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

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 geben und diverse Eigenarten näher 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 altbewerten 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 wie bisher 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 wäre hier die Abwicklung von Zahlungsverkehr mit Hilfe von Echtzeitüberweisungen[2], hierbei gilt es hinsichtlich schwankenden Zahlungsverkehrs über einen gewissen Zeitraum das laufende System dem 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 Microsof (Azure) gesetzt werden. Die behandelten Themen dieser Thesis beziehen sich vorallem 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 Operation- und Entwicklungsabteilung. Projekte erleben weniger Handoffs zwischen Teams, ein Entwickler betreut ein Projekt bis in die Deploymentphase, was vorher in dem Maße nicht möglich gewesen ist. 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 Containern untersuchen. Der Prototyp beinhaltet mehrere 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 neuere 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.

2.0.1. Begriffserklärung ✓✓

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 bespielsweise 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 zusammensetzen. 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 ebenfalls die betreffenden System-Komponenten.

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 Verwalten von benötigten Instanzen ermöglicht, hierbei werden also Instanzen je nach Bedarf hoch oder heruntergefahren [BA18, Seite 70 ff.].

2.0.2. 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 vorallem durch ihre minimale Startzeit für diesen Usecase qualifizieren.

Zusammengefassung:

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

2.1. 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 bestehende 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 Produktion 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). 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 kommt. Wenn eine

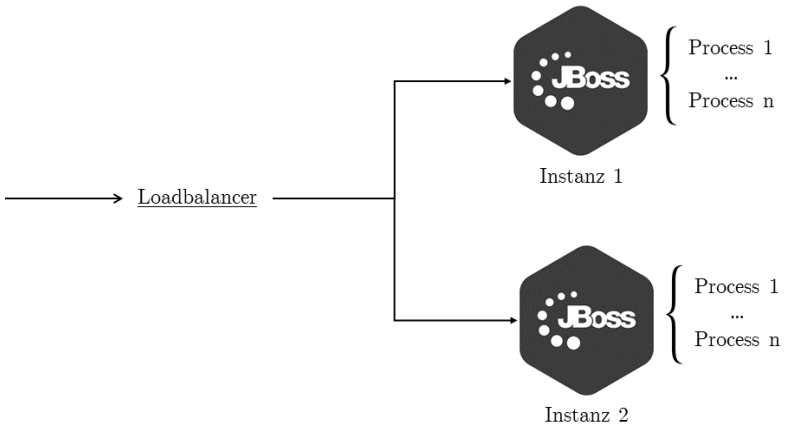


Abbildung 3.1.: JBoss Systemaufbau

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.

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-basiert* 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 eigentlich 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 eventuell 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 in Produktion ✓✓

Eine Skalierung ist mit monolithischen Strukturen nur dadurch möglich, den ausführbare Code auf zusätzlichen Servern zu deployen, dies wird auch *horizontale Skalierung genannt.* Jede dieser Kopien nutzt die gleiche Resourcenanzahl, was es zu einem ineffizienten Design macht, da sie sich nicht dynamisch der gegebenen Last anpassen. Tatsächlich ist ein es 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 Resourcen 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 des Problemfokuses 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 (kurz *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], fig. 4.1).

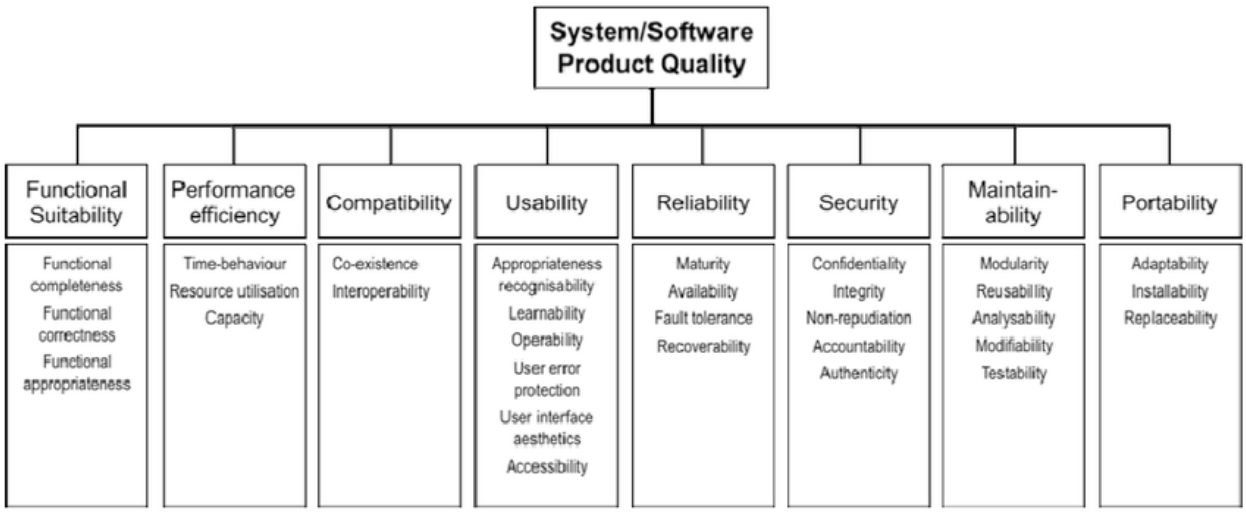


Abbildung 4.1.: ISO 25010

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* 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 4.2). Je nach Teilbereich lassen sich hierbei einzelne Gebiete automatisiert oder mittels anderweitiger Werkzeuge testen. Diese Tests sind in einem System in Produktion 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, deshalt werden die Kriterien unter anderem auch *system-quality attributes* genannt [BA18, Seite 335]. Im Folgenden werden die für das Projekt relevanten Kriterien zusammengefasst dargestellt.

