

Frankfurt University of Applied Sciences

Fachbereich 2: Informatik & Ingenieurwissenschaften

Bachelorarbeit

zur Erlangung des Grades eines

Bachelor of Science (B. Sc.)

im Studiengang

Informatik

Thema: Untersuchung selbstheilender Microservices mit Docker
Autor: Max Byszio <max@byszio.net>

Erstprüfer: Prof. Dr. Christian Baun
Zweitprüfer: Prof. Dr. Thomas Gabel

Abgabedatum: 18.01.2018

EIDESSTATTLICHE VERSICHERUNG

Ich versichere an Eides statt, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Diese Versicherung bezieht sich auch auf in der Arbeit gelieferte Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Unterschrift:

Ort, Datum:

DANKSAGUNG

Hiermit möchte ich allen danken, die mich während meines Studiums unterstützt haben.

Zunächst möchte ich Prof. Dr. Baun und Prof. Dr. Gabel für die Betreuung und Prüfung dieser Arbeit danken.

Besonders danke ich meiner Familie, meiner Freundin und meinen Freunden, die mir stets zur Seite standen und mich motiviert haben.

ZUSAMMENFASSUNG

Diese Bachelorthesis untersucht Lösungen zur Überwachung und Reparatur von Anwendungen, die in Docker-Containern eingesetzt sind. Als bestehende Software sind Kubernetes und Docker-Swarm beschrieben und analysiert. Anhand dessen wird eine prototypische Lösung entwickelt, die auf die Problematiken der bestehenden Softwarelösungen eingeht.

Mit Hilfe von JavaScript, Java und dem Framework Spring ist die entwickelte Lösung umgesetzt. Diese wird genauer beschrieben und die Arbeitsweise dieser ist erläutert. Sie erlaubt es dem Nutzer Container, die in einer lokalen Docker-Instanz ausgeführt werden, zu überwachen und im Problemfall automatisch zu reparieren.

In einer Dokumentation ist erklärt, wie die Lösung aufgesetzt wird und die Einstellmöglichkeiten dieser sind erläutert.

ABSTRACT

This bachelor thesis examines solutions to monitor and repair software that is run in docker containers.

The already established solutions Docker-Swarm and Kubernetes are described and analyzed. Based on the analysis, a new software was developed which covers the flaws that are present in the established solutions.

With the help of JavaScript, Java and the framework Spring the new software was implemented. Further described are the underlying workings of the software and its features. It allows the user to setup automatic monitoring and repairing of containers in a local docker environment.

An added documentation describes the installation of the software and the features that are offered by it.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS	7
1. EINLEITUNG	8
1.1 MOTIVATION.....	8
1.2 ZIELSETZUNG.....	8
2. GRUNDLAGEN	9
2.1 DOCKER.....	9
2.1.1 Begrifflichkeiten.....	11
2.1.2 Aufbau eines Docker-Containers.....	11
2.1.3 Health-Checks.....	12
2.1.4 Verteilte Systeme: Docker-Swarm und Kubernetes.....	13
2.1.4.1 Docker-Swarm.....	13
2.1.4.2 Kubernetes.....	14
2.2 MICROSERVICES.....	18
3. ANFORDERUNGSANALYSE UND KONZEPTION	20
3.1 ZU ERWARTENDE ANWENDUNGSFÄLLE.....	20
3.1.1 Anwendungsfall 1: "Identifizieren von nicht korrekt funktionierenden Anwendungen".....	20
3.1.2 Anwendungsfall 2: "Reparieren von nicht korrekt funktionierenden Anwendungen".....	20
3.1.3 Anwendungsfall 3: "Neu aufsetzen einer Anwendung".....	20
3.1.4 Anwendungsfall 4: "Analyse des Problems nach der Reparatur einer Anwendung".....	21
3.2 FACHLICHE ANFORDERUNGEN.....	21
3.2.1 Anforderungen an die Funktionsüberwachung.....	21
3.2.1.1 Funktionale Anforderungen an die Funktionsüberwachung.....	21
3.2.1.2 Nicht-funktionale Anforderungen an die Funktionsüberwachung.....	21
3.2.2 Anforderungen an die Reparaturfunktion.....	22
3.2.2.1 Funktionale Anforderungen an die Reparaturfunktion.....	22
3.2.2.2 Nicht-funktionale Anforderungen an die Reparaturfunktion.....	22

3.3 ANALYSE BESTEHENDER LÖSUNGEN	22
3.3.1 Analyse von Docker-Swarm.....	23
3.3.2 Analyse von Kubernetes	23
3.3.3 Bewertung bestehender Lösungen.....	24
3.4 KONZEPT	26
3.4.1 Einsatz von Docker	26
3.4.1.1 Architektur	26
3.4.1.2 Speicherung von benötigten Daten	26
3.4.1.3 Berechtigungen und Sicherheit bei der Nutzung von Docker	27
3.4.2 Überwachung der korrekten Funktion einer Anwendung	27
3.4.2.1 Mögliche Wege der Überwachung	27
3.4.2.2 Festlegen der zu überwachenden Anwendungen	27
3.4.2.3 REST- Schnittstelle	28
3.4.2.4 Sichern der Daten der zu überwachenden Anwendung.....	29
3.4.2.5 Sichern von Daten über überwachte Anwendung zu Analyseziwecken	30
3.4.3 Reparieren einer defekten Anwendung	30
3.4.3.1 Schnittstelle zur Beauftragung einer Reparatur	30
3.4.3.2 Möglichkeiten der Reparatur einer Anwendung	30
3.4.3.3 Rückmeldung nach einer Reparatur	31
3.4.3.4 Konzept der Datenbank	31
3.4.4 Aufbau beispielhafter Microservices	32
4. REALISIERUNG	33
4.1 AUFBAU BEISPIELHAFTER MICROSERVICES	33
4.2 AUFBAU DER ANWENDUNG	34
4.2.1 Realisierung der Datenbank	34
4.2.2 Aufbau der REST-API.....	37
4.2.3 Aufbau der Spring Anwendung.....	38
4.2.3.1 Controller	38
4.2.3.2 Services	39
4.2.3.3 Datenbezogene Klassen	40

4.2.4 Aufbau der Funktion zur Überwachung der korrekten Funktionsweise einer Anwendung.....	41
4.2.5 Aufbau der Funktion zum Reparieren einer nicht korrekt funktionierenden Anwendung.....	42
4.2.6 Aufbau der Benutzeroberfläche	43
4.3 EINSATZ DER ANWENDUNG	44
5. BEWERTUNG DER REALISIERUNG	46
5.1 BEWERTUNG BASIEREND AUF DEN GESTELLTEN ANFORDERUNGEN	46
6. DOKUMENTATION DER SOFTWARE	48
7. SCHLUSSBETRACHTUNG	55
7.1 ZUSAMMENFASSUNG.....	55
7.2 IDEEN ZUR WEITERFÜHRUNG DES PROJEKTES	56
QUELLEN.....	57

ABBILDUNGSVERZEICHNIS

GRAFIKEN

Abbildung 1 - Docker auf einem Hostrechner	9
Abbildung 2 - "Copy-on-write" Technologie	10
Abbildung 3 - Health-Status eines Containers in der Konsole	13
Abbildung 4 - Kubernetes Master mit zugewiesenen nodes	15
Abbildung 5 - Übersicht eines Pod	16
Abbildung 6 - Übersicht eines Controllers der Art "Deployment"	17
Abbildung 7 - Übersicht zweier Services	18
Abbildung 8 - Übersicht der API-Schnittstelle	29
Abbildung 9 - Konzeptioneller Aufbau der Datenbank	32
Abbildung 10 - Schema der Datenbank (erstellt mit IntelliJ Idea)	35
Abbildung 11 - Übersicht der REST-API	37
Abbildung 12 - Aufbau der Spring Anwendung	38
Abbildung 13 - Startseite der Benutzeroberfläche	49
Abbildung 14 - Statussymbole	49
Abbildung 15 - Erstellen eines Jobs	50
Abbildung 16 - Hinzufügen von Health-Checks	51
Abbildung 17 - Hinzufügen von Reparaturschritten	52
Abbildung 18 - Liste aller Analytischen Daten eines Jobs	53
Abbildung 19 - Liste der Protokolle	54

CODE-ABBILDUNGEN

Code-Abbildung 1 – Erstellen eines Containers des Python-Microservices	34
Code-Abbildung 2 – Erstellen eines Containers des Java-Microservices	34
Code-Abbildung 3 - Deklaration der Job Controller Klasse	39
Code-Abbildung 4 - Deklaration eines Endpunktes	39
Code-Abbildung 5 - Deklaration der Entitätsklasse Job	40
Code-Abbildung 6 - Deklaration von Attributen	40
Code-Abbildung 7 - Dynamisches füllen einer Tabelle	44
Code-Abbildung 8 – Erstellen des Containers der Binocular-Anwendung	48
Code-Abbildung 9 – Persistieren der Daten des Binocular-Containers	48
Code-Abbildung 10 – Einbinden der bestehenden Daten von Containern	48

1. EINLEITUNG

1.1 MOTIVATION

Die Motivation, in dieser Bachelorarbeit das Thema "Untersuchung selbstheilender Microservices mit Docker" zu behandeln, basiert auf dem Problem, dass es immer unvorhergesehen dazu kommen kann, dass Anwendungen nicht wie gewünscht reagieren oder ganz ausfallen. Mit Docker eingesetzte, Anwendungen werden in sogenannten Containern ausgeführt. Diese sind nur darauf ausgelegt die Anwendung laufen zu lassen. Sie führen jedoch keine weiteren Maßnahmen zur Erhaltung dieser aus. Daher sind sie als kurzlebig zu bezeichnen. Das folgende Zitat beschreibt dies passend:

"Anstatt jede Komponente eines Systems so zu gestalten, dass diese niemals Probleme bekommt, kann man, indem man annimmt, dass die Komponenten abstürzen, das gesamte System stabiler machen, indem man Abstürze beim Aufbau des Systems berücksichtigt." ¹

Auf der einen Seite bietet Docker dem Nutzer die Möglichkeit unkompliziert und schnell Anwendungen auf einem Rechner aufzusetzen und auszuführen, auf der anderen Seite ist Docker eine komplexe Softwarelösung und bietet viele Konfigurationsmöglichkeiten.

Da Docker häufig im Zusammenhang mit Microservices genutzt wird und hierbei eine hohe Verfügbarkeit von großer Bedeutung ist, muss bei Problemen mit laufenden Anwendungen schnell reagiert werden. Daher bietet sich hierbei die Nutzung einer Lösung zur Überwachung und Heilung von Containern an.

1.2 ZIELSETZUNG

Beim Einsatz von Docker auf einem Rechner können Anwendungen zwar neu gestartet werden, doch beseitigt dies häufig nicht das ursprüngliche Problem. Zudem kann sich der automatische Neustart von Containern negativ auf das System auswirken, beispielsweise wenn ein Container durchgehend neu startet und dabei zu Beginn etwas herunterlädt oder eine rechenintensive Operation ausführt.

Eine Lösung für das automatische "Reparieren" defekter Container und deren Anwendungen soll im Rahmen der vorliegenden Bachelorarbeit erarbeitet werden. Hierzu werden zunächst bestehende Lösungen untersucht und verglichen. Abschließend soll mit Hilfe von Docker eine Möglichkeit geschaffen werden, die Anwendungen, die in Containern verpackt sind, automatisch auf ihre gewünschte Funktionsweise hin zu überprüfen und im Problemfall selbstständig zu reparieren.

Wenn eine ausgefallene Anwendung nicht repariert werden kann, soll die gewünschte Funktionsweise dieser durch Erstellen eines neuen Containers der Anwendung wiederhergestellt werden. Denkbare Szenarien, die die Reparatur eines Containers verhindern können, sind beispielsweise fehlerhafte Konfigurationsdateien oder Probleme mit einer Firewall.

¹[Rensin 2015]: Seite 3 (eigene Übersetzung)

2. GRUNDLAGEN

In diesem Kapitel sind die zum späteren Verständnis dieser Arbeit benötigten Technologien und Begriffe erläutert. Hierbei handelt es sich um grundlegende Technologien, Arbeitsweisen und bereits bestehende Lösungen und deren Funktionsweisen.

2.1 DOCKER

Docker ist eine Open-Source Software zur Virtualisierung von Betriebssystemen um darin Anwendungen auszuführen. Es arbeitet mit sogenannten Containern in denen die Systeme laufen, um sie voneinander zu trennen. Diese Container enthalten alle notwendigen Abhängigkeiten, um die vorgesehene Anwendung darin auszuführen, sind unabhängig von dem System auf dem sie laufen und somit auch portabel.

Docker arbeitet auf dem Kernel des Betriebssystems, auf dem es installiert ist (bei Installationen auf Windows und Mac OS x wird eine virtuelle Maschine gestartet in welcher Docker anschließend ausgeführt wird). Zur Kommunikation mit dem Kernel nutzt Docker die Schnittstelle "runc" und unterstützt so einen offenen Standard für die Containerisierung von Systemen. Abbildung 1 zeigt einen Überblick über den schematischen Aufbau einer Docker-Instanz.

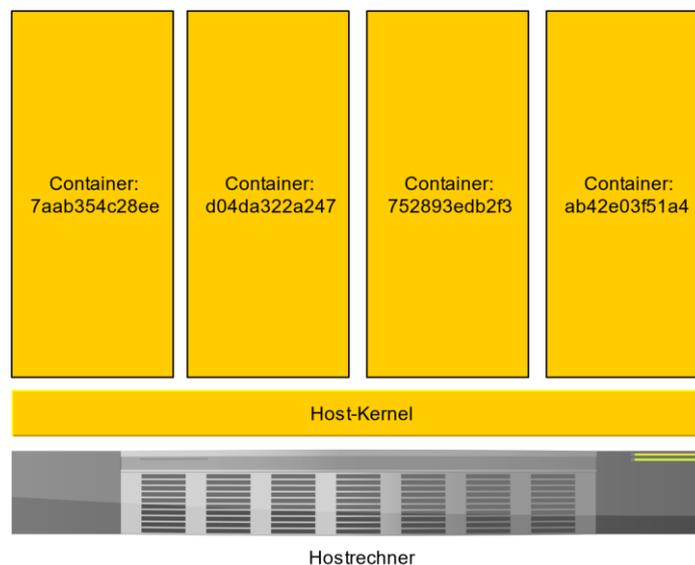


ABBILDUNG 1 - DOCKER AUF EINEM HOSTRECHNER

Dies bietet einige Vorteile gegenüber den üblicherweise zur Virtualisierung von Systemen genutzten virtuellen Maschinen. Da Docker bereits auf dem Kernel des Hostrechners aufbaut, spart es so Ressourcen, die bei einer virtuellen Maschine, durch ein für jedes System von Grund auf laufendes Betriebssystem, genutzt werden würden.

Um die Anwendungen dennoch voneinander zu trennen, erhält jeder Container eine eigene Netzwerkschnittstelle, ein eigenes Dateisystem und einen abgetrennten Namensraum.

Ein weiterer wichtiger Bestandteil von Docker sind Images. Ein Image ist eine Vorlage für ein System und die Anwendung, die darin laufen soll. Jeder Schritt, der bei der Konfiguration dieser Vorlage geschieht, ist in einer Schicht beschrieben, aus der das Image anschließend besteht.

Bei der Erstellung eines Containers aus einem Image bietet sich ein weiterer Vorteil, die so genannte "copy-on-write" Technologie [Abbildung 2]. Diese erlaubt es, dass beliebig viele Container ein Image nutzen können, ohne dass dieses für jeden Container erneut kopiert werden muss. Möglich ist dies durch das Erstellen einer Schicht auf Container-Ebene, die über den Schichten des Images liegt und somit die Dateien des Images "überdeckt".

Wird nun eine Datei bearbeitet, so kopiert Docker sie zunächst aus dem Image in die Schicht auf Container-Ebene und verändert sie dort, so dass das Image für alle Container gleichbleibt, die Datei für den einzelnen Container jedoch verändert ist.

Durch den Aufbau der Images in Schichten und die garantierte Beständigkeit dieser, durch die "copy-on-write" Technologie, kann ein Image aktualisiert werden, indem Docker nur die Schichten herunterlädt, die sich geändert haben.

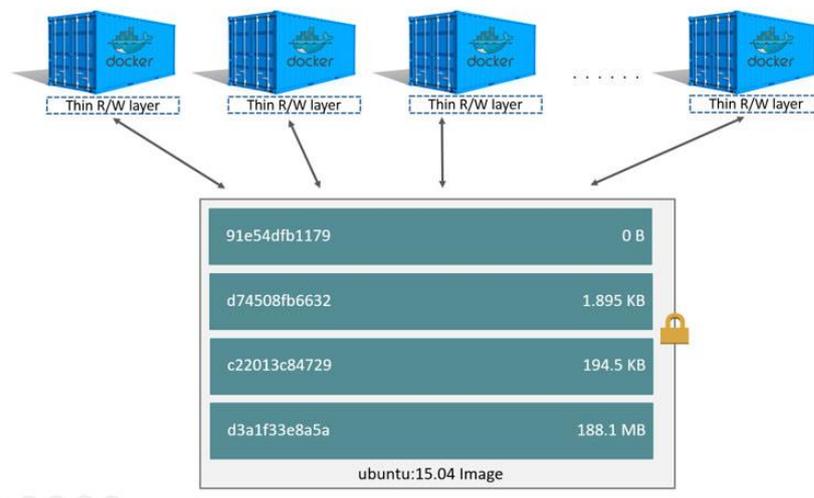


ABBILDUNG 2 - "COPY-ON-WRITE" TECHNOLOGIE²

Da Container unabhängig und immer vorhersehbar laufen sollen, leert Docker die Schicht auf Container-Ebene beim Stoppen des Containers. Persistieren von Daten ist jedoch trotzdem durch das Erstellen von sogenannten Mappings möglich. Dabei wird ein Verzeichnis des Hostrechners in ein Verzeichnis des Containers eingebunden. Das Verzeichnis des Hostrechners überlagert sozusagen das Verzeichnis des Containers und somit werden die Daten im Verzeichnis des Hostrechners gespeichert.

