HAW HAMBURG

Bachelorarbeit

Lion Pierau

Konzept und Umsetzung eines Tenant Integrators zur Zustandsvorgabe einer Multi-Tenancy in Kubernetes

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Lion Pierau

Konzept und Umsetzung eines Tenant Integrators zur Zustandsvorgabe einer Multi-Tenancy in Kubernetes

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. Olaf Zukunft

Eingereicht am: 08. September 2021

Lion Pierau

Thema der Arbeit

Konzept und Umsetzung eines Tenant Integrators zur Zustandsvorgabe einer Multi-Tenancy in Kubernetes

Stichworte

Multi-Tenancy, Tenant, Kubernetes, GitLab

Kurzzusammenfassung

Ziel dieser Arbeit ist die Umsetzung eines Tenant Integrators für Kubernetes. Dieser soll Studenten aus GitLab als Tenant-Ressource in Kubernetes speichern und synchronisieren. Diese Tenant-Ressourcen beschreiben den Soll-Zustand für eine Multi-Tenancy in Kubernetes und weisen Studenten so ein Stück des Clusters zu. Ein besonderer Fokus liegt dabei auf die Erweiterbarkeit und Zuverlässigkeit des Integrators. Der Integrator sollte ohne Ausfälle in der Informatik Compute Cloud der HAW betrieben werden können und auch in Zukunft leicht erweiterbar sein.

Lion Pierau

Title of Thesis

Concept and implementation of a tenant integrator for state specification of a multi-tenancy in Kubernetes

Keywords

Multi-Tenancy, Tenant, Kubernetes, GitLab

Abstract

The goal of this thesis is the implementation of a tenant integrator for Kubernetes. This should store and synchronize students from GitLab as tenant resource in Kubernetes. These tenant resources describe the target state for a multi-tenancy in Kubernetes and assign students a piece of the cluster. A particular focus is on the extensibility and

reliability of the integrator. The integrator should be able to run in HAW's Informatics Compute Cloud without failures and should be easily extensible in the future.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

CRD Custom Resource Definition.

Go Golang.

HAW Hochschule für Angewandte Wissenschaften.

ICC Informatik Compute Cloud.

k8s Kubernetes.

1 Einleitung

1.1 Problemstellung und Motivation

In der internen Cloud Umgebung, der sogenannten Informatik Compute Cloud (ICC) der Hochschule für Angewandte Wissenschaften (HAW), wird Kubernetes als Orchestrierungstool für containerisierte Cloud-Anwendungen genutzt. Dabei nutzen eine Vielzahl von Studenten Kubernetes zur Bereitstellung ihre Anwendungen. Im Gegensatz zu den meisten Unternehmen, bei denen alle Mitarbeiter zusammen an einem System arbeiten, entwickeln Studenten ihre Anwendungen eher alleine oder in kleinen Gruppen. Ein Zugriff auf die eigenen Anwendungen von nicht involvierten Kommilitonen ist meistens nicht gewünscht.
Eine denkbare Lösung, um Studenten separierten Zugriff zu ermöglichen, wäre es jedem ein eigenes Cluster zur Verfügung zu stellen. Bei dem benötigten Verwaltungsaufwand für Kubernetes ist dies jedoch keine akzeptable Lösung. Aus diesem Grund muss das vorhandene Cluster für Studenten durch Zugriffsbeschränkungen nutzbar gemacht werden.
Eine Möglichkeit zur Unterteilung des Clusters in Kubernetes sind Namespaces. Diese bieten nur eine einfache technische Trennung, indem sie den Anwendungsbereich der Ressourcen festlegen. Zudem bieten sie eine logische Trennung für den Entwickler, indem die Sichtbarkeit auf Systeme im Cluster nach Namespaces separiert darstellt werden. Jedoch wird dadurch keine strikte Trennung zu Anwendungen in anderen Namespaces geboten. Ressourcen in einem Namespace können durch Auslastung von Hardwareressourcen oder Netzwerkkommunikation immer noch Auswirkungen auf Ressourcen in einem anderen Namespace haben.
Diese Form der Unterteilung ist für die meisten Unternehmen ausreichend, da es dadurch möglich ist Teams, Projekte und verschiedene Systeme logisch voneinander zu trennen. Wird jedoch, wie im Fall der HAW, eine Unterteilung für viele unterschiedliche Interessensgruppen benötigt, sind Namespaces nicht ausreichend. Für die Trennung von Nutzern, die voneinander isoliert ein Cluster nutzen wollen, wird eine Abstraktionsebene

über den Namespaces benötigt. Für den Nutzer sollte es nicht möglich sein zu unterscheiden, ob er sich in einem eigenen Cluster befindet, oder er es mit anderen Nutzern teilt; außer er berechtigt diese dazu. Diese Funktionalität eines Systems wird als Multi-Tenancy bezeichnet. Ein Tenant ist dabei ein Nutzer oder eine Gruppe von Nutzern, die eine Sicht auf das Cluster teilen, also zusammen ein virtuelles Cluster nutzen.

1.2 Zielsetzung

Als ersten Schritt zur Multi-Tenancy in der ICC ist es notwendig die Tenant-Objekte zu identifizieren und diese mit allen notwendigen Informationen bereitzustellen, so dass auf dessen Grundlagen eine Multi-Tenancy umgesetzt werden kann. Die HAW verwaltet eine GitLab Instanz, in der Studenten Nutzerkonten besitzen und ihre Projekte verwalten. Daher soll GitLab als Informationsquelle genutzt werden.

Ziel dieser Arbeit ist es, eine Anwendung zu konzipieren und zu implementieren, die durch Nutzungsinformationen aus GitLab eine geeignete Abbildung zu Tenant-Objekten für die ICC schafft. Alle Informationen zu einem Tenant sollen in Kubernetes gespeichert und aktuell gehalten werden. Anhand der gespeicherten Informationen soll es möglich sein eine Anwendung zu entwickeln, die eine Multi-Tenancy umsetzt. Dabei sollte einem Kernkonzept von Kubernetes gefolgt werden, bei dem der gewünschte Zustand beschrieben, und anhand der Beschreibung automatisch umgesetzt werden kann. Es müssen also alle benötigten Informationen bereits in dem Tenant-Objekt vorhanden sein, damit nicht auf weitere Informationsquellen zugegriffen werden muss.

Zum Zeitpunkt der Arbeit existiert bereits ein Integrator in der ICC, welcher allerdings durch Ausfälle und fehlende Synchronisation einige Fehler aufweist. Daraus ergeben sich für diese Arbeit einige aus Erfahrung gewonnene Anforderungen. Unter anderem sollte der neue Integrator zuverlässig und dauerhaft erreichbar sein. Zudem ist der aktuelle Integrator ein Monolith und dadurch nur umständlich erweiterbar. Es ist Beispielsweise nur schwer möglich andere Informationsquellen als GitLab anzubinden. Aus diesem Grund soll der neue Integrator aus einem generischen Kern bestehen und es dadurch möglich machen ihn für beliebige Informationsquellen zu erweitern. Zusammenfassend lässt sich sagen, dass bei der Entwicklung des Systems eine qualitativ hochwertige Anwendung erzielt werden soll. Der existierende Integrator wird dabei nicht als Vorlage genutzt.

Die Anwendung für die Umsetzung der Multi-Tenancy wird des Weiteren als **Tenant-Manager** bezeichnen. Diese wird vorgeben, welche Informationen für einen Tenant benötigt werden. Die Entwicklung dieser Anwendung wird **nicht** Teil dieser Arbeit sein und eine erste Definition für ein Tenant-Objekt wurde bereits erarbeitet. Auf Möglichkeiten zur Optimierung dieses Objektes wird jedoch eingegangen.
Die zu entwickelnde Anwendung für die Abbildung und Speicherung der definierten Tenant-Objekten wird des Weiteren als **Tenant-Integrator** bezeichnet.

1.3 Gliederung der Arbeit

1.3.1 Kapitel 1, Einleitung

Im ersten Kapitel wird die Zielsetzung dieser Arbeit näher beleuchtet, die zugrundeliegende Motivation und der Aufbau dieser Arbeit.

1.3.2 Kapitel 2, Grundlagen

In diesem Kapitel werden die Grundlagen erklärt, die für das Verständnis dieser Bachelorarbeit notwendig sind. Dazu gehört Multi-Tenancy, Grundlagen der Anforderungsanalyse, GitLab, Docker-Container und Kubernetes. Das Verständnis von Dingen, die in der Hochschule gelehrt werden, wird vorausgesetzt.

1.3.3 Kapitel 3, Anforderungsanalyse

Das dritte Kapitel beschreibt unter anderem das Erfassen von Nutzerrollen und die sich daraus resultierenden funktionalen Anforderungen. Dadurch soll sichergestellt werden, dass das zu implementierende System alle notwendigen Funktionalitäten erfüllt. Darüber hinaus wurden Qualitätsanforderungen gesammelt, um unter anderem die Verständlichkeit, Wartbarkeit und Performance des Systems zu verbessern.

1.3.4 Kapitel 4, Systementwurf

Im vierten Kapitel wird ein Konzept des System entworfen und auf die wichtigsten Entwurfsentscheidungen eingegangen.

1.3.5 Kapitel 5, Realisierung

In diesem Kapitel wird die Umsetzung des Tenant Integrators erläutert, welche im Systementwurf konzipiert wurde. Dabei wird nur auf die wichtigsten Teile des realisierten Endproduktes eingegangen werden.

1.3.6 Kapitel 6, Evaluation

Im letzten Kapitel werden die erhobenen Anforderungen verifiziert und es wird validiert inwieweit dies den Erwartungen der zukünftigen Nutzern entspricht. Zudem wird die Arbeit zusammenfassend bewertet und Zukunftsperspektiven diskutiert. Dazu werden verschiedene Vorschläge zur Weiterentwicklung des Projekts erhoben.

2 Grundlagen

2.1 Multi-Tenancy

Im Cloud-Computing ist **Multi-Tenancy** ein Architekturstil, bei dem eine Instanz mehrere Kunden bedient. Der häufigste Grund für die Nutzung dieses Architekturstils ist es, Hardwareressourcen einzusparen und das eigene Produkt damit günstiger anbieten zu können. Dabei ist ein **Tenant** ein Nutzer oder eine Gruppe von Nutzern, die dieselbe Sicht auf die Anwendung teilen. (vgl. Krebs, Momm and Kounev, Kapitel 1 und 2.2) Im Fall der ICC sind Tenants einzelne Studenten und Professoren, aber auch Arbeitsgruppen für Projekte oder Praktika.

In einem Dokument der Arbeitsgruppe für Multi-Tenancy von Google wird Multi-Tenancy von (Jessie Frazelle) wie folgt in zwei Gruppen aufteilen:

Soft Multi-Tenancy Mehrere Nutzer aus der selben Organisation im selben Cluster. Benutzer werden nicht als aktiv böswillig eingestuft, da sie sich innerhalb derselben Organisation befinden, aber als potenziell unfallgefährdet oder "bösartig", wenn sie das Unternehmen verlassen. Ein großer Schwerpunkt der Soft Multi-Tenancy liegt auf der Vermeidung von Unfällen.

Hard Multi-Tenancy Mehrere Nutzer von verschiedensten Orten in demselben Cluster. Jeder im Cluster wird als potentieller Übeltäter eingestuft und daher sollte niemand Zugriff auf Ressourcen von anderen Tenants haben.

2.2 Anforderungsanalyse

2.2.1 Nutzerrollen (User Roles)

> „A **user role** is a collection of defining attributes that characterize a population of users and their intended interactions with the system.“ (Cohn, 2004, Seite 32)

Durch das Analysieren von Nutzerrollen wird sichergestellt, dass keine Anforderungen vergessen werden und die Anwendung auf alle Benutzer abgestimmt ist. Jede Nutzerrolle bringt ihre eigenen Anforderungen mit und ohne sie würden auch die Anforderungen vergessen werden.

Normalerweise gibt es Überschneidungen bei den verschiedenen Nutzerrollen. Aus diesem Grund gibt es die folgenden Schritte, um eine nützliche Menge von Nutzerrollen zu identifizieren (vgl. Cohn, 2004, Seite 33-37):

Brainstorming Um eine Anfangsmenge von Nutzerrollen zu identifizieren, kann einfaches Brainstorming benutzt werden. Am besten bringt man dafür den Kunden und so viele Entwickler wie möglich an einem Tisch oder einer Wand zusammen. Dabei sollen ohne Diskussionen Nutzerrollen gesammelt werden. Dies sollte bis zu dem Punkt gemacht werden an dem es jedem schwerfällt sich neue Nutzerrollen einfallen zu lassen. Meist dauert dies nicht länger als fünfzehn Minuten.

Organisieren Wenn die Gruppe fertig ist, die Nutzerrollen zu identifizieren, ist es Zeit diese zu organisieren. Dies wird gemacht, indem die Karten so auf dem Tisch oder der Wand verschoben werden, dass ihre Position die Beziehung zwischen diesen Karten verrät. Dies kann so realisiert werden, dass ähnliche Karten sich überlappen und um so stärker die Ähnlichkeit dieser Karten, desto größer die Überlappung.

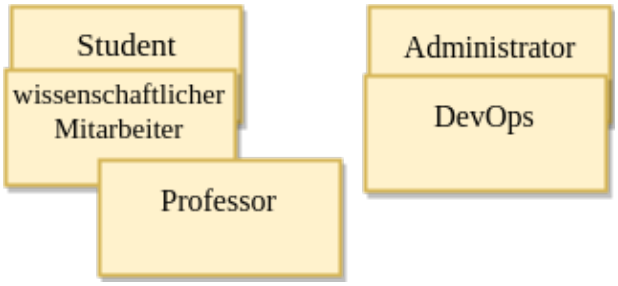

Abbildung 2.1: Beispiel für das Organisieren von Nutzerrollen.
(Angelehnt an: Cohn, 2004, Seite 34)

Bei Karten mit möglichen gleichen Nutzerrollen liegen diese also übereinander, wie z.B. in der Abbildung 2.1 der *Student* und der *wissenschaftliche Mitarbeiter*. Dabei sind zwei Nutzerrollen gleich, wenn sie die selben Anforderungen an unsere Anwendung mitbringen. Die Karten sollten so platziert werden, dass der Schriftzug trotz Überlappung noch sichtbar ist.

Konsolidieren Nach dem Gruppieren geht es darum, aussagekräftige Nutzerrollen aus den gruppierten Karten auszuarbeiten. Dazu wird am besten bei den Karten gestartet, die sich komplett überlappen. Zu Anfang erklären die Verfasser der überlappenden Karten noch einmal die Nutzerrollen, um Missverständnisse zu vermeiden. Nach einer kurzen Diskussion mit der Gruppe, ob die Karten gleich sind, können die Karten entweder zusammengefasst werden oder eine der beiden Karten fliegt in den Papierkorb. Je nachdem, ob eine der beiden Karten ausreicht, die Nutzerrolle zu beschreiben oder eine grobere Beschreibung gefunden werden muss. Aus Abbildung 2.1 könnten z.B. der *Administrator* und *DevOps* zusammengefasst werden. Zu sehen ist dies in Abbildung 2.2. Dabei wurde sich z.B. dazu entschieden, den *Administrator* zu behalten und *DevOps* zu entfernen. Das Team hätte sich aber genauso gut für *DevOps* oder einen gänzlich anderen Begriff entscheiden können. Wichtig ist nur, dass der Begriff den Nutzer der Anwendung möglichst gut repräsentiert und dies ist kontextabhängig.

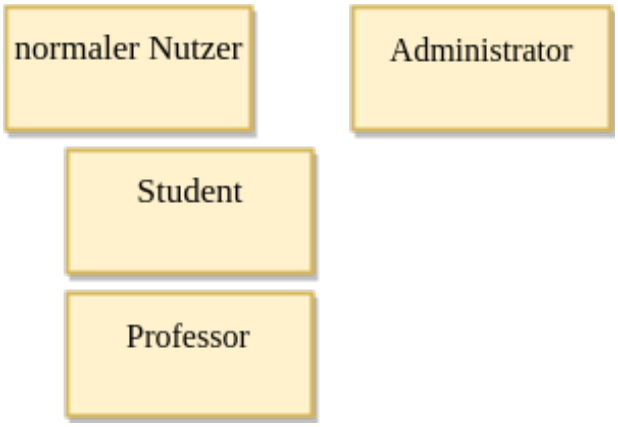

Abbildung 2.2: Beispiel für das Konsolidieren.
(Angelehnt an: Cohn, 2004, Seite 36)

Als Gegenbeispiel zum *Administrator*, wurde sich in Beispiel 2.2 dazu entschieden den *Studenten* und den *Professor* zu behalten, da diese sich in einzelnen aber signifikanten Fällen unterscheiden und andere Anforderungen mit sich bringen. Zudem wurde den beiden für ihre gemeinsamen Anforderungen eine übergeordnete Rolle zugeteilt. Im Beispiel 2.2 ist dies der *normale Nutzer*, je nach Kontext der Anwendung kann auch ein treffenderer Name gefunden werden. Der *wissenschaftliche Mitarbeiter* hingegen wurde entfernt, da er die selben Anforderungen wie der *Professor* hat. Im Anschluss sollte die Gruppe alle Karten in den Papierkorb befördern, die unwichtig für den Erfolg der Anwendung sind.

Verfeinern Nachdem nun die finalen Nutzerrollen beim Konsolidieren erfasst wurden und ein grundlegendes Verständnis geschaffen wurde, wie diese Rollen miteinander in Beziehung stehen, ist es Zeit die Rollen zu modellieren, indem ihnen Merkmale gegeben werden. Dabei ist jedes Merkmal hilfreich, dass die Nutzerrolle von anderen Rollen unterscheidet. Hier sind einige Beispiele:

- Die Häufigkeit in der der Nutzer die Anwendung nutzt.
- Die Erfahrung des Nutzers in dem Anwendungsbereich der Software.
- Den Grad von Fertigkeiten die der Nutzer für Computer und Software mitbringt.
- Das allgemeine Ziel des Benutzers für die Nutzung der Software.

Über die Standardmerkmale hinaus, sollten weitere Merkmale in Erwägung gezogen werden, die für die zu entwickelnde Anwendung von Bedeutung sein könnten.

2.2.2 User Stories

Mit User Stories wird ein Problem von traditionellen Anforderungskatalogen gelöst, bei denen genau beschrieben wurde, was man wollte. Dabei lesen unterschiedliche Leute dieses Dokument und stellen sich im Zweifel darunter völlig unterschiedliche Dinge vor. Erst mitten in der Entwicklung der Software, oder sogar erst bei der Auslieferung, wurden diese Unstimmigkeiten klar. Um diesem Problem aus dem Weg zu gehen, steht bei User Stories das gemeinsame Verständnis im Mittelpunkt und nicht das Schreiben detaillierter Anforderungen (vgl. Patton, 2015, 97-99).

Eine User Story beschreibt eine Anforderung, die einen Wert für den Nutzer oder den Kunden der Software hat, und soll zu Gesprächen anregen, damit alle dasselbe Bild von einer Anforderung bekommen (vgl. Cohn, 2004, Seite 4).

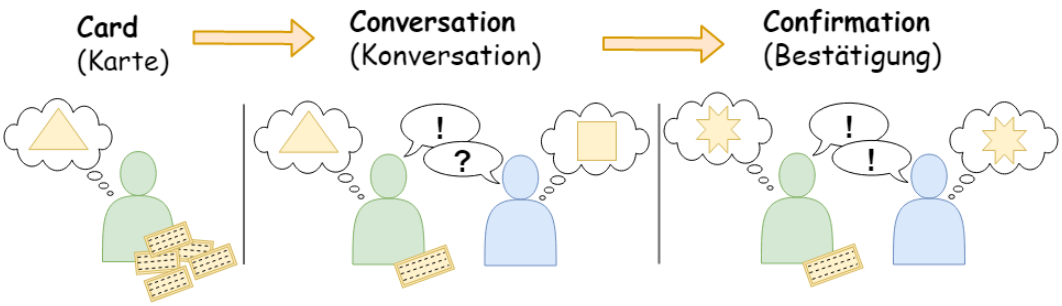

Abbildung 2.3: Ron Jeffries drei Cs.
(Angeleht an: Patton, 2015, Seite 101)

Ron Jeffries (2000) beschrieb den Prozess von User Stories sehr einprägsam in einer Alliteration mit den drei Aspekten *"Card, Conversation and Confirmation"*:

Card (Karteikarte) Ist eine visuelle Repräsentation der Anforderung, die für die Planung als Erinnerung verwendet wird. Dabei sollen auf dessen Grundlage die genauen Details diskutierbar sein und nicht dokumentiert werden. Die Anforderung kann auch digital festgehalten werden, jedoch wurden User Stories traditionell handgeschrieben auf Karten notiert.

Conversation (Konversation) Gespräche über die Story, die dazu dienen, die Details der Story zu konkretisieren und dazu führen sollen, dass alle dieselbe Vorstellung bekommen.

Confirmation (Bestätigung) Die ausdiskutierten Details werden dann als Akzeptanztests festgehalten; meist auf der Rückseite der Karte. Durch die Akzeptanztests ist auch der Fertigstellungsgrad einer Story bestimmbar. Das frühe Definieren der Tests ist hilfreich, da dadurch mehr Annahmen und Erwartungen des Kunden an die Entwickler weitergegeben werden und der Kunde automatisch auf Spezialfälle hinweist.

User Stories bieten gegenüber Lastenheften eine Vielzahl von Vorteilen:

- Regen zur verbalen Kommunikation an.
- Sind für Kunden und Entwickler verständlich.
- Haben die richtige Größe für die Planung.
- Sind in der agilen Entwicklung nutzbar.
- Ermutigen die Details zu verschieben, bis das beste Verständnis dafür da ist.

(vgl. Cohn, 2004, 4, 12-14)

Um beim Schrieben von User Stories keine wichtigen Informationen zu vergessen, empfiehlt es sich Vorlagen zu verwenden. Von den ersten Verwendern der User Stories wurde dabei folgendes Format geprägt:

Als <*Rolle*> möchte ich <*Ziel/Wunsch*>, um <*Nutzen*>

(vgl. Cohn, 2004, 81)

Wenn Stories zu groß sind, werden diese häufig als **Epic** bezeichnet. Epics können in zwei oder mehr kleinere Stories zerteilt werden. Es sollte dabei jedoch darauf geachtet werden, dass die Stories nicht zu klein werden und bereits schon alle Details enthalten, denn diese sollen, wie schon erwähnt, durch die Karte diskutierbar sein. (vgl. Cohn, 2004, 6)

2.2.3 User Story Mapping

User Story Mapping ist eine Technik, mit der ein Big Picture der User Stories ermöglicht wird. Es wird üblicherweise mit dem grundlegenden Erzählfluss der Story begonnen und die Karten in einer Ebene ausgelegt.

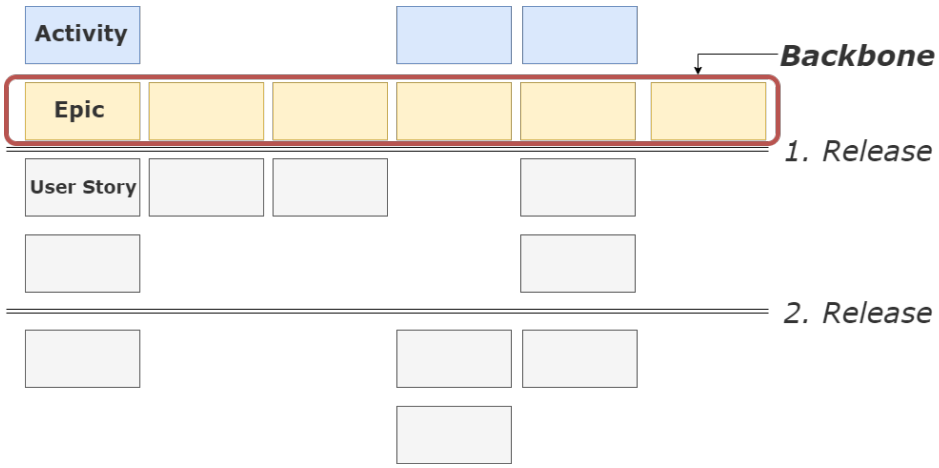

Abbildung 2.4: Überblick über die Story Mapping Technik.

Dabei werden die Karten so von links nach rechts ausgelegt, dass sie eine Story erzählen, von all denen, die das System benutzen, und was sie damit tun. Diese Ebene wird *Backbone* genannt. Es kann hilfreich sein noch eine Ebene nach oben zu gehen, um Dinge stärker zusammenzufassen und eine bessere Übersichtlichkeit zu schaffen. Die Karten in dieser obersten Ebene werden als *Activities* bezeichnet (vgl. Patton, 2015, Seite 26-27). Siehe Abbildung 2.4 zur Verdeutlichung. Unter den *Activities* befinden sich die *Epics*, welche sehr grobe User Stories sind. Wenn Epics unter Activities genutzt werden, erzählen die Epics die Geschichte der Activity von links nach rechts. Als letzten Detaillierungsgrad kommen darunter die User Stories. Im Gegensatz zu den Epics, sind diese auf eine für die Entwicklung geeignete Größe zugeschnitten worden.

Nachdem die Story in voller Breite festgehalten wurde, müssen die Details ergänzt werden. Die Karten, die in der obersten Reihe liegen, werden zu Kernpunkten, und darunter werden die Details gesammelt. Das Mapping soll dabei helfen, die Lücken in der Story zu entdecken und diese mit Details zu füllen. Es kann jeder einzelne Schritt des Backbones betrachtet werden und mit den folgenden Fragen können die Details ausgearbeitet werden:

- Was genau macht ein User hier?

- Was könnte er stattdessen tun?
- Was würde die User Experience verbessern?
- Was passiert, wenn diese Sache schiefgeht?
- Was wäre, wenn...?

(vgl. Patton, 2015, Seite 15, 28)

2.2.4 Funktionale Anforderungen

Funktionale Anforderungen sind alle in den User Stories festgehaltenen Anforderungen. Sie stehen für eine Funktionalität der Anwendung, die einen Nutzen für einen Nutzer der Anwendung mit sich bringt.

2.2.5 Qualitätsanforderungen

Die Qualität ist eines der Hauptziele guter Softwarearchitektur, da sie den Wert einer Anwendung beeinflusst. Gute Qualität verbessert die Verständlichkeit, Wartbarkeit und Performance eines Systems. Aus diesem Grund sollten Qualitätsanforderungen explizit als Entwurfsziele definiert werden. Allerdings lässt sich Qualität nur indirekt messen und verschiedene Stakeholder haben unterschiedliche Anforderungen an die Qualität. (vgl Starke, 2018, 39–40)

Die folgenden Tabellen 2.1 bis 2.5 zeigen die in dieser Arbeit verwendeten Qualitätsmerkmale mit ihren Teilmerkmalen gemäß DIN/ISO 9126:

Änderbarkeit	Aufwand, der zur Durchführung vorgegebener Änderungen notwendig ist. Änderungen sind dabei Korrekturen, Verbesserungen oder Anpassungen an Änderungen der Umgebung, der Anforderungen und der funktionalen Spezifikationen.
Analysierbarkeit	Aufwand, um Mängel oder Ursachen von Versagen zu diagnostizieren oder um änderungsbedürftige Teile zu bestimmen.
Modifizierbarkeit	Aufwand zur Ausführung von Verbesserungen, zur Fehlerbeseitigung oder Anpassung an Umgebungsänderungen.
Stabilität	Wahrscheinlichkeit des Auftretens unerwarteter Wirkungen von Änderungen.
Testbarkeit	Aufwand, der zur Prüfung der geänderten Software notwendig ist.

Tabelle 2.1: Qualitätsmerkmal Änderbarkeit nach DIN/ISO 9126. Quelle: in Anlehnung an (Starke, 2018, Seite 41–42)

Benutzbarkeit	Aufwand, der zur Benutzung erforderlich ist, und individuelle Beurteilung der Benutzung durch eine festgelegte oder vorausgesetzte Benutzergruppe; hierunter fällt auch der Bereich Softwareergonomie.
Verständlichkeit	Aufwand für den Benutzer, das Konzept und die Anwendung zu verstehen.
Erlernbarkeit	Aufwand für den Benutzer, die Anwendung zu erlernen (z.B. Bedienung, Ein-, Ausgabe).
Bedienbarkeit	Aufwand für den Benutzer, die Anwendung zu bedienen.

Tabelle 2.2: Qualitätsmerkmal Benutzbarkeit nach DIN/ISO 9126. Quelle: in Anlehnung an (Starke, 2018, Seite 41–42)

Effizienz	Verhältnis zwischen dem Leistungsniveau der Software und dem Umfang der eingesetzten Betriebsmittel unter festgelegten Bedingungen.
Zeitverhalten	Antwort- und Verarbeitungszeiten sowie Durchsatz bei der Funktionsausführung.
Verbrauchsverhalten	Anzahl, Menge und Dauer der benötigten Betriebsmittel für die Erfüllung der Funktionen.

Tabelle 2.3: Qualitätsmerkmal Effizienz nach DIN/ISO 9126. Quelle: in Anlehnung an (Starke, 2018, Seite 41–42)

Zuverlässigkeit	Fähigkeit der Software, ihr Leistungsniveau unter festgelegten Bedingungen über einen festgelegten Zeitraum zu bewahren.
Reife	Geringe Versagenshäufigkeit durch Fehlzustände.
Fehlertoleranz	Fähigkeit, ein spezifiziertes Leistungsniveau bei Software-Fehlern oder Nicht-Einhaltung ihrer spezifizierten Schnittstelle zu bewahren.
Wiederherstellbarkeit	Fähigkeit, bei einem Versagen das Leistungsniveau wiederherzustellen und die direkt betroffenen Daten wiederzugewinnen.

Tabelle 2.4: Qualitätsmerkmal Zuverlässigkeit nach DIN/ISO 9126. Quelle: in Anlehnung an (Starke, 2018, Seite 41–42)

Funktionalität	Vorhandensein von Funktionen mit festgelegten Eigenschaften; diese Funktionen erfüllen die definierten Anforderungen.
Angemessenheit	Eignung der Funktionen für spezifizierte Aufgaben, z.B. aufgaben-orientierte Zusammensetzung von Funktionen aus Teilfunktionen.
Richtigkeit	Liefern der richtigen oder vereinbarten Ergebnisse oder Wirkungen, z.B. die benötigte Genauigkeit von berechneten Werten.
Interoperabilität	Fähigkeit, mit vorgegebenen Systemen zusammenzuwirken. Hierunter fällt auch die Einbettung in die Betriebsinfrastruktur.
Ordnungsmäßigkeit	Erfüllung von anwendungsspezifischen Normen, Vereinbarungen, gesetzlichen Bestimmungen und ähnlichen Vorschriften.

Tabelle 2.5: Qualitätsmerkmal Funktionalität nach DIN/ISO 9126. Quelle: in Anlehnung an (Starke, 2018, Seite 41–42)

2.3 Systementwurf

2.3.1 Architektursichten nach Starke

„Eine Sicht zeigt das System aus einer spezifischen Perspektive. Sie abstrahiert von Details, die für diese Perspektive nicht von Bedeutung sind. Sichten erlauben die Konzentration auf bestimmte Details oder bestimmte Aspekte.“ (Starke, 2018, Seite 156) „Die Diagramme oder textlichen Beschreibungen einer Sicht können auch unterschiedliche Abstraktionsebenen oder Detaillierungsstufen beschreiben.“ (Starke, 2018, Seite 156)

Dr. Gernot Starke nennt vier Sichten, um Softwarearchitekturen ausreichend zu beschreiben. Diese werden in Abbildung 2.5 mit ihrer Wechselwirkung dargestellt. Jede Sicht kann Auswirkungen auf die anderen haben. Aus diesem Grund müssen die Sichten parallel ausgearbeitet werden. (vgl. Starke, 2018, Seite 155–158)

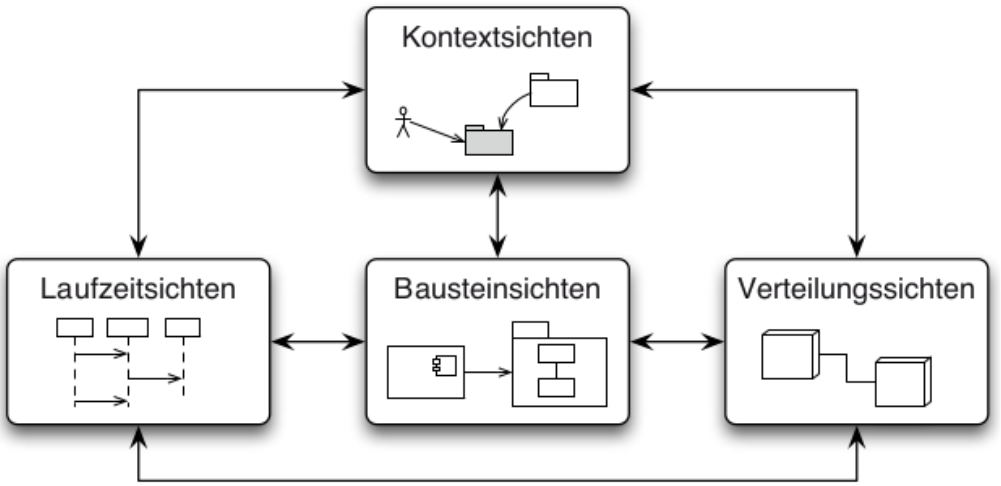

Abbildung 2.5: Die vier Arten von Sichten und ihre Wechselwirkung nach Starke (2018) Seite 157.

Kontextabgrenzung

Die Kontextabgrenzung ist eine Abstraktion über den übrigen Sichten und beschreibt die Einbettung des Systems in seine Umgebung. Dafür wird das System und seine Nachbarsysteme als Blackbox aus der Vogelperspektive dargestellt. Dazu gehören auch die Schnittstellen der Systeme. (vgl. Starke, 2018, Seite 158, 161)

Bausteinsicht

In der Bausteinsicht werden die statischen Strukturen des Systems und deren Schnittstellen beschrieben (vgl. Starke, 2018, Seite 158). Starke (2018) bezeichnet Quellcode in verschiedensten Detaillierungsgraden als Bausteine. Dazu gehören Klassen, Prozeduren, Programme, Pakete, Komponenten oder Subsysteme (Seite 163).

Laufzeitsicht

Die Laufzeitsicht beschreibt dynