Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Mieke Narjes

Konzeption und Umsetzung eines automatischen Kubernetes Deployments in die ICC der HAW in Hinblick auf spätere Weiterverwendung in der Lehre.

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Mieke Narjes

Konzeption und Umsetzung eines automatischen Kubernetes Deployments in die ICC der HAW in Hinblick auf spätere Weiterverwendung in der Lehre.

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr.-Ing. Olaf Zukunft

Eingereicht am: 22. Juli 2018

Mieke Narjes

Thema der Arbeit
Konzeption und Umsetzung eines automatischen Kubernetes Deployments in die ICC der HAW in Hinblick auf spätere Weiterverwendung in der Lehre.

Stichworte
Continuous Integration, Continuous Deployment, Docker, GitLab, Informatik Compute Cloud, Kubernetes, Microservice, Monolith

Kurzzusammenfassung
Ziel dieser Arbeit ist die Umsetzung eines automatischen Deployments mehrerer zusammenarbeitender Anwendungen in die Informatik Compute Cloud der HAW. Die Projektstruktur wird mit besonderem Augenmerk auf eine spätere Nutzung in der Lehre erarbeitet und diskutiert. Dazu werden verschiedene Methoden zum Projektmanagement genutzt, um nach einer Analyse des Sachverhaltes den Umfang des Vorhabens zu planen und anschließend durchzuführen.

Mieke Narjes

Title of the thesis
Conception and Realization of a Kubernetes Deployment to the ICC of the HAW considering further use in teaching.

Keywords
Continuous Integration, Continuous Deployment, Docker, GitLab, Informatics Compute Cloud, Kubernetes, Microservice, Monolith

Abstract
The objective of this thesis is to implement an automatic Deployment of several applications that work together into the Informatics Compute Cloud of the HAW. The project structure will be determined and discussed turning special attention to a further use in teaching. For that purpose, several methods for project management will be used to plan and execute the project after a broad analysis of the circumstances.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Glossar

Big-Bang-Integration	Big-Bang-Integration beschreibt die Integration vieler Änderungen auf einmal. Dies führt oft zu Fehlern und einer aufwändigen Fehlersuche, da nicht klar ist, welche Änderung die Probleme herbeigeführt hat (vgl. Swartout, 2014, S.39).
CD	Continuous Deployment
CI	Continuous Integration
Deploy-Key	Deploy-Keys erlauben read-only oder read-write Zugriff auf eines oder mehrere Projekte mit einem SSH-Schlüssel-Paar (vgl. GitLab, 2018b).
Deployment	Als Deployment bezeichnet man die Verteilung, Installation, Wartung und/ oder die Deinstallation von Software (vgl. Williams, 2017).
DevOps	DevOps setzt sich zusammen aus *Development* und *Operations*. Das Development umfasst die Softwareentwicklung und alle daran beteiligten Arbeitskräfte, wohingegen Operations den Entwicklungsprozess, den Betrieb, das Testen und die Bereitstellung von neuen Versionen umfasst (vgl. Berngruber, 2017, S.15).

Eventual consistency	In verteilten Datenbanken können Änderungen nicht sofort auf alle Kopien geschrieben werden. Somit befindet sich das System nicht zu jeder Zeit in einem konsistenten Zustand, erreicht diesen aber immer wieder, indem die Veränderungen irgendwann (eventually) angeglichen werden (vgl. Oracle, 2012, S.1).
HLS	HAW-Logistics-System
ICC	Informatik Compute Cloud
Iteration	Iterative Entwicklung beschreibt die stetige Wiederholung von Projektphasen. So kann in jeder Iteration die Analyse, Konzeption, Implementierung und das Testen durchgeführt werden. Mit jeder Iteration erhält man ein etwas vollständigeres Produkt mit der Möglichkeit Fehler früher zu erkennen und gegenzusteuern (vgl. Cockburn, 2001, S.48).
Least connected	Least connected ist ein Loadbalancing-Algorithmus der an denjenigen Server weiterleitet, der die wenigsten Anfragen hat, um eine Überlastung anderer Server entgegenzuwirken (z. B.wenn die Bearbeitung einer Anfrage länger dauert) (vgl. Nginx, 2018a).

REST	Representational-State-Transfer
Round-Robin	Round-Robin ist ein Loadbalancing-Algorithmus, der allen Servern in einer Ringstruktur nacheinander Anfragen weiterleitet (vgl. Nginx, 2018c). Der Algorithmus wird auch in vielen anderen Bereichen mit ähnlichen Anforderungen verwendet.
User-Story-Map	User-Story-Maps dienen der Organisation von User-Stories sowohl in zeitlicher Reihenfolge als auch inhaltlich. Die erarbeiteten User-Stories werden dabei Aufgaben (*Tasks*) zugeordnet und nach Priorität von oben nach unten sortiert (vgl. Jeff Patton, 2015, S.19 ff.).
VM	Virtual Machine

1 Einleitung

1.1 Motivation

Die frühe Softwareentwicklung am Großrechner mit Lochkarten war wenig fehlertolerant und schwergängig. Um die zugeteilte Rechenzeit optimal zu nutzen, waren Entwickler angehalten, ihre Programme noch vor deren Ausführung umfassend zu durchdenken und allen Anforderungen gerecht zu werden. Heutige Rechner sind erschwinglich, klein und schnell. Anwendungen können mit ganz anderen Geschwindigkeiten entwickelt werden als zuvor und sind zunehmend verteilt. Auf diesem schnelllebigen Markt sind kurze Releasezyklen als Reaktion auf Veränderungen vorteilhaft, um mit der Entwicklung mithalten zu können. Microservices und DevOps sind vor diesem Kontext populäre Begriffe in der Softwareentwicklung.

Des Weiteren werden Anwendungen mit Erstarkung der Containertechnologie zunehmend plattformunabhängig und damit noch einfacher in verschiedenen Produktionsumgebungen einsetzbar (vgl. Mouat, 2016, S.3). Die manuelle Ausführung von Builds oder Deployments ist fehleranfällig und verlängert die sogenannte “time to market“. Als Reaktion werden viele der notwendigen Schritte zur Auslieferung einer Anwendung automatisiert (vgl. Augsten, 2017).

Orchestrierungstools wie Kubernetes werden eingesetzt, um die Menge an Containern zu überwachen, und spielen damit eine zentrale Rolle zur Entwicklung und Bereitstellung von containerisierten Anwendungen (vgl. Kubernetes, 2018n). Um derartige Technologien in Unternehmen umzusetzen, sind qualifizierte Fachkräfte zunehmend gefragt.

1.2 Zielsetzung

Ziel dieser Arbeit ist der Aufbau einer nutzerfreundlichen Projektstruktur mit einem automatischen Deployment zur Informatik Compute Cloud der HAW. Als Grundlage dient ein bereits seit mehreren Semestern von Studierenden entwickeltes Logistik-System, das weiterhin zur Lehre eingesetzt werden soll. Mit dem Kubernetes-Deployment wird das System um eine derzeit aktuelle Thematik ergänzt. Im Rahmen eines Wahlpflichtkurzes sollen Studierende sich durch das weiterentwickelte Projekt wichtige Kompetenzen in diesem Bereich aneignen können. Erfahrungsgemäß ist die Einarbeitung in ein unbekanntes System, gerade bei fremden Fachgebieten, zunächst mühsam und zeitaufwendig. Hierzu wird eine Infrastruktur geschaffen, die den Einstieg und die Weiterentwicklung nach Möglichkeit erleichtert und Fehlerquellen reduziert.

1.3 Aufbau der Arbeit

Kapitel 1, Einleitung

Im ersten Kapitel wird die Zielsetzung der Thesis näher beleuchtet und die zugrunde liegende Motivation. Zusätzlich wird der Aufbau der Arbeit kurz dargelegt.

Kapitel 2, Grundlagen

Das zweite Kapitel bietet Grundlagen, die zum Verständnis dieser Bachelorarbeit nötig sind. Das Verständnis von UML-Diagrammen, Versionskontrolle mit Git, dem HTTP-Protokoll, JSON, XML, YAML sowie die Kenntnis von Java, C#, Typescript und HTML wird vorausgesetzt. Das Kapitel umreißt sowohl grundlegende Techniken und Begriffe im Projektmanagement und Softwareengineering als auch Docker, Kubernetes und GitLab.

Kapitel 3, Analyse und Spezifikation

Das dritte Kapitel dient der Erfassung von Stakeholdern und den sich daraus ergebenden Anforderungen. Dies beinhaltet verschiedene Persona, eine Sammlung von User-Stories und eine Spezifikation, die im Projektverlauf umgesetzt wird.

Kapitel 4, Architektur und Planung

Im vierten Kapitel werden anhand der vorangegangenen Analyse die Projektstruktur und ein Projektplan erarbeitet. Neben der Architektur und Planung wird außerdem eine Risikoanalyse durchgeführt und diskutiert.

Kapitel 5, Durchführung

Das fünfte Kapitel beschreibt die Umsetzung der in den vorangegangenen Kapiteln erarbeiteten Pläne und Anforderungen. Dies umfasst die Durchführung des Refactorings, der Dockerisierung der Anwendungen und das automatische Deployment in ein Kubernetes-Cluster.

Kapitel 6, Evaluation

Die Evaluation diskutiert die zuvor beschriebenen Lösungen. Es wird betrachtet, ob die Anwendung die zuvor erarbeiteten Anforderungen erfüllt und inwieweit dies den Erwartungen der zukünftigen NutzerInnen entspricht.

Kapitel 7, Fazit und Ausblick

Im siebten Kapitel wird die Arbeit zusammenfassend bewertet und Zukunftsperspektiven diskutiert. Dazu werden verschiedene Vorschläge zur Weiterentwicklung des Projekts dargelegt.

2 Grundlagen

2.1 Softwareentwicklung

2.1.1 Schichtenarchitektur

> „In Informations- oder Websystemen hat sich die Schichtenbildung (layering) als ein klassisches Mittel zur Strukturierung etabliert. Eine Schicht bietet den darüber liegenden Schichten bestimmte Dienste (services) an. Eine Schicht kapselt die Details ihrer Implementierung gegenüber der Außenwelt. Sie kann dabei ein beliebig komplexes Subsystem darstellen." Starke (2011)

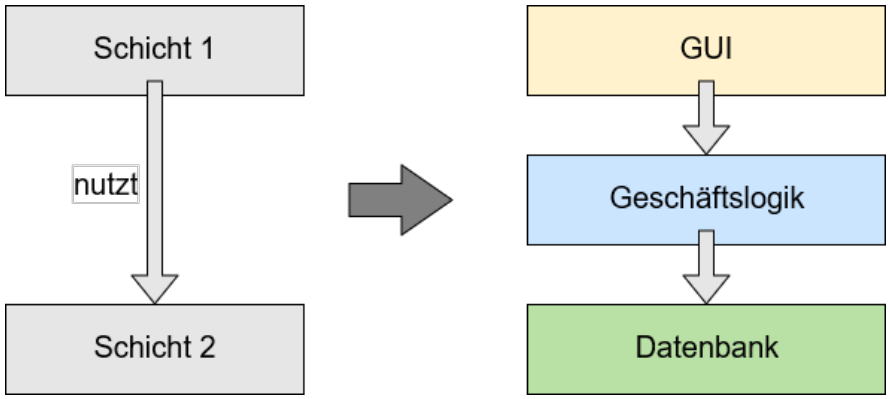


Abbildung 2.1: Schematik und Beispiel - Schichtenarchitektur

Abb. 2.1 zeigt schematisch den Aufbau einer *Schichtenarchitektur* und eine mögliche Umsetzung. Wichtig zu beachten ist, dass die obere Schicht immer auf die Nächst untere zugreift und nicht umgekehrt, um zirkuläre Abhängigkeiten zu vermeiden.

2.1.2 Monolithische Applikation

Einer der bekanntesten Architekturstile ist die *monolithische Applikation*. Dieser Stil zeichnet sich durch die Bearbeitung aller Geschäftsprozesse innerhalb einer fest zusammenhängenden Applikation aus (vgl. Diedrich, 2018, S.5). Eine monolithische Applikation muss aus diesem Grund im Ganzen skaliert und bei Änderung neu gebaut und deployed werden. In der Regel führen Fehler in Einzelstrukturen des Monolithen zum Absturz des Gesamtsystems.

2.1.3 Microservice

> „Microservices sind kleine, eigenständige Services, die kollaborieren bzw. sich gegenseitig zuarbeiten." Newman (2015)

Wie in Abb. 2.2 illustriert kann eine Schichtenarchitektur zum Beispiel monolithisch oder als *Microservice* (ebenfalls ein Architekturstil) aufgebaut sein. Im Gegensatz zum Monolithen bildet in letzterer Variante jede Schicht einen eigenen Microservice, der für eine ganz konkrete Aufgabe zuständig ist und diese komplett erfüllt. Ähnlich wie Komponenten bieten die Microservices Schnittstellen nach außen, damit andere Services diese nutzen können. Auf diese Weise können Microservices mit wenig Aufwand ausgetauscht oder skaliert werden. Durch die klare Kapselung können innerhalb der Services beliebige Technologien verwendet werden, solange die Schnittstellen weiterhin angeboten werden. Bei einer Änderung eines Services kann auf ein komplettes Redeployment der Anwendung verzichtet werden, indem nur der betroffene Service neu deployed wird (vgl. Newman, 2015, S.11 ff., S.24 ff.).

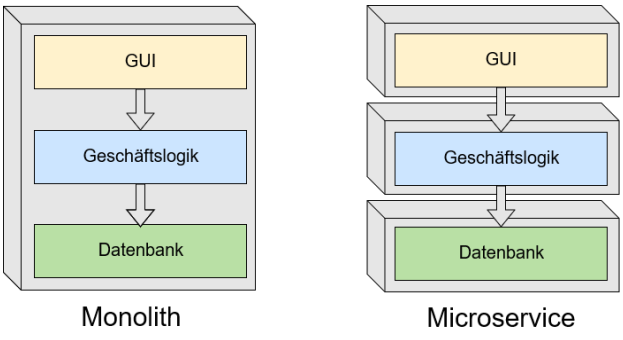


Abbildung 2.2: Vergleich - Monolith und Microservices

2.2 Anforderungsanalyse

Die *Anforderungsanalyse* befasst sich mit der Ermittlung von Anforderungen an ein System. Es wird zwischen drei Anforderungsarten unterschieden:

Funktionale Anforderungen *Funktionale Anforderungen* beschreiben die gewünschten Funktionen für das fertige Produkt (vgl. Klaus Pohl, 2015, S.8).

Qualitätsanforderungen *Qualitätsanforderungen* beschreiben die Qualität von Funktionen eines Produkts (z. B. Performanz, Portabilität) und beeinflussen damit häufig die Systemarchitektur (vgl. Klaus Pohl, 2015, S.9).

Randbedingungen "*Randbedingungen* können von den Projektbeteiligten nicht beeinflusst werden. [...] Randbedingungen werden, im Gegensatz zu funktionalen Anforderungen und Qualitätsanforderungen, nicht umgesetzt, sondern schränken die Umsetzungsmöglichkeiten, d. h. den Lösungsraum im Entwicklungsprozess, ein." Klaus Pohl (2015)

Im Folgenden werden einige Hilfsmittel zur Anforderungsanalyse vorgestellt.

2.2.1 Stakeholder

> „Anspruchsgruppen [= *Stakeholder*] sind alle internen und externen Personengruppen, die von den unternehmerischen Tätigkeiten gegenwärtig oder in Zukunft direkt oder indirekt betroffen sind." Thommen (2018)

In der Softwareentwicklung sind damit vor allem NutzerInnen und EntscheidungsträgerInnen bzw. GeldgeberInnen gemeint. Allerdings besteht der Betroffenenkreis oft aus deutlich mehr und teils unauffälligeren Personengruppen, weshalb eine initiale Stakeholderanalyse sinnvoll ist.

2.2.2 Persona

> „The personas method is widely used by technology designers and human-computer interaction researchers to describe users and customers [...]." Chapman u. a. (2008)

Eine *Persona* ist die Beschreibung einer fiktiven Person, die einen wichtigen Kunden oder eine Nutzergruppe des zu entwickelnden Produktes oder Services darstellt. Sie umfasst Informationen über demografische Daten, Verhalten und Ziele (vgl. Chapman u. a., 2008, S.1). Dies hilft den Entwicklern, zu überprüfen, ob ihr Produkt den Anforderungen einer Zielgruppe entspricht.

2.2.3 Value-Proposition-Canvas

Ein *Value-Proposition-Canvas* dient - oft in Verbindung mit einer Persona - der Analyse von Kundenbedürfnissen und den daraus resultierenden Möglichkeiten zur Produktentwicklung (vgl. Osterwalder u. a., 2015, XV ff.). Abb. 2.3 zeigt den Aufbau eines solchen Diagramms nach Alexander Osterwalder.

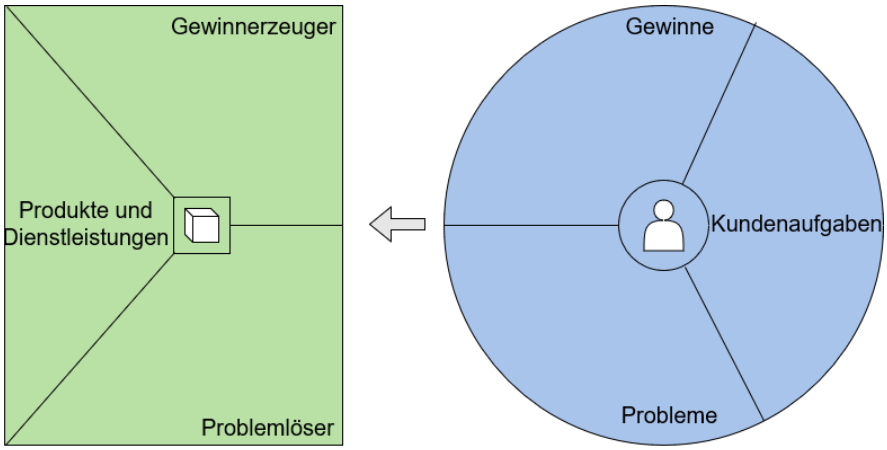


Abbildung 2.3: Beispiel - Value-Proposition-Canvas

Kundenaufgaben

Kundenaufgaben stellen Aufgaben und Probleme dar, die ein Kunde zu erledigen bzw. zu lösen hat. Neben technischen Anforderungen kann dies auch soziale und emotionale Aufgaben umfassen.

Probleme

Probleme beschreiben Schwierigkeiten, mit denen der Kunde bei seinen Aufgaben in Berührung kommt. Es wird betrachtet, was dem Kunden beispielsweise zu teuer ist oder zu lange dauert. Ebenfalls in diese Kategorie fallen Funktionen, welche bisherigen Lösungen fehlen oder Risiken, die der Kunde fürchtet einzugehen.

Gewinne

Gewinne sind Elemente, die der Kunde einsparen möchte. Dabei kann es sich beispielsweise um Geld, Zeit oder Aufwand handeln. Kundenwünsche und Leistungen, die die Lösung eines Kundenproblems erleichtern, gehören ebenfalls in diese Kategorie.

Produkte und Dienstleistungen

Produkte und Dienstleistungen beschreiben Funktionen, die dem Kunden seine Arbeit erleichtern und seine Bedürfnisse befriedigt.

Problemlöser

Problemlöser umschreiben, was das eigene Produkt im Gegensatz zu bestehenden Lösungen besser macht und inwieweit es den Problemen eines Kunden entgegenwirkt.

Gewinnerzeuger

Gewinnerzeuger sind Funktionen, die dem Kunden die Arbeit erleichtern oder Kundenwünsche erfüllen.

(vgl. Osterwalder u. a., 2015, S.13 ff., S.28 ff.)

2.2.4 User-Story

User-Stories beschreiben Anforderungen an Software von Nutzerseite aus und werden zur Anforderungsanalyse genutzt. Üblicherweise werden vor Erstellung der User-Stories sogenannte *User-Roles* identifiziert. Dies sind Nutzergruppen, die die Software nach Fertigstellung nutzen würden. Bei einem Shop-System sind dies meist die KäuferInnen. Aber auch ServicemitarbeiterInnen und GeschäftsführerInnen haben oft Zugriff auf die Software, allerdings mit ganz anderen Anforderungen als Kunden/ Kundinnen. Anhand dieser *User-Roles* werden Funktionen identifiziert, die die geplante Software enthalten sollte.

Diese Funktionen werden als User-Stories in der Form „Als *<User-Role>* möchte ich *<Funktion>* tun können“ notiert und anschließend im Team diskutiert, um ein gemeinsames Verständnis der Anforderung zu erreichen (vgl. Jeff Patton, 2015, S.3 ff.).

2.2.5 Use-Case

Ein Anwendungsfall (*Use-Case*) beschreibt das Verhalten von und die Interaktion mit einem System unter verschiedenen Bedingungen. Ähnlich wie User-Stories dient es der Herstellung eines gemeinsamen Verständnisses in Entwicklerteams und Stakeholdern (vgl. Cockburn, 2010, S.15 f.). Nach Ermittlung der User-Stories können in Use-Cases konkrete Anwendungsfälle erarbeitet werden. Beispielsweise werden für einen E-Mail Klienten alle Schritte, die zum Abschicken einer Mail nötig sind, festgehalten. Es gibt verschiedene Möglichkeiten Use-Cases zu dokumentieren. In Tabelle 2.1 ist eine simple tabellarische Form dargestellt.

