fhwedel
UNIVERSITY OF APPLIED SCIENCES

Department of Computer Science

Bachelor Thesis

Vergleich eines Usecases mit Serverless Technologie gegenüber Spring Boot Technologie

am Beispiel von Instant Payments

Eingereicht am:

25. August 2021

Eingereicht von:
Silas Hoffmann
Traberweg 52
22159 Hamburg
Tel.: (040) 643 94 73
E-mail: inf103088@stud.fh-wedel.de

Referent:
Prof. Dr. Dennis Säring
Fachhochschule Wedel
Feldstraße 143
22880 Wedel
Phone: (041 03) 80 48-43
E-mail: dennis.saering@fh-wedel.de

Betreut von:
Kai Roßdeutscher
DPS Engineering GmbH
Eiffestraße 78
20537 Hamburg
Phone: (040) 25 15 41-44
E-mail: kai.rossdeutscher@dps.de

Vergleich eines Usecases mit Serverless Technologie gegenüber Spring Boot Technologie

am Beispiel von Instant Payments

Bachelor Thesisvon Silas Hoffmann

Das ist ein Einleitungstext!

Irgendein Copyright Text Layout done with LaTeX 2ε, KOMA-Script and BibTeX.

Inhaltsverzeichnis

Abbildungsverzeichnis

List of Listings

1 Einleitung

1.1 Überblick

Google, Spotify oder Netflix: Immer mehr Unternehmen setzen heutzutage auf Virtualisierungsplattformen. Aber warum sind Technologien, wie zum Beispiel *Docker* als Containerisierungsplattform oder *Docker Swarm* als Orchestrator, so beliebt? In der folgenden Thesis werde ich zunächst erläutern, welche Technik hinter dem Docker-Ökosystem steckt, um anschließend einen Überblick über die verfügbaren Technologien zu geben und diverse Eigenarten näher zu erläutern (was bedeutet es serverless zu deployen etc.). Es wird außerdem einen detaillierten Vergleich zweier moderner Technologien hinsichtlich der Eignung für diese Art von Plattform erarbeitet.

1.2 Motivation

Die Idee und die Umsetzung der Arbeit erfolgte in Zusammenarbeit mit der DPS Engineering Gmbh. Hierbei handelt es sich um ein europaweit tätiges Software- und Consultingunternehmen für die Finanz- und Retailbranche. Das Leistungsportfolio umfasst Softwareprodukte und IT-Services für die bekannten Vertreter der deutschen Kreditwirtschaft und des Handels[1]. Die mir zugeteilte Abteilung verwaltet und erweitert eine Plattform zur Abwicklung von Echtzeitüberweisungen. Um diese in Zukunft noch effizienter zu gestalten, möchte sich das Unternehmen von den altbewährten Java-Enterprise Technologien hin zu neuartigen Cloud-Technologien orientieren. Der besondere Fokus hierbei liegt auf der ressourcenschonenderen Arbeitsweise dieser Technologien. Man möchte in Zukunft darauf verzichten stets alle verfügbaren Ressourcen direkt zu verwalten und sich auf das automatische Skalieren dieser Systeme verlassen können. So ist es beispielsweise möglich, dynamisch neue Instanzen benötigter Komponenten zu erzeugen, als sich auf ein starres System zu verlassen, welches fortdauernd mit der gleichen Arbeitslast arbeitet. Dies erspart dem Unternehmen nicht nur Kapital, sondern führt ebenfalls zu einer besseren Klimabilanz (Stichwort „*new green economy*“). Ein Beispiel stellt dabei die Abwicklung von Zahlungsverkehr mit Hilfe von Echtzeitüberweisungen[2] dar, hierbei gilt es hinsichtlich schwankenden Zahlungsverkehrs über einen gewissen Zeitraum das laufende System den Umständen angepasst zu skalieren. Des Weiteren verändert dieser Technologiezweig die Art des produktiven Deployments. Hierbei möchte sich das Unternehmen ebenfalls von den herkömmlichen Application Servern verabschieden, welche im Vergleich einen relativ hohen Wartungsaufwand mit sich bringen. Dabei soll der Übergang zu den Cloud-Technologien erst mittels einer hauseigenen Lösung erfolgen und im Nachgang auf einen Cloud-Provider wie Amazon (AWS) oder Microsoft (Azure) gesetzt werden. Die behandelten Themen dieser Thesis beziehen sich vor allem auf eine beispielhafte Konfiguration einer hausinternen Lösung, um zu analysieren, inwiefern sich dies überhaupt rentiert.

[1] Linkedin: `https://de.linkedin.com/company/dps_group`
[2] Erklärung, siehe Abschnitt 4.2.1

Neben diesen vor allem an das Kapital der Firma gebundenen Gründen steht auch die Wartbarkeit der implementierten Lösungen im Fokus. Mittels lose gekoppelter Elemente verschwimmt die Grenze zwischen Operations- und Entwicklungsabteilung. Projekte erleben weniger Handoffs zwischen Teams, ein Entwickler betreut ein Projekt bis in die Deploymentphase, was vorher in dem Maße nicht möglich war. Abhängig von Abstraktion der Cloud Technologie ermöglicht die lose Kopplung der Komponenten eine geordnetere Arbeitsweise, die eine verbesserte Codequalität mit sich bringen kann.

Ein Fokus, welchen ich mir in dieser Thesis im Detail anschauen möchte, betrifft das Start-up-Verhalten einer Anwendung in genau solch einer virtualisierten Umgebung. Denn unter den genannten Effizienzgesichtspunkten möchte man heutzutage auch nötige Ressourcen vor allem auf Anfrage verwenden und nicht mehr rund um die Uhr laufen lassen, selbst wenn dies zu einem gegebenen Zeitpunkt eigentlich gar nicht nötig wäre. Um das Prinzip der Ressourcennutzung auf Anfrage etwas anschaulicher zu gestalten, werde ich eine vereinfachte Kopie einer realen Anwendung vom Unternehmen nachbauen und hinsichtlich der Startzeiten von Container-Instanzen untersuchen. Der Prototyp setzt sich hierbei aus verschiedenen Logikkomponenten zusammen. Diese Komponenten ermöglichen eine detaillierte Analyse des Prototypen hinsichtlich der genannten Effiziensgesichtspunkte und gewährt die Austauschbarkeit bestimmter Module, um verschiedene Technologien gegenüberzustellen.

beinhaltet verschiedene Komponenten zur Abarbeitung der Logik, um im Nachhinein auf die Performanz hinsichtlich der genannten Effiziensgesichtspunkte zu veranschaulichen und zwischen den verwendeten Technologien zu vergleichen.

2 Zielsetzung

Nachdem im letzten Kapitel ein kurzer Einblick in die Motivation für die Portierung alter JEE Technologie in moderner IT-Infrastruktur beschrieben wurde, wird im Folgenden darauf eingegangen, welcher wesentlichen Problemstellung sich die DPS gegenüber konfrontiert sieht und wie ich versuchen werde dies zu untersuchen.

2.1 Begriffserklärungen

Cloudtechnologie Der Begriff „*Cloud-Infrastruktur*“ wird oft auch unter dem Namen „*Infrastructure-as-a-Service (IaaS)*“ vermarktet. Er bezieht sich auf das Abstrahieren der Unternehmensinfrastruktur von konkreter Hardware. Es wird generell in drei verschiedene Kategorien unterschieden [BA18, Seite 54]:

1. *Private Cloud:* Hauseigene Hardware wird virtualisiert in eigenen Rechenzentren betrieben. Hierbei werden die Server beispielsweise als virtuelle Maschine eingerichtet, anstatt auf einer „*Baremetal-Lösung*“ zu basieren, bei der keinerlei Abstrahierung verbaut wurde. Die Grenze zur traditionellen Infrastruktur wird hierbei dadurch gezogen, wie diese Infrastruktur verwaltet wird. Wenn dies durch API-Schnittstellen oder SDKs wie zum Beispiel VMware oder vSphere geschieht, spricht man von der *Cloud*.
2. *Public Cloud:* Bei der privaten Cloud ist das Unternehmen allerdings noch selbst für die Wartung und Instandhaltung der tatsächlich involvierten Hardware zuständig. Bei der Public-Cloud wird dies an externe Dienstleister wie zum Beispiel Amazon (AWS) oder Microsoft (Azure) ausgelagert. Hierbei kommt insbesondere der Begriff *IaaS* zum Tragen, da diese Dienstleistung als ein Service betrachtet wird, welcher vom Benutzer nur noch verwendet wird, dessen Ausführung allerdings völlig irrelevant ist.
3. *Hybrid Cloud:* dieser Begriff beschreibt eine Mischung der beiden Varianten.

Komponentenstack Dieser Begriff beschreibt eine Anwendung, die sich aus einer Ansammlung von Komponenten zusammensetzt. Diese einzelnen Komponenten laufen typischerweise jeweils in einem Container, der wiederum von einer Containerisierungsplattform verwaltet wird.

Orchestrierung Dieser Begriff bezieht sich auf das Verwalten von Komponenten eines Systems. Hierbei ist der komplette Lebenszyklus einer Anwendung zu betrachten. Es wird sowohl die Initialisierung, die Ressourcennutzung zur Laufzeit als auch das kontrollierte Herunterfahren der Komponente betrachtet. Obwohl dieser Begriff oftmals im Kontext einer containerisierten Anwendung beziehungsweise eines Komponentenstacks verwendet wird, trifft er ebenfalls auf andere Bereiche. Beispielsweise verwaltet / orchestriert ein IoC Container[1] ebenfalls die betreffenden System-Komponenten.

[1] Inversion of Control: Das Framework übernimmt die Verwaltung bestimmter Objektinstanzen

Backend-Technologie Serverseitige Komponenten basieren auf bestimmten Technologien. Dies können einzelne Programmiersprachen sein, in der Regel sind jedoch unterschiedliche Frameworks gemeint, die verwendet werden können, um die geforderte Business-Logik zu implementieren.

Serverless Functions Serverless Umgebungen bilden eine weitere Abstraktionsstufe zu IaaS Plattformen und werden auch als „*Function-as-a-Service*“ Technologien bezeichnet. Genau wie schon bei IaaS Plattformen gibt es eine Aufteilung zwischen der tatsächlichen Infrastruktur und der virtualisierten Umgebung. Zusätzlich wird jedoch die Funktionalität des automatischen Verwaltens von benötigten Instanzen ermöglicht, hierbei werden also Instanzen je nach Bedarf hoch oder heruntergefahren [BA18, Seite 70 ff.].

2.2 Kernziele

Da das Unternehmen im Banking Bereich tätig ist, stellt eine hohe Verfügbarkeit der Komponenten eines der wesentlichen Kernziele dar. Auf die neuen Technologien bezogen bedeutet dies, dass es zu ermitteln gilt, wie viel Zeit ein Skalierungsprozess in einem orchestrierten Komponentenstack im Detail braucht. Dazu müssen diverse Metriken erhoben werden, welche im Kapitel 4.1 beschrieben werden. Wichtig an dieser Stelle: Es soll vor allem untersucht werden, inwiefern eine Java-Anwendung als cloudfähig bezeichnet werden kann. Denn wenn die Initialisierungsphase zu viel Zeit in Anspruch nimmt, werden Timeouts riskiert, was es unter allen Umständen zu vermeiden gilt. Eine Alternative versprechen serverless Technologien, welche sich vor allem durch ihre minimale Startzeit für diesen Usecase qualifizieren.

Zusammengefassung:

- Cloudfähigkeit zweier moderner Technologien ermitteln
- Dabei Verfügbarkeit sowie Skalierungsdauer prüfen

2.3 Lösungsweg

Um festzustellen, wie gut die jeweils betrachtete Backend-Technologie im Endeffekt für den Betrieb in einer containerbasierten Anwendung tatsächlich geeignet ist, wird ein fiktiver Workflow mit Hilfe eines Komponentenstacks implementiert, der in der Lage ist, verschiedene Lastszenarien abarbeiten zu können. Hierbei ist es dem Benutzer möglich, testweise zur Laufzeit zwischen den verwendeten Backend-Technologien zu wechseln, um einen entsprechenden Vergleich anstellen zu können. Es werden diverse Skripte bereitgestellt, die es dem Benutzer erlauben, beliebig komplexe Lasten zu simulieren. Um eine Auswertung auch grafisch aufarbeiten zu können, wird der Stack ebenfalls mit Komponenten zur Visualisierung ausgestattet. Diese Visualisierungs-Elemente greifen dabei nicht in den Ablauf des restlichen Stacks ein, sondern kommunizieren lediglich mit bereits bestehenden Schnittstellen um einen möglichst ungestörten Ablauf und damit unverfälschte Datensätze zu garantieren.

Der wesentliche Usecase des Systems besteht in der Verarbeitung von generierten Payments, welche nach mehreren fiktiven Arbeitsschritten verschiedenste Datensätze generieren. Der genaue Gegenstand der Bearbeitung ist allerdings eher nebensächlich, da er so gut wie keinen Einfluss auf die für diesen Usecase interessanten Metriken besitzt.

3 Ist-Analyse

Nachdem im letzten Kapitel ein kurzer Überblick über die bestehende Problemstellung sowie den geplanten Lösungsweg gegeben wurde, folgt ein Einblick in die bestehenden Technologien um die Vorteile modernerer Technologien zu beleuchten.

3.1 JBoss (Microprofile)

Die aktuelle Verarbeitung von Payments innerhalb der Anwendung läuft in der Produktivumgebung auf vier Instanzen des kommerziellen Applikation-Servers „*JBoss*“. Im Development wird hierbei eine Open-Source Variante namens „*Wildfly*“ verwendet. In diesen Application Servern werden *.war* Dateien deployed, welche den ausführbaren Code der Anwendung beinhalten. Der Application Server bietet dem Backend der Anwendung eine Laufzeitumgebung, in der die Anwendung ausgeführt werden kann. JBoss bietet nun standardisierte Schnittstellen nach dem jeweils festgelegten Java-Enterprise-Standard, um zum Beispiel die Kommunikation mit der Außenwelt zu ermöglichen oder der Anwendung eine Persistenzschicht zur Verfügung zu stellen. Außerdem laufen die Instanzen im *Microprofile* Modus, der es möglich macht, dass bestimmte Konfigurationsparameter ausgelagert werden. So können Applikationen auf unterschiedlichen Systemen deployed werden, ohne komplett neu gebaut zu werden (siehe [mic]). Um eine gleichmäßige Aufteilung der Last zu gewährleisten, teilt ein sogenannter „*Load Balancer*“ die eingehenden Nachrichten den entsprechenden Instanzen zu (siehe Abbildung 3.1 JBoss Systemaufbau). Jede der vorhandenen Instanzen besitzt eine minimale sowie maximale Anzahl an parallel ausführbaren Prozessen. Diese Angaben werden auch „*max. / min. Poolsize*“ genannt. Eine minimale Poolsize muss gegeben sein, um sicherzustellen, dass eine gewisse Grundlast, falls nötig, sofort bearbeitet werden kann, daher darf diese Anzahl auch nicht Null betragen. Die maximale Poolsize stellt sicher, dass es zu keiner Überlastung des Systems

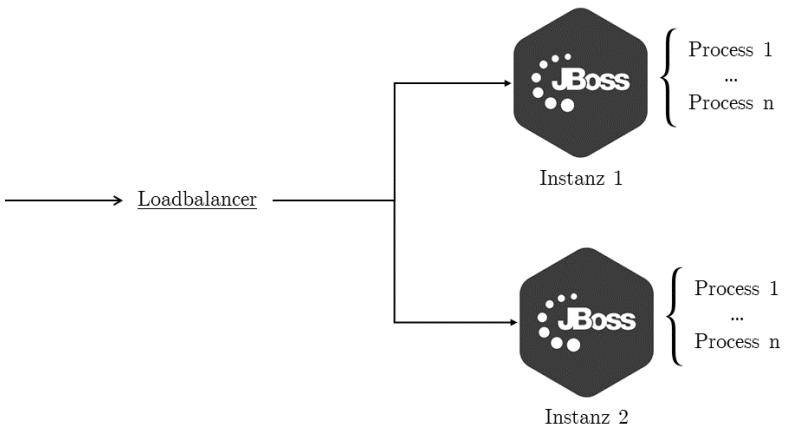


Abbildung 3.1: JBoss Systemaufbau

kommt. Wenn eine Instanz bereits mit der maximale Anzahl an Prozessen arbeitet, wird dies dem Loadbalancer signalisiert und welcher wiederum der entsprechenden Komponente in diesem Zeitraum keine weiteren Nachrichten mehr zuteilt. Um zu gewährleisten, dass die Nachrichten nicht verloren gehen, werden sie in eine Warteschlange („*Request Queue*“) geschrieben, welche lediglich dazu gedacht ist, den Overhead abzuspeichern. Wie die Daten im Detail verarbeitet werden, ist für die weitere Betrachtung irrelevant und wird daher nicht weiter erläutert. Eine vereinfachte Implementierung wird im späteren Prototypen dennoch vorhanden sein. Die dazugehörige Beschreibung befindet sich in Abschnitt 4.2.1 Festlegung eines fiktiven Workflows.

3.2 Probleme

Bezüglich der Systemarchitektur einer klassischen Java-Enterprise Anwendung handelt es sich in der Regel um eine monolithische Struktur. Die ausführbare Anwendung wird mit Hilfe von einer einzigen Codebase entwickelt, gebaut und deployed. Dem gegenüber steht die *service-basierte* Designstruktur, dabei handelt es sich um eine mittels Containern ermöglichten Umgebung. Weshalb monolithische Systeme den heutigen Anforderungen in manchen Aspekten nicht mehr genügen und inwiefern eine service-basierte Struktur hierbei Abhilfe schafft, wird im folgenden näher erläutert [BA18, Seite 42 ff.].

3.2.1 Skalierte Entwicklung

Wenn viele Entwickler an derselben Anwendung arbeiten, muss jeder einzelne zumindest ein grundlegendes Verständnis der gesamten Codebase besitzen. Außerdem kommt es während der Entwicklungszeit zwangsweise zu einer Reihe von Mergekonflikten. Mit den richtigen Designprinzipien bezüglich der losen Kopplung der einzelnen Komponenten sowie der Modularisierung des Systems, sollte dies zwar umgangen werden können, in der Realität entwickelt sich ein solches System allerdings ständig weiter. Da das System diese Prinzipien allerdings nicht explizit erzwingt, werden diese Prinzipien häufig mit der Zeit vernachlässigt. Funktionalität aus dem Modul zu extrahieren und in eigenständige Services auszulagern führt hierbei zu einem effizienteren Workflow. Insbesondere hinsichtlich der Arbeitsaufteilung ermöglicht der service-basierte Aufbau die Implementierung klarer Schnittstellen, wodurch viele Entwickler parallel arbeiten können.

3.2.2 Unabhängiges Deployment

Eine monolithische Applikation wird mit Hilfe eines einzelnen Artifacts deployed. Dies stellt insbesondere dann ein Problem dar, wenn neue Funktionalität implementiert wird oder einzelne Teile der Applikation häufiger verändert werden als andere. Bei allen genannten Szenarien muss stets die gesamte Applikation neu deployed werden, obwohl die Veränderungen nur einzelne Komponenten der Anwendung betreffen. Bei einem stark gekoppelten System muss jede neu hinzugefügte Funktionalität verstärkt getestet werden, da es möglicherweise Abhängigkeiten zwischen Komponenten gibt, die auf den ersten Blick als solche gar nicht erkennbar sind. Der Service-basierte Ansatz bietet hier Abhilfe, da es möglich ist ausschließlich dedizierte Teile des Systems (neu) zu deployen beziehungsweise nur noch diejenigen Komponenten testen zu müssen, welche überhaupt verändert wurden, ohne dabei Risiken einzugehen.

3.2.3 Skalierung einer Produktivumgebung

Eine Skalierung ist mit monolithischen Strukturen nur dadurch möglich, den ausführbaren Code auf zusätzlichen Servern zu deployen, dies wird auch *horizontale Skalierung genannt*. Jede dieser Kopien nutzt die gleiche Ressourcenanzahl, was es zu einem ineffizienten Design macht, da sie sich nicht dynamisch der gegebenen Last anpassen. Tatsächlich ist ein jedoch ein viel größeres Problem, wenn ein bereits aufgeteiltes System an seine Kapazitätsgrenze stößt, denn Java-Anwendungen besitzen eine relativ lange Initialisierungsphase. Um den heutigen Anforderungen der dynamischen Skalierung gerecht zu werden, werden Ressourcen auf Abruf gebraucht. Die Systeme sollen so schnell wie möglich verfügbar sein. Außerdem ist es bei monolithischen Systemen ausgeschlossen nur die betreffenden Komponenten des Systems zu skalieren, die durch eine erhöhte Last überhaupt betroffen sind.

4

Vorgehensmodell

Nachdem im letzten Kapitel der aktuelle Aufbau der Payment-Anwendungen sowie die durch die monolithische Struktur hervorgebrachten Probleme erläutert wurden, wird im Folgenden auf die Untersuchung der Container-Startzeit eingegangen.

4.1 Anforderungen an Daten zur Messung des Startup-Verhaltens von Containern

Der Fokus des Projekts liegt auf der Ermittlung der Cloudfähigkeit der unterschiedlichen System-Komponenten. Um dies festzustellen, muss im Vorwege eine Klassifizierung der Anforderungen stattfinden. Anhand dieser Klassifizierung werden die relevanten Messdaten beziehungsweise Metriken ermittelt. Die internationale Organisation für Normung (*ISO*) verfasste für die generelle Klassifizierung von qualitativen Anforderungen an Softwareprodukte bereits einen ausführlichen Kriterienkatalog, welcher im Folgenden als Grundlage für die beschriebene Auswahl verwendet wird (siehe ISO/IEC 25010:2011 [iso11], Abbildung 4.1 iso25010).

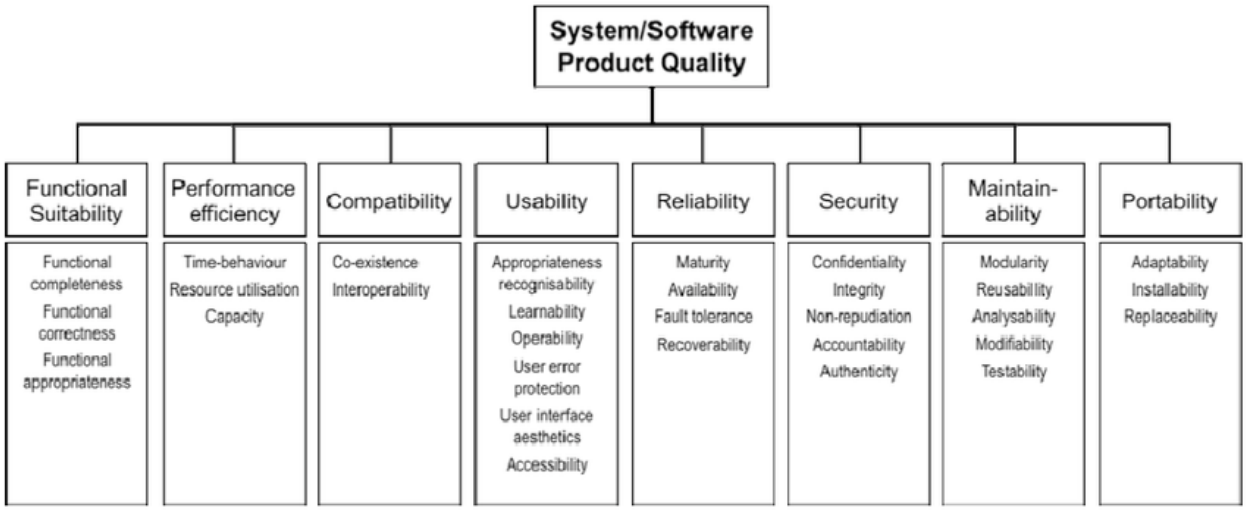


Abbildung 4.1: ISO 25010 [iso]

Die ISO-Norm 25010 besteht aus acht Hauptkategorien und 31 Unterkriterien, wobei lediglich die erste Hauptkategorie der „*Funktionalität*“ dabei als *funktionale Qualitätseigenschaft* (engl. „*functional requirement*“) auftritt. Die restlichen Hauptkriterien lassen sich der Kategorie der *non-functional requirements* (siehe nächster Absatz) zuordnen. Diese Unterscheidung dient als eine erste Abstraktionsschicht. Functional requirements beschreiben hierbei die Korrektheit des Produktes [BA18, Seite 335]. „*Functional requirements specifies a function that a system or system component*

must be able to perform“ [Eid05, Seite 19]. So lässt sich feststellen ob die Applikation den inhaltlichen Ansprüchen, sowohl aus technischer als auch aus Unternehmenssicht, genügen. Hierbei wird auf eine Vielzahl unterschiedlicher Tests zurückgegriffen, welche sich auf breit gefächerte Abstraktionstiefen etc. beziehen lassen (siehe Abbildung 4.2 Agile Testing Quadrants). Je nach Teilbereich lassen sich hierbei einzelne Gebiete automatisiert oder mittels anderweitiger Werkzeuge testen. Diese Tests sind in einer Produktivumgebung unabdingbar, für die Messdatenerhebung zur Beurteilung der Cloudfähigkeit können diese allerdings vernachlässigt werden.

Zur Eingrenzung der Anforderungen an die Messdaten sind primär die *Non-Functional Requirements* von Bedeutung. Sie beschreiben nicht *was* vom System geleistet werden muss, sondern *wie* dies geschehen soll, deshalb werden die Kriterien unter anderem auch *system-quality attributes* genannt [BA18, Seite 335]. Im Folgenden werden die für das Projekt relevanten Kriterien zusammengefasst dargestellt.

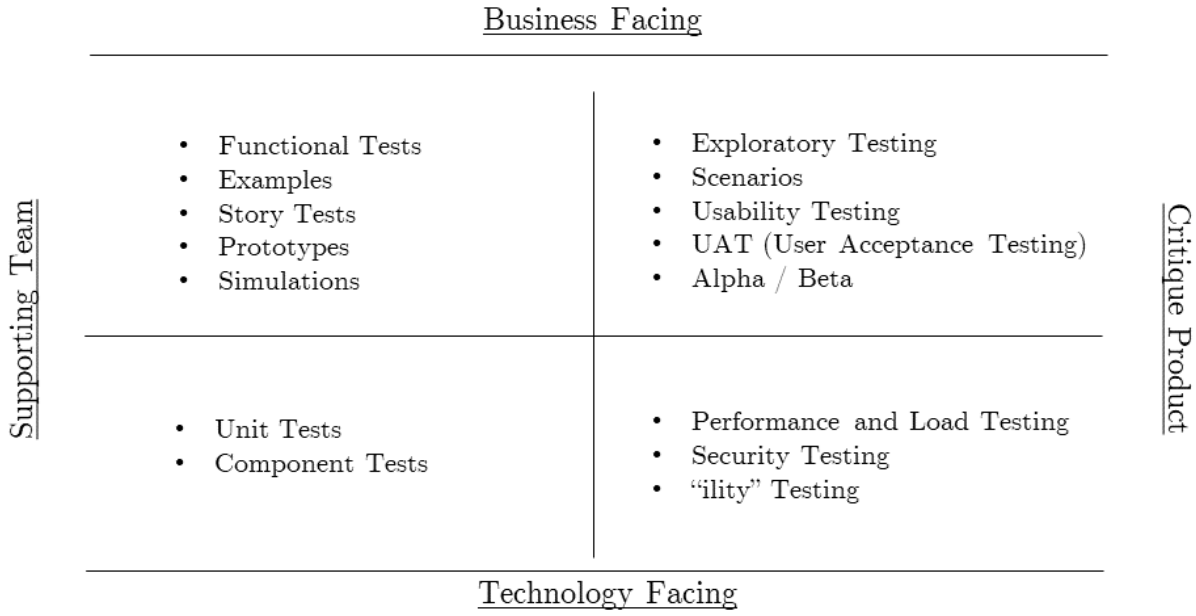


Abbildung 4.2: Agile Testing Quadrants

Leistungsfähigkeit

„Performance of a product or an app defines how a product/app is performing or behaving as compared to its expected behavior“ [nfr]. Die Leistungsfähigkeit einer Anwendung setzt sich aus den Punkten *Zeitverhalten*, *Ressourcennutzung* sowie *Kapazität* zusammen. Das Zeitverhalten wird vor allem durch die Reaktionszeit der Applikation geprägt. Es wird gemessen, wie schnell auf Benutzereingaben eingegangen werden kann. Eng verwandt ist dabei auch der Begriff der *Latenzzeit*, da dieser beschreibt, wie lange die Beantwortung einer Anfrage in der Praxis letztendlich dauert. Die Reaktionszeit dient hierbei als ein Threshold der nicht überschritten werden sollte. Typischerweise wird die Reaktionszeit pro Client festgelegt, es ist jedoch auch möglich, dies für eine Menge von mehreren Clients festzulegen. Ein Beispiel wäre, dass für einen bestimmten Anteil der Interaktionen eine maximale Reaktionszeit von n Sekunden festgelegt wird. Aber auch das zweite Unterkriterium der *Ressourcennutzung* darf nicht vernachlässigt werden, so dürfen beispielsweise Kommunikationsschnittstellen nicht überlastet werden und Connectionpools nicht überlaufen, da es sonst zu Timeouts kommen könnte. Die Kapazität beschreibt zum Beispiel die Anzahl von möglichen

parallelen Anfragen oder wie viele Nachrichten übermittelt, beziehungsweise vom System gespeichert werden können.

Ein weiterer Unterpunkt, welchen man der Leistungsfähigkeit zuschreiben kann, ist der des *Durchsatzes*. Hierunter wird die Anzahl der verarbeiteten Nachrichten über einen definierten Zeitraum verstanden. Dieser Begriff wird in der ISO Norm zwar nicht erwähnt, stellt hierbei jedoch eine weiteres Kriterium dar, gerade im Hinblick die Banking-Anwendung. Die verarbeiteten Daten können sich hierbei auch auf Verschiedenes beziehen. Hiermit kann zum Beispiel die zeitliche Aufteilung von Anfragen an eine API sowie deren Rückmeldung gemeint sein. Oder sie definieren eine bestimmte Anzahl von Nachrichten bezüglich eines Brokers, die durch das System gereicht werden. Es kann sich aber auch ganz einfach auf die Anzahl der Datenbankanfragen etc. beziehen. Diese Art der Performanz wird in der Regel mittels spezieller Tools, wie zum Beispiel „*Gatling*“ oder „*JMeter*“ in sogenannten *Performance* oder *Load Tests* validiert. Zusammenfassend stellt die Leistungsfähigkeit mit ihren Subkategorien eine der wohl wichtigsten Kriterien dar.

Skalierbarkeit

Ein weiteres wichtiges Kriterium, das jedoch nur indirekt in der ISO-Norm 25010 Norm auftritt, ist die Skalierbarkeit des Systems[1]. Bei der Beurteilung gilt es in erster Linie zwischen dem Skalieren der *Hardware* sowie der *Software* zu unterscheiden [nfr]. Die Hardwareskalierung könnte man grob dem Unterkriterium „*Ressourcennutzung*“ zuordnen. Denn in einer modernen containerisierten Cloud-Umgebung soll es möglich sein, die einzelnen Komponenten nicht nur auf einem Gerät beliebig zu skalieren, sondern, falls die aktuelle Hardware an ihre Kapazitätsgrenzen stößt, auch zusätzliche Hardwarekomponenten hinzuziehen zu können. Inwiefern dies automatisiert oder manuell geschieht, lässt sich lediglich fallabhängig bestimmen, es sollte jedoch evaluiert werden, ob dies generell ermöglicht wird. Metriken zum Ermitteln der Skalierungszeitpunkte könnten zum Beispiel die Anzahl der parallel arbeitenden Benutzer der Applikation (engl. „*concurrent users*“), die Latenzzeit oder die CPU-Auslastung der beteiligten Maschinen sein. Relevant ist jedoch ebenfalls, dass eine Applikation nicht nur von aktiv eingehenden Ereignissen skaliert wird, sondern auch ein Zeitverhalten aufweist. Da die Umwelteinflüsse durchaus fehlerbehaftet sein können und die Verbindung der Komponenten unterbrochen werden kann, kann hierbei zusätzlich ein internes Skalierungsverhalten verwendet werden.

Bei der Hardwareskalierung gibt es zwei wesentliche Arten, die *vertikale* und die *horizontale Skalierung*. Der Begriff der *vertikalen Skalierung* (engl. „*scale up*“ Skalierung) beschreibt das Modifizieren der servereigenen Ressourcen [nfr, Absatz *Scalability*]. Dazu können zum Beispiel die Anzahl der CPU-Kerne oder der verfügbare Speicher gehören. Dieses Verfahren wird in monolithischen Strukturen mit älteren Systemen verwendet, die nicht darauf ausgelegt sind, sich auf mehrere Server zu verteilen. Dagegen steht der Begriff der *horizontalen Skalierung* (engl. „*scale out*“ Skalierung), welcher gerade dieses Auslagern bestimmter Komponenten oder Prozessinstanzen auf neue Hardware beschreibt. Letzterer Ansatz wird in der Cloud-Technologie verfolgt und nicht nur auf unternehmensinternen Systemen, sondern auch großen *public cloud* Dienstleistern wie zum Beispiel Amazon (AWS) oder Microsoft (Azure), verwendet.

Dennoch muss die Software auf eine Skalierung der Hardware ausgelegt sein. Manche Probleme lassen sich mit mehr Rechenperformance nicht lösen. Wenn beispielsweise die Übertragungszeiten bestimmter Kommunikationsschnittstellen einen Flaschenhals darstellen, ändert eine Skalierung hieran nichts. Alter Legacy-Code könnte ebenfalls Performanceeinbußen mit sich ziehen. Ohne ein angebrachtes Session-Management wird das Skalieren der Hardware dieses Problem nicht

[1] siehe „*resource utilisatzion*“ der Hauptkategorie „*performance effeciency*“ der vorgestellten ISO-Norm

lösen können. Diese Grundfunktionalitäten werden heutzutage jedoch von fast allen Frameworks standardmäßig mitgeliefert, dennoch gilt es dies im Vorwege zu validieren.

Kompatibilität

Dieses Hauptkriterium beschreibt, wie ausgeprägt die nebenläufige Verarbeitung des Systems ist. Ist es möglich, dass mehrere Komponenten gleichzeitig arbeiten können, ohne sich gegenseitig zu behindern? Gerade in einer containerisierten Umgebung ist dies zu ermitteln. Der Unterbegriff *Ko-Existenz* definiert dieses Verhalten. Der zweite Unterbegriff der „*Interoperabilität*“ grenzt hierbei die Kommunikation von der Nebenläufigkeit ab und beschreibt, inwiefern die verschiedenen Komponenten in der Lage sind untereinander Daten auszutauschen. Wie genau dieser Datenaustausch aussieht, kann sich von System zu System unterscheiden. Zusammenfassend lässt sich sagen, dass diese Kriterien in einer containerbasierten Anwendung zwar gelten müssen, sie allerdings dennoch derartig abstrakt gehalten sind, dass es schwieriger ist hier messbare Metriken zu generieren. Entweder das System arbeitet hinsichtlich dieser Kriterien fehlerfrei oder es ist fehlerbehaftet, einen messbaren Mittelweg gibt es nicht direkt.

Zuverlässigkeit

Ein weiteres Hauptkriterium der ISO-Norm 25010 bezieht sich auf die *Zuverlässigkeit* eines Systems. Es beschreibt die garantierte Laufzeit einer Applikation [nfr]. Dies kann auf verschiedene Weisen festgehalten werden. Eine typische Angabe wäre zum Beispiel der Anteil der Tage im Jahr, Tage pro Monat, oder Stunden am Tag, die ein Service ansprechbar ist. Allgemein lässt sich diese Angabe als Metrik der *Verfügbarkeit* charakterisieren. In einer containerisierten Umgebung kommt es immer wieder zu Ausfällen einzelner Systeme, was es bei diesem Punkt besonders zu beachten gilt. Das Gesamtsystem soll hiervon jedoch unberührt bleiben. Dieses Prinzip wird in heutigen Systemen auch „*Design for failure*“ genannt [BA18, Seite 267].

Das Hauptkriterium setzt sich unter anderem aus dem Unterkriterium der *Reife* zusammen. Hierbei sind diejenigen Anforderungen gemeint, die in einer normalen Prozessabarbeitungsphase ohne besondere Ereignisse gegeben sein müssen. Die Anforderungen können sich zum Beispiel auf die Anzahl der zu verarbeitenden Nachrichten oder der Komplexität der zu bewältigenden Aufgaben zur gängigen Betriebszeit handeln. Außerdem wird eine gewisse *Fehlertoleranz* vom System erwartet. Um dies zu testen, genügt es bereits dokumentierte, allerdings fehlerbehaftete, Benutzereingaben in einer Testumgebung zu tätigen und das Systemverhalten zu analysieren. Eine standardisierte Metrik gibt es auch hier nicht. Das letzte Unterkriterium der *Zuverlässigkeit* stellt die *Wiederherstellbarkeit* nach einem beschriebenen Fehlerfall dar, falls es doch einmal zu einem Ausfall einzelner Komponenten kommen sollte. Generell gilt: auch dieses Hauptkriterium gibt Aufschluss über die Eignung gewisser technologischer Ansätze zum Arbeiten als Cloudkomponente, dennoch beschreibt sie die relevanten Umstände in manchen Teilen zu abstrakt, als das es Gegenstand dieser Thesis sein könnte.

Irrelevante Themenfelder

Weitere Hauptkriterien der ISO-Norm betreffen die *Nutzbarkeit* sowie die *Sicherheit* der Anwendung. Da der Fokus des Projekts aber darauf liegt die Skalierbarkeit und Performance des Systems zu testen, können diese Punkte jedoch vernachlässigt werden. Die Themengebiete der *Wartbarkeit* sowie der *Übertragbarkeit* sind in einer containerisierten Plattform zwar ebenfalls bedeutsam, wurden wegen ihrer schwierigen Messbarkeit allerdings im Rahmen dieser Thesis verworfen.

Zusammenfassung

Im letzten Abschnitt wurde auf die verschiedenen Kriterien der ISO-Norm 25010 eingegangen sowie einige weitere relevante Kriterien hinsichtlich der nicht funktionalen Anforderungen hinzugefügt. Hierbei wurde bezüglich der Anwendung der Kriterien auf den zu entwickelnden Prototypen bereits eine minimale Vorauswahl getroffen, wobei erst im nächsten Abschnitt eine letztendliche Festlegung der betrachteten Kriterien sowie deren genaue Erfassung erörtert wird. Im Folgenden werden die wesentlichen Kriterien noch einmal zusammengefasst.

- Lediglich *nicht funktionale Anforderungskriterien* für dieses Projekt von Bedeutung
- Leistungsfähigkeit
  - Zeitverhalten (Reaktionszeit)
  - Ermittlung von Bottlenecks
  - Leistungsfähigkeit (Durchsatz / Latenzzeit)
- Skalierbarkeit
  - Evaluierung, ob automatisiert oder manuell gesteuert werden kann
  - Reaktion auf externe sowie interne Ereignisse
  - Ermitteln, welche Skalierungsart angebracht (*scale out* gegenüber *scale up*)
  - Ermitteln, ob Skalierung überhaupt zielführend
  - Zeitverhalten relevant
- Kompatibilität
  - Ko-Existenz (Nebenläufigkeit überhaupt möglich?)
  - Interoperabilität (Kommunikation zwischen parallelen Komponenten)
- Zuverlässigkeit
  - Garantierte Laufzeit
  - *Design for failure* Prinzip erfüllt?
  - Wiederherstellbarkeit
- Irrelevante sowie nicht betrachtete Themenfelder
  - Irrelevant
    - Nutzbarkeit
    - Sicherheit
  - out of scope
    - Wartbarkeit
    - Übertragbarkeit

4.2 Anforderungen an den Prototypen

Bei diesem Projekt soll sowohl eine Orchestrierungsplattform evaluiert als auch ein entsprechendes Framework hinsichtlich der Eignung im Kontext der Kernziele der Thesis verglichen werden (siehe Abschnitt 2.2 Kernziele).

Hinsichtlich der Skalierung soll es beispielsweise möglich sein, anhand festgelegter Regeln eine automatische *horizontale* Skalierung bestimmter Komponenten vorzunehmen. Dieses Verhalten soll sowohl in hinterlegten Datensätzen als auch mithilfe eines eigens dafür eingerichteten Dashboards nachvollziehbar dargestellt werden. Auf welche Technologien sowohl bei der Skalierung als auch der Darstellung beziehungsweise bei der Berechnung der Metriken zurückgegriffen wird, wird vom Arbeitgeber nicht vorgegeben. Lediglich hinsichtlich der Komponenten zur Verarbeitung der Businesslogik wurde eine Vorauswahl getroffen.

4.2.1 Festlegung eines fiktiven Workflows

Der Prototyp soll eine bereits bestehende Applikation nachbilden, welche die Zahlungsabwicklung sogenannter Echtzeitüberweisungen durchführt. „*Die Echtzeitüberweisung (englisch Instant Payment) ist eine Zahlungsart, bei der Guthaben innerhalb weniger Sekunden dem Empfänger final gutgeschrieben werden. Inzwischen gibt es in zahlreichen Ländern weltweit Echtzeitzahlungssysteme. Im einheitlichen Euro-Zahlungsverkehrsraum, dem sogenannten Sepa-Raum, gibt es seit dem 21. November 2017 einen entsprechenden Standard: Die Sepa Instant Payments (SCTInst) sind 24 Stunden an 365 Tagen im Jahr verfügbar. Die Gutschrift muss in maximal 10 Sekunden erfolgen, faktisch werden die meisten Transaktionen allerdings in weniger als 3 Sekunden durchgeführt*“ [ins].

Da der Fokus auf der Untersuchung der Start-up-Zeit der Komponenten liegt, wird lediglich eine minimale, beispielhafte Implementierung erfolgen, welche die Arbeitsschritte der eigentlichen Applikation vereinfacht darstellen soll. Es soll im Rahmen dieser Arbeit eher als ein *proof of concept* gelten. Allerdings werden im System konkrete Nachrichten im XML Format vermittelt, welche einer XSD-Spezifikation folgen wie sie im realen Umfeld ebenfalls genutzt wird (siehe Listing 4.1 Payment Format).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Document xmlns="urn:iso:std:iso:20022:tech:xsd:pain.001.001.09">

  ...

  <DbtrAcct>
    <Id>
        <IBAN>AT331200000696200104</IBAN>
    </Id>
  </DbtrAcct>
  <Cdtr>
    <Nm>DE NAME</Nm>
    <PstlAdr>
        <Ctry>DE</Ctry>
        <AdrLine>Point Courrier 201</AdrLine>
        <AdrLine>IT 91191 Gif sur Yvette</AdrLine>
    </PstlAdr>
  </Cdtr>

  ...

</Document>
```

Listing 4.1: Payment Format

Sobald eine neue Nachricht eingetroffen ist, sollen drei Arbeitsschritte ausgeführt werden:

1. Es soll geprüft werden, ob das eingegangene XML der in den Metainformationen angegebenen XSD-Spezifikation folgt. Wenn dies nicht der Fall sein sollte, wird die Nachricht zwar bestätigt (engl. *acknowledged*), sodass sie aus der Eingangsqueue im Message Broker entfernt, allerdings bei der weiteren Verarbeitung ignoriert wird.
2. Falls es sich um valides XML-Format handelt, wird ein Feld aus dem XML-Inhalt ausgelesen.
3. In einem letzten Schritt wird das extrahierte Element in eine Datenbank geschrieben, damit auch eine Persistenz-Operation in die Verarbeitungszeit einfließt.

4.2.2 Artefaktbasierte Technologie

Wie bereits angedeutet, wird die aktuell laufende Java-Enterprise Applikation mit einer Implementierung basierend auf dem moderneren Spring Boot Framework ersetzt. Spring Boot bietet im Kern eine sehr ähnliche Funktion und kann in gewisser Hinsicht als eine modernere Version des JEE Standards angesehen werden. Der Austausch ist vor allem hinsichtlich der zu gewinnenden Messwerte / Ergebnisse sinnvoll, um einen aktuelleren Referenzpunkt bezüglich der service-basierten Struktur zu erlangen.

4.2.3 Skriptbasierte Technologie

Um bezüglich des Spring Boot Frameworks einen sinnvollen Vergleichspunkt anzusetzen, wird dem eine Skript-Technologie gegenübergestellt, da diese im *Cloud Native* Bereich ebenfalls sehr häufig verwendet werden. „*Eine native Cloud-Anwendung (NCA) ist ein Programm, das speziell für eine Cloud-Computing-Architektur entwickelt wurde. NCA nutzen die Vorteile von Cloud-Computing-Frameworks, die aus lose gekoppelten Cloud-Services zusammengesetzt sind.*“ [def]. Mit dieser doch abstrakt gehaltenen Definition sind hinsichtlich der verarbeitenden Komponenten in erster Linie

die modernen Skript-Frameworks gemeint. Hierbei existieren mehrere Alternativen, wo es seitens des Unternehmens ebenfalls keinerlei genaue Vorgaben hinsichtlich des spezifischen Frameworks beziehungsweise der zu nutzenden Technologie gab. Um allerdings ein möglichst repräsentatives Bild zu schaffen, wurde hierbei das Node.js Backend-Framework ausgewählt, da dieses als eines der am weitest verbreiteten Frameworks in diesem Bereich gilt. Sowohl die Node.js als auch Spring Boot Komponenten sollen jeweils die beschriebenen Arbeitsschritte implementieren.

4.3 Anforderungen an die Containerplattform / Orchestrierungsplattform

Unter dem Begriff des „*Containers*“ wird in der Entwickler-Community meist ein „*Docker-Container*“ verstanden. Allgemein beschreibt dieser Begriff in der Softwareentwicklung ein abstraktes Konzept auf Betriebssystem-Level zur Virtualisierung oder Prozessisolation. Es gibt zahlreiche Implementierungen wie beispielsweise CoreOS's „*rkt*“ oder Microsoft's „*Hyper-V*“. Docker stellt hierbei lediglich eine weitere Implementierung dieser Technologie dar [BA18, Seite 63 ff.]. Dennoch hat sich Docker gegenüber anderen Implementierungen vor allem über die einfache Benutzerführung sowie die zentralisierten Mechanismen zum *Teilen*, *Pullen* und *Pushen* durchgesetzt. Die generelle Funktionsweise sowie die Vorteile von Docker wurden bereits ausführlich erläutert. Da die Docker Technologie von allen vorgestellten Implementierungen am weitesten verbreitet ist, soll diese auch im Prototypen Verwendung finden.

Eine Containerplattform zeichnet sich allerdings neben der genutzten Implementierung über weitere Komponenten aus. Eine der Wichtigsten stellt hierbei der Orchestrator dar. „*This component is responsible for starting, stopping, and managing the container processes. This technology is often referred to as container infrastructure as a service (CIaaS), and in the Docker ecostystem this is typically provided by Docker Swarm or Kubernetes*“ [BA18, Seite 64 ff.]. Hierbei wird auch die Portabilität sowie die Reproduzierbarkeit eines Systems als eine wesentliche Anforderung vorausgesetzt [doc, Kapitel Orchestration]. Da Kubernetes im Unternehmen bereits Anwendung findet, wurde sich hierbei auf Docker Swarm festgelegt, um ein Demo-Projekt auszuarbeiten, an dem sich in Zukunft andere Projekte orientieren können. Swarm grenzt sich gegenüber anderen Technologien vor allem durch den nativen Support innerhalb des Docker Ökosystems ab. Man benötigt keine weiteren externen Tools, sofern eine Docker Installation vorliegt [Sop16, Seite 10 ff.]. Des Weiteren ist es möglich Images in einem skalierten System zu deployen ohne weitere Anpassungen am Image selbst vornehmen zu müssen. Diese Images können außerdem in Clustern organisiert werden, welche direkt über ein alternatives Tool namens „*Docker Compose*“ konfiguriert werden können.

„*Docker Compose allows the declarative specification of applications/services and associated dependencies and data stores. This pattern does allow for the flexible execution of a collection of dependent services on a local development machine*“ [BA18, Seite 173]. Um diese Spezifikation auch in einer produktiven Umgebung auszuführen, wird auf einen Orchestrator wie zum Beispiel *Docker Swarm* zurückgegriffen. Das Swarm Tool wird hierbei massiv in Produktions-Umgebungen eingesetzt, da es in der Lage ist Tausende von Containern zeitgleich zu verwalten und über einen dezentralen Discovery Service anzusteuern.

Da es im Cluster vermehrt zur Ausführung mehrerer Instanzen des gleichen Images kommen kann, ist es nicht mehr möglich einen Service mit einer spezifischen IP-Adresse zu versehen. Bei untereinander abhängigen Services braucht es einen Kommunikationsmechanismus zwischeneinander. Dieser wird durch einen „*Discovery Service*“ organisiert. Ein solcher Service ist neben dem Vermitteln von Anfragen auch in der Lage Konfigurationsänderungen der einzelnen Services im System zu organisieren sowie die Verfügbarkeit der jeweiligen Instanzen zu prüfen [Sop16, Seite 45 ff.].

4.4 Anforderungen an den Lasttest

Es soll möglich sein, das System in einem einheitlichen Format mit Nachrichten zu versorgen. Hierfür benötigt man eine Benutzerschnittstelle, die es ermöglicht dem Benutzer möglichst aussagekräftige Befehle mit minimalen Konfigurationsaufwand zur Verfügung zu stellen. Diese Befehle müssen anschließend intern wiederum in detaillierte Nachrichten umgewandelt werden, welche durch das System bearbeitet werden können. Hierfür gibt es die Möglichkeit einer Benutzeroberfläche im Browser oder der Festlegung eines Formats im Body einer Http-Anfrage an eine REST-API. Eine Skalierung auf mehrere Systeme sollte einmal testweise erfolgen, für die Generierung der Metriken ist dies allerdings contraproduktiv, da es die Ergebnisse verfälschen könnte. Diese Verfälschung könnte auftreten, da es dem Design der Orchestrierungsplattform geschuldet ist, dass der Anwender wenig Einfluss auf die Ressourcenzuteilung hat. Hierbei kann es passieren, dass eine leistungsschwächere Maschine deutlich weniger Anfragen zugewiesen bekommt. Dies könnte man zwar dadurch umgehen, dass man zwei exakt gleichbestückte Server für den Lasttest verwendet. Hierbei würden im Nachhinein allerdings im besten Fall die gleichen Ergebnisse erlangt werden, wie bei einem Testlauf auf einer einzelnen Maschine.

4.5 Anforderungen die Visualisierung und das Monitoring zur Unterstützung der Auswertung

Es soll möglich sein, Messwerte automatisch vom System generieren zu lassen. Dazu müssen die Datensätze persistent hinterlegt werden, um auch im Nachhinein nachvollziehen zu können wie die Metriken entstanden sind. Außerdem muss eine grafische Aufarbeitung erfolgen. Diese Visualisierung soll in Echtzeit oder einem zeitlich festgelegten Intervall aktualisiert werden können. Bezüglich des genauen Werkzeugs werden keinerlei Vorgaben gegeben, um dem Benutzer jedoch jeglichen Installationsaufwand zu ersparen wäre eine webbasierte Darstellung von Vorteil.

4.6 Zusammenfassung

Es folgt eine kurze Zusammenfassung des letzten Abschnitts.

- Wesentliche Anforderungen Container
  - Portabilität
  - Reproduzierbarkeit
- Wesentliche Anforderungen Orchestrator
  - Automatisierte Wartung
    - Container Updates
    - Austausch fehlerhafter Container
  - ermöglicht Skalierung bestimmter Services
  - Anforderungen an den Lasttest
    - Benutzerschnittstelle zum Übersetzen von Anfragen in Nachrichten, welche vom System interpretiert werden können
    - Testlauf nur auf einer einzelnen Maschine um Ergebnisse nicht zu verfälschen

- Anforderungen Visualisierung / Monitoring
  - Messwerte vom System generieren lassen
  - Datensätze persistent hinterlegen
  - Visualisierung in Echtzeit

5 Problemlösung

5.1 Bestimmung von Daten zur Messung des Start-up-Verhaltens von Containern

Nachdem im letzten Abschnitt auf mögliche Kriterien zur Messung der relevanten Metrikinformationen eingegangen wurde, erfolgt im Folgenden eine spezifische Festlegung auf bestimmte Kriterien, welche auf das zu lösende Problem zugeschnitten werden.

5.1.1 Leistungsfähigkeit

Die wesentlichen Metriken stellen hierbei der Datendurchsatz sowie die Latenzzeit dar. Die Latenzzeit sollte die komplette Pipeline abdecken, also vom Dateneingang durch den Benutzer bis hin zur tatsächlichen Speicherung der Daten in der Persistenzschicht. Die gesamte Strecke lässt sich allerdings auch in noch kleinere Abschnitte unterteilen, um bei unerwarteten Werten festzustellen, an welcher Stelle genau es zu einem Engpass kommt. Der Prototyp soll Daten des Benutzers annehmen, daraus Nachrichten generieren und diese an entsprechende Konsumenten weiterleiten. Die Unterteilung der gesamten Pipeline sollte hierbei zumindest in zwei Abschnitten erfolgen.

- Hierbei ist vor allem von Interesse, ob sich der Zeitrahmen vom Eingang der Daten bis zum Erhalt der Daten durch die Konsumenten während der Skalierungstests in irgendeiner Weise verändert. Dies wird jedoch nicht erwartet, da beide Arten von Testläufen auf dieselbe Eingangsschnittstelle zurückgreifen. Eine Veränderung deutete auf ein tieferliegendes Problem bei der Orchestrierung des Stacks hin.
- Außerdem sollte der Zeitrahmen vom Erhalt der Daten durch die Konsumenten bis hin zur Interaktion mit der Persistenz-Schicht gesondert betrachtet werden. Dieser Abschnitt stellt den variablen Teil der Pipeline dar und ist von größtem Interesse.

Auf Werkzeuge wie zum Beispiel *Gatling*[1] oder *JMeter*[2] muss hierbei verzichtet werden, da diese nicht in der Lage sind in die Pipeline einzugreifen um die einzelnen Abschnitte gesondert betrachten beziehungsweise messen zu können. Die Metriken müssen sowohl mithilfe der kompilierten Arbeitsversion als auch der Skript-Version generiert werden um eine Gegenüberstellung durchführen zu können. Hierbei sollte es außerdem möglich sein, den zeitlichen Verlauf der Metriken darzustellen um nachvollziehen zu können, ob es Phasen gibt, in denen die Bearbeitung von dem berechneten Durchschnittswert abweicht. Je nach Stärke der Abweichung gilt es zu evaluieren, ob zusätzliche statistische Kenngrößen diesbezüglich berechnet werden sollten. Ein weiteres mögliches Kriterium wäre in diesem Zusammenhang zum Beispiel die Standardabweichung. Da die Latenzzeit stark mit dem eigentlichen Datendurchsatz zusammenhängt, ist es nicht nötig, diese noch einmal gesondert zu berechnen um Redundanz zu vermeiden.

[1] `https://gatling.io/`
[2] `https://jmeter.apache.org/`

5.1.2 Skalierbarkeit

Da es sich bei dem Projekt um eine containerisierte Cloud-Anwendung handelt, muss hierbei eine *vertikale Skalierung* stattfinden. Um möglichst akkurate Werte zu erreichen, müssen die zu skalierenden Container in engem Austausch mit dem System stehen. Es muss festgehalten werden, wann genau der Befehl zum Skalieren getätigt wird und wann die Initialisierungsphase eines entsprechenden Containers abgeschlossen wurde. Als vollkommen initialisiert gilt ein Container, wenn neben dem eigenen Filesystem auch sämtliche relevante Kommunikationsschnittstellen gestartet sind und benötigte Datenverbindungen bestehen. Darüber hinaus sei noch erwähnt, dass die erhobenen Metriken jeweils getrennt pro Backend-Technologie erhoben werden sollen, damit eine Gegenüberstellung gewährleistet werden kann.

Bezüglich der Skalierung gilt, dass insbesondere ein vorausschauendes Regelmodell entworfen werden soll, das die Container stufenweise erst dann skaliert, wenn eine Vielzahl von Nachrichten eine gewisse Latenzzeit überschreiten oder der Message Broker (Definition siehe 5.2.2 Broker) eine entsprechende Menge an unbeantworteten Nachrichten hält. Hierbei soll eine sinnvolle Festlegung von Grenzwerten gefunden werden. Um im Nachhinein genaue Vergleichswerte ermitteln zu können, sollen lediglich die Container der Konsumenten-Instanzen skaliert werden. Die restlichen Stackkomponenten laufen jeweils als eine Singleton-Instanz. Der Skalierungsprozess soll sowohl mit eingegangenen Nachrichten, als auch durch spezifische Benutzeranfragen gestartet werden können. Diese Unterteilung soll in den persistierten Datensätzen erkennbar sein.

Die mit eingegangenen Nachrichten generierten Skalierungen sollen anhand des festgelegten Stufenmodells ausgegeben werden. Hierbei wird der Durchschnitt aller in dieser Stufe / Gruppe enthaltenen Skalierungen gebildet und dargestellt. Es soll allerdings ebenfalls möglich sein, die durchschnittlichen Werte sowohl für einzelne Containeranzahlen als auch für alle durchlaufenen Skalierungen geben. Ähnlich zu den Metriken der *Leistungsfähigkeit* gilt, sie sollen ebenfalls zeitlich aufbereitet werden. Falls sich hierbei Anomalien hinsichlich der zeitlichen Entwicklung abzeichnen lassen, können zusätzliche statistische Metriken hinzugezogen werden. Es könnte beispielsweise von Interesse sein, ob es bestimmte Zeitpunkte gibt, in denen die Skalierungszeit im Vergleich zum Gesamtdurchschnitt deutlich kürzer beziehungsweise länger dauert. Hierbei lässt sich zum Beispiel ermitteln, wie häufig diese Abweichungen auftreten, ob sie immer zum gleichen Zeitpunkt während einer Skalierungsphase auftreten und ob wie ausgeprägt diese Abweichungen ist.

5.1.3 Zusätzliche Kriterien

Um einen funktionsfähigen Prototypen erstellen zu können, der die beiden wesentlichen Kenngrößen überhaupt erheben kann, müssen bereits viele der im vorherigen Abschnitt erläuterten Kriterien gelten. Um skalierte Konsumenten zu ermöglichen, sind die beiden Aspekte des ISO-Hauptkriterium der „*Kompatibilität*“ der Nebenläufigkeit (*Ko-Existenz* sowie *Interoperabilität* siehe Abschnitt 4.1 Kompatibilität) unabdingbar. Ein funktionierender Prototyp zeigt hierbei auf, dass die Aspekte zwangsweise erfüllt sein müssen. Diese werden daher nicht erneut gesondert betrachtet. Auch das Kriterium der „*Zuverlässigkeit*“ (siehe Abschnitt 4.1 Zuverlässigkeit) wird durch eine funktionierende Implementierung bereits in Teilen erfüllt, da es beim genutzten Orchestrator zum gegenwärtigen Zeitpunkt keine Möglichkeit für eine definierte Ausführungsreihenfolge gibt. Die Komponenten müssen im Zweifelsfall mehrfach starten um eine Verbindung aufbauen zu können. Sämtliche Anforderungen bezüglich des Prototypen, der Orchestrierungsplattform, des Lasttests sowie deren Visualisierung wurden im letzten Abschnitt (siehe 4.2 Anforderungen an den Prototypen) ausreichend detailliert erläutert und können komplett übernommen werden. Für die Umsetzung dieser Anforderungen siehe Abschnitt 5.2 Implementierung des Prototypen.

5.1.4 Zusammenfassung

Der Übersichtlichkeit halber hier noch einmal eine stichwortartige Zusammenfassung der zu ermittelnden Metriken hinsichtlich der betrachteten Kriterien:

- Latenzzeit im Durchschnitt sowie als zeitliche Historie
  - Abschnitt *vor* Datenaufnahme gesondert betrachten
  - Abschnitt *nach* Datenaufnahme gesondert betrachten
  - Gesamte Pipeline betrachten
- Skalierungsdauer jeweils pro verwendeter Backend-Technologie festzuhalten
  - Skalierung anhand eingehender Nachrichten mithilfe Stufenmodell
  - Skalierung mittels direkter Benutzeranfrage (ohne eingehende Nachrichten)
  - Metriken als Datensätze in einer Datenbank hinterlegt
  - Durchschnittliche Startzeit pro Containeranzahl
  - Durchschnittliche Startzeit pro Skalierungsstufe
  - Gesamtdurchschnittliche Startzeit als einzelner Messwert
  - Metriken zeitlich visualisieren

5.2 Implementierung des Prototypen

Nachdem im letzten Abschnitt eine Festlegung auf spezifischen Kriterien zur Erhebung von Messwerten vorgenommen wurde, folgt eine detaillierte Beschreibung der Implementierung des zu erstellenden Prototypen. Der komplette Zugang zum Quellcode sowie alle zugehörigen Konfigurationsdateien kann auf Anfrage per E-Mail bereitgestellt werden.

5.2.1 Schichtenmodell

Der Arbeitsfluss der Anwendung wird in Abbildung 5.1 visuell dargestellt. Die unterschiedlichen Komponenten werden dabei in vier verschiedene Schichten eingeteilt:

- Persistenzschicht (engl. *persistence layer*): Beinhaltet sämtliche Komponenten, die für das Abspeichern gegebener Datensätze in dazugehörige Datenbanken zuständig sind. Um Nebenläufigkeit zu ermöglichen, wird hier ebenfalls auf Schnittstellen über einen Message-Broker zurückgegriffen. Eine Manipulation der Daten findet auf dieser Ebene nicht statt.
- Verarbeitungsschicht (engl. *processing layer*): Beinhaltet sämtliche Komponenten, die für die direkte Verarbeitung der Business-Logik zuständig sind. Die Kommunikation zwischen den Komponenten findet über eine REST-Schnittstelle sowie einen Message-Broker statt. Der Message-Broker stellt hierbei vor allem eine nebenläufige Verarbeitung der konsumierenden Komponenten sicher. Allerdings bietet dieser ebenfalls die Schnittstelle zur Skalierungsschicht (engl. *scaling layer*).

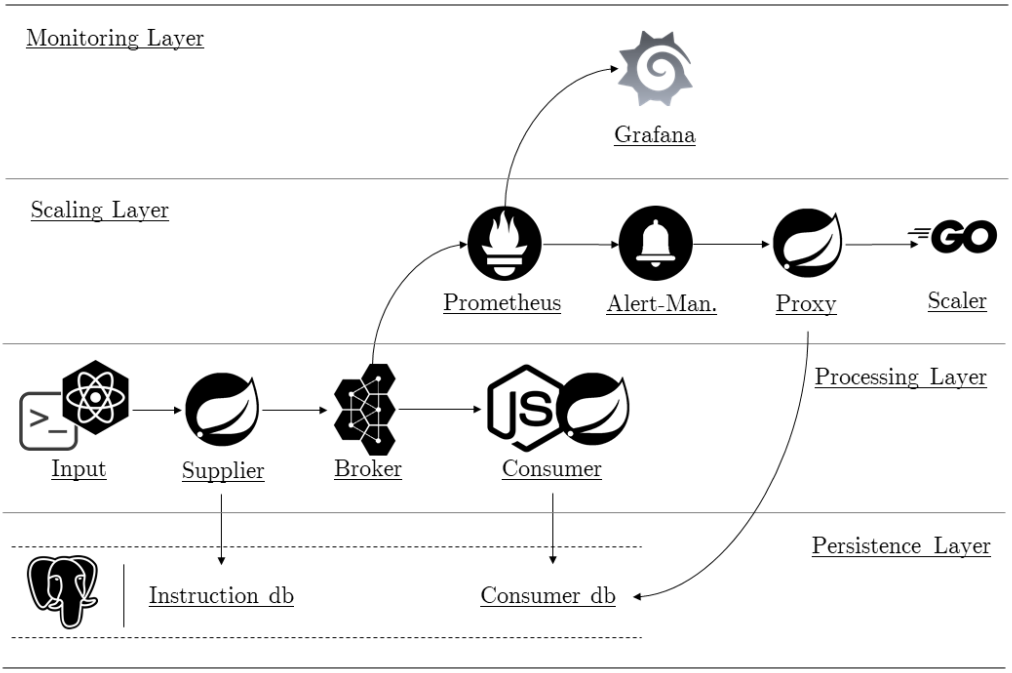


Abbildung 5.1: Komponenten-Stack im Überblick

- Skalierungsschicht (engl. *scaling layer*): Beinhaltet sämtliche Komponenten zum Skalieren der konsumierenden Komponenten. Hierbei wird auf eine universelle Schnittstelle des Message Broker zurückgegriffen um entsprechende Metriken abzugreifen, die für die Evaluierung hinterlegter Regeln zur Skalierung verwendet werden. Ansonsten wird in Komponenten dieser Schicht das Zeitverhalten des Initialisierungsprozesses der konsumierenden Komponenten überwacht und an Schnittstellen der Persistenzschicht weitergeleitet.
- Monitoring Ebene: Diese Schicht besteht aus einer einzigen Komponente, deren Aufgabe es ist erhaltene Daten visuell darzustellen. Um einen zeitlichen Überblick zu geben, werden die Daten intern vom Werkzeug gespeichert. Da es sich um lokal verwaltete Datensätze handelt, die für den Rest der Applikation keinerlei Bedeutung darstellen, wurde darauf verzichtet eine Lösung zu finden, in der diese ebenfalls in einer Komponente der Persistenzschicht abgespeichert wird.

5.2.2 Komponenten im Überblick

Es folgt eine kurze Zusammenfassung der Funktionalität sowie der Anforderungen an die jeweiligen Komponenten.

Input

Um es dem Benutzer zu ermöglichen, gezielte Messwerte zu erfassen, wird eine REST-Schnittstelle vom System bereitgestellt. Es ist zwar möglich, dass der Benutzer selbstständig Anfragen für diese Schnittstelle generiert und absendet, beabsichtigt ist allerdings, dass der Benutzer auf vordefinierte Skripte oder eine entsprechende Benutzeroberfläche zurückgreift. Die Benutzeroberfläche ist sehr funktional gehalten, sodass es zwar möglich ist, hierüber Anfragen an das System zu stellen, dennoch empfiehlt sich gerade für komplexere Anfrageszenarien der Gebrauch der bereitgestellten Bash-Skripte. Bezüglich der Skripte gibt es ebenfalls diverse Abstraktionsschichten. So ist es zum Beispiel möglich,

mithilfe einer definierten Grammatik Anforderungen zu definieren, die sich beliebig kombinieren lassen. Hierzu wurden mehrere vordefinierte Dateien angelegt, deren Inhalt über ein entsprechendes Skript an das Backend geschickt werden kann. Allerdings gibt es weitere Skripte, die auf diesem Prozess aufsetzen und somit eine Abstraktionsschicht aufbauen, sodass sich der Endbenutzer hiermit nicht auseinandersetzen muss (detaillierte Beschreibung siehe Abschnitt *ss:Input* Input).

Supplier

Diese Komponente ist in der Lage, die vereinfachten Anfragen des Benutzers zu interpretieren und in Nachrichten umzuwandeln, die vom System verarbeitet werden können. Hierbei werden bereits an dieser Stelle der Verarbeitungspipeline diverse Informationen an die ursprüngliche Nachricht angeheftet, um im späteren Verlauf entsprechende Metriken zu berechnen. Außerdem erfolgt eine erste Kommunikation mit der Persistenzschicht, in der die übersetzten Benutzeranfragen abgespeichert werden. Hierbei wird direkt auf die Datenbank zugegriffen, da im System lediglich eine Singleton-Instanz des *Suppliers* vorhanden ist. Zwar unterstützt die verwendete Postgres-Datenbank mehrere Klienten zur gleichen Zeit, nativ ist diese Anzahl jedoch begrenzt und muss angemessen konfiguriert werden. Des Weiteren stellt die Supplier-Komponente zwei Modi hinsichtlich der Geschwindigkeit bereit, in der Nachrichten an den Broker übermittelt werden. So ist es zum Beispiel möglich, Nachrichten über einen gewissen Zeitraum hinweg abzuschicken oder aber eine Transaktion zu bilden, in der alle auf einmal geschickt werden.

Broker

Ein Message-Broker stellt die Funktionalität bereit, Nachrichten über ein gegebenes Protokoll an mehrere Konsumenten zu vermitteln. Hierzu wurde eine Implementierung von Apache namens „*Active MQ*[3]“ gewählt. Diese besitzt zwei Betriebsmodi, einmal das Arbeiten mit einer *Topic* sowie mit einer *Queue*. Eine Topic stellt ein *Publisher-Subscriber-Muster* bereit, in dem alle eingehenden Nachrichten an alle registrierten Konsumenten verschickt werden [SBD11, Seite 33 ff.]. Demgegenüber steht eine *Queue* (Warteschlange), die eingehende Nachrichten lediglich an einen einzelnen Konsumenten übermittelt, wobei der Broker hierbei als eine Art Load Balancer[4] agiert. In dem Prototypen wurde hierbei stets auf Warteschlangen zurückgegriffen, da eingehende Nachrichten stets von einer einzelnen Komponenteninstanz beantwortet werden. Bevor die Nachrichten jedoch aus einer der beiden Datenstrukturen entfernt werden, muss der betreffende Konsument eine *Acknowledgement-Nachricht* an den Broker schicken, um zu signalisieren, dass die Nachricht nicht nur angenommen, sondern auch korrekt verarbeitet werden konnte. Je nach Empfangskomponente wird dieses Acknowledgement direkt von der verwendeten Library geschickt oder aber im Programmcode selbst explizit gesetzt. Es gilt weiterhin noch hervorzuheben, dass die eingeschriebenen Konsumenten bei neuen Nachrichten stets benachrichtigt werden und es clientseitig keine Logik braucht, um zum Beispiel event- oder intervallbasiert eine Abfrage an den Broker zu steuern. Da diese Komponente ebenfalls den Einstiegspunkt für das Monitoring System *Prometheus* (siehe Abschnitt 5.2.2 Prometheus) darstellt, benötigt der Broker eine Schnittstelle über die er diesem Tool Messdaten zur Verfügung stellen kann. Die von ActiveMq bereitgestellt Schnittstelle nennt sich „*Java Management Extension API (JMX)*“ und ist die Standard-API zur Verwaltung von Java Applikationen [SBD11, Seite 331 ff.].

[3] `https://github.com/apache/activemq`
[4] Erklärung *Load Balancer* siehe Abschnitt 3.1 JBoss (Microprofile)

Consumer

Die konsumierenden Komponenten empfangen die generierten Nachrichten und können beliebig skaliert werden. Sie implementieren die in Abschnitt 4.2.1 definierten Arbeitsschritte. Hierbei wird auf diverse Libraries zurückgegriffen, wobei die restliche Logik einfach gehalten wurde. Die Komponenten kommunizieren lediglich über den Message-Broker mit dem Nachrichten-Supplier und über einen weiteren Nachrichten-Broker mit der Persistenzschicht um die extrahierten Elemente abzulegen.

Prometheus

„*Prometheus is an open source systems monitoring and alerting toolkit*“ [Bra18, Seite 400]. Es ist möglich über eine definierte Anfragensprache Daten Dritter zu verarbeiten. Diese Daten können über ein einfaches Textformat von den Komponenten ausgegeben werden. Es ist möglich dieses Textformat händisch zu schreiben, allerdings wird in der Praxis vermehrt auf Libraries, die auf den Client zugeschnitten wurden, gesetzt. Prometheus ist unter der Apache 2.0 Lizenz veröffentlicht [5], und wurde primär in der Programmiersprache Go implementiert. Das intervallbasierte Anfragen (engl. „*scrapen*“) der Metriken wird von der Prometheus-Komponente selbst durchgeführt, die zu überwachenden Komponenten müssen sich selbst nicht darum kümmern, Daten an Prometheus zu übermitteln. Im Prototypen werden hierbei mehrere Warteschlangen überwacht. Diese Warteschlangen bilden die Grundlage zur Visualisierung sowie zur Überwachung der nötigen Skalierungsschritte.

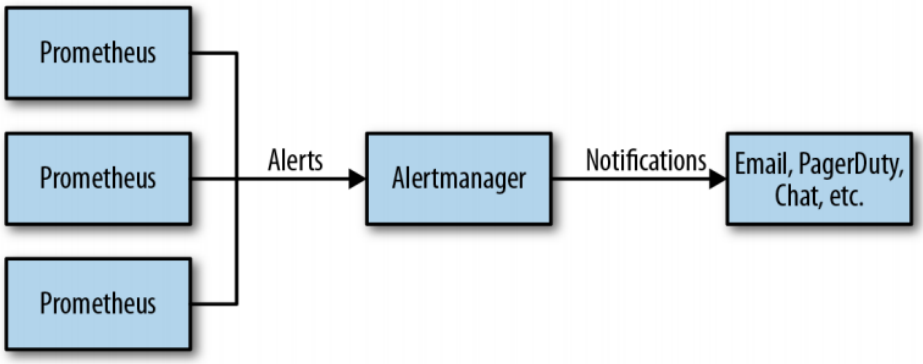


Abbildung 5.2: Alert Manager - Übersicht [Bra18, Seite 291]

Alert Manager

„*Alerting is one of the components of monitoring, allowing you to notify a human when there is a problem*“ [Bra18, Seite 291]. Prometheus bietet hierbei die Möglichkeit mithilfe der funktionalen Anfragesprache *PromQl* diverse Bedingungen zu definieren unter denen dies geschehen soll. Da es in einer produktiven Containerumgebung durchaus möglich ist, dass mehrere Prometheus-Instanzen parallel arbeiten, wurde das Benachrichtigen in eine weitere Komponente (den *Alert Manger*) ausgelagert. Dieser synchronisiert, sammelt und gruppiert die Alerts der verschiedenen Prometheus-Instanzen und sendet Benachrichtigungen an definierte Nachrichtenkanäle, die sich beispielsweise aus einem Emailpostfach, einer Pagernachricht oder Chatnachricht auf Plattformen wie Slack zusammensetzen können (siehe Abbildung 5.2 Alert Manager - Übersicht). Im Prototypen wurden die eingehenden Alerts derartig übersetzt, dass sie von der Proxy-Komponente interpretiert werden können.

[5] https://github.com/prometheus/prometheus

Scaler Proxy

Diese Komponente bietet eine REST-Schnittstelle, die im Alert-Manager hinterlegt wird. Sobald eine der Regeln anschlägt, wird der Aufruf an diese Komponente weitergeleitet. Im Endeffekt dient diese Komponente lediglich als Proxy-Service, da ihre primäre Aufgabe darin besteht diese Nachricht an eine weitere Komponente weiterzuleiten. In einer produktiven Umgebung würde diese Komponente komplett enfallen, da es möglich ist den Alert-Manager derartig zu konfigurieren, dass er direkt die öffentlichen Schnittstellen der Scaler-API anspricht. Es wurde sich dennoch für das Zwischenschalten eines solchen Proxy-Services entschieden um genauere Messwerte zu erhalten. Gerade hinsichtlich der Initialisierungsphasen von Containern ist es angebracht so kurz vorher wie möglich einen Timestamp zu setzen. Da es keine Möglichkeit gibt direkt in die Konfiguration der Scaler-API einzugreifen ist diese Lösung der nächstbeste Ansatz. Sobald eine Skalierungsanfrage weitergeleitet wurde, wird diese noch unbeantwortete Anfrage intern in einer Datenstruktur abgelegt. Sobald ein entsprechender Container komplett initialisiert wurde, ruft dieser eine weitere Schnittstelle des Proxy-Services auf. Daraufhin wird ein *Acknowledgement-Timestamp* in der hinterlegten Anfrage gesetzt und an die Persistenzschicht weitergeleitet.

Außerdem bietet diese Komponente die Möglichkeit für den Benutzer, direkt Exemplare eines Consumers hochzufahren. Die verwendete Schnittstelle dient sowohl als Umgehung des herkömmlichen Programmflusses als auch zur Generierung der Metrikberechnung, ohne auf den nachrichtenbasierten Skalierungsalgorithmus zurückgreifen zu müssen (siehe Abschnitt 6.1.2 Aufschlüsselung nach Containeranzahl). Über dedizierte Endpunkte können diese Metriken als .csv Datei ausgelesen werden.

Scaler

„*The goal of the Docker Scaler project is to provide a REST HTTP interface to scale services and nodes*“ [Fan18]. Außerdem werden mit jeder Skalierungsanfrage Statusinformationen über die aktuelle sowie zukünftige Anzahl von Instanzen des zu skalierenden Services zurückgegeben, die es dem Proxy-Service ermöglichen, davon ausgehend einen Skalierungsstop für neue Anfragen beizubehalten oder aufzuheben. Dieser ist nötig, damit das System keine weiteren Skalierungsprozesse startet, wenn bereits aktive Skalierungen vorgenommen werden. Im Prototypen wurde diese Komponente mit minimaler Konfiguration direkt in den Stack übernommen.

Grafana

Wenn es zum Alert durch den Alert Manager kommt, wird in einer Produktivumgebung der erste Schritt sein die Performanz des Systems durch dedizierte Dashboards zu überprüfen [Bra18, Seite 97]. Grafana ist ein Werkzeug, das dies über eine Weboberfläche direkt im Browser ermöglicht. Es bietet die Möglichkeit Graphen, Tabellen und weitere Visualisierungskomponenten zu erstellen um zum Beispiel die Latenzzeit oder CPU Auslastung zu überprüfen. Diese Metriken können für das ganze System oder nur einen Teil generiert werden. Es ist das bevorzugte Visualisierungswerkzeug für Prometheus, bietet allerdings ebenfalls Unterstützung für verschiedene weitere Systeme wie zum Beispiel *Graphite*, *Elasticsearch* oder *PostgreSQL*. Im Prototypen wurden hierbei sämtliche Metriken der Warteschlangen, sowie sonstiger überwachter Komponenten, grafisch aufbereitet. Hierbei ist auch ein zeitlicher Ablauf zu erkennen.

Mock scaler api

Diese Komponente ist nicht Kernbestandteil des Komponenten-Stacks. Sie dient lediglich während der Entwicklungszeit dazu, die Scaler-API zu emulieren. So ist es zum Beispiel möglich lokal eine Instanz des Scaler-Proxy Projekts in einer beliebigen IDE als Standard Spring Projekt auszuführen. Sämtliche Skalierungs-Anfragen können von dieser Komponente angenommen werden, da sie auf dem gleichen Port wie der „*echte*“ Scaler läuft, wobei sie dabei ebenfalls in der Lage ist diverse Rückgabenachrichten an den Klienten zu übergeben.

5.2.3 Input

Der Benutzer besitzt die Möglichkeit, sowohl über eine Benutzeroberfläche als auch über Bash-Skripte Benchmark Anfragen an das System zu stellen. Im Folgenden wird beides im Detail erläutert.

Aufbau

In der folgenden Abbildung ist der Aufbau des Bash-Projektes dargestellt. Der Benutzer besitzt drei Möglichkeiten, Anfragen an das System zu stellen. Für jeden Einstiegspunkt wurde ein entsprechendes Bash-Skript zur Verfügung gestellt.

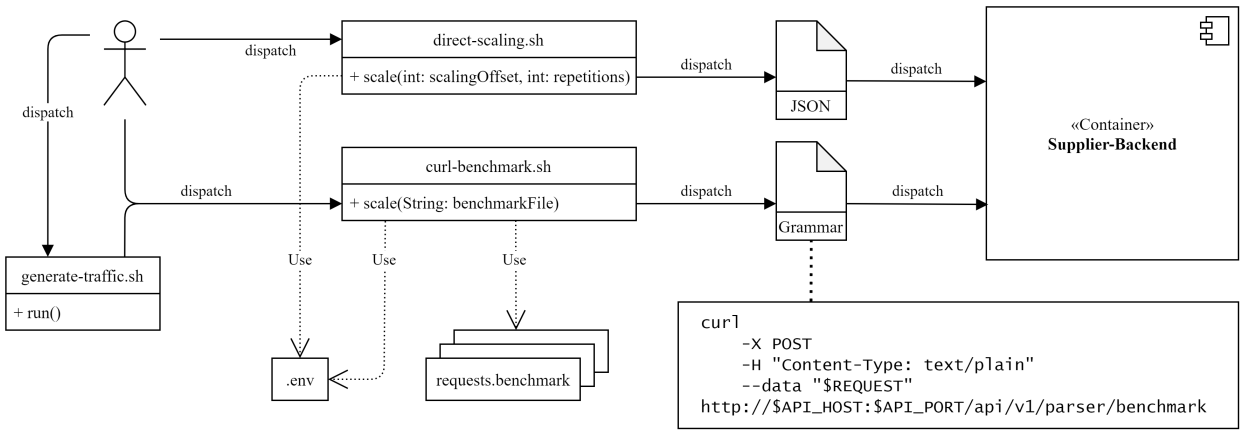


Abbildung 5.3: Bash Input UML

- *direct-scaling.sh*: bietet die Möglichkeit eine direkte Skalierungsanfrage an das System zu senden, ohne Payment-Messages verwenden zu müssen. Als Parameter werden die Skalierungsanzahl sowie die Wiederholungen erwartet (genauere Erklärung siehe 5.2.3 Manuelles Skalieren).
- *curl-benchmark.sh*: sendet eine Skalierungsanfrage mit dem Dateiinhalt einer Datei, dessen Pfad als Eingabeparameter verarbeitet wird. Die Datei besitzt die Dateiendung *.benchmark*, wobei der Inhalt einer spezifizierten Grammatik folgt (siehe 5.2.3 Skalieren mithilfe einer Grammatik).
- *generate-traffic.sh*: dient als vereinfachter Einstiegspunkt für den Benutzer. Hierbei werden rekursiv alle verfügbaren *.benchmark* Dateien des Verzeichnisses der Reihe nach dem *curl-benchmark* Skript übergeben.

Im folgenden Listing (siehe Listing 5.2.3 Aufbau) ist ein Auszug der Projekt-Struktur abgebildet. Neben den genannten Skripts sind sämtliche *.benchmark* Dateien nach Services in Unterordner aufgeteilt worden. Die *.env* Datei beinhaltet die Verbindungsdaten zum Komponenten-Stack. Diese

müssen bei der ersten Ausführung vom Benutzer angeglichen werden. Im Wesentlichen beschränkt sich dies auf die IP-Adresse des Systems, falls der Benutzer in der Stack-Konfiguration entsprechende Änderungen vornimmt, sind diese an dieser Stelle ebenfalls zu vermerken.

```
$ tree request-scripts/ -a -L 3 --charset=ascii
request-scripts/
|-- curl-benchmark.sh
|-- direct-scaling.sh
|-- .env
|-- generate-traffic.sh
|-- README.md
`-- requests
    |-- mixed
    |   |-- benchmark_large_mixed.benchmark
    |   |-- benchmark_medium_mixed.benchmark
    |   `-- benchmark_small_mixed.benchmark
    ...
```

Listing 5.1: Bash Skript - Struktur

Manuelles Skalieren

Hierfür wird auf ein Bash-Skript namens „*direct-scaling*“ zurückgegriffen. Das Skript nimmt zwei Parameter entgegen. Der Erste beschreibt die Anzahl der Instanzen, zu der das System skaliert werden soll, wobei hierbei eine inkrementelle Erhöhung von einer Instanz pro Durchlauf durchgeführt wird. Der zweite Parameter gibt die Wiederholungen dieser testweisen Skalierung an. Der Aufruf `./direct-scaling.sh 10 20` resultiert beispielsweise darin, dass sowohl für den Node.js als auch Spring Boot Service jeweils 10 Skalierungsschritte durchgeführt werden. Wenn zum Beispiel bereits ein Node.js-Container läuft, wird im ersten Test eine weitere Instanz angefordert. Sobald die beiden Instanzen verfügbar sind, wird der Alert-Manager durch die Prometheus-Komponente angewiesen die überflüssige Instanz zu löschen. Sobald wieder ein einziger Container läuft, wird dieser Schritt neunzehn weitere Male wiederholt. Anschließend erfolgt die nächste Skalierung bei der nun zwei neue Container kreiert werden sollen, anstatt wie vorher nur ein weiterer. Auch hier folgen 20 Wiederholungen. Diese Schritte werden solange durchgeführt bis zehn Container zwanzig Mal instanziiert wurden.

```
...

request() {
  curl "http://$HOST:$DIRECT_SCALING_PORT/manual-scale?additionalCnt=$1&service=$4"
}

node_request() {request $1 $2 $3 'NODE'}
spring_request() {request $1 $2 $3 'SPRING'}

for scalingOffset in $(seq 1 $1)
do
  for curr_rep in $(seq 1 $2)
        do
    node_request $scalingOffset $curr_rep $2
    spring_request $scalingOffset $curr_rep $2
  done
done
```

Listing 5.2: direct-scaling.sh

Skalieren mithilfe einer Grammatik

Das beschriebene Skript sowie der zugrunde liegende Http-Endpunkt bieten eine Möglichkeit, das Skalieren als Reaktion auf unbeantwortete Nachrichten zu umgehen. Um jedoch einen herkömmlichen Testdurchlauf zu starten, greift der Benutzer auf einen Endpunkt vom *Supplier* zurück:

```
curl
  -X POST
  -H "Content-Type: text/plain"
  --data "$REQUEST"
  "http://$API_HOST:$API_PORT/api/v1/parser/benchmark"
```

Es handelt sich hierbei um eine POST-Anfrage. Im Body wird der Dateiinhalt eines für diesen Zweck verfassten Skripts angeheftet. Im Projekt wurden diverse Beispiele verfasst. Diese liegen im Ordner *./requests* und tragen die Dateiendung „*.benchmark*“. Der Inhalt korrespondiert zu einer spezifizierten Grammatik (siehe Listing 5.2.3 Skalieren mithilfe einer Grammatik).

```
request     := batch*
batch       := serviceName { instruction | instruction,* };
serviceName := SPRING | NODE
instruction := BENCHMARK ( messageCnt, duration ) | WAIT ( duration )
messageCnt  := [0-9]+
duration    := [0-9]+
```

Es können beliebig viele Batchanfragen an dem Anfragenbody angeheftet werden. Eine Batchanfrage stellt in diesem Zusammenhang eine Gruppe von Skalierungsanfragen an einen bestimmten Service dar, wobei diese mindestens eine Skalierungsinstruktion enthalten müssen. In diesem System gibt es zwei wesentliche Skalierungsinstruktionen:

1. BENCHMARK: Nimmt zwei Parameter entgegen. Der erste spezifiziert, wie viele neue Instanzen erstellt werden sollen, während der Zweite angibt, über welchen Zeitraum dies geschehen soll. Alle numerischen Angaben müssen einen positiven Ganzzahlwert enthalten. Da das System selbstständig in der Lage sein soll diese Instanzen wiederum auf ein gesetztes Minimum zu minimieren, wurde sich aktiv dagegen entschieden, das Herunterskalieren in die Grammatik aufzunehmen, um dem Benutzer eine möglichst einfach gehaltene Schnittstelle zur Verfügung zu stellen.

2. WAIT: Diese Instruktion erwartet lediglich einen einzigen Parameter, der angibt wie viele Millisekunden gewartet werden soll, bevor die nächste Instruktion dem System übergeben wird.

Damit der Benutzer allerdings direkt Anfragen ausführen kann, ohne sich mit der Grammatik beschäftigen zu müssen, wurde ein weiterer Skript-Aufsatz für das *curl-benchmark* Skript entwickelt. Dieses trägt den Namen „*generate-traffic.sh*“ und sucht rekursiv absteigend in der eigenen Directory nach Dateien mit der entsprechenden Endung. Anschließend werden diese Skripte an das *curl-benchmark* der Reihe nach als Parameter übergeben.

Benutzeroberfläche

Im Rahmen dieser Thesis wurde die Benutzeroberfläche mithilfe des Javascript-Frameworks *React.js* entwickelt. Es ist möglich über diverse Select-Felder die voreingestellten Pfaden, sowie Payment Optionen auszuwählen. Außerdem ist es möglich, die Anzahl sowie die Zeitspanne der Anfrage zu spezifizieren. Beim Initialisierungsprozess dieses Projekts starten zuallererst alle dargestellten React-Komponenten. React.js ist ein komponentenbasiertes Framework, hierbei werden sämtliche visuellen Elemente als einzelne Komponenten betrachtet, wobei es möglich ist, diese Komponenten zu gruppieren und somit neue Komponenten zu formen. Außerdem ermöglicht das Framework diesen Komponenten einen sogenannten *State* anzuheften, in dem Daten hinterlegt werden können. Wenn diese an irgendeiner Stelle modifiziert werden, erkennt React dies und rendert die betroffene Komponente sowie alle untergeordneten Komponenten neu. Dies ist insbesondere hilfreich um die Formulareinträge direkt abzuspeichern. Bei der Initialisierung versucht die Frontend-Komponente die darzustellenden Optionen vom *Supplier* zu fetchen. Dieser beantwortet diese Anfrage mit den Daten, die im Anschluss in der Oberfläche angezeigt werden. Wenn der Benutzer darauf folgend Daten auswählt beziehungsweise Daten in die Felder einträgt, werden hierfür zugeschnittene Eventhandler-Funktionen ausgelöst. Diese Eventhander registrieren die Datenauswahl und speichern sie direkt im State der Komponente ab. Wenn der Benutzer nun auf den Button zum Absenden der Anfrage klickt, muss hierbei lediglich der State ausgelesen werden und an die entsprechende Backend-API-Schnittstelle geschickt werden. In Abbildung 5.4 React - Benutzeroberfläche wurde hierfür der aktuelle Stand der Oberfläche dargestellt.

Abbildung 5.4: React - Benutzeroberfläche

Während der Entwicklungszeit des restlichen Systems wurde das Arbeiten mit der Benutzeroberfläche allerdings vermehrt zum Flaschenhals. Sobald ein neues Feature oder ein neuer Datensatz in eine Anfrage integriert wurde oder sich das zugrunde liegende Datenmodell änderte, war es jedes Mal ein großer Aufwand dies ebenfalls in der Oberfläche anzupassen. Da die vorher spezifizierte Grammatik sowie die zugrunde liegende API einen deutlich flexibleren Ansatz und eine komplexere Ablauflogik ermöglichen wurde sich bereits früh in der Entwicklung entschlossen diesen Ansatz der grafischen Benutzerführung weiterzuverfolgen. Die Oberfläche dient hierbei eher als ein *Proof of concept*, um darzustellen, wie ein solches Projekt konfiguriert werden muss. Da dem skriptbasierten Ansatz dieselben API-Schnittstellen zugrunde liegen wie dem oberflächenbasierten Ansatz, ist es ohne Probleme möglich das aktuelle React-Projekt den überarbeiteten Gegebenheiten anzupassen, falls dies in Zukunft gewünscht sein sollte.

5.2.4 Supplier Backend

Bei dieser Komponente handelt es sich um ein Spring Projekt, dessen Aufgabe es ist, Benutzeranfragen in Nachrichten zu übersetzen, die vom System interpretiert und verarbeitet werden können. Die Schnittstelle für den Benutzer besteht hierbei aus einer REST-API, die über Tools wie zum Beispiel curl angesprochen werden kann.

Generelle Spring Projektübersicht

Da diese Komponente bezüglich der Struktur eines Spring-Projekts alle wesentlichen Merkmale besitzt, wird die genutzte Projektstruktur an dieser Stelle etwas näher erläutert.

```
$ tree stack/supplier-backend/ -a -L 7 --charset=ascii
stack/supplier-backend/
|-- Dockerfile
|-- pom.xml
`-- src
    `-- main
        |-- java
        |   `-- dps
        |       `-- hoffmann
        |           `-- producer
        |               |-- config
        |               |-- controller
        |               |-- model
        |               |-- properties
        |               |-- repository
        |               |-- response
        |               |-- service
        |               `-- SupplierBackendApplication.java
        `-- resources
            |-- application-dev.properties
            |-- application-prod.properties
            `-- application.properties
```

Listing 5.3: Supplier Backend - Struktur

Das Muster, nach dem sämtliche Komponenten des Komponenten-Stacks entwickelt wurden, wird als *Layered Architecture Pattern* oder *n-tier pattern* bezeichnet [ore]. Es stellt einen weitverbreiteten Standard in Java Enterprise Applikationen dar und zeichnet sich durch die klare Unterteilung der verschiedenen Hierarchien aus (siehe Abbildung 5.5). Sämtliche Packages innerhalb der Spring-Projekte finden sich innerhalb einer der Schichten wieder. Es sei ebenfalls hervorzuheben, dass sämtliche Schichten stets nur mit ihren direkt angrenzenden Schichten kommunizieren. Dies stellt ein übersichtliches Design sicher.

- *controller*: Dieses Package stellt alle Klassen bereit, welche direkt vom Benutzer angesprochen werden. Typischerweise handelt es sich zumindest im implementierten Stack um Rest-Schnittstellen, aber auch andere Typen (Soap, etc.) bieten entsprechende Implementierung an. Klassen dieses Packages werden allgemeinhin als Controller bezeichnet und bilden eine erste Interaktionsschicht, welche keinerlei Business-Logik enthält. Bei diesem Package handelt es sich um einen Teil des Presentation-Layers. Die Daten werden angenommen aber nicht direkt verarbeitet. Dies geschieht in sogenannten Services, die spezifische Funktionalität implementieren. Controller dienen hierbei als Verteiler der eingehenden Nachrichten an diese Services. Die Service-Instanzen werden den Controllern über den Spring-Context mittels Dependency

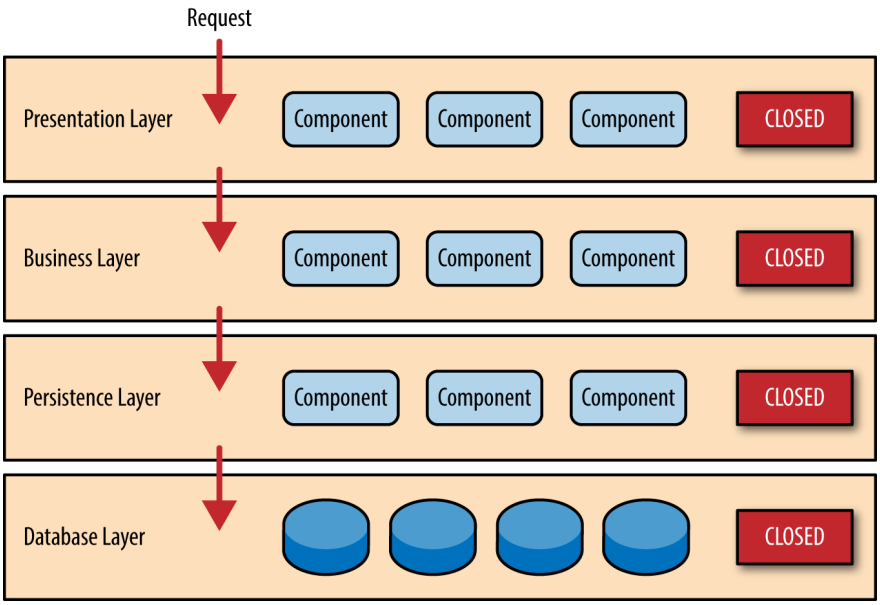


Abbildung 5.5: Layered Architecture [ore]

Injection[6] bereitgestellt, sodass sich der Entwickler selbst nicht mit der Instanziierung etc. zu beschäftigen braucht.

- *service*: Dieses Package enthält ausschließlich Service-Implementierungen. Ein Service stellt einen logischen Funktionsbaustein der Anwendung dar und kann über den Spring-Context bereitgestellt werden. Services werden dem *Business Layer* des Schichtenmodells zugeordnet.
- *repository*: Spring bietet über entsprechende Dependency-Starter-Konfigurationen die JPA Implementierung *Hibernate* als nativen Support für die Anbindung an eine Datenbank. Dieses Package enthält Interfaces welche die Treiberschnittstelle beerben. Mithilfe spezifizierter Namenskonventionen ist es möglich Methodensignaturen für Datenbankanfragen zu gestalten. Das Framework generiert hieraus die entsprechenden SQL-Abfragen. Klassen dieses Packages lassen sich dem *Persistence Layer* des Modells zuordnen.
- *config*: Klassen dieses Packages instanziieren, beziehungsweise konfigurieren Spring-Beans. Es gibt bei vielen voreingestellten Spring-Komponenten die Möglichkeit Beans über Annotations in der implementierenden Klasse zu erzeugen. In bestimmten Situationen wird jedoch Zugriff auf die Instanz selbst zum Initialisierungszeitpunkt gebraucht um entsprechende Konfigurationsschritte zu unternehmen. Dies geschieht standardmäßig in diesem Package. Da dieses Package lediglich eine unterstützende Funktion einnimmt, wird es im Schichtenmodell nicht direkt aufgeführt. Um jedoch eine übersichtliche Projektstruktur zu gewährleisten, werden diese Klassen nicht in Packages der zu konfigurierenden Komponenten mit eingefügt, sondern gesondert angelegt.
- *model*: Klassen, welche lediglich zum Abbilden bestimmter Datensätze genutzt werden, residieren in diesem Package. Hierbei ist es irrelevant, ob es sich um JPA-Entitäten oder herkömmliche POJOs handelt. Da es sich bei der Funktionalität der Klassen dieses Packages ebenfalls um eine unterstützende Funktion handelt, wird es ebenfalls nicht direkt im Schichtenmodell aufgeführt.
- *properties*: Spring bietet die Möglichkeit Daten aus den Konfigurationsdateien innerhalb der Resources direkt in den Application-Context zu laden. Diese Klassen bilden eine Unterkategorie der beschriebenen Konfigurationsklassen.

[6]Dependency Injection: Verwendung von Objektinstanzen, welche durch einen IoC Container verwaltet werden.

- *utils*: Klassen, die lediglich mit Business-Logik gefüllt sind und nicht vom Spring-Container verwaltet werden, sondern direkt im Programmcode instanziiert oder aufgerufen werden, residieren in diesem Package. Bei der Funktionalität der Klassen dieses Packages handelt es sich in der Regel um statische Hilfsmethoden die von Instanzen des *Business Layers* verwendet werden.
- *resources*: Die genutzte Verbindungs-Konfiguration zwischen Komponenten des Stacks werden innerhalb der *.properties* Dateien verwaltet. Es ist möglich auf diese Information über den Dependency-Injection-Mechanismus innerhalb der Klassen zuzugreifen. Die hierbei hinterlegten Daten können zum Buildzeitpunkt direkt in das Artifact eingebunden werden. Sie stehen zur Laufzeit zur Verfügung und stellen ähnlich wie die Klassen des *model* packages lediglich relevante Daten dar, beinhalten allerdings keine zusätzliche Programmlogik.

Programmfluss

Im folgenden UML-Diagramm (siehe Abbildung 5.6) wurden ein Teil der Klassen des Suppliers dargestellt. Es wurde sich hierbei auf die wesentlichen Logikbausteine konzentriert.

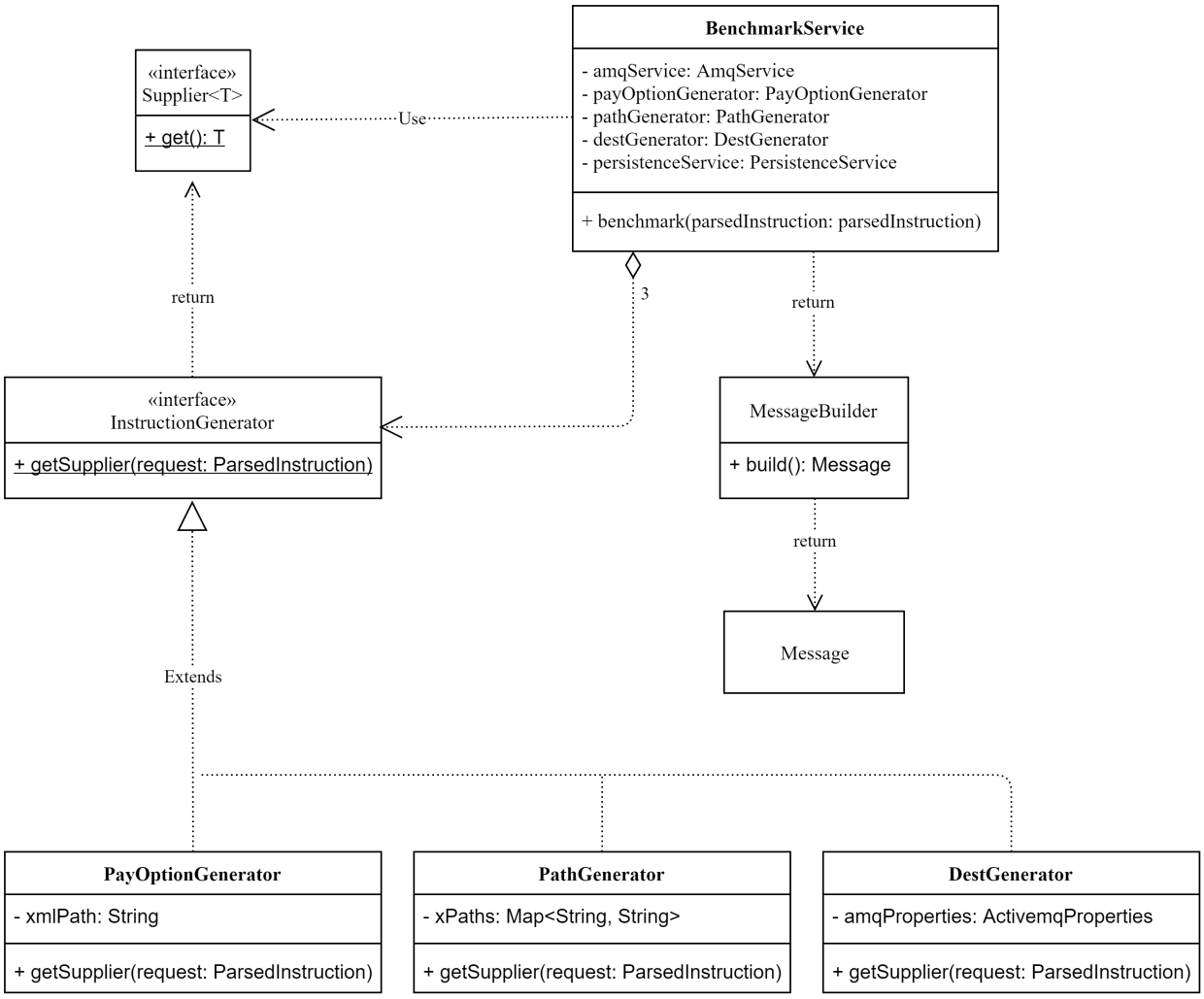


Abbildung 5.6: Backend Supplier UML

Es gibt primär drei Einstiegspunkte für den Programmfluss. Der Benutzer möchte mithilfe der Benutzeroberfläche oder eines Curl-Aufrufs neue Nachrichten generieren, oder es wird auf die

Dateiinhalte, die einer Grammatik folgen zurückgegriffen. Bei beiden Endpunkten wird im Controller eine Instanz vom Typ *ParsedInstruction* erstellt. Diese enthält alle relevanten Informationen, die vom verarbeitenden *BenchmarkService* gebraucht werden. Bei der API-Anfrage durch den Curl-Aufruf lässt sich das Instanziieren direkt über das Parsen vom empfangenen JSON umsetzen, während bei der Grammatik ein weiterer Service (*RequestParserService*) verwendet wird, der aus der erhaltenen Grammatik diverse *ParsedInstruction* Instanzen erstellt.

Wie im UML-Diagramm zu erkennen, werden aus der erhaltenen Instruktionsinstanz über verschiedene Generator-Implementierungen diverse Callbacks erstellt, die zum Absetzen der Nachrichten in das System genutzt werden. Ein Auszug des zugrunde liegenden Codes folgt in Listing 5.4 Supplier - Service;

```
1
2       ...
3
4       // create generator instances
5       Supplier<String> xPathSupplier = pathGenerator.getSupplier(parsedInstruction);
6       Supplier<String> paymentSupplier = payOptionGenerator.getSupplier(parsedInstruction);
7       Supplier<String> destinationSupplier = destGenerator.getSupplier(parsedInstruction);
8       BiConsumer<PaymentMessage, Supplier<String>> amqConsumer =
9               amqService.getConsumer(sessionIsTransacted);
10
11      ...
12
13      for (int i = 0; i < parsedInstruction.getMessageCnt(); i++) {
14
15          // use generator instances to dynamically create message
16          PaymentMessage payment = PaymentMessage.builder()
17                  .batchId(batchId)
18                  .xPath(xPathSupplier.get())
19                  .content(paymentSupplier.get())
20                  .sentTimestamp(now())
21                  .build();
22
23          amqConsumer.accept(payment, destinationSupplier);
24
25          Thread.sleep(durationMillis);
26      }
27  }
28
29  ...
```

Listing 5.4: Supplier - Service

In dem beschriebenen Listing ist zu erkennen, wie diese Generatoren genutzt werden. Sie extrahieren aus der gegebenen Instruktionsinstanz die jeweils relevanten Werte für ihren Anwendungsfall (Zeile 5 – 9) und können bei der Erstellung der Nachrichten als Callback verwendet werden, und dynamisch neue Werte erstellen (Zeile 16 – 21). Hierbei wurde auf diese Callback-Struktur zurückgegriffen, da die gegebenen Werte in der Instruktion selbst als Instruktionen verstanden werden sollen.

Beispiel xPathSupplier Der XPath gibt an, welches XML-Element bei der Verarbeitung durch den Consumer im Detail extrahiert werden soll (bspw. IBAN, Betrag etc.). Es ist jedoch auch möglich, dass der Benutzer bei der API-Anfrage hierbei keinen genauen XPath angibt sondern möchte, dass hierbei ein beliebiges Element extrahiert werden soll. Dazu wird der Wert „*Randomize*“ in die Instruktion eingetragen. Dies ist sinnvoll um etwas Variation in die spätere Verarbeitung zu bekommen um bessere Messwerte erzielen zu könnnen. Im System wurden hierfür beispielhaft

einige XPath Pfade manuell hinterlegt. Der Supplier hat intern Zugriff auf diese Sammlung. Je nachdem ob ein spezifischer Wert ausgegeben werden oder dieser variieren soll, wird stets der selbe Wert oder aber unterschiedliche Werte ausgegeben. Ähnliches gilt für die anderen verfügbaren Generator-Instanzen.

5.2.5 Consumer-Komponente

Im folgenden UML-Diagramm ist die grundlegende Klassenstruktur der beteiligten Logikbausteine des Spring-Konsumenten zu erkennen. Da das Node.js-Projekt einen sehr ähnlichen Aufbau besitzt, wird im Folgenden lediglich der Ablauf der Spring Komponente zusammengefasst. Sämtliche Informationen lassen sich jedoch nahtlos auf die Node.js-Umgebung übertragen.

Der Einstiegspunkt für den Spring-Konsumenten stellt der *AmqService* dar. Nachrichten werden mithilfe dieser Komponente aus der Warteschlange des ActiveMq Brokers ausgelesen und an den *WorkerService* delegiert. Über eine *ObjectMapper* Instanz wird aus dem übergebenen String eine Objektinstanz eines Datenobjektes (*InputPaymentMessage*) generiert (siehe Listing 5.5). Dieses Objekt wird an einen weiteren Service übergeben, der ein Element entsprechend der Vorgaben im Datenobjekt extrahiert. Wenn das zugrunde liegende XML nicht XSD-konform ist oder das Element zum spezifizierten Pfad nicht gefunden werden kann, wird eine Null-Referenz ausgegeben. Es folgt ein Nullcheck, bevor die Nachricht der Persistenzschicht übergeben wird.

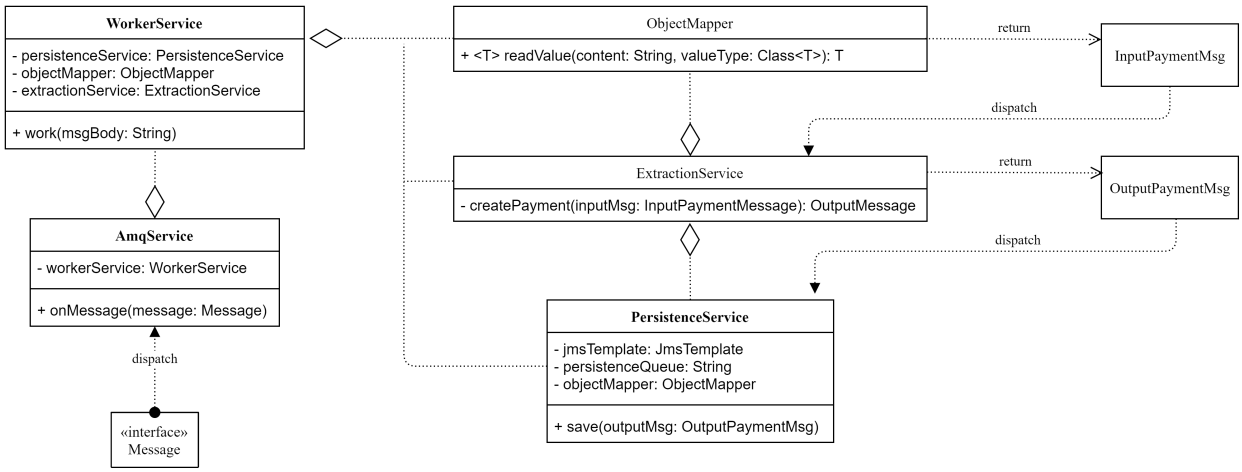


Abbildung 5.7: Consumer UML

```
1
2   ...
3
4   @SneakyThrows
5   public void work(String msgBody) {
6     InputPaymentMsg inputMessage = objectMapper.readValue(msgBody, InputPaymentMsg.class);
7     OutputPaymentMsg outputMessage = extractionService.createPayment(inputMessage);
8     if (outputMessage != null) {
9       persistenceService.save(outputMessage);
10    }
11    Thread.sleep(3000);
12  }
13
14  ...
```

Listing 5.5: WorkerService - Consumer Logik

5.2.6 Prometheus

Regelsatz für Skalierung Prometheus ist eine Komponente, die zum Monitoring und zur Steuerung der Skalierung genutzt werden kann (siehe Abschnitt 5.2.2 Prometheus). Für dieses Projekt musste diese lediglich derartig konfiguriert werden, dass entsprechende Alerts generiert werden und an die Alter-Manager-Komponente weitergeleitet werden können. Diese Komponente informiert erst anschließend die (Proxy-)Scaler-Komponente. Die auszuwertenden Regeln residieren jedoch bereits in der Prometheus Komponente. Das Regelwerk wurde nach folgendem Schema entworfen:

$\underline{\text{QL3}}$ $QB2 < MC$	UP *abs(CB0 – CB3)*	UP *abs(CB1 – CB3)*	UP *abs(CB2 – CB3)*	OK _
$\underline{\text{QL2}}$ $QB1 < MC \leq QB2$	UP *abs(CB0 – CB2)*	UP *abs(CB1 – CB2)*	OK _	DOWN *abs(CB2 – CB3)*
$\underline{\text{QL1}}$ $QB0 < MC \leq QB1$	UP *abs(CB0 – CB1)*	OK _	DOWN *abs(CB1 – CB2)*	DOWN *abs(CB1 – CB3)*
$\underline{\text{QL0}}$ $MC \leq QB0$	OK _	DOWN *abs(CB0 – CB1)*	DOWN *abs(CB0 – CB2)*	DOWN *abs(CB0 – CB3)*
	$\underline{\text{CL0}}$ $CB0 == CC$	$\underline{\text{CL1}}$ $CB0 < CC \leq CB1$	$\underline{\text{CL2}}$ $CB1 < CC \leq CB2$	$\underline{\text{CL3}}$ $CB2 < CC \leq CB3$

Diese Tabelle bestimmt das Skalierungsverhalten des Systems. Aus ihr werden die Regeln für die Prometheus-Konfiguration abgeleitet. Um diese Tabelle etwas anschaulicher erläutern zu können, werden die folgenden Werte für entsprechende Umgebungsvariablen angenommen.

```
CB0=1    CB2=10    QB0=15    QB2=100    CC: Container Count
CB1=5    CB3=30    QB1=30               MC: Message Count
```

Auf der horizontalen Achse wird die aktuelle Containeranzahl eines Systems in verschiedene Zustände eingeteilt. *CL0* bis *Cl3* beschreibt hierbei das sogenannte *Container Level*. Diese, Level geben Abschnitte an, in denen sich die Containeranzahl zum aktuellen Zeitpunkt befinden kann. Diese Grenzen wurde, mithilfe von Variablen festgelegt und mit dem Kürzel *CB0* bis *CB2* versehen, dies steht für den Begriff „*Container Bound*“. Die aktuelle Containeranzahl (engl. *Container Count*) selbst wurde in der Tabelle mit *CC* abgekürzt. Wenn im System zu einem betrachteten Zeitraum

beispielsweise fünf Containerinstanzen einer Konsumenten-Komponente laufen (im nachfolgenden stets als *Consumer-Container* bezeichnet), dann befindet sich das System im Zustand *Container Level 1*, da die untere Grenze des Intervalls mit *CB0* auf 1 und die obere auf 5 gesetzt wurde.

Eine ähnliche Eingrenzung gibt es ebenfalls bezüglich der vertikalen Achse, wobei die unbeantworteten Nachrichten einer Warteschlange in einem Message-Broker betrachtet werden. Hierbei steht das *QL0* bis *QL3* für das sogenannte „*Queue Level*“. Über die *Queue Bound* (kurz *QB0* bis *QB3*) werden hierbei die Grenzen der Intervalle festgelegt. Wenn beispielsweise 101 unbeantwortete Nachrichten in der Warteschlange residieren, befindet sich das System im Zustand *QL3*, da die *Queue Count* (kurz *QC*) größer ist als die Grenze *QB2* (festgelegt auf 100).

Aus einer Überkreuzung dieser beiden Informationen lässt sich ableiten wie das System skaliert werden soll. Um beim beschriebenen Beispiel zu bleiben, würde sich das System im Feld *CL1QL3* befinden. Hierbei gibt es zu wenig Container für die gegebene Nachrichtenanzahl. Die Containeranzahl muss demnach nach oben skaliert werden und zwar um die Differenz der beiden Container-Intervall-Grenzen *CB1* und *CB3*. Wenn diese Skalierung durchgeführt wurde, befindet sich das System anschließend im Zustand *CL3QL3*. Die Hauptdiagonale der Tabelle beschreibt die Idealzustände; hierbei muss keine weitere Skalierung vorgenommen werden.

Ein wesentlicher Nachteil dieser Herangehensweise ist, dass das System beim Skalieren über mehrere Intervallgrenzen hinweg mit mehreren Zellen in Berührung kommt. So befindet sich das System bei der beschriebenen Skalierung beispielsweise für einen Moment im Zustand *CL2QL3*. Je nach Konfiguration des Systems schlägt diese Regel hierbei an, selbst wenn sich das System bereits in einem Skalierungsprozess befindet. Es gibt mehrere Möglichkeiten dieses Problem zu umgehen. Es ist zum Beispiel möglich das zeitliche Anfragenintervall durch Steuerungsinformationen vom Broker zu erhöhen, in der Hoffnung, dass das System bereits die gezielte Containeranzahl erreicht hat, sobald die nächste Anfrage erfolgt. Dies hat allerdings zur Folge, dass das System generell sehr lange braucht um zu erkennen, ob eine Skalierung bei der nächsten Aktion nötig ist oder nicht. Da dies einen impraktikablen Ansatz darstellt, wurde diese Verantwortlichkeit an den Proxy-Scaler übergeben. Da diese Proxy-Komponente die tatsächlichen Skalierungsanfragen an die Docker-Scaler-API navigiert, ist sie ebenfalls in der Lage entsprechende Nachrichten zurückzuhalten, wenn klar ist, dass es sich hierbei um einen „*Fehlalarm*“ handelt, bei dem lediglich eine Skalierungsstufe während eines bereits laufenden Prozesses durchlaufen wird.

Ein Satz Sonderregeln, die bisher nicht erwähnt wurden, betreffen den Initialisierungszustand. Hierbei wird geprüft, ob die Minimalanzahl von Containern läuft. Falls dies nicht der Fall sein sollte, wird eine Skalierungsanfrage für genau diese Anzahl generiert.

Konfiguration

```
1  groups:
2  - name: springScaleAlert
3    rules:
4      - alert: spring_baseline
5        expr: >
6          org_apache_activemq_Broker_ConsumerCount{
7            brokerName="localhost",
8            destinationName="${AMQ_SPRING_QUEUE_NAME}",
9            destinationType="Queue",
10           instance="activemq:8080",
11           job="services"
12         } < ${SCALING_CB0}
13
14 ...
```

Listing 5.6: Umgebungsvariablen - Prometheus Regelsatz

Dieser Ausschnitt aus dem hinterlegten Regelwerk stellt die als letztes erwähnte Sonderregel für die minimale Containeranzahl dar. Hiermit wird beispielhaft beschrieben, wie die erläuterten Regeln aus der Tabelle codiert werden. Das Schlüsselwort *groups* dient allein dazu verwandte Regeln zusammenzufassen. Der Rest ist selbsterklärend. Erwähnenswert ist, dass der Wert des Schlüssels *alert* in dieser Form ohne Modifikationen an den Alert-Manager weitergeleitet und dort entsprechend verarbeitet wird.

Im folgenden Listing ist die Dateistruktur der Konfigurationsdateien von Prometheus dargestellt.

```
1 $ tree stack/data/prometheus/ --charset=ascii
2 stack/data/prometheus/
3 |-- alert-unparsed.yml
4 |-- alert.yml
5 |-- prometheus.yml
6 `-- targets
7     `-- alertmanager.json
```

Listing 5.7: Prometheus – Konfigurationsstruktur

In der Datei „*alert-unparsed.yml*“ werden die Regeln nach der bisher beschriebenen Struktur abgelegt. Hierbei wurden keinerlei feste Werte für die Intervallgrenzen eingesetzt. Zur Laufzeit der Komponente ist dies allerdings nicht zulässig. Um dennoch die Flexibilität des Formats mit Umgebungsvariablen nutzen zu können wurde ein Skript verfasst, das vor dem Starten der Komponente ausgeführt wird und die Datei „*alert.yml*“ generiert. Bei Ausführung des Skripts werden die hinterlegten Werte der Umgebungsvariablen in das gegebene Skript eingesetzt[7].

In der „*prometheus.yml*“ Datei befindet sich ein Eintrag, der dem System mitteilt, dass diese generierte Datei für die Regelauswertung verwendet werden soll. In der *prometheus.yml* Datei werden ansonsten weitere Einstellungen bezüglich des Scraping-Intervalls, der Adressierung der zu überwachenden Komponenten sowie die Angabe des Namens des Alert-Managers konfiguriert[8].

Der Ordner *target* enthält die Routing Konfiguration für die Anbindung der generierten Alerts zur Alert-Manager-Komponente.

5.2.7 Alter-Manager

Diese Komponente ist in diesem Projekt sehr unkompliziert gehalten. Es wird lediglich ein Webhook erstellt, an den erhaltene Alerts ohne jegliche Modifikation weitergeleitet werden. Die komplette Konfiguration der Komponente ist unter folgender Url einsehbar: `https://github.com/derMacon/serverless-bsc-thesis/blob/main/anhang/refs/alertmanager.yml`

5.2.8 Scaler Proxy

Im folgenden UML-Diagramm (siehe Abbildung 5.8 Scaler-Proxy UML) wird die grundlegende Klassenstruktur der Scaler-Proxy-Komponente zusammengefasst.

Diese Komponente stellt sowohl die Funktionalität bereit, Alerts vom Alert-Manager, als auch direkte Skalierungsanfragen vom Benutzer selbst anzunehmen. Für beide Szenarien gibt es einen entsprechenden Endpunkt. Beide Typen von Anfragen werden intern allerdings erst in ein Zwischenformat überführt, das wiederum vom *RequestService* interpretiert werden kann. Die wesentliche

[7] Datei: `https://github.com/derMacon/serverless-bsc-thesis/blob/main/anhang/refs/alert-unparsed.yml`
[8] Datei: `https://github.com/derMacon/serverless-bsc-thesis/blob/main/anhang/refs/prometheus.yml`

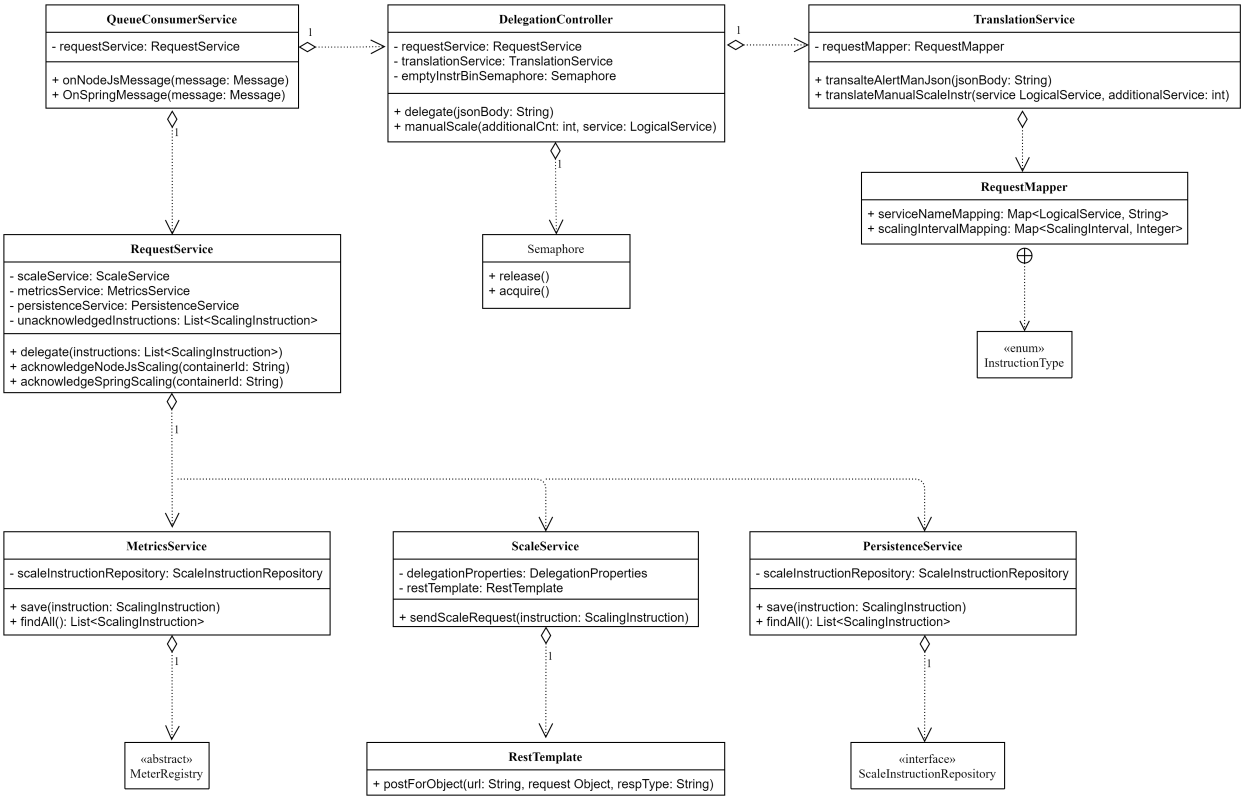


Abbildung 5.8: Scaler-Proxy UML

Kernfunktionalität dieses Services wurde in Listing 5.8 Proxy Scaler – RequestService dargestellt. Bei der Implementierung des Regelwerks in der Prometheus-Komponente wurde erwähnt, dass es einen Mechanismus zum Skalieren über mehrere Stufen des Models von der Proxy-Komponente geben muss (siehe Abschnitt 5.2.6 Regelsatz für Skalierung). Dies wurde in der Proxy-Komponente über ein Acknowledgement der einzelnen hochgefahrenen Container ermöglicht. Es wird intern eine Liste von Skalierungsinstruktionen verwaltet. Jeder Container, der im Zuge einer der Instruktionen gestartet wird, generiert eine ID, die diesen Container identifiziert und gibt diese dem Proxy-Service über eine Rest-Schnittstelle bekannt. Dadurch kann die Proxy-Komponente festzustellen, ob sich das System aktuell in einem Skalierungsprozess befindet oder nicht. Wenn nun neue Skalierungsinstruktionen eingespeist werden (zum Beispiel in einer der beschriebenen Übergangsphasen in der Tabelle), dann ist es der Komponente möglich, mithilfe einer einfachen Überprüfung der Liste diese zu verwerfen (Zeile 2). Im Anschluss wird über sämtliche generierten Skalierungsinstruktionen iteriert. Die verwaltete Liste wird mit neuen Einträgen gefüllt und die Anfragen werden an die Skalierungs-API geschickt, die als letztes Abstraktionslevel ohne weitere Funktionalität die Skalierung für den Benutzer übernimmt.

```
1  public boolean delegate(List<ScalingInstruction> instructions) {
2    if (!unacknowledgedInstructions.isEmpty()) {
3        return false;
4    }
5    boolean scaledToMinRepl = false;
6    for (ScalingInstruction instruction : instructions) {
7        instruction.setReceivedRequestTimestamp(now());
8        if (instruction.getScalingDirection() == UP) {
9            unacknowledgedInstructions.add(instruction);
10       }
11       scaledToMinRepl = scaledToMinRepl || scaleService.sendScaleRequest(instruction);
12   }
13   return scaledToMinRepl;
14 }
```

Listing 5.8: Proxy Scaler – RequestService

5.2.9 Docker-Scaler

Dieses Open Source Projekt[9] bietet eine Rest-Schnittstelle über die es möglich ist einen zugrunde liegenden Docker-Swarm zu skalieren. Die komplette Anfrage kann innerhalb einer einzigen POST-Anfrage abgeschickt werden. In diesem Projekt wurde hierfür der folgende Endpunkt verwendet.

- URL: /v1/scale-service
- Methode: POST
- Parameter:
  - service: Name des zu skalierenden Services
  - scale: Richtung in die skaliert werden soll
  - by: Anzahl zusätzlicher / überflüssiger Replicas

5.2.10 Grafana

Dieses Werkzeug wird zur Erstellung von Graphen hinsichtlich erhaltener Metriken verwendet. Diese Daten werden von allen relevanten Stack-Komponenten über eine einheitliche Schnittstelle bereitgestellt und im Prometheus Client gebündelt. In der Grafana-Komponenten wird hierbei lediglich die Prometheus-Komponente als *Datasource* deklariert, um Zugriff auf diese Metriken zu erhalten [Bra18, Seite 99 ff.]. In Abbildung 5.9 Prometheus - Datesource ist diese Konfiguration in Auszügen zu erkennen. Hierbei wird im Wesentlichen die Netzwerk-Adresse sowie der Port spezifiziert.

Über die in Prometheus deklarierten *Targets* (überwachten Komponenten) können die wichtigen Informationen ausgelesen werden. Die Abbildung 5.10 zeigt hierbei auf, wie das fertige Dashboard aussieht. Die Aufnahme wurde zum Zeitpunkt der Benchmarks-Tests, die als Grundlage für diese Analyse verwendet wurden, getätigt. Es lassen sich alle Metriken der Ergebnisanalyse (siehe Abschnitt 6 Ergebnisanalyse) in diesem Dashboard wiederfinden.

Um ein Dashboard zu erstellen, wird auf das Webinterface der Komponente selbst zugegriffen. Es gibt hierbei zwar auch die Möglichkeit Dashboards im JSON-Format einzulesen sowie zu

[9] `https://thomasjpfan.github.io/docker-scaler`

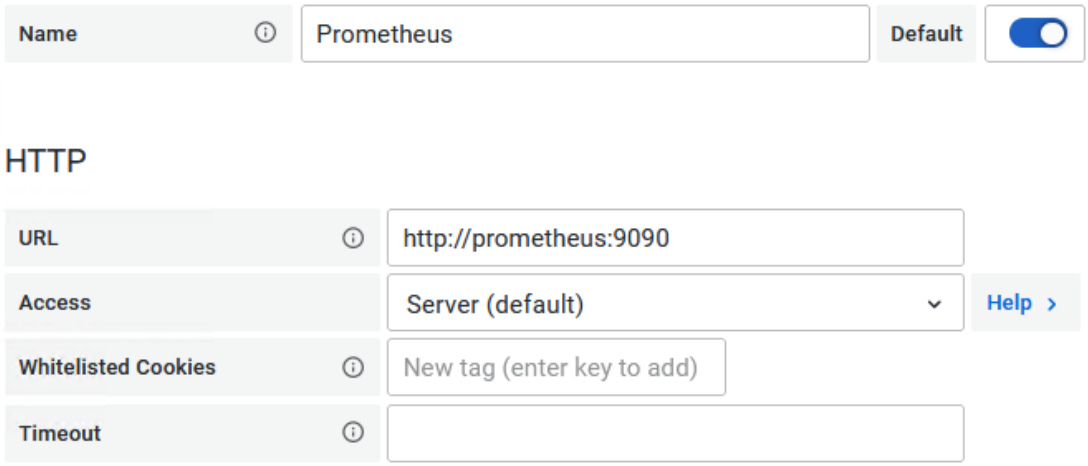


Abbildung 5.9: Prometheus - Datesource

exportieren, der Einfachheit halber wurde in diesem Projekt allerdings hierauf verzichtet. In der folgenden Abbildung ist zu erkennen, wie die Konfiguration einer Dashboard-Komponente über das Webinterface funktioniert (siehe Abbildung 5.11).

Das wesentliche Formularelement, das während der Konfiguration verwendet wird, stellt die Anfrage an die definierte Datasource dar (*query*). Hierbei wird auf eine Anfragesprache namens *PromQL* zurückgegriffen um verschiedene Metriken auslesen zu können. In der Abbildung ist beispielsweise zu erkennen, dass bei einem Messagebroker mit dem Namen „*localhost*“ eine Warteschlange (engl. *queue*) mit dem Namen „*springqueue*“ mit der Adresse „*activemq:8080*“ die aktuelle Größe der Warteschlange ausgeben soll[10]. Diese Anfrage wird in einem konfigurierbaren Intervall wiederholt (hier alle 15 Sekunden). Es ist auch möglich mehrere Anfragen pro Graph zu definieren um mehrere Werte gegenüberzustellen sowie mehrere Anfrageergebnisse zu verknüpfen. Die erhaltenen Daten werden anschließend automatisch in einem Graphen dargestellt (siehe Abbildung 5.12).

In diesem Beispiel wurde eine größere Menge von Nachrichten für den Spring-Konsumenten in die Warteschlange geladen. Die unbeantworteten Nachrichten wurden hierbei in Gelb dargestellt. Da zum Start der Anwendung nur ein einziger Container des Services aktiv läuft, werden im Nachhinein weitere Container hochgefahren (siehe orange Kurve). Es ist zu erkennen, dass die Nachrichtenanzahl zu Anfang sehr langsam abnimmt, mit ansteigender Containeranzahl jedoch immer schneller sinkt.

Die Daten hinsichtlich der Node.js Komponenten sowie die Zahlenwerte für die gemessenen Start-up-Zeiten werden gleichermaßen in eigenen Dashboard-Komponenten dargestellt. Da die Konfiguration dieser, der Vorgestellten allerdings deutlich ähnelt, wurde hierbei darauf verzichtet diese ebenfalls zu erläutern. Da das Dashboard im Allgemeinen einen eher überprüfenden Charakter einnimmt um nachvollziehen zu können wie die letztendlichen Messwerte berechnet wurden, ist diese gesamte Komponente auch von nebensächlicher Bedeutung.

5.3 Deployment auf der Containerplattform

Nachdem im letzten Abschnitt beschrieben wurde, wie die einzelnen Stack-Komponenten implementiert wurden, wird im Folgenden auf das Deployment mithilfe von Containern im Docker-Swarm-Stack eingegangen.

[10] Die hier genutzte Anfrage lässt sich zum Beispiel bei der ActiveMq-Dokumentation nachlesen `https://docs.splunk.com/Observability/gdi/activemq/activemq.html`

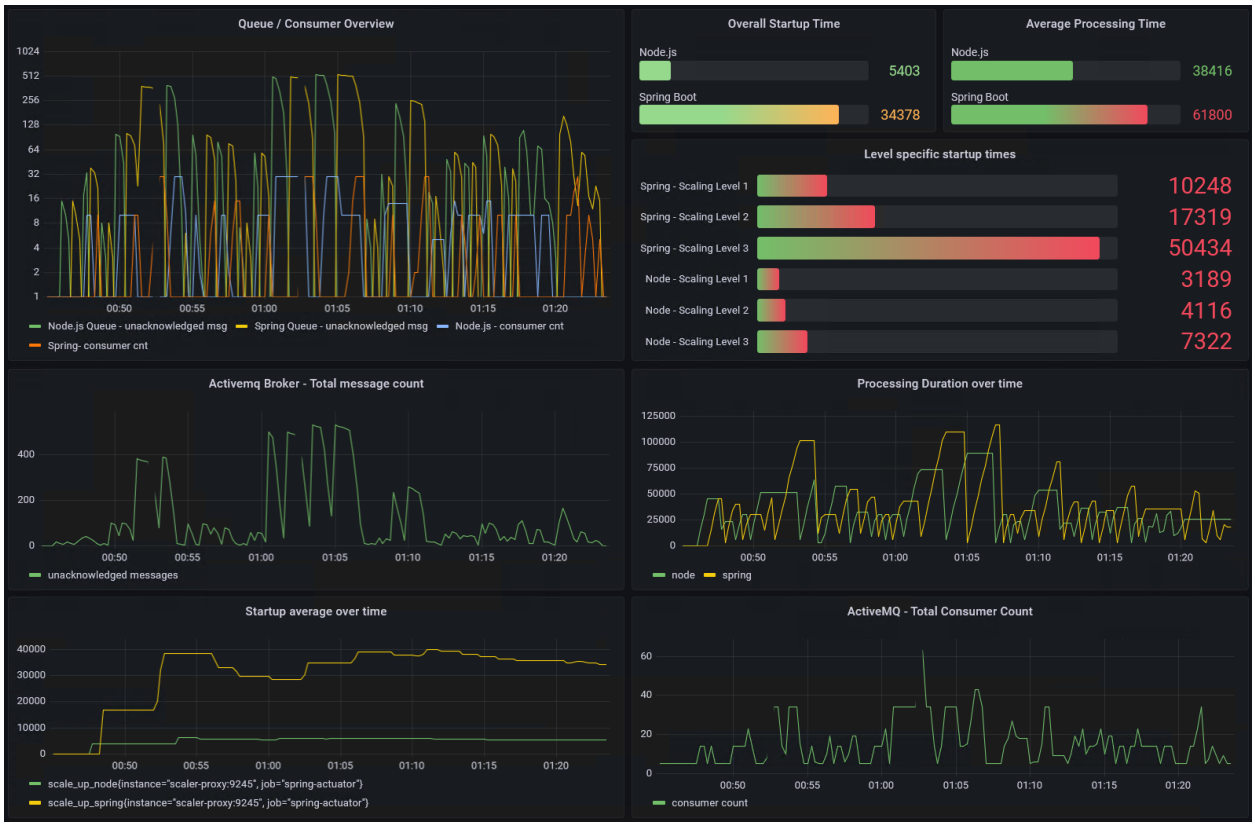


Abbildung 5.10: Grafana Dashboard

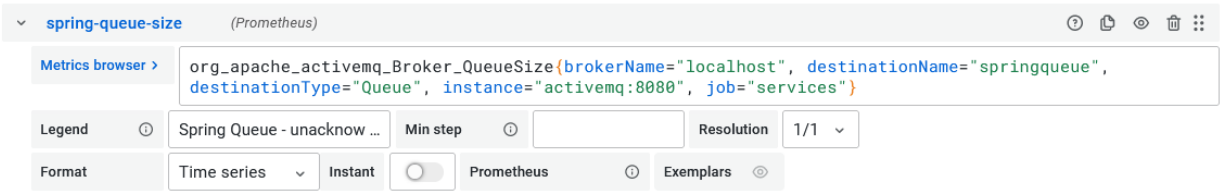


Abbildung 5.11: Grafana - Graphenkonfiguration

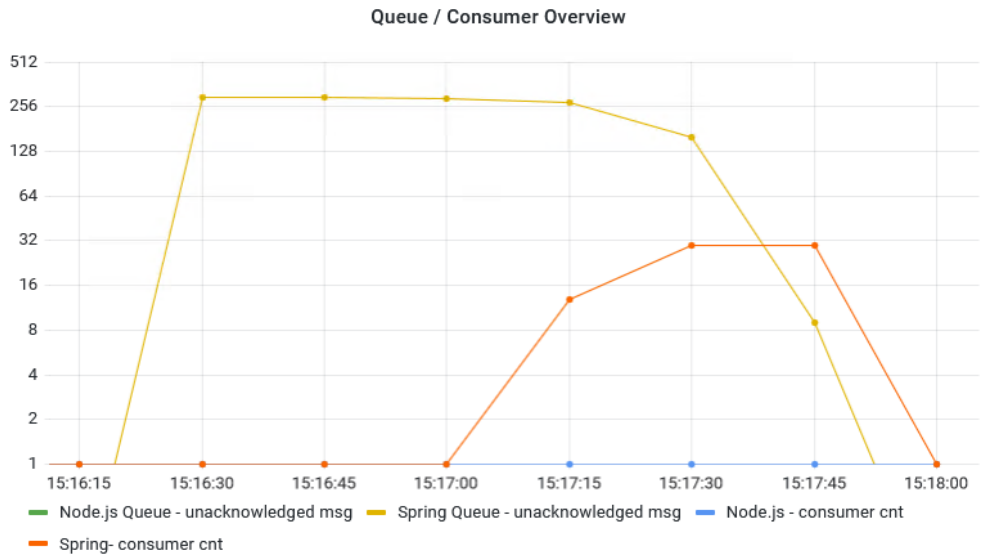


Abbildung 5.12: Grafana - Beispielhafter Graph

5.3.1 Build

Hierbei werden zuerst die Buildvorgänge der einzelnen Komponenten beschrieben, im Anschluss daran wird ein Überblick über die Orchestrierung des Stacks gegeben.

Spring Komponenten

Um das Projekt im Komponenten-Stack als Dockercontainer zu deployen, muss in einem ersten Schritt ein Docker-Image erstellt werden. Dieses kann im Anschluss als Container instanziiert werden. Zum Erstellen eines Docker-Images wird auf ein sogenanntes „*Dockerfile*“ zurückgegriffen, das die Beschreibung des Build-Prozesses für dieses Image enthält. Da alle Spring-Projekte im Komponenten-Stack mit einem fast identischen Dockerfile versehen wurden, gilt die anschließende Erläuterung in gleichem Maß für alle Spring-Projekte dieses Stacks.

Damit die folgende Erklärung der genutzten Dockerfiles verständlich dargestellt werden kann, folgt eine kurze Zusammenfassung der Struktur eines Docker-Images. „*A Docker image is made up of filesystems layered over each other*“ [Tur14, Seite 71]. Als erste Schicht wird hierbei ein bootbares Filesystem eingeführt, das einem typischen Linux-Bootsystem ähnelt. Anschließend kann dieses Modell mit weiteren Schichten erweitert werden. Hierfür wird der sogenannte „*union mount*“ Mechanismus verwendet, der es ermöglicht mehrere read-only Dateisysteme in das Root-Dateisystem zu integrieren. Bei einem neu hinzugefügten Dateisystem ist es möglich auf die Unterordner der vorher hinzugefügten Systeme zuzugreifen, umgekehrt gilt dies nicht. In einer Docker-Umgebung spricht man bei diesen aufgeschichteten Dateisystemen von Docker-Images. „*Images can be layered on top of one another. The image below is called the parent image and you can traverse each layer until you reach the bottom of the image stack where the final image is called the base image. Finally, when a container is launched from an image, Docker mounts a read-write filesystem on top of any layers below. This is where whatever processes we want our Docker container to run will execute*“ [Tur14, Seite 71]. In Abbildung 5.14 Docker Volumes – Schaubild wurde dies auch noch einmal grafisch aufgearbeitet.

Um ein Docker-Image zu erstellen, wurde bei diesem Projekt auf eine Beschreibungsdatei namens „*Dockerfile*“ zurückgegriffen.

```
1  FROM maven:3.8.1-openjdk-11 AS build
2  COPY src /usr/src/app/src
3  COPY pom.xml /usr/src/app
4  RUN mvn -f /usr/src/app/pom.xml clean package -DskipTests
5
6  FROM gcr.io/distroless/java
7  COPY --from=build /usr/src/app/target/supplier-backend.jar /usr/app/supplier-backend.jar
8  EXPOSE 9245
9  ENTRYPOINT ["java","-jar","/usr/app/supplier-backend.jar"]
```

Listing 5.9: Dockerfile - Supplier

- Zeile 1: Ein Dockerfile beginnt mit der Angabe eines sogenannten „*Baseimages*“. Dies kann eine minimale Linux Distribution sein, oder wie in diesem Fall ein System, auf dem bereits diverse Konfigurationen definiert wurden. Bei dem verwendeten Image wurde eine Java Runtime sowie das Buildprogramm *Maven* bereits vorinstalliert. Beides wird für die Ausführung des Spring-Projekts gebraucht.

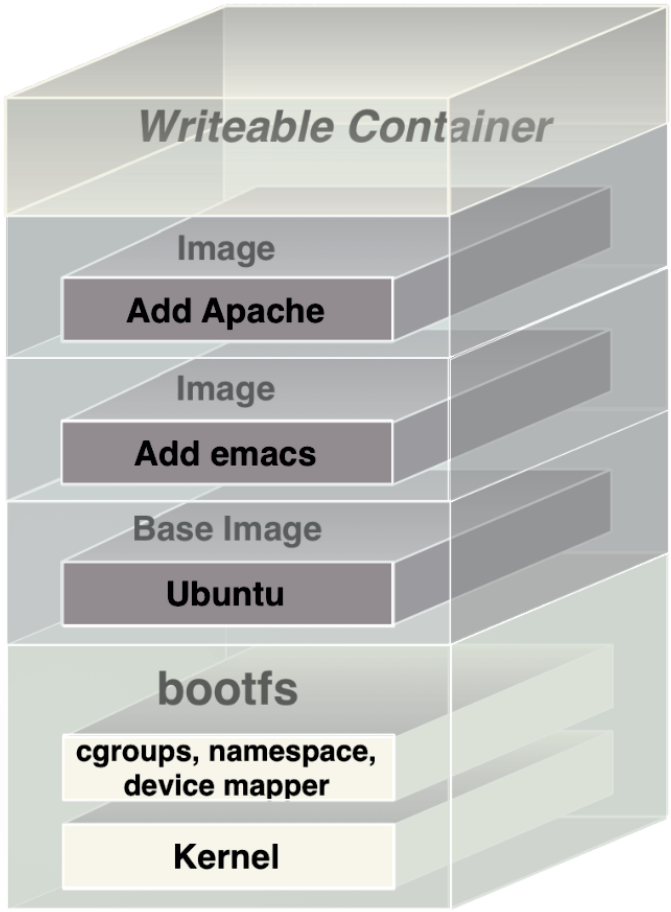


Abbildung 5.13: Docker Image – Schaubild [Tur14, Seite 72]

- Zeile 2: Der Buildprozess wird in dem Root-Verzeichnis des Spring-Projekts ausgeführt. Hier befinden sich in einem Unterordner alle Quellcodedateien, die mit diesem Befehl in das *usr* Verzeichnis des zu erstellenden Images kopiert.
- Zeile 3: Da alle Spring-Projekte dieses Stacks mithilfe des Build-Tools *Maven* gebaut werden, wird die *pom.xml* Beschreibungsdatei aus dem Root-Verzeichnis ebenfalls in den Container kopiert. Diese Datei enthält alle wesentlichen Metainformationen des Projekts. Dazu gehören beispielsweise sämtliche Abhängigkeiten, verwendete Build-Plugins und Projektinformationen.
- Zeile 4: Der Run-Befehl ermöglicht es den nachfolgenden Shell-Befehl in einer Linux-Shell innerhalb des Containers selbst auszuführen. Hierbei wird die ausführbare .jar Datei gebaut, die im späteren Verlauf im Dateisystem deployed wird.
- Zeile 6: Es folgt eine erneute Anweisung zum Laden eines Base-Images, dies ist Teil eines sogenannten *multi-stage* Builds. Wie im letzten Abschnitt erwähnt, werden beim Bauen von Docker-Images verschiedene Dateisysteme ineinander gemounted, hierbei entsteht eine Art Schichtenmodell. Bezogen auf ein Dockerfile wird mit jeder einzelnen Befehlszeile der Datei ein neues Image erstellt. Um am Ende ein möglichst kleines Docker-Image zu erhalten, wird versucht diese Schichtanzahl so minimal wie möglich zu halten. Hierbei gibt es die Möglichkeit, das Bauen der ausführbaren Datei auszulagern und lediglich die fertige ausführbare Datei in den Container zu laden. Mit diesem Ansatz ist es allerdings nicht möglich, das Image direkt auf dem Server zu bauen, es sei denn die Datei liegt vor oder das System stellt alle

notwendigen Buildwerkzeuge bereit. Dies ist sehr impraktikabel, weswegen in der Praxis hierbei ein sogenannter „*multistage build*“ verwendet wird [doc, Kapitel multi stage build]. Hierbei kann auf einzelne Stufen Bezug genommen werden und spezifizierte Dateien von einem Image direkt in ein anderes kopiert werden. Bezogen auf das dargestellte Dockerfile, ist es möglich die ausführbare Datei des vorherigen Buildprozesses zu kopieren ohne sämtliche Metadaten, die bei dem Buildprozess anfallen, ebenfalls kopieren zu müssen.

- Zeile 7: Wie im letzten Schritt erwähnt, wird hierbei die ausführbare Datei vom Buildprozess in das aktuelle Image herüberkopiert. Hierfür wird sich des Schlüsselworts *–from* bedient. Die vorher generierten Schichten, die für im Buildprozess verwendet werden, werden bei der Erstellung des letztendlichen Images wieder verworfen, da sie mit einem neuen Baseimage überschrieben wurden.

- Zeile 8: Der Port, über den sämtliche Kommunikation mit der Außenwelt stattfindet, wird freigegeben.

- Zeile 9: Der Entrypoint ähnelt dem *CMD*-Befehl stark, es ist zum Ausführungszeitpunkt des Containers allerdings schwieriger diesen zu überschreiben.

Anmerkung: Vorteile der Imageschichtung Der wesentliche Vorteil durch die Schichtung der Images während des Buildprozesses ist der, dass es so einfacher möglich ist, Images zu cachen. Es müssen schließlich nur die Teile des Images neu gebaut werden, die sich tatsächlich geändert haben. Dazu wird die Schicht ermittelt, ab der eine erste Abweichung zum vorherigen Build festgestellt werden kann. Anschließend wird lediglich diese Schicht, sowie alle Schichten darüber neu gebaut. Die *parent layer* werden allerdings weiterhin wiederverwendet. Beim Erstellen von Images sollte daher Wert darauf gelegt werden, den dynamischen Teil des Buildvorgangs möglichst weit an das Ende eines Dockerfiles zu legen, um den Caching-Mechanismus der Docker-Engine bestmöglich auszunutzen.

Node.js

Überblick Das Node.js-Projekt wurde, mithilfe des npx „*create-react-app*“-Starters initialisiert. Npx ist ein package runner für den node package manager (kurz npm). Npx stellt dabei ein CLI-Werkzeug bereit, welches bei Installation von npm direkt mit ausgeliefert wird und dient zur Unterstützung zur Verwaltung von Abhängigkeiten. Im Bezug auf das Node.js-Projekt ist es beispielsweise möglich hiermit ein Projekt zu initialisieren, das auf der Programmiersprache *Typescript* basiert und nicht wie herkömmlich, auf Javascript. Typescript stellt eine Erweiterung von Javascript dar, es ist demnach möglich Javascript-Code in einem Typescript-Projekt auszuführen. Es wurde sich für Typescript entschieden, da diese Sprache die vier wesentlichen Prinzipien objektorientierter Sprachen bereitstellt. Diese setzten sich aus der *Kapselung von Code*, der *Vererbung*, *Abstraktion* und *Polymorphie* zusammen [typa]. Dies ermöglicht eine klarer strukturierte Codebasis und ist über ein objektorientiertes Modell ebenfalls einfacher mit dem Spring Boot Pendant zu vergleichen.

Kompilieren der Sourcen Ähnlich wie Java wird auch ein Typescript-Projekt kompiliert, hierbei erfolgt die Kompilierung allerdings in Javascript-Code. Die Konfiguration des Typescript-Compiler *tsc* befindet sich in einer Datei namens *tsconfig.json* (siehe Listing 5.10 tsconfig.json).

```
{
"compilerOptions": {

  /* Basic Options */
  "target": "es5",

  /* Specify ECMAScript target version: 'ES3' (default), 'ES5', 'ES2015', 'ES2016', 'ES2017', '
      ES2018', 'ES2019', 'ES2020', or 'ESNEXT'. */
  "module": "commonjs",
  "outDir": "./dist",
  "rootDir": ".",

  /* Strict Type-Checking Options */
  "strict": true,
  "esModuleInterop": true,

  /* Advanced Options */
  "skipLibCheck": true,
  "forceConsistentCasingInFileNames": true
}
}
```

Listing 5.10: tsconfig.json

Die wichtigsten Konfigurationsparameter im Überblick:

- *target*: Hierbei wird der Javascript-Standard angegeben, in den die Typescript-Quellen übersetzt werden sollen.
- *outdir*: Ausgabeverzeichnis in dem die übersetzten Javascript-Dateien abgelegt werden sollen.
- *rootdir*: Root-Verzeichnis des Typescript-Projekts.
- *strict*: Compilerflag hinsichtlich der Typisierung und sonstiger Javascript-Eigenheiten. Ersetzt die Notwendigkeit, in jeder einzelnen Quellcodedatei einen „*use strict*“ Befehl angeben zu müssen, um die Datei im ECMAScript *strict mode* zu parsen[11].

Ähnlich wie bei einem Maven-Projekt eine *pom.xml* Datei verwendet wird, um die Projekt-Konfigurationseinstellungen festzuhalten, wird bei einem Typescript-Projekt eine sogenannte *package.json* erstellt (siehe Listing 5.11 Typescript – package.json, die komplette Datei ist unter folgender Url einsehbar: `https://github.com/derMacon/serverless-bsc-thesis/blob/main/anhang/refs/package.json`

[11] für weiter Informationen siehe Dokumentation [typb, Kapitel tsconfig]

```
1  {
2  "name": "node-consumer",
3  "main": "index.js",
4  "scripts": {
5    "test": "echo \"Error: no test specified\" && exit 1",
6    "build": "tsc"
7  },
8  "author": "hoffmann",
9
10 ...
11
12 "dependencies": {
13   "@types/node": "^15.6.1",
14   "@types/pg": "^8.6.0",
15   "dotenv": "^10.0.0",
16   "fs": "^0.0.1-security",
17   "ini": "^2.0.0",
18   "libxmljs2": "^0.27.0",
19   "pg": "^8.6.0",
20   "stompit": "^1.0.0",
21   "typescript": "^4.3.2",
22   "xmldom": "^0.6.0",
23   "xpath": "^0.0.32"
24 },
25 "devDependencies": {
26   "@types/ini": "^1.3.30"
27 }
28 }
```

Listing 5.11: Typescript – package.json

In dem angegebenen Dateiinhalt lassen sich vor allem Metainformationen des Projekts identifizieren (Projektname, Autor). Außerdem werden Projekt-Abhängigkeiten definiert, wobei die letztendlich genutzten Packages mit den entsprechenden Versionsnummern in einem dedizierten Format aufgezählt werden. Diese sind standardmäßig Javascript Projekte. Da Typescript dennoch die Möglichkeit bietet, über einen definierten *any type* eine untypisierte Variante von Variablen etc. zu erstellen, können diese Bibliotheken ohne weitere Konfiguration genutzt werden. Um allerdings die Typisierungsfunktionen von Typescript sinnvoll nutzen zu können, müssen diese Informationen entweder als getrennte Abhängigkeit getrennt importiert werden (siehe Zeile 13 / 14) oder sie sind bereits in der importierten Bibliothek enthalten und können nativ genutzt werden. Zur Überprüfung kann die Ausgabe des TSC-Compilers zurate gezogen werden.

Dockerbuild Um ein Dockerimage für das Typescript-Projekt zu erstellen, wird auf folgendes Dockerfile zurückgegriffen 5.12 Dockerfile - Typescript Projekt.

```
1  # stage 1 building the code
2  FROM node:10.15.3 AS builder
3  WORKDIR /usr/app
4  COPY package*.json ./
5  RUN npm install
6  COPY . .
7  RUN npm run build
8
9  # stage 2
10 FROM node:10.15.3-alpine
11 WORKDIR /usr/app
12 COPY package*.json ./
13 RUN npm install --production
14 COPY --from=builder /usr/app/dist ./dist
15 COPY --from=builder /usr/app/schema.sql .
16 COPY --from=builder /usr/app/specification.xsd .
17 CMD node dist/src/index.js
```

Listing 5.12: Dockerfile - Typescript Projekt

Dieses verhält sich relativ ähnlich zu dem der Spring-Boot-Projekte (siehe 5.2.4 Generelle Spring Projektübersicht). Es werden ausführbare Quellen erstellt, die wiederum in einem zweiten Build-Schritt des multistage Buildverfahrens (siehe 5.3.1 Spring Komponenten) verwendet werden. Der wesentliche Unterschied beim ersten Schritt ist der, dass hierbei nicht eine einzelne ausführbare Datei erstellt wird, sondern die entsprechenden Javascript-Verzeichnisse generiert werden. Hierzu werden alle für den Build relevanten Typescript- und sonstige Quellen in einem Zwischenschritt in einem Containerimage abgelegt. Anschließend erfolgt der Compileraufruf. Im zweiten Schritt werden lediglich die relevanten Quellen herüberkopiert und der Rest verworfen.

5.3.2 Komponentenstack

Um den Stack zu starten wird auf verschiedene Skripte sowie Beschreibungsdateien zugegriffen. Im Folgenden wird erst die Beschreibung des Container-Netzwerks und anschließend der Build- sowie Start-Vorgang des Projekts beschrieben.

Docker-Compose

Einleitung Es ist möglich, ein Netzwerk von Containern manuell über das CLI der Docker-Engine zu starten, hierbei können Mounts bestimmter Verzeichnisse angelegt, Kommunikations-Netzwerke definiert und Ports freigegeben werden. Mithilfe von Shell-Skripten wäre es möglich, diese Schritte zu automatisieren. Dieses Vorgehen erscheint aber schon bei kleineren Projekten sehr impraktikabel und unübersichtlich, hierbei wird daher auf Werkzeuge wie zum Beispiel *Docker-Compose*[12] zurückgegriffen. Dieses Tool bietet die Möglichkeit, über eine Beschreibungssprache einen Stack zu definieren, in dem sämtliche Abhängigkeiten bereits aufgelöst werden können.

Container Networking Container werden hierbei in Clustern organisiert die es ermöglichen mithilfe des *Docker-Link*-Mechanismus die Container über ein definiertes Netzwerk miteinander in Verbindung treten zu lassen. „*The Docker server acts as a virtual bridge and the containers are clients behind it*“ [MK15, Seite 13]. Jeder Container besitzt ein eigenes virtuelles Ethernet Interface, das eine eigene IP-Adresse allokiert und mit der beschriebenen Docker-Bridge verbunden ist. „*Docker lets you bind*

[12] `https://github.com/docker/compose`

ports on the host to the container so that the outside world can reach your container“. Hierbei wird ein lokales Netzwerk allokiert, über das alle Container direkt miteinander kommunizieren können.

Stack-Konfiguration Um beim Beispiel des *Suppliers* zu bleiben, folgt eine kurze Zusammenfassung der Stack-Konfiguration für den Spring-Supplier-Container. Die gesamte Docker-Compose Konfigurationsdatei befindet sich ebenfalls im Anhang (siehe Listing 5.13 Docker Compose - Ausschnitt Supplier Definition).

```
1
2   ...
3
4   supplier-backend:
5     image: supplier-backend
6     build: supplier-backend/
7     ports:
8       - "${SUPPLIER_BACKEND_PORT}:${SUPPLIER_BACKEND_PORT}"
9     restart: unless-stopped
10    networks:
11      - scaler
12    environment:
13      - DATABASE_HOST=history-db
14      - DATABASE_USER=${SUPPLIER_USER_NAME}
15      - DATABASE_PASSWORD=${SUPPLIER_USER_PASSWORD}
16      - DATABASE_NAME=${SUPPLIER_DATABASE_NAME}
17      - DATABASE_PORT=${SUPPLIER_DATABASE_PORT}
18      - AMQ_NODE_QUEUE_NAME=${AMQ_NODE_QUEUE_NAME}
19      - AMQ_SPRING_QUEUE_NAME=${AMQ_SPRING_QUEUE_NAME}
20      - AMQ_BROKER_HOSTNAME=activemq
21      - AMQ_BROKER_PORT=${AMQ_TCP_PORT}
22      - SERVER_PORT=${SUPPLIER_BACKEND_PORT}
23      - SPRING_PROFILES_ACTIVE=prod
24
25  ...
```

Listing 5.13: Docker Compose - Ausschnitt Supplier Definition

Die Konfigurationsdatei wurde im yaml Dateiformat erfasst, wobei es hier ein Key-Value Mapping zwischen dem Namen eines zu setzenden Konfigurationsparameters und dem definierten Wert gibt.

- *image*: über diesen Schlüsselwert wird der Name des zugrunde liegenden Images für den zu erstellenden Container gesetzt.
- *build*: gibt an, in welchem Verzeichnis die Quelltextdateien liegen. Über das CLI des Compose-Werkzeugs ist es möglich, nicht auf das vorhandene Image bei Containererstellung zurückzugreifen, sondern das Image neu zu bauen. Hierbei wird auf diesen Konfigurationsparameter zurückgegriffen.
- *networks*: es wird ein Netzwerk referenziert, in dem dieser Container agieren soll. Alle Container, mit denen kommuniziert wird, werden über einen Identifier dem gleichen Netzwerk übergeben.
- *environment*: hierbei werden die aufgeführten Umgebungsvariablen in den Container geladen. Es ist irrelevant, ob es sich im Build-Kontext um tatsächliche Umgebungsvariablen handelt (siehe Zeile 14) oder ob die Werte an dieser Stelle erst definiert werden (siehe Zeile 13), für eine ausgeführte Anwendung innerhalb des Containers lässt sich kein Unterschied feststellen.

Neben den wesentlichen beschriebenen Konfigurationsparametern gilt es noch die *Volume* Definition (siehe Listing 5.14 Docker Compose - Volume Definition) zu erwähnen.

```
volumes:
  - consumer-db:/var/lib/postgresql/dat
```

Listing 5.14: Docker Compose - Volume Definition

Volumes stellen eine Möglichkeit dar, um Daten auf dem Datenträger des Hosts zu persistieren. Das folgende Schaubild gibt einen Überblick über die verfügbaren Varianten.

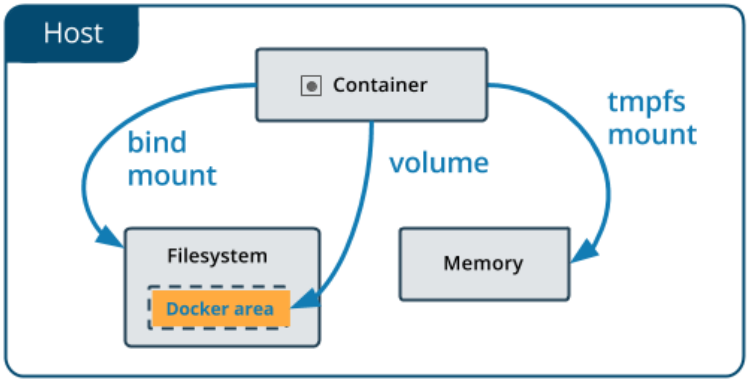


Abbildung 5.14: Docker Volumes – Schaubild [doc, Kapitel /storage/volumes/]

Daten, die über Volumes persistiert werden, sind vom Betriebssystem des Hosts komplett unabhängig. Die Daten können dabei entweder in dem *writeable layer* des Containers selbst gespeichert werden[13]. Diese Daten werden allerdings verworfen, sobald der Container gelöscht wird. Um Daten auch weiterhin zu behalten und in neue Container-Instanzen laden zu können, wird hierbei auf Volumes zurückgegriffen. Es gibt zwei Varianten von Volumes [doc, Kapitel /storage/volumes]:

1. *Named volumes*: have a specific source from outside the container, for example `awesome:/bar`.
2. *Anonymous volumes*: have no specific source so when the container is deleted, instruct the Docker Engine daemon to remove them.

Im Projekt wurde auf anonyme Volumes zurückgegriffen (siehe Listing 5.14 Docker Compose - Volume Definition). Diese werden direkt vom Docker-Daeomon verwaltet, wobei sich der Anwender damit nicht zu beschäftigen braucht. Neben den genannten Volume-Varianten gibt es ebenfalls die Möglichkeit, sogenannte *bind mounts* zu verwenden. Diese sind jedoch abhängig von der Verzeichnisstruktur des Hosts sowie vom Betriebssystem, welches auf dem Host ausgeführt wird und gelten deshalb als veraltet. In der Dokumentation von Docker selbst wird empfohlen Volumes zu verwenden.

Docker Swarm Mit diesem Werkzeug ist es möglich, den definierten Komponentenstack in einer skalierten Umgebung darzustellen. Das Tool agiert als Orchestrator[14]. Die Container agieren hierbei im sogenannten „*Swarm Mode*“. Dabei können mehrere Maschinen derartig agieren, dass es für den Anwender den Anschein erweckt, als würde er lokal auf der eigenen Maschine arbeiten, wenn in Wirklichkeit ein ganzes Rechner-Netzwerk mit dem Hosten der Container betraut werden kann. Im Zusammenhang mit dieser Thesis ist allerdings das Skalieren einzelner Services der primäre Vorteil des Tools. Es ist möglich mehrere sogenannte *Replicas* zu starten und auch wieder herunterzufahren, ohne

[13] Erinnerung: Alle Schichten des Images werden als *read-only* Systeme deployed, erst während der Initialisierungsphase des Containers wird eine weitere Schicht hinzugefügt, die während der Containerlaufzeit bearbeitet werden kann.
[14] Begriffserklärung siehe Abschnitt 4.3 Anforderungen an die Containerplattform / Orchestrierungsplattform

dass die restlichen Stack-Komponenten hiervon beeinflusst werden. Um dies nutzen zu können, muss der definierte Stack von diesem Werkzeug übernommen werden. Zum gegenwärtigen Zeitpunkt bietet das Swarm-Werkzeug allerdings lediglich teilweise Unterstützung für das Einlesen der docker-compose Beschreibungsdatei. Vor allem wird das Setzen von Umgebungsvariablen mithilfe definierter Variablen des Buildsystems nicht unterstützt. In einer Produktivumgebung muss es jedoch die Möglichkeit geben, Umgebungsvariablen über Variablen des Buildsystems zu definieren, die Alternative wäre ansonsten auf *hardcoded values* zurückzugreifen. Dies ist gerade hinsichtlich sensitiver Daten, zum Beispiel bei Anmeldedaten für Datenbanken etc., katastrophal. Um dennoch in der Lage zu sein bestimmte Konfigurationsparameter mit Umgebungsvariablen füllen zu können, gibt es die Möglichkeit das Docker-Compose-Werkzeug selbst als Präprozessor zu verwenden. Dabei wird das tool-konforme docker-compose Skript in ein Swarm-konformes Skript umgewandelt und die Umgebungsvariablen direkt mit den im System hinterlegten Werten ausgewechselt (siehe Schaubild 5.15 Docker Compose – Schaubild). Hierfür reicht ein entsprechender Aufruf über das CLI der Docker-Engine. Anschließend kann ein Swarm auf der Hostmaschine initialisiert und die Stack-Konfiguration deployed werden. In der Ausführung wird hierfür wieder auf ein Skript zurückgegriffen, was diese manuellen Shell-Befehle automatisiert[15].

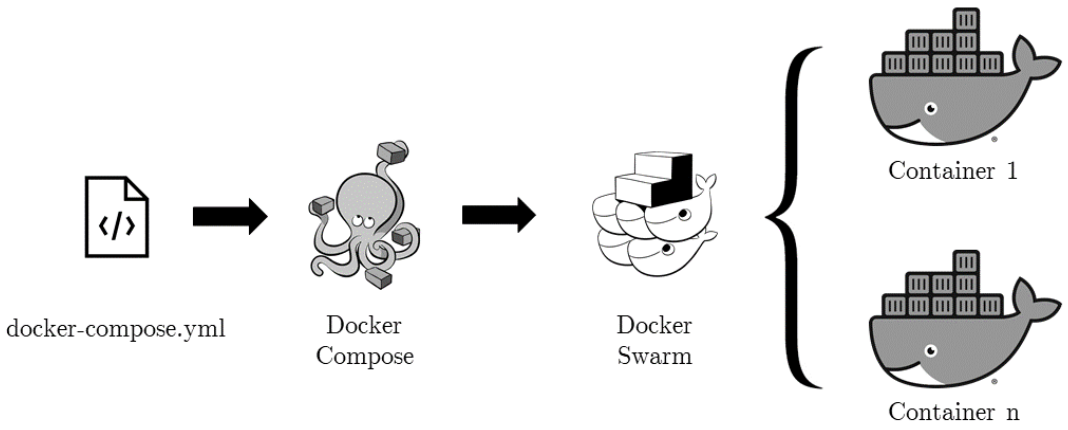


Abbildung 5.15: Docker Compose – Schaubild

```
1
2   ...
3
4   echo 'use docker-compose as preprocessor for environmental variables'
5   docker-compose -f $STACK_DIR/docker-compose.yml config > $STACK_DIR/docker-compose-parsed.yaml
6
7   if [ ! isSwarmNode ]; then
8     echo 'init swarm'
9     docker swarm init
10  fi
11
12  echo 'redeploy stack'
13  docker stack rm vossibility
14  docker stack deploy --compose-file $STACK_DIR/docker-compose-parsed.yaml vossibility
```

Listing 5.15: start-stack-services.sh

[15]kompletter Dateiinhalt unter folgender Url einsehbar: `https://github.com/derMacon/serverless-bsc-thesis/blob/main/anhang/refs/start-stack-services.sh`

Über den Befehl `docker stack services vossibility` lässt sich anschließend kontrollieren, ob der Stack gestartet ist und wie viele Serviceinstanzen jeder Komponente aktiv laufen. Die Ausgabe ist in Listing 5.16 zu erkennen.

```
$ docker stack services vossibility
ID            NAME                                MODE        REPLICAS  IMAGE
                                  PORTS
mct16nrd6e91  vossibility_activemq                replicated  1/1       bwolf/activemq-prometheus:
    latest *:8080->8080/tcp, *:8161->8161/tcp, *:61613->61613/tcp, *:61616->61616/tcp
izrbys8vxive  vossibility_alertmanager            replicated  1/1       prom/alertmanager:v0.20.0
          *:9093->9093/tcp
jibw9j22qsrd  vossibility_consumer-db             replicated  1/1       postgres:13-alpine
                *:9292->9292/tcp
rfocmjjuwzp8  vossibility_consumer-persistence    replicated  1/1       consumer-persistence:latest
        *:8965->8965/tcp
xu48nv8p1mvi  vossibility_grafana                 replicated  1/1       grafana/grafana:latest
             *:3000->3000/tcp
orsxfojfj7u3  vossibility_history-db              replicated  1/1       postgres:13-alpine
                *:7004->7004/tcp
yh3ykz5ixfaw  vossibility_node-consumer           replicated  1/1       node-consumer:latest
ubeyrrzvatpq  vossibility_pgadmin                 replicated  1/1       dpage/pgadmin4:latest
               *:5050->5050/tcp
7m4bpz6t166b  vossibility_prometheus              replicated  1/1       prom/prometheus:v2.21.0
            *:9000->9090/tcp
bfx1zqkg5oq0  vossibility_scaler                  replicated  1/1       thomasjpfan/docker-scaler:
    master *:8743->8080/tcp
32bd0chtwvjg  vossibility_scaler-proxy            replicated  1/1       scaler-proxy:latest
                *:9245->9245/tcp
twrq504kngog  vossibility_spring-consumer         replicated  1/1       spring-consumer:latest
              *:7143->7143/tcp
j57rwkmj7xr3  vossibility_supplier-backend        replicated  1/1       supplier-backend:latest
             *:8284->8284/tcp
f4wdecxpqxks  vossibility_supplier-frontend       replicated  1/1       supplier-frontend:latest
           *:7384->80/tcp
```

Listing 5.16: Dockerfile - Supplier

5.4 Implementierung des Lasttests

Nachdem im letzten Abschnitt auf die Konfiguration des Stacks eingegangen wurde, wird im Folgenden erklärt, welche Szenarien für die Messdatenerhebung durchgeführt wurden und unter welchen Bedingungen das System diese bearbeitet hat.

5.4.1 Testszenarien

Für die Erhebung der Messdaten wurde auf die in 5.2.3 beschriebenen Skripte zurückgegriffen. Es wurde ein Aufruf des Skripts *generate-traffic.sh* durchgeführt. Dieser sammelt im ausgeführten Verzeichnis rekursiv alle verfügbaren *.benchmark* Dateien und schickt diese der Reihe nach über das *curl-benchmark.sh* Skript an die Supplier-Schnittstelle. Die Skripte wurden dabei derartig organisiert, dass zu Beginn Skripte mit weniger Nachrichten in das System eingepflegt wurden, sodass es zu kleineren Skalierungssprüngen kam. Im späteren Verlauf wurde die Nachrichtenanzahl jedoch stetig erhöht, was wiederum in einem größeren Skalierungsaufkommen resultierte. Die folgende Abbildung (siehe 5.16 Benchmark - Queues) zeigt die Daten des Message Brokers.

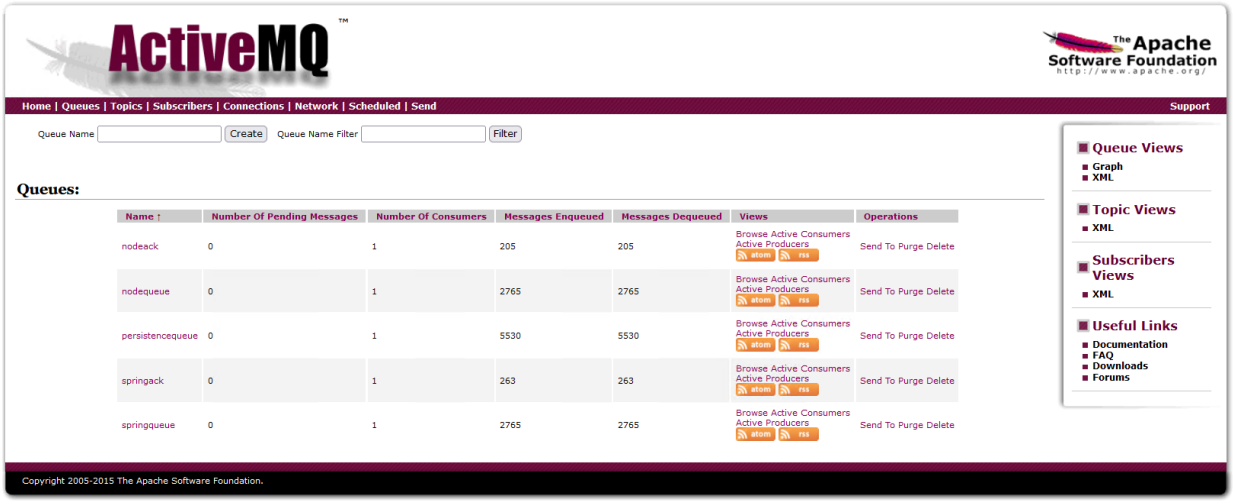


Abbildung 5.16: Benchmark - Queues

Es wurden für dieses Testszenario insgesamt 5530 Nachrichten verschickt und bearbeitet. Dies ist daran zu erkennen, dass die *persistencequeue* entsprechend viele Einträge verwaltet hat. Die Hälfte wurde jeweils von der Node.js- sowie Spring-Boot-Komponente bearbeitet (siehe *nodequeue* sowie *springqueue*). In der Praxis muss ein System allerdings eine deutlich höhere Last verarbeiten können. Das Produktivsystem des Unternehmens ist in der Lage 1000 Nachrichten pro Sekunde verarbeiten zu können. In entsprechenden Lasttests wird dies über einen Zeitraum von einer Stunde oder mehr getestet. Für diesen Prototypen ist die genannte Anzahl allerdings völlig ausreichend um einen Überblick über das System zu bekommen. Außerdem wurde in den verarbeitenden Komponenten des Prototypens eine künstliche Verlangsamung hinzugefügt um eine bessere Nachvollziehbarkeit zu gewährleisten, welche in der Produktivumgebung natürlich entfällt.

Der primäre Unterschied der beiden Services zeigt sich jedoch in der hochgefahrenen Containeranzahl. Um dem System mitzuteilen, dass ein neuer Container erfolgreich hochgefahren ist, setzt dieser eine Nachricht in die *nodeack* beziehungsweise die *springack* Warteschlange ab. Wie zu erkennen ist, wurden insgesamt 205 Node.js-Container und 263 Spring Boot Container über den gesamten Zeitraum des Testszenarios hochgefahren. Dieser Unterschied lässt sich mit der unterschiedlich langen Initialisierungsphase der Komponenten begründen (mehr dazu 5.2.3 Input).

Das zweite Testszenario betrifft die direkte Skalierung mithilfe des Skripts *direct-scaling.sh*. Hierbei wird die Proxy-Scaler Komponente direkt angesprochen (Erklärung siehe 5.2.3 Input). Der Aufruf wurde mit einem *Offset* Parameter von 29 und einer Wiederholungsanzahl von 10 getätigt. Insgesamt wurden für dieses Testszenario demnach 290 Skalierungsanfragen pro Service an die Proxy-Komponente geleitet. Beide Services wurden hierbei 4350 Mal neu hochgefahren[16].

5.4.2 Testbedingungen

Sämtliche erhobenen Daten werden auf einem System getestet, wie es in der folgenden Tabelle beschrieben wurde.

[16]Summe 1 bis 29 = 435, jeder Skalierungsschritt wurde 10 Mal wiederholt: 435 * 10 = 4350

Prozessor	Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz
Kerne	6 Prozessoren á 16 Kerne
RAM	16 GB
Storage	150 GB
Distro	CentOS 8

Tabelle 5.1: Server Specs

6 Ergebnisanalyse

Nachdem die Implementierung des Komponentenstacks im letzten Abschnitt detailliert beschrieben wurde, werden im Folgenden die erhaltenen Metriken erläutert. Es wird insbesondere auf die Bedeutung einzelner Messwerte sowie auf mögliche Begründungen dieser eingegangen. Die genauen Messwerte sind unter folgender Url einsehbar: `https://github.com/derMacon/serverless-bsc-thesis/tree/main/anhang/data`

Metrik	Dauer
Node.js	
Gesamtdurchschnitt	38416
Nachrichteneingang	35027
Verarbeitungsdauer	3388
Spring Boot	
Gesamtdurchschnitt	61800
Nachrichteneingang	58771
Verarbeitungsdauer	3029

Tabelle 6.1: Latenzzeiten

6.1 Ergebnisse

Für die Messung der folgenden Ergebnisse wurden in erster Linie zwei Lasttests durchgeführt. Der Erste bezog sich auf eine Skalierung als Reaktion auf Payment-Nachrichten, die vom System empfangen wurden. Die zweite Messung erfolgte zu direkt veranlassten Skalierungsschritten über eine gesonderte Schnittstelle vom Skalierer-Proxy-Service, die komplett auf eingehende Payment-Nachrichten verzichtet, dadurch allerdings auch das beschriebene Stufenmodell außer Kraft setzt (siehe Abschnitt 5.2.1 Schichtenmodell). Diese zweite Messung ermöglichte erst die genaue Übersicht über die Containeranzahl im Bezug zur Skalierungszeit (siehe Abbildung 6.3 Startzeit Container - Anzahl spezifisch).

6.1.1 Latenzzeit

Die wichtigsten Messdaten bezüglich der Leistungsfähigkeit des Systems beziehen sich auf den Datendurchsatz beziehungsweise die entsprechende Latenzzeit. In der vorherigen Tabelle wurden diese für die beiden betrachteten Backendtechnologien gegenübergestellt. Alle Angaben wurden in Millisekunden erfasst. Das System benötigt zur Bearbeitung einer eingegangenen Nachricht durch einen Node.js Konsumenten im Schnitt 38.4 Sekunden, während sich dieser Wert bei Spring Boot auf 61.8 Sekunden beläuft. Diese Werte bilden lediglich einen Durchschnitt aller erhaltenen

Metriken ab, eine genauere Aufteilung bezüglich des Zusammenhangs zu den Skalierungsgrößen erfolgt im nächsten Abschnitt (siehe Abschnitt 6.1.2 Aufschlüsselung nach Skalierungsstufe). Der Gesamtdurchschnitt setzt sich aus zwei Zeitangaben der Pipeline zusammen:

1. Der Nachrichteneingang: Diese Metrik beschreibt den Zeitraum zwischen erhaltener Anfrage im System und Acknowledgement durch die Konsumer-Komponente, dass die Nachricht nun bearbeitet wird. Sie beläuft sich bei der Node.js Komponente auf ungefähr 35.0 Sekunden und 58.8 Sekunden bei der Spring-Boot-Komponente.
2. Die Verarbeitungsdauer: Diese Metrik beschreibt den Zeitraum zwischen Acknowledgement durch den Konsumer, dass die Nachricht erhalten wurde, sowie dem Abspeichern des extrahierten Wertes durch den Konsumenten in der Datenbank. Dieser Wert beläuft sich bei beiden Implementierungen auf etwas über drei Sekunden. Bei dieser Metrik liegt der Fokus allerdings auf dem minimalen Zeitunterschied im Millisekundenbereich, da in beiden Implementierungen eine künstliche Verlangsamung (*Sleep-Funktion*) eingebaut wurde um die erhaltenen Messwerte besser nachvollziehen zu können. Bei Node.js liegt die tatsächliche Verarbeitungsdauer bei 388 Millisekunden *(total 3388)*, während sie bei Spring Boot bei 29 Millisekunden liegt *(total 3029)*.

6.1.2 Skalierungsdauer

Diese Gruppe von Metriken bezieht sich auf die genauen Zeiträume, die benötigt werden, um einen Container mit der entsprechenden Implementierung hochzufahren. Die ermittelten Zeiten stellen im Folgenden die durchschnittlichen Initialisierungszeiten eines einzelnen Containers dar. Sie beziehen sich nicht auf die Gesamtdauer der Initialisierungsphase aller beteiligten Container.

Aufschlüsselung nach Services Die abstrakteste Metrik bezieht sich auf den durchschnittlichen Zeitraum zum Initialisieren der Container eines bestimmten Services. Hierbei wird zwischen der Node.js sowie Spring-Boot-Komponente zum Konsumieren der Eingangsnachrichten unterschieden. Die Container, welche eine Node.js-Instanz beinhalten, benötigen im Schnitt 6.6 Sekunden zum Hochfahren, während diejenigen mit einer Spring-Instanz 35.3 Sekunden benötigen. Diese verhältnismäßig großen Zeiträume lassen sich unter anderem auf die Anzahl der hochgefahrenen Instanzen zurückführen. Eine genauere Aufschlüsselung dieses Prozesses erfolgt in den nächsten Abschnitten.

Aufschlüsselung nach Skalierungsstufe Um einen beispielhaften Skalierungsalgorithmus zu implementieren, wurde ein Regelsatz verfasst, der vom Alert-Manager zur Laufzeit automatisch in einem festgelegten Intervall ausgewertet wird. Um diesen Regelsatz zu Vorführungszwecken nicht unnötig ausführlich zu gestalten, wurde das Hinzufügen neuer Instanzen in Stufen organisiert.

Im durchgeführten Testszenario betrug die Grenze zum Überschreiten der ersten Stufe eine Anzahl von 15, für die zweite Stufe 30 und für die dritte Stufe 100 unbeantwortete Nachrichten in der jeweiligen Warteschlange. Wenn im Folgenden von einem „*Burst*“ gesprochen wird, ist hiermit das Überschreiten einer dieser Grenzen gemeint. Bei dem kleinstmöglichen Burst bezüglich der Warteschlange der Node.js-Komponente, nimmt das System im Schnitt 2.9 Sekunden zum Starten der Container in Anspruch um der Arbeitslast nach Skalierung mit insgesamt neuen 5 Containern-Instanzen zu begegnen. Bei der Spring-Boot-Komponente wurde hierfür ein Wert von 8.1 Sekunden gemessen. Im mittleren Grenzintervall wurde bezüglich Node.js eine Initialisierungsdauer von 3.8 Sekunden und seitens Spring 16.0 Sekunden festgestellt um nach der Skalierung mit 10 Instanzen zu arbeiten. Beim höchsten Burst kam es bei der Node.js-Komponente zu einer Initialisierungsphase von 6.9 Sekunden und bezüglich Spring zu 36.6 Sekunden. Hierbei sollen am Ende 30 Containerinstanzen

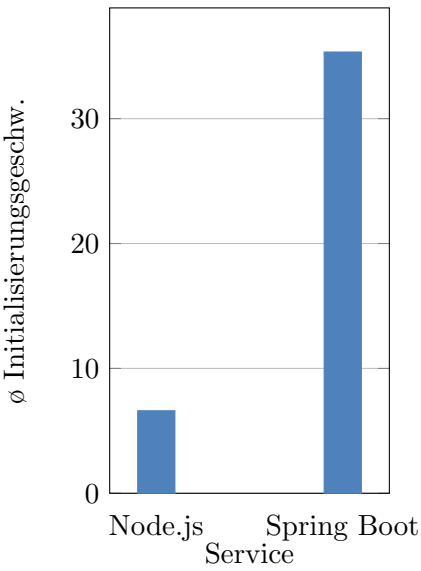


Abbildung 6.1: Serviceübersicht

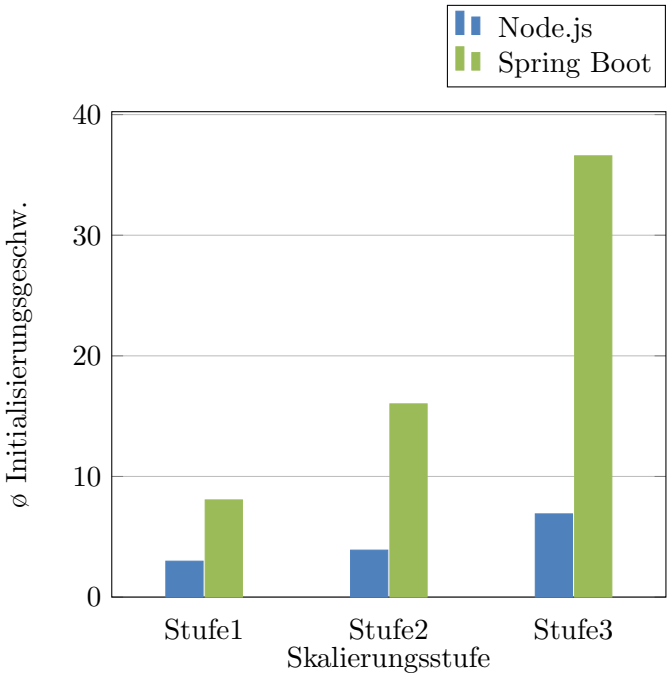


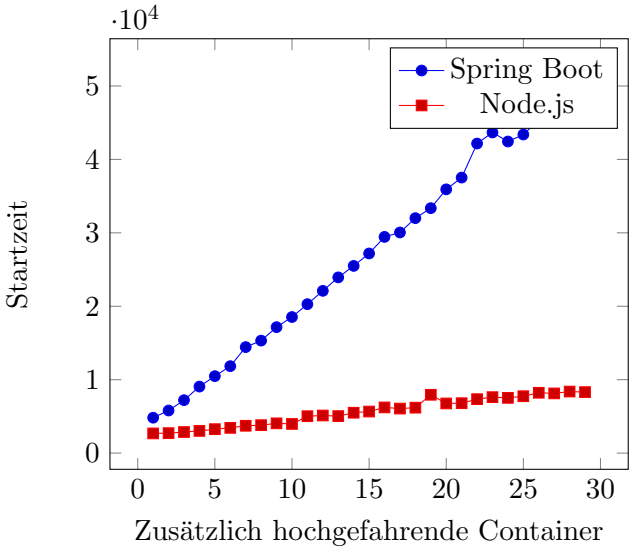
Abbildung 6.2: Stufenübersicht

des Konsumenten einsatzbereit sein. Da beim Skalierungsalgorithmus mit den beschriebenen Stufen gearbeitet wird, kann hierbei nicht genau gesagt werden, wie viele Container tatsächlich im Endeffekt initialisiert wurden, da stets mit den Differenzen zwischen den Stufen gearbeitet wird. Wenn zum Beispiel bereits n Container laufen und neue Nachrichten einen Burst verursachen, sollen nur die benötigten Instanzen kreiert werden, um genau die fehlende Anzahl zu decken. Die genaue Übersicht nach parallelen Instanziierungen wurde im folgenden Paragraphen genauer beschrieben (siehe Abschnitt 6.1.2 Aufschlüsselung nach Containeranzahl).

Aufschlüsselung nach Containeranzahl Die detaillierteste Metrik beschreibt die Zeiten aufgeschlüsselt nach Anzahl gleichzeitig hochfahrender Container (siehe 6.3 Startzeit Container - Anzahl spezifisch). Wie im Graphen erkennbar, handelt es sich um ein lineares Wachstum. Mit jedem zusätzlichen zeitgleich erstellten Container, dauert der Initialisierungsprozess bei der Spring-Implementierung im Schnitt 1611 Millisekunden länger. Bei der Node.js Implementierung liegt dieser Wert bei 194 Millisekunden. So dauert das Initialisieren eines einzelnen neuen Containers bei der Spring-Boot-Komponente 4,8 Sekunden, während sich dieser Wert bei der Node.js-Komponente lediglich auf 2,6 Sekunden beläuft. Um diese nach Containeranzahl aufgeschlüsselten Werte zu erhalten, wurde nicht wie bei den anderen Skalierungstests auf die öffentliche Schnittstelle mittels Nachrichtengenerierung zurückgegriffen, es wurde stattdessen eine interne Schnittstelle mittels dediziertem Skript verwendet um möglichst störungsfreien Zugriff auf die orchestrierende Komponente des Stacks zu erlangen. Die nachfolgenden Metriken wurden allerdings durch die öffentliche Schnittstelle generiert.

Zeitliche Aufschlüsselung Wie in Bild 5.10 auf Seite 40 zu erkennen ist, wurden diverse Panels zur zeitlichen Übersicht bereitgestellt. Diese wurden jedoch vor allem zu Kontrollzwecken implementiert. Diese visuelle Darstellung stellte sich während der Entwicklungszeit als sehr hilfreich dar, um Fehler frühzeitig zu erkennen und beheben zu können. Für die Auswertung der Ergebnisse haben diese allerdings wenig Relevanz. Der Vollständigkeit halber soll an dieser Stelle dennoch eine kurze Zusammenfassung der zeitbasierten Messwerte erfolgen. Für alle bisher behandelten Metriken gibt es Panels im Grafana Dashboard. Da die zeitliche Erfassung in Grafana selbst allerdings

Abbildung 6.3: Anzahlspezifisch



intervallbasierte Abfragen stellt, sind diese im Vergleich zu den Werten, welche in der Datenbank durch das System generiert und abgelegt wurden, sehr ungenau. Selbstverständlich wurde in der bisherigen Analyse ausschließlich auf die persistierten Daten zurückgegriffen. Bezüglich der Konsumenten wurde neben dem zusammengefassten durchschnittlichen Startverhalten auch der jeweils aktuelle Initialisierungszeitpunkt dargestellt. Hierbei war eine klare Korrelation zwischen der Anzahl der parallel startenden Konsumenten mit der erhöhten Initialisierungsdauer zu beobachten. Außerdem wurden diverse Metriken aus dem Activemq-Broker selbst ausgelesen. So ist es zum Beispiel möglich nachzuvollziehen, wie viele Nachrichten sich zu einem gegebenen Zeitpunkt nicht nur in einer bestimmten Warteschlange sondern auch innerhalb des ganzen Systems befinden. Außerdem wurde die zeitliche Abfolge zwischen einem Burst an unbeantworteten Nachrichten und dem Anstieg der Containerinstanzen in einem Dashboard-Panel dargestellt.

Zusammenfassung Es folgt eine Zusammenfassung des letzten Abschnitts.

- Zwei Lasttests
  - Skalierung als Reaktion auf Payment-Nachrichten
  - Skalierung als Reaktion auf direkte Anweisungen durch den Benutzer
- Latenzzeit als Durchschnitt über alle erhaltenen Werte
  - Node.js – 38.4 Sekunden
  - Spring Boot – 61.8 Sekunden
- Nachrichteneingang
  - Node.js – 35.0 Sekunden
  - Spring Boot – 58.8 Sekunden
- Verarbeitungsdauer
  - Node.js – 388 Millisekunden
  - Spring Boot – 29 Millisekunden

- Skalierungsdauer beschreibt Zeitraum zum Hochfahren eines Containers einer Technologie
  - Aufschlüsselung nach Services beschreibt die gesamtdurchschnittliche Dauer
  - Aufschlüsselung nach Skalierungsstufe beschreibt die durchschnittliche Dauer nach Größe des Bursts
  - Aufschlüsselung nach Containeranzahl beschreibt die durchschnittliche Dauer pro Containeranzahl
  - in allen Kategorien sind die Initialisierungszeiten der Node.js Technologie geringer als die der Spring Boot Technologie

6.2 Analyse

6.2.1 Latenzzeit

Hinsichtlich der Latenzzeit überrascht vor allem die deutliche Diskrepanz zwischen der Initialisierungsdauer der unterschiedlichen Technologien. Durch die Unterteilung der Pipeline in die zwei Messwerte ist erkennbar, dass diese auf die zeitliche Dauer bis zum Nachrichteneingang zurückzuführen ist. Hierbei läuft die Dauer der Initialisierungsphasen weit auseinander (siehe Abschnitt 6.2.2).

Bezüglich der Verarbeitungsgeschwindigkeit ist die Spring-Boot-Komponente jedoch im Vergleich schneller. Dies ist auf die Natur einer kompilierten Sprache zurückzuführen. Java Code wird im Vorwege in entsprechenden Bytecode übersetzt. Dieser Bytecode kann von der JVM interpretiert werden. Um die Performance zu steigern werden mithilfe des JIT-Compilers einzelne Methoden zur Laufzeit in Maschinencode umgewandelt, die direkt auf dem Prozessor ausgeführt werden können. Eine Skriptsprache wie Javascript stellt diese Performancevorteile nicht zur Verfügung wobei der Javascript-Code innerhalb der Node.js-Komponente zur Laufzeit direkt interpretiert wird. Da in den betrachteten Komponenten nur minimale Logik verbaut wurde, ist der festgestellte zeitliche Unterschied in der Bearbeitungsdauer mit 359 Millisekunden zwar nicht derartig gravierend, wie der Unterschied hinsichtlich der Initialisierungsphase der Container, allerdings lässt sich dadurch auch erkennen, dass die Spring Komponente durchschnittlich 13 mal schneller arbeitet als die Node.js Komponente. In Anbetracht der Komplexität der realen Banking-Anwendung gilt es, ein spezifisches Konzept für das Regelwerk zum Starten der Container zu entwerfen. Hierbei muss eine Untersuchung bezüglich der Rentabilität eines neuen startenden Containers gegenüber eines laufenden Containers evaluiert werden. Diese Untersuchung kann sich aus mehreren Aspekten zusammensetzen. Dabei könnte zum Beispiel betrachtet werden, wie viele Nachrichten eine bereits einsatzfähige Komponente verarbeiten kann, bevor eine neu initialisierte Komponente bereit ist Nachrichten entgegenzunehmen. Ebenfalls ist die Wahrscheinlichkeit, dass eine bereits laufende Komponente während der Bearbeitung abstürzt geringer, als die einer neu startenden Komponente (fehleranfälliger Verbindungsaufbau). Diese Wahrscheinlichkeit sowie der beschriebene Initialisierungsoverhead müssen bei der Ermittlung der idealen Threshholds zur Skalierung des Systems betrachtet werden.

nochmal
ob Teil r
Compile
rekt

6.2.2 Skalierungsdauer

Besonders auffällig ist, dass bei beiden Technologien das Initialisieren einzelner Container deutlich schneller abläuft, als wenn mehrere Container gleichzeitig hochfahren oder eine gewisse Anzahl an Container bereits laufen. Dies lässt sich auf die geteilten Ressourcen der Container zurückführen. In der Cloud wird hinsichtlich parallel arbeitender Container auch von „*noisy neighbors*“ gesprochen

[MK15, Seite 67 ff.]. Dieses Konzept beschreibt ein generelles Problem bezüglich Cloud-Systemen. Im Gegensatz zu traditionellen virtuellen Maschinen ist es bei Docker-Containern nicht möglich, den verfügbaren Speicher sowie CPU-Zeit direkt über den Container zu regulieren. Hierbei muss auf die *cgroup* Funktionalität des Linux-Kernels zurückgegriffen werden. „*Control groups, usually referred to as cgroups, are a Linux kernel feature which allow processes to be organized into hierarchical groups whose usage of various types of resources can then be limited and monitored*“ [man]. Dies wurde beim Prototypen nicht konfiguriert. Insbesondere die Initialisierungsphase von neuen Containern, lässt sich hierdurch optimieren. Im folgenden Abschnitt folgt eine kurze Diskussion über die verfügbaren Möglichkeiten diesbezüglich (siehe Abschnitt 6.3.2).

Allerdings gibt es nicht nur Abweichungen hinsichtlich der Geschwindigkeit bei unterschiedlicher Containeranzahl, auch die verwendete Technologie ist von Bedeutung. Die erheblich abweichende Initialisierungsgeschwindigkeit der beiden unterschiedlichen Komponenten lässt sich im Kern auf die Initialisierung des Spring-Containers sowie des dazugehörigen Application-Contexts zurückführen. „*Eines der Schlüsselelemente des Spring-Frameworks ist die Bereitstellung von Infrastruktur auf Anwendungsebene*“ [Sim18, Seite 53 ff.]. Hiermit sind in erster Linie die Mechanismen der „*dependency injection*“ sowie der „*aspektorientierten Programmierung*“ gemeint. Diese lose Kopplung erlaubt die Erstellung von hochspezialisierten Komponenten *(hohe Kohäsion)* und ermöglicht einen deutlich übersichtlicheren Quellcode. Kern des Spring Frameworks ist der *Spring Container*. Dieser verwaltet fachliche und nichtfachliche Objekte, die eine Anwendung ausmachen. Die verwalteten Objekte werden als *Beans* bezeichnet. „*Eine Bean ist ein Objekt, das vom Spring-Container instanziiert und konfiguriert wurde und dessen Lebenszyklus vom Container verwaltet wird. Die Abhängigkeiten zwischen Beans sind als Metadaten im Container verfügbar*“ [Sim18, Kapitel 3.1.1]. Durch die Verwendung des Spring-Containers können sich die Entwickler mehr auf die Implementierung der Businesslogik und weniger auf die Konfiguration des Systems konzentrieren. Das Node.js-Framework besitzt diese Mechanismen nicht und der vom Entwickler geschriebene Code wird zur Laufzeit direkt ausgeführt. Dies erschwert zwar den späteren Verlauf eines Projektes, birgt allerdings keinen Konfigurationsoverhead. Spring bietet über die gegebene Grundfunktionalität mit jeder importierten Dependency weitere automatische Konfigurationsschritte. Es gibt zwar Möglichkeiten diese zusätzliche Konfiguration zu gewissen Teilen außer Kraft zu setzen, dennoch ändert dies nichts an dem grundlegenden „*Problem*“ der längeren Initialisierungsphase (siehe Abschnitt 6.3.1). Um noch einmal zu betonen, wie viele Schritte zur Initialisierung einer Spring-Bean durchlaufen werden müssen, folgt ein kurzer Exkurs.

Exkurs: Initialisierung Spring Bean

Im folgenden Schaubild wurden die wesentlichen Schritte des Lebenszyklusses einer Spring-Bean exemplarisch dargestellt. Das kontrollierte Herunterfahren *destruction* wird hierbei nicht weiter betrachtet.

Während der *Initialisierungsphase* wird die genutzte Bean-Instanz vom Framework erstellt. Dies kann mittels eines direkten Konstruktoraufrufs oder sonstigen Mustern (Factory, Builder etc.) geschehen. Anschließend werden die erstellten Instanzen einem sogenannten „*Componentscan*“ unterzogen. Hierbei werden die vom Spring-Container verwalteten Beans mit benötigten Referenzen abhängiger Beans gefüllt. Dieser Mechanismus bildet die Basis der sogenannten dependency injection. Hierzu gibt es mehrere Möglichkeiten.

1. Es wird geschaut, welche der Instanzen ein Interface vom Typ *Aware* implementieren. Bei dem Supertypen handelt es sich um ein reines Markerinterface. Jedoch gibt es verschiedene Subtypen, die Setter-Methoden für einzelne kontextabhängige Ressourcen bieten. Über diese Methoden wird dem Framework beispielsweise signalisiert, dass eine Bean, welche das Interface

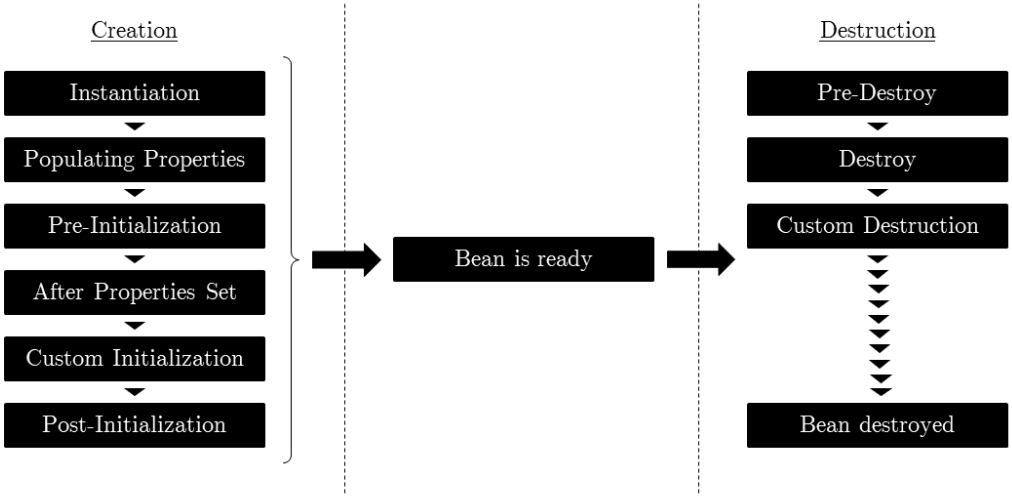


Abbildung 6.4: Spring Bean Lifecycle [beab]

ApplicationContextAware implementiert, eine Instanz vom Typ *ApplicationContext* injiziert bekommen soll. Ähnliches gilt zum Beispiel für Beans, die Zugriff auf Umgebungsvariablen benötigen (siehe Listing 6.1). Über diese Interfaces ist es außerdem möglich direkt in den Initialisierungsprozess einer Bean einzugreifen, indem eine eigene Aware Implementierung erstellt wird.

2. Ab der Spring Version 4.3 und höher ist es möglich einen Konstruktor zu erzeugen, der lediglich die zu injizierenden Felder setzen muss.

3. Außerdem ist es möglich mittels der *Autowired*-Annotation Instanzen zu setzen.

```
1  @Component
2  public class LoggerService implements EnvironmentAware {
3
4          private Environment environment;
5
6          @Override
7          public void setEnvironment(Environment environment) {
8                  this.environment = environment;
9          }
10 }
```

Listing 6.1: Bean - EnvironmentAware [beaa]

Im nächsten Schritt folgt der Verarbeitungsschritt der *Pre-Initialization*. Hierbei werden diverse Arbeitsschritte durchlaufen, bevor die eigentliche Initialisierung stattfindet. Hierzu gehören zum Beispiel auch Methoden mit der *PostContstructor*-Annotation, die auf Felder der Klasse zugreifen können, ohne im Konstruktor vertreten sein zu müssen. Es folgen noch weitere Schritte, auf die hierbei nicht weiter eingegangen wird.

Diese Beschreibung sollte lediglich einen kurzen Einblick über die Möglichkeiten geben, welche durch den Initialisierungsprozess dem Anwender zur Verfügung gestellt werden. Hierbei ist es nicht nur möglich, genau nachzuvollziehen, in welcher Phase sich eine Bean momentan befindet. Der Prozess bietet ebenfalls die Möglichkeit, mithilfe spezifizierter Schnittstellen an einer beliebigen Stelle eigene Logikbausteine zu integrieren. Um der Philosophie des Spring-Frameworks gerecht zu werden, braucht es auch einen solch komplexen Aufbau.

6.2.3 Zusammenfassung

Es folgt eine kurze Zusammenfassung des letzten Abschnitts.

- Noisy Neighbors: Skalieren in der Cloud bedeutet Kampf um Ressourcen
  - cgroups als wesentlicher Steuerungsmechanismus
  - Erörterung der Konfigurationsmöglichkeiten im nächsten Abschnitt
- Skalierungsdauer: erhöhte Initialisierungsdauer von Spring auf zwei wesentliche Faktoren zurückzuführen
  - Spring Initialisierungsaufwand (IoC, Spring Context, AOP)
  - Ressourcenbegrenzung in der Cloud (CPU Shares)
- Exkurs: Spring Initialisierungsschritte
  - Möglichkeiten, eigene Logikbausteine unterzubringen
  - Verarbeitungsschritte schaffen Nachvollziehbarkeit

6.3 Diskussion

Im letzten Abschnitt wurde auf die zwei wesentlichen Kenndaten der ermittelten Metriken eingegangen. Dabei wurden mögliche Begründungen für die erhaltenen Messdaten präsentiert. Es folgte außerdem ein kurzer Exkurs zum Initialisierungsprozess einer Spring-Bean. Generell stellt die lange Initialisierungsphase der Spring-Komponenten ein ernsthaftes Problem dar. Welche Schritte zur Optimierung dieses Verhaltens durch die zugrunde liegende Technologie ermöglicht werden und inwiefern das System dies mithilfe der schnellen Verarbeitungsdauer auszugleichen vermag, wird im folgenden Abschnitt beschrieben.

6.3.1 Spring-Bean - Optimierung der Initialisierungsphase

Die deutlich längere Initialisierungsphase der Spring-Komponente lässt sich in erster Linie auf die beschriebene Komplexität der einzelnen Anwendungsschritte zurückführen. Um einem derart mächtigem Framework wie Spring Boot gerecht zu werden, benötigt es einen ähnlich ausgeprägten Initialisierungsmechanismus für den Spring-Container und allen damit verbundenen Anforderungen. Da das Projekt allerdings verschiedene Abhängigkeiten aufweist, wird auch der Initialisierungsprozess entsprechend erweitert. Mit dem Befehl `mvn spring-boot:run -Ddebug` können zum Beispiel alle automatischen Konfigurationsschritte ausgegeben werden. Wie im Artikel [spr] beschrieben, ist es durchaus möglich, einige der Konfigurationen wegzulassen, wenn die dazugehörigen Komponenten nicht genutzt werden.

Es ist außerdem denkbar, auf einen anderen Servlet-Container[1] wie „*Undertow*“ zuzugreifen. Dieser stellt eine leichtgewichtigere Implementierung eines Webservers bereit und ermöglicht dadurch eine

[1] Definition Servlet Container [ser]

Performancesteigerung. Dennoch wurde sich für den Prototypen dagegen entschieden, weder einzelne Konfigurationen manuell zu entfernen, noch die Container-Implementierung zu wechseln, da sich das Projekt auch nach Bearbeitung der Thesis eventuell weiterentwickelt und als Beispielprojekt gelten könnte. Hierbei gilt es darauf zu achten auch zukünftig vermeidbare Fehlerfälle zu vermeiden.

6.3.2 Ressourcenoptimierung

Docker bietet nativ die Möglichkeit sowohl Speicher als auch CPU-Ressourcen für spezifische Container oder den gesamten Stack zu regulieren. Dadurch könnten unter anderem mehr Ressourcen für das Hochfahren neuer Container zur Verfügung gestellt werden.

Bezüglich des Aufteilens der zugewiesenen CPU-Zeit modelliert Docker intern einen Pool mit 1024 Shares. Es ist nun möglich, über die beschriebenen Konfigurationsmechanismen für einen Container anzugeben, wie viele Shares dieser vom Scheduler zugewiesen bekommen soll. Falls ein Containern beispielsweise lediglich die Hälfte der verfügbaren Ressourcen verwenden soll, wird ein Wert von 512 angegeben, wenn er allerdings so viel CPU-Zeit wie möglich in Anspruch nehmen soll, wird hierbei ein Wert von 1024 angegeben. Diese Anteile repräsentieren lediglich den Bedarf eines Containers. Ein Container mit 1024 angegebenen Shares besetzt die CPU nicht ausschließlich, sondern signalisiert nur, dass dem Container so viel Zeit wie möglich zugewiesen werden soll. Außerdem ist es möglich mithilfe des sogenannten „*CPU pinning*“ einzelne CPU-Kerne bestimmten Containern zuzuweisen. So ist es beispielsweise realisierbar einzelne Kerne primär zum Hochfahren neuer Komponenten zu verwenden.

Neben der Anpassung der Rechenleistung ist es ebenfalls möglich den verfügbaren Speicher eines Containers zu modifizieren. „*While constraining the CPU only impacts the application's priority for CPU time, the memory limit is a hard limit*“ [MK15, Seite 68 ff.]. Bei einem unausgelasteten System würde auch eine niedrige Anzahl an CPU-Shares in einem größeren Anteil von Rechenleistung resultieren, bei der Begrenzung des Speichers gibt es diese Option nicht. Eine feingranularere Anpassung der System-Ressourcen ist außerdem mit den sogenannten „*user limits (ulimits)*“ möglich.

6.3.3 Optimierte Ausführungsreihenfolge

Um die gesamte Startzeit bei gleichzeitigem Starten von Node.js und Spring Containern minimieren zu können, wäre es ebenfalls angebracht eine Strategie hinsichtlich der Reihenfolge der Services zu entwickeln. In der Analyse der Daten war zu erkennen, dass ein Spring-Container im Schnitt 13 Mal schneller arbeitet als ein Node-Container. Um diesen Unterschied auszugleichen sollten entsprechend mehr Node Container gestartet werden um hierbei einen Ausgleich zu schaffen. Bereits während des Hochfahrprozesses der einzelnen Container sollte ein ähnliches Verhältnis eingehalten werden. Bisher findet hier eine strikte Trennung der Services statt, es wurden also entweder n Spring-Container und / oder n Node-Container gestartet, eine Vermischung der Anfragen würde die Effizienz des Systems hinsichtlich des Datendurchsatzes sowie der Latenzzeit steigern. Dies in Kombination mit den vorher vorgestellten Stellschrauben zur Ressourcenoptimierung bietet eine solide Basis für das Optimierungspotenzial der Initialisierungs- sowie Verarbeitungsphase der Anwendung.

6.3.4 Aussagen über Produktivumgebung

Die Aussagen, die während der Analyse der Daten getroffen wurden, sind hinsichtlich eines Systems in einer Produktivumgebung nur in Teilen aussagekräftig. Ein wesentliches Hauptkriterium der ISO-Norm 25010 wurde hierbei vernachlässigt, denn es wurde nur sehr eingeschränkt auf die Verfügbarkeit des Systems und dem *Design for failure* eingegangen. Wenn zentrale Komponenten ausfallen, gibt es

lediglich ein definiertes Default-Verhalten des Orchestrators. So wird versucht, die Komponente selbst sowie sämtliche Kommunikationspartner neu zu starten. Dieses Verhalten hat vor allem Einfluss auf den Initialisierungszeitraum eines Containers. Da beide betrachteten Technologien auf eigene Weise Mechanismen zum erneuten Verbindungsaufbau im Falle eines Ausfalls zur Verfügung stellen, ist diese Betrachtung für die reine Evaluierung der Technologie von nicht allzu großer Bedeutung. Im Zweifelsfall wird ein Container mehrmals gestartet, die grundlegenden Messdaten lassen sich bereits an einem störungsfreien Ablauf ablesen. Falls es zu einem späteren Zeitpunkt während der Portierung der Banking-Anwendung hin zu einer der betrachteten Technologien kommen sollte, muss dieses Fehlerverhalten jedoch ausgiebig getestet werden um späteres unerwartetes Verhalten zu vermeiden.

Damit das Verhalten des Systems bezüglich eines Ausfalls bestimmter Komponenten betrachtet werden kann, stehen unterschiedliche Werkzeuge zur Verfügung. Ein etabliertes Werkzeug stellt das open source Projekt „*Chaos Monkey*[2]“ dar, das von Netflix entwickelt und veröffentlicht wurde. Es ist allerdings auch möglich Werkzeuge für spezifische Orchestrator zu nutzen (Übersicht [BA18, Seite 369]). Diese Art von Werkzeugen ermöglicht das kontrollierte Aussetzen bestimmter Netzwerkverbindungen sowie das Herunterfahren ganzer Services über einen festgelegten Zeitraum.

Die Aussage, dass die Spring-Komponente durchschnittliche 13 Mal schneller arbeitet als das Node.js-Pendant ist für den beschriebenen Anwendungsfall zwar korrekt, allerdings wurden hierbei lediglich die erläuterten Schritte abgearbeitet (siehe Abschnitt 4.2.1 Festlegung eines fiktiven Workflows). Auf das Laufzeitverhalten einer komplexeren Anwendung kann hierbei nicht geschlossen werden. Eventuell lässt sich dieser Unterschied lediglich auf einen Verbindungsaufbau oder Ähnliches zurückführen, wobei die restliche Logik hiervon nicht betroffen ist. Dies lässt sich mithilfe weiterer Timestamps innerhalb der Komponenten ermitteln.

6.3.5 Beurteilung der Technologien hinsichtlich des beschriebenen Usecases

Der Fokus der Thesis liegt auf dem Start-up-Verhalten der Komponenten. Hierbei ist festzustellen, dass Node.js einen deutlichen Vorteil gegenüber dem Spring-Boot-Framework vorzuweisen hat. Im Banking-Bereich ist dies unabdingbar. Die Performanz lässt sich allerdings zum gegenwärtigen Zeitpunkt nur eingeschränkt beurteilen. Mit den erhaltenen Daten zeichnet sich hierbei im Gesamtbild für den Anwendungsfall des Unternehmens jedoch eine Empfehlung für die Verwendung neuwertiger Skript-Technologien aus.

[2] `https://github.com/Netflix/chaosmonkey`

6.3.6 Zusammenfassung

Es folgt eine kurze Zusammenfassung des letzten Abschnitts.

- lange Startzeit der Spring Komponente auf Initialisierungsphase zurückzuführen
  - Möglichkeit überflüssige Konfiguration ungenutzter Komponenten zu entfernen
  - Möglichkeit auf anderen Servlet-Container umzusteigen
- Ressourcenoptimierung um Ressourcen während Initialisierungsprozess zur Verfügung zu stellen
  - CPU-Shares
  - Speicher
  - User Limits
- Optimierte Ausführungsreihenfolge während Initialisierungsphase
  - Vermischung beider Service-Initialisierungsphasen
  - Node.js proportional zur Verarbeitungsgeschwindigkeit skalieren
- Finale Beurteilung: beschriebene Vorteile von Skript-Technologien überwiegen

7 Zusammenfassung

Nachdem im letzten Abschnitt sämtliche Messwerte vorgestellt, analysiert und beurteilt wurden, folgt nun eine Zusammenfassung der gesamten Thesis.

Die Idee und die Umsetzung der Arbeit erfolgte in Zusammenarbeit mit der DPS Engineering Gmbh. Hierbei handelt es sich um ein europaweit tätiges Software- und Consultingunternehmen für die Finanz- und Retailbranche. Das Unternehmen möchte in Zukunft auf moderne Technologien setzen, um ihre Plattform für die Echtzeitüberweisungen performanter und ressourcensparender zu gestalten. Da das Unternehmen im Banking Bereich tätig ist, stellt außerdem eine hohe Verfügbarkeit der Komponenten eines der wesentlichen Kernziele dar. Auf die neuen Technologien bezogen bedeutet dies, dass es zu ermitteln gilt, wie viel Zeit ein Skalierungsprozess in einem orchestrierten Komponentenstack im Detail braucht. Um festzustellen, wie gut die jeweils betrachtete Backend-Technologie im Endeffekt für den Betrieb in einer containerbasierten Anwendung tatsächlich geeignet ist, wurde ein fiktiver Workflow mithilfe eines Komponentenstacks implementiert, der in der Lage ist, verschiedene Lastszenarien abarbeiten zu können. Hierbei wurden im späteren Verlauf austauschbare Komponenten implementiert, um eine Gegenüberstellung anhand erhaltener Messwerte zu ermöglichen.

Es folgte ein kurzer Einblick in die aktuelle Produktivumgebung und die damit verbundene monolithische Struktur. Es wurde ebenfalls auf die Probleme dieser Architektur eingegangen, welche unter anderem als Motivation der Thesis betrachtet werden können.

Nachdem die Kernziele im Detail erläutert wurden, erfolgte eine Festlegung sämtlicher Anforderungen für das Projekt. Hierbei wurden diese für die verschiedenen Teile des Projekts gesondert festgelegt. Hinsichtlich der Daten zur Messung des Startup-Verhaltens von Containern wurde auf die ISO Norm 25010 zurückgegriffen. Diese stellt einen Kriterienkatalog zu qualitativen Anforderungen an Softwareprodukte dar. Sie beinhaltet unter anderem eine Beschreibung der *Leistungsfähigkeit*, *Skalierbarkeit*, *Kompatibilität* und *Zuverlässigkeit* eines Produkts. Die Festlegung der zu betrachtenden Anforderungen erfolgte im nachfolgenden Abschnitt. Anschließend wurden die Anforderungen an den Prototypen formuliert, hierbei wurde insbesondere ein fiktiver Workflow festgelegt. Dieser orientiert sich an der Ablauflogik des Softwareprodukts innerhalb der Produktivumgebung. Primär erfolgt eine Überprüfung, ob es sich bei dem Inhalt einer eingehenden Nachricht um valides XML handelt (XSD-Konformität), anschließend wird ein Feld aus dem XML ausgelesen und in einer Datenbank hinterlegt. Danach folgten die Anforderungen Containerplattform sowie die Orchestrierungsplattform, wobei festgelegt wurde, dass in erster Linie die Portabilität und die Reproduzierbarkeit für das Arbeiten mit Containern von Bedeutung sind, während die automatische Wartung sowie die Skalierung für die Orchestrierungsplattformen am wichtigsten sind. Bezüglich der Anforderungen des Lasttests wurde auf eine benötigte Benutzerschnittstelle sowie die Hinterlegung der Daten in einer Persistenzschicht eingegangen. Die Anforderungen für das Monitoring und der Darstellung der Daten bestehen aus dem Visualisieren der zeitlichen Zusammenhänge und der Persistenz der Daten in einer internen Datenbank.

Nachdem im letzten Abschnitt die möglichen Qualitätsmerkmale erläutert wurden, erfolgte nun eine Festlegung auf die zu betrachtenden Merkmale. Hierbei wurden die *Leistungsfähigkeit* sowie die

Skalierbarkeit gewählt. Es wurde außerdem darauf eingegangen, auf welche Weise diese zu testen sind. Hierzu wurde die Pipeline in mehrere Etappen unterteilt, zu denen Timestamps erzeugt werden sollen, um möglichst aussagekräftige Ergebnisse zu erlangen.

Im Anschluss hieran erfolgte die Beschreibung der Implementierung des Prototypen. Es wurde auf das zugrunde liegende Schichtenmodell, die Methodik des Deployments sowie die Ablauflogik auf Komponentenbasis eingegangen.

Am Ende der Thesis folgte eine Ergebnisanalyse. Hierbei wurden die erhaltenen Messwerte im Detail erläutert. Hierbei wurden zwei Lasttests durchgeführt. Der Erste gibt Payment-Nachrichten in das System. Hierbei muss das System selbstständig in der Lage sein eine entsprechende Skalierung vorzunehmen. Im zweiten Test wird die Skalierung als Reaktion auf direkte Anweisungen des Benutzers vorgenommen. Die Ergebnisse hierbei zeigen, dass es hinsichtlich der Initialisierungsdauer der Komponenten mit Node.js als implementierende Technologie einen deutlichen Vorteil gegenüber der Spring-Boot-Technologie zu verzeichnen gibt. Die Latenzzeit im gesamtdurchschnittlichen Vergleich zeigt bei der Node.js-Technologie beispielsweise einen Wert von 38.4 Sekunden, während dieser Wert bei der Spring Boot Technologie bei 61.8 Sekunden liegt. Auch bei einer feingranulareren Untersuchung hinsichtlich der Burstgröße oder der einzelnen Containeranzahlen ergibt sich ein ähnliches Bild. Dennoch gilt es noch hervorzuheben, dass die Spring Boot Technologie einen deutlichen Performancevorteil hinsichtlich der reinen Verarbeitungszeit der Nachrichten aufweist. Die Node.js-Komponente benötigt im Schnitt 388 Millisekunden um eine Nachricht zu verarbeiten, während dieser Wert bei Spring Boot bei 29 Millisekunden liegt.

Im darauf folgenden Abschnitt wurden diese Werte noch einmal näher analysiert und es wurden mögliche Begründungen für das gemessene Verhalten aufgezeigt. Die lange Startzeit der Spring-Boot-Komponente lässt sich beispielsweise unter anderem auf die komplexe Konfiguration des Frameworks und der damit verbunden Bean-Instanzen zurückführen. Die performantere Verarbeitungsgeschwindigkeit lässt sich dagegen vor allem dadurch begründen, dass es sich bei der verwendeten Programmiersprache des Spring-Boot-Frameworks um eine kompilierte Sprache handelt.

8 Ausblick

Nachdem im letzten Abschnitt eine Zusammenfassung der gesamten Thesis beschrieben wurde, folgt ein Ausblick für den Fortbestand dieses Projekt und die hieraus erlangten Erkenntnisse.

Wie bereits in der Diskussion (siehe Abschnitt 6.3 Diskussion) beschrieben, lassen sich die erlangten Ergebnisse nicht zwangsweise auf eine Produktivumgebung übertragen. Es wurde hierbei in erster Linie ermittelt, wie sich die Start-up-Zeiten in einem störungsfreien System verhalten. In einem nächsten Schritt gilt es mögliche Konfigurationen in einem störungsbehafteten System zu testen. Außerdem wäre es denkbar, einen noch effizienteren Start-up-Prozess zu gestalten und somit die Initialisierungsdauer weiter zu verkürzen. Dies kann über die Docker-Runtime selbst, die Ausführungsreihenfolge oder weitere Konfigurationsmechanismen in den zugrunde liegenden Technologien selbst geschehen. Auch können weitere Komponenten wie zum Beispiel der verwendete Broker oder die Datenbanken skaliert werden, was ebenfalls einen deutlichen Performanceanstieg mit sich führen würde.

Außerdem gilt es weitere vorgestellte Aspekte der ISO-Norm zu untersuchen. Beispielsweise sollte ein solches System in der Lage sein, trotz eines möglichen Ausfalls einiger Komponenten weiterhin in einem stabilen Zustand zu agieren. Dies kann über verschiedene Werkzeuge getestet werden und stellt einen wesentlichen Faktor für den weiteren Auswahlprozess dar.

Diese Thesis soll als ein erster Schritt zur Evaluierung verschiedener cloudfähiger Technologien betrachtet werden. Je nach Usecase der Anwendung müssen im entsprechenden Bereich weitere Systemkomponenten und spezifischere Tests entworfen werden. In dieser Ausarbeitung erfolgte lediglich ein erster Einblick in eine mögliche Konfiguration eines Komponentenstacks, den es in Zukunft für das jeweilige Anwendungsgebiet zu erweitern und zu verfeinern gilt.

Literaturverzeichnis

[BA18] Daniel Bryant and Marin-Perez Abraham. *Continuous delivery in Java essential tools and best practices for deploying code to production.* O'Reilly, 2018.

[beaa] Aware Interface - Zusammenfassung. `http://www.javabyexamples.com/quick-guide-to-spring-aware-interfaces`. Aufgerufen am: 08-07-2021.

[beab] Spring Bean Lifecycle. `https://reflectoring.io/spring-bean-lifecycle/`. Aufgerufen am: 08-07-2021.

[Bra18] Brian Brazil. *Prometheus: Up and Running: Infrastructure and Application Performance Monitoring.* O'Reilly Media, 2018.

[def] Definition Native Cloud-Anwendungen. `https://www.computerweekly.com/de/definition/Native-Cloud-Anwendung`. Aufgerufen am: 05-07-2021.

[doc] Docker Dokumentation. `https://docs.docker.com/`. Aufgerufen am: 06-07-2021.

[Eid05] Petter L. H. Eide. Quantification and traceability of requirements. `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.6464&rep=rep1&type=pdf`, 2005.

[Fan18] Thomas J. Fan. Docker scaler - github pages. `https://thomasjpfan.github.io/docker-scaler/`, 2018.

[ins] Cloud Native - Definition. `https://www.dertreasurer.de/themen/instant-payments-revolution-im-zahlungsverkehr/`. Aufgerufen am: 26-06-2021.

[iso] ISO 2050 - Schaubild. `https://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/is-management/Systementwicklung/Management-der-Systementwicklung/Software-Qualitatsmanagement/Qualitatsmerkmale-von-Software/index.html`. Aufgerufen am: 27-07-2021.

[iso11] Iso/iec 25010:2011. `https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en`, 2011.

[man] Manpages Linux - cgroups. `https://man7.org/linux/man-pages/man7/cgroups.7.html`. Aufgerufen am: 08-07-2021.

[mic] microprofile Spezifikation. `https://access.redhat.com/documentation/en-us/jboss_enterprise_application_platform_continuous_delivery/15/html/configuration_guide/eclipse_microprofile`. Aufgerufen am: 25-06-2021.

[MK15] Karl Matthias and Sean P. Kane. *Docker: Up and Running - Shipping Reliable Containers in Production.* „O'Reilly Media, Inc.“, Sebastopol, 2015.

[nfr] non functional requirements - Evaluierung. `https://www.dotnetcurry.com/project-management/1462/non-functional-requirements-nfrs`. Aufgerufen am: 26-06-2021.

[ore] Layered Architecture - Definition. `https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html`. Aufgerufen am: 26-06-2021.

[SBD11] Bruce Snyder, Dejan Bosanac, and Rob Davies. *ActiveMQ in Action* -. Manning Publications, Birmingham, 2011.

[ser] Servlet Container - Definition. `https://dzone.com/articles/what-servlet-container`. Aufgerufen am: 29-07-2021.

[Sim18] Michael Simons. *Spring Boot 2 - Moderne Softwareentwicklung mit Spring 5*. dpunkt.verlag, Heidelberg, 2018.

[Sop16] Fabrizio Soppelsa. *Native docker clustering with swarm*. Packt Publishing Limited, 2016.

[spr] Spring Boot - Performantere Konfiguration. `https://www.alexecollins.com/spring-boot-performance/`. Aufgerufen am: 08-07-2021.

[Tur14] J. Turnbull. *The Docker Book: Containerization Is the New Virtualization*. James Turnbull, 2014.

[typa] Typescript - Anforderungen an OOP. `https://rachelappel.com/2015/01/02/write-object-oriented-javascript-with-typescript/`. Aufgerufen am: 12-07-2021.

[typb] Typescript - Dokumentation. `https://www.typescriptlang.org/docs/handbook/intro.html`. Aufgerufen am: 12-07-2021.

9

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Ort, Datum Silas Hoffmann