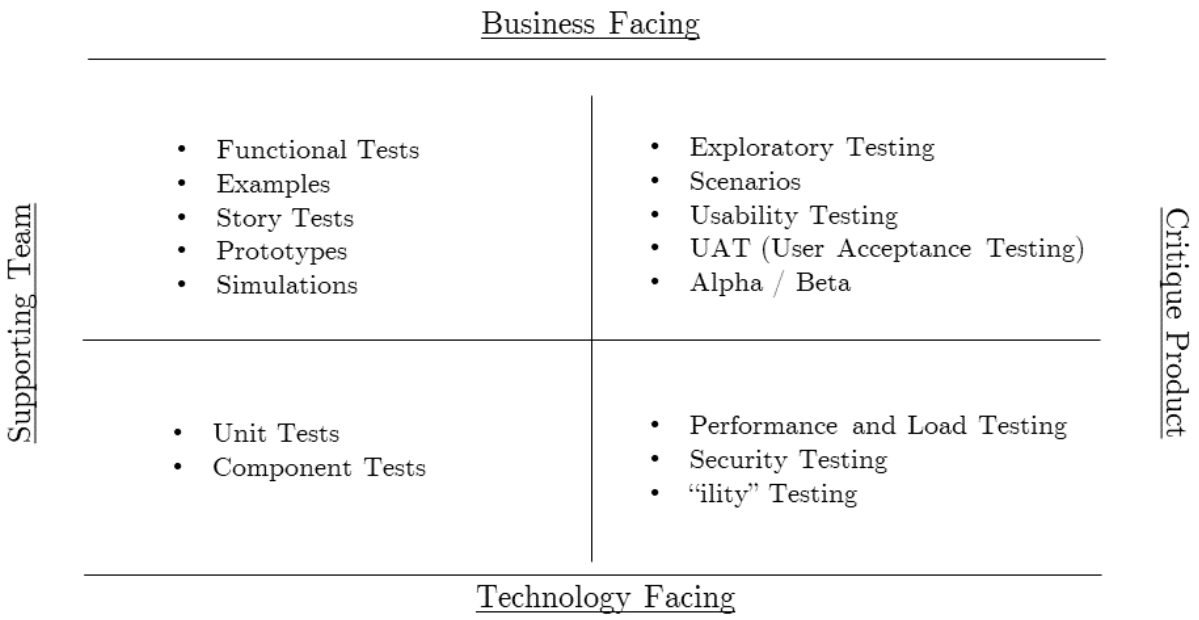


Abbildung 4.2.: Agile Testing Quadrants

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 verwand 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 ein bestimmter Anteil der Interaktionen eine maximale Reaktionszeit von n Sekunden festgelegt wird. Aber auch das zweite Unterkriterium der *Resourcennutzung* darf nicht vernachlässigt werden, so dürfen beispielsweise bestimmte Kommunikationsschnittstellen nicht überlastet werden und diverse Connectionspools 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*. Dieser Begriff wird in der ISO Norm zwar nicht erwähnt, stellt hierbei jedoch eine weiteres Kriterium dar, gerade im Hinblick auf den Anwendungsfall der 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 es ist die Rede einer bestimmten Anzahl von Nachrichten bezüglich eines Brokers, die durch das System gereicht werden. Es kann sich aber auch ganz banal 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. Alles in allem 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 kommt 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 Kapaziätsgrenzen stößt, auch zusätzliche Hardwarekomponenten hinzuziehen 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 arbeiten 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, wodurch eine selbstständige zeitabhängige Skalierung erfolgen kann. Dies macht den Skalierungsprozess robuster, da die Komponente dadurch etwas von seiner Umwelt abstrahiert werden kann (welche in der Regel durchaus fehlerbehaftet sein kann).

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

[1] siehe „*resource utilisatzion*“ der Hauptkategorie „*performance effeciency*“

Kompatabilität ✓✓

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 genau Erfassung erörtert wird. Im Folgenden werden die wesentlichen Kriterien noch einmal zusammengefasst.

- Lediglich *Nicht funktionale Anforderungskriterien* für dieses Projekt von Bedeutung
- Leistungsfähigkeit
  - Zeitverhalten (Reaktions- / Latenzzeit)
  - 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
- Kompatabilität
  - 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 Prototypen ✓

Bei diesem Projekt soll sowohl eine Orchestrierungsplattformen evaluiert als auch ein entsprechendes Framework hinsichtlich der Eignung im Kontext der Kernziele der Thesis verglichen werden (siehe Abschnitt *Kernziele* 2.0.2).

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 sowie 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 fiktiver Workflow ✓

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 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 Auszug 4.1).

```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 oder nicht. Wenn dies nicht der Fall sein sollte, wird die Nachricht zwar acknowledged, sodass sie aus der Eingangsqueue im Message Broker enfernt wird, 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 dem moderneren Spring Boot Framework ersetzt. Spring Boot bietet im Kern 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 erlagen.

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 Arbeitgebers ebenfalls keinerlei genaue Vorgaben hinsichtlich des spezifischen Frameworks beziehungsweise der zu nutzenden Technologie gab, um allerdings ein möglichst repräsentatives Bild zu schaffen, wurde hierbei auf das Node.js Backend-Framework ausgewählt, da dies 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. Anfoderungen an Containerplattform / Orchestrierungsplattform ✓

Unter dem Begriff des „emphContainers“ wird in der Entwicklercommunity meist ein „emphDocker-Container“ verstanden. Dies ist allerdings nicht zwangsweise korrekt. 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 [**?**, 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 Vorteile von Docker wurden bereits relativ 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 Implentierung 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 Portablität 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.]. Desweiteren 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 [**?**, Seite 45 ff.].

Zusammenfassung ✓

- Wesentliche Anforderungen Container
  - Portabilität
  - Reproduzierbarkeit
- Wesentliche Anforderungen Orchestrator
  - Automatisierte Wartung
    - Container Updates
    - Austausch fehlerhafter Container
  - ermöglicht Skalierung bestimmter Services

4.4. Anforderungen an 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 ohne allzu viel Konfigurationsaufwand zur Verfügung zu stellen. Diese Befehle müssen anschließend intern wiederum in detaillierte Nachrichten umgewandelt werden, welche durch das System gereicht 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 zwar einmal testweise erfolgen, für die Generierung der Metriken ist dies allerdings eher hinderlich, 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 Lastentest verwendet. Hierbei würden im Nachhinein allerdings im besten Fall die gleichen Ergebnisse erlangt werden, wie bei Testlauf auf einer einzelnen Maschine.

4.5. Anforderungen Visualisierung und Monitoring zur Unterstützung der Auswertung ✓

Es muss 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.

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 Werte festzustellen, an welcher Stelle genau es zu einem Engpass kommt. Die Unterteilung der gesamten Pipeline sollte zumindest in zwei Abschnitten erfolgen.

- Hierbei ist vor allem von Interesse, ob sich der Zeitenrahmen vom Eingang der Daten bis zum Erhalt der Daten durch die Konsumenten während der Skalierungstests in irgendeiner Weise verändert. Dies sollte zwar nicht der Fall sein, da beide Arten von Testläufen auf dieselbe Eingangsschnittstelle zurückgreifen. Dennoch ist es wichtig festzustellen, falls dies dennoch der Fall sein sollte, denn dann handelt es sich um ein tieferliegendes Problem bei der Orchestrierung des Stacks.
- Außerdem sollte der Zeitrahmen von 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 anstellen 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 oder nicht. 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 Cloudanwendung 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 und benötigte Datenverbindungen bestehen. Darüber hinaus sei noch erwähnt, dass die erhobenen Metriken sollen jeweils getrennt pro Backend-Technologie erhoben werden, 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 eine entsprechende Menge an unbeantworteten Nachrichten hält. Hierbei soll eine sinnvolle Festlegung 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 Benutzeranfrage 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 ungewöhnliche Auschweifungen hinsichlich der zeitlichen Entwicklung abzeichnen lassen, können zusätzliche statistische Metriken hinzugezogen werden.

Im Ausblick erwähnen, dass es möglich ist Datenbanken, Broker etc. zu skalieren Gegenstand der nächsten Arbeit...

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 „*Kompatabilität*“ der Nebenläufigkeit (*Ko-Existenz* sowie *Interoperabilität* siehe Abschnitt 4.1) 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 **??**) wird durch eine funktionierende Implementierung bereits in Teilen erfüllt, da es beim genutzten Orchestrator zum gegenwärtigen Zeitpunkt noch keine Möglichkeit für eine definierte Ausführungsreihenfolge gibt. Die Komponenten müssen im Zweifelsfall mehrfach starten um eine Verbindung aufbauen zu können. Eine genaue Evaluierung sowie eine kritische Betrachtung dieses Aspektes erfolgt in Abschnitt . Sämtliche Anforderungen bezüglich des Prototypen, Orchestrierungsplattform, Lasttest sowie Visualisierung wurden im letzten Abschnitt (siehe 4.2) ausreichend detailliert erläutert und können komplett übernommen werden. Für die Umsetzung dieser Anforderungen siehe Abschnitt *Implementierung* 5.2.

Chaos Monkey Betrachtung an Ende der Thesis packen

Abschnitt einfügen

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 als zeitlich visualisiert darstellen

5.2. Implementierung Prototyp

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 Quellcode sowie alle zugehörigen Konfigurationsdateien befinden sich in einem Github-Repository unter folgender Url `https://github.com/derMacon/stack-scale-benchmark` .

5.2.1. Schichtenmodell ✓

Der Arbeitsfluss der Anwendung wurde in Abbildung 5.1 visuell dargestellt. Die unterschiedlichen Komponenten wurden 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*).

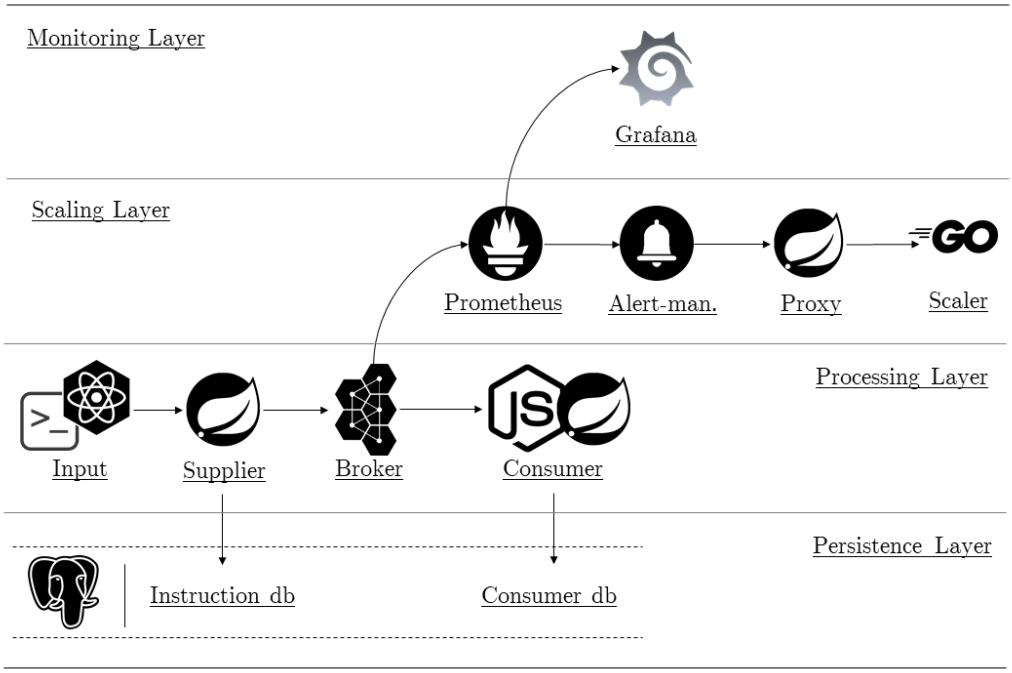


Abbildung 5.1.: Komponenten-Stack im Überblick

- 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 verzichtet eine Lösung zu finden, in der diese ebenfalls in einer Komponente der Persistenzschicht abzuspeichern.

5.2.2. Komponenten im Überblick ✓

Es folgt eine kurze Zusammenfassung der Funktionalität sowie der Anforderungen an die jeweilien 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 können. Allerdings gibt es weitere Skripte, die auf diesem Prozess aufsetzen und somit eine Abstraktionsschicht aufbauen, sodass sich der Endbenutzer hiermit nicht auseinandersetzen muss.

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 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, 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 mehre 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. 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. Es gilt ebenfalls 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) 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.].

Consumer ✓

Die konsumierenden Komponenten können beliebig skaliert werden und implementieren die in Abschnitt 4.2.1 definierten Arbeitsschritte. Hierbei wird auf diverse Libraries zurückgegriffen, wobei die restliche Logik eher schlicht gehalten wurde. Die Komponenten kommunizieren lediglich über

[3] https://github.com/apache/activemq
[4] Erklärung *Load Balancer* siehe Abschnitt 3.1

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.

checken ob Figur am Ende immer noch nic unter die folgende Absatz auf Seite passt, gg aendern

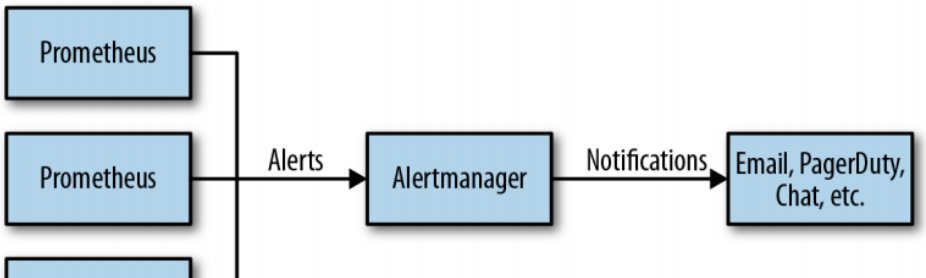


Abbildung 5.2.: Alert Manager - Übersicht [Bra18, Seite 291]

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).

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 weiterzuleitet. In einer produktiven Umgebung würde diese Komponente komplett enfallen, da es möglich ist den Alert-Manager derartig zu konfigurieren, dass er direkt die

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

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 als Umgehung des herkömmlichen Programmflusses und wird zur Generierung der skalierungsschichtenunabhängigen Metrikberechnung verwendet (siehe Abschnitt 6.1.2). 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 starten, wenn bereits aktive Skalierungen vorgenommen werden.

Grafana ✓

Wenn es zum Alert durch den Alert Manager kommt, wird in einer Produktivumgebung der Erste Schritt sein die Performanz des Systems durch dedizierte Dashboard 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*.

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.

Bash ✓

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

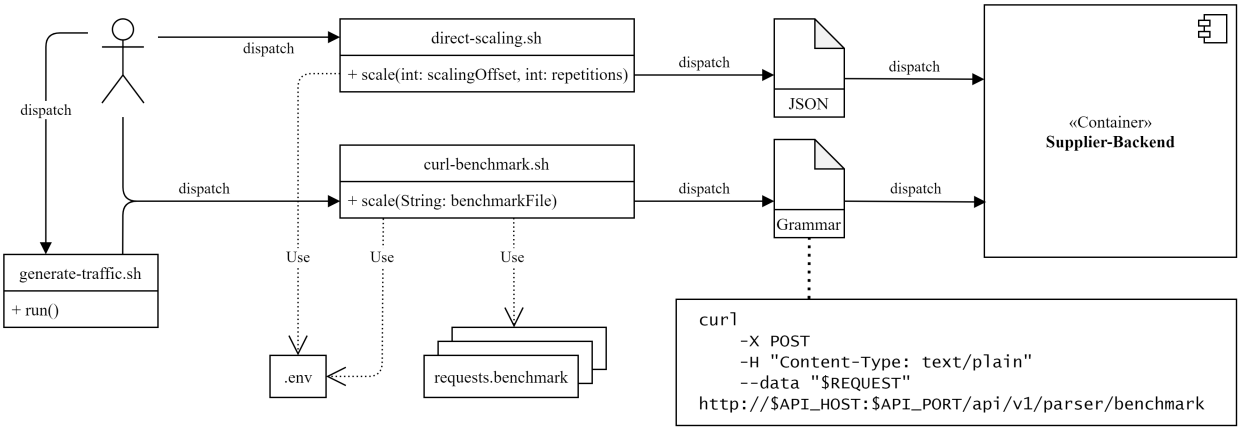


Abbildung 5.3.: Bash Input UML

- *direct-scaling*: 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).
- *curl-benchmark*: sendet eine Skalierungsanfrage mit dem Dateiinhalt einer Datei, dessen Pfad als Eingabeparameter erwartet wird. Die Datei besitzt die Dateiendung *.benchmark* wobei der Dateiinhalt einer spezifizierten Grammatik folgt (siehe 5.2.3).
- *generate-traffic*: dient vereinfachter Einstiegspunkt für den Benutzer. Hierbei werden rekursiv alle verfügbaren *.benchmark* Dateien der Directory und übergibt diese der Reihe nach dem *curl-benchmark* Skript.

Im folgenden Listing (siehe 5.2.3) ist ein Auszug der Projekt-Struktur abgebildet. Neben den genannten Skripts sind sämtliche *.benchmark* Dateien nach services aufgeteilt 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 das auf die IP-Adresse des Systems, falls der Benutzer in der Stack-Konfiguration entsprechende Änderungen vornimmt, sind diese hier ebenfalls zu vermerken.

```
1  $ tree request-scripts/ -a -L 3 --charset=ascii
2  request-scripts/
3  |-- curl-benchmark.sh
4  |-- direct-scaling.sh
5  |-- .env
6  |-- generate-traffic.sh
7  |-- README.md
8  `-- requests
9      |-- mixed
10     |   |-- benchmark_large_mixed.benchmark
11     |   |-- benchmark_medium_mixed.benchmark
12     |   `-- benchmark_small_mixed.benchmark
13     ...
```

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, das 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 verfügbar ist, wird dieser Schritt neunzehn weitere Male wiederholt. Nachdem dieser Schritt abgearbeitet wurde, erfolgt die nächste Skalierung bei der nun zwei neue Container kreiert werden sollen. 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

Skalieren mithilfe von Grammatik Das beschriebene Skript sowie die 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 Beispielskript verfasst. Diese liegen im Ordner *./requests* und tragen die Dateiendung „*.benchmark*“. Der Inhalt korrespondiert zu einer spezifizierten Grammatik (siehe Listing 5.2.3).