Use Case	<Ziel kurz beschrieben>	
Anwendungskontext	<längere Beschreibung falls nötig>	
Ebene	<Hauptaufgabe, Subfunktion, Überblick,...>	
Primärakteur	<Rolle/Beschreibung des Hauptakteurs>	
Stakeholder+Interessen	**Stakeholder**	**Interessen**
	<Name>	<Interessen>
	...	...
Vorbedingungen	<Was muss vorher erfüllt sein? Stand der Dinge?>	
Invarianten	<Was muss in jedem Fall erreicht werden?>	
Nachbedingungen	<Bei erfolgreichem Ausgang befriedigten Interessen>	
Trigger	<Aktion im System, die den Use Case auslöst>	
Beschreibung	**Schritt**	**Aktion**
	1	<Beschreibung: Trigger bis Zielausgabe>
	...	...
Erweiterungen	**Schritt**	**Verzweigende Aktion**
	1a	<Bedingung die Verzweigung auslöst> <Titel/Aktion des Teil-Use Case>
	...	...
Fehlerfälle	<Fehlerfälle>	

Tabelle 2.1: Beispiel - Use-Case-Tabelle angelehnt an (Cockburn, 2010, S.153)

Neben der schriftlichen Dokumentation bietet es sich an, Anwendungsfälle zusätzlich in Use-Case-Diagrammen zu visualisieren (vgl. van Randen u. a., 2016, S.2). Es ist zu beachten, dass nicht jede User-Story als Use-Case ausgearbeitet werden muss und auch nicht jeder Use-Case. Wenn es dem Verständnis nicht hilft oder die User-Story nicht wichtig oder komplex genug ist, sollte darauf verzichtet werden.

2.3 Projektplanung

2.3.1 Kanban

Kanban ist ein Verfahren, das bei Toyota zur Optimierung und Visualisierung der Lagerungs- und Arbeitsschritte eingeführt wurde (vgl. Brunner, 2017, S.105). Sogenannte Kanban-Boards (siehe Abb. 2.4) werden in der IT vorwiegend in der agilen Softwareentwicklung zur Aufgabenverteilung und -überwachung genutzt.

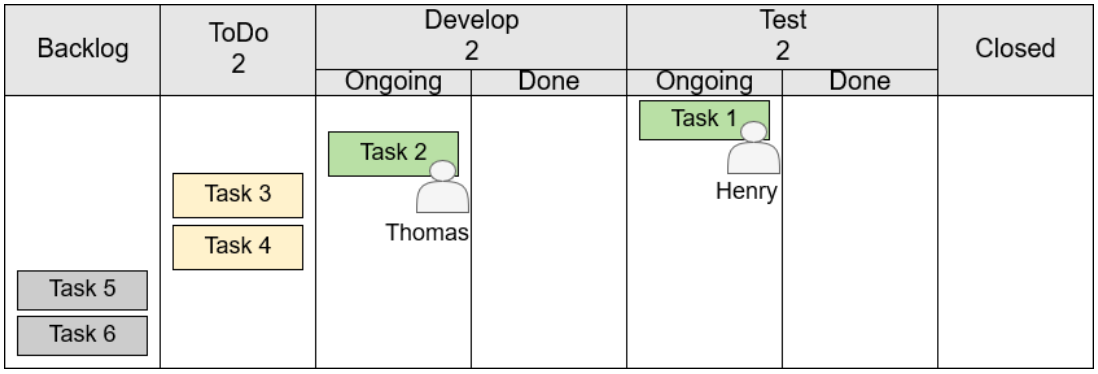


Abbildung 2.4: Beispiel - Kanban-Board

Die erarbeiteten User-Stories (siehe 2.2.4) werden nach Priorität in die *Backlog*-Spalte eingetragen und alle in einer Iteration zu bearbeitenden User-Stories in die *ToDo*-Spalte verschoben. Teammitglieder können nach dem *Pull-Prinzip* Aufgaben übernehmen und in die *Develop-Ongoing*-Spalte ziehen. Das *Pull-Prinzip* besagt, dass niemandem eine Aufgabe zugewiesen wird (Push), sondern immer direkt von einem Teammitglied übernommen werden muss (Pull). So wird der Überlastung einzelner entgegengewirkt. Um anderen Teammitgliedern die Weiterbearbeitung einer Aufgabe zu ermöglichen, wird diese anschließend in die jeweilige *Done*-Spalte verschoben und kann dann erneut

gezogen werden (vgl. Kniberg, 2011, S.17.ff, S.23.ff., S.29.ff).
Ein weiteres Prinzip ist die Beschränkung der Work-in-Progress (WIP). Demnach dürfen sich in den "Bearbeitungsspalten" nie mehr als eine festgelegte Menge an User-Stories befinden. Dadurch wird die Arbeit auf wenige Aufgaben fokussiert und User-Stories die den Arbeitsfluss aufhalten können besser identifiziert werden (vgl. Kniberg, 2011, S.50.ff). Die Beschränkung wird als Zahl in der jeweiligen Spalte notiert.

In diesem Beispiel wurden die Spalten Backlog, ToDo, Develop, Test und Closed verwendet. Je nach Bedarf können diese allerdings an den Arbeitsprozess ergänzt und angepasst werden.

2.3.2 Gantt-Diagramm

Gantt-Diagramme (siehe Abb. 2.5) benannt nach Henry L. Gantt dienen der Planung von Aufgaben in Rücksichtnahme von zeitlichen Einschränkungen oder Abhängigkeiten untereinander. Aufgaben oder Ereignisse werden als unterschiedlich lange Balken im Diagramm dargestellt und können zum Beispiel als voneinander abhängig markiert werden, indem sie durch eine Linie verbunden werden.

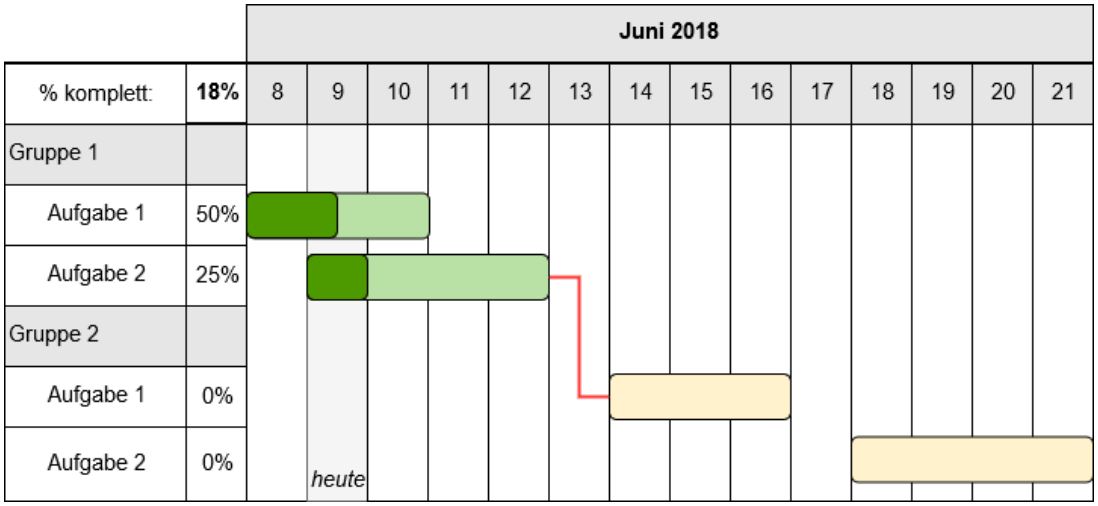


Abbildung 2.5: Beispiel - Gantt-Diagramm

Die Länge des Balkens hängt von der geplanten Dauer der Aufgabe ab und wird in Abhängigkeit des Zeitstrahls eingetragen (vgl. Wilson, 2003, S.5). Bei Bedarf kann mit modernen Tools der Projektfortschritt prozentual eingetragen werden. Gerät eine Aufgabe in Verzug, so wirkt sich das direkt auf alle von ihr abhängigen Folgeaufgaben aus. Auf diese Weise kann auch ein komplexer Projektverlauf mit zeitlichen Überschneidungen und Abhängigkeiten übersichtlich dargestellt werden.

2.3.3 Risikomanagement

> „Ein Risiko ist die Auswirkung von Unsicherheit auf Ziele. Ein Risiko charakterisiert sowohl Gefahren [als] auch Chancen“ Ebert (2014)

Die erfolgreiche Projektdurchführung bedingt ein durchdachtes *Risikomanagement*, um negative Folgen im Zweifelsfall aufzufangen. Hierzu werden regelmäßig Risiken identifiziert, bewertet, abgeschwächt und anschließend weiter kontrolliert. Der Prozess ist zyklisch, da im Projektverlauf manche Risiken eliminiert werden und neue auftauchen. Es können verschiedene Techniken zur Risikoidentifikation angewendet werden, die in dieser Arbeit jedoch keine Rolle spielen und somit hier nicht weiter ausgeführt werden. Zur Bewertung wird oft folgende Formel verwendet:
“`Risiko = Auswirkungen x Eintrittswahrscheinlichkeit`“.
Auswirkungen und Eintrittswahrscheinlichkeit bewegen sich in einem definierten Wertebereich. Je höher der ermittelte Risikowert, desto dringlicher ist die Abschwächung eines solchen Risikos. Man kann Risiken ignorieren, die Eintrittswahrscheinlichkeit reduzieren oder den zu erwartenden Schaden einschränken (vgl. Ebert, 2014, S.8, 18, 79 ff.).

2.4 Verwendete Applikationen

2.4.1 HAW-Logistics-System

Das HAW-Logistics-System (HLS) ist eine monolithische .NET-Anwendung zur Annahme, Planung und Durchführung von Transportaufträgen. Es ist in der Lage Frachtrouten, Sendungen und Kundendaten zu verwalten und darzustellen. Eine Buchhaltungskomponente ist zuständig für die Verwaltung von Rechnungen und der Bezahlung von

Frachtführern. Die Buchhaltungskomponente kommuniziert mit der Bank (siehe 2.4.2) über den Message-Queue Dienst RabbitMQ. Im Folgenden finden sich einige Eckdaten zu diesem Projekt:

- **Sprachen**: C# und Typescript
- **Frameworks/ Tools**: .NET und Angular 4
  - *.NET* ist ein plattformübergreifendes Framework zur Anwendungsentwicklung und Ausführung auf Windows, Linux und Mac. Es ermöglicht die Arbeit mit mehreren Sprachen, Editoren und Bibliotheken (vgl. Microsoft, 2018).
  - *Angular* ist ein Frontend-Framework zur Erstellung von Frontend-Applikationen in NodeJS (vgl. Angular, 2018).
- **Datenbanken**: Neo4j und PostgreSQL
  - *Neo4J* ist eine Graphdatenbank, zur Speicherung, Bearbeitung und Darstellung von graphbasierten Daten (vgl. Neo4J, 2018).
  - *PostgreSQL* ist eine relationale open-source Datenbank, die im HLS-System zur Speicherung von Kundendaten verwendet wird (vgl. PostgreSQL, 2018).
- **Kommunikationsmechanismen**: REST und RabbitMQ
  - *Representational-State-Transfer (REST)* ist ein Architekturstil zur Kommunikation in verteilten Systemen. Hierbei werden über HTTP JSON-Nachrichten ausgetauscht, welche alle nötigen Informationen zur Weiterverarbeitung einer Ressource enthalten (vgl. Tilkov u. a., 2015, S.11 ff.). Eine Ressource ist mit einem Java-Objekt vergleichbar.
  - *RabbitMQ* ist ein open-source Message-Broker (vgl. RabbitMQ, 2018), der zur Kommunikation zwischen dem Bank-Backend und dem HLS-System verwendet wird. Zusätzlich dient es der internen Kommunikation zwischen einzelnen HLS-Komponenten.

Weitergehende Informationen zu diesem Projekt befinden sich auf der CD unter “wiki/hls.wiki“.

2.4.2 Bank

Die Bank-Anwendung dient der Verwaltung von Kundenkonten als zentrales Managementsystem. Auf Anweisung des HLS-Systems führt die Bank Transaktionen aus und benachrichtigt dieses bei Zahlung einer Rechnung. Im Folgenden finden sich einige Eckdaten zu diesem Projekt, welches sich in zwei Module - Backend und Frontend - aufteilt:

Backend:

- **Sprachen**: Java
- **Frameworks/Tools**: Spring Boot und Maven
  - *Spring Boot* ist ein Framework zur Erstellung von Standalone-Applikationen mit dem Leitmotiv "Convention over Configuration"(vgl. Simons, 2018, S.91). Im Kern bedeutet es, dass per default alles über Konventionen festgelegt ist und der Nutzer/ die Nutzerin erst bei abweichenden Wünschen aktiv werden und eine eigene Konfiguration schreiben muss.
  - *Apache Maven* basiert auf dem Konzept Project-Object-Model und dient hier als Dependencymanagementsystem für Javasysteme (vgl. Maven, 2018).
- **Datenbanken**: H2
  - Die Bank-Anwendung nutzt die In-Memory Datenbank von *H2* (vgl. H2, 2018a). Dementsprechend werden die Daten nur bis zur Beendigung der Anwendung persistiert (vgl. H2, 2018b).
- **Kommunikationsmechanismen**: REST und RabbitMQ (siehe 2.4.1)

Frontend:

- **Sprachen**: Typescript
- **Frameworks/ Tools**: Angular 5 (siehe 2.4.1)

Weitergehende Informationen zu diesem Projekt befinden sich auf der CD unter "wiki/bank-backend.wiki" und "wiki/bank-frontend.wiki".

2.5 Docker-Container

> „Build, Ship and Run, Any App, Anywhere." - Docker Mantra
> (Mouat, 2016, Geleitwort)

> „*Container* sind eine schlanke und portable Möglichkeit, beliebige Anwendungen und ihre Abhängigkeiten zu verpacken und transportabel zu machen."
> Mouat (2016)

Vergleichbar mit einer Virtual Machine (VM) können in *Containern* vom Host-System isoliert Anwendungen gestartet werden. Allerdings sind VMs vergleichsweise schwergewichtig und langsam.

Ähnlich wie Container in der Logistik, dienen Software-Container der einheitlichen Verpackung von Ware (= Software). Die so verpackte Software sieht nach außen hin immer gleich aus und kann von jeder Maschine gestartet werden, die Containertechnologie beherrscht. Sofern ein Endnutzer/ eine Endnutzerin zum Beispiel eine Docker-Engine für den Umgang mit Containern laufen hat, können diese unabhängig vom Host-System gestartet werden. Die Probleme mit der Portierung von Anwendungen durch unterschiedliche Programmierumgebungen und Einstellungen können so eliminiert werden. Docker ist eine der bekanntesten Container Plattformen.

Container teilen sich Ressourcen mit dem Host-System und können innerhalb kurzer Zeit gestartet oder gestoppt werden. Der geringe Aufwand ermöglicht die Unterhaltung von vielen Containern gleichzeitig, um beispielsweise ein verteiltes System realitätsnah zu emulieren. Docker-Container können über Localhost-Ports miteinander kommunizieren, solang sie auf der gleichen Maschine laufen.

Abb. 2.6 zeigt drei Anwendungen je in Virtual Machines und in Containern. Während jede Applikation eine eigene VM mit *Gast-OS* benötigt, leistet eine einzige Docker-Engine diese Abstraktion im Beispiel in Abb. 2.6. *App B* und *App C* nutzen zudem die gleichen Bibliotheken, benötigen aber je Virtual Machine eine eigene Kopie. Über die Docker-Engine können sie sich die gleiche Bibliothek teilen (vgl. Mouat, 2016, S.3 ff.).

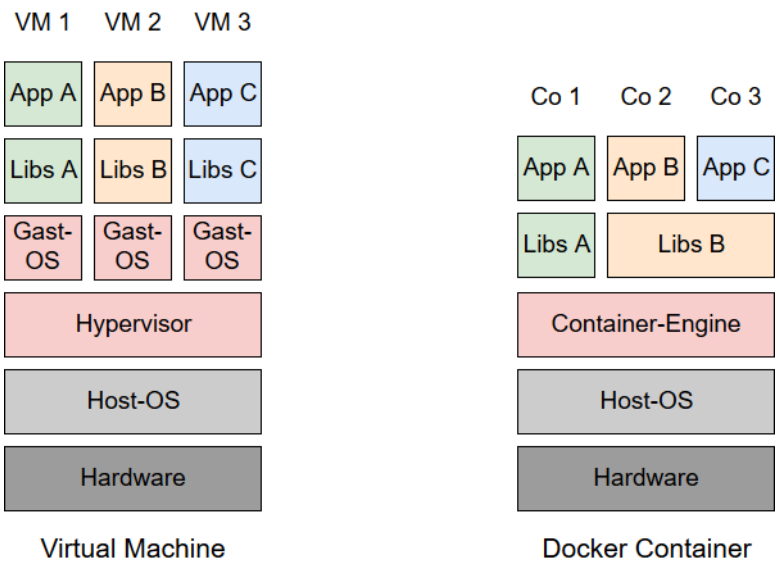


Abbildung 2.6: Vergleich - Virtual Machine und Docker-Container

2.5.1 Docker-Image

Docker-Images definieren in sogenannten Dockerfiles, wie eine Anwendung für einen Container bereitgestellt werden sollen (containerisieren/ dockerisieren). Das Codebeispiel 2.1 (übernommen aus Docker, 2018a) zeigt eine typische Image-Definition für eine Python-Anwendung, die über Port 80 erreichbar sein soll.
Die meisten Docker-Container basieren auf Unix Images. Neben den üblichen dockerspezifischen Befehlen "FROM, WORKDIR, ADD, COPY, CMD" und "EXPOSE" können mit "RUN" Unix Befehle ausgeführt werden, um beispielsweise zusätzliche Anwendungen für das Image zu installieren oder Konfigurationen vorzunehmen.

```
1 FROM python:2.7-slim
2 WORKDIR /app
3 ADD . /app
4 RUN pip install --trusted-host pypi.python.org -r req.txt
5 EXPOSE 80
6 ENV NAME World
7 CMD ["python", "app.py"]
```

Quellcode 2.1: Beispiel - Dockerfile

"FROM" definiert ein sogenanntes *Baseimage*, auf welches das eigene Image aufbauen soll. Im Beispiel handelt es sich um einen Linux Container mit den nötigsten Paketen für

Python:2.7 installiert. Mit "WORKDIR" wird der Ordner "/app" als Arbeitsverzeichnis definiert und angelegt, sofern dieser nicht vorhanden ist. Angenommen das Dockerfile befindet sich im Rootverzeichnis des zu dockerisierenden Python-Projektes. Dann werden mit "ADD . /app" alle sich dort befindenden Dateien in den "/app"-Ordner des Images geschrieben. Durch "CMD" wird schließlich "app.py" ausgeführt (vgl. Docker, 2018a). Der Befehl "`docker build -t <imagename> .`" im Verzeichnis mit dem Dockerfile baut das Image.

2.5.2 Multi-Stage Build

Vereinfacht formuliert wird bei Erstellung eines Images für jedes "FROM"-Statement ein Container gestartet, in dem die Befehle ausgeführt werden. Besonders für Applikationen, welche die Ausführung bestimmter build-Befehle benötigen, sind *multi-stage builds* hilfreich. So können sowohl unnötige Dateien im Image als auch die manuelle Ausführung solcher Befehle vor Erstellung jedes Docker-Images vermieden werden.

```
1  FROM maven:3-jdk-8-slim AS BUILD
2  WORKDIR /mvn-app
3  COPY pom.xml .
4  COPY src/main ./src/main
5  RUN mvn clean install
6
7  FROM openjdk:8-jdk-slim
8  COPY --from=BUILD /mvn-app/target/someapp.jar app.jar
9  ENV JAVA_OPTS=""
10 ENTRYPOINT exec java [...] -jar /app.jar
```

Quellcode 2.2: Beispiel - multi-stage build Dockerfile

Es können mehrere "FROM"-Statements in einem Dockerfile aufgerufen werden - diese werden sequenziell von oben nach unten abgearbeitet. Im Beispiel 2.2 wird im oberen Abschnitt die Jar einer Maven-Applikation erstellt und anschließend in einem Java Container ausgeführt. Der folgende Container kann über das "`--from=BUILD`"-Flag auf die Erzeugnisse der oberen Stage zugreifen, da dieser mit "AS BUILD" gekennzeichnet ist (vgl. Docker, 2018c).

Das Endergebnis ist ein Docker-Image, das ausschließlich die Jar enthält. Daten, die im obigen Abschnitt geladen wurden, verfallen am Ende des build-Vorgangs, sodass Images noch weiter verschlankt werden und build-Prozesse automatisiert stattfinden können.

2.5.3 Docker-Compose

Oft sind zum Starten einer Anwendung mehrere Programme nötig, die mit dieser interagieren (beispielsweise eine Postgres Datenbank). Jedes dieser Programme manuell zu installieren, zu konfigurieren und auszuführen ist vergleichsweise mühsam und fehleranfällig. Docker bietet mit *docker-compose* eine Möglichkeit alle nötigen Programme auf Befehl zu starten und bei Bedarf zu konfigurieren (vgl. Docker, 2018b).

```
1  version: '3'
2
3  services:
4   bank-frontend:
5    depends_on:
6     - bank-backend
7    image: "bank-frontend:local"
8    ports:
9     - 80:80
10
11  bank-backend:
12   image: "bank-backend"
13   ports:
14    - 8081:8081
15   environment:
16    - SPRING_RABBITMQ_ADDRESSES=amqp://guest:guest@rabbitmq:5672
```

Quellcode 2.3: Definition - docker-compose für die Bank-Anwendung

Das Codebeispiel 2.3 zeigt eine typische Anwendung für docker-compose. Ein Nutzer/eine Nutzerin möchte eine Frontend-Anwendung starten und benötigt dafür eine laufende Instanz einer Backend-Applikation. Unter "`services`" können solche Anwendungen mit Images, Ports und gegebenenfalls weiteren Variablen konfiguriert werden. Da das Frontend von einer laufenden Backend-Instanz abhängig ist, kann der Servicename mit "`depends_on`" angegeben werden, sodass das Frontend nicht zu früh startet. Navigiert man in einem Terminal zum docker-compose-File und führt "`docker-compose up`" aus, so werden die Container automatisch gestartet.

2.6 Kubernetes

> „Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.“ Kubernetes (2018n)

Kubernetes dient als Orchestrierungstool für Container-Anwendungen. Unter Orchestrierung versteht man das Management von Containern, welche für die Anwendung nötig sind. Darunter fallen unter anderem *Skalierung*, *Health Monitoring*, *Load Balancing* oder *Service Discovery*.

Skalierung kann horizontal und vertikal erfolgen. Unter horizontaler Skalierung versteht man die Vermehrung von Serviceinstanzen, während vertikale Skalierung die Erweiterung eines Services durch mehr Ressourcen (z. B. Datenbank, CPU,...) beschreibt. Kubernetes erkennt automatisch erhöhte Applikationslast und kann entsprechend der Konfiguration skalieren (vgl. Baier, 2017, S.107, S.114 ff.). *Health Monitoring* beschreibt die Überwachung des Zustands von Applikationen oder Containern. Dabei sind je nach Anwendung verschiedene Aspekte wie Speicher, CPU oder andere Auffälligkeiten interessant. Kubernetes kann fehlerhafte Container oder überlastete *Nodes* erkennen und reagieren (vgl. Kubernetes, 2018i). Existieren mehrere Instanzen einer Anwendung, wird ein *Loadbalancer* benötigt, der Nutzeranfragen nach vorgegebenen Algorithmen (z. B. Round-Robin oder Least connected) verteilt (vgl. Nginx, 2018b). Kubernetes bietet hierfür vorinstallierte Lösungen. Um die Zusammenarbeit von Anwendungen zu gewährleisten, müssen neue Container das bestehende Netz über Broadcast oder Registrierung bei einer zentralen Instanz eingebunden werden. Diesen Vorgang bezeichnet man als *Service Discovery* (vgl. Richardson, 2015).

2.6.1 Cluster

Ein *Kubernetes-Cluster* besteht aus mehreren Nodes. Abb. 2.7 zeigt den schematischen Aufbau eines Clusters und wird im Folgenden näher erläutert.

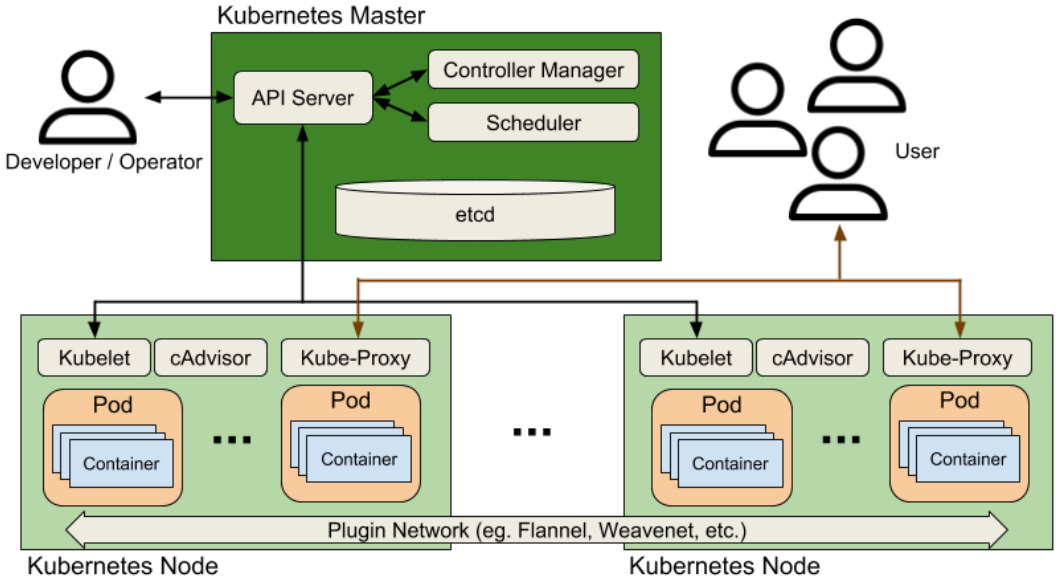


Abbildung 2.7: Schematischer Aufbau - Kubernetes Cluster in Anlehnung an Wikipedia und Bildmaterial von Lucy-g (2018)

Pod

Pods repräsentieren als kleinste deploybare Einheit einen laufenden Prozess im Cluster und enthalten einen bis wenige eng gekoppelte Container. Neben den Containern besitzt ein Pod Speicherressourcen, eine individuelle Netzwerkadresse und Instruktionen zur Ausführung der Container (vgl. Kubernetes, 2018m).

Master Node

Jedes Cluster besitzt eine *Master Node*, die für das Management des Clusters zuständig ist. Die Skalierung, das Scheduling und Updates werden zentral von dort gesteuert (vgl. Kubernetes, 2018j). Änderungen am Cluster müssen von hier durch den Entwickler (z. B. über das Command Line Tool *kubectl*) vorgenommen werden.

API Server
: Der *API Server* arbeitet hauptsächlich mit REST und dient als Schnittstelle zu allen anderen Diensten sowohl innerhalb der Master Node als auch außerhalb (vgl. Kubernetes, 2018h).

etcd
: *etcd* ist eine Key-Value Datenbank, in welcher der API Server bei Bedarf Informationen speichert (vgl. etcd, 2018).

Controller Manager
: Der *Controller Manager* überwacht den Status des Clusters und sorgt dafür, dass Elemente immer in den gewünschten Zustand zurückkehren (vgl. Kubernetes, 2018d).

Scheduler
: Der *Scheduler* entscheidet, wann und auf welcher Node Pods gestartet werden (vgl. Kubernetes, 2018f).

Node

Nodes - auch bekannt als Minions - können je nach Cluster physische oder virtuelle Maschinen sein (vgl. Kubernetes, 2018l). Sie sind auf das Management von Pods ausgelegt und nutzen dafür die folgenden Elemente:

Kubelet
: Das *Kubelet* läuft auf jeder Node und dient als Vermittler zwischen Master und anderen Nodes (vgl. Kannan und Marmol, 2015). Es nimmt - meistens durch den API Server - Pod-Spezifikationen in Form von YAML oder JSON entgegen, startet diese und überwacht deren Zustand (vgl. Kubernetes, 2018g).

cAdvisor

cAdvisor erkennt automatisch laufende Container und sammelt Daten über CPU-, Speicher- und Netzwerkauslastung, um sie anschließend zur Verfügung zu stellen (vgl. cAdvisor, 2018).

Kube-Proxy

Der *Kube-Proxy* fungiert als Loadbalancer und verteilt Nutzeranfragen auf die laufenden Pods (vgl. Kubernetes, 2018e).

Minikube

Minikube ist ein Kubernetes Cluster bestehend aus einer einzelnen Node, das sich gut zur lokalen Entwicklung von kleineren Deployments eignet (vgl. Kubernetes, 2018k).

Informatik Compute Cloud

> „Die [*Informatik Compute Cloud (ICC)*] ist eine vom AI Labor zur Verfügung gestellte Container Cloud Umgebung, in der Mitglieder des Departments Applikationen in Form von Docker-Containern betreiben können."
> Hüning und Behnke (2018b)

2.6.2 Kubernetes-Deployment

Ein *Kubernetes-Deployment* definiert den gewünschten Zustand einer Applikation, sodass der Controller Manager des Kubernetes Clusters die Anforderungen umsetzen kann. Das Beispiel 2.4 zeigt den Aufbau einer möglichen Konfiguration im YAML-Format. In jedem *Deployment-File* muss eine API-Version angegeben und durch "`kind: Deployment`" ergänzt werden. Die Metadaten beschreiben den Inhalt näher während "`spec`" die eigentliche Konfiguration enthält. Über "`replicas`" kann angegeben werden, wie viele Instanzen dieser Anwendung verfügbar sein sollten. Vergleichbar mit dem Konzept *Eventual consistency* bei Datenbanken wird hier jederzeit darauf hingearbeitet, dass die angegebene Anzahl immer wieder erreicht wird - dies bedeutet nicht, dass sie immer vorhanden ist (z. B. können wegen Ausfällen oder Updates auch zeitweise weniger oder mehr Instanzen gleichzeitig laufen) (vgl. Kubernetes, 2018c).

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Quellcode 2.4: Beispiel - Deployment-File

Image-Pull-Secret

Um in eigenen Deployments Images aus einer privaten Registry zu nutzen (z. B. AI Labor GitLab Registry) reicht ein "`docker login`" vor Ausführung der Deployment-Files nicht aus. Kubernetes kann sich stattdessen mit *Image-Pull-Secrets* authentifizieren, welche vorher generiert und in die Deployment-Files eingetragen wurden (vgl. Kubernetes, 2018o). Die Generierung und Speicherung der Secrets ist auf der CD unter "wiki/orchestration.wiki/Dokumentation/Imagepullsecrets" näher beschrieben.

2.6.3 Service

Docker-Container können über *localhost* kommunizieren, sofern sie sich im gleichen Pod befinden. Darüber hinaus weist Kubernetes jedem Pod eine *cluster-private-IP* zu, um den Nachrichtenaustausch zwischen Pods zu ermöglichen. Diese IP lässt sich auslesen. Da Pods jederzeit neu gestartet werden können, ist es nicht möglich, über diese IP zuverlässig Kommunikationswege zu anderen Anwendungen zu konfigurieren (vgl. Kubernetes, 2018b). Um dieses Problem zu lösen, können *Services* definiert werden, welche nicht direkt anhand ihrer IP-Adresse, sondern über Labels identifiziert werden können.

```
1  kind: Service
2  apiVersion: v1
3  metadata:
4    name: my-service
5  spec:
6    selector:
7      app: MyApp
8    ports:
9    - protocol: TCP
10     port: 80
11     targetPort: 9376
```

Quellcode 2.5: Beispiel - Service-Definition

Das Codebeispiel 2.5 definiert ein Service-Objekt namens "`my-service`", dessen Pods über TCP Port 9376 erreichbar sind. "`targetPort`" definiert den Port, über den ein Container Nachrichten annimmt und "`port`" ist der Service-Port, den andere Pods nutzen können, um den Service zu erreichen. Das Service-Objekt wird über REST an den API-Server geposted. Durch das Label können jederzeit die aktuellen IP-Adressen des Services (Endpoints) festgestellt werden, sodass die Kommunikation auch bei Änderungen möglich bleibt (vgl. Kubernetes, 2018p). Um einen Service stattdessen nach außen hin über eine feste IP erreichbar zu machen, können sogenannte *NodePorts* verwendet werden.

Durch "`kubectl describe svc <Servicename>`" lassen sich verfügbare Informationen zu einem Service anzeigen, um beispielsweise die aktuellen Endpoints zu inspizieren (vgl. Kubernetes, 2018b). Eine beispielhafte Ausgabe zeigt das Beispiel 2.6.

```
$ kubectl describe svc my-service
Name:                   my-service
Namespace:              default
Labels:                 <none>
Annotations:            <none>
Selector:               app=My-App
Type:                   ClusterIP
IP:                     10.0.162.149
Port:                   <unset> 80/TCP
TargetPort:             9736/TCP
Endpoints:              10.244.2.5:9376,10.244.3.4:9376
Session Affinity:       None
Events:                 <none>
```

Quellcode 2.6: Beispiel - Endpoints

2.6.4 Zugriff

Zugriff auf das Kubernetes Cluster, das nicht über NodePort verfügbar gemacht wurde, erhält man über das Command Line Interface *kubectl*. Zur Überprüfung, auf welches Cluster *kubectl* gerade zugreift, kann "`kubectl config view`" verwendet werden. Um eine Anwendung direkt zu erreichen, wird der Zugriff über die REST-API des API-Servers empfohlen (vgl. Kubernetes, 2018a). Im Allgemeinen wird "`kubectl proxy`" für den Zugriff verwendet. Eine ausführliche Beschreibung der resultierenden URLs befindet sich auf der CD unter "wiki/orchestration.wiki/home".

Eine weitere Möglichkeit ist die Portweiterleitung durch "`kubectl port-forward <podID> <port>:<port-to-be-forwarded>`". Die Applikation kann anschließend über "`localhost:<port>`" erreicht werden (vgl. Kubernetes, 2018q).

2.7 GitLab

GitLab ist ein Tool zur Versionsverwaltung von Projekten durch Git. Neben einem integrierten Wiki bietet es Projektmanagementtools wie das Anlegen und zuweisen von Aufgaben (Issues). Zusätzlich gibt es verschiedene Möglichkeiten zur Verwaltung von Images und Continuous Integration.

2.7.1 Continuous Integration

Continuous Integration (CI) integriert regelmäßig Änderungen im Code und testet diese. Dadurch können Fehler früh erkannt und die sogenannte *Big-Bang-Integration* verhindert werden (vgl. Fowler, 2006). GitLab bietet dafür ein "`.gitlab-ci`"-File, welches der Definition einzelner Integrationsschritte (Pipeline) dient. Darin können Abhängigkeiten geladen und anschließend der Code getestet werden. Sogenannte *GitLab-Runner* führen die CI-Schritte (z. B. Build und Test) aus. Nach erfolgreichem Durchlauf kann die Änderung in die "Mainline" - meist Master oder Develop-Branch (siehe Atlassian, 2018, GitFlow) - integriert werden. Neben der Möglichkeit, im CI-File direkt Variablen zu definieren, können diese als *Secret Variables* im Repository hinterlegt werden, sodass Passwörter und andere sicherheitskritische Elemente nicht öffentlich sichtbar sind. Variablen können mit "`$VARIABLENNAME`" aufgerufen werden (vgl. GitLab, 2018a).

```
1 image : maven:3-jdk-8-alpine
2
3 stages:
4   - test
5 maven-test:
6   stage: test
7   script:
8    - mvn test
```

Quellcode 2.7: Beispiel - GitLab-CI

Das Codebeispiel 2.7 zeigt eine einfache CI-Definition für ein Maven-Java-Projekt. Der Gitlab-Runner kann Docker-Image nutzen (vgl. Lenzo, 2016). Zur Ausführung von Maven- oder Java-spezifischen Befehlen muss als "`image`" ein entsprechendes Docker-Image angegeben werden. *Stages* definieren verschiedene Phasen, welche einzeln abgearbeitet werden. In diesem Fall gibt es nur die "`test`"-stage. Einer Stage können

verschiedene *Jobs* zugeordnet werden. Der "`maven-test`"-Job führt im "`script`"-Teil den Test-Befehl von Maven aus. Sind diese Tests erfolgreich, kommt die Pipeline ohne Fehler zum Ende.

2.7.2 Continuous Deployment

Continuous Deployment (CD) geht über die CI hinaus. Neben den beschriebenen Integrationsschritten führt jede erfolgreiche Pipeline zu einem automatischen Deployment der Applikation in eine Produktionsumgebung, wie zum Beispiel Kubernetes (vgl. Ramos, 2016).

Trigger

Es befindet sich selten der gesamte relevante Code für eine Anwendung in einem Repository. Um die CI-Pipeline von anderen Repositories zu starten, können sogenannte *Trigger* eingesetzt werden. Das Beispiel 2.8 zeigt den Einsatz eines Triggers in GitLab. Jede Pipeline hat eine eigene Adresse (siehe "`PIPELINE`") über die sie per "`curl`"-Befehl mit einem vorher generierten Token (siehe "`TRIGGER`") gestartet werden kann. Das Token muss im aufrufenden Repository als Secret Variable hinterlegt werden (vgl. GitLab, 2018c).

```
1  image : maven:3-jdk-8-alpine
2
3  variables:
4    HAW_GITLAB: https://gitlab.informatik.haw-hamburg.de
5    PIPELINE: $HAW_GITLAB/api/v4/projects/2117/trigger/pipeline
6  stages:
7    - trigger-build
8  trigger_build:
9    only:
10     - master
11   script:
12     - "curl -X POST -F variables[service]=$CI_PROJECT_NAME: /
13       $CI_COMMIT_SHA -F token=$TRIGGER -F ref=master $PIPELINE"
14   stage: trigger-build
```

Quellcode 2.8: Beispiel - Trigger

2.7.3 GitLab-Registry

Die *GitLab-Registry* dient der Speicherung und Bereitstellung von Images. Über die GitLab-CI und Terminalbefehle können Docker-Images in die Registry gepushed oder gepulled werden. Im Hinblick auf automatische Deployments bietet es sich an, nach erfolgreichen Testläufen ein Docker-Image zu erstellen und in der Registry zu hinterlegen, wie im Beispiel 2.9. Um ein Image in eine Docker Registry pushen zu können, muss man sich zuerst mit "`docker login`" authentifizieren. "`docker build`" erstellt das Image, das mit "`docker push`" anschließend in die gewünschte Registry gepushed wird (vgl. Pundsack, 2016).

```
image : docker-hub.informatik.haw-hamburg.de/icc/docker-dind

variables:
  DOCKER_HOST: tcp://localhost:2375
  DOCKER_REGISTRY: docker-hub.informatik.haw-hamburg.de
  DOCKER_GROUP: group
  IMAGE_NAME: example

services:
 - docker-hub.informatik.haw-hamburg.de/icc/docker-dind

stages:
  - dockerize

dockerize:
  stage: dockerize
  script:
    - docker login -u $USER -p $PASSWORD $DOCKER_REGISTRY
    - docker build -t $IMAGE_NAME .
    - docker push $DOCKER_REGISTRY/$DOCKER_GROUP/$IMAGE_NAME
```

Quellcode 2.9: Beispiel - GitLab-Registry

3 Analyse und Spezifikation

3.1 Stakeholder

Im Zentrum der Fragestellung steht das Deployment des HAW-Logistics-System und der Bank, sodass Stakeholder bezüglich der tatsächlichen Nutzung dieser Systeme an dieser Stelle nicht in Betracht gezogen werden.
Die Anwendungen sind für ein Wahlpflichtprojekt ab dem 4 Semester in den Informatikstudiengängen der HAW Hamburg vorgesehen. Die Lehrveranstaltung wird von Professoren und Professorinnen des Studiengangs Angewandte Informatik durchgeführt werden, da der Fokus auf Inhalten der Vorlesungen "Software Engineering" und "Architektur von Informationssystemen" liegt. Üblicherweise wird die Veranstaltung zusätzlich von einer wissenschaftlichen oder studentischen Hilfskraft begleitet. Daraus ergeben sich die folgenden Nutzergruppen:

Lehrkräfte

Professor/in

Als Leiter/in der Veranstaltung trifft ein Professor/eine Professorin die maßgeblichen Entscheidungen sowohl über den Verlauf und Inhalt des Wahlpflichtfaches als auch über das Bestehen der teilnehmenden Studierenden. Umfassende Einblicke in das Fach "Software Engineering" und damit verbundene Technologien sind vorhanden, wie auch jahrelange Erfahrung im Umgang mit beziehungsweise der Lehre von Studierenden.

Mitarbeiter/in

Studentische und wissenschaftliche MitarbeiterInnen befinden sich in der Regel im Masterstudiengang oder arbeiten an einer Dissertation. Sie besitzen umfangreiche Informatikkenntnisse, um sich im Zweifelsfall mit neueren Technologien

in kurzer Zeit vertraut zu machen, was im Rahmen ihrer Anstellung erwartet wird. Neben Betreuung der Studierenden werden unter Umständen Zuarbeiten für den Professor/die Professorin nötig.

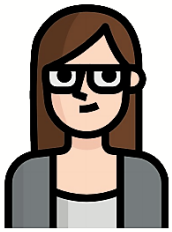
Studierende

Bis zum 4. Fachsemester haben Studierende aller Informatikstudiengänge an der HAW mindestens eine Lehrveranstaltung zum Thema "Software Engineering" besucht. Die Studierenden beherrschen danach grundlegende Kompetenzen im Bereich Projektmanagement und im Umgang mit Entwurfsmustern. Sie sind ebenfalls in der Lage sich weitere Kompetenzen eigenverantwortlich anzueignen. Das Fachwissen von Studierenden der höheren Semester unterscheidet sich aufgrund der unterschiedlichen Ausrichtung der Studiengänge stärker. Es wird im Rahmen des Wahlpflichtfachs erwartet, dass fehlende Kenntnisse selbstständig nachgearbeitet werden, und bedürfen daher keiner weiteren Beachtung (vgl. HAW, 2016a,b,c).

3.1.1 Persona

Je Nutzergruppe wurde basierend auf den vorhandenen Informationen eine Persona erstellt. Auf eine Unterscheidung zwischen Professor/in und Mitarbeiter/in wurde an dieser Stelle verzichtet, da diese eng zusammenarbeiten. MitarbeiterInnen sind fachlich als eine Mischung aus Professor/in und Student/in zu sehen, sodass ihre Ansprüche erfüllt werden, sobald beide Nutzergruppen im Projektaufbau beachtet werden. Folgende Personas wurden mit Xtensio erstellt. Referenzen für genutztes Bildmaterial finden sich im Literaturverzeichnis (Hernandez (2018); Freepik (2018); StickPNG (2018); Monkik (2018)).

Maya



"Zielstrebige Person mit wenig Zeit für schlechten Code"

Alter: 20+
Job: Student/in der Angewandten Informatik HAW
Lokation: Hamburg

Logisches
Technikaffin
Ehrgeizig

Skills

Ziele

- Weiterentwicklung des HLS/Bank-Systems
- Erfolgreicher Studienabschluss

Probleme/Risiken

- Motivationsprobleme
- Langsames Vorankommen durch
  - unverständliche Dokumentation
  - umständlichen Code
  - Aufwändige Prozesse
- Zeitknappheit

Persönlichkeit

Introversion	Extroversion
Denken	Fühlen
Faktenbasiert	Intuition

Bio

Maya hat bereits 3-4 Semester Programmiererfahrung und beschäftigt sich im Rahmen eines Wahlpflichtmoduls mit dem HLS-System. Java, GitLab und Gradle sind ihr wohlbekannt. Erste Erfahrungen mit Angular4/5 und C# wurden bereits gesammelt, sind aber noch weit vom selbstbewussten Umgang entfernt. Mit REST-Schnittstellen ist sie bekannt und lernt diese gerade näher kennen - ebenso den Umgang mit der GitLab-CI.

Sie möchte das Wahlpflichtfach möglichst unfallfrei und ohne ausufernden Zeitaufwand hinter sich bringen, denn es gibt noch weitere Fächer die ihre Aufmerksamkeit fordern.

Stefan

"Wünscht sich gute Studenten am Ende des Semesters"

Alter: 45+
Job: Professor an der HAW
Lokation: Hamburg

Fachwissen
Lehre
Studentennah

Ziele

- Spaß an der Lehre und Forschung
- Erfolgreiche Durchführung des Wahlpflichtfachs

Probleme/Risiken

- Hoher Wartungsaufwand des Wahlpflicht-Codes
  - Umständliche Installationsprozesse
  - unvollständige Dokumentation
- Hilflose Studenten
  - Begleitungsaufwand
- Andere Verpflichtungen

Skills

Persönlichkeit

Introversion	Extroversion
Denken	Fühlen
Faktenbasiert	Intuition

Bio

Stammt aus Nordrhein-Westfalen, hat in Ulm studiert und arbeitet seit 8 Jahren an der HAW Hamburg als Professor für Software-Engineering und Software-Architektur.
Leitet gemeinsam mit Prof. Ulrike Steffens das HAWAI-Labor.

Abbildung 3.1: Personas - Studentin und Professor

3.2 Value-Proposition-Canvas

Der Value-Proposition-Canvas in Abb. 3.2 beider Personas zeigt, dass die meisten Probleme durch eine klare Struktur und Dokumentation des Systems zu beheben sind.

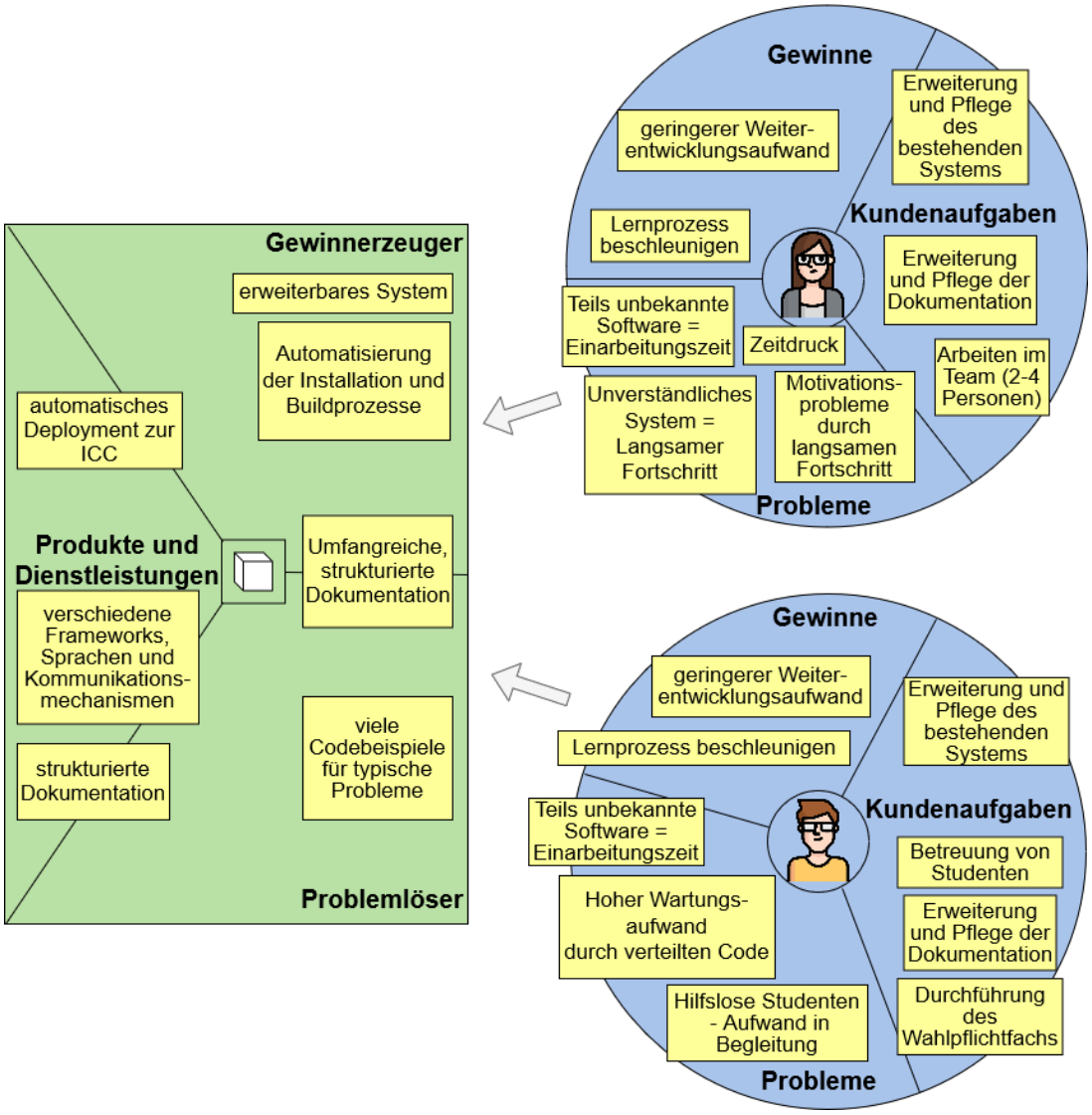


Abbildung 3.2: Value-Proposition-Canvas - Studentin und Professor

3.3 User-Stories

Anhand der Personas und dem zugehörigen Value-Proposition-Canvas konnten folgende User-Stories (siehe 2.2.4) - ohne Anspruch auf Vollständigkeit - ermittelt werden:

User Group: Professor/in

Als Professor/in möchte ich ...

- die Projekte flexibel erweitern und Gruppen zuweisen können.
- fehlende Codeelemente oder Dokumentation ohne großen Aufwand nachpflegen können.
- alle Informationen für die Ausführung der Anwendung und späteren Zugriff darauf haben.
- den Studierenden möglichst unkompliziert die Zusammenhänge zwischen den Projekten erklären können.

User Group: Student/in

Als Student/in möchte ich ...

- mich möglichst schnell in Aufbau und Zusammenhänge der bestehenden Projekte einarbeiten.
- die Projekte möglichst flexibel weiterentwickeln können.
- Anlaufstellen für Eigenrecherchen zu mir noch unklaren Themen bezüglich der Projekte haben.
- die Projekte lokal bauen und ausführen können.
- die Projekte manuell dockerisieren und in die ICC deployen können.
- die Projekte automatisch dockerisieren und in die ICC deployen können.

3.4 Anforderungen

Im Folgenden werden die User-Stories in drei Kategorien aufgeteilt und diskutiert.

Erweiterung und Wartbarkeit

Als Lehrkraft/ Student/in möchte ich Projekte flexibel weiterentwickeln können.

Flexibilität erreicht man besonders durch Entkopplung. Die Reduzierung von Abhängigkeiten untereinander und klare Definition von Schnittstellen sollten somit beim Aufbau einer Infrastruktur im Vordergrund stehen.

Als Lehrkraft möchte ich Projekte Gruppen für Aufgaben zuordnen können.

Um Projekte verschiedenen Gruppen zuordnen zu können, damit diese produktiv daran arbeiten können, sollten sie nur eine geringe Zahl an Abhängigkeiten besitzen. Dies entspricht den Anforderungen, die bereits oben beschrieben sind.

Als Lehrkraft möchte ich die Projekte warten können.

Zur Fehlererkennung und -behebung ist eine funktionierende CI und die Verwendung von Versionskontrollmechanismen nötig. Updates und andere Neuerungen sollten nach Möglichkeit ohne großen Aufwand möglich sein. Hierzu bietet sich die zentrale Speicherung von Konfigurationen an.

Aufbau und Dokumentation

Als Lehrkraft / Student/in möchte ich den Aufbau verstehen und erklären können.

Um schnell den Aufbau eines Systems zu verstehen ist eine klare Struktur in der Dokumentation essenziell. Neben der Dokumentation von Einzelprojekten ist auch die Wiedergabe von Zusammenhängen zwischen beteiligten Repositories wichtig.

Als Student/in möchte ich Ansätze für weitere Recherchen haben.

Neben den wichtigsten Hinweisen in der Dokumentation zu verschiedenen Themengebieten, sollte Studierenden die Möglichkeit gegeben werden, eigene Recherchen zu betreiben. Als Ansatzpunkt sollten in der Dokumentation hilfreiche Links und “FAQ“ zur Verfügung gestellt werden.

Als Lehrkraft/ Student/in möchte ich Informationen zu den Projekten finden können.
Wichtige Informationen wie Setup und Hinweise zur Ausführung von Projekten oder Skripten sind übersichtlich in die Dokumentation einzupflegen.

Ausführung und Deployment

Als Student/in möchte ich die Projekte lokal ausführen können.
Um die Projekte lokal ausführen zu können, ist das Starten mehrerer Programme nötig. Um den Prozess zu vereinfachen bietet sich die Erstellung von docker-compose-Files an, die je nach Anforderung verschiedene Images starten.

Als Student/in möchte ich die Projekte manuell in die ICC/ lokal deployen können.
Projekte sollen mit Kubernetes manuell deploybar sein. Dies umfasst die Möglichkeit, direkt mit der Informatik Compute Cloud oder lokal mit Minikube zu arbeiten. Dafür müssen entsprechende Deployment-Files und Docker-Images zur Verfügung gestellt werden. Um Projekte zu containerisieren, müssen Dockerfiles zur Verfügung gestellt werden. Damit die Erstellung möglichst einfach gestaltet wird, sollten multi-stage builds verwendet und so alle nötigen Befehle innerhalb des build-Prozesses für das Image ausgeführt werden. Es sollen keine Vorbedingungen erfüllt werden müssen.

Als Student/in möchte ich die Projekte automatisch in die ICC deployen können.
Das Gesamtsystem soll automatisch neu in die ICC deployed werden, wenn die CI für eines der Projekte erfolgreich durchgelaufen ist. Hierfür bietet sich die Nutzung von Triggern an, die eine gesonderte CI-Pipeline für das Deployment anstößt. Im Rahmen der CI müssen die Projekte dazu automatisch containerisiert und in die jeweilige GitLab-Registry gespeichert werden.

3.4.1 Funktionale Anforderungen

1. Projekte als Repositories im AI-Labor GitLab zur Verfügung stellen.
2. Aufteilung der Projekte auf mehrere Repositories und vorzugsweise Verwendung von Microservice-Architekturen.
3. GitLab-CI für jedes Repository:
   3.1. Testen
   3.2. Dockerisieren
   3.3. In GitLab-Registry speichern
   3.4. Mit jeder erfolgreichen Pipeline das Deployment in die ICC triggern
4. Konfigurations-Files je Projekt gesammelt in separatem Ordner.
5. Projekte dokumentieren:
   5.1. Klare Struktur
   5.2. Setup-Dokument mit Informationen zur Installation an zentraler Stelle
   5.3. Hinweise zur Ausführung der Projekte
   5.4. Je Repository ein FAQ-Dokument
6. Erstellung von docker-compose-Files für unterschiedliche Szenarien und anschließende Dokumentation.
7. Erstellung von Dockerfiles je Projekt.
8. Erstellung von Deployment- und Service-Files für das Kubernetes-Deployment je Repository.

3.4.2 Qualitätsanforderungen

1. Die Projekte sollen auf Linux, Mac und Windows lauffähig und installierbar sein.
2. Docker-Images sollen ohne vorige Ausführung von build-Befehlen erstellbar sein.

3.4.3 Randbedingungen

1. Die Projekte müssen für das Deployment in die ICC per CI-Pipeline das aktuellste kubectl-Image, das vom AI Labor zur Verfügung gestellt wird, nutzen.
2. Damit Images in privaten Registries für lokale Deployments erreichbar sind, müssen auf jeder Maschine, die diese Deployments durchführen soll, Image-Pull-Secrets hinterlegt werden.
3. Damit Images in privaten Registries in anderen GitLab-CI-Pipelines verwendet werden können, müssen die Repositories durch Deploy-Keys Zugriff erhalten.

3.5 Spezifikation

Im Folgenden sind vier Use-Cases mit je einem BPMN Modell erarbeitet, welche die Kernfunktionalitäten des Systems verdeutlichen. Die BPMN-Modelle sind mit Camuda umgesetzt worden. Anfragen zu GitLab-Registries und Docker-Hub sind beispielhaft dargestellt, um die Abhängigkeiten zu verdeutlichen. Die Notation muss nicht der tatsächlichen Reihenfolge entsprechen.

Docker-Compose

Der Use-Case zeigt das Starten der vollständigen Applikation mit docker-compose (Tab. 3.1 und Abb. 3.3). Es wird erwartet, dass alle Anwendungsrepositories als Image lokal vorhanden sind und müssen, falls nötig, aus den GitLab-Registries gepulled werden. Die übrigen Images werden bei Bedarf von Docker-Hub geladen. Neben dem Start des Gesamtsystems sind noch folgende Kombinationen denkbar:

- NUR Bank: bank-backend, bank-frontend
- NUR HLS: hls - sinnvoller, wenn das HLS in einzelne Microservices aufgeteilt werden würde
- NUR Benötigte Nebensysteme: neo4j, rabbitmq, postgres

Use Case	Starten des Systems mit docker-compose.	
Anwendungs-kontext	Der User nutzt ein docker-compose File um die Anwendungen lokal zu starten.	
Ebene	Subfunktion, Überblick	
Primärakteur	Student	
Stakeholder	**Stakeholder**	**Interessen**
	Student	Testen von Änderungen einer Anwendung. Testen des Zusammenspiels des Systems.
	Professor	s.o. und schnelle Demonstration des Systems.
Vor-bedingungen	Alle Anwendungsimages sind lokal vorhanden. Falls die Images für neo4j, rabbitmq oder postgres dies nicht sind, ist eine aktive Internetverbindung nötig.	
Invarianten	Das System läuft lokal in Containern und ist erreichbar.	
Nach-bedingungen	Codeänderungen in Bezug auf das Zusammenspiel der Anwendungen können lokal getestet werden	
Trigger	-	
Beschreibung	**Schritt**	**Aktion**
	1	parallel werden Neo4j, RabbitMQ und Postgres in Containern gestartet
	2	Sobald Postgres gestartet ist → das bank-backend starten
	3	Sind alle Container hochgefahren → das hls-System starten
	4	Sobald das bank-backend läuft → das bank-frontend starten
Fehlerfälle	**Schritt**	**Fehlerfall**
	1	Docker-Hub ist nicht erreichbar: Abbruch

Tabelle 3.1: docker-compose - Starten des Systems

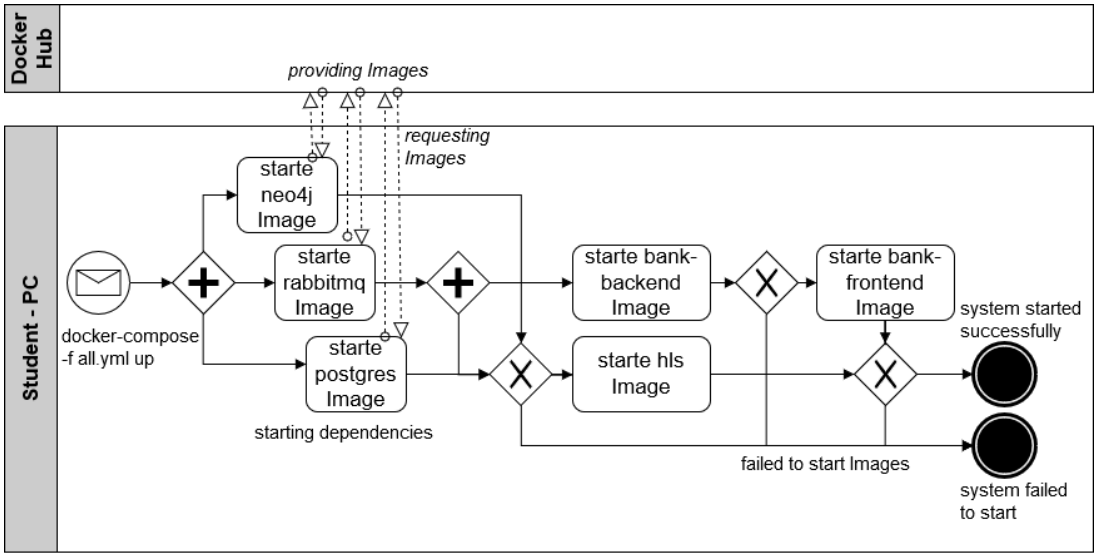


Abbildung 3.3: BPMN Modellierung - Start des Systems durch docker-compose

Manuelles Deployment

Soll das System manuell über die Command-Line deployed werden, hat ein Nutzer/eine Nutzerin zwei Möglichkeiten: das Deployment in die ICC (Tab. 3.2 und Abb. 3.4) oder zu Minikube (Tab. 3.3 und Abb. 3.5). Beim Deployment in die ICC werden die Images aus der GitLab-Registry genutzt und müssen dementsprechend dort vorhanden und erreichbar sein. "kubelogin" ist ein Skript, das von der HAW zur Verfügung gestellt wird und dem Nutzer/der Nutzerin die Möglichkeit gibt, sich mit der ICC zu verbinden. Das Deployment zu Minikube setzt voraus, dass alle Anwendungsimages lokal vorhanden sind. Sie können bei Bedarf aus der entsprechenden Registry gepulled werden. Minikube muss zudem in einer VM gestartet sein.
Da beide Deployments unterschiedliche Images verwenden, werden unterschiedliche Deployment-Files benötigt. Dies bedeutet zusätzlichen Aufwand, da ein erfolgreiches Deployment zu Minikube nicht gleichzeitig garantiert, dass für die ICC das Gleiche gilt. Die Arbeit mit Minikube ermöglicht die autarke Weiterentwicklung bei Problemen mit der ICC oder der Internetverbindung.

Use Case	Manuelles Deployment in die ICC	
Anwendungs-kontext	NutzerInnen können das System über die Kommandozeile in die ICC deployen.	
Ebene	Subfunktion, Überblick	
Primärakteur	Student	
Stakeholder	**Stakeholder**	**Interessen**
	Student	Testen des Deployments unabhängig von GitLab.
	Professor	s.o.
Vor-bedingungen	kubelogin und eine aktive Internetverbindung muss lokal vorhanden sein.	
Invarianten	Das System läuft in der ICC	
Nach-bedingungen	Das Deployment Images aus der Registry getestet werden	
Trigger	-	
Beschreibung	**Schritt**	**Aktion**
	1	Der Nutzer authentifiziert sich gegenüber der ICC mit kubelogin und baut eine Verbindung auf.
	2	Händisch werden alle Anwendungen mit “kubectl apply -f <deploymentfile.yaml>“ deployed.
	3	Die ICC läd die benötigten Images aus der GitLab-Registry und Docker-Hub und startet diese in Pods.
Erweiterungen	**Schritt**	**Verzweigende Aktion**
	2a	Deployment des HLS-Systems
	2b	Deployment des Bank-Backends
	2c	Deployment des Bank-Frontends
Fehlerfälle	**Schritt**	**Fehlerfall**
	1	Login schlägt fehl (falsche Login-Daten, ICC nicht erreichbar): Abbruch
	2	Ein Deployment schlägt fehl (Syntaxerror, ICC nicht erreichbar): Abbruch
	3	Images können nicht geladen werden (Authentifizierungsprobleme, Image nicht vorhanden, falsche Imagenamen): Abbruch, muss manuell nachgeschaut und gelöst werden

Tabelle 3.2: Manuelles Deployment - ICC

Use Case	Manuelles Deployment zu Minikube	
Anwendungs-kontext	User haben die Möglichkeit das System über die Kommandozeile zu Minikube zu deployen.	
Ebene	Subfunktion, Überblick	
Primärakteur	Student	
Stakeholder	**Stakeholder**	**Interessen**
	Student	Deployment zu Minikube, Testen von Änderungen eines/mehrerer Deploymentfiles oder einer Anwendung
	Professor	s.o.
Vor-bedingungen	Die Anwendungsrepositories müssen lokal vorhanden und Minikube installiert sein.	
Invarianten	Das System läuft auf Minikube	
Nach-bedingungen	Das Deployment konnte mit lokalen Images getestet werden.	
Trigger	-	
Beschreibung	**Schritt**	**Aktion**
	1	Dockerisieren einer/mehrerer veränderten Anwendung/en.
	2	Starten von Minikube.
	3	Deployment der Anwendungen nacheinander über die Kommandozeile.
Erweiterungen	**Schritt**	**Verzweigende Aktion**
	3a	Deployment des HLS-Systems
	3b	Deployment des Bank-Backends
	3c	Deployment des Bank-Frontends
Fehlerfälle	**Schritt**	**Fehlerfall**
	1	Dockerisierung schlägt fehl (Syntaxerror): Abbruch
	2	Minikube startet nicht (Konfigurationsfehler): Abbruch
	3	Das Deployment einer oder mehrerer Anwendungen schlägt fehl(Syntaxerror,...): Abbruch
	4	Minikube kann benötigte Images nicht laden(Docker-Hub nicht erreichbar): Im Prozess selbst nicht erkennbar, muss manuell nachgeprüft und gelöst werden

Tabelle 3.3: Manuelles Deployment - Minikube

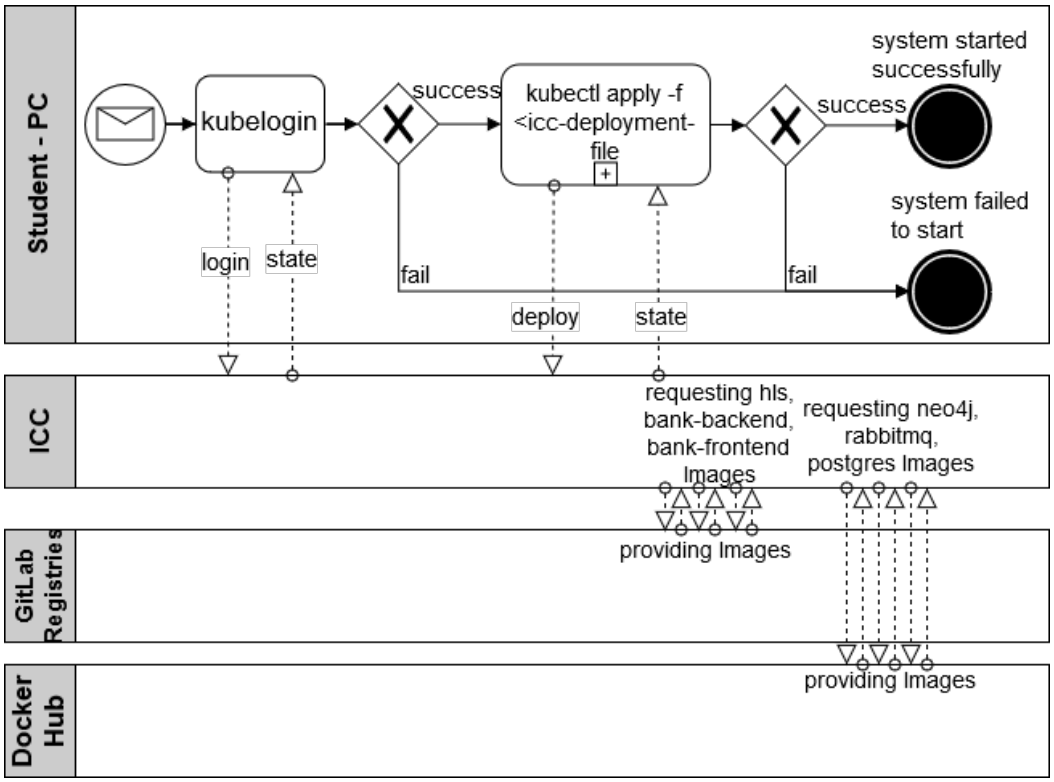


Abbildung 3.4: BPMN Modellierung - Manuelles Deployment in die ICC

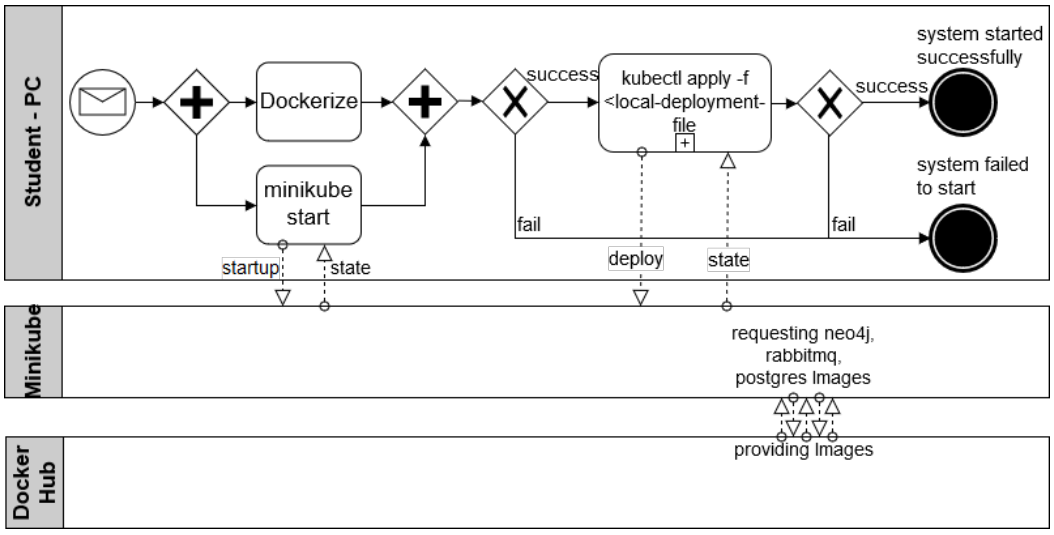


Abbildung 3.5: BPMN Modellierung - Manuelles Deployment zu Minikube

Automatisches Deployment

Das automatische Deployment spielt sich überwiegend in der CI ab. Pushed ein Entwickler auf ein Anwendungsrepository, so wird die Pipeline gestartet und wie in Abb. 3.6 durchgeführt. Die Deployment-Files werden in einem Repository zentral gespeichert und von dort in den Namespace des Repositories deployed. Da die Repositories intern mit Namespaces in der ICC synchronisiert werden (vgl. Hüning und Behnke, 2018a), liegt diese Lösung nahe, um Probleme durch unterschiedliche Namespaces zu vermeiden. Die ICC benötigt für das Deployment die Images der Anwendungen aus den jeweiligen Registries (In Abb. 3.6 als ein Pool notiert) und die Images der benötigten Programme von Docker-Hub. Das HLS-System benötigt vor der Testing-Stage noch eine build-Stage und das Frontend der Bank nutzt ein Image aus der Registry zur Dockerisierung. Diese Abläufe sind im Use-Case unter "Erweiterungen" erläutert.

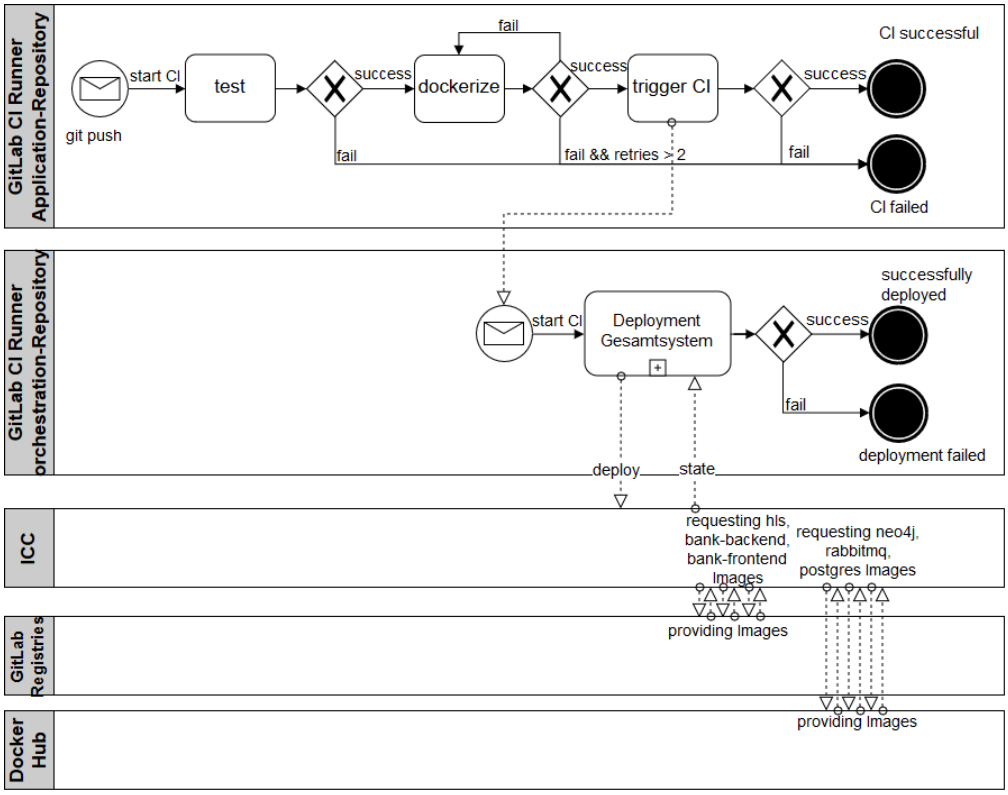


Abbildung 3.6: BPMN Modellierung - Automatisches Deployment zur ICC

Use Case	Durchführung einer typischen CI-Pipeline	
Anwendungs-kontext	Nach einem push durch einen Nutzer auf ein Anwendungsrepository wird eine CI ausgeführt.	
Ebene	Hauptaufgabe, Überblick	
Primärakteur	Student	
Stakeholder	**Stakeholder**	**Interessen**
	Student	mit einem Befehl(git push) Pipeline anstoßen, Anwendung testen,dockerisieren und in die ICC deployen.
	Professor	s.o. schnelle Demonstration des Systems.
Vor-bedingungen	Aktivierte GitLab-Runner, ICC online, GitLab Registries verfügbar, Git-Hub verfügbar	
Invarianten	Die CI-Pipeline im Anwendungs- und im orchestration-Repository müssen erfolgreich durchlaufen. System in die ICC deployed und erreichbar.	
Nach-bedingungen	Das System ist getestet, versioniert, als Image in der GitLab-Registry verfügbar.	
Trigger	Push auf ein Anwendungsrepository	
Beschreibung	**Schritt**	**Aktion**
	1	Anwendung wird getestet.
	2	Anwendung wird dockerisiert + in GitLab-Registry gespeichert.
	3	CI-Pipeline im orchestration-Repository wird getriggered.
	4	Anwendungen werden je mit “kubectl apply“ in die ICC deployed.
	5	Benötigte Images von Docker-Hub/GitLab-Registries geladen
Erweiterungen	**Schritt**	**Verzweigende Aktion**
	0a	Das HLS-System wird vor Schritt 1 gebaut
	2a	Das bank-frontend benötigt zum Dockerisieren das frontend-ci-image aus der GitLab-Registry.
Fehlerfälle	**Schritt**	**Fehlerfall**
	0a	Der Build des HLS-Systems schlägt fehl(Syntaxerror): Neuer Versuch oder Fehlschlagen und Abbruch der Pipeline
	1	Die Tests schlagen fehl: Fehlschlagen und Abbruch der Pipeline
	2	Die Dockerisierung schlägt fehl(Fehlendes Image, Syntaxerror): Neuer Versuch oder Fehlschlagen und Abbruch der Pipeline.
	3	Der curl-Command schlägt fehl(Ziel nicht erreichbar, Syntaxerror): Neuer Versuch oder Fehlschlagen und Abbruch der Pipeline
	4	Ein oder mehrere Deployments schlagen fehl(Syntaxerror, ICC unerreichbar): Fehlschlagen und Abbruch der Pipeline
	5	Images können nicht geladen werden (Authentifizierungsprobleme, Image nicht vorhanden, falsche Imagenamen): Kann in der CI-Pipeline nicht abgefangen werden und muss manuell nachgeschaut und gelöst werden

Tabelle 3.4: Automatisches Deployment - Zur ICC über die GitLab-CI

4 Architektur und Planung

4.1 Architektur

4.1.1 Ist-Zustand

Das HAW-Logistics-System und die Bank-Anwendung liegen als monolithische Applikationen mit vorhandener CI als Repositories im AI-Labor GitLab vor. Das HLS-System ist in zwei Varianten verfügbar. Eine ist bereits für das Deployment in die ICC konfiguriert und die Andere besitzt eine zusätzliche Buchhaltungskomponente, die mit der Bank-Anwendung kommuniziert.
Aufgaben:

- Beide HLS-Varianten müssen zu einem System migriert und dockerisiert werden.
- Die Bank-Applikation besteht bereits aus zwei Modulen (Back- und Frontend) und kann dadurch unkompliziert in zwei Microservices geteilt werden, um die Projekte zu entkoppeln.

4.1.2 Soll-Zustand

Ausgangspunkt für eine übersichtliche und flexible Projektstruktur ist eine klare, nachvollziehbare Aufteilung der Repositories. Alle Repositories werden in einer Gruppe verwaltet. Es wird zwischen Anwendungs- und Hilfsrepositories unterschieden. Hilfsrepositories beinhalten Images oder Skripte, welche die Arbeit am Projekt unterstützen sollen. Anwendungsrepositories beinhalten die Applikationen selbst.

Abb. 4.1 zeigt den Projektaufbau als Deploymentdiagramm mit zusätzlichen Informationen über den Einfluss des scripts-Repositories und wird im Folgenden näher erläutert.

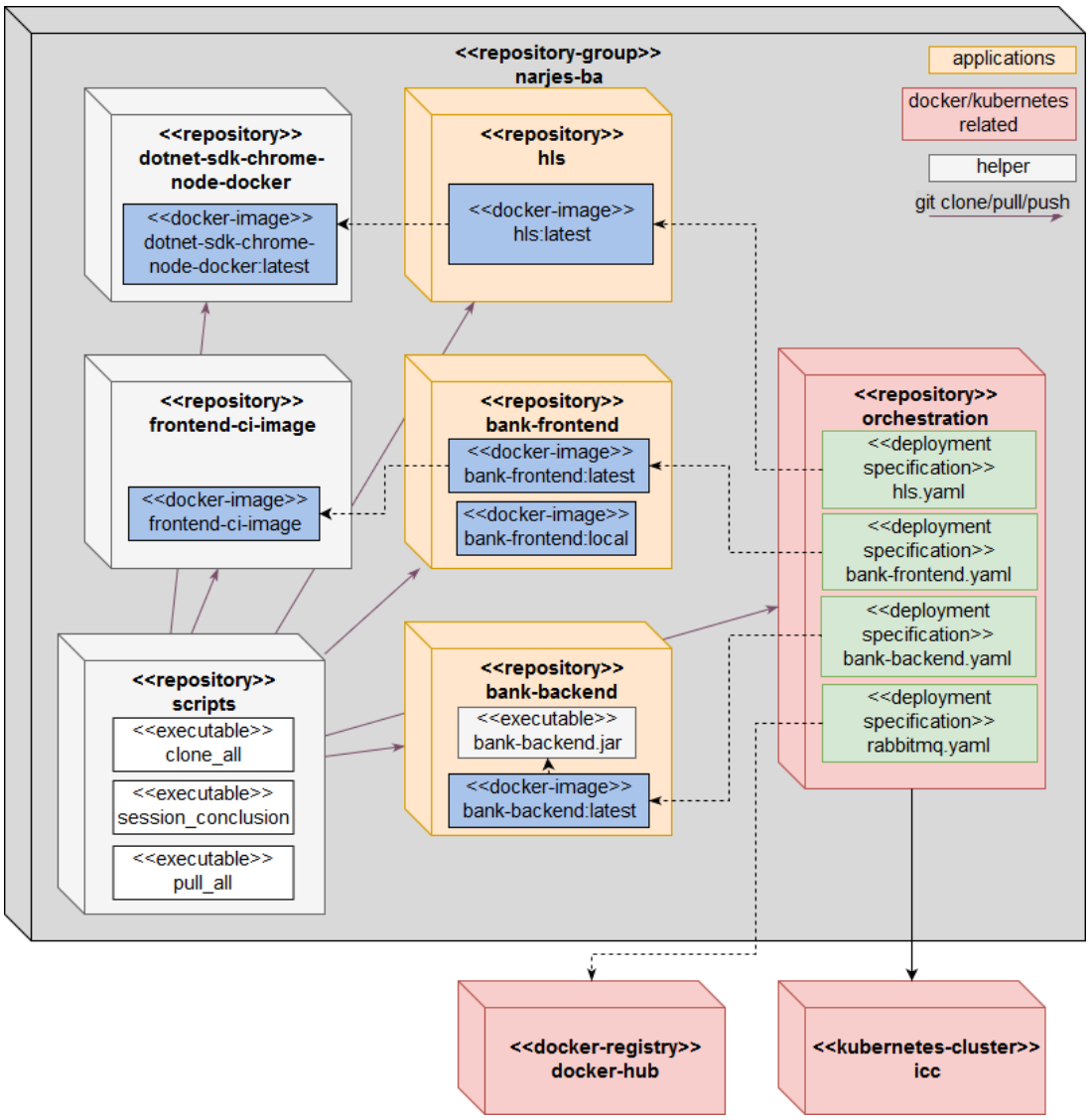


Abbildung 4.1: Projektaufbau

scripts Dieses Repository hat keinen direkten Projektbezug. Es beinhaltet projektübergreifende Informationen als Wiki und Skripte, die das Setup sowie die Interaktion mit den Git-Repositories vereinfachen. Es bildet damit eine zentrale Anlaufstelle zum strukturierten Arbeiten im Projekt, ohne auf inhaltliche Aspekte der Projekte Einfluss zu nehmen. Die Skripte müssen im bash- und shell-Format zur Verfügung stehen, um auf allen angeforderten Plattformen lauffähig zu sein.

frontend-ci-image In diesem Repository wird ein Node-Image gespeichert, das für die CI von Angular-Frontend-Projekten verwendet werden kann. Damit besitzt der Nutzer/die Nutzerin die volle Kontrolle über die npm-Versionen.

dotnet-sdk-chrome-node-docker Für das HLS-Projekt wird ein .NET-Image für die CI benötigt, das ursprünglich in einem anderen studentischen Repository verfügbar war. Um von externen Elementen unabhängig zu sein, wird dieses Image in einem eigenen Repository gespeichert.

orchestration Dieses Repository ist für das Kubernetes-Deployment zuständig. Das Repository beinhaltet alle Deployment-Files und die docker-compose Dokumente. Das Repository dient als Sammelstelle für alle Elemente, die direkt mit dem Deployment zu tun haben. Die docker-compose-Files dienen der Orchestrierung von lokal laufenden Containern und bieten damit die Möglichkeit, ohne ein Deployment das Zusammenspiel der Container zu testen.

Die Anwendungen kommunizieren über REST und RabbitMQ. Abb. 4.2 zeigt Schnittstellen und Kommunikationswege in Bezug auf die geplante Deploymentstruktur. Das Diagramm nutzt verschiedene UML-Elemente zur Notation der Schnittstellen und die aus dem Grundlagenkapitel bereits bekannte Darstellung von Pods und Containern, um die relevanten Informationen in einem Diagramm zu bündeln.

Mit Ausnahme der Kommunikationsverbindung zur Bank-Anwendung über RabbitMQ liegt das HLS-System bereits in dieser Form vor. Im HLS-System werden Back- und Frontend im Rahmen dieses Projekts nicht getrennt, da dieses System sich komplizierter gestaltet als die Bank und so möglicherweise größere Fehlerquellen beim Refactoring birgt. Die Bank-Anwendung wird als Back- und Frontend in getrennten Pods gestartet, mit der Konsequenz, dass diese nicht mehr über localhost kommunizieren können. Da das Frontend hinter einem nginx-Proxy läuft, kann dies intern gehandhabt werden und bedarf so keiner Änderung im Code. Auf diese Weise können die Projekte weiterhin lokal ausgeführt werden, ohne dass die Adressen händisch geändert werden müssten. Das HLS-Systen und die Bank kommunizieren über RabbitMQ, welches ebenfalls als Image zur Verfügung gestellt wird. Der *Informationsfluss* im Diagramm zeigt, wer an welche Queue Nachrichten abschickt oder welche liest.

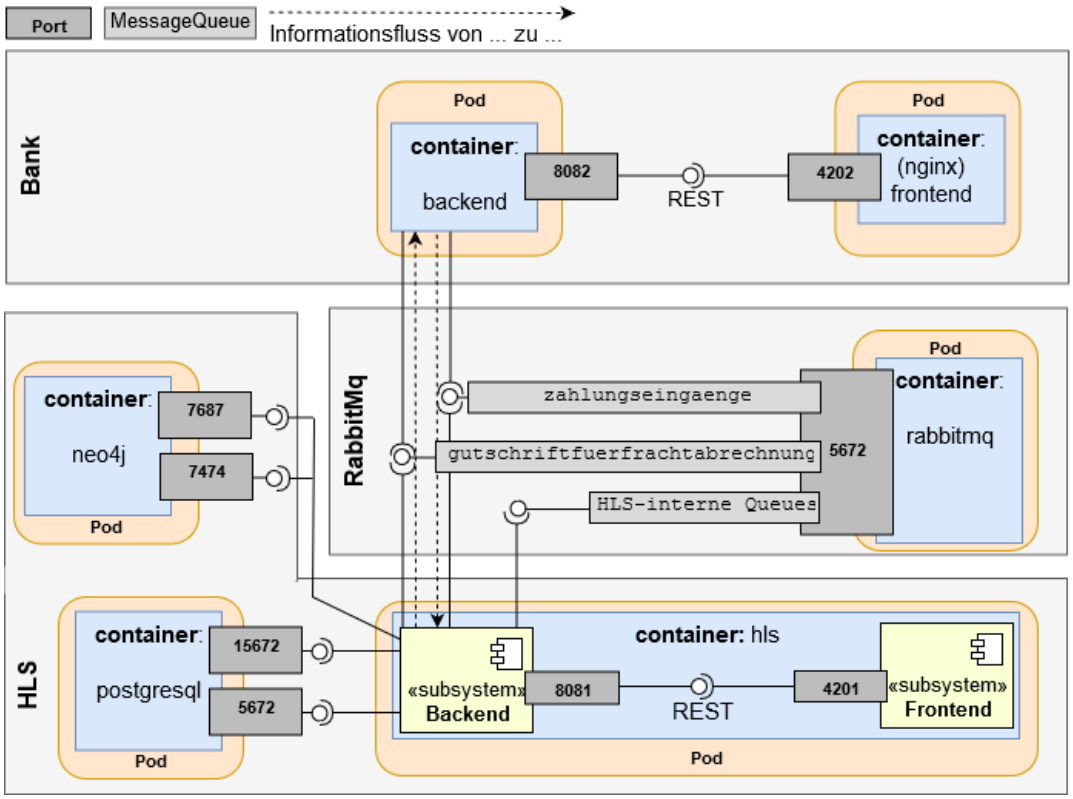


Abbildung 4.2: Architektur - Gesamtsystem

4.2 Projektplanung

Zur Planung von Softwareprojekten wird oft eine User-Story-Map verwendet. Dies setzt die Nutzung der in der Anforderungsanalyse ermittelten User-Stories voraus. Die vorhandenen User-Stories beschreiben die nötigen Schritte zur Umsetzung der Funktionalitäten nicht und eignen sich wenig zur Planung des Vorhabens. Aus diesem Grund werden für dieses Projekt die funktionalen und qualitativen Anforderungen als Aufgabenpool genutzt. Von der Nutzung einer User-Story-Map wird daher abgesehen und ein lineares Verfahren ins Auge gefasst.

Die Anforderungen sind in vielen Teilen voneinander abhängig und können nicht in beliebiger Reihenfolge ausgeführt werden. Um den Projektverlauf optimal verfolgen und planen zu können, bietet sich das Gantt-Diagramm an. Ursprünglich wurde der Plan nach Fachlichkeit strukturiert und zwischen Infrastruktur, Refactoring, Kubernetes-Deployment, Einarbeitung und Dokumentation unterschieden. Diese Kategorisierung ist sinnvoll, wenn in größeren Teams gearbeitet wird, die jeweils nur bestimmte Sichten auf den Projektverlauf benötigen. In diesem Fall handelt es sich um ein Projekt, das nur von einer Person durchgeführt wird, sodass diese Aufteilung zu einem sehr unübersichtlichen Plan führen würde. Aus diesem Grund wird nach zusammenhängenden Aufgabenbereichen strukturiert, mit Ausnahme des Dokumentationsbereiches. Abb. 4.3 zeigt den initialen Projektplan.

Zu Beginn wird die nötige Infrastruktur für die Durchführung des Projekts aufgebaut und der vorhandene Code refactored. Anschließend wird das scripts-Repository erstellt, da es eine helfende Rolle zur Entwicklung der übrigen Projekte spielt. Es besteht damit die Möglichkeit, die Effektivität der dort zur Verfügung gestellten Hilfsmittel direkt zu testen und zu nutzen. Danach wird das Kubernetes-Deployment vorbereitet, indem die Dockerfiles geschrieben und die Images in die GitLab-Registries gespeichert werden. Die Dokumentation soll während des ganzen Projekts vervollständigt werden. Da die Führung einer Dokumentation während der Bearbeitung einer konkreten Aufgabe in vielen Fällen zurückbleibt, werden zusätzliche Tage ausschließlich zu diesem Zweck eingeplant.

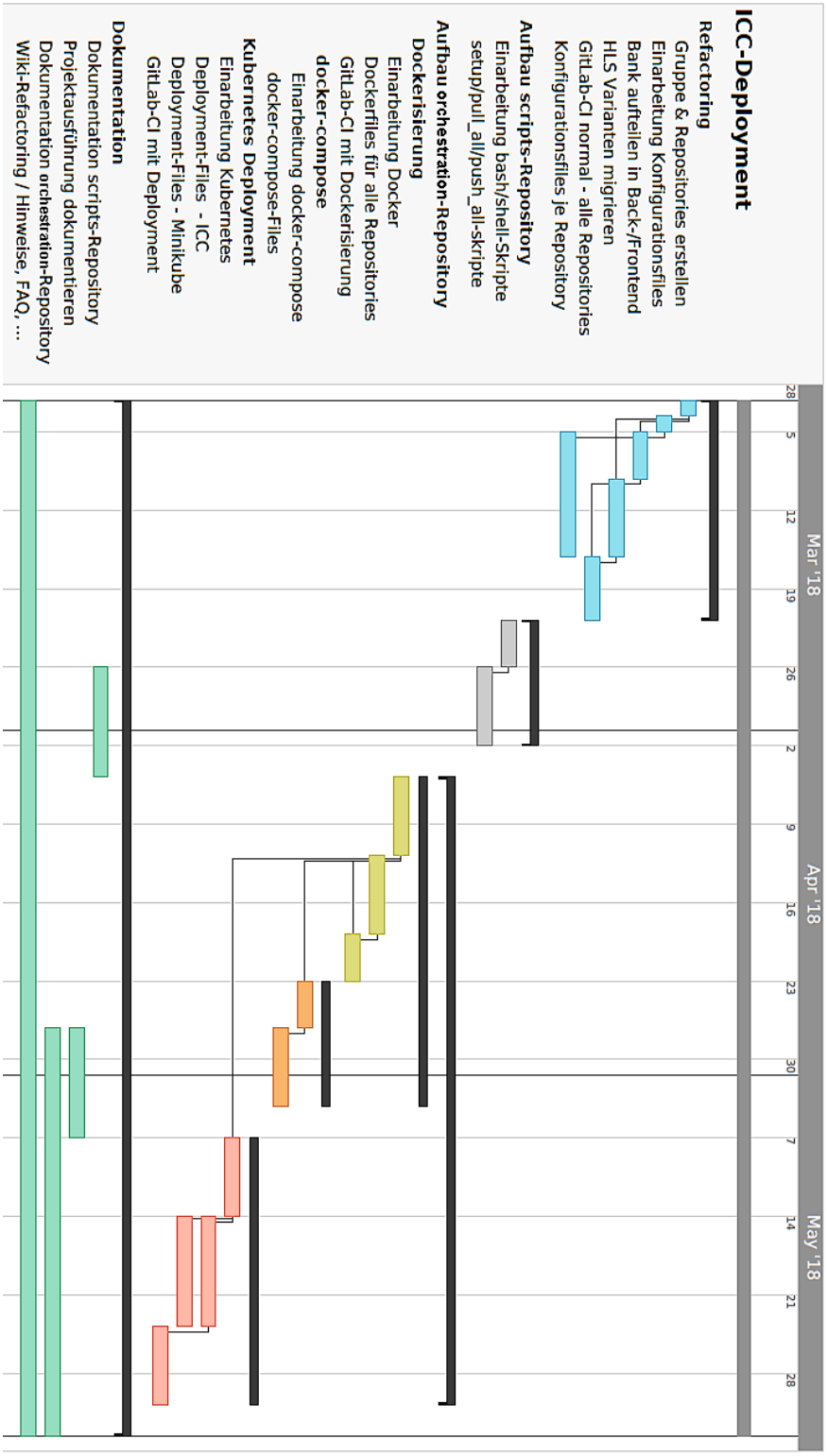


Abbildung 4.3: Gantt-Diagramm - Projektplan, erstellt mit TeamGantt

4.3 Risikomanagement

Die Tabelle 4.1 zeigt die am meisten zu beachtenden Risiken bei der Umsetzung des Vorhabens. Da dieses Projekt nur von einer Person durchgeführt wird, ist der Spielraum im Umgang mit Risiken sehr gering. Es besteht keine Möglichkeit zusätzliche Mitarbeiter zu engagieren, um Rückstände aufzuholen oder Aufgaben an eine andere Abteilung abzugeben. Der Umgang beschränkt sich demnach überwiegend auf die Einplanung zusätzlicher Tage, um die Auswirkungen zu reduzieren (siehe Tabelle 4.2).

ID	Auslöser	Eintrittswkt.	Auswirkungen	Priorität	Konsequenz
1	Entwicklung falscher Funktionen	2	2	4	Zeiteinbußen, unklare Struktur
2	Unnötige Features	2	2	4	Zeiteinbußen, unklare Struktur
3	häufige Änderung von Anforderungen/Zielen	1	3	3	Zeiteinbußen, unklare Struktur
4	Fehleinschätzung eigener Fähigkeiten	3	5	15	Demotivation, Zeiteinbußen
5	unrealistische Planung	4	5	20	Zeiteinbußen
6	Probleme mit der ICC	3	4	12	Zeiteinbußen, ggf. Umstrukturierung
7	Unvorhersehbare Zeiteinbußen	2	5	10	Zeiteinbußen

Tabelle 4.1: Risikoanalyse - Teil 1

Besonders große Tragweite besitzen die Risiken 5. und 6., da sie kaum angemessen behandelbar sind. Die Abschwächung der übrigen Risiken basieren darauf, dass der Projektplan einigermaßen realistisch ist. Trifft diese Annahme nicht zu, erhöht sich der zu erwartende Zeitaufwand unkontrollierbar.

Das Deployment in die ICC bildet den Kern dieser Arbeit, sodass ein Ausweichen auf ein anderes Kubernetes-Cluster nicht möglich ist. Treten Probleme mit dem Cluster auf, so müssen diese in Zusammenarbeit mit den Verantwortlichen gelöst werden. Liegen die Fehler aufseiten der ICC, muss mit längeren Verzögerungen gerechnet werden, da es sich hierbei um ein externes System handelt und keine Möglichkeit besteht, direkt Einfluss auf die Fehlerbehandlung zu nehmen. Zusätzliche Anstrengungen im Bereich Risikomanagement bringen aufgrund des geringen Handlungsspielraumes viel Aufwand für wenig Nutzen und werden somit nicht unternommen.

ID	Auslöser	Maßnahmen	Effektivität
1	Entwicklung von falschen Funktionen	klare Anforderungen, Absprache mit Prüfer	zuverlässig sofern Projektplan realistisch
2	Unnötige Features	klare Anforderungen, Absprache mit Prüfer	zuverlässig sofern Projektplan realistisch
3	häufige Änderung von Anforderungen/Zielen	klare Anforderungen, Absprache mit Prüfer	zuverlässig sofern Projektplan realistisch
4	Fehleinschätzung der eigenen Fähigkeiten	Zeitpuffer, Hilfe in Anspruch nehmen	zuverlässig sofern Projektplan realistisch
5	unrealistische Planung	Zeitpuffer	gering, zu große Auswirkungen
6	Probleme mit der ICC	Zeitpuffer, Austausch mit ICC	gering, unkalkulierbarer Aufschub
7	Unvorhersehbare Zeiteinbußen	Zeitpuffer	zuverlässig sofern Projektplan realistisch

Tabelle 4.2: Risikoanalyse - Teil 2

5 Durchführung

Die Aufgaben im Projekt wurden zu Beginn in Kanban-Boards je Repository verwaltet und überwacht. In Anbetracht der Menge beteiligter Repositories und der daraus resultierenden Anzahl an zu verwaltenden Kanban-Boards stellte sich früh heraus, dass dieser Ansatz in diesem, von nur einer Person durchgeführten, Vorhaben mehr Aufwand als Übersicht schafft. Als Alternative wurde die Sammlung aller Aufgaben in einem zentralen Kanban-Board oder die Verwaltung in einem Notizbuch in Betracht gezogen. Letzteres stellte sich als hilfreich heraus, da das Heft zusätzlich als eine Art Logbuch fungierte und so den Projektverlauf festhält. Der Projektplan ist nicht so komplex, als dass der Überblick zu den Aufgaben schnell verloren ginge. Auf die zusätzliche Nutzung eines zentralen Kanban-Boards wurde aus diesem Grund verzichtet. Im Folgenden werden die im Projektplan festgehaltenen Meilensteine in ihrer Ausführung beschrieben.

5.1 Refactoring

Die Trennung von bank-backend und bank-frontend gestaltete sich einfach. Da beide Systeme über REST Daten austauschen, mussten lediglich beide Module in eigene Projekte ausgelagert und in eigene Repositories gepushed werden. Dem bank-backend wurde ein Konfigurationsfile hinzugefügt, das an zentraler Stelle gespeichert und beim Startvorgang der Anwendung geladen wird. Zusätzlich wurden die “.gitlab-ci.yml“-Files getrennt.

Die Migration der zwei HLS-Varianten gestaltete sich schwieriger, da das Projekt in Gänze nur ohne die Veränderungen (Buchhaltungskomponente und Kubernetes-Deployment) bekannt war. Es gab wenig Informationen über die tatsächlich veränder-

ten Dateien. Mit einiger Verzögerung konnte ein fehlender Projektbericht eingesehen werden, der einen groben Überblick über die vorgenommenen Änderungen bezüglich des Kubernetes-Deployments ermöglichte. Die HLS-Variante mit der zusätzlichen Buchhaltungskomponente wurde zudem in fehlerhaftem Zustand übernommen, sodass die Tests nicht erfolgreich durchlaufen. Erst gegen Ende des Projekts konnte die Migration der beiden Varianten teilweise durchgeführt werden.

Dazu wurde als Basis das System mit der vorhandenen Buchhaltungskomponente verwendet und die Änderungen zum Kubernetes-Deployment wie in der anderen Variante vorgenommen. Bemühungen, die übernommenen Fehler zu korrigieren, wurden aus Zeitgründen eingestellt, da sie bereits zur Fertigstellung des HLS-Buchhaltungsprojektes vorhanden waren und augenscheinlich keine schwerwiegenden Probleme verursachten. Dieser Fall musste weiter im Risikomanagement beobachtet werden, da spätere Fehler möglicherweise auf diese Versäumnisse zurückzuführen sind.

Neben den bereits genannten Schritten wurden keine Versuche unternommen, die Konfiguration des HLS-Systems zu zentralisieren, da angenommen wurde, dies sei bereits gegeben. Im Laufe des Projektes stellte sich allerdings heraus, dass Konfigurationen an mindestens vier verschiedenen Stellen im Code vorgenommen werden, was die Konfiguration selbst sehr aufwendig macht. Beispielsweise wird eine REST-Konfiguration (CD: “code/hls/4_Startup/hls-core-startup/RestConfigurationFiles/hosting.json“) aufgesetzt, deren URL bei Programmstart (CD: “code/hls/4_Startup/hls-core-startup/Program.cs“) noch einmal gesetzt wird. Dies mag in Einzelfällen nötig sein, erschwert jedoch die Verständlichkeit des Codes und führt zu unerwarteten Fehler, da ein Nutzer/eine Nutzerin davon ausgeht, die Hosting-Adresse an nur einer Stelle ändern zu müssen. Da dieser Umstand zu spät erkannt wurde, konnte leider keine Korrektur mehr stattfinden.

Der Standardport wurde im Nachhinein von Port 8080 auf Port 8081 gelegt, um Kollisionen mit gegebenenfalls anderen laufenden Systemen zu vermeiden.

5.2 Aufbau des scripts-Repository

Das scripts-Repository soll die Entwicklung bei Bearbeitung mehrerer Repositories und das Setup unterstützen. Dazu wurden drei Skripte erstellt: “clone_all“, “pull_all“ und “session_conclusion“. Das erste Skript klont alle Repositories und ihre Wikis in den Projektordner. Für den Aufbau der Projektstruktur ist somit nur das Klonen des scripts-Repositories in einen Projektordner der Wahl und die Ausführung des “clone_all“-Skripts nötig. Die Skripte “pull_all“ und “session_conclusion“ dienen der Versionierung aller Repositories auf einmal. Sie sind zur Unterstützung gedacht, wenn an mehreren Projekten gleichzeitig gearbeitet wird und die Versionierung einzelner Repositories schnell vergessen wird. Die Mehrfachimplementierung der Skripte für Windows und Mac/Linux-NutzerInnen stellte eine Belastung dar, da dies bei Änderungen an mehreren Stellen korrigiert werden musste. Da sich keine Alternative zu diesen Schritten bietet, um Linux/ Mac und Windows Skripte zu verbinden, und die Änderung der Skripte bei korrekter Implementierung selten vorkommt, wurde dennoch an diesem Vorhaben festgehalten.

Ein weiteres Problem stellte das “pull_all“-Skript dar. Sollten Änderungen in diesem Skript vorgenommen worden sein, werden diese bei Ausführung desselben übernommen und können zu Konflikten führen. In der Regel lässt sich dies mit einem erneuten “`git pull`“ im “scripts“-Repository beheben. Wie bereits diskutiert, ist die regelmäßige Änderung dieser Skripte nicht vorgesehen, sodass die Lösung dieses Problems wenig Priorität besitzt.

5.3 Dockerisierung

Ursprünglich konnten Docker-Images im Nexus-Repository der HAW gespeichert werden. Im Projektverlauf wurde der Zugriff darauf durch die Entwickler der ICC auf einen reinen Lese-Zugriff umgestellt. Zwar besaß diese Änderung das Potenzial den Projektverlauf durch nötige Anpassungen zu stören, sie wurde jedoch früh angekündigt und umfassend mit allen Konsequenzen und zu ergreifenden Maßnahmen dokumentiert (siehe Behnke, 2018). Dementsprechend konnte die Umstellung reibungslos verlaufen.

Im Folgenden werden die implementierten Dockerfiles der Anwendungen vorgestellt. **Hinweis:** Aus Darstellungsgründen wurden in die Codebeispiele teils syntaktisch nicht korrekte Zeilenumbrüche eingefügt (gekennzeichnet durch "\").

bank-backend

```
1  FROM nexus.informatik.haw-hamburg.de/maven:3-jdk-8-slim AS BUILD
2  WORKDIR /mvn-app
3  COPY pom.xml .
4  COPY src/main ./src/main
5  RUN mvn clean install -Dmaven.test.skip=true \
6    -DdependencyLocationsEnabled=false
7
8  FROM nexus.informatik.haw-hamburg.de/openjdk:8-jdk-slim
9  COPY --from=BUILD /mvn-app/target/bank-backend-0.0.1-SNAPSHOT.jar \
10   app.jar
11 ENV JAVA_OPTS=""
12 ENTRYPOINT exec java $JAVA_OPTS \
13   -Djava.security.egd=file:/dev/./urandom -jar /app.jar
```

Quellcode 5.1: Dockerfile - bank-backend

Das bank-backend benötigt als Maven-Projekt ein Maven-Image als Basis. Der Code 5.1 zeigt die Implementierung des Dockerfiles. Zunächst wird mit "`mvn clean install`" eine *jar* der Applikation erstellt und anschließend in ein leichtgewichtiges Java-Image ("`openjdk:8-jdk-slim`") kopiert. Die "`slim`"-Images enthalten eine sehr schlanke Linux-Installation, die nur die nötigsten Pakete enthält. Der CI wurde eine Stage hinzugefügt, welche die Dockerisierung und Speicherung in der GitLab-Registry durchführt. Mit den multi-stage builds läuft die CI-Pipeline schnell durch.

Zuvor gab es Probleme durch sehr große Downloads beim Maven-Build und daraus resultierenden Timeouts, obwohl die Dockerisierung lokal funktionierte. Ursprünglich wurde versucht das Problem über verschiedene Dockerfiles für lokale und automatische Dockerisierung zu beheben. Dazu wurden die Dockerfiles mit Skripten in eigene Ordner beziehungsweise bei Bedarf in das Root-Verzeichnis kopiert, sodass immer das benötigte File gerade im Root-Verzeichnis lag.

Da die Ausführung von Skripten in der GitLab-CI keine Rückgabewerte liefert und keine Fehler innerhalb des Skriptes zurückgibt, kann die Pipeline erfolgreich durchlaufen, ohne dass diese erkannt werden. Aus diesem Grund wurde von dieser Möglichkeit abgesehen und die Lösung wie im Code 5.1 erarbeitet. Parallel haben Mitarbeiter der ICC an dem Downloadproblem des Docker-Images für die GitLab-CI gearbeitet und schließlich gelöst. An dieser Stelle ist das Risiko bezüglich der Probleme mit der ICC zu einem akuten Problem geworden. Eine Abwälzung oder schnelle Lösung war nicht möglich und führte zu enormen Zeiteinbußen, die mit etwa zwei Wochen die eingeplante Toleranz im Projektplan deutlich überstiegen.

bank-frontend

```
1  FROM docker-hub.informatik.haw-hamburg.de/narjes-ba/\
2    frontend-ci-image AS BUILD
3  MAINTAINER Mieke Narjes <mieke.narjes@haw-hamburg.de>
4  WORKDIR /ng-app
5  COPY package.json .
6  RUN yarn
7  COPY e2e tsconfig.json tslint.json angular.json ./
8  COPY src ./src
9  RUN $(npm bin)/ng build --prod --build-optimizer
10
11 FROM nexus.informatik.haw-hamburg.de/nginx:1.13.3-alpine
12 COPY nginx/nginx.conf /etc/nginx/nginx.conf
13 ## Remove default nginx website
14 RUN rm -rf /usr/share/nginx/html/*
15 COPY --from=BUILD /ng-app/dist /usr/share/nginx/html
```

Quellcode 5.2: Dockerfile - bank-frontend

Das bank-frontend benötigt Node als Basis-Image. Dazu wurde ein entsprechendes Repository (“frontend-ci-image“) mit einem eigenen Docker-Image für Node angelegt. Dies ermöglicht die beliebige Erweiterung des Images. Grund für diese Maßnahmen waren Probleme mit veralteten npm-Versionen auf offiziellen Node-Images. Es musste für die CI mit einem Deploy-Key zugänglich gemacht werden.

Der Code 5.2 zeigt die Implementierung des Dockerfiles. Für das Frontend-Image wird die App mit "`ng build`" gebaut. Zuvor muss "`npm install`" ausgeführt und die dafür nötigen Dateien (insbesondere "package.json") kopiert werden. Im eigentlichen Image wird ein Nginx-Image als Basis verwendet, damit ein Nginx-Proxy für die Kommunikation mit dem Frontend verwendet werden kann. Dazu muss eine Proxy-Konfiguration erstellt und die optimierte Frontend-Applikation aus dem zuvor verwendeten Image kopiert werden. Das Image enthält damit lediglich eine Nginx-Installation, die Proxy-Konfiguration und die Frontend-Applikation.

Ursprünglich wurden zwei Images erstellt, in der Annahme, dass das lokale Image eine andere Proxy-Konfiguration benötigt als das für die ICC. Da die Services sowohl in den docker-compose-Files als auch in den Deployment-Definitionen identische Bezeichner besitzen, kann auf eine zusätzliche Konfiguration verzichtet werden.

Das bank-frontend verfügt über keine Tests, sodass die "`.gitlab-ci.yml`" lediglich die Dockerisierung enthält. Durch den multi-stage build muss hierzu ein Login zum GitLab des AI Labors, die Dockerisierung und anschließend die Speicherung des Images in die Registry durchgeführt werden.
Updates durch den Node Package Manager führten zu Fehlern bei der Ausführung von npm-Befehlen, sodass der Abschluss dieser Aufgabe einige Tage mehr benötigte, als erwartet. Zusätzlich wurde ein fehlerhaftes Update einer Dependency (*flexlayout-5.0.0-beta.14-17a70ee* veröffentlicht (vgl. GitHub, 2018)). Der Fehler konnte mit der Anpassung der Version behoben werden.

Die Möglichkeit, bestimmte Versionsbereiche in der "`package.json`" zur Angabe der Abhängigkeiten zu nutzen, ist eine bequeme Art, sich die Versionsverwaltung zu vereinfachen. Dennoch birgt dies das Risiko möglicherweise fehlerhafte Versionen zu nutzen, sodass ohne eigenes Zutun die Anwendung nicht mehr lauffähig ist. Dementsprechend wurden die zuletzt genannten Punkte in das Risikomanagement aufgenommen.

hls

Das HLS-System besaß bereits ein Dockerfile für die Applikation und die Postgres-Datenbank, welche von einem Studenten im Rahmen eines Wahlpflichtprojektes erstellt wurden. Im Original bestanden beide Images lediglich aus der letzten Buildstage (hls-Dockerfile ab Zeile 16 und postgresdb-Dockerfile ab Zeile 12).

```
1  FROM docker-hub.informatik.haw-hamburg.de/narjes-ba/\
2    frontend-ci-image:latest as BUILD-FRONTEND
3  WORKDIR /ng-app
4  [...] # siehe bank-frontend Dockerfile
5  RUN $(npm bin)/ng build --prod --build-optimizer
6
7  FROM docker-hub.informatik.haw-hamburg.de/narjes-ba/\
8    dotnet-sdk-node-docker AS BUILD-HLS
9  WORKDIR /source
10 COPY . ./
11 COPY --from=BUILD-FRONTEND /ng-app/wwwroot   \
12   ./4_Startup/hls-core-startup
13 COPY dockerfiles/hls/hls-global-configuration.json \
14   2_Util/Common/Configurations
15 RUN dotnet restore
16 RUN dotnet publish -c Release -o out
17
18 FROM microsoft/aspnetcore:2.0
19 #Copy dll and configuration files to container image
20 COPY --from=BUILD-HLS source/4_Startup/hls-core-startup ./
21 EXPOSE 8082
22 RUN ls
23 #define Entrypoint on container start
24 ENTRYPOINT ["dotnet", "out/hls-core-startup.dll"]
```

Quellcode 5.3: Dockerfile - hls

Der Code 5.3 zeigt die Erweiterung des HLS-Dockerfiles durch multi-stage builds. Die Erstellung des Frontends entspricht den Schritten, die in bank-frontend bereits dargelegt wurden. Anschließend wird der Output des Frontend-Builds (“wwwroot“) kopiert und die Applikation als Ganzes mit “`dotnet publish`“ gebaut, sodass sie optimiert und ausführbar in einem Ordner zur Verfügung steht. Dieser Ordner wird in ein eigenes Image kopiert und mit einem “ENTRYPOINT“ beim Start ausgeführt. Somit ist der Bau des Frontends in der CI nur noch für die Frontend-Tests notwendig, sodass eine bisher vorhandene und zeitaufwendige Stage (“`publish`“) eingespart werden konnte. Im Image ist zusätzlich ein separates Konfigurationsfile zur Angabe

der Nebensysteme (Postgres, RabbitMQ und Neo4J) nötig, damit die Namensauflösung korrekt funktionieren kann. Statt beispielsweise für RabbitMQ den Host “localhost“ anzugeben, wird “rabbitmq“ verwendet, was der Servicebezeichnung in den Deployment- und docker-compose-Files entsprechend wird. Diese Konfiguration wurde in einem separaten Ordner gespeichert und im build-Schritt des Images an die entsprechende Stelle kopiert.

Das folgende Postgres-Dockerfile 5.4 ist nötig, um SQL-Tabellen anzulegen und Zugangsdaten zu konfigurieren. In der CI-Pipeline wird in der *build*-Stage die Anwendung mit “`dotnet build`“ gebaut und anschließend ein SQL-Skript (“init_database.sql“) erzeugt, das dem Postgres-Image hinzugefügt wird. Angestrebt war die Bündelung aller für den Bau des Images nötigen Schritte im Dockerfile. Die dotnet-Befehle zur Erzeugung des SQL-Skriptes benötigen eine laufende Postgres-Instanz, sodass der Bau durch multi-stage builds nicht ohne Weiteres möglich war.

```
1  FROM docker-hub.informatik.haw-hamburg.de/\
2    narjes-ba/dotnet-sdk-node-docker AS BUILD
3
4  WORKDIR /src
5  COPY ./ ./
6  RUN dotnet restore
7  RUN dotnet build
8  RUN (cd 4_Startup/hls-core-startup \
9    && dotnet ef migrations add HLSMigration \
10   && dotnet ef migrations script > init_database.sql)
11
12 FROM library/postgres
13 ENV POSTGRES_USER hls
14 ENV POSTGRES_PASSWORD hls2017
15 ENV POSTGRES_DB hlsdb
16 COPY --from=BUILD src/4_Startup/hls-core-startup/\
17   init_database.sql/docker-entrypoint-initdb.d/
18 EXPOSE 5432
```

Quellcode 5.4: Dockerfile - Postgres HLS-Datenbank

Als Lösungsansatz wurde die Nutzung eines Docker-in-Docker-Images in Betracht gezogen, um dort den Postgres-Container zu starten. Dieses Image ist allerdings sehr schwergewichtig und verhielt sich nicht wie erwartet. Erfolgreich war die Ergänzung des bereits vorhandenen .NET-Basis-Images um eine PostgreSQL-Installation.

Der Code 5.4 zeigt wie das für die Datenbank nötige SQL-Skript erzeugt und anschließend in ein schlankes Postgres-Image kopiert wird. Zudem wurde ein Skript geschrieben, um die Dateien bei Bedarf in das root-Verzeichnis zu kopieren. Beim Bau beider Images benötigt das Laden des Kontextes lokal viel Zeit. Um die Zeiten zu verringern, sollten die ".dockerignore"-Files durch alle nicht notwendigen Dateien ergänzt werden. Dies wurde aus Zeitgründen unterlassen.

Wie im Abschnitt Refactoring bereits beschrieben, sind die HLS-Tests teils nicht erfolgreich. Um trotzdem die CI zur Dockerisierung und dem Deployment durchführen zu können, sind die Tests auskommentiert. Zudem besteht das Problem, dass die Test-Stages aus unbekannten Gründen keine Pods für die CI zugewiesen bekommen, während die übrigen Stages korrekt ausgeführt werden. Da die Tests ohnehin nicht erfolgreich durchlaufen, wird auch dieser Fehler aus Zeitgründen ignoriert.

Leider ist das HLS-Image nicht vollständig lauffähig, da es aus unbekannten Gründen bei Ausführung mit docker-compose kein Frontend ausliefert mit der Fehlermeldung, das "mainbundle.js" könne nicht geladen werden. Greift man über "`docker exec -it <containerID>`" auf den Container zu und überprüft die dort vorliegenden Dateien, so ist die als fehlend angegebene Datei in vollem Umfang vorhanden. Aus Zeitgründen konnte dieser Fehler nicht korrigiert werden.

5.4 docker-compose

Die docker-compose-Files konnten nach erfolgreicher Erstellung der Docker-Images schnell aufgebaut werden. Der Code 2.3 im Grundlagenkapitel zeigt das docker-compose-File, das die Bank-Anwendung startet. Wichtig hierbei ist, die Abhängigkeiten zwischen den Anwendungen zu kennen und die Startreihenfolge im Blick zu behalten, damit es zu keinen Konflikten kommt. Diese Abhängigkeiten können bei den Services im docker-compose-File direkt angegeben werden, sodass sich Docker selbst um das Scheduling kümmert.

Insgesamt sind vier Dateien entstanden, die durch "`docker-compose -f <filename.yml> up`" gestartet werden können. "all.yml" startet die gesamte Applikation mit allen Nebensystemen. Trotz der Angabe von Abhängigkeiten (durch "`depends_on`"), startet das HLS-System oft schneller als die Nebensysteme hochfahren können. Der HLS-Service muss im Anschluss erneut gestartet werden ("`docker-compose -f all.yml up hls`"). Es ist davon auszugehen, dass Health-Checks zur Verfügung stehen, um solche Zeitfaktoren mit in die Startreihenfolge mit einzubeziehen. Dies wurde aus Zeitgründen nicht weiter verfolgt.

"bank.yml" startet ausschließlich das bank-backend und bank-frontend. "dependents.yml" startet alle Nebensysteme (Neo4J, RabbitMQ und PostgreSQL) während "hls.yml" das HLS-System startet. Es ist zu beachten, dass die Abhängigkeiten zwischen den Containern nur innerhalb eines docker-compose-Files berücksichtigt werden. Somit müssen die Nebensysteme vor den Anwendungen gestartet werden, um fehlschlagende Verbindungsversuche mit RabbitMQ, Postgres oder Neo4J zu vermeiden.

5.5 Kubernetes-Deployment

Mit dem Wissen über die Abhängigkeiten zwischen den Projekten und der benötigten Ports, war die Zusammenstellung der Deployment-Files wenig aufwendig. Jede Anwendung wurde zu einem Service zusammengefasst. Neben den Services *bank-backend*, *bank-frontend* und *hls* wurde zusätzlich ein *rabbitmq*-Service benötigt, um die Message-Queue umzusetzen. Postgres und Neo4J waren ursprünglich im HLS-Service enthalten, da sie von keinem anderen Service verwendet wurden. Da es wiederholt zu Abstürzen kam, sobald die Nebensysteme langsamer starteten als das HLS, wurden sie in der finalen Version in eigene Deployments ausgelagert. Eine zuverlässigere Alternative wäre auch hier die Nutzung von Health- oder Ready-Checks, um die korrekte Reihenfolge beim Start zu garantieren.

Die Anwendungen wurden je mit aussagekräftigen Labels bzw. Selektoren versehen, um die Service Discovery durch Kubernetes zu ermöglichen. Von der Verwendung von NodePorts wurde abgesehen, da sie keinen Mehrwert für die interne Nutzung dieses

Systems bietet. Zudem ist der Wertebereich der Ports so hoch (30000-32767) angelegt, dass sie in manchen Netzwerken - meist aus Sicherheitsgründen - geblockt werden (vgl. Thompson, 2017).

Aus Übersichtsgründen sind die Service-Definitionen direkt im Deployment-File untergebracht. Durch "`---`" getrennt, werden die Abschnitte als zwei unterschiedliche Dokumente behandelt. Da weder die Service-Definition noch das Deployment so lang sind, dass sie ein eigenes Dokument bräuchten und so alle Informationen zu einer Anwendung an einem Platz gesammelt sind. Die Gefahr, Anpassungen in einer anderen Datei nach Änderungen zu vergessen wird so reduziert.

Die Deployments können in der GitLab-CI des orchestration-Repositories durchgeführt werden. Dementsprechend ist für jede CI der Anwendungsrepositories eine zusätzliche Stage hinzugefügt worden, welche die Ausführung der CI im orchestration-Repository anstößt. Darin werden zunächst alle vorhandenen Deployments und Services gelöscht, um sicherzugehen, dass keine veralteten Elemente ins neue Deployment aufgenommen werden und anschließend die Deployments nach und nach gestartet. Da alle Nebensysteme in eigenen Deployments laufen, kann hier auf die Startreihenfolge geachtet und so Programmabstürze durch fehlende Verbindungen vermieden werden. Race-Conditions sind an dieser Stelle nicht ausgeschlossen und müssten durch Ready-Checks eliminiert werden. Um auf die Images in den privaten Registries zugreifen zu können, mussten dem Namespace manuell Image-Pull-Secrets hinzugefügt werden, die ebenfalls in den DeploymentFiles angegeben werden mussten. Dazu wurden von GitLab Tokens generiert werden, die anschließend mit einem "`kubectl`"-Befehl zu einem Secret gewandelt wurden. Die GitLab-Tokens sind nach Generierung nicht mehr lesbar und müssen an einem sicheren Ort gespeichert werden, da sie einem Kubernetes-Deployment Zugriff auf die Images in privaten Registries ermöglichen. Der Verlust der Tokens stellt dennoch kein großes Problem, da im Zweifelsfall neue erzeugt und hinzugefügt werden können.

Von der Implementierung zusätzlicher Minikube-Deployment-Files wurde abgesehen, da sich herausstellte, dass die Nutzung lokaler Images für das Deployment nicht ohne Weiteres möglich ist und so der Mehrwert verloren geht. Die Möglichkeit, neu erstellte Images oder geänderte Deployments schnell in Minikube zu testen, kann daher nicht angeboten werden. Als Konsequenz konnten die Duplikate der Deployment-Files gelöscht werden.

In der finalen Version des Deployments ist die Bank-Anwendung voll funktionsfähig und auch die Nebensysteme werden korrekt gestartet und sind erreichbar. Das HLS-System stürzt stetig ab, da es aufgrund einer *IOException* in Bezug auf Neo4J Timeouts erhält. Die Fehlersuche wurde aufgrund von Zeitmangel eingestellt.

5.6 Dokumentation

Die regelmäßige Anpassung der Dokumentation verlangte viel Disziplin. Grund hierfür war eine fehlende Grundstruktur. Die schnelle Anpassung der Dokumentation bei Änderungen wurde auf diese Weise nicht unterstützt, da zu Beginn erst das Wiki in ihrer Grundstruktur erstellt hätte werden müssen. Die Anpassung vorhandener Wikis stellte sich als wenig zeitaufwendig und gut in den Entwicklungsprozess integrierbar heraus. Die regelmäßige Ergänzung der Dokumentation ist wichtig, da bei der Sammlung aller Erkenntnisse und Änderungen in einer Session oft Details oder auch wichtige Elemente vergessen werden.

Die Erstellung eines FAQ je Dokumentation hat sich als problematisch herausgestellt. Eine Mehrfachnennung verschiedener Themen sollte vermieden werden, damit Änderungen nicht an mehreren Stellen gleichzeitig erfolgen müssen. An dieser Stelle bietet sich mit hoher Wahrscheinlichkeit die Einrichtung eines zentralen Dokuments (z. B. im scripts-Wiki) an, sodass immer klar ist, an welcher Stelle gesucht werden muss. Die Erfahrung zeigt, dass auch der Aufbau einer Dokumentationsstruktur gut durchdacht sein muss, um späteren NutzerInnen die Arbeit und Pflege mit dem Wiki zu erleichtern.

In der finalen Version sind die Dokumentationen der Bank um alle Änderungen und Hinweise ergänzt worden, die im Laufe des Projektes vorgenommen wurden, beziehungsweise aufgefallen sind. Die Hinweise zum Starten der HLS-Anwendung im HLS-README sind angepasst und jedem Repository ein Hinweis auf das zentrale scripts-Repository hinzugefügt worden. Das scripts-Repository bietet eine ausführliche Anleitung zur Installation des Projektes und eine Liste an Aufgaben, die noch zu erledigen sind. Zu Beginn der Dokumentation werden Zusammenhänge der Repositories dargestellt und mit einer Stichwortliste auf die übrigen Wikis verwiesen. Das orchestration-Repository enthält alle Informationen zur Ausführung der Anwendungen mit docker-compose oder als Deployment, sowie hilfreiche Befehle und Adressen.

6 Evaluation

6.1 Verifikation

In der Verifikation wird diskutiert, inwieweit das finale System den Anforderungen und der Spezifikation entspricht. Dazu werden zunächst die erarbeiteten Use-Cases aus der Spezifikation betrachtet und anschließend die Anforderungen.

6.1.1 Use-Cases

Docker Compose

Dieser Use-Case beschäftigt sich mit dem Starten der Anwendungen durch docker-compose. Ziel ist das Testen von Änderungen und des Zusammenspiels der Anwendungen. Zusätzlich sollen mit docker-compose die Nebensysteme gestartet werden können, damit diese nicht einzeln heruntergeladen und installiert werden müssen.

Dieses Ziel wurde zu großen Teilen erfüllt. Alle Systeme können insgesamt oder einzeln über docker-compose gestartet werden und miteinander kommunizieren. Das HLS-System fährt, den Logs nach zu urteilen, ordnungsgemäß hoch, ist jedoch im Browser nicht erreichbar und zeigt bisher nicht erklärbares Verhalten. Startet man die Anwendung lokal ohne Image, so verhält sich das System erwartungsgemäß und ohne Fehler, auch im Zusammenspiel mit den durch docker-compose gestarteten Containern.

Manuelles Deployment

Das Manuelle Deployment dient dem Testen und der Veränderung von Deployments unabhängig von GitLab. Über die Command Line kann das Deployment zur ICC fast vollständig durchgeführt werden. Der Use-Case erfüllt alle Erweiterungen bis auf das HLS-Deployment, das vermutlich durch Fehler in der Anwendung selbst nicht erfolgreich ist. Wie in der Durchführung diskutiert, wurde das Deployment zu Minikube im Projektverlauf als nicht sinnvoll verworfen.

Automatisches Deployment

Das automatische Deployment verfolgt das Ziel, das System komplett neu zu deployen, sobald Änderungen in einer GitLab-CI erfolgreich getestet wurden (Continuous Deployment). Mit Ausnahme der bereits beschriebenen Probleme mit dem HLS-System ist dies voll erfüllt. Der Use-Case wurde durch einige Punkte erweitert, indem das alte Deployment bei Neuanstoß der CI-Pipeline im orchestration-Repository komplett gelöscht wird und die CI des HLS-Projektes zur Dockerisierung das dotnet-Image benötigt.

6.1.2 Anforderungen

Sofern die Anforderungen nicht bereits durch einen Use-Case abgedeckt sind, werden diese hier gesondert betrachtet.

Funktionale Anforderungen

Die Forderung, dass jedes Git-Repository eine CI-Pipeline mit Tests enthalten soll, wird zu großen Teilen erfüllt. Die Repositories zur Speicherung der erweiterten Images besitzen keine Tests, da es dort keine Code zum Testen gibt. Lediglich bank-frontend Projekt ist, was die Testabdeckung angeht, unvollständig.

Wie bereits im Abschnitt über das Refactoring dargelegt, ist die Sammlung aller Konfigurationsfiles für das HLS-System aus Zeitgründen nicht vorgenommen worden und ist damit nur für die Projekte der Bank-Anwendung erfüllt.

Die Projektdokumentation wurde zu großen Teilen erfüllt. Das zentrale Setup-Dokument und die Hinweise zur Ausführung der Projekte sind umfassend vorhanden, ebenso wie ein FAQ-Dokument je Repository. Wie bereits in der Durchführung diskutiert, wird die Erstellung von FAQ-Bereichen je Dokumentation nur bedingt als sinnvoll betrachtet und sollte erneut überdacht werden.

Die Dokumentation ist mit einem Wiki pro Anwendung und Hilfsrepository klar strukturiert. Zusätzlich besitzen die Repositories der Bank einheitliche Ordnersysteme, sodass NutzerInnen sich dort schneller eingewöhnen können.

Qualitätsanforderungen

Die Qualitätsanforderungen werden in den Projekten voll erfüllt. Alle Docker-Images sind durch multi-stage builds ohne vorige Ausführung von build-Befehlen erstellbar und Skripte werden für alle angestrebten Plattformen zur Verfügung gestellt.

6.2 Validation

Im Kern benötigen NutzerInnen dieses Systems Möglichkeiten zum einfachen Deployment sowie zur Erweiterung und Pflege der Anwendungen und ihrer Dokumentation im Team oder als Einzelperson. Dies wird durch eine längere Einarbeitungszeit und eine komplexe Projektstruktur erschwert und führt zu langsamen Fortschritt. Die Beschleunigung des Lernprozesses durch weiterführenden Informationen im Wiki und die Möglichkeit, Aufgaben leichter aufzuteilen, zählen zu den erwünschten Eigenschaften des Systems. Die Validation beschäftigt sich mit der Frage, inwieweit diese Eigenschaften durch das finale System erfüllt werden. Ohne ausgiebige Tests durch potenzielle NutzerInnen selbst, lässt sich diese Frage nicht endgültig klären. An dieser Stelle werden eigene Erfahrungen aus dem Entwicklungsprozess im Umgang mit dem System als Ausgangspunkt genutzt.

Es ist davon auszugehen, dass die meisten NutzerInnen das System und benötigte Programme zu Beginn herunterladen und installieren müssen. Durch das Setup-Skript wird die Anzahl der von Hand zu klonenden Repositories von sieben auf eines reduziert. Zusätzlich ist die Installation von Docker, NodeJS, kubectl, .NET und Maven nötig, was im Wiki mit allen nötigen Links beschrieben ist. Die Installationen sind umfangreich und kosten Zeit, sind durch die verfügbaren Links jedoch einfach durchzuführen.

Die Bank-Anwendung ist durch die geringe Größe und Aufteilung zum Microservice schnell zu verstehen. Die Zuständigkeiten sind einfach erfassbar und im Wiki gut dokumentiert, sodass die schnelle Einarbeitung in dieses System gegeben ist. Während der Entwicklung gestaltete sich die Arbeit mit der Bank-Anwendung als sehr angenehm und unkompliziert.

Das scripts-Repository als zentrale Anlaufstelle zur Installation und den Skripten zur Verwaltung der Git-Repositories kam sowohl bei Neuinstallationen auf fremden Laptops als auch in der täglichen Entwicklung zum Einsatz. Die Restriktion auf die Arbeit mit dem Master-Branch ist mitunter störend und für die Arbeit im Team mit hoher Wahrscheinlichkeit nicht geeignet. Sofern Gruppenarbeit stattfindet und Aufgaben klar verteilt werden, ist die Nutzung dieser Skripte mit hoher Wahrscheinlichkeit unnötig, da sich die Arbeiten pro Person auf wenige Repositories beschränken werden. Somit ist der Bedarf an solchen geskripteten Hilfen in diesen Fällen nicht gegeben und stellt kein richtiges Problem dar.

Das HLS-System ist mit seiner tiefen Ordnerstruktur das komplexeste der vorhandenen Systeme. Das Frontend ist in dieser Struktur nicht klar vom übrigen System getrennt, sodass die Orientierung teilweise schwerfällt. Auch die Verteilung der Konfigurationsdateien erschwert die Navigation und damit die Einarbeitung. Bei Bedarf kann eine umfassende Dokumentation zurate gezogen werden, sofern die Hinweise aus der zentralen Setup-Datei nicht ausreichen. Allerdings macht die Komplexität des Systems und die bloße Masse an Informationen den Einstieg schwer. Dies hat sich im Projektverlauf nicht verändert und stellt weiterhin ein Problem zur schnellen Einarbeitung dar. Die Repositories mit eigens erstellten Images sind leicht zu erreichen

und beinhalten je ein Dockerfile und ein Wiki. Sie werden selten verändert und sind in ihrem Aufbau einfach, sodass sie schnell verständlich sind.

Mit der Zentralisierung aller Deployment-Files im orchestration-Repository fällt die Wartung der Deployments in der Entwicklung sehr leicht. Die Dokumentation bietet alle wichtigen Informationen zum Zusammenspiel der einzelnen Container und Pods, sodass NutzerInnen schnell in der Lage sind die Struktur zu verstehen und zu Entwicklungszwecken zu nutzen. Das Deployment kann durch die Trigger in der GitLab-CI ohne umfangreiche Kenntnisse durchgeführt werden, sodass die Weiterentwicklung grundsätzlich auch ohne einen hohen Kenntnisstand zur Kubernetes-Technologie durchgeführt werden kann. Dies verschafft neuen NutzerInnen vermutlich Zeit, sich zu Beginn in die Projektstrukturen einzuarbeiten, ohne sich über eine zusätzliche Technologie Gedanken machen zu müssen. Zudem bieten die laufenden Deployments viele Möglichkeiten, sich an bestehenden Lösungen zu orientieren und daraus zu lernen.

Jedes Repository besitzt eine eigene Dokumentation. Als zentralen Sammelpunkt bietet das scripts-Wiki einen Überblick über die übrigen verfügbaren Wikis und nennt stichwortartig die dort behandelten Themen. Auf diese Weise können Studierende sich schnell einen Überblick verschaffen. Problematisch wird die Suche nach Informationen, sobald es thematische Überschneidungen in den Repositories gibt. Beispielsweise enthalten alle Repositories Dockerfiles. Diese Zwischenthemen finden in einer solchen Struktur keinen festen Platz und können dadurch nicht sicher von NutzerInnen gefunden werden.

Da im Wiki nicht nur auf die rein fachliche Dokumentation geachtet wurde, sondern auch viele hilfreiche Befehle und Adressen zur Verfügung gestellt werden, ist der Einstieg in die Technologien und das Debugging von Anwendungen, Containern oder Deployments um einiges erleichtert.

7 Fazit und Ausblick

7.1 Fazit

Das Projekt ist mit dem Ziel, ein automatisches Kubernetes-Deployment und eine nutzerfreundliche Infrastruktur zu schaffen, gestartet. Es sollte die Einarbeitung in die oft noch unbekannte Software erleichtern und Studierende auf ein derzeit gefragtes Thema der IT-Branche vorbereiten.

Besondere Schwierigkeiten brachten hierbei die Einbindung fremder Software und Veränderungen von Nebensystemen während des Projektverlaufes. Dies zeigt, wie wichtig die genaue Beobachtung von Nebensystemen in Bezug auf Änderungen ist und wie schnell sich diese auswirken können. Auch die Ergänzung der Dokumentation ohne eine zuvor erstellte Struktur führte zu Problemen. Dennoch wurde die Dokumentation in ihrem Grundaufbau klar gegliedert und umfassend durch alle Neuerungen ergänzt.

Das automatische Deployment und die docker-compose-Files zur einfacheren Ausführung des Systems wurden mit Ausnahme des HLS-Systems erfolgreich implementiert und bieten den Studierenden viele Möglichkeiten, sich mit diesem derzeit gefragten Themengebiet zu beschäftigen.

7.2 Ausblick

Um das Projekt zu vervollständigen, bietet sich die Korrektur der im HLS-System vorhandenen Fehler an, um spätere Probleme zu vermeiden. Anschließend kann das Deployment gänzlich durchgeführt werden. Auch die Aufteilung des HLS-Systems in Microservices erscheint sinnvoll, wenngleich die Erfahrungen bei gleichzeitiger Bearbeitung eines Monolithen und mehrerer Microservices durchaus lehrreich sein kann.
Der Vorteil, den die Nutzung von Microservices für Deployments bietet, geht durch das vollständige Redeployment in der GitLab-CI verloren. Bei der Weiterentwicklung sollte demnach auf ein Deployment hingearbeitet werden, das nur diejenigen Anwendungen redeployed, die auch verändert wurden. Race-Conditions beim Start der Anwendungen oder des Deployments sollten außerdem durch Health- und Ready-Checks eliminiert werden, um zukünftige Fehler aufgrund von veränderten Startzeiten zu vermeiden.
Zur Erleichterung der Einarbeitung sollte die Dokumentation ergänzt und weiter strukturiert werden. Die angesprochene Problematik mit Themen, die zu mehreren Wikis passen, sollte gelöst werden. Zusätzlich könnte die Installation des Systems weiter vereinfacht werden, um eine bessere Grundlage zur Weiterentwicklung zu schaffen und Fehler durch Nebensysteme zu vermeiden. Denkbar sind Installationsskripte für Windows, Linux und Mac, welche die Anwendungen herunterladen und mit allen nötigen Umgebungsvariablen installieren. Ebenfalls denkbar wäre die Bereitstellung eines Docker-Images, das alle nötigen Installationen enthält. So könnten Studierende unabhängig von ihrem Betriebssystem in garantiert der gleichen Entwicklungsumgebung arbeiten, was Lehrkräften mit hoher Wahrscheinlichkeit Wartungs- und Betreuungsaufwand einsparen würde.
Das scripts-Repository sollte dahingehend ergänzt werden, dass die Arbeit auf zusätzlichen Branches beispielsweise durch Parameterübergabe bei Ausführung der Skripte möglich ist. Hierzu sollte jedes Repository den gleichen Branch nutzen (z. B. develop), um verschiedene Versionsierungsworkflows wie GitFlow zu ermöglichen.

Literaturverzeichnis

[Angular 2018] Angular: *QuickStart*. 2018. – URL https://angular.io/guide/quickstart

[Atlassian 2018] Atlassian: *GitFlow Workflow*. 2018. – URL https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

[Augsten 2017] Augsten, Stephan: *Was ist Continuous Deployment?* 2017. – URL https://www.dev-insider.de/was-ist-continuous-deployment-a-652804/

[Baier 2017] Baier, Jonathan: *Getting Started with Kubernetes, Second Edition*. Packt Publishing, 2017. – URL https://www.ebook.de/de/product/29271176/jonathan_baier_getting_started_with_kubernetes_second_edition.html. – ISBN 1787283364

[Behnke 2018] Behnke, Lutz: *Die Suche nach der richtigen Image Registry*. 2018. – URL https://icc.informatik.haw-hamburg.de/blog/2018/05/10/die-richtige-registry.html

[Berngruber 2017] Berngruber, Fritz Oscar S.: *Integration von DevOps in den laufenden Projektbetrieb am Beispiel des delegs-Forschungsprojekts*. 2017. – URL http://edoc.sub.uni-hamburg.de/haw/volltexte/2017/4074/

[Brunner 2017] Brunner, Franz J.: *Japanische Erfolgskonzepte*. Carl Hanser Verlag GmbH & Co. KG, 2017. – URL https://www.ebook.de/de/product/29260298/japanische_erfolgskonzepte.html

[cAdvisor 2018] cAdvisor: *cAdvisor*. 2018. – URL https://github.com/google/cadvisor

[Chapman u. a. 2008] Chapman, Christopher N. ; Love, Edwin ; Milham, Russell P. ; ElRif, Paul ; Alford, James L.: Quantitative Evaluation of Personas as Information. In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 52 (2008), sep, Nr. 16, S. 1107–1111

[Cockburn 2001] Cockburn, Alistair: *Agile Software Development*. Addison-Wesley Professional, 2001. – URL https://www.amazon.com/Agile-Software-Development-Alistair-Cockburn/dp/0201699699?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0201699699. – ISBN 0201699699

[Cockburn 2010] Cockburn, Alistair: *Use Cases effektiv erstellen*. Verlagsgruppe Hüthig Jehle Rehm, 2010. – URL https://www.ebook.de/de/product/7066125/alistair_cockburn_use_cases_effektiv_erstellen.html. – ISBN 3826617967

[Diedrich 2018] Diedrich, Sebastian: *Konzeption und prototypische Realisierung einer Microservice Architektur zur graph-basierten Speicherung von medizinischen Gesundheits- und Abrechnungsdaten zwecks Analyse und Auswertung*. 2018. – URL http://edoc.sub.uni-hamburg.de/haw/volltexte/2018/4268/

[Docker 2018a] Docker: *Get Started, Part 2: Containers*. 2018. – URL https://docs.docker.com/get-started/part2/#dockerfile

[Docker 2018b] Docker: *Overview of Docker Compose*. 2018. – URL https://docs.docker.com/compose/overview/

[Docker 2018c] Docker: *Use multi-staged builds*. 2018. – URL https://docs.docker.com/develop/develop-images/multistage-build/#use-multi-stage-builds

[Ebert 2014] Ebert, Christof: *Risikomanagement kompakt*. Springer-Verlag GmbH, 2014. – URL https://www.ebook.de/de/product/21796262/christof_ebert_risikomanagement_kompakt.html. – ISBN 3642410472

[etcd 2018] etcd: *etcd*. 2018. – URL https://github.com/coreos/etcd

[Fowler 2006] Fowler, Martin: *Continuous Integration*. 2006. – URL https://www.martinfowler.com/articles/continuousIntegration.html. – Zugriffsdatum: 2018-06-01

[Freepik 2018] Freepik: *Icons*. 2018. – URL http://www.freepik.com

[GitHub 2018] GitHub: *5.0.0-beta.14-17a70ee require Angular 6*. 2018. – URL https://github.com/angular/flex-layout/issues/714

[GitLab 2018a] GitLab: *Ci/CD Variables*. 2018. – URL https://docs.gitlab.com/ee/ci/variables/#variables

[GitLab 2018b] GitLab: *Deploy keys*. 2018. – URL https://docs.gitlab.com/ce/ssh/README.html#deploy-keys

[GitLab 2018c] GitLab: *Triggering pipelines through the API*. 2018. – URL https://docs.gitlab.com/ee/ci/triggers/

[H2 2018a] H2: *H2 Database Engine*. 2018. – URL http://www.h2database.com/html/main.html

[H2 2018b] H2: *In-Memory Databases*. 2018. – URL http://www.h2database.com/html/features.html#in_memory_databases

[HAW 2016a] HAW: *Modulhandbuch Angewandte Informatik*. 2016. – URL https://www.haw-hamburg.de/fileadmin/user_upload/TI-I/PDF/modulhandbuecher/Modulhandbuch_AI_2016_10_18.pdf

[HAW 2016b] HAW: *Modulhandbuch Technische Informatik*. 2016. – URL https://www.haw-hamburg.de/fileadmin/user_upload/TI-I/PDF/modulhandbuecher/Modulhandbuch_TI_2016_10_18.pdf

[HAW 2016c] HAW: *Modulhandbuch Wirtschaftsinformatik*. 2016. – URL https://www.haw-hamburg.de/fileadmin/user_upload/TI-I/PDF/modulhandbuecher/Modulhandbuch_WI_2016_10_18.pdf

[Hernandez 2018] Hernandez, Alfredo: *Repository Icons*. 2018. – URL https://www.flaticon.com/authors/alfredo-hernandez

[Hüning und Behnke 2018a] Hüning, Christian ; Behnke, Lutz: *Arbeiten in Gruppen: Synchronisation von Gitlab mit Kubernetes*. 2018. – URL https://userdoc.informatik.haw-hamburg.de/doku.php?id=docu:informatikcomputecloud#arbeiten_in_gruppensynchronisation_von_gitlab_mit_kubernetes

[Hüning und Behnke 2018b] Hüning, Christian ; Behnke, Lutz: *Informatik Compute Cloud (ICC)*. 2018. – URL https://userdoc.informatik.haw-hamburg.de/doku.php?id=docu:informatikcomputecloud

[Jeff Patton 2015] Jeff Patton, Peter E.: *User Story Mapping - Die Technik für besseres Nutzerverständnis in der agilen Produktentwicklung*. O'Reilly Vlg. GmbH & Co., 2015. – URL https://www.ebook.de/de/product/23507334/jeff_patton_peter_economy_user_story_mapping_die_technik_fuer_besseres_nutzerverstaendnis_in_der_agilen_produktentwicklung.html. – ISBN 3958750672

[Kannan und Marmol 2015] Kannan, Vishnu ; Marmol, Victor: *Resource Usage Monitoring Kubernetes*. 2015. – URL https://kubernetes.io/blog/2015/05/resource-usage-monitoring-kubernetes/

[Klaus Pohl 2015] Klaus Pohl, Chris R.: *Basiswissen Requirements Engineering*. Dpunkt.Verlag GmbH, 2015. – URL https://www.ebook.de/de/

product/23907285/klaus_pohl_chris_rupp_basiswissen_requirements_engineering.html. – ISBN 3864902835

[Kniberg 2011] Kniberg, Hanrik: *Learn From The Trenches*. 2011. – URL https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0ahUKEwjGre_58svbAhWME5oKHSohAssQFggrMAA&url=https%3A%2F%2Fwww.crisp.se%2Ffile-uploads%2FLean-from-the-trenches.pdf&usg=AOvVaw2NURMG8JaNUtoyg_ekseU1

[Kubernetes 2018a] Kubernetes: *Accessing Clusters*. 2018. – URL https://kubernetes.io/docs/tasks/access-application-cluster/access-cluster/

[Kubernetes 2018b] Kubernetes: *Connecting Applications with Services*. 2018. – URL https://kubernetes.io/docs/concepts/services-networking/connect-applications-service/

[Kubernetes 2018c] Kubernetes: *Deployments*. 2018. – URL https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[Kubernetes 2018d] Kubernetes: *Kube Controller Manager*. 2018. – URL https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/

[Kubernetes 2018e] Kubernetes: *kube-proxy*. 2018. – URL https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/

[Kubernetes 2018f] Kubernetes: *kube-scheduler*. 2018. – URL https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/

[Kubernetes 2018g] Kubernetes: *Kubelet*. 2018. – URL https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/

[Kubernetes 2018h] Kubernetes: *Kubernetes objects*. 2018. – URL https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/

[Kubernetes 2018i] Kubernetes: *Labels, Deployments, Services and Health Checking*. 2018. – URL https://kubernetes.io/docs/tutorials/k8s201/#health-checking

[Kubernetes 2018j] Kubernetes: *Master Components*. 2018. – URL https://kubernetes.io/docs/concepts/overview/components/#master-components

[Kubernetes 2018k] Kubernetes: *Minikube*. 2018. – URL https://github.com/kubernetes/minikube

[Kubernetes 2018l] Kubernetes: *Nodes*. 2018. – URL https://kubernetes.io/docs/concepts/architecture/nodes/

[Kubernetes 2018m] Kubernetes: *Pod Overview*. 2018. – URL https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/

[Kubernetes 2018n] Kubernetes: *Production-Grade Container Orchestration*. 2018. – URL https://kubernetes.io/

[Kubernetes 2018o] Kubernetes: *Pull an Image from a Private Registry*. 2018. – URL https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/

[Kubernetes 2018p] Kubernetes: *Services*. 2018. – URL https://kubernetes.io/docs/concepts/services-networking/service/

[Kubernetes 2018q] Kubernetes: *Use Port Forwarding to Access Applications in a Cluster*. 2018. – URL https://kubernetes.io/docs/tasks/access-application-cluster/port-forward-access-application-cluster/

[Lenzo 2016] Lenzo, Marco: *Continuous delivery of a Spring Boot application with GitLab CI and Kubernetes*. 2016. – URL https://about.gitlab.com/2016/12/14/continuous-delivery-of-a-spring-boot-application-with-gitlab-c

[Lucy-g 2018] Lucy-g: *User Icon*. 2018. – URL https://www.flaticon.com/authors/lucy-g

[Maven 2018] Maven: *Introduction*. 2018. – URL https://maven.apache.org/

[Microsoft 2018] Microsoft: *What is .NET?* 2018. – URL https://www.microsoft.com/net/learn/what-is-dotnet

[Monkik 2018] Monkik: *Avatare*. 2018. – URL https://www.flaticon.com/authors/monkik

[Mouat 2016] Mouat, Adrian: *Docker*. Dpunkt.Verlag GmbH, 2016. – URL https://www.ebook.de/de/product/26153698/adrian_mouat_docker.html. – ISBN 386490384X

[Neo4J 2018] Neo4J: *Neo4J*. 2018. – URL https://neo4j.com/

[Newman 2015] Newman, Sam: *Microservices*. MITP Verlags GmbH, 2015. – URL https://www.ebook.de/de/product/23955218/sam_newman_microservices.html. – ISBN 3958450814

[Nginx 2018a] Nginx: *Least connectad load balancing*. 2018. – URL http://nginx.org/en/docs/http/load_balancing.html#nginx_load_balancing_with_least_connected

[Nginx 2018b] NGINX: *Load Balancing*. 2018. – URL https://www.nginx.com/resources/glossary/load-balancing/

[Nginx 2018c] NGINX: *What is Round-Robin Load Balancing?* 2018. – URL https://www.nginx.com/resources/glossary/round-robin-load-balancing/

[Oracle 2012] ORACLE: *De-mystifying "eventual consistency" in distributed systems*. 2012. – URL https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=6&ved=0ahUKEwjksdfdwu7bAhVSb1AKHX57CfQQFghYMAU&url=http%3A%2F%2Fwww.oracle.com%2Ftechnetwork%2Fproducts%2Fnosqldb%2Fdocumentation%2Fconsistency-explained-1659908.pdf&usg=AOvVaw1ooAqD6uwxgiGAiZ5OIpvU

[Osterwalder u. a. 2015] OSTERWALDER, Alexander ; PIGNEUR, Yves ; BERNARDA, Greg ; SMITH, Alan: *Value Proposition Design*. Campus Verlag GmbH, 2015. – URL https://www.ebook.de/de/product/23320320/alexander_osterwalder_yves_pigneur_greg_bernarda_alan_smith_value_proposition_design.html. – ISBN 359350331X

[PostgreSQL 2018] POSTGRESQL: *PostgreSQL*. 2018. – URL https://www.postgresql.org/

[Pundsack 2016] PUNDSACK, Mark: *GitLab Container Registry*. 2016. – URL https://about.gitlab.com/2016/05/23/gitlab-container-registry/

[RabbitMQ 2018] RABBITMQ: *RabbitMQ*. 2018. – URL https://www.rabbitmq.com/

[Ramos 2016] RAMOS, Marcia: *Continuous Integration, Delivery, and Deployment with GitLab*. 2016. – URL https://about.gitlab.com/2016/08/05/continuous-integration-delivery-and-deployment-with-gitlab/

[van Randen u. a. 2016] Randen, Hendrik J. van ; Bercker, Christian ; Fieml, Julian ; Randen, Hendrik J. van: *Einführung in UML*. Gabler, Betriebswirt.-Vlg, 2016. – URL https://www.ebook.de/de/product/26717631/hendrik_jan_van_randen_christian_bercker_julian_fieml_hendrik_jan_van_randen_einfuehrung_in_uml.html. – ISBN 3658144114

[Richardson 2015] Richardson, Chris: *Service Discovery in a Microservices Architecture*. 2015. – URL https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/

[Simons 2018] Simons, Michael: *Spring Boot 2*. Dpunkt.Verlag GmbH, 2018. – URL https://www.ebook.de/de/product/30014133/michael_simons_spring_boot_2.html. – ISBN 3864905257

[Starke 2011] Starke, Gernot: *Effektive Software-Architekturen*. Carl Hanser Verlag GmbH & Co. KG, aug 2011

[StickPNG 2018] StickPNG: *Kubernetes Logo*. 2018. – URL http://www.stickpng.com/img/icons-logos-emojis/tech-companies/kubernetes-logo

[Swartout 2014] Swartout, Paul: *Continuous Delivery and Devops - A QuickStart Guide Second Edition*. PACKT PUB, 2014. – URL https://www.ebook.de/de/product/23550379/paul_swartout_continuous_delivery_and_devops_a_quickstart_guide_second_edition.html. – ISBN 1784399310

[Thommen 2018] Thommen, Prof. Dr. Jean-Paul: *Definition Anspruchsgruppen*. 2018. – URL https://wirtschaftslexikon.gabler.de/definition/anspruchsgruppen-27010/version-250673

[Thompson 2017] Thompson, Joe: *Think Before you NodePort in Kubernetes*. 2017. – URL https://oteemo.com/2017/12/12/think-nodeport-kubernetes/

[Tilkov u. a. 2015] Tilkov, Stefan ; Eigenbrodt, Martin ; Schreier, Silvia ; Wolf, Oliver: *REST und HTTP*. Dpunkt.Verlag GmbH, 2015. – URL https://www.ebook.de/de/product/21290987/stefan_tilkov_martin_eigenbrodt_silvia_schreier_oliver_wolf_rest_und_http.html. – ISBN 3864901200

[Williams 2017] Williams, Elisa: *What is the Meaning of Deployment in Software*. 2017. – URL https://pdf.wondershare.com/business/what-is-software-deployment.html

[Wilson 2003] Wilson, James M.: Gantt charts: A centenary appreciation. In: *European Journal of Operational Research* 149 (2003), sep, Nr. 2, S. 430–437

Index

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Hamburg, 22. Juli 2018

Mieke Narjes