zusätzlich sind sie Sprach- und Betriebssystemunabhängig.

Dennoch sollten Konfigurationen auch verwaltet werden und möglichst automatisiert beim Deployment reproduziert herangezogen werden.

5.4 Unterstützende Dienste

Der vierte Faktor besagt: „Unterstützende Dienste als angehängte Ressourcen behandeln“.

Bei unterstützenden Diensten handelt es sich um Dienste, die eine App für ihre Funktionalität benötigt. Dies können u.a. Datenspeicher wie MySQL, Message-Systeme, SMTP-Dienste für E-Mails oder Cache-Systeme sein.

Wenn eine Anwendung in einer Cloud-Umgebung laufen soll, muss man auch den Dateispeicher oder die Festplatte als unterstützenden Dienst behandeln. Im Gegensatz zu herkömmlichen Enterprise Anwendungen, bei welchen auf eine Datei von der Festplatte geschrieben oder von einer Datei von der Festplatte gelesen wurde, sollte der Dateispeicher in Cloud-Umgebungen einen unterstützenden Dienst darstellen, der als Ressource an die Anwendung gehängt wird.

Eine Zwölf-Faktor-App macht keine Unterscheidung zwischen lokalen Diensten, wie z.B. einem Dateispeicher, oder Diensten von Dritten. Diese Dienste sind über eine URL abrufbar, die wiederum in der Konfiguration gespeichert sind. Jeder unterstützende Dienst stellt eine Ressource dar.

Wie bereits unter dem Kapitel „Konfiguration“ beschrieben, kann sich die Konfiguration von Deploy zu Deploy unterscheiden. So könnte beispielsweise in einem Deploy auf eine lokale Datenbank zugegriffen werden und in einem anderen Deploy auf Datenbank von einem Drittanbieter, ohne dass im Code eine Änderung ersichtlich ist.

An diesem Beispiel wird deutlich, dass es für eine Cloud-Anwendung wichtig ist, dass die Konfiguration vom Code getrennt sein muss. Wenn man diesen Grundsatz auf die Ressourcenbindung überträgt, ergeben sich dafür einige Regeln:

- Die Anwendung sollte die Notwendigkeit für eine bestimmte Ressource deklarieren, die tatsächliche Durchführung der Ressourcenbindung sollte sie aber der Cloud-Umgebung überlassen.
- Es sollte möglich sein, Ressourcen an die Anwendung anzuhängen oder zu entfernen, ohne die Anwendung erneut deployen oder gar implementieren zu müssen.

[17] Konfiguration: The Twelve-Factor App. (2017). Abgerufen am 18. April 2018 von https://12factor.net/de/config

Letzteres spielt beispielweise dann eine Rolle, wenn eine Datenbank aufgrund eines Hardwareproblems ausfällt. In diesem Fall muss es möglich sein, dass der Administrator eine neue Datenbank aufsetzt, ohne dass eine Codeänderung vorgenommen werden muss.

Zusätzlich soll kurz auf die sogenannten „Circuit Breakers“ eingegangen werden. Bei Circuit Breakers handelt es sich um ein Pattern, welches von Bibliotheken und Cloud-Angeboten unterstützt wird. Dieses ermöglicht es, die Kommunikation mit fehlerhaften Diensten zu beenden und einen Fallback Path bereitzustellen. Der Circuit Breaker befindet sich häufig zwischen der Anwendung und seinen unterstützenden Diensten.

5.5 Build, release, run

Der fünfte Faktor besagt, dass die Build- und die Run-Phase strikt getrennt werden müssen.

Eine Codebase durchläuft drei Phasen, durch welche sie in einen Deploy umgewandelt wird. Zuerst durchläuft sie die Build-Phase, in welcher das Code-Repository in ein kompiliertes Code-Bündel übersetzt wird. Dieses Bündel wird dann in der Release-Phase mit der Konfiguration zusammengeführt, die sich, wie bereits beschrieben, außerhalb der Anwendung befindet. Daraus ergibt sich dann eine unveränderliche Version. Diese wird dann in der Run-Phase in eine Cloud-Umgebung geliefert und ausgeführt.

Worum es in diesem Kapitel geht ist, dass jede dieser Phasen isoliert und separat ausgeführt wird. Im Folgenden sollen die einzelnen Phasen genauer beschrieben werden.

5.5.1 Build

In der Build-Phase wird ein Code-Repository ein versioniertes, binäres Artefakt umgewandelt. Hier werden die deklarierten Abhängigkeiten geholt und gebündelt. Dieses Bündel wird auch „Build“ genannt. In Java könnte ein solches Build beispielweise eine WAR- oder JAR-Datei sein. Builds werden idealerweise von einem Continuous-Integration-Server erstellt, wie z.B. Jenkins. Ein einzelner Build sollte in einer beliebigen Anzahl von Umgebungen released oder deployed werden können.

Der Vorteil an dem Einsatz eines Continuous-Integration-Server, abgekürzt CI-Server, ist, dass sich dieses Muster „ein Build, viele Deploys“ damit umsetzen lässt. Wenn man sich immer darauf verlassen kann, dass die Codebase überall funktioniert, gibt es keine Probleme mehr beim Release. Das kontinuierliche Deployment und die schnellen Releases sind zwei der größten Vorteile der Cloud-Native Philosophie.

5.5.2 Release

In der Release-Phase wird der Output der Build-Phase mit der umgebungs- und anwendungsspezifischen Konfiguration kombiniert und daraus dann ein weiteres unveränderbares Artefakt erzeugt, ein Release. Releases müssen eindeutig sein und jede Version sollte idealerweise

einer eindeutigen ID versehen sein, z.B. mit einem Zeitstempel oder einer inkrementierenden Nummer.

Aufgrund der 1:n-Beziehung zwischen Builds und Releases darf ein Release nicht mit der Build-ID versehen werden. Der CI-Server erstellt ein Artefakt und kann dieses dann für die Entwicklung- und Produktionsumgebung bereitstellen. Jedes dieser Releases sollte eindeutig sein, da jedes das ursprüngliche Build mit den umgebungsspezifischen Konfigurationseinstellungen kombiniert.

Wenn ein Fehler auftritt, hat man so die Möglichkeit zu überprüfen, was in einer bestimmten Umgebung released wurde und ggf. den Release auf die vorherige Version zurücksetzen. Aus diesem Grund müssen Releases unveränderlich und eindeutig identifizierbar sein.

5.5.3 Run

Die Run-Phase wird normalerweise auch vom Cloud-Anbieter ausgeführt, Entwickler müssen die Anwendung dennoch lokal ausführen können.

Es gibt Unterschiede zwischen den Anbietern, meistens wird aber die Anwendung in einem Container-Image, z.B. Docker, platziert, woraufhin dann ein Prozess gestartet wird um die Anwendung auszuführen. Wenn eine Anwendung ausgeführt ist, ist die Cloud-Runtime dafür verantwortlich, diese am Leben zu halten, ihren Zustand zu überwachen und viele andere administrative Aufgaben wie dynamische Skalierung und Fehlertoleranz durchzuführen. Das Ziel dieses Faktors ist es, die Liefergeschwindigkeit zu maximieren und dabei durch automatisiertes Testen und Deployen eine hohe Transparenz über die Qualität sicher zu stellen.

5.6 Prozesse

Der sechste Faktor behandelt zustandslose Prozesse. Anwendungen sollten als einzelner, zustandsloser Prozess ausgeführt werden.

Die Frage die sich hier stellt ist, wie ist es möglich, einen Prozess aufzubauen, der keinen Zustand aufrechterhält?

Eine zustandslose Anwendung nimmt keine Annahmen über Speicherinhalte vor. Sie kann während der Bearbeitung einer Transaktion einen Übergangszustand erstellen und konsumieren, diese Daten sollten jedoch alle zu dem Zeitpunkt verloren sein, wenn der Client eine Antwort erhalten hat. Das bedeutet, dass jeder dauerhafte Zustand außerhalb der Anwendung sein muss, der durch einen unterstützenden Dienst bereitgestellt wird. Dies bedeutet lediglich, dass Zustände zwar erlaubt sind, sich jedoch außerhalb der Anwendung befinden müssen.

Ein gängiges Muster besteht darin, häufig verwendete Daten im RAM oder dem Dateisystem als kurzfristigen Cache für die Dauer einer Transaktion zwischen zu speichern. Eine Zwölf-Faktor-App geht jedoch nie davon aus, dass etwas, das in diesem Cache zwischengespeichert ist, für einen künftigen Request verfügbar sein wird. Dieser künftige Request kann nämlich auch von einem anderen Prozess bearbeitet werden.

Wenn sich Prozesse Daten teilen müssen, z.B. einen Sitzungsstatus für mehrere Prozesse, sollte dieser externalisiert und über einen unterstützenden Dienst bereitgestellt werden.

Es gibt viele Caching-Produkte von Drittanbietern, wie beispielsweise Gemfire oder Redis, die so konzipiert sind, dass sie als unterstützender Dienst verwendet werden können. Sie können für den Sitzungsstatus oder auch zum Zwischenspeichern von Daten, die Prozesse für ihren Startvorgang benötigen, verwendet werden

5.7 Bindung an Ports

Web-Anwendungen werden oft in Containern ausgeführt, beispielsweise eine Java-Anwendung in Tomcat.

In einer Nicht-Cloud-Umgebung werden Web-Anwendungen in diesen Containern bereitgestellt, welche dann für das Zuweisen von Ports für Anwendungen beim Start verantwortlich sind. In einer Cloud-Umgebung sollte jedoch der Cloud-Anbieter die Portzuordnung übernehmen, da dieser üblicherweise auch das Routing, die Skalierung, hohe Verfügbarkeit und Fehlertoleranz verwaltet. Dazu muss der Cloud-Anbieter bestimmte Netzwerkaspekte verwalten, einschließlich die Zuordnung von Hostnamen zu den Ports und die Zuordnung externer Portnummern zu containerinternen Ports.

Beispielsweise läuft ein Dienst innerhalb der Anwendung auf der Portnummer 8080, von außen wird der Dienst jedoch über eine andere Portnummer aufgerufen. In einer Cloud-Anwendung sollten Dienste jedoch nicht über eine Kombination von IP-Adresse und Portnummer aufgerufen werden, sondern über Namen in Form von URL’s.

5.8 Nebenläufigkeit

Im achten Faktor geht es darum, Anwendungen mithilfe des Prozessmodells zu skalieren.

Wenn eine herkömmliche Anwendung die Grenze ihrer Kapazität erreichte, bestand die Lösung darin, die Anwendung einfach „größer“ zu machen, z.B. durch Hinzufügen von CPUs oder RAM. Dieses Vorgehen nennt man auch vertikale Skalierung. In einer Cloud-Anwendung hingegen gibt es auch die Möglichkeit, eine horizontale Skalierung durchzuführen. Anstatt einen einzigen großen Prozess noch größer zu machen, können mehrere Prozesse erstellt werden und so die Last der Anwendung auf mehrere Prozesse verteilen. Durch die zustandslosen Shared-Nothing-Prozesse kann die horizontale Skalierung voll ausgenutzt werden und mehrere Instanzen einer Anwendung können gleichzeitig ausgeführt werden. Eine Zwölf-Faktor-App kann, aufgrund ihrer Orientierung am Unix-Prozess-Modell, für die Verarbeitung verschiedenster Aufgaben konzipiert werden. Dazu wird zu jeder Aufgabe ein Prozesstyp zugewiesen.

„Zum Beispiel können HTTP-Requests durch einen Web-Prozess bedient werden und langlaufende Hintergrundarbeiten durch einen Worker-Prozess.“[18]

Die einzelnen Prozesse können weiterhin z.B. mittels Threads ihr internes Multiplexing verwalten.

5.9 Einweggebrauch

Der neunte Faktor sorgt für Robustheit durch schnellen Start und problemlosen Stopp.

Die Prozesse einer Cloud-Anwendung sind „wegwerfbar“, das bedeutet, dass sie schnell gestartet oder gestoppt werden können. Eine Anwendung kann nur schnell skaliert, deployed, released oder wiederhergestellt werden, wenn sie schnell gestartet und ordnungsgemäß heruntergefahren werden kann.

Eine langsame Startzeit sollte möglicherweise Warnungen auslösen, wenn die Anwendung ihre Integritätsprüfung nicht besteht. Extrem langsame Startzeiten können sogar verhindern, dass die App überhaupt in der Cloud startet. Dadurch kann der Release-Phase und der Skalierung mehr Agilität verliehen werden. Außerdem sorgt ein schneller Start für Robustheit, da Prozesse bei Bedarf einfacher auf neue physikalische Maschinen verschoben werden können. Wenn die App nicht schnell heruntergefahren wird, kann dies auch die Fähigkeit beeinträchtigen, sie nach einem Fehlschlag wieder hochzufahren. Außerdem besteht die Gefahr, dass Ressourcen nicht ordnungsgemäß entsorgt werden können, wodurch Daten beschädigt werden können.

Viele Anwendungen führen beim Start eine Reihe lang dauernder Aktivitäten aus, wie z.B. das Abrufen von Daten zum Füllen eines Cache oder das Vorbereiten anderer Laufzeitabhängigkeiten. In einer Cloud-Architektur muss diese Art von Aktivität separat behandelt werden. Beispielsweise könnte ein Cache in einen unterstützenden Dienst ausgelagert werden, sodass die Anwendung schnell hoch- und runterfahren kann, ohne vorher Ladevorgänge auszuführen.

„Für einen Web-Prozess kann ein problemloses Herunterfahren erreicht werden, indem er aufhört an seinem Service-Port zu lauschen (und damit alle neuen Requests ablehnt), die laufenden Requests zu Ende bearbeitet und sich dann beendet. […] Für einen Worker-Prozess wird ein problemloser Stopp erreicht, indem er seinen laufenden Job an die Workqueue zurückgibt.“[19]

[18] Nebenläufigkeit: The Twelve-Factor App. (2017). Abgerufen am 18. April 2018 von https://12factor.net/de/concurrency

[19] Einweggebrauch: The Twelve-Factor App. (2017). Abgerufen am 18. April 2018 von https://12factor.net/de/disposability

5.10 Dev-Prod-Vergleichbarkeit

In dem zehnten Faktor geht es darum, Entwicklung, Staging und Produktion so ähnlich wie möglich zu halten.

In der herkömmlichen Softwareentwicklung gibt es große Lücken zwischen Entwicklungs- und Produktionsumgebungen, schon allein wegen der hohen Infrastrukturkosten. Beispielsweise unterscheiden sich die in der Entwicklung und Qualitätssicherung verwendeten Datenbanktreiber von der Produktion. Sicherheitsregeln, Firewalls und Umgebungskonfigurationen sind ebenfalls unterschiedlich. Ein weiteres Beispiel ist, dass Entwickler nur in der Lage sind, in einer Umgebung zu deployen, aber in einer anderen wiederum nicht. Wenn zwischen all diesen Faktoren große Unterschiede vorhanden sind, haben Entwickler meist „Angst“ vor dem Deployment, da sie sich nicht sicher sein können, dass wenn die Anwendung in einer Umgebung funktioniert, sie das auch in einer anderen Umgebung tut.

Der Zweck dieses Faktors, also eben solche Lücken zu schließen, ist es, dass die Entwicklung und auch die gesamte Organisation darauf vertrauen kann, dass die Anwendung überall, also in jeder Umgebung, funktionieren wird.

Es gibt nahezu unbegrenzte Möglichkeiten, solche Lücken zu schaffen. Die Lücken, die jedoch am häufigsten auftreten, zeigen sich in drei Gebieten:

- Zeit
- Personal
- Werkzeuge

Auf die Gebiete, wie die Lücken entstehen und verhindert werden können, soll im Folgenden kurz eingegangen werden.

5.10.1 Die Zeit-Lücke

In vielen Organisationen kann es Wochen oder sogar Monate dauern, bis ein Code, den ein Entwickler eingecheckt hat, in Produktion geht. Wenn solch große Zeitlücken auftreten, kann es passieren, dass die Entwickler vergessen, welche Änderungen in einem Release vorgenommen wurden oder sogar wie der Code aussah. Es sollte deshalb versucht werden, die Zeit zwischen dem Einchecken des Codes und der Produktion zu Minuten oder Stunden zu verkürzen. Ein Entwickler arbeitet also an Code, der noch am selben Tag deployed wird.

Am Ende einer ordnungsgemäßen Continuous Deployment-Pipeline sollten automatisierte Tests in verschiedenen Umgebungen ausgeführt werden, bis die Änderung automatisch in die Produktion übertragen wird.

5.10.2 Die Personal-Lücke

Die Personal-Lücke entsteht dadurch, dass Entwickler den Code schreiben, aber Operatoren ihn wiederum deployen.

Die Zwölf-Faktor-App verkleinert diese Lücke, indem die Entwickler, die den Code geschrieben haben, auch am Deployment und der Produktion intensiv beteiligt sind.

In „Beyond the Twelve-Factor App“ unterscheidet sich die Meinung des Autors von der Meinung der ursprünglichen Zwölf-Faktor-App. Der Autor erklärt, dass niemals Menschen Anwendungen deployen sollten, zumindest nicht außerhalb ihrer eigenen Workstation. In einer ordnungsgemäßen Build-Pipeline wird eine Anwendung automatisch für alle Umgebungen deployed und kann basierend auf Sicherheitsbeschränkungen innerhalb des CI-Tools manuell in anderen Umgebungen bereitgestellt werden. Man sollte also per Knopfdruck auf bestimmte Ereignisse reagieren können.

5.10.3 Die Werkzeug-Lücke

Entwickler werden oft dazu verleitet, einen leichtgewichtigen unterstützenden Dienst in der lokalen Umgebung zu benutzen, der dem Dienst, der in der Produktion verwendet wird, ähnelt. Durch solche Kompromisse vergrößert sich die Lücke zwischen Entwicklung und Produktion. Die Zwölf-Faktor-Regel besagt deswegen, dass der Entwickler keine unterschiedlichen Dienste in Entwicklung und Produktion verwenden sollte. Denn wenn hierbei Unterschiede entstehen, kann es passieren, dass Code, der in der Entwicklung funktioniert und auch Tests besteht, in der Produktion dann wiederum nicht mehr funktioniert. Zudem sind moderne Dienste dank Paketierungssystemen wie apt-get einfach zu installieren zu starten. Außerdem können Tools wie Docker dazu beitragen, produktionsähnliche Umgebungen für Entwickler zugänglicher zu machen.

In einer Zwölf-Faktor-App ist jeder Commit ein Kandidat für das Deployment. Wenn eine Änderung eingecheckt wird, sollte diese nach kurzer Zeit auch in die Produktion gelangen. Wenn sich die Entwicklungs-, Test- und Produktionsumgebungen voneinander unterscheiden, kann nicht mehr genau vorhergesagt werden, wie sich die Codeänderung in der Produktion verhalten wird. Wie im vorangegangen bereits erwähnt ist für ein kontinuierliches und schnelles Deployment jedoch wichtig, dass immer darauf vertraut werden kann, dass die Anwendung überall funktioniert.

5.11 Logs

Logs sollten laut dem elften Faktor wie Ereignisströme behandelt werden.

Logs sind eine Folge von zeitlich sortierten Ereignissen aller laufenden Prozesse und unterstützenden Diensten. Eine Zwölf-Faktor-App befasst sich jedoch nicht mit dem Routing oder der Speicherung ihrer Output-Streams.

In der herkömmlichen Entwicklung war es üblich, dass die Entwickler das Ziel der Logs streng kontrollieren konnten. Konfigurationsdateien richten den Speicherort auf der Festplatte ein, an dem sich die Log-Dateien befinden. Cloud-Anwendungen können jedoch keine Annahmen über das

Dateisystem machen, auf dem sie ausgeführt werden. Die laufenden Prozesse einer Cloud-Anwendungen schreiben ihre Logeinträge in stdout (Standardausgabe). In einer lokalen Umgebung kann ein Entwickler diese Logeinträge über ein Terminal beobachten.

Einer der Gründe, warum die Anwendung die Protokolle nicht vollständig kontrollieren sollte, ist die Skalierbarkeit. Bei einer festen Anzahl von Instanzen auf einer festen Anzahl von Servern ist das Speichern von Protokollen auf der Festplatte sinnvoll. Wenn die Anwendung jedoch eine dynamische Anzahl von laufenden Instanzen besitzt, und nicht bekannt ist, wo diese laufen, muss sich der Cloud-Anbieter um die Protokolle kümmern. Durch die Vereinfachung dieses Prozesses kann die Codebase reduziert werden und die Entwickler können sich auf die Kernaufgabe konzentrieren.

5.12 Admin-Prozesse

Der zwölfte und letzte Faktor besagt, dass Admin- und Management-Aufgaben als einmalige Vorgänge behandelt werden.

Während die Prozesse zur Erledigung der üblichen Aufgaben einer App laufen, wie der Verarbeitung von Web-Requests, möchten Entwickler nebenbei oft einmalige administrative Aufgaben an der App erledigen. Dazu gehört z.B. das Starten einer Datenbankmigration, das einmalige Ausführen von Skripten aus dem Repo oder aus der Konsole heraus beliebigen Code zu starten. Diese Administrationsprozesse sollten in einer Umgebung laufen, die der Umgebung der üblichen Prozesse gleich ist. Sie benutzen die gleiche Codebase und Konfiguration wie jeder Prozess, der gegen ein Release läuft.

Der Verfasser von „Beyond the Twelve-Factor Apps" kritisiert diesen Faktor allerdings. Der Autor Kevin Hoffmann ist hier der Meinung, dass dieser Faktor irreführend ist. Die oben genannten administrativen Prozesse sollten zu eigenen Funktionalitäten in der Anwendung umstrukturiert werden, damit auch diese Teil des Release-Prozesses sind.

6. Beispiel für die wichtigsten Punkte der Zwölf-Faktor-Apps Methode

Im vorangegangenen Kapitel wurde ein ausführlicher Überblick über die Zwölf-Faktor-Apps Methode gegeben. Die Methode beschreibt die zwölf Faktoren, die beachtet werden müssen, um eine Cloud-Anwendung zu entwickeln und zu betreiben. Im Folgenden sollen die wichtigsten Punkte dieser App anhand eines Anwendungsbeispiels erläutert werden.

Als Beispiel dient der Cloud-Anbieter Salesforce.com, welcher Geschäftsanwendungen für Unternehmen über das Internet zur Verfügung stellt und vor allem auf Kundenbeziehungsmanagement (CRM) und Vertriebsautomatisierung spezialisiert ist.[20]

Nun soll zunächst die Frage geklärt werden: Wozu setzen Unternehmen ein CRM-System ein?

Dies soll am Beispiel eines Autohändlers erläutert werden. Der Autohändler möchte einen konkreten Überblick über seine Kundenbeziehungen haben, z.B. wenn es um den Bereich Leasing geht. Durch das Kundenbeziehungsmanagement können die Vertragslaufzeiten überwacht und kontrolliert werden, um den Kunden kurz vor Ende der Laufzeit spezifische Angebote machen zu können. Das Unternehmen sieht also „per Knopfdruck", welche Autos in welchem Zeitraum an welchen Kunden verkauft wurden.

Diese Daten sollen jedoch nicht mehr in der Schublade der einzelnen Verkäufer liegen, sondern in einem zentralisierten CRM-System verwaltet werden um Auswertungen über die gesamte Kundenbasis durchführen zu können. Nun hat das Unternehmen die Möglichkeit, ein solches System entweder als fertige Lösung eines Softwareherstellers wie SAP zu lizensieren, lokal zu installieren und zu betreiben, oder ein solchen System selbst zu entwickeln.

Was braucht also ein Unternehmen, um ein fertiges oder selbst entwickeltes CRM-System betreiben können?

Zum einen können die Anwendungen lokal auf den Rechnern der einzelnen Mitarbeiter laufen, welche wiederum auf einen zentralen Server zugreifen, um einen gemeinsamen Zustand herzustellen, welcher mit Hilfe von Zugriffsschutzmechanismen verwaltet werden kann. Dazu müssen die lokalen Anwendungen zunächst auf alle Benutzerrechner installiert und die Server angeschafft und in einem Netzwerk betrieben werden. Dafür braucht das Unternehmen ein Rechenzentrum, welches vor allem vor unbefugten Zugriffen gesichert werden muss. Eine solche Infrastruktur kann hohe Kosten verursachen. Wie in Kapitel 4 bereits beschrieben, besteht eine der Ursachen darin, dass die Infrastruktur für Spitzenlasten gerüstet sein muss und die Kosten mit steigender Last überproportional ansteigen. Außerdem wird ein organisatorischer Prozess für das Ausrollen von Updates sowie den Support für die Benutzer benötigt.

Um das Problem bzw. den Aufwand für die Administration (z.B. Updates) der Software auf den einzelnen Arbeitsplatzrechnern zu entschärfen, könnte man eine Web-Anwendung entwickeln. Diese

[20] *Salesforce.com: Wikipedia.* (April 2018). Abgerufen am 19. April 2018 von https://de.wikipedia.org/wiki/Salesforce.com

kann dann über einen Standard Webbrowser aufgerufen werden, sodass die wesentliche Anwendungslogik nur noch auf dem Server läuft und der Browser die Benutzerschnittstelle anbietet.

Um das Problem der teuren Infrastruktur zu umgehen, besteht die Möglichkeit, diese über einen Infrastructure-as-a-Service Anbieter zu beziehen. Diese Lösungen bieten den Vorteil, dass sie sich dynamisch nach der Nutzeranzahl skalieren lassen und somit hohe Kosten mit steigender Nutzung vermieden werden können. Das Unternehmen kann die Anwendung also nun skalieren, entwickelt und verwaltet das CRM-System jedoch immer noch selbst.

Es gibt zusätzlich noch die Möglichkeit, einen Plattform-as-a-Service Dienst zu nutzen. CRM-Systeme bieten alle eine ähnliche Funktionalität und verwenden ähnliche Techologien. Sie benötigen beispielsweise eine Datenbank, Webserver, Autorisierung, Mandantenfähigkeit oder eine Reportfunktionalität. Aus diesem Grund stellen Cloud-Anbieter eine Plattform bereit, auf dessen Basis-Funktionen die einzelnen Unternehmen ihre eigenen Anwendungen entwickeln können. Der Autohändler entwickelt also die Anwendung noch selbst, lädt diese aber in eine bestehende Plattform hoch.

Die nächste Stufe stellen dann die Software-as-a-Service Dienste dar, um welche es in der Zwölf-Faktor-App Methode eigentlich geht. Hier wird also nicht mehr nur die Infrastruktur oder die Plattform, sondern die gesamte Anwendung als Dienst bereitgestellt. Der Autohändler hat bei der Nutzung eines solchen Dienstes vor allem den Vorteil, dass dieser aufgrund des Pay-Per-Use-Prinzips nach Nutzung bezahlt werden kann und sich nicht um Betrieb und Support kümmern muss.

Wie bereits am Anfang des Kapitels erwähnt, stellt Salesforce.com, im Folgenden auch Salesforce genannt, ein CRM-System als einen SaaS Dienst bereit.

Nun wird das Ganze aus der Sicht von Salesforce betrachtet, wobei sich folgende Frage stellt: Was muss nun bei der Entwicklung eines CRM-Systems als Cloud-Anwendung im Gegensatz zu der Entwicklung eins eigenen CRM-Systems durch einen Autohändler beachtet werden?

Zunächst muss die Funktionalität unterschiedlichen Benutzern zur Verfügung gestellt werden. Als Softwareanbieter liefert man also keine einzelnen Softwarepakete mehr, die der Autohändler dann deployed, sondern es muss sichergestellt werden, dass die Software allen Kunden hochverfügbar bereit steht.

Wie in Kapitel 4 bereits beschrieben, spielt hier die Skalierbarkeit eine große Rolle. Das System lässt sich beliebig skalieren, wenn neue Instanzen der Anwendung schnell hochgefahren werden können, wie der neunte Faktor der Methode beschreibt. Bei großen Lasten können somit schnell und möglichst automatisiert neue Instanzen erzeugt und die Last somit verteilt werden. Um von einer Anwendung mehrere Instanzen erstellen zu können, sollte der erste Faktor der Methode beachtet werden, welcher dem Prinzip nachgeht: eine Codebase, viele Deploys. Salesforce möchte seinen Dienst beispielsweise in Europa und auch in den USA anbieten. In diesem Fall besteht die Anforderung, die Daten der amerikanischen Benutzer in amerikanischen Rechenzentren, und die Daten der europäischen Benutzer in europäischen Rechenzentren zu verwalten. Dazu muss eine einzige Codebase auf verschiedene

Arten deployed werden. Das bedeutet für die Entwickler, dass man nicht mehr wie früher eine Codebase für ein System für ein bestimmtes Rechenzentren deployed, sondern eine Codebase in unterschiedlichen Konfigurationen deployen möchte, und zwar einerseits in der Cloud in der USA und andererseits in der Cloud in Europa.

Wie in Faktor drei der Konfiguration beschrieben, unterscheidet sich nämlich die Konfiguration in den verschiedenen Deploys. Im Rahmen des Anwendungsbeispiels gibt es jeweils eine Konfiguration für die USA und für Europa. Es könnten nämlich beispielweise unterschiedliche Verschlüsselungsverfahren für die Datenhaltung gefordert sein. Die Unterschiede in der Konfiguration von den USA zu Europa sollen jedoch nicht im Code stehen. Der Code muss unabhängig davon sein, in welchem Markt und mit welcher Konfiguration er deployed wird, also allem, was umgebungsspezifisch ist.

Wenn im Code auf eine bestimmte Ressource zugegriffen wird, darf die Adressierung dieser Ressource nicht direkt im Code stehen, sondern muss in der Konfiguration enthalten sein. Diese holt sich die Anwendung aus einer nicht im Code enthaltenen Quelle, wie einer Konfigurationsdatei oder, wie die Zwölf-Faktor-Apps Methode vorschreibt, in einer Umgebungsvariable. Diese Information wird beim Bauen eines Releases für ein spezielles Deployment, beispielsweise für die USA, aus der Konfiguration entnommen. Hier wird dann also der Code mit der Konfiguration zusammengeführt.

Die Phasen Build und Run müssen laut dem Faktor fünf strikt getrennt betrachtet werden. Die Codebase wird, unabhängig von Markt, in dem die Anwendung später laufen wird, in der Build-Phase zunächst in ein kompiliertes, versioniertes Artefakt umgewandelt und dieses dann in der Release-Phase mit der Konfiguration kombiniert. Dieses eindeutige Release wird dann in der Run-Phase in der jeweiligen Cloud in Europa und in den USA ausgeführt. Wie bereits im Kapitel dieses Faktors erwähnt, ergibt sich durch dieses Verfahren ein erheblicher Vorteil, da die Anwendung kontinuierlich deployed werden kann und nicht nur alle paar Monate. Da man sich durch dieses Vorgehen immer sicher sein kann, dass die Codebase in jeder Umgebung ausgeführt werden kann, also unabhängig davon, in welchem Markt sie ausgeführt wird, muss man keine „Angst" mehr vor einem Deployment haben. Somit kann Salesforce seine Anwendung auch in vielen weiteren Märkten deployen, ohne etwas an der Codebase zu verändern.

Wenn Salesforce ein neues Feature bereitstellten möchte, da es von Kunden angefordert wird, soll dieses schnell eingebracht und deployed werden können. Dazu ist es notwendig, dass die Anwendung nicht aus einem großem Monolithen besteht, sondern auf mehrere Komponenten verteilt wird. Somit können mehrere Teams bei der Entwicklung an verschiedenen Features und Komponenten arbeiten, wie beispielsweise Sales Analytics oder die Adressverwaltung der Kunden im CRM-System. Wenn ein Team an einer Funktion arbeiten möchte, muss sich dieses nicht den gesamten Quellcode aus dem Repository auschecken, sondern nur eine einzelne Komponente.

Einen weiteren wichtigen Punkt der Methode stellt der vierte Faktor dar. Dieser besagt, dass die unterstützenden Dienste wie angehängte Ressourcen behandelt werden. Salesforce benötigt beispielsweise aufgrund der unterschiedlichen Märkte einen Währungsrechner, um Preise für den

jeweiligen Markt berechnen zu können. Die Währungskurse basieren jedoch in den USA und in Europa auf unterschiedlichen Börsen.

Bei der Entwicklung einer klassischen Anwendung könnte für den Währungsrechner eine library eingebunden werden, die beispielweise speziell die API der Börse Frankfurt anspricht. In der Entwicklung der Cloud-Anwendung hingegen muss der Währungsdienst beliebig austauschbar sein. Man baut also keine börsenspezifische library ein, sondern verwendet einen Dienst, der den entsprechenden Währungsrechner anbindet. So kann dieser einerseits in den verschiedenen Märkten, anderseits aber auch innerhalb eines Marktes ausgetauscht werden. Dies soll aber nicht die Anwendungslogik betreffen, sondern durch Konfigurationen festgelegt werden.

Bezogen auf den oben beschrieben Punkt des schnellen Aufbauens von Instanzen soll noch auf den Faktor sechs der Methode eingegangen werden. Dieser behandelt die zustandslosen Prozesse. Wenn die Anwendung speziell für den Autohändler entwickelt wird, kann ein Zustand gehalten werden. Weil die Cloud-Anwendung jedoch vielen Unternehmen angeboten werden soll, soll der Zustand in einem Prozess nicht für die verschiedenen Mandanten gehalten werden. Bei steigender Last sollen schnell weitere Prozesse instanziiert werden, und nicht erst darauf gewartet werden, bis ein Zustand geladen wurde. Da eine Cloud-Anwendung viele Anwender hat, müssen Prozesse gestartet werden, eine Anfrage bearbeiten, und sobald die mit der Bearbeitung der Anfrage fertig sind, wieder heruntergefahren werden, ohne dass es Auswirkungen auf den Rest der Anwendung hat. Wie in Kapitel 4 bereits anhand der Worker und Supervisor beschrieben, kann es beispielweise vorkommen, dass bei der Bearbeitung der Anfrage etwas schief geht, der Prozess dann stirbt, aber der Rest der Anwendung weiterhin funktioniert. Auf diesem Prozessmodell baut wiederum die in Faktor neun beschriebene Nebenläufigkeit auf. Im Rahmen der horizontalen Skalierung können schnell neue Prozesse erzeugt werden und aufgrund ihrer Zustandslosigkeit mehrere Instanzen der Anwendung nebenläufig ausgeführt werden.

Dies wird an der Funktion des Reporting des CRM-Systems deutlich, welche eine Auskunft über den Auftragseingang gibt. Diese Erstellung eines Reports, bestehend aus Auswertungen und ggf. Grafiken, kann einige Zeit in Anspruch nehmen. Während dieser Prozess läuft, soll der Benutzer davon jedoch nichts mitbekommen. Deswegen muss dieser Prozess nebenläufig zu allen anderen Prozessen ausgeführt werden. Der Benutzer soll lediglich eine Nachricht erhalten, wenn der Report fertig erstellt wurde und zum Download bereitsteht. Dies bedeutet aber bezogen auf das in Kapitel 4 beschriebene CAP-Theorem, also dass die Daten nur „eventuell konsistent“ sind, also dass ein Auftrag, der in der Zwischenzeit erfasst wird während der Report erstellt wird, darin nicht berücksichtigt ist.

In der klassischen Entwicklung würde die Konsistenz hergestellt werden, in dem der Auftrag in dem Moment nicht angelegt werden kann, da dies durch die Reporterstellung gesperrt ist. Eine andere Möglichkeit wäre, den Report zum Zeitpunkt eines eintreffenden Auftrags abzubrechen, da er ja sonst kein richtiges Ergebnis liefern würde.

Bei einer Cloud-Anwendung behandelt man diesen Fall bezogen auf das CAP-Theorem so, dass der Report eben nur die Aufträge bis zum Start des Reports berücksichtigt und dies dem Anwender mitgeteilt wird.

Auch der dritte Faktor, welcher die Abhängigkeiten behandelt, stellt einen wichtigen Punkt für das Entwickeln und Betreiben der Cloud-Anwendung dar. Bei der klassischen Entwicklung des CRM-Systems kann davon ausgegangen werden, dass Abhängigkeiten wie der Klassenpfad oder Datenbanktreiber in einem Rechenzentrum verwaltet bzw. aufgelöst werden. Man ist der Annahme, dass wenn die Anwendung deployed wird, alle Abhängigkeiten aufgelöst sind, da sie vorher schon bereitgestellt wurden.

Salesforce kann bei der Cloud-Anwendung jedoch nicht so vorgehen, da die Anwendung nicht davon ausgehen kann, dass ein die Abhängigkeiten bereits durch ein Rechenzentrum deklariert und aufgelöst wurden. Die Cloud-Anwendung muss also in einer Abhängigkeitsdeklaration angeben, welche Abhängigkeiten sie benötigt, sodass diese ggf. automatisiert durch diese Deklaration bereitgestellt werden können.

Idealerweise werden die Abhängigkeiten bei einem automatisierten Aufsetzen der Infrastruktur per „Infrastructure as Code“ aufgelöst, d.h. für jede Konfiguration existiert Code, der die Infrastruktur per API der Cloud-Plattform konfigurationsspezifisch erzeugt.

7. Fazit

In dieser Arbeit wurde deutlich, dass sich die Prinzipien bei der Entwicklung einer Cloud-Anwendung ein einigen Punkten von denen der herkömmlichen Softwareentwicklung unterscheiden. Durch die veränderten Anforderungen an solche Applikationen, darunter die Skalierbarkeit, Hochverfügbarkeit und Elastizität, ergeben sich neue Regeln, die bei der Entwicklung und des Betriebs einer Cloud-Anwendung beachtet werden müssen. Die Zwölf-Faktor-Apps Methode erläutert diese Regeln. Aus ihnen lässt sich die Wichtigkeit des kontinuierlichen Deployments, der kurzen Releasezyklen und vor allem die Möglichkeit, eine Anwendung in vielen verschiedenen Umgebungen zu deployen, entnehmen. Sie verdeutlicht auch eines der Grundprinzipien einer Cloud-Anwendung, und zwar dass sich eine solche Anwendung, da sie nicht wie herkömmliche Anwendungen nur in lokalen Umgebungen ausgeführt werden, nie auf das Vorhandensein von Abhängigkeiten oder Speichersystemen verlassen kann.

Wenn man bei der Entwicklung einer Anwendung diese Prinzipien beachtet ist man dazu in der Lage, ein echtes Cloud-Native-System entwickeln und betrieben zu können und damit die Potenziale dieser Technologie nutzbar zu machen.

Literaturverzeichnis

Abhängigkeiten: The Twelve-Factor App. (2017). Abgerufen am 17. April 2018 von https://12factor.net/de/dependencies

Bundesamt für Sicherheit in der Informationstechnik. (kein Datum). *Cloud Computing Grundlagen*. Abgerufen am 11. April 2018 von https://www.bsi.bund.de/DE/Themen/DigitaleGesellschaft/CloudComputing/Grundlagen/Grundlagen_node.html

Codebase: The Twelve-Factor App. (2017). Abgerufen am 17. April 2018 von https://12factor.net/de/codebase

Dr. Alleweldt, F., Dr. Kara, S., Fielder, A., Brown, I., Weber, V., & McSpedden-Brown, N. (1. Mai 2012). *Cloud Computing.* Abgerufen am 11. April 2018 von http://www.europarl.europa.eu/RegData/etudes/etudes/join/2012/475104/IPOL-IMCO_ET(2012)475104_DE.pdf

Einweggebrauch: The Twelve-Factor App. (2017). Abgerufen am 18. April 2018 von https://12factor.net/de/disposability

Fraunhofer-Institut für Arbeitswirtschaft und Organisation. (2017). Abgerufen am 11. April 2018 von Fraunhofer Cloud: https://www.cloud.fraunhofer.de/de/faq/publicprivatehybrid.html

Friedrichsen, U., & Dr. Kepser, S. (März 2011). *Cloud Computing: Umdenken für Architekten in der Cloud*. Abgerufen am 15. April 2018 von Codecentric: https://www.codecentric.de/publikation/rien-ne-va-plus-umdenken-fuer-architekten-in-der-cloud/

Henzel, T. (8. Januar 2016). *Cloud Computing: SaaS, PaaS & IaaS einfach erklärt*. Abgerufen am 16. April 2018 von https://www.datenschutzbeauftragter-info.de/die-cloud-saas-paas-und-iaas-einfach-erklaert/

Karlstetter, F. (23. Mai 2014). *Fünf gute Gründe für die Cloud*. Abgerufen am 18. April 2018 von cloudcomputing-insider.de: https://www.cloudcomputing-insider.de/fuenf-gute-gruende-fuer-die-cloud-a-447047/

Konfiguration: The Twelve-Factor App. (2017). Abgerufen am 18. April 2018 von https://12factor.net/de/config

Nebenläufigkeit: The Twelve-Factor App. (2017). Abgerufen am 18. April 2018 von https://12factor.net/de/concurrency

Salesforce.com: Wikipedia. (April 2018). Abgerufen am 19. April 2018 von https://de.wikipedia.org/wiki/Salesforce.com

The Twelve-Factor App. (2017). Abgerufen am 18. April 2018 von https://12factor.net/de/