² Quelle: <https://docs.docker.com/engine/userguide/storagedriver/images/sharing-layers.jpg>, abgerufen am 20.11.2017

2.1.1 BEGRIFFLICHKEITEN

Container: Eine für sich geschlossene Umgebung in welcher eine Anwendung läuft.

Dockerfile: Eine Textdatei, die die Befehle zum Erstellen eines Docker-Images enthält.

Image: Eine Vorlage für einen Container, bestehend aus Schichten (sogenannte Layer).

Layer: Eine Schicht eines Docker-Images, die eine Zeile aus dem Dockerfile darstellt, aus dem das Image gebaut wurde.

Mapping: Mapping beschreibt das Einbinden eines Pfades des Hostrechners in das Dateisystem eines Containers. Mappings können beim Erstellen eines Containers angegeben werden.

Repository: Zentrale Inhaltsquelle für bereits gebaute Images.

Docker-Kommandozeilenbefehle (Zusammenfassung):

build:	Startet den Bau eines Images aus dem angegebenen Dockerfile.
create:	Erstellt einen Container aus dem angegebenen Image.
start:	Startet den Container mit dem angegebenen Namen.
stop:	Stoppt den Container mit dem angegebenen Namen.
rm:	Löscht den Container mit dem angegebenen Namen.
pull:	Lädt das angegebene Image aus einem Repository herunter.
push:	Lädt das angegebene lokale Image in ein Repository hoch.

[vgl.: Matthias, Kane 2016; Docker 2017a]

2.1.2 AUFBAU EINES DOCKER-CONTAINERS

Um einen Docker-Container zu erstellen wird zunächst ein Image benötigt. Dieses kann entweder aus einem Repository heruntergeladen oder selbst erstellt werden. Beim Erstellen eines Images ist es ebenfalls möglich, aber nicht nötig, auf ein Repository zurückzugreifen und auf einem schon bestehenden Image aufzubauen.

Ein Image, welches selbst erstellt wird, muss zunächst aus einem Dockerfile gebaut werden. Das Dockerfile enthält die Kommandos, die später beim Bauen des Images ausgeführt werden sollen. Images sind über so genannte Tags genauer spezifiziert. Beim Bauen eines Images kann ein Tag angegeben werden, unter welchem das Image zu finden ist. Dies dient beispielsweise der Versionierung von Anwendungen, da jedes Tag eine andere Version zur Verfügung stellen kann. Standardmäßig wird beim Nutzen eines Images ohne Spezifizieren eines Tags der „latest“ Tag verwendet, welcher auf die aktuellste Version des Images verweist.

Für das Dockerfile gilt eine vorgegebene Syntax, bestehend aus einer definierten Zahl von Instruktionen.

Die Wichtigsten sind hierbei:

- `ADD src dest`: Hinzufügen der Datei/des Ordners `src` zum Dateisystem des Containers durch Kopieren nach `dest`. `src` kann auch eine URL oder ein Archiv sein und wird entsprechend heruntergeladen und/oder entpackt.
- `CMD command param1...paramN`: Standardmäßig beim Start des Containers auszuführender Befehl `command` mit Parametern `param1` bis `paramN`. Sobald die Ausführung des Befehls abgeschlossen ist oder auf eine andere Weise gestoppt wurde, ist auch der Container gestoppt.
- `COPY src dest`: Hinzufügen der Datei/des Ordners `src` zum Dateisystem des Containers durch Kopieren nach `dest`.
- `EXPOSE port/protocol`: Freigeben eines Port `port` aus dem Container heraus, um ihn im Docker Netzwerk verfügbar zu machen.
- `HEALTHCHECK command`: Überprüfen des Containers basierend auf dem Kommando `command`. Unter 2.1.3 Health-Checks beschrieben
- `FROM imagename`: Basisimage `imagename` auf dem das zu bauende Image basieren soll.
- `RUN command param1...paramN`: Ausführen des Befehls `command` mit Parametern `param1` bis `paramN` in der Kommandozeile des Containers.
- `VOLUME path`: Erstellen eines Volumens des Dateisystems des Containers an Stelle `path`. Dieses Volumen kann später in andere Container oder vom Hostrechner eingebunden werden.
- `WORKDIR path`: Setzen des aktuellen Pfades auf `path` in dem gearbeitet werden soll.

Nachdem das Dockerfile erstellt wurde, kann das entsprechende Image gebaut werden. Dabei werden in chronologischer Reihenfolge die im Dockerfile angegebenen Kommandos ausgeführt. Jedes dieser Kommandos bildet eine Schicht des fertig gebauten Images. Schlussendlich kann der Nutzer mit dem gebauten Image ein Container erstellen, in dem die fertige Anwendung, so wie sie im Dockerfile beschrieben wurde, läuft. [vgl.: Matthias, Kane 2016; Docker 2017b]

2.1.3 HEALTH-CHECKS

Docker bietet von Haus aus die Möglichkeit, so genannte Health-Checks bei der Erstellung eines Containers zu definieren. Diese werden als "HEALTHCHECK" bei der Definition innerhalb des Dockerfiles bezeichnet, beziehungsweise wird "--health-cmd" als Argument bei der Erstellung eines Containers über die Kommandozeile übergeben.

Ein Health-Check besteht aus einem Kommando welches der Docker-Daemon regelmäßig auf der Kommandozeile des Containers ausführt. Der Nutzer definiert dieses Kommando. Es soll bei korrekter Funktion der Anwendung im Container 0 und bei fehlerhafter Funktion 1 zurückgeben. Ein Container erhält, entsprechend seiner korrekten Funktion den Status "healthy" oder "unhealthy", also korrekt funktionierend, beziehungsweise nicht korrekt funktionierend. Ist noch kein Health-Check ausgeführt worden, so

erhält der Container den Status „starting“.
Bei der Definition eines Health-Checks lassen sich weitere Parameter angeben:

- interval:** Das Intervall, in dem der Health-Check des Containers ausgeführt werden soll.
- timeout:** Die Zeit, die der Health-Check ausgeführt wird, bevor er als fehlgeschlagen gilt.
- start-period:** Die Zeit, die nach dem Start des Containers gewartet wird, bevor der erste Health-Check ausgeführt wird.
- retries:** Die Anzahl an Health-Checks die fehlschlagen müssen, bevor dem Container der Status „unhealthy“ zugewiesen wird, er also als nicht korrekt funktionierend gilt.

Die Übersicht aller Container zeigt bei diesen, die einen Health-Check angegeben haben, ihren entsprechenden Status [Abbildung 3].

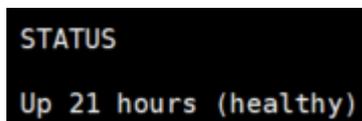


ABBILDUNG 3 - HEALTH-STATUS EINES CONTAINERS IN DER KONSOLE

Über „docker events“ veröffentlicht der Docker-Daemon verschiedene Statusnachrichten zur Docker-Instanz und den Containern. Unter anderem veröffentlicht er bei der Änderung des Health-Status' eines Containers dessen neuen Status als „health_status“ Event. Events erlauben es, bestimmte Statusmeldungen automatisch zu erhalten. Die Notwendigkeit periodisch den Docker-Daemon nach den Änderungen, die im Event enthalten sind, abzufragen, wird somit umgangen. [vgl.: Docker 2017b]

2.1.4 VERTEILTE SYSTEME: DOCKER-SWARM UND KUBERNETES

Docker-Swarm und Kubernetes sind Softwarelösungen zum verteilten Rechnen mit Hilfe von Docker. Gerade bei der Nutzung von auf Containern basierender Architektur ist das Verteilen der Last, die das gesamte Konstrukt zu tragen hat, wichtig. Dem Anwender kann so ein problemfreies Arbeiten mit dem System ermöglicht werden.

2.1.4.1 Docker-Swarm

Docker-Swarm ist auf zwei verschiedenen Weisen einsetzbar: „Swarm-Standalone“ und „Swarm-Mode“.

Swarm-Standalone ist eine zusätzliche Installation zu Docker und bietet die Möglichkeit bestehende Docker-Instanzen so zu vereinen, dass diese als eine „virtuelle“ Docker-Instanz zusammengefasst werden. Auf dieser Instanz kann anschließend, wie auf einer einzigen Docker-Instanz, mit den bekannten Kommandos gearbeitet werden. Nachteil

hierbei ist das komplexe Zusammenschließen der einzelnen Instanzen und das Fehlen von wichtigen Funktionalitäten, beispielsweise eines so genannten Load-Balancers³.

Bereits seit Version 1.12 in Docker integriert ist der "Swarm-Mode", ebenfalls eine Möglichkeit Docker-Instanzen auf verschiedenen Servern zu vereinen. Im Gegensatz zu "Swarm-Standalone" werden die Instanzen nicht über die bekannten Docker-Kommandos, sondern über ein weiteres Set an Befehlen gesteuert. Statt Containern werden so genannte Services genutzt. Diese verteilt Swarm über einen integrierten Load-Balancer auf die Instanzen als so genannte Tasks. Ein Service ist eine Anwendung, die später in dem verteilten System laufen soll, während ein einzelner Task der eigentliche Container ist, der die Anwendung auf einer der Instanzen im Swarm-Cluster laufen lässt. Bei der Erstellung eines Service wird festgelegt, wie oft dieser innerhalb der verteilten Systeme repliziert werden soll. Des Weiteren können bei der Erstellung Argumente genutzt werden, die von der Erstellung eines einzelnen Docker-Containers bekannt sind, beispielsweise das Verknüpfen eines Ports oder eines Verzeichnisses auf dem Hostrechner mit dem Container. Ein weiterer Vorteil im Vergleich zu Swarm-Standalone ist hierbei, dass die Instanzen noch einzeln angesprochen werden können und somit flexibler sind. [vgl.: Docker Inc. Soppelsa, Kaewkasi 2016]

2.1.4.2 Kubernetes

Kubernetes ist eine von Google als Open-Source-Software entwickelte Lösung für die Bereitstellung, Verteilung und Verwaltung von Containern.

Kubernetes lässt sich mit dem Docker-Swarm Mode vergleichen, bietet darüber hinaus jedoch noch mehr Funktionalität und somit auch Komplexität. Beispielsweise bietet es Unterstützung für andere Containerisierungs-Lösungen, auf die jedoch in diesem Umfang nicht eingegangen wird.

Kubernetes arbeitet in Clustern, also Verbunden von einzelnen Rechnern, in einer sogenannten Master-Slave-Architektur. Die Master-Slave-Architektur besteht, aus einem Hauptrechner, dem Master, und ihm untergeordneten Rechnern, den Slaves. Diese werden in Kubernetes als Node bezeichnet. In Abbildung 4 ist diese Architektur zusammen mit den in der Node laufenden Hilfsprogrammen visualisiert.

Der Master nimmt im Falle von Kubernetes die vom Nutzer gewünschten Befehle an und verteilt diese an die Slaves. Er bietet darüber hinaus diverse Verwaltungsanwendungen für das Cluster, unter anderem eine API-Schnittstelle (Application Processing Interface), über die sich die Nodes mit ihm verbinden und über welche die Aufgaben verteilt werden.

Die Nodes sind lediglich dafür ausgelegt Container laufen zu lassen und werden über einen ständig ausgeführten Prozess namens „kubenet“ überwacht und gesteuert. Über diesen Prozess kommunizieren Master und Node miteinander. Der Master kann so feststellen sobald ein Node-Rechner nicht mehr verfügbar ist oder diesem mitteilen falls Anwendung, ein so genannter Pod, auf diesem gestartet werden soll. Neben der ebenfalls

³ Load-Balancer: Eine Softwarelösung zum Verteilen von Last auf mehreren Systemen.

auf jeder Node laufenden Containerisierungs-Lösung, ist zur Konfiguration der komplexen Netzwerkstruktur die Anwendung "kube-proxy" mit installiert. Diese sorgt für korrekt funktionierende Routen zum Master und zwischen den Nodes. So ist es möglich, dass sich der Master und die Nodes in verschiedenen Netzwerken befinden.

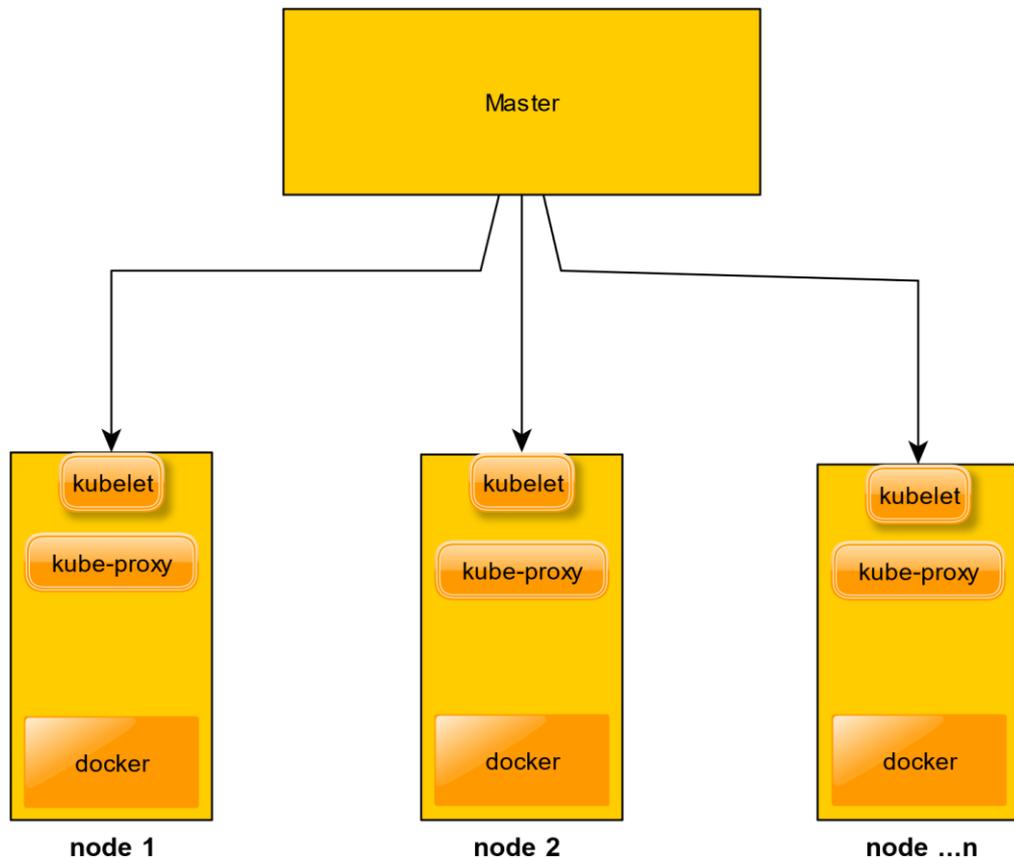


ABBILDUNG 4 - KUBERNETES MASTER MIT ZUGEWIESENEN NODES

Der wichtigste Bestandteil von Kubernetes sind Pods [Abbildung 5], sie stellen die eigentlich laufende Anwendung dar. Ein Pod läuft auf einer Node, ist ein einzelner Container oder eine Gruppe von Containern und umfasst alles, was diese benötigen. Dazu zählen Netzwerk, Speicherplatz und Bedingungen, die die laufende Anwendung erfüllen soll.

Ein Pod stellt einen "logischen Host" zusammen, einen virtuellen Zusammenschluss mehrerer Container. Dieser wird auch genutzt, wenn der Pod nur einen Container beinhaltet, bietet in diesem Fall jedoch keine Vorteile. Der "logische Host" sorgt dafür, dass Anwendungen in verschiedenen Containern, genau so laufen, als wären sie auf einem Rechner zusammen eingesetzt. Dies bringt gerade bei Anwendungen, die aufeinander angewiesen sind, Vorteile, da zwischen Docker-Containern getrennte Ressourcen geteilt werden können (bspw. „namespaces“, „cgroups“, internes Netzwerk und IP-Adresse).

Mehrere Container lassen auf diese Weise miteinander verknüpfen, sodass diese eine für sich geschlossene Architektur bilden. Ein Microservice muss deshalb nicht, wie sonst üblich in einem einzelnen Container zur Verfügung gestellt werden, sondern kann aus mehreren Containern in einem Pod bestehen.

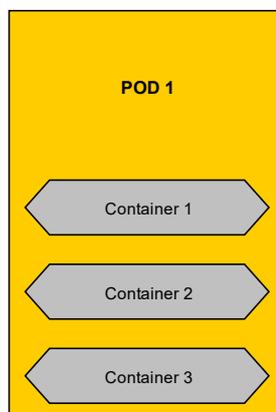


ABBILDUNG 5 - ÜBERSICHT EINES POD

Das Erstellen eines Pods geschieht üblicherweise nicht direkt vom Nutzer, sondern über einen Controller. Da ein Pod nicht darauf ausgelegt ist, im Fall einer Fehlfunktion oder eines anderen Problems weiterhin zu funktionieren, übernimmt der Controller diese Aufgabe. Er erstellt und überwacht die benötigten Pods entsprechend des gewünschten Status dieser.

Ein von Kubernetes mitgelieferter Controller nennt sich "deployments" und wird von der Syntax, ähnlich wie ein Pod, erstellt. Als Vorteile stellt er die Möglichkeit zur Verfügung, automatisch Replikationen der von ihm verwalteten Pods zu erstellen und diese mit so genannten "liveness and readiness probes" zu überwachen. Im Problemfall werden die Replikationen eingesetzt, um die defekten Pods neu bereitzustellen. Ein Controller umhüllt sozusagen den zu erstellenden Pod, wie in Abbildung 6 visualisiert.

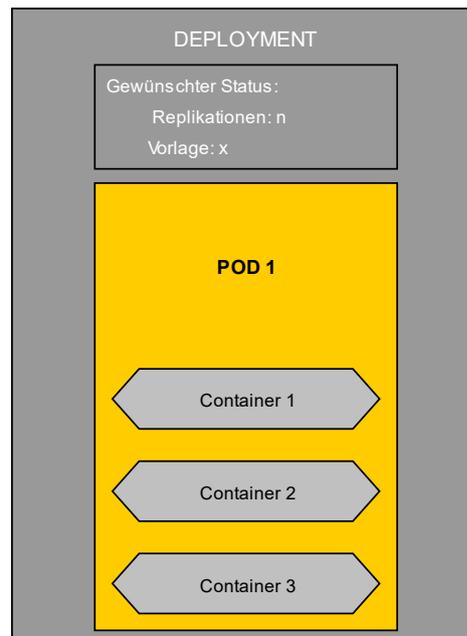


ABBILDUNG 6 - ÜBERSICHT EINES CONTROLLERS DER ART "DEPLOYMENT"

Problematisch ist bei Nutzung eines Controllers die korrekte Veröffentlichung der Pods nach außen, sodass der Nutzer immer auf den gewünschten Pod weitergeleitet wird. Da die verschiedenen Replikationen der Pods auf verschiedenen Nodes im Kubernetes-Cluster laufen können und diese über verschiedene IP-Adressen erreichbar sind, bietet Kubernetes mit Services eine Lösung für dieses Problem. Ein Service besitzt eine eigene IP-Adresse und ist der für den Nutzer gedachte Ansprechpunkt der Anwendung. Die eingesetzten Pods und Replikationen dieser sind dem Service bekannt, so dass dieser die Anfrage des Nutzers entsprechend, zwischen diesen verteilen kann. Abbildung 7 zeigt die beispielhafte Architektur eines Kubernetes-Clusters mit zwei Pods und jeweils einer Replikation. Diese sind auf zwei Nodes verteilt und werden durch jeweils einen Service für jeden Pod an den Nutzer freigegeben. [vgl.: Resin 2015; Kubernetes 2017]

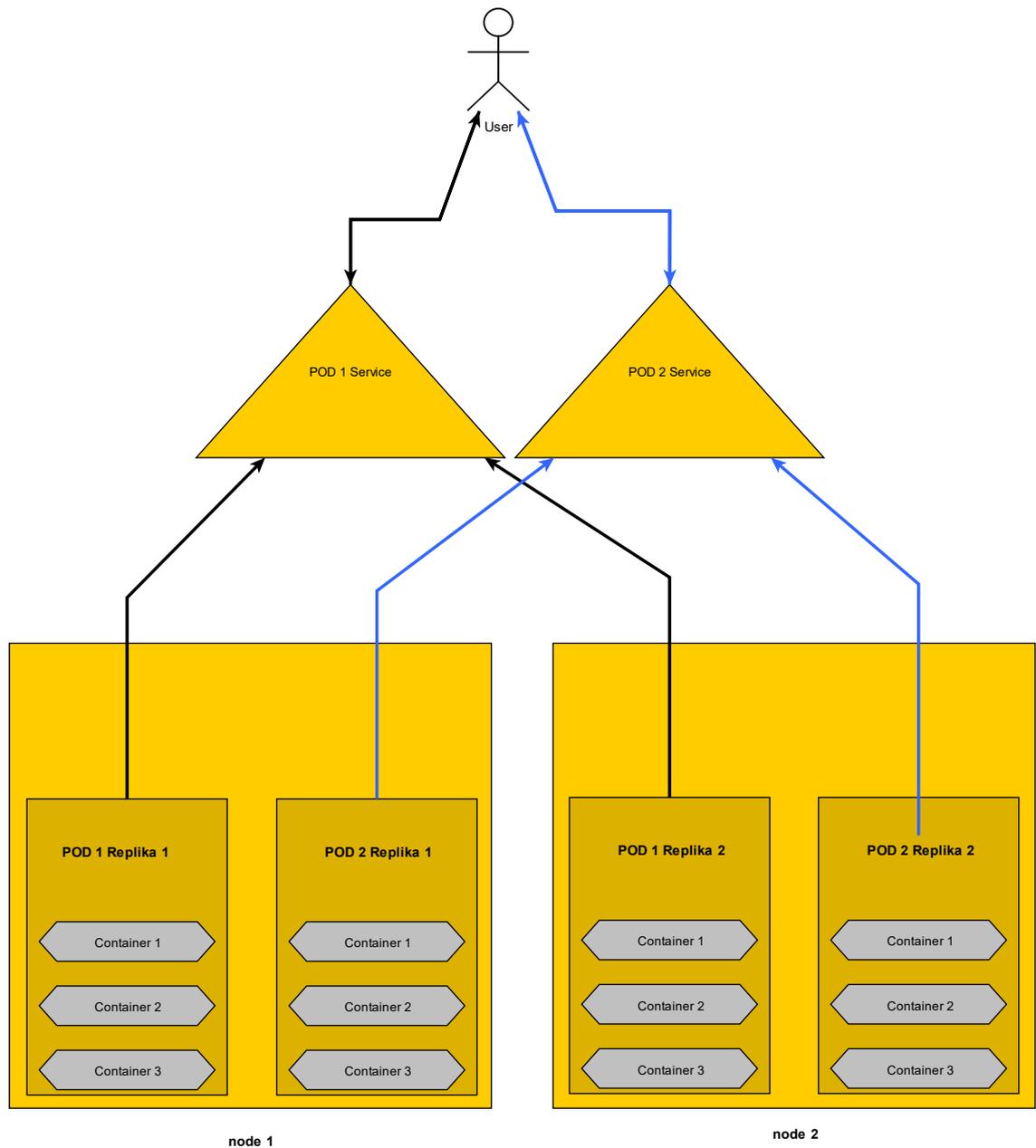


ABBILDUNG 7 - ÜBERSICHT ZWEIER SERVICES

2.2 MICROSERVICES

Microservices sind ein Architekturstil beim Erstellen von Anwendungen in der Informatik.

Im Gegensatz zu den früher häufig entwickelten monolithischen Anwendungen, die die gesamte Funktionalität enthielten, wird bei der Entwicklung von Microservices jede Funktionalität der Anwendung separat entwickelt, versioniert, getestet und eingesetzt. Hierbei werden die einzelnen Microservices nach fachlichem Zusammenhang getrennt und so klein wie möglich gehalten. Ein Entwicklerteam ist anschließend für die

Konzeption bis hin zur Fertigstellung dieses Microservices zuständig und arbeitet an einem genau definierten Teil der kompletten Architektur.

Die Aufteilung in Microservices bringt den Vorteil der einfacheren Wartung der einzelnen Komponenten und ermöglicht den problemlosen Austausch dieser, sowohl innerhalb der Software, als auch später beim Einsetzen der Software auf Servern. Hieraus ergibt sich auch der Vorteil von Docker in Zusammenhang mit Microservices, da diese einfach auf verschiedenen Servern einsetzbar sind und nur bedingt voneinander abhängig sind.

Durch die Unabhängigkeit der einzelnen Komponenten ist es möglich, verschiedene Programmiersprachen und/oder verschiedene Technologien innerhalb der einzelnen Anwendungen zu nutzen. Um die reibungslose Kommunikation zwischen den einzelnen Komponenten zu gewährleisten, sieht die Microservice-Architektur einen untereinander unabhängigen Kommunikationsweg vor. Dieser wird meist über HTTP mit Hilfe von APIs, die die einzelnen Komponenten zur Verfügung stellen, realisiert. Wichtig ist, dass der Weg der Kommunikation, zum Beispiel die entsprechenden API-Schnittstellen, genau definiert und dokumentiert werden, um diese später einfach austauschen zu können.

Durch den Einsatz von mehr Komponenten gegenüber monolithischen Systemen, entsteht hier ein Verlust von Ausfallsicherheit, da mehr Systeme auch mehr Probleme haben können.

Ein weiterer Nachteil besteht darin, dass die einzelnen Komponenten zunächst mehr Entwicklungszeit in Anspruch nehmen, da jede für sich entwickelt werden muss. Unter Umständen kann es so, auch durch von jedem Microservice einzeln benötigte Abhängigkeiten und Laufzeitumgebungen, zu einem Mehrverbrauch an Ressourcen kommen. [vgl.: Newman 2015]

3. ANFORDERUNGSANALYSE UND KONZEPTION

Der Grundstein für eine gut funktionierende Anwendung ist zunächst die Analyse der Anforderungen an die Software und das Erstellen eines Konzeptes dieser. Dadurch kann der Zeitaufwand abgeschätzt werden und es ist klar, welche Funktionen und Nebenbedingungen die Software erfüllen muss.

Im folgenden Abschnitt werden die Anforderungen an die prototypische Lösung gestellt, anhand derer diese abschließend bewertet wird, und es wird ein Konzept für die fertige Lösung erarbeitet.

3.1 ZU ERWARTENDE ANWENDUNGSFÄLLE

Zunächst müssen die von der Software zu erfüllenden Funktionalitäten festgelegt werden.

Dies geschieht anhand der zu erwartenden Anwendungsfälle.

3.1.1 ANWENDUNGSFALL 1: "IDENTIFIZIEREN VON NICHT KORREKT FUNKTIONIERENDEN ANWENDUNGEN"

Der erste Schritt um Anwendungen zu "reparieren" ist zu erkennen, ob diese überhaupt ein Problem haben.

Eine Anwendung, deren Status und Bedingungen für die korrekte Funktion dem System bekannt sind, soll regelmäßig auf die gegebenen Bedingungen hin überprüft werden.

Wird hierbei eine nicht korrekt funktionierende Anwendung entdeckt, soll diese an die nächste Funktionalität, der Funktion zu Reparatur der nicht korrekt funktionierenden Anwendung gemeldet werden.

3.1.2 ANWENDUNGSFALL 2: "REPARIEREN VON NICHT KORREKT FUNKTIONIERENDEN ANWENDUNGEN"

Sobald eine nicht korrekt funktionierende Anwendung entdeckt und gemeldet wurde, sollen Schritte eingeleitet werden, um diese zu reparieren.

Hierzu soll zunächst der Container, in dem die Anwendung läuft, neu gestartet werden. Sollte dies die Funktionalität der Anwendung nicht wiederherstellen, sollen weitere angegebene Schritte unternommen werden. Dazu zählt das Entfernen von eventuell angegebenen Mappings oder erneutes Neustarten des Containers. Falls dies das Problem nicht löst und die Anwendung innerhalb eines definierten Zeitraumes erneut gemeldet wird, soll diese an die nächste Funktionalität gemeldet werden, die die Anwendung neu aufsetzt.

3.1.3 ANWENDUNGSFALL 3: "NEU AUFSETZEN EINER ANWENDUNG"

Sobald die Anwendung nicht repariert werden konnte, soll diese neu aufgesetzt werden. Das heißt der alte Container, in dem die Anwendung lief, soll gestoppt und ein neuer Container mit der Anwendung soll gestartet werden. Die alten Daten des gestoppten Containers bleiben erhalten und können vom neuen Container weiter genutzt werden.

3.1.4 ANWENDUNGSFALL 4: "ANALYSE DES PROBLEMS NACH DER REPARATUR EINER ANWENDUNG"

Sobald eine Anwendung repariert oder neu aufgesetzt wurde, soll das Vorkommnis mit Zeitangaben und Protokollierung des Fehlers dokumentiert werden. So ist es dem Nutzer später möglich, den aufgetretenen Fehler nachvollziehen und beheben zu können. Bei einem neu aufgesetzten Container soll hierzu der alte Container erhalten bleiben, sodass der Nutzer diesen untersuchen kann.

3.2 FACHLICHE ANFORDERUNGEN

Die Analyse der fachlichen Anforderungen lässt sich in zwei Bereiche unterteilen. Aufgrund der vorher beschriebenen, zu erfüllenden Anwendungsfälle können die funktionalen Anforderungen festgelegt werden, die Beschreibung der Funktionalität. Der zweite Bereich, die nicht funktionalen Anforderungen, hängt sowohl von der einzelnen Funktionalität ab und beschreibt die Qualität, die die Funktionalität liefern muss. Dies umschließt unter anderem wie schnell, genau oder benutzerfreundlich die Funktionalität arbeiten muss.

3.2.1 ANFORDERUNGEN AN DIE FUNKTIONSÜBERWACHUNG

Die Funktionsüberwachung ist dafür zuständig Anwendungen, die in Containern auf der lokalen Docker-Instanz laufen, auf ihre korrekte Funktion hin zu überwachen.

3.2.1.1 Funktionale Anforderungen an die Funktionsüberwachung

Dem Nutzer muss es möglich sein, die zu überwachenden Anwendungen inklusive der Überwachungsparameter anzugeben. Zu diesen Überwachungsparametern zählen, die Bedingungen, unter denen eine Anwendung als korrekt funktionierend gilt, die Abstände, in denen die Anwendung überprüft werden muss und die Anzahl der fehlgeschlagenen Überprüfungen, bevor weitere Maßnahmen eingeleitet werden.

Ebenso muss es dem Nutzer möglich sein, die Anwendung, die überwacht werden soll und die Angaben zu einer zu überwachenden Anwendung zu verändern oder einen Überwachungsauftrag über eine Anwendung komplett zu löschen.

Die zu überwachenden Anwendungen müssen periodisch, basierend auf den Angaben des Nutzers, auf ihre korrekte Funktion hin überprüft werden. Hierbei muss die Funktionsüberwachung erkennen, falls eine Anwendung die Überprüfung nicht bestehen kann, beispielsweise wenn sie momentan neu startet oder aktualisiert wird.

Sobald eine nicht funktionierende Anwendung auffällt, muss dies protokolliert und weitergeleitet werden, damit sie repariert werden kann.

3.2.1.2 Nicht-funktionale Anforderungen an die Funktionsüberwachung

Bei der Überprüfung einer Anwendung auf ihre korrekte Funktionsweise darf die Anwendung dadurch nicht zu viele Anfragen erhalten. Basierend auf den Spezifikationen des Systems, muss eine angebrachte Zeit zwischen den Überprüfungen ermittelt werden.

Des Weiteren muss die Überprüfung der Funktionsweise zeitlich so reagieren, dass eine nicht korrekt funktionierende Anwendung möglichst früh erkannt wird und der Nutzer so nur eine möglichst kurze Zeit die Anwendung nicht nutzen kann.

Es darf zudem keine Anwendungen fälschlicherweise als nicht korrekt funktionierend eingestuft werden. Hierzu ist es notwendig diese eventuell mehrmals zu überprüfen.

Die Erstellung neuer Überwachungsaufgaben muss der Nutzer in einer verständlichen Benutzeroberfläche erledigen können. Die Erstellung, Änderung oder Löschung eines Auftrages muss möglichst wenig Handlungsbedarf des Nutzers erfordern.

3.2.2 ANFORDERUNGEN AN DIE REPARATURFUNKTION

Nachdem eine Anwendung als nicht korrekt funktionierend erkannt wurde, kommt die Reparaturfunktion zum Einsatz, die versucht die Anwendung wieder zu ihrer korrekten Funktion zu bringen.

3.2.2.1 Funktionale Anforderungen an die Reparaturfunktion

Bereits beim Anlegen einer Überwachungsaufgabe muss es dem Nutzer möglich sein anzugeben, wie im Falle einer Reparatur vorgegangen werden soll. Ist dies nicht erfolgt, muss die Reparaturfunktion eine vordefinierte Routine abarbeiten, die versucht die Anwendung zu reparieren.

Die Anwendung muss die Angaben des Nutzers, beziehungsweise die vorgefertigte Routine, sukzessiv durchlaufen und nach jedem Teilschritt muss geprüft werden, ob dies die korrekte Funktion der Anwendung wiederhergestellt hat.

Sollte keiner dieser Schritte die Anwendung repariert haben, muss der Nutzer umgehend darüber informiert werden, so dass er das Problem beheben kann.

3.2.2.2 Nicht-funktionale Anforderungen an die Reparaturfunktion

Bei der Reparatur der Anwendung ist es notwendig, dass stets nur die tatsächlich von der Funktionsüberwachung weitergeleiteten Anwendungen repariert werden. Muss eine Anwendung repariert werden, dürfen keine Daten verloren gehen, da diese zur Analyse des Problems essentiell sind. Ebenso müssen die Parameter, mit denen die ursprüngliche Anwendung eingerichtet wurde, erhalten bleiben um die korrekte Funktionsweise nicht einzuschränken.

Eine weitergeleitete Anwendung muss umgehend repariert werden, damit sie so schnell wie möglich wieder verfügbar ist.

3.3 ANALYSE BESTEHENDER LÖSUNGEN

Im Folgenden werden hierbei Docker-Swarm und Kubernetes betrachtet, da diese beiden Lösungen momentan die bedeutendsten Technologien sind, die Docker auf verteilten Systemen ermöglichen. [vgl.: Hecht 2016]

Beide Technologien erlauben das Verteilen von Docker-Containern auf verschiedene Rechner. Hierbei bieten sowohl Kubernetes, als auch Docker-Swarm, die Möglichkeit diese Container zu überwachen und bei festgestellten Problemen zu reagieren.

3.3.1 ANALYSE VON DOCKER-SWARM

Bei der Nutzung von Docker-Swarm wird über Services bestimmt, wie welche Anwendungen laufen sollen. Ein Service ist vergleichbar mit einem Container, der auf mehreren Rechnern im verteilten System laufen kann.

Basierend auf der angegebenen Anzahl an gewünschten Replikationen werden anschließend, entsprechend der Tasks, Container erstellt. Die Anfragen an diese werden durch den, von Docker-Swarm gelieferten Load-Balancer, verteilt.

Ebenso wie Docker bietet Docker-Swarm die Möglichkeit bei der Erstellung eines Services Health-Checks mit anzugeben und so den Service auf die korrekte Funktion hin zu überwachen. Diese Health-Checks werden auch auf die Tasks übertragen und somit Containern dieses Services zugewiesen.

Passiert es nun, dass ein Container eines Tasks nicht mehr korrekt funktioniert und der entsprechende Health-Check fehlschlägt, so wird der Task inklusive Container entfernt, ein neuer Task vom Service erstellt und durch den Load-Balancer zur Verfügung gestellt.

3.3.2 ANALYSE VON KUBERNETES

Ebenso wie Docker-Swarm bietet Kubernetes eine Lösung zur Überwachung und automatischen Replikation und Reparatur von Anwendungen.

Bei der Nutzung eines „deployments“ werden mit Kubernetes eingesetzte Pods mit sogenannten „liveness and readiness probes“ überwacht und bei Fehlfunktion automatisch neu aufgesetzt.

Bei der Erstellung eines Pod kann für jeden seiner Container angegeben werden unter welchen Umständen er als „live“, also funktionierend gilt. Es können Befehle angegeben werden, die wie ein Docker-Health-Check funktionieren und bei einem Fehlercode während der Ausführung des Kommandos den Container als defekt markieren. Es können ebenfalls direkt HTTP oder TCP Anfragen an den Container gesendet werden. Anhand der Antwort bestimmt Kubernetes den Status des Containers.

Bei der Erstellung einer „probe“ kann angegeben werden, ab welchem Zeitpunkt ab Start des Containers und in welchem Intervall diese ausgeführt werden soll.

Ist eine Anwendung mit Berechnungen oder Laden einer Ressource beschäftigt und kann deshalb eine „liveness probe“ nicht bestehen, so kommt eine „readiness probe“ zum Einsatz. Diese „readiness probe“ sorgt dafür, dass der Container, in der Zeit in der er aufgrund einer Aufgabe beschäftigt ist und nicht reagieren kann, keine weiteren Anfragen erhält, um ihn nicht weiter zu belasten.

Diese „readiness probes“ sind in ihrer Funktion identisch zu „liveness probes“ und bieten die gleichen Möglichkeiten der Überwachung. Sie sollten als sinnvolle Ergänzung zu „liveness probes“ angesehen werden. Bei der Einrichtung dieser sollte man darauf achten, dass sie häufiger als „liveness probes“ ausgeführt werden, um dem Container Zeit zu geben seine Aufgaben abzuarbeiten.

Die Nutzung von “liveness and readiness probes” in Pods, welche einem Service zugeordnet sind, bringt weitere Vorteile, als nur das simple Neustarten des defekten Containers.

Bei der Fehlfunktion eines Containers und einer fehlgeschlagenen “probe”, leitet der Service an diesen Container zunächst keine Anfragen weiter, sondern leitet sie an weitere ihm bekannte Container weiter.

Beim Aktualisieren eines Pods oder eines eingesetzten Controllers wartet Kubernetes immer zunächst darauf, dass die aktualisierte Version läuft, bevor die alte Version gestoppt und entfernt wird. Hierbei bringt der Einsatz von “readiness probes” den Vorteil, dass diese zunächst erfolgreich sein müssen, bevor die alte Version entfernt wird. Während dies geschieht, wird durch den Service noch die alte Version zur Verfügung gestellt, um einen Ausfall zu verhindern. Dies verringert das Risiko nach und während Aktualisierungen Probleme mit nicht korrekt laufenden Anwendungen zu bekommen.

3.3.3 BEWERTUNG BESTEHENDER LÖSUNGEN

Die bestehenden Lösungen, müssen anhand der gegebenen Anforderungen bewertet werden, um aufzuzeigen, welche Vorteile eine eigens entwickelte Lösung bringen würde.

Docker-Swarm bietet dem Nutzer mit Health-Checks eine Möglichkeit seine eingesetzten Anwendungen auf ihre korrekte Funktionsweise hin zu überwachen.

Sobald eine Anwendung als “unhealthy” eingestuft wird, wird dies zusammen mit dem letzten Teil des Logs des Containers protokolliert. Zum Erstellen, Ändern oder Löschen eines Health-Checks ist es notwendig, dass der entsprechende Container entfernt und von Hand neu angelegt wird. Entsprechend der nicht-funktionalen Anforderungen zur Funktionsüberwachung benötigt dies viel Handlungsbedarf durch den Nutzer.

Bei der Konfiguration eines Swarm Health-Checks ist es nicht möglich, eine Routine anzugeben, die versucht dem bestehenden Container wieder zur korrekten Funktionsweise zu verhelfen. Der Container wird lediglich durch einen identischen, neu aufgesetzten Container ersetzt. Der alte Container, mit Hilfe dessen eine Analyse des Problems möglich gewesen wäre, wird entfernt.

Unter Einbeziehung der von Docker zur Verfügung gestellten Dokumentation zu Health-Checks, fällt auf, dass diese zunächst in keinem Zusammenhang mit Docker-Swarm stehen und der Nutzer keinen direkten Überblick über die Funktionsweise von Health-Checks und Reparaturen von Containern mit Docker-Swarm erhält. Diese werden nicht als Funktionalität von Docker-Swarm hervorgehoben, sind jedoch vorhanden.

Die Nutzung dieser bereits bestehenden Funktionalität soll mit der zu entwickelnden Lösung erleichtert werden, so dass ein leicht verständlicher Weg mit grafischer Oberfläche zur Verwaltung der Überwachung von Containern zur Verfügung steht.

Kubernetes bietet dem Nutzer eine umfassende Funktionalität zur Reparatur von eingesetzten Containern. Diese ist umfangreicher als bei Docker-Swarm, geht aber mit der erhöhten Komplexität von Kubernetes einher.

Bei Erkennung einer nicht korrekt funktionierenden Anwendung wird diese zunächst nicht mehr dem Nutzer präsentiert, sondern es wird eine Replik zur Verfügung gestellt, sofern dies so eingerichtet wurde. Dies erhöht die Verfügbarkeit, entsprechend der nicht-funktionalen Anforderungen. Der Vorfall wird zusammen mit den letzten Ausgaben der Anwendung protokolliert.

Zudem bringt die Unterscheidung zwischen "liveness" und "readiness probes" einen klaren Vorteil, da nicht korrekt funktionierende Anwendungen zwar zunächst nicht mehr dem Nutzer präsentiert werden, aber eine gewisse Zeit erhalten, um ihre Aufgaben abzuarbeiten. Dies hilft dabei, fälschlicherweise als defekt erkannte Anwendungen zu erhalten.

Bei Kubernetes ist es dem Nutzer ebenfalls nicht möglich eine Routine anzugeben, nach der zunächst versucht wird, der Anwendung wieder zur korrekten Funktion zu verhelfen. Eine Anwendung, in diesem Fall in einem Pod, wird bei erkannten Problemen neu aufgesetzt und die alte Version entfernt. Dies führt dazu, dass die Analyse des ursprünglichen Problems erschwert wird.

Auch wird dem Nutzer keine grafische Oberfläche zum Verwalten der Überwachung seiner Container geboten.

Zusammenfassend lässt sich feststellen, dass die bestehenden Lösungen zwar kurzfristig gegen ein Problem helfen können, dem Nutzer jedoch die Analyse des Ursprungs dessen erschweren. Defekte Anwendungen werden sofort entfernt, wodurch der Nutzer keine Möglichkeit hat, genau nachzuverfolgen, wie das Problem entstanden ist und wie es in Zukunft vermieden werden kann.

Des Weiteren fehlt dem Nutzer mit den bestehenden Lösungen die Möglichkeit zu bestimmen, was im Falle eines Problems mit einer Anwendung geschehen soll um diese zu reparieren.

Um dem Nutzer den Einstieg und die Wartung der Überwachung so komfortabel wie möglich zu gestalten, fehlt die grafische Oberfläche. Docker-Swarm und Kubernetes bieten zwar Lösungen für die Reparatur von Anwendungen in einem Cluster, jedoch ist keine Lösung für die Reparatur von Anwendungen auf einer einzelnen, lokalen Docker-Instanz verfügbar. Dies soll die zu entwickelnde Lösung erreichen.

3.4 KONZEPT

Bevor eine Lösung zur Überwachung und Heilung von Anwendungen in Docker-Containern erstellt werden kann, muss zunächst ein Konzept erarbeitet werden.

Anhand dieses Konzeptes kann anschließend die Anwendung aufgebaut werden.

Die Anwendung soll vom Nutzer über eine Web-Oberfläche erreichbar sein. Deshalb ist die Erstellung einer REST-API, die später von einer Web-Applikation angesprochen werden kann, geplant. Die REST-API soll durch ein Backend, welches in Java geschrieben ist, zur Verfügung gestellt und über ein JavaScript-Frontend dem Nutzer präsentiert werden. Java bietet im Zusammenhang mit REST-APIs diverse bestehenden Frameworks und eine aktive Gemeinschaft von Interessierten und Entwicklern.

JavaScript in Zusammenhang mit einer Oberfläche wie Bootstrap bietet dem Nutzer eine übersichtliche Oberfläche und wird heutzutage von fast jeder Webseite⁴ genutzt.

Diese Aufteilung in zwei Komponenten dient der Vereinfachung der Entwicklung und Wartung der Anwendung. Die Web-Oberfläche ist so unabhängig von der im Hintergrund laufenden Lösung.

3.4.1 EINSATZ VON DOCKER

Bei der zu entwickelnden Lösung ist Docker ein wichtiger Bestandteil der Funktionalität. Da Kubernetes und Swarm bereits tiefgehende Möglichkeiten zur Überwachung bieten, wird die zu entwickelnde Lösung auf lokale Docker-Instanzen ausgelegt sein.

3.4.1.1 Architektur

Dem Nutzer soll eine minimale Installation geboten werden, um einen einfachen Einstieg in das Thema der Überwachung und Heilung von Anwendungen in Docker-Containern zu bieten. Dazu soll die Lösung in einem Docker-Container laufen, welcher über die, bei Docker standardmäßig eingerichtete, "Docker Hub Repository" zur Verfügung gestellt wird.

Über die Schnittstelle "docker.sock", einem Unix-Socket, welcher über ein Mapping vom Hostrechner in den Container eingebunden ist, kann die Lösung mit der Docker-Instanz auf dem Hostrechner kommunizieren.

3.4.1.2 Speicherung von benötigten Daten

Alle Daten, die von der Lösung erstellt werden, sollen in einem zentralen Ordner liegen, der über ein Mapping auf Dateisystem persistiert wird. Dieser Ordner muss mit Lese- und Schreibrechten in den Container eingebunden werden.

Alternativ können die Daten in einem Docker-Volume persistiert werden, sofern gewünscht.

⁴ Vgl.: <https://w3techs.com/technologies/details/cp-javascript/all/all>, abgerufen am 02.12.2017

3.4.1.3 Berechtigungen und Sicherheit bei der Nutzung von Docker

Die Nutzung der "docker.sock" Schnittstelle stellt stets ein Risiko dar, da ein Container der Zugriff darauf hat, auf den Hostrechner zugreifen kann. Dies kann beispielsweise durch das Erstellen eines Containers geschehen, der das gesamte Dateisystem des Hostrechners als Mapping erhält.

Die größten Sicherheitsrisiken muss der Nutzer handhaben. Er kann entscheiden, ob Anwendungen von außerhalb des Netzwerkes erreichbar sind oder welche Ressourcen, Schnittstellen und Mappings diese erhalten.

Bei der Erstellung eines Containers und dem Nutzen von Mappings kann durch falsches Setzen von Schreib- und Leseberechtigungen Datenverlust entstehen. Auch hierauf soll der Nutzer aufmerksam gemacht werden.

Um den Nutzer in seinen Möglichkeiten die Lösung einzusetzen nicht einzuschränken, soll die zu entwickelnde Lösung keine Sicherheitsmerkmale bieten.

3.4.2 ÜBERWACHUNG DER KORREKTEN FUNKTION EINER ANWENDUNG

Die Konzeption der Überwachung einer Anwendung ist notwendig, um festzulegen, wie diese überwacht werden soll.

3.4.2.1 Mögliche Wege der Überwachung

Da Anwendungen Funktionen bieten können, die für den Nutzer über verschiedene Wege erreichbar sein müssen, sollten möglichst viele dieser Wege überwachbar sein.

Eine Möglichkeit dies zu tun, ist das Nutzen der von Docker gegebenen Überwachungsmethode, der Health-Checks. Diese bieten die Funktion, über die Kommandozeile des Containers Befehle auszuführen, mit denen die Anwendung auf ihre korrekte Ausführung hin geprüft werden kann.

Eine weitere Möglichkeit wäre die Entwicklung eines Frameworks, welches in die Anwendung implementiert ist und regelmäßig Statusmeldungen zum Status dieser an eine Überwachungssoftware sendet.

Um dem Nutzer eine möglichst selbsterklärende und unkomplizierte Lösung zu bieten, ist die Nutzung von Health-Checks die hierbei präferierte Methode. Der größte Vorteil gegenüber der zweiten Methode ist die Unabhängigkeit von der eigentlichen Anwendung und der Sprache, in der sie geschrieben ist.

3.4.2.2 Festlegen der zu überwachenden Anwendungen

Um dem Nutzer die Handhabung der Anwendung zu erleichtern, soll es ihm möglich sein, die zu überwachenden Anwendungen in einer Weboberfläche anzugeben.

Dazu soll die zu entwickelnde Anwendung einen Webserver implementieren, der sie nach außen hin verfügbar macht. Mit Hilfe einer CSS-Bibliothek, wie Bootstrap⁵, kann diese für den Nutzer, ansprechend gestaltet, präsentiert werden.

Hierbei soll der Nutzer aus bereits bestehenden Anwendungen, die in Containern in der Docker-Instanz auf dem Hostrechner laufen, wählen. Um dies zu ermöglichen, ist vorgesehen, dass die Lösung mit der "docker.sock" Schnittstelle und somit dem Docker-Daemon kommuniziert.

Ein Auftrag zur Überwachung einer Anwendung wird im Folgenden Job genannt.

Die für die Überwachung gewählten Anwendungen sollen anschließend mit den angegebenen Überwachungsparametern neu aufgesetzt werden, so dass sie umgehend von den Überwachungsmechanismen profitieren.

Bei der Festlegung eines Jobs sollen folgende Parameter unterstützt werden:

- Name des zu überwachenden Containers
- Health-Checks Parameter:
 - health-interval (Intervall, in dem der Health-Check ausgeführt wird)
 - health-timeout (Zeit, die ein Health-Check benötigen darf, bevor er als fehlgeschlagen gilt)
 - health-retries (Anzahl der Versuche bevor der Container repariert wird)
- Health-Checks, die ausgeführt werden:
 - Kommando, das ausgeführt wird
- Vorgehen im Falle eines Problems:
 - Neustarten
 - Neu aufsetzen
 - Daten entfernen

3.4.2.3 REST- Schnittstelle

Wie bereits unter 3.3 Konzept erwähnt, soll die Lösung in Java und JavaScript realisiert werden. Da die Funktionalität zur Überwachung einer Anwendung die Nutzerschnittstelle darstellt, soll diese alle nötigen API-Endpunkte bereitstellen, die der Nutzer benötigt.

Die API soll REST konform sein, dadurch ist sie weitestgehend selbsterklärend.

Die API soll unter dem Präfix /api/ erreichbar sein. Die folgenden Funktionen sind vorgesehen:

- /job:
 - GET: Listet alle gespeicherten Jobs
 - POST: Erstellt einen Job mit den mitgegebenen Daten
- /job/{id}:
 - GET: Zeigt den Job mit Id {id} an

⁵ Verfügbar unter: <https://getbootstrap.com/>

- PATCH: Editiert den Job anhand der mitgegebenen Daten
- DELETE: Löscht den Job
- /container:
 - GET: Listet alle Container, die auf der Docker-Instanz des Hostrechners vorhanden sind
 - POST: Erstellt einen neuen Container anhand der mitgegebenen Daten
- /analytic_data/{jobId}:
 - GET: Listet alle Analysepunkte, die zum Job mit Id {jobId} gesichert sind
- /analytic_data/{jobId}/{timestamp}:
 - GET: Listet die Daten des Analysepunktes mit dem Zeitstempel {timestamp} und dem Job mit Id {jobId}

PATCH	/job/{id}	Update a specific job
DELETE	/job/{id}	Delete a specific job
GET	/job/{id}	View job details
GET	/analytic_data/{job_id}/{timestamp}	Get specified analytic data by job id and timestamp
GET	/analytic_data/{job_id}	Lists analytic data for job
POST	/job	Creates a new job
GET	/job	Lists all jobs
POST	/container	Creates a new container
GET	/container	Lists all existing containers

ABBILDUNG 8 - ÜBERSICHT DER API-SCHNITTSTELLE⁶

3.4.2.4 Sichern der Daten der zu überwachenden Anwendung

Die Sicherung der Daten, die zu einer Anwendung einer Überwachungsaufgabe gehören, ist wichtig, falls eine Anwendung nicht mehr korrekt funktioniert. So kann die Anwendung anschließend mit den gleichen Parametern neu aufgesetzt werden.

Die Speicherung der Daten soll in einer Datenbank geschehen, die persistiert werden soll, damit beim Neustart der Anwendung keine Daten verloren gehen.

Da die zu entwickelnde Anwendung auch in einem Docker-Container laufen soll, ist es sinnvoll, dass die Datenbank ebenfalls in einem Container läuft. Um auch in diesem Fall die Komplexität beim Aufsetzen der Anwendung niedrig zu halten, soll die Datenbank in dem Image der Anwendung mitgeliefert werden, so dass nur ein Container erstellt werden muss.

⁶ Erstellt mit <https://editor.swagger.io/>

Eine direkt in eine Java-Anwendung implementierbare Lösung wäre eine H2 Datenbank, die über eine JDBC-Datenbankschnittstelle an die Anwendung angebunden ist. Dies bringt den Vorteil, dass keine weiteren Abhängigkeiten benötigt werden. Die Datenbank wird durch die Anwendung zur Verfügung gestellt.

3.4.2.5 Sichern von Daten über überwachte Anwendung zu Analyse Zwecken

Um die Analyse des Problems einer reparierten Anwendung zu vereinfachen, sollen verschiedene Daten über diese gesammelt werden. Diese Daten beinhalten Zeitstempel mit entsprechenden Protokollen, die die Anwendung kurz vor der Fehlfunktion ausgegeben hat, Daten, die von der Anwendung erstellt und genutzt wurden, sowie die Protokolle des fehlgeschlagenen Health-Checks.

Da bereits eine Datenbank (wie in Abschnitt 3.4.2.4 beschrieben) im Image der Anwendung mitgeliefert werden soll, kann die Lösung diese ebenfalls für die Sicherung der Daten über die Anwendung mitnutzen. Hierbei ist es vorgesehen die Pfade, in denen die Dateien gesichert werden, in der Datenbank abzulegen.

Mit Hilfe der "docker.sock" Schnittstelle sollen die Protokolle kurz nach der Fehlfunktion einer Anwendung abgerufen werden. Da von einer Anwendung genutzte Daten nur bei Persistieren außerhalb des Containers per Volume oder Mapping einen Neustart überdauern, sollen auch nur diese Daten für eine Analyse gesichert werden. Dies soll durch das Erstellen einer Kopie des Volumes oder des Ordners auf dem Dateisystem des Hostrechners geschehen. Diese Daten könnten sonst beim erneuten Starten des Containers verändert werden.

3.4.3 REPARIEREN EINER DEFEKTEN ANWENDUNG

Bei Feststellung einer nicht korrekt funktionierenden Anwendung müssen umgehend die definierten Schritte eingeleitet werden, damit die Anwendung ohne Verzögerung wieder korrekt funktioniert.

3.4.3.1 Schnittstelle zur Beauftragung einer Reparatur

Um von der Anwendungsüberwachung bei der Fehlfunktion einer Anwendung einen neuen Auftrag zur Reparatur dieser zu erhalten, soll eine Schnittstelle zur Verfügung gestellt werden, die diese annimmt. Da das Aufsetzen von mehreren Docker-Containern die Komplexität erhöhen würde, ist es weniger komplex die Funktionalitäten zusammen in einem Container zu realisieren.

Hierbei ist eine externe Schnittstelle nicht mehr notwendig und der Auftrag kann intern von der „Funktionalität zur Erkennung nicht korrekt funktionierender Anwendungen“ per Funktionsaufruf übergeben werden.

3.4.3.2 Möglichkeiten der Reparatur einer Anwendung

Die Reparatur einer Anwendung kann aus mehreren Schritten bestehen. Der Nutzer soll, die Schritte, die unternommen werden um die Anwendung zu reparieren, selbst bestimmen können.

Hierzu müssen bereits bei der Erstellung eines Überwachungsauftrags die entsprechenden Schritte angegeben werden. Nach jedem Schritt soll die Anwendung erneut auf ihre korrekte Funktion überprüft werden um diese nicht unnötig den nächsten Schritt durchlaufen zu lassen.

Die folgenden Schritte soll die Lösung bieten:

1. Der Container wird neu gestartet.
2. Die Anwendung wird in einem neuen Container gestartet. Hierzu wird der alte Container gestoppt. Unter neuem Namen wird ein neuer Container mit den Parametern des alten Containers gestartet.
3. Die Daten, die dem defekten Container via Mapping zugewiesen sind, werden gelöscht und die Anwendung wird neu aufgesetzt.

Das Ändern von Parametern kann dazu führen, dass die Anwendung zwar die Health-Checks besteht, aber von außerhalb des Systems nicht mehr verfügbar ist und soll deshalb zunächst nicht unterstützt werden.

3.4.3.3 Rückmeldung nach einer Reparatur

Sobald die Anwendungsüberwachung eine Anwendung an die Reparaturfunktion übergibt, soll ihr Status in der Datenbank dahingehend geändert werden. Wenn die Reparatur abgeschlossen ist, soll der Status wieder zurückgesetzt werden. Nach einer Reparatur, soll die Reparaturfunktion dies an die Anwendungsüberwachung melden, da diese anschließend eventuell einen Container überwachen muss, der einen anderen Namen hat.

Eine nicht erfolgreiche Reparatur soll mit in der Datenbank verzeichnet werden, sodass nicht erneut versucht wird die Anwendung zu reparieren.

3.4.3.4 Konzept der Datenbank

Wie in 3.4.2.4 erwähnt soll eine H2 Datenbank verwendet werden. Diese kann so konfiguriert werden, dass die Java-Anwendung sie zur Verfügung stellt. Per Mapping ist sie auch außerhalb des Docker-Containers, in dem die Anwendung später laufen soll, verfügbar. Hierzu muss bei Deklaration der Datenbank in der Java-Anwendung angegeben werden, dass die Datenbank ihre Daten in einer Datei sichert.

Die Datenbank soll nach den gegebenen benötigten Daten des Konzeptes erstellt werden. Konzeptionell sieht diese wie folgt aus.

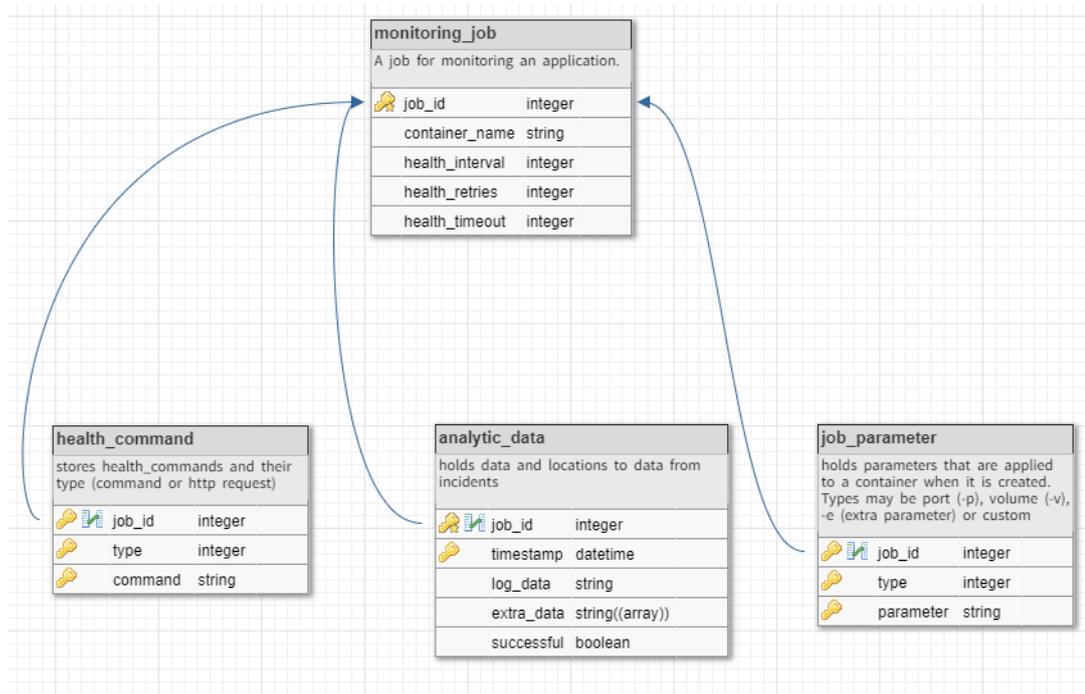


ABBILDUNG 9 - KONZEPTIONELLER AUFBAU DER DATENBANK⁷

3.4.4 AUFBAU BEISPIELHAFTER MICROSERVICES

Um die Lösung testen zu können, müssen beispielhafte Microservices entwickelt werden.

Hierzu sollen zwei Webserver konfiguriert werden, die per HTTP ansprechbar sind. Über eine Web-Oberfläche sollten weitere Server im Container gestartet und gestoppt werden können.

Ein Webserver soll in Python mit dem Framework Flask⁸ implementiert werden, da dies plattformunabhängig läuft und durch die Syntax, ähnlich Pseudo-Code, verständlich ist.

Der zweite Webserver soll in Java implementiert werden, um die Unabhängigkeit der Technologien zwischen Microservices zu demonstrieren. Dadurch ist es auch möglich, die Unabhängigkeit der benutzten Technologien bei zu überwachenden Anwendungen zu zeigen.

Für beide Webserver sollen entsprechende Docker-Images aus passenden Dockerfiles erstellt werden, die anschließend in der Docker-Hub-Repository zur Verfügung stehen.

⁷ Erstellt mit <http://dbdesigner.net>

⁸ Verfügbar unter: <http://flask.pocoo.org>

4. REALISIERUNG

Die gebaute Lösung besteht aus einer Anwendung, die als Docker-Container eingesetzt werden kann. Der Nutzer kann Container seiner lokalen Docker-Instanz per Internetbrowser zur Überwachung anmelden. Konfigurierbar sind die Health-Checks der Container, sowie die zu unternehmenden Schritte bei erkannter Fehlfunktion eines Containers.

4.1 AUFBAU BEISPIELHAFTER MICROSERVICES

Zur Überprüfung der entwickelten Lösung auf korrekte Funktion, wurden zwei beispielhafte Microservices entwickelt. Diese stellen jeweils einen Microservice dar, da sie eine einzelne gekapselte Funktion zur Verfügung stellen.

Beide Anwendungen bestehen aus einem Webserver, der unter Port 5000 einen simplen HTTP-Server bereitstellt. Über die definierten Endpunkte lassen sich weitere Server innerhalb des Containers starten. Jeder hinzugefügte Server erhält einen um 1 inkrementierten Port zugewiesen und bietet nur den Endpunkt „/“. Die Server werden in einer Liste gesichert und können so vom Nutzer wieder entfernt werden. Der Server unter Port 5000 bietet die folgenden Endpunkte, um die weiteren Server zu verwalten:

/add: Fügt einen Server hinzu und startet ihn.
/kill: Stoppt und entfernt den zuletzt hinzugefügten Server.
/static/static.html: Eine HTML Seite, die vom Dateisystem bereitgestellt wird.

Mit Hilfe der Anwendungen können so Health-Checks die den HTTP-Server auf Verfügbarkeit oder erwarteten Statuscode hin prüfen, genutzt werden. Es kann auch überprüft werden, ob eine Datei im Dateisystem verfügbar ist, beziehungsweise das Mapping eines Ordners vom Hostrechner in den Container erfolgreich war. Bei Start einer Anwendung wird automatisch ein weiterer Server auf Port 5001 gestartet. Dies dient dazu mehrere Health-Checks direkt auf einem Container der Anwendung zu ermöglichen.

Die Anwendungen sind in Java und Python geschrieben. Die mit Python geschriebene Anwendung stellt den Webserver mit Hilfe von Flask zur Verfügung. Flask bietet in diesem Fall den Webserver und die Erstellung von Endpunkten auf diesem. Bei Aufruf eines Endpunktes wird eine Funktion ausgeführt.

Der im Java Development Kit (JDK) mitgelieferte HTTP-Server stellt den Webserver, der in Java geschriebenen Anwendung, bereit. Dieser bietet ebenfalls die Möglichkeit bei Aufruf eines Endpunktes eine Funktion auszuführen. In diesem Fall wird dies über eine Klasse erreicht, die das Interface `HttpHandler` implementiert und die Funktion „`handle()`“ überschreibt, die bei Aufruf des Endpunktes ausgeführt wird.

Beide Anwendungen sind als Docker-Images im Docker-Hub verfügbar⁹. Da diese als git-Repository öffentlich zur Verfügung stehen, wird über den Docker-Hub bei einer

⁹ Verfügbar unter: https://hub.docker.com/r/mxml/binocular_microservices/

Veränderung des Quellcodes automatisch das Image neu gebaut. Mit Hilfe von Tags lassen sich beide Anwendungen über ein Image zur Verfügung stellen.

Der Tag "python" bietet die Python Anwendung und lässt sich wie folgt einsetzen:

```
docker create -p 5000-5010:5000-5010 \  
-v /path/to/static:/usr/src/python-webapp/static \  
mxml/binocular_microservices:python
```

Code-Abbildung 1 – Erstellen eines Containers des Python-Microservices

Für die Java-Anwendung ist der Tag "java" konfiguriert. Die Java Anwendung lässt sich mit dem in Code-Abbildung 2 beschriebenen Kommando starten.

```
docker create -p 5000-5010:5000-5010 \  
-v /path/to/static:/usr/src/java-webapp/static \  
mxml/binocular_microservices:java
```

Code-Abbildung 2 – Erstellen eines Containers des Java-Microservices

Bei beiden Befehlen muss lediglich der Pfad zum „static“ Ordner auf dem Hostrechner angepasst werden. Dadurch wird jeweils ein Container erstellt, der die Ports 5000 bis 5010 auf die gleichen Ports des Hostrechners freigibt. Bei der Deklaration des Images für die Container wird jeweils der entsprechende Tag verwendet.

4.2 AUFBAU DER ANWENDUNG

Die entwickelte Anwendung zur Überwachung und Reparatur von defekten Containern besteht aus zwei Teilen, die von einem Webserver zur Verfügung gestellt werden. Ein Webserver stellt dem Nutzer eine Benutzeroberfläche zur Verfügung. Im Hintergrund läuft eine Java Anwendung, die die Eingaben der Benutzeroberfläche über eine REST konforme Schnittstelle entgegennimmt, verarbeitet, sowie Überwachung und Reparatur der Anwendungen übernimmt.

Die Anwendung basiert grundlegend auf dem Konzept und der beschriebenen Datenstruktur und Schnittstellen. Während der Entwicklung sind jedoch kleinere Modifikationen in das Projekt eingeflossen, die aus erkannten Problematiken und Verbesserungen erstanden sind.

4.2.1 REALISIERUNG DER DATENBANK

Wie vorgesehen werden die Daten, die von der Lösung benötigt werden, in einer H2 Datenbank gesichert. Diese ist zusammen mit JPA (Java Persistence API) und Hibernate eingesetzt. Diese beiden Technologien bieten vereint die Funktion, automatisch das Schema der Datenbank aus dem bestehenden Code zu erstellen. Dadurch vereinfachen sie die

spätere Sicherung von Daten in der Datenbank. Dies ist unter „4.2.3 Aufbau der Spring Anwendung“ tiefergehend beschrieben.

Bei der Erstellung der Anwendung hat sich während der Entwicklungsphase, zusammen mit den Funktionen, auch die Struktur der Datenbank verändert, welche in Abbildung 10 visualisiert ist.

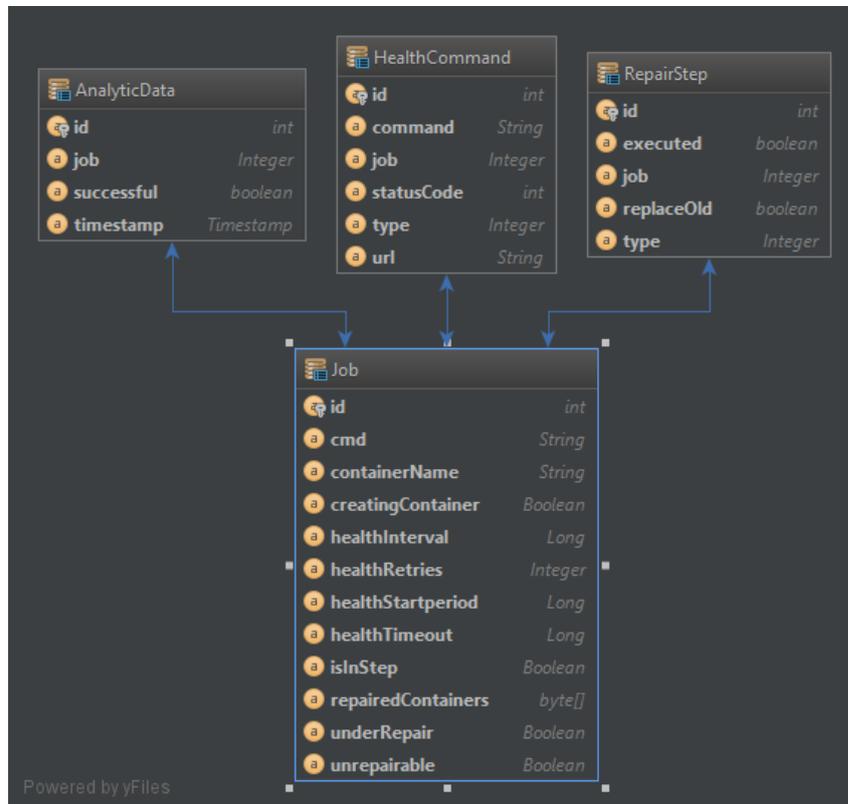


ABBILDUNG 10 - SCHEMA DER DATENBANK¹⁰

Ein Job ist ein Grundobjekt, welches für jede zu überwachende Anwendung erstellt wird. Dieses enthält alle Objekte der anderen Klassen, die in der Datenbank persistiert werden.

Jedes Objekt erhält zunächst eine eindeutige ID, die der späteren Identifikation bei REST-Abfragen und Bearbeitungen dient.

Ein Objekt des Typs Job hat die folgenden Attribute:

- Vom Nutzer benötigte Attribute:
 - containerName: Der Name des Containers, in dem die zu überwachende Anwendung läuft. Das ist nicht der primäre Schlüssel, da der Name sich bei einer Reparatur ändern kann.
- Vom Nutzer optional angebbare Attribute:
 - healthInterval: Das Intervall, mit dem ein Health-Check eines Containers ausgeführt werden soll, standardmäßig auf 15 Sekunden festgelegt.

¹⁰ Erstellt mit IntelliJ Idea

- healthRetries: Die Anzahl der fehlgeschlagenen Health-Checks bevor ein Container als „unhealthy“ angesehen wird, standardmäßig auf 3 Versuche festgelegt.
- healthStartperiod: Die Zeit, die vergehen soll, bevor ein erster Health-Check ausgeführt wird, standardmäßig auf 8 Sekunden festgelegt.
- healthTimeout: Die Zeit, die ein Health-Check maximal benötigen darf, bevor er als fehlgeschlagen angesehen wird, standardmäßig auf 2 Sekunden festgelegt.
- Von der Anwendung gesetzte Attribute
 - id: Ein eindeutiger Identifikationsschlüssel, der dem Objekt automatisch zugewiesen wird.
 - cmd: Die Art des auszuführenden Health-Checks („CMD“ oder „CMD-Shell“). Diese ist standardmäßig auf „CMD-Shell“ gesetzt, wird aber, sollte bereits ein Health-Check der Art „CMD“ vorhanden sein, geändert.
 - creatingContainer: Ein von der Anwendung genutzter Wert, der aussagt, ob ein Container momentan neu erstellt wird, um währenddessen die Health-Checks auszusetzen.
 - isInStep: Dieser Wert wird genutzt um einen Job, der sich momentan in einem Schritt der Reparatur befindet, zu erkennen, um zunächst keine weiteren Schritte zur selben Zeit zu beginnen.
 - repairedContainers: Eine Liste der Namen aller Container, die durch die Reparaturfunktion bearbeitet wurden.
 - underRepair: Ein Wert, der aussagt, dass sich der Job momentan in der Reparatur befindet.
 - unrepairable: Dieser Wert beschreibt, ob ein Container als nicht reparierbar angesehen wird. Dies geschieht, falls ein Job, alle Reparaturschritte durchlaufen hat, ohne anschließend seine Health-Checks zu bestehen.

Alle weiteren Objekte erhalten ebenfalls einen eindeutigen Identifikationsschlüssel und stehen in einer ManyToOne Beziehung mit einem Objekt vom Typen Job. Das bedeutet, dass ein Job-Objekt mehrere Objekte der anderen Klasse zugewiesen haben kann, die Objekte der anderen Klassen aber jeweils nur eines des Typen Job. Diese Objekte werden nicht benötigt und können auch später einem Job hinzugefügt werden.

Ein AnalyticData-Objekt wird erstellt, wenn für einen Job eine Reparatur eingeleitet wird. Darauf basierend wird das Attribut „timestamp“ gesetzt, welches dem Zeitstempel des Vorfalles entspricht. Nach Abschluss der Reparatur wird entsprechend des Erfolges der Reparatur das Attribut „successful“ gesetzt.

Bei der Erstellung eines Jobs lassen sich sowohl Health-Checks, als auch Schritte zur Reparatur der Anwendung angeben. Diese werden als Objekte der Klassen HealthCommand und RepairStep gesichert.

Bei der Definition eines Health-Checks wird ein HealthCommand-Objekt dem entsprechenden Job zugewiesen. Dabei lässt sich zwischen der Erstellung eines Health-Checks entweder mit einem Kommando oder mit der Überprüfung einer URL auf einen

gewünschten Statuscode hin wählen. Diese Entscheidung wird im „type“-Attribut gespeichert und entsprechend wird das „command“-Attribut mit dem Kommando versehen oder das „url“- und „statusCode“-Attribut werden gesetzt.

Um die Vorgehensweise bei einer Reparatur einer Anwendung bestimmen zu können, dienen Objekte der RepairStep Klasse. Diese beschreiben jeweils einen Schritt der bei der Reparatur eines Jobs durchgeführt werden soll. Es lassen sich drei verschiedene Typen von Reparatureschritten wählen, die beliebig aneinandergereiht werden können. Diese werden unter 4.2.5 „Aufbau der Funktion zum Reparieren einer nicht korrekt funktionierenden Anwendung“ behandelt. Mit Setzen des Wertes „replaceOld“, lässt sich festlegen, ob der Container des Jobs bei der Reparatur überschrieben oder behalten werden soll. Um keine Schritte mehrfach auszuführen, wird mit dem Wert „executed“ festgehalten, welcher Schritt bereits ausgeführt wurde.

4.2.2 AUFBAU DER REST-API

Die im Konzept beschriebene REST-API ist während der Entwicklung des Projektes verändert worden. Sie wurde um die Möglichkeit, einen Eintrag der analytischen Daten zu löschen, erweitert. Die zunächst bestehende Idee, Container direkt über die Nutzeroberfläche erstellen zu können, wurde fallen gelassen und somit wurde auch der Endpunkt zur Erstellung eines Containers entfernt. Die Übersicht [Abbildung 11] zeigt alle Endpunkte, die die Anwendung zur Verfügung stellt.

container-controller Container Controller		∨
GET	/api/container	Lists all running containers
analytics-controller Analytics Controller		∨
GET	/api/analytics/{id}	Lists analytic entries for job with id
GET	/api/analytics/{id}/{dataid}	Returns analytic entry for job with id and timestamp
DELETE	/api/analytics/{id}/{dataid}	Deletes analytic entry for job with id and timestamp
job-controller Job Controller		∨
GET	/api/job/	Lists all jobs
POST	/api/job/	Creates a new job
GET	/api/job/{id}	Returns job with id
DELETE	/api/job/{id}	Deletes job with id
PATCH	/api/job/{id}	patchJobById

ABBILDUNG 11 - ÜBERSICHT DER REST-API¹¹

¹¹ Erstellt mit <https://editor.swagger.io/>

4.2.3 AUFBAU DER SPRING ANWENDUNG

Die eigentliche Logik der Anwendung ist in Java entwickelt und wird mit dem Framework Spring Boot¹² betrieben. Spring vereinfacht bei der Entwicklung von Webanwendungen deren initiale Einrichtung und bietet nützliche Funktionen in Verbindungen mit anderen Java-Frameworks, die im Folgenden erläutert werden.

Der Aufbau der Anwendung [Abbildung 12] ist entsprechend der von Spring gelieferten Komponenten unterteilt in Controller, Services und datenbezogene Klassen.

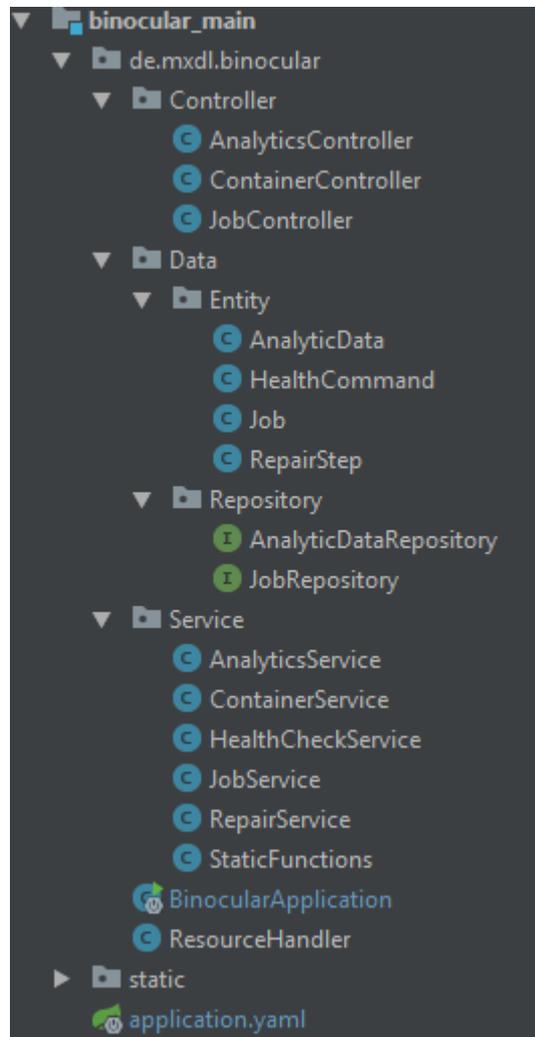


ABBILDUNG 12 - AUFBAU DER SPRING ANWENDUNG

4.2.3.1 Controller

Ein Controller in einer Spring-Anwendung ist eine Klasse, die eine Schnittstelle nach außen darstellt und Endpunkte definiert, die über HTTP angesprochen werden können. Alle Klassen des Projektes, die als Controller eingeordnet werden, sind in dem Package „Controller“ vereint [Abbildung 12]. Ein Controller sollte keine Logik beinhalten, sondern einen Service aufrufen, der die Logik implementiert.

¹² Verfügbar unter: <https://projects.spring.io/spring-boot/>

Beispielhaft hierfür ist ein Ausschnitt der Klasse `JobController.java` erläutert. Bei der Definition der Klasse [Code-Abbildung 3] werden zunächst die Spring spezifischen Java-Annotationen aufgerufen. Diese weisen der Klasse bestimmte Metadaten zu und verfügen immer über das Präfix `@`.

Mit der Annotation „`@RestController`“ wird Spring mitgeteilt, dass dies eine Klasse ist, die einen Controller darstellt, der REST konform operiert und dem Nutzer zu jedem Aufruf eines Endpunktes eine Antwort im „Http-body“ liefert. Durch diese Annotation wird die Klasse beim Bauen der Anwendung automatisch von Spring erkannt und die entsprechenden Endpunkte freigegeben. Um jedem weiteren Endpunkt der Klasse ein Präfix in der URL zuzuweisen, ohne diesen bei jeder Definition eines Endpunktes angeben zu müssen, wird die „`@RequestMapping`“ Annotation genutzt. Wenn diese als Klassenannotation eingesetzt wird, wird die angegebene URL, als Präfix aller Endpunkte der Klasse genutzt. Die Annotationen der Controller Klassen sind identisch und unterscheiden sich nur in ihrem Endpunkt und Namen.

```
@RestController
@RequestMapping("/api/job")
public class JobController {
```

Code-Abbildung 3 - Deklaration der Job Controller Klasse

Bei der Erstellung von Endpunkten innerhalb der Klasse [Code-Abbildung 4] werden diese als Annotation über der Funktion beschrieben, die bei Aufruf dieses Endpunktes ausgeführt wird. Hierbei lassen sich die URL, sowie die HTTP-Methode festlegen, die der Endpunkt unterstützen soll. Es lassen sich für einen Endpunkt mehrere Funktionen festlegen, sofern sie jeweils eine andere HTTP-Methode unterstützen. [vgl.: Spring 2017, RV 2016]

```
@RequestMapping(value = {"", "/"}, method = RequestMethod.GET)
public ResponseEntity getJobs() {
    return jobService.getJobs();
}
```

Code-Abbildung 4 - Deklaration eines Endpunktes

4.2.3.2 Services

Ein Service innerhalb einer Spring Anwendung ist dazu gedacht die Logik aus einer Controller Klasse zu implementieren. Dieser wird vor der Klassendefinition mit der Annotation „`@Service`“ versehen. Von einem Service sollte es stets nur ein Objekt geben, welches zustandslos operiert. Die Definition eines solchen Objektes geschieht über die Annotation „`@Autowired`“, welche Spring beim Bauen der Anwendung erkennt. So kann Spring ein einziges Objekt der Klasse erstellen und dieses bei jeder Deklaration mit der „`@Autowired`“-Annotation einbinden. [vgl.: Spring 2017, RV 2016]

4.2.3.3 Datenbezogene Klassen

Datenbezogene Klassen beschreiben in diesem Fall alle Klassen, die beim Aufbau der Anwendung in das Package „Data“ eingeordnet wurden. Hierbei lässt sich wiederum zwischen „Entity“, also Entitäten, und „Repository“, also Repositorien, unterscheiden. Eine Entität ist eine Datenstruktur, die mit Hilfe von Spring und JPA persistiert werden kann. Als Beispiel für den Aufbau einer Entitätsklasse dient die Job-Klasse, die die Daten eines Jobs sichert. Vor der Klassendeklaration [Code-Abbildung 5] erhält die Klasse das Merkmal „@Entity“. JPA legt solche Klassen automatisch bei Start der Anwendung als Tabelle in der Datenbank an und kann so Objekte der Klassen in der Datenbank persistieren. Zur Anpassung der Darstellung der Klasse in der Datenbank wird mit der Annotation „@Table“, die Tabelle in der Datenbank anders als der Klassenname benannt und ein Index für die Tabelle festgelegt. Um Probleme bei der Abfrage von Objekten aus der Datenbank zu umgehen, legt die Annotation „@Cacheable“ fest, dass die Daten bei jeder Abfrage erneut geladen werden sollen.

```
@Entity
@Cacheable(false)
@Table(name = "job", indexes = @Index(columnList = "id"))
public class Job {
```

Code-Abbildung 5 - Deklaration der Entitätsklasse Job

Alle Attribute der Klasse [Code-Abbildung 6] sind ebenfalls automatisch in der Tabelle der Datenbank angelegt. Hierbei lässt sich mit Annotation wieder die Tabelle anpassen. Mit der Annotation „@Id“ lässt sich der Schlüssel der Tabelle festlegen. Mindestens ein Attribut der Klasse muss mit „@Id“ annotiert sein. Durch die Annotation mehrere Attribute kann ein Schlüssel aus mehreren Attributen (Composite-Key) festgelegt werden. In diesem Fall handelt es sich um einen einzelnen Schlüssel, der automatisch von JPA erstellt und für jedes neue Objekt der Klasse erhöht wird. Dies ist mit der Annotation „@GeneratedValue“ beschrieben. Ähnlich wie bei der Anpassung des Tabellennamens, lassen sich die Spaltennamen der einzelnen Attribute über die „@Column“ Annotation anpassen. [vgl.: Spring 2017, RV 2016]

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "id", unique = true)
private int id;

@Column(name = "container_name")
private String containerName;
```

Code-Abbildung 6 - Deklaration von Attributen

4.2.4 AUFBAU DER FUNKTION ZUR ÜBERWACHUNG DER KORREKTEN FUNKTIONSWEISE EINER ANWENDUNG

Die Überwachung der Container besteht aus der Funktion, die es ermöglicht eine Anwendung überwachbar zu machen und der eigentlichen Überwachung und Weiterleitung defekter Anwendungen an die Reparaturfunktion. Mit Hilfe des Frameworks „Docker Client“¹³, welches unter der Apache 2.0 Lizenz von Spotify zur Verfügung gestellt wird, ist es möglich mit dem „docker.sock“-Socket zu kommunizieren. Das Framework erlaubt es über ein Objekt des Typ `DockerClient` mit der verbundenen Docker-Instanz zu arbeiten.

Bevor eine Anwendung in Form eines Jobs auf ihre korrekte Funktion hin geprüft werden kann, muss der entsprechende Job erstellt werden. Anschließend, analysiert die Lösung zunächst den aktuell laufenden Container der Anwendung und speichert die Konfiguration dessen zwischen. Ursprünglich war geplant diese Parameter mit in der Datenbank zu persistieren. Da diese jedoch vom Container ausgelesen werden können und somit immer aktuell sind, ist das nicht notwendig. Der Container wird anschließend gestoppt, damit ein temporärer Container mit eventuell identischen Port-Mappings gestartet werden kann. Unter einem neuen Namen wird anschließend anhand der kopierten Konfiguration und dem vom Nutzer gegebenen Daten, der temporäre Container erstellt. Der wichtigste Schritt ist hierbei das Zusammenführen der bereits bestehenden Health-Checks des Containers mit den vom Nutzer hinzugefügten Health-Checks in Form von `HealthCommand`-Objekten. Diese werden anschließend zu einem Health-Check vereint, entsprechend der von Docker gestellten Vorgaben angepasst und dem Container hinzugefügt.

Dieser wird gestartet und daraufhin überwacht, ob der Startvorgang erfolgreich ist. Sollte es dabei zu Problemen kommen oder sollte der Container nicht starten, wird der ursprüngliche Container wiedereingesetzt und der temporäre Container gelöscht. Startet der Container problemlos, so werden er und der ursprüngliche Container gelöscht. Anschließend wird der eigentliche Container, der dem Job zugeordnet ist, unter dem Namen und mit der Konfiguration des ursprünglichen Containers und der vom Nutzer angegebenen Health-Checks erstellt und gestartet. Das verhindert, dass bei der Erstellung eines Jobs nicht der ursprünglich korrekt funktionierende Container durch Falscheingaben des Nutzers oder bei Problemen in der Konfiguration durch einen nicht korrekt funktionierenden Container ersetzt wird.

Diese Vorgehensweise wird ebenfalls bei der Veränderung eines Jobs durchgeführt. Hierbei werden jedoch nicht alle Konfigurationsdaten des alten Containers übernommen, um beispielsweise im Job gelöschte Health-Checks nicht zu kopieren. Da die Health-Checks des ursprünglichen Containers bei der erstmaligen Erstellung des Job-Containers als `HealthCommand`-Objekte erstellt und dem Job zugewiesen werden, kommt es zu keinem Verlust von Health-Checks.

Um defekte Container zu erkennen, wird der von Docker zur Verfügung gestellte `HealthState` genutzt. Dazu wird in einem Abstand von fünf Sekunden asynchron der Status aller

¹³ Verfügbar unter: <https://github.com/spotify/docker-client>

Container abgefragt, für die ein Job existiert. Hierbei wird überprüft, ob sich ein Job schon in der Reparatur befindet oder momentan der Container für diesen Job erstellt wird. Ist dies beides nicht der Fall, wird der Job dem Reparatur-Service gemeldet. Sollte sich der Job in der Reparatur befinden oder als nicht reparierbar eingestuft worden sein, so wird der Status im Reparatur-Service aktualisiert. Diese Überprüfung ist asynchron. Das ist notwendig, um bei der Überprüfung keine eventuell laufenden Schritte zu unterbrechen. Da es zu Problemen in Zusammenhang mit nicht aktuellen Daten innerhalb der Datenbank kommen kann, werden in der Reparaturfunktion bestimmte Attribute erneut abgefragt, um keine inkorrekten, alten Daten zu nutzen. Denn sobald ein Objekt mit JPA gesichert oder verändert wird, wird dieses erst persistiert, wenn die entsprechende Funktion ausgeführt wurde.

4.2.5 AUFBAU DER FUNKTION ZUM REPARIEREN EINER NICHT KORREKT FUNKTIONIERENDEN ANWENDUNG

Erkennt die Funktion zur Überwachung einen Container eines Jobs als nicht korrekt funktionierend, meldet sie dies dem Service zur Reparatur. Dieser startet anschließend den Reparaturvorgang, sofern er momentan noch keine Reparatur für diesen Job unternimmt oder der Container den „exit-code 137“ besitzt, den der Container erhält sobald der Nutzer ihn stoppt.

Bevor die Reparatur beginnt, werden zunächst die Daten des Vorfalls gesammelt, als `AnalyticData`-Objekt gesichert und dem Job zugeordnet. Diese Daten beinhalten zunächst das Protokoll der Anwendung, sowie das Protokoll der Health-Checks. Im Ordner `/data` wird für den Vorfall ein Ordner angelegt, der auf der Struktur `/data/{jobid}/{timestamp}` basiert. Darin werden die Daten als Textdateien gesichert.

Anschließend wird für diesen Job das Attribut „underRepair“ gesetzt, so dass keine weitere Reparatur begonnen wird. Basierend auf den, dem Job zugehörigen, `RepairStep`-Objekten wird daraufhin die Reparatur ausgeführt. Hierbei bekommt der Job, falls der Nutzer keine Schritte definiert hat, eine Standardroutine zugeordnet, um die Reparatur zu ermöglichen.

Alle vorhandenen Schritte zur Reparatur werden bis zum ersten Schritt durchgegangen, der noch nicht ausgeführt wurde. Entsprechend dem Typen des Schrittes wird anschließend der Container behandelt. Es gibt drei Typen, die folgende Aktionen durchführen:

Restart: Der Container wird neu gestartet.

Reset: Die Daten, die dem Container per Mapping zugeordnet sind, werden in das Verzeichnis des aktuellen `AnalyticData`-Objektes gesichert. Ist dies erfolgreich, werden die Daten entfernt und der Container neu aufgesetzt. Durch das Attribut „replaceOld“ kann bestimmt werden, ob ein neuer Container erstellt oder der alte Container überschrieben werden soll.

Replace: Der Container wird neu aufgesetzt. Durch das Attribut „replaceOld“ kann bestimmt werden, ob ein neuer Container erstellt oder der alte Container überschrieben werden soll.

Ist ein Container durch das Attribut „replaceOld“ erstellt worden, so erhält dieser den Namen des ursprünglichen Containers des Jobs mit einem angehängten Index. Dieser Index erhöht sich bei einer erneuten Reparatur, nicht jedoch bei einem weiteren Schritt der laufenden Reparatur. Der Name des alten Containers wird im „repairedContainers“ Attribut gesichert, um dies verfolgen zu können.

Sobald ein Job als „underRepair“ markiert ist, werden alle Health-State Meldungen von einer separaten Funktion empfangen, um eine doppelte Bearbeitung des Jobs zu vermeiden.

Anhand dieser Meldungen ist es dem Reparatur-Service möglich den Status der Reparatur zu erkennen. Ist ein Container repariert worden, hat er zunächst den Status „restarting“. Hier gilt die Reparatur noch nicht als erfolgreich abgeschlossen. Erst wenn der Status „healthy“ lautet, wird der Container wieder als korrekt funktionierend angesehen, die Reparatur als erfolgreich vermerkt und der Ausführungsstatus der Schritte zurückgesetzt. Sollte der Container widererwarten als „unhealthy“ gemeldet werden, so beginnt der nächste Schritt in der Reparatur. Sollten alle Schritte ausgeführt worden sein, ohne dass der Container erfolgreich repariert wurde, wird er als „unrepairable“ markiert und die Reparatur als nicht erfolgreich vermerkt.

Der Reparatur-Service nimmt dennoch weitere Meldungen zum Status des Containers entgegen, um bei einem Wechsel des Status aufgrund einer unvorhergesehenen Änderung reagieren zu können. Dies dient dazu, den Container, sollte er wieder funktionieren, im Problemfall erneut reparieren zu können.

4.2.6 AUFBAU DER BENUTZEROBERFLÄCHE

Die Benutzeroberfläche ist eine AngularJS-Anwendung¹⁴, die über den Webserver der Spring-Anwendung bereitgestellt wird. Das Design basiert auf der Bootstrap-Bibliothek.

AngularJS ist ein JavaScript-Framework, welches die Entwicklung einer dynamischen Webanwendung vereinfacht. Dank der Erweiterbarkeit von AngularJS sind diverse Frameworks verfügbar. Bootstrap ist eine quelloffene Bibliothek zur Gestaltung von Webseiten und ist als speziell angepasste Version für die Nutzung mit AngularJS verfügbar. Durch die von der Spring-Anwendung zur Verfügung gestellte REST-API, kommuniziert die Benutzeroberfläche mit der eigentlichen Anwendung. Dies geschieht mit dem AngularJS Modul „http“, welches das Senden von HTTP-Anfragen und Erhalten der entsprechenden Antworten ermöglicht. Auf einer Hauptseite wird dem Nutzer beim Öffnen der Webseite zunächst eine Liste

¹⁴ <https://angularjs.org>

aller erstellten Jobs angezeigt. Diese Liste wird als JSON abgerufen und dynamisch erstellt [Code-Abbildung 7].

In diesem Fall wird, für jeden Job in der Liste mit dem Namen „jobs“, eine Zeile in der Tabelle mit dem HTML-Tag „tr“ erstellt. Dies ist durch die AngularJS-Anweisung „ng-repeat“ definiert. Innerhalb der Zeile kann anschließend auf das entsprechende Objekt zugegriffen und Attribute ausgelesen und angezeigt werden. Daten, die hierbei angezeigt werden sollen, werden in geschweifte Klammern gesetzt.

```
<tr ng-repeat="job in jobs">
  [...]
  <td style="font-size: medium;">{{ job.id }}</td>
  [...]
</tr>
```

Code-Abbildung 7 - Dynamisches füllen einer Tabelle

Beim Erstellen oder Editieren eines Jobs kommt das quelloffene AngularJS-Modul „angular-schema-form“¹⁵ zur Verwendung. Dieses erlaubt mit Hilfe von JSON ein Formular zu erstellen, welches aus der Eingabe ein JSON-Objekt erstellt. Ebenfalls lässt sich das Formular mit einem bereits bestehendem JSON-Objekt befüllen. Da die REST-API die Job-Objekte als JSON zur Verfügung stellt und diese bei Änderungen auch nur als valide JSON-Objekte annimmt, muss lediglich das JSON des Formulars erstellt werden, welches wiederum korrekte Job-Objekte aus dem Formular zur Verfügung stellt.

Bei der Navigation auf der Benutzeroberfläche fällt auf, dass sich der Nutzer stets nur auf einer Seite befindet. Dies wird durch das AngularJS-Modul ngRoute bewerkstelligt. Dies erlaubt das Aufteilen der Seite in einzelne HTML-Dateien. In einer JavaScript-Datei werden die einzelnen Routen der Website definiert und festgelegt, welche HTML-Datei für welche Route angezeigt werden soll. Durch Controller, die die Datenverarbeitung einer AngularJS-Applikation übernehmen, lassen sich für jede Route eigene Funktionen und Daten festlegen. [Smith 2015, Google 2017]

Eine Beschreibung aller Funktionen der Benutzeroberfläche ist unter 6. Dokumentation zu finden.

4.3 EINSATZ DER ANWENDUNG

Die entwickelte Software, wird als Docker-Container ausgeliefert. Dieser beinhaltet alles, was die Anwendung zur korrekten Funktion benötigt. Auf dem Rechner, muss lediglich Docker installiert sein und der Nutzer den Docker-UNIX-Socket docker.sock per Mapping in den Container einbinden.

¹⁵ Verfügbar unter <https://github.com/json-schema-form/angular-schema-form>

Damit die Anwendung per Browser erreichbar ist, ist es bei der Erstellung eines Containers der Anwendung notwendig, dass der Nutzer den Port freigibt

Um die Daten der Analysefunktion zu persistieren muss der Pfad „/data“ des Containers in das Dateisystem des Hostrechners gemappt werden. Darin befinden sich die Daten die während einer Reparatur gesammelt wurden.

Sollte das Sichern von Daten bestehender Container bei deren Reparatur gewünscht sein, müssen die Verzeichnisse, die diese nutzen, ebenfalls per Mapping in den Container eingebunden werden.

5. BEWERTUNG DER REALISIERUNG

Um den Erfolg der Realisierung bewerten zu können, muss diese anhand der gestellten Anforderungen betrachtet werden. Sofern diese erfüllt sind, lässt sich die Realisierung als Erfolg betrachten.

5.1 BEWERTUNG BASIEREND AUF DEN GESTELLTEN ANFORDERUNGEN

Die gestellten Anforderungen sind in die Funktionen, die die Anwendung erfüllen soll, unterteilt.

Die funktionalen Anforderungen an die Funktionsüberwachung (Abschnitt 3.2.1.1) beschreiben, was die realisierte Lösung dem Nutzer für Möglichkeiten zur Überwachung von Anwendungen bieten soll.

Es ist dem Nutzer möglich zu überwachende Anwendungen und Bedingungen der Überwachung anzugeben, diese zu bearbeiten und zu löschen. Die Anwendungen werden anschließend entsprechend der Vorgaben überwacht und im Falle eines Problems erkannt. Sofern die Anwendung die Überprüfung nicht besteht und dies nicht auf die nicht korrekte Funktion zurückzuführen ist, wird kein Schritt zur Reparatur unternommen. Dies ist bei Neustart der Anwendung oder gewolltem Anhalten der Anwendung durch den Nutzer der Fall. Bei Nichtbestehen einer Überprüfung, weil eine Funktion der Anwendung eingeschränkt ist, wird dies protokolliert und die Anwendung wird zur Reparatur freigegeben.

Die nicht-funktionalen Anforderungen an die Funktionsüberwachung beschreiben, wie die realisierte Lösung die funktionalen Anforderungen erfüllen muss. Sofern das Intervall in dem die Anwendung überprüft werden soll nicht angegeben ist, wird dies auf 15 Sekunden festgelegt und nicht vom System abhängig gemacht, wie in den Anforderungen beschrieben. Da ein Bewerten der Systemleistung und der Kapazitäten der Anwendung sehr komplex ist, kann diese Bewertung vom Nutzer am genauesten geschehen. Schlimmstenfalls vergehen so nach der Entstehung eines Problems drei mal 15 Sekunden (Intervall mal Versuche), bevor es erkannt wird. Das bedeutet im Falle eines Problems einen prozentualen Ausfall von etwa 0,06% am Tag durch das nicht angepasste Intervall. Durch die Anzahl der Health-Checks die nacheinander fehlschlagen dürfen, bevor eine Anwendung repariert wird, soll keine Anwendung fälschlicherweise repariert werden.

Auf der Übersichtsseite der Benutzeroberfläche werden alle Funktionen vereint, die die Anwendung zur Verfügung stellt. Dem Nutzer wird so der Einstieg in die Anwendung vereinfacht. Die in Abschnitt 6. geschriebene Dokumentation erläutert die Anwendung weitergehend, falls Funktionen nicht allein durch die Benutzeroberfläche verständlich sind.

Es sind alle Anforderungen an die Funktion zur Überwachung einer Anwendung durch die realisierte Lösung erfüllt und in manchen Teilen erweitert.

Die Anforderungen zur Reparatur einer Anwendung legen fest, wie die Anwendung im Falle eines Problems behandelt werden soll. So ist es, wie in den Anforderungen beschrieben, bereits bei der Erstellung eines Jobs möglich, die Schritte einer Reparatur

anzugeben. Diese werden durchlaufen und die Anwendung wird nach jedem Schritt erneut auf ihre korrekte Funktion überprüft. Die Anforderung den Nutzer bei einer fehlgeschlagenen Reparatur zu benachrichtigen, ist nicht erfüllt worden, da dies mehr Konfigurationsaufwand für den Nutzer bedeutet und so die Hürde für den Einstieg in die Anwendung erhöht hätte. Dem Nutzer werden stattdessen bei Aufruf der Benutzeroberfläche direkt auf der Übersichtsseite die Status der Anwendungen angezeigt.

Die nicht-funktionalen Anforderungen sind bereits teilweise durch die Funktion zur Überwachung der Anwendungen erfüllt. Da diese lediglich Anwendungen, die defekt sind, an die Reparaturfunktion meldet und diese selbst keine Anwendung überprüft, werden nur gemeldete Anwendungen repariert. Alle Daten einer Anwendung werden stets gesichert, beziehungsweise nicht verändert, so dass keine Daten verloren gehen. Durch das Kopieren der Parameter der ursprünglichen Anwendung werden diese alle auf einen eventuell neuen Container der Anwendung übertragen, so dass diese Anforderung erfüllt ist. Durch die Nutzung von mehreren Threads bei der Abfrage der Status der Anwendungen werden diese im Fall eines Problems in diesem separaten Thread bearbeitet, so dass mehrere Reparaturen parallel stattfinden können. Dies erfüllt die Anforderung, dass möglichst wenig Zeit vergeht, in der die Anwendung nicht erreichbar ist.

Im Vergleich zu den bestehenden Lösungen, sind die genannten Schwachpunkte in der entwickelten Lösung eingeflossen. Dem Nutzer kann so Anwendungen auf lokalen Docker-Instanzen überwachen. Die Benutzeroberfläche erleichtert ihm, neben der Dokumentation, den Einstieg in die Nutzung der Lösung. Um nicht nur die Reparatur zu ermöglichen, sondern ebenfalls das ursprüngliche Problem erkennen zu können dient die Sicherung von Daten, die die bestehenden Lösungen nicht, beziehungsweise nur eingeschränkt, bieten.

Die gestellten Anforderungen werden bis auf die Benachrichtigung des Nutzers erfüllt, sodass die Realisierung der Lösung zur Überwachung und Reparatur von Anwendungen als Erfolg anzusehen ist.

6. DOKUMENTATION DER SOFTWARE

Diese Dokumentation dient als Benutzerhandbuch für die entwickelte Anwendung, im folgenden Binocular genannt. Binocular ist das englische Wort für Fernglas. Es steht in diesem Fall symbolisch für die Überwachung der laufenden Container.

Binocular wird als Image im Docker Hub zur Verfügung gestellt. Es trägt den einzigartigen Identifikationsschlüssel mxml/binocular und ist unter <https://hub.docker.com/r/mxml/binocular/> verfügbar.

Um die Anwendung als Container laufen zu lassen sind mindestens folgende Argumente notwendig:

```
docker run -d \  
    -v /var/run/docker.sock:/var/run/docker.sock \  
    -p 8080:8080 \  
    mxml/binocular
```

Code-Abbildung 8 – Erstellen des Containers der Binocular-Anwendung

Das Einbinden des „docker.sock“-Sockets in den Container ist notwendig, damit die Anwendung mit der Docker-Instanz kommunizieren kann. Durch das Freigeben des Ports 8080 ist die Anwendung auch vom Hostrechner erreichbar. Hierbei ist zu beachten, dass die Anwendung, nicht durch Zugriff von außen gesichert ist und daher nur in einem Netzwerk freigegeben werden sollte, welches nicht von Dritten verwendet wird.

Zum Persistieren der Daten, die bei einer Reparatur eines Jobs gesammelt werden, ist es zudem notwendig, den Ordner /data/ des Containers in einen Ordner des Hostrechners einzubinden.

Hierzu kann das Kommando um das folgende Mapping erweitert werden:

```
-v /path/to/data:/data/
```

Code-Abbildung 9 – Persistieren der Daten des Binocular-Containers

Der Pfad /path/to/data muss man hierbei mit dem Verzeichnis des Hostrechners ersetzen, in dem die Daten gesichert werden sollen.

Um später alle Funktionen nutzen zu können, ist es notwendig den Speicherort der von Containern genutzten Mappings mit dem gleichen Pfad wie auf dem Hostrechner einzubinden.

Als Beispiel hierfür lässt sich ein Container mit einem Mapping seines /config Ordners in den Ordner /docker/config des Hostrechners hinzuziehen. Soll der Container überwacht werden und sollen alle Funktionen verfügbar sein, so muss der Ordner /docker/config des Hostrechners in den Ordner /docker/config des Containers in dem die Binocular Anwendung läuft, eingebunden werden.

```
-v /docker/config:/docker/config
```

Code-Abbildung 10 – Einbinden der bestehenden Daten von Containern

Sobald die Anwendung aufgesetzt wurde und gestartet ist, kann diese über einen Browser über die IP-Adresse des Rechners und den freigegebenen Port erreicht werden. Falls die Anwendung auf dem lokalen Rechner läuft, wäre dies die Adresse `http://localhost:8080`.

Die folgenden Seiten stellt die Anwendung zur Verfügung:

- **Startseite (URL-Suffix: /!#/)**

Diese Seite [Abbildung 13] wird standardmäßig angezeigt, sobald der Nutzer die Anwendung aufruft. Sie zeigt alle angelegten Jobs und erlaubt das Interagieren mit diesen.



ABBILDUNG 13 - STARTSEITE DER BENUTZEROBERFLÄCHE

Die roten Kreise sind zur besseren Erläuterung der Ansicht eingefügt und nicht in der Benutzeroberfläche vorhanden. Entsprechend der Zahlen in den Kreisen sind folgende Funktionen gegeben:

1. Knopf, über den der Job im System gelöscht werden kann
2. Knopf, über den der Job editiert werden kann (unter **Job editieren** beschrieben)
3. Knopf, über den die Liste aller zu dem Job gehörenden Einträge zu den analytischen Daten angezeigt wird (unter **Anzeige der Daten zur Analyse** beschrieben)
4. Einzigartige Schlüsselnummer des Jobs
5. Der Status des Jobs, dargestellt über ein Icon. Hierbei gibt es drei Varianten [Abbildung 14], die den entsprechenden Status symbolisieren.
 - i. Ein grüner Daumen nach oben zeigt an, dass der Job ohne Probleme funktioniert.
 - ii. Der orangene Daumen nach unten symbolisiert, dass die Anwendung des Jobs sich momentan in der Reparatur befindet.
 - iii. Sind alle Reparaturversuche für einen Job fehlgeschlagen, wird ein roter Daumen nach unten als Status angezeigt.
6. Der aktuelle Name des Containers, der zum Job gehört.
7. Knopf, über den ein neuer Job zum System hinzugefügt werden kann (unter **Job hinzufügen** beschrieben)



ABBILDUNG 14 - STATUSSYMBOLE

- **Job hinzufügen (URL-Suffix: /!#/job)**

Diese Ansicht [Abbildung 15,16,17] erlaubt es einen Job hinzuzufügen. Die Seite wurde zum besseren Verständnis in dieser Beschreibung in mehrere Abbildungen aufgeteilt.

Edit job

Fields marked with * are required

*** Container to monitor**

Interval in which the container's health should be checked in seconds

Retries of a failed health-check before a repair is started

Timeout before a health-check is treated as failed in seconds

Time in seconds before the first health-check is executed

ABBILDUNG 15 - ERSTELLEN EINES JOBS

Abbildung 15 zeigt alle Attribute eines Jobs, die als einzelne Werte angegeben werden können.

Die roten Kreise sind zur Beschreibung der Felder eingefügt und nicht in der Benutzeroberfläche zu sehen.

Erläuterung der Abbildung:

1. Dieses Feld bietet eine Liste aller momentan auf der Docker-Instanz laufenden Container. Aus dieser Liste kann anschließend der Container mit der Anwendung, die überwacht werden soll, ausgewählt werden. Der Container in dem die Binoocular-Anwendung läuft ist nicht aufgeführt.

2. In diesem Feld wird angegeben in welchem Intervall der Container auf seine korrekte Funktion hin überprüft werden soll.
3. Mit diesem Eintrag wird festgelegt, wie oft ein Health-Check eines Containers fehlgeschlagen darf, bevor eine Reparatur unternommen wird.
4. Der in diesem Feld angegebene Wert wird genutzt, um festzulegen, wie lange ein Health-Check maximal dauern darf, bevor er als fehlgeschlagen angesehen wird.
5. In diesem Feld wird angegeben, wie lange mindestens gewartet werden soll, bevor der erste Health-Check ausgeführt wird

Health Checks

Type of check 6 ×

Command 6

Command to execute

7

Type of check 8 11 ×

HttpCall 8

URL to check

9

Expected status code

10 -

12 + Add

ABBILDUNG 16 - HINZUFÜGEN VON HEALTH-CHECKS

Im folgenden Abschnitt werden die Health-Checks angegeben, die zu den eventuell schon bestehenden Health-Checks des gewählten Containers hinzugefügt werden.

Erläuterung der Abbildung:

6. Auswahlmöglichkeit „Command“ als Typ eines Health-Checks. Erlaubt das Hinzufügen eines Kommandos, welches als Health-Check ausgeführt wird.
7. Feld, dessen Inhalt als Kommando des Health-Checks genutzt wird.
8. Die Auswahlmöglichkeit „HttpCall“ erlaubt es einen Health-Check hinzuzufügen, der eine URL auf einen Statuscode prüft.
9. Feld, in dem die URL angegeben wird, die geprüft werden soll.
10. In diesem Feld wird der Statuscode angegeben, auf den die URL geprüft werden soll.
11. Knopf zum Entfernen des entsprechenden Health-Checks
12. Knopf zum Hinzufügen eines weiteren Health-Checks

Steps that should be gone through when trying to repair the container

Type of repair ×

Restart 13 ▾

Type of repair ×

Replace 14 ▾

Replace the old container (otherwise it is kept for analytic 15 purposes)

Type of repair ×

Reset 16 ▾

Replace the old container (otherwise it is kept for analytic purposes)

+ Add

Submit 17

ABBILDUNG 17 - HINZUFÜGEN VON REPARATURSCHRITTEN

Dieser letzte Abschnitt erlaubt das Hinzufügen von Reparaturschritten zum Job. Das Hinzufügen und Löschen geschieht wie unter Punkt 11 und 12 beschrieben. Die Reparaturschritte werden entsprechend ihrer Angabe von oben nach unten durchgeführt.

Erläuterung der Abbildung:

13. Auswahlmöglichkeit für die Art eines Reparaturschrittes. In diesem Fall ist „Restart“ gewählt. Hierbei wird der Container neu gestartet, falls es zu einem Problem kommt.
14. Auswahlmöglichkeit „Replace“, die bei einem Problem den Container neu aufsetzt.
15. Kontrollkästchen, mit dem festgelegt wird, ob der neu aufgesetzte Container den alten Container überschreiben soll. Dies ist der Fall, wenn das Kästchen gewählt wurde.
16. Auswahlmöglichkeit „Reset“, bei der die Daten des Containers, die per Mapping zugewiesen sind, zunächst extern gesichert und dann entfernt werden. Der Container wird anschließend neu aufgesetzt.
17. Knopf zum Anlegen des in der Maske beschriebenen Jobs.

- **Job editieren (URL-Suffix: /!#/job/{jobid})**

Basierend auf der in der URL angegebenen ID, also der Schlüsselnummer des Jobs, werden die Daten zu diesem geladen. In einer Maske, die der entspricht, die bei der Erstellung eines Jobs angezeigt wird, kann man diese bearbeiten. Einzig das Attribut „Containername“ wird nicht mehr angezeigt, da dies nicht mehr geändert werden kann.

- **Anzeige der Daten zur Analyse (URL-Suffix: /!#/analytics/{jobid})**

Diese Ansicht [Abbildung 18] bietet eine Liste aller, der zum Job mit der in der URL angegebenen jobid gehörenden, Einträge von analytischen Daten. Folgende Funktionen und Informationen werden geboten:

Analytics for job with id 1

		# ▾	Successful repair	Timestamp
 	 	 1	 	 "2017-12-28T15:01:09.038Z"
		2		"2017-12-28T15:01:44.033Z"
		3		"2017-12-28T16:25:55.485Z"

ABBILDUNG 18 - LISTE ALLER ANALYTISCHEN DATEN EINES JOBS

1. Knopf zum Löschen des Eintrages. Hierbei wird nur der Eintrag aus der Liste entfernt, jedoch nicht die dazugehörigen Daten.

2. Knopf, der zu einer Liste mit allen zum Eintrag gehörenden Daten weiterleitet.
 3. Eindeutige Schlüsselnummer des Eintrages.
 4. Symbolisierung des Erfolges der Reparatur zu diesem Eintrag. Sofern erfolgreich erscheint ein grüner Daumen nach oben.
 5. Zeitstempel des Eintrages.
- **Liste aller zu einem Eintrag gehörenden Daten (URL-Suffix: `/!#/analytics/{jobid}/{id}`)**

Diese Ansicht [Abbildung 19] bietet dem Nutzer eine Übersicht über die zum Job gehörenden Daten. Hierbei werden nur die Protokolle angezeigt, da nur bei diesen sichergestellt ist, dass der Browser sie anzeigen kann. Anderes, wie die Sicherungen der Daten bei einem „Reset“ Schritt, können über den in das Dateisystem eingebundenen Ordner analysiert werden.

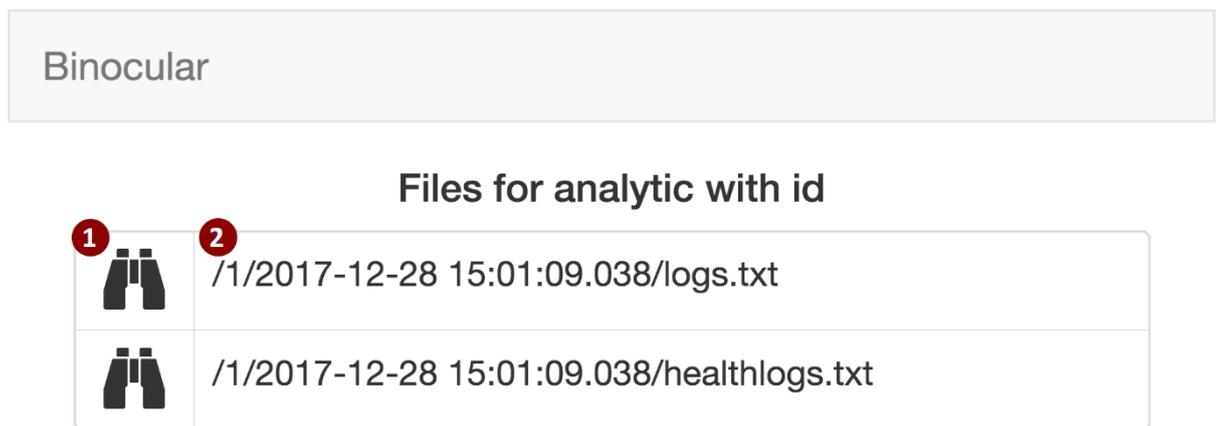


ABBILDUNG 19 - LISTE DER PROTOKOLLE

Erläuterung der Abbildung:

1. Knopf, über den die gewählte Datei des Eintrages angesehen werden kann.
2. Name und Ort der Datei im Ordner der Analysedaten.

7. SCHLUSSBETRACHTUNG

7.1 ZUSAMMENFASSUNG

Der Abschnitt Grundlagen dieser Arbeit beschreibt die Funktionsweise von Docker. Dies ist eine Softwarelösung, die es erlaubt Anwendungen in verschiedenen, unabhängigen und voneinander getrennten Betriebssystemen auf einem Rechner auszuführen. Mit Hilfe von Docker-Swarm und Kubernetes, beides Softwarelösungen, die zum Teil auf Docker basieren, ist es möglich diese Anwendungen auf mehreren Rechnern zu verteilen. Dies bringt Redundanz der Anwendungen und ermöglicht mehr Rechenleistung. Diese beiden Lösungen bieten dem Nutzer die Möglichkeit eingesetzte Anwendungen zu überwachen und im Problemfall neu aufzusetzen. Hierbei ist es durch die Redundanz möglich, die problembehaftete Anwendung durch eine weitere, bereits Laufende zu ersetzen.

Anhand von zu erwartenden Anwendungsfällen sind die Anforderungen an die Lösungen gestellt worden. Diese beschreiben, welche Funktionen in der Realisierung vorhanden sein müssen. Festgelegt wurde, dass die Lösung es dem Nutzer ermöglichen muss, seine mit Docker eingesetzten Anwendungen, mit geringem Aufwand zu überwachen und reparieren zu lassen. Die Reparatur muss automatisch geschehen, nachdem dies eingerichtet wurde. Des Weiteren müssen Daten zum Zeitpunkt eines Problems gesammelt werden, um den Ursprung des Problems herausfinden zu können.

Anschließend ist eine Analyse der bestehenden Lösungen formuliert. Basierend auf der Analyse und den Anforderungen sind diese bewertet worden. Hierbei fiel vor allem auf, dass keine grafische Oberfläche geboten wird und keine Möglichkeit besteht, eine Routine zur Reparatur einer Anwendung zu bestimmen. Dadurch, dass die bestehenden Lösungen nur für verteilte Systeme ausgelegt sind, geht die zu entwickelnde Lösung auf diese Lücke ein und erlaubt die Überwachung und Reparatur von Anwendungen auf einer einzelnen, lokalen Docker-Instanz.

Basierend auf dieser Erkenntnis beschreibt der darauffolgende Abschnitt das Konzept der Anwendung und der genutzten Datenbank. Er beinhaltet die geplanten Funktionen und die dazugehörigen, vorgesehenen Technologien. Die Anwendung, die die Überwachung und Reparatur übernimmt, soll mit Java und dem Framework Spring-Boot entwickelt werden. Über eine Schnittstelle, die REST-konform ist, ist die Anwendung erreichbar. Mit Hilfe von JavaScript soll ein Webserver die Benutzeroberfläche zur Verfügung stellen, über die der Nutzer mit der Java-Anwendung kommuniziert.

Um die Lösung auf ihre korrekte Funktion hin prüfen zu können, ist im Konzept vorgesehen, dass Microservices entwickelt werden, die in ihrer Funktion eingeschränkt werden können. Dies soll den Absturz und Defekt einer Anwendung widerspiegeln.

Anhand dieses Konzeptes ist anschließend die Realisierung des Projektes dargelegt. Zunächst sind die entwickelten Microservices erläutert, die als Docker-Image verfügbar sind. Über diese lassen sich Webserver starten und stoppen, sodass die entwickelte Lösung sie überwachen kann.

Weiter ist die fertige Lösung zur Überwachung und Reparatur von Containern in lokalen Docker-Instanzen beschrieben. Die Umsetzung der Lösung und der darin eingesetzten Technologien wird in diesem Abschnitt behandelt und anhand von Beispielen erklärt. Bis auf kleine Änderungen entspricht die fertige Lösung dem beschriebenen Konzept.

Das Ergebnis der Entwicklung ist basierend auf den Anforderungen und der Realisierung der Lösung im Weiteren beschrieben. Die Anforderungen, sowohl funktional, als auch nicht-funktional, sind fast vollständig erfüllt. Da die nicht erfüllten Anforderungen die Funktionsweise der Lösung nicht einschränken und begründet nicht umgesetzt wurden, ist das Ergebnis der realisierten Lösung als erfolgreich betrachtet.

Abschließend ist die Benutzeroberfläche der Anwendung als Dokumentation beschrieben. Diese erleichtert dem Nutzer die Bedienung der Lösung und erläutert das Aufsetzen der Anwendung.

7.2 IDEEN ZUR WEITERFÜHRUNG DES PROJEKTES

Während der Entwicklung der Anwendung haben sich aufgrund der Gegebenheiten Ideen entwickelt, die als Verbesserungen in spätere Versionen der Software einfließen könnten.

Die zunächst wichtigste Erweiterung wäre die Möglichkeit, dem Anwender das Nutzen vom privaten Docker-Repositories zu ermöglichen. Da diese für den Nutzer zunächst eine zusätzliche Konfigurationsmöglichkeit sind, wäre dies für Nutzer von Vorteil, die bereits mehr mit Docker vertraut sind.

Bereits bei der Analyse der bestehenden Lösung mit Docker-Swarm zeigten sich Verbesserungsmöglichkeiten. Dazu zählen die Erstellung von Health-Checks durch eine Benutzeroberfläche oder das Sichern von Daten zur Problemanalyse bei einem fehlgeschlagenen Health-Check. Durch die kaum vorhandene Dokumentation zur Reparatur von Containern wäre es hierbei für den Nutzer ebenfalls von Vorteil, die Möglichkeit zu haben, eine Anwendung nutzen zu können, die das Arbeiten mit Docker-Swarm in Zusammenhang mit Health-Checks und sich reparierenden Containern erleichtert. Diese Idee ist jedoch nicht die Realisierung eingeflossen, da dies in dem Zeitrahmen nicht in der Qualität der realisierten Lösung möglich gewesen wäre.

Das Projekt könnte weitergehend daraufhin ausgeweitet werden, dass der Nutzer die Parameter eines Containers direkt in einem erstellten Job anpassen kann. Dies könnte wiederum dafür verwendet werden, dem Nutzer die Möglichkeit zu bieten, bei einem Reparaturschritt bestimmte Parameter wegzulassen oder hinzuzufügen, um damit die Wahrscheinlichkeit einer erfolgreichen Reparatur zu erhöhen.

QUELLEN

[Docker Inc. 2017a] Docker Inc. (2017): Docker Documentation. <https://docs.docker.com/get-started>, abgerufen am 20.11.2017

[Docker Inc. 2017b] Docker Inc. (2017): Dockerfile reference. <https://docs.docker.com/engine/reference/builder/>, abgerufen am 21.11.2017

[Google 2017] Google (2017): AngularJS API Docs, <https://docs.angularjs.org/api>, abgerufen am 03.12.2017

[Hecht 2016] Hecht, Lawrence (2016): The present state of container orchestration. <https://thenewstack.io/tns-research-present-state-container-orchestration/>, abgerufen am 06.12.2017

[Kubernetes 2017] The Kubernetes Authors (2017): Kubernetes Documentation. <https://kubernetes.io/docs/home/>, abgerufen am 27.11.2017

[Matthias, Kane 2016] Matthias, Karl & Kane, Sean: Docker Praxiseinstieg (2016): Deployment, Testen und Debugging von Containern in Produktivumgebungen. mitp Professional

[Newman 2015] Newman, Sam (2015): Microservices: Konzeption und Design. mitp Professional

[Resin 2015] Rensin, David (2015): Kubernetes - Scheduling the Future at Cloud Scale. O'REILLY

[RV 2016] RV, Rajesh (2016): Spring Microservices. Packt Publishing

[Smith 2015] Smith, Chris (2015): Angular Basics, <http://www.angularjsbook.com/angular-basics/chapters/>, abgerufen am 03.12.2017

[Soppelsa, Kaewkasi 2016] Soppelsa, Fabrizio & Kaewkasi, Chanwit (2016): Native Docker Clustering with Swarm. Packt Publishing

[Spring 2017] Phillip Webb , Dave Syer , Josh Long , Stéphane Nicoll , Rob Winch , Andy Wilkinson , Marcel Overdijk , Christian Dupuis , Sébastien Deleuze , Michael Simons (2017): Spring Boot Reference Guide, <https://docs.spring.io/spring-boot/docs/1.5.9.RELEASE/reference/pdf/spring-boot-reference.pdf>, abgerufen am 01.12.2017