```
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 den Anfragenbody angeheftet werden. Eine Batch stellt in diesem Zusammenhang eine Gruppe an Skalierungsanfragen an einen bestimmten Service dar, wobei diese mindestens eine Skalierungsinstruktion enthalten muss. 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 Zahlenwert enthalten. Da das System selbstständig in der Lage 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 schlicht gehaltene Schnittstelle zur Verfügung zu stellen.
2. WAIT: Diese Instruktion erwartet lediglich einen Paramter, 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.

UI Erklärung einbinden...

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.

Implementierung

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 5.4). Sämtliche Packages innerhalb der Spring-Projekte finden sich innerhalb einer der Layer wieder. Es sei ebenfalls hervorzuheben, dass sämtliche Layer stets nur mit ihren direkt angrenzenden Schichten kommunizieren. Dies stellt ein weitaus übersichtlicheres Design sicher.

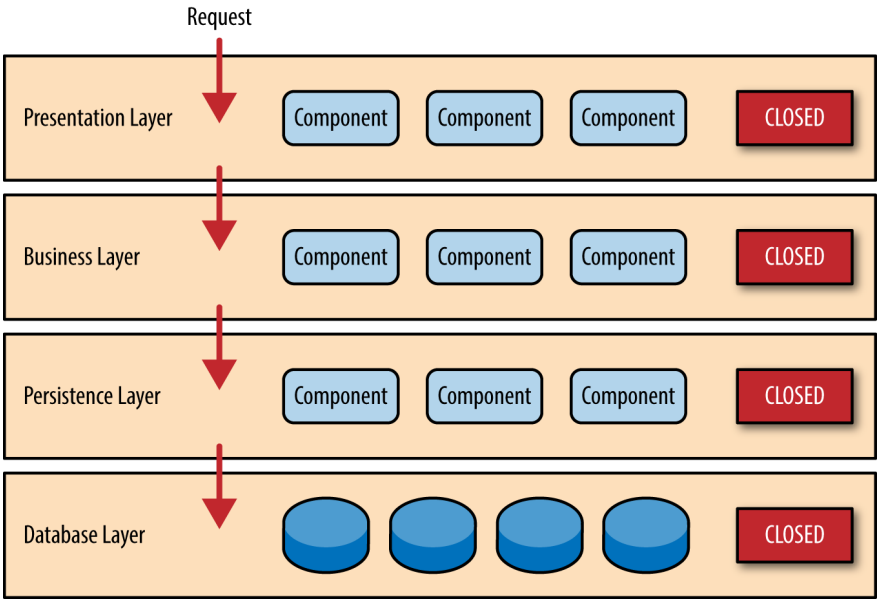


Abbildung 5.4.: Layered Architecture [ore]

- *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.

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, welche spezifische Funktionalität implementieren. Controller dienen hierbei als ein Verteiler der eingehenden Nachrichten zu diesen Services. Die Service-Instanzen werden den Controllern Über den Spring-Context mittels Dependency Injection bereitgestellt, sodass sich der Entwickler selbst sich 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 in den Spring-Context injected werden.
- *repository*: Spring bietet über das Framework die JPA Implementierung *Hibernate* nativen Support für die Anbindung an eine Datenbankverbindung. Dieses Package enthält Interfaces welche die Treiberschnittstelle beerben. Mit Hilfe spezifizierter Namenskonventionen ist es möglich Methodensignaturen Datenbankanfragen zu gestalten. Das Framework generiert hieraus die entsprechenden SQL-Abfragen.
- *config*: Klassen dieses Packages konfiguieren 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 Situation wird jedoch Zugriff auf die Instanz selbst zum Initialisierungszeitpunkt gebraucht um entsprechende Konfigurationsschritte zu unternehmen. Dies geschieht standardmäßig in diesem Package.
- *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.
- *properties*: Spring bietet die Möglichkeit Daten aus den Konfigurationsdateien innerhalb der Resources direkt in den Application-Context zu laden.
- *utils*: Klassen, die lediglich mit Business-Logik gefüllt sind und nicht vom Spring-Container verwaltet werden, sondern direkt im Programmcode instanziiert beziehungsweise aufgerufen werden, residieren in diesem Package.
- *resources*: Die genutzte Verbindungs-Konfiguration zwischen Komponenten des Stacks werden innerhalb der *.properties* Dateien verwaltet. Es ist möglich diese Information über den Dependency-Injection-Mechnismus innerhalb der Klassen hierauf zuzugreifen.

Programmfluss Im folgenden UML-Diagramm (siehe Abbildung 5.5) wurden ein Teil der Klassen des Suppliers dargestellt. Es wurde sich hierbei auf die wesentlichen Logikbausteine konzentriert.

Es gibt im primär zwei 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 Instruktions-Instanz über verschiedene Generator-Implementierungen diverse Callbacks erstellt, die zum absetzen der Nachrichten in das System genutzt werden. Ein Auszug des zugrundeliegenden Codes folgt in Listing 5.4;

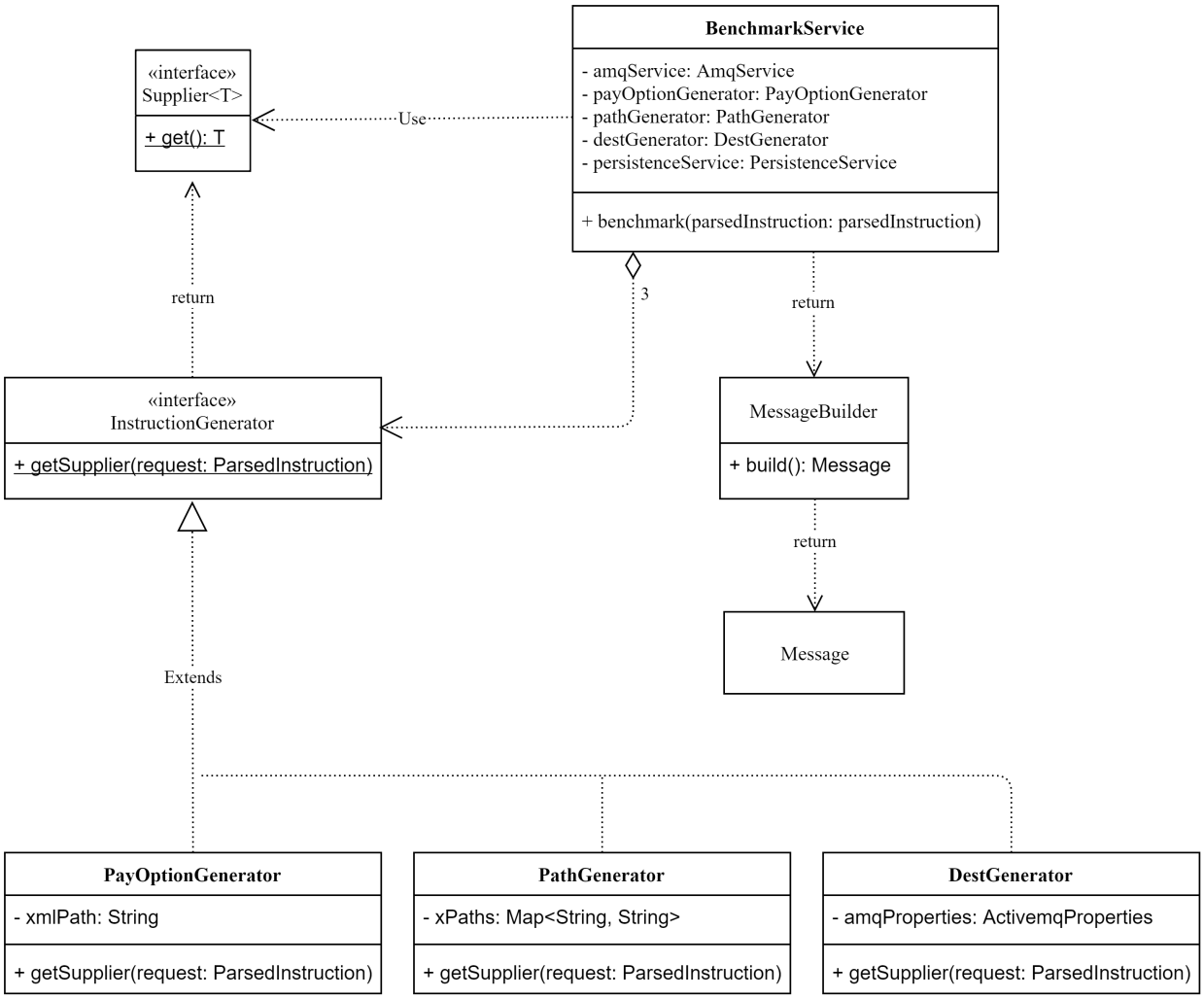


Abbildung 5.5.: Backend Supplier UML

```
    ...

    // create generator instances
    Supplier<String> xPathSupplier = pathGenerator.getSupplier(parsedInstruction);
    Supplier<String> paymentSupplier = payOptionGenerator.getSupplier(parsedInstruction);
    Supplier<String> destinationSupplier = destGenerator.getSupplier(parsedInstruction);
    BiConsumer<PaymentMessage, Supplier<String>> amqConsumer =
           amqService.getConsumer(sessionIsTransacted);

    ...

    for (int i = 0; i < parsedInstruction.getMessageCnt(); i++) {

        // use generator instances to dynamically create message
        PaymentMessage payment = PaymentMessage.builder()
               .batchId(batchId)
               .xPath(xPathSupplier.get())
               .content(paymentSupplier.get())
               .sentTimestamp(now())
               .build();

        amqConsumer.accept(payment, destinationSupplier);

        Thread.sleep(durationMillis);
    }
}

...
```

In dem beschriebenen Listing ist zu erkennen, wie diese Generatoren genutzt werden. Sie extrahieren aus der gegebenen Instruktion-Instanz 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). Der Grund warum hierbei auf diese Callback-Struktur zurückgegriffen wurde, ist der, dass 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 Konsumer 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. Konsumer-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. Sämliche Informationen lassen sich jedoch nahtlos auf die Node.js Umgebung übertragen.

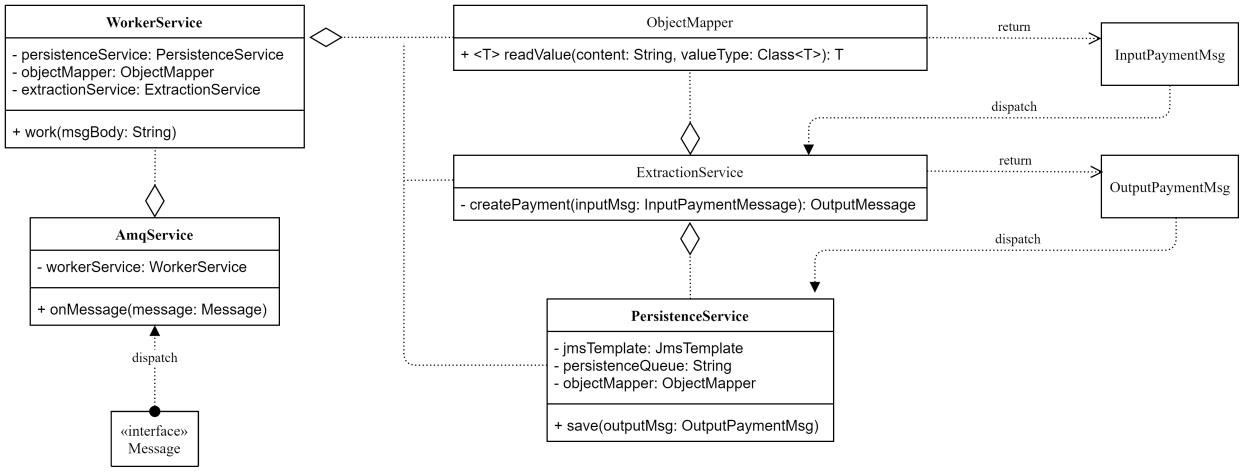


Abbildung 5.6.: Consumer UML

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*) geniert (siehe Listing 5.5). Dieses Objekt wird an einen weiteren Service übergeben, der ein Element entsprechend der Vorgaben im Datenobjekt extrahiert. Wenn das zugrundeliegende 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.

```
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 - Konsumer Logik

5.2.6. Prometheus

Regelsatz für Skalierung Die Prometheus ist eine Komponente, die zum Monitoring genutzt werden kann (siehe Abschnitt 5.2.2). 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. Die Regeln wurden nach folgendem Schema entworfen.

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

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

```
CB0=1     CB2=10     QB0=15     QB2=100
CB1=5     CB3=30     QB1=30
```

Auf der horizontale 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, die Grenzen wurde mithilfe von Variablen festgelegt. Die Grenzen wurden 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 das System zu einem

betrachteten Zeitraum beispielsweise fünf Containerinstanzen einer Konsumenten-Komponente laufen (im nachfolgenden stets als *Consumer-Container* bezeichent). Dann befindet sich das System im Zustand *Container level 1*, da die untere Grenze des Intervals mit *CB0* auf 1 und die obere Grenze auf 5 gesetzt wurde.

Eine ähnliche Eingrenzung gibt es ebenfalls für die vertikale 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 uneantwortete Nachrichten in der Warteschlange residieren, befindet sich das System im Zustand *QL3*, da die *queue count* (kurz *QC*) größer der 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 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 keinerlei weitere Skalierung vorgenommen werden.

Ein wesentlicher Nachteil dieser Herangehensweise ist, dass das System beim Skalieren über mehrere Intervallsgrenzen 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. Es gibt mehrere Möglichkeiten dieses Problem zu umgehen. Es ist zum Beispiel möglich das zeitliche Anfrageninterval nach Steuerungsinformationen vom Broker zu erhöhen, in der Hoffnung, dass das System bereits die gezielte Containeranzahl erreicht hat, wenn 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 sehr inpraktikabler Ansatz darstellt, wurde diese Verantwortlichkeit an den Proxy-Scaler übergeben. Da diese 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 lediglich 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 mit genau dieser 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 minimal Containeranzahl. Hiermit wird beispielhaft beschrieben wie die erläuterten Regeln aus der Tabelle kodiert werden. Das Schlüsselwort *groups* dient lediglich dazu verwandte Regeln zusammenzufassen. Der Rest ist selbsterklärend, erwähnenswert ist lediglich, dass die Wert des Schlüssels *alert* in dieser Form wortwörtlich an den Alertmanager weitergeleitet und dort entsprechend verarbeitet wird.

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

```
$ tree stack/data/prometheus/ --charset=ascii
stack/data/prometheus/
|-- alert-unparsed.yml
|-- alert.yml
|-- prometheus.yml
`-- targets
    `-- alertmanager.json
```

Listing 5.7: Umgebungsvariablen - Prometheus Regelsatz

In der Datei „*alert-unparsed.yml*“ werden die Regeln nach der bisher beschriebenen Struktur abgelegt. Hierbei wurden keinerlei festen 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. 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 befinden sich ansonsten weitere Einstellungen bezüglich des Scraping Intervalls, der Adressierung der zu überwachenden Komponenten sowie die Angabe des Namens des Alert-Managers. Die komplette Konfigurationsdatei befindet sich im Anhang (siehe Listing A.1).

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 einfach gehalten. Es wird lediglich ein Webhook erstellt an den erhaltene Alerts ohne jegliche Modifikation weitergeleitet werden. Im Anhang wurde die komplette Konfiguration der Komponente aufgerführt (siehe Listing lst:alertManConfig).

5.2.8. Scaler Proxy

Im folgenden UML-Diagramm (siehe Abbildung 5.7) 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 Kernfunktionalität dieses Services wurde in Listing 5.8 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). Dies wurde in der Proxy-Komponente über ein Acknowledgement der einzelnen hochgefahrenen Container

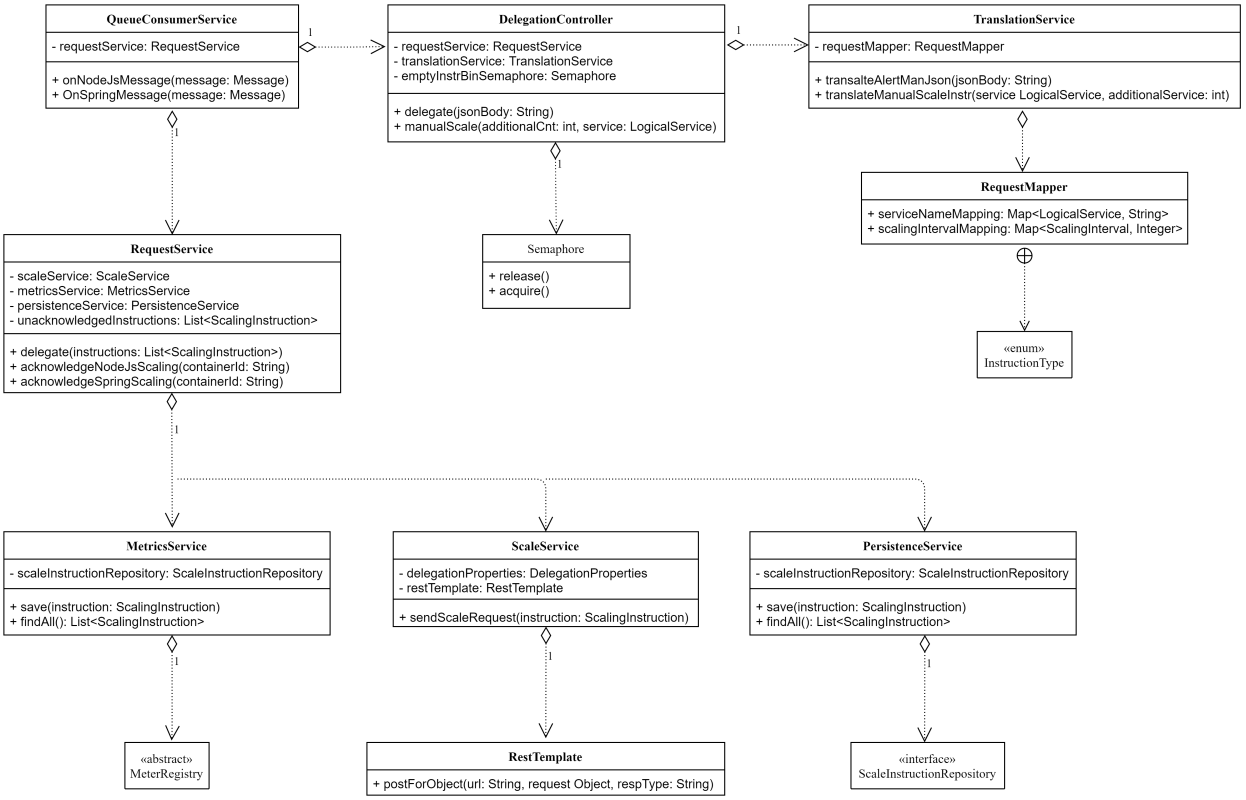


Abbildung 5.7.: Scaler-Proxy UML

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 ist es möglich 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, welche 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 [6] 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 zusätzlichen / überflüssigen Replicas

5.2.10. Grafana

5.2.11. Spring Boot

5.2.12. Node.js

Projektübersicht

Das Projekt beinhaltet neben sämtlichen Quellcode-Dateien ein Dockerfile, eine package.json und eine tsconfig Konfigurationsdatei.

Dockerfile Bei dem verwendeten Dockerfile handelt es sich um eine multi-stage Konfiguration.

Quellen raussuchen

Zeilennu

```
# stage 1 building the code
FROM node:10.15.3 AS builder
WORKDIR /usr/app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# stage 2
FROM node:10.15.3-alpine
WORKDIR /usr/app
COPY package*.json ./
RUN npm install --production

COPY --from=builder /usr/app/dist ./dist

COPY --from=builder /usr/app/schema.sql .
COPY --from=builder /usr/app/specification.xsd .

CMD node dist/src/index.js
```

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

Typescript Konfiguration

Quellcode

```
$ tree stack/node-consumer/ -a -L 3 --charset=ascii
stack/node-consumer/
|-- Dockerfile
|-- .dockerignore
|-- package.json
|-- package-lock.json
|-- schema.sql
|-- scripts
|   `-- buildimage.sh
|-- specification.xsd
|-- src
|   |-- index.ts
|   |-- model
|   |   |-- PaymentInput.ts
|   |   |-- PaymentMessage.ts
|   |   `-- ResultWrapper.ts
|   |-- service
|   |   |-- AmqService.ts
|   |   `-- WorkerService.ts
|   `-- utils
|       |-- ElementExtractor.ts
|       |-- IdGenerator.ts
|       `-- XsdChecker.ts
`-- tsconfig.json
```

Listing 5.9: Node.js Projektstruktur

Die Programmlogik wurde in drei wesentliche packages aufgeteilt (siehe Listing).

Listing einkommentiere und refe renzieren

- *model*: Dieses Package beinhaltet sämtliche Datenstrukturen, die vom System zur Bearbeitung der eingehenden Nachrichten verwendet werden.
- *model*: Dieses Package beinhaltet sämtliche Komponenten welche logisch klar abgetrennte Aufgabe implementieren und über Schnittstellen miteinander kommunizieren. Hierbei wurde sich an der Architektur einer typischen Spring-Anwendung orientiert.
- *model*: Dieses Package beinhaltet sämtliche Funktionalität der abzuarbeitenden Businesslogik.

5.3. Implementierung mittels Containerplattform

5.3.1. Build

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 sogenannten „*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 folgende Beschreibung in gleichem Maß für alle weiteren Spring-Projekte.

Multista Build er klären / Quelle raussu chen

```
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 5.10: Supplier - Bi Consumer

- Zeile 1: Ein Dockerfile beginnt meist 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.

5.3.2. Container Lifecycle

- Auf verschiedene Schichten eingehen
- Auf Ergebnisse beziehen

5.3.3. Docker Swarm

- Prototypen im Detail erlaeutern

5.4. Implementierung Lasttest

5.4.1. Timeline

5.4.2. Testbedingungen

- Kommt in den Anhang
- hat Prof. zwar als eigenes Kapitel erwaehnt, bin mir aber nicht sicher ob das wirklich noetig ist
- auf welcher Hardware werden Tests durchgefuehrt?
- chaos monkey / Stoerfaelle erlaeutern

5.5. Implementierung Visualierung und Monitoring zur Unterstützung der Auswertung

-

Tabelle 5.1.: Server Specs

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

6 Ergebnisanalyse

Nachdem im letzten Abschnitt ausführlich beschrieben wurde wie der komplette Komponentenstack im Detail implementiert 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 wurden im Anhang aufgelistet und sind ebenfalls unter folgender Url einsehbar: https://github.com/derMacon/serverless-bsc-thesis/tree/main/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 Lastentests 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 Stufenmodell[1] . Diese zweite Messung ermöglichte erst die genaue Übersicht über die Containeranzahl im Bezug zur Skalierungszeit (siehe Abbildung 6.3).

6.1.1. Latenzzeit ✓

Die wichtigsten Messdaten bezüglich der Leistungsfähigkeit des Systems beziehen sich auf den Datendurchsatz beziehungsweise die entsprechende Latenzzeit. In der 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 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). Der Gesamtdurchschnitt setzt sich aus zwei Zeitangaben der Pipeline zusammen:

1. 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.

[1] siehe Abschnitt ??

nochmal schauen ob Überleitung noch stimmt wenn Implementierungsteil geschrieben wurde...

Verweis auf Anhang mit allen Tabellen für Metrikinformationen

am Ende nochmal nachschauen ob URL weiterhin stimmt

Messwerte im Anhang noch einmal gesammelt darstellen

Referenz in Fußnote einbinden

Ref einfügen

2. 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 Werte 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, welche 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 brauchen. 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.

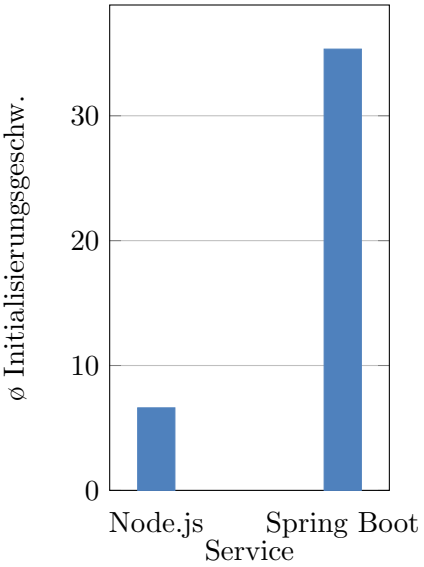


Abbildung 6.1.: Serviceübersicht

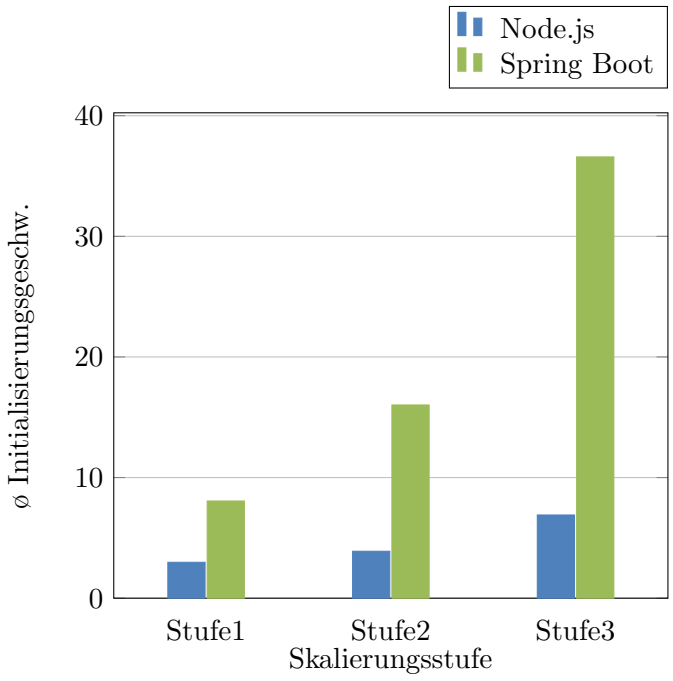


Abbildung 6.2.: Stufenübersicht

Aufschlüsselung nach Skalierungsstufe ✓ Um einen beispielhaften Skalierungsalgorithmus zu implementieren, wurde ein Regelsatz verfasst, der vom Alert-Manager zur Laufzeit automatisch in

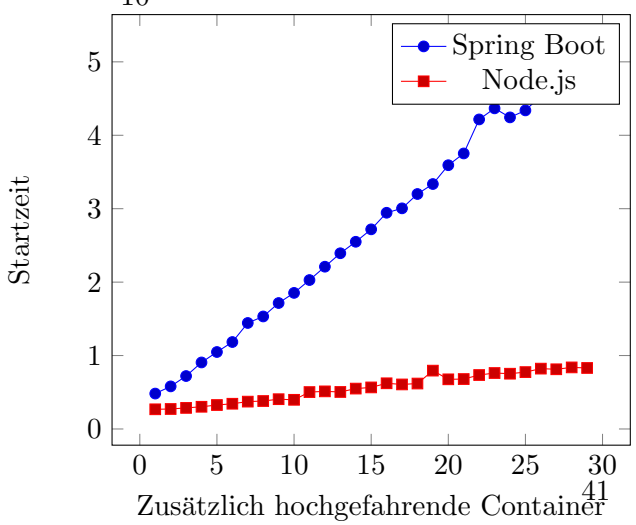
einem festgelegten Intervall ausgewertet wird. Um diesen Regelsatz zu Vorführungszwecken nicht unnötig ausführlich zu gestalten, wurde das Hinzufügen neuer Instanzen in Stufen organisiert.

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 aufseiten von Spring 16.0 Sekunden festgestellt um nach der Skalierung mit 10 Instanzen zu arbeiten. Beim höchsten Burst werden kam es zu einer bei Node.js zu einer Initialisierungsphase von 6.9 Sekunden und bezüglich Spring zu 36.6 Sekunden. Hierbei sollen am Ende 30 Containerinstanzen 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 Paragraphen (siehe Abschnitt 6.1.2) genauer beschrieben.

ref pruefen

Aufschlüsselung nach Containeranzahl ✓ Die grundlegendste Metrik beschreibt die Zeiten aufgeschlüsselt nach Anzahl gleichzeitig hochfahrender Container (siehe 6.3). 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-Koponente lediglich 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 störungsfrei direkt auf die Komponente, die das Skalieren orchestriert zuzugreifen. Die nachfolgenden Metriken wurden allerdings durch die öffentliche Schnittstelle generiert.

Abbildung 6.3.: Anzahlspezifisch



Zeitliche Aufschlüsselung ✓ Wie in Bild 6.4 zu sehen, wurde 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 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.

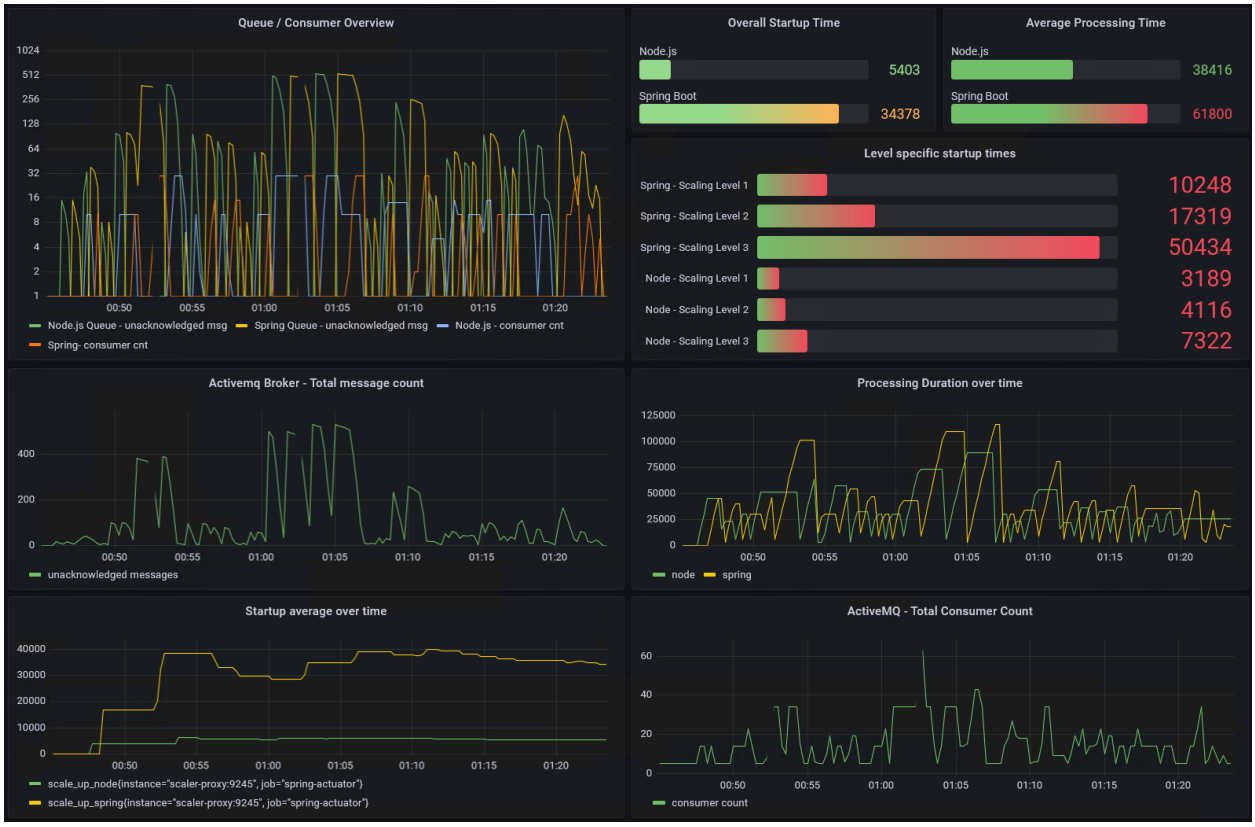


Abbildung 6.4.: Grafana Dashboard

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, während eine Skriptsprache wie Javascript innerhalb der Node.js-Komponente zur Laufzeit 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 so gravierend wie der Unterschied hinsichtlich der Initialisierungsphase der Container, allerdings lässt sich dadurch auch erkenne, dass die Spring Komponente durchschnittliche 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 beschriebenen Initialisierungsoverhead müssen bei der Ermittlung der idealen Threshholds zur Skalierung des Systems betrachtet werden.

Quellen interpretierte vs kompilierte Sprachen

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 Resourcen 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 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).

Die erheblich abweichende Initialisierungsgeschwindigkeit der beiden unterschiedlichen Komponenten lässt sich im Kern auf auf die Initialisierungs des Spring-Containers sowie des dazugehörgien 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 die lose Kopplung der Komponenten, die eine spezialisierte Aufgabe bearbeiten *(hohe Kohäsion)*. Diese Strukturierung ermöglicht eine 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. Der vom Entwickler geschriebene Code wird zur Laufzeit direkt ausgeführt. Dies erschwert zwar den späteren Verlauf eines Projektes, birgt allerdings auch 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.

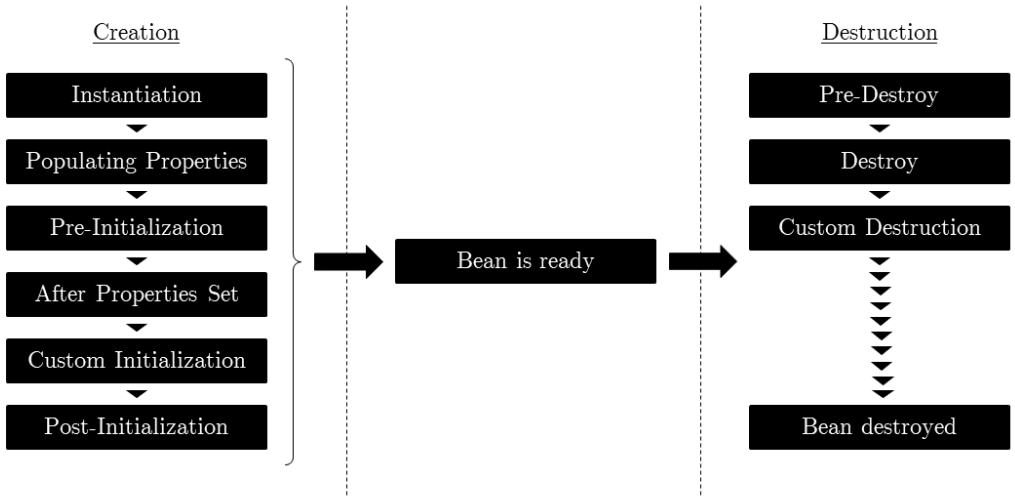


Abbildung 6.5.: Spring Bean Lifecycle [beab]

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 die erstellten Instanzen einem sogenannten „*componentscan*“ unterzogen. Hierbei werden abhängige Beans untereinander mittels dependency injection in den Instanzen referenziert. 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 Kontext abhängige Ressourcen bieten. Über diese Methoden wird dem Framework beispielsweise signalisiert, dass eine Bean, welche das Interface *ApplicationContextAware* implementiert eine Instanz vom Typ *ApplicationContext* injeziert bekommen soll. Ähnliches gilt beispielsweise 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.
2. Ab der Spring Version 4.3 und höher ist es möglich einen Konstruktor zu erzeugen, der lediglich die zu injezierenden 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 eigentlich Initialisierung stattfindet. Hierzu gehören zum Beispiel auch Methoden mit der *PostContstructor* Annotation, welche 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 die der Initialisierungsprozess dem Anwender bietet. Hierbei ist es nicht nur möglich genau nachzuvollziehen, in welcher Phase sich eine Bean momentan befindet, sonder der Prozess bietet die Möglichkeit mittels 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

- Noisy Neighbors: Skalieren in der Cloud bedeutet Kampf um Ressourcen
  - cgroups als wesentlicher Steuerungsmechanismus
  - Erörterung der Konfigurationsmöglichkeiten im nächstem Abschnitt
- Skalierungsdauer: erhöhte Initialisierungsdauer von Spring auf zwei wesentliche Faktoren zurückzuführen
  - Spring Initialisierungsaufwand (IoC, Spring Context, AOP)
  - Resourcenbegrenzung in der Cloud (CPU Shares)
- Exkurs: Spring Initialisierungsschritte
  - Möglichkeiten eigene Logikbausteine unterzubringen
  - Verarbeitungsschritte schaffe 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 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 deutliche 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 ausgebeben werden. Wie im Artikel [spr] beschrieben ist es durchaus möglich einige der Konfigurationen wegzulassen, wenn die dazugehörigen Komponenten nicht genutzt werden.

Es ist außerdem möglich, auf einen anderen Servlet-Container wie „*Undertow*“ zuzugreifen. Dies soll ebenfalls eine Performancsteigerung mit sich ziehen. 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 noch weiterentwickelt und es später vielleicht zu sonderbaren Fehlerfällen kommen könnte.

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 dem Aufteilen der zugewiesenen CPU-Zeit modelliert Docker intern einen Pool mit 1024 Shares. Es ist nun möglich für einen Container anzugeben wie viele Shares dieser vom Scheduler zugewiesen bekommen soll. Falls ein Container beispielsweise lediglich die Hälfte der verfügbaren Resourcen verwenden soll, wird ein Wert von 512 angeben, 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 angebenen 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*“ einzene CPU-Kerne bestimmten Containern zuzuweisen. So ist es beispielsweise 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.

ulimit re
cherchie-
ren S.72

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 dem Hochfahrprozess der einzelnen Container sollte ein ähnliches Verhältnis eingehalten werden. Bisher find 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 jede 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.

ref einbi
den

Um das Verhalten des Systems bezüglich Ausfalls bestimmter Komponenten zu betrachten, 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 [**?**, Seite 369]).

Die Aussage, dass die Spring-Komponente durchschnittliche 13 mal schneller arbeitet ist für den beschriebenen Anwendungsfall zwar korrekt, allerdings wurden hierbei lediglich die beschriebenen Schritte abgearbeitet (siehe Abschnitt **??**). 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 von hiervon nicht betroffen ist. Dies lässt sich mithilfe weiterer Timestamps innerhalb der Komponenten ermitteln.

ref einbi
den

6.3.5. Zusammenfassung

Schauen
ob Begri
„Servlet-
Containe
irgendwo
erklärt
wurde...

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

- 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

7

Zusammenfassung

8 Ausblick

9

Literaturverzeichnis

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.

[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.

[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.

A Anhang

A.1. Imlementierung

A.1.1. Prometheus - Konfiguration

```
global:
      scrape_interval: 1s
      scrape_timeout: 1s
      evaluation_interval: 10s

rule_files:
      - alert.yml

scrape_configs:
      - job_name: services
        metrics_path: /metrics
        static_configs:
             - targets:
                    - 'prometheus:9090'
                    - 'activemq:8080'
      - job_name: spring-actuator
        metrics_path: /actuator/prometheus
        static_configs:
             - targets:
                    - 'scaler-proxy:9245'
                    - 'consumer-persistence:8965'

alerting:
      alertmanagers:
        - static_configs:
            - targets:
                  - alertmanager:9093
```

Listing A.1: Prometheus Konfiguration

```
route:
  receiver: 'proxy'
  group_wait: 1s
  group_interval: 10s
  repeat_interval: 10s

receivers:
  - name: 'proxy'
    webhook_configs:
      - url: 'http://scaler-proxy:9245/delegate'
        send_resolved: false
```

Listing A.2: Alert Manager - Konfiguration

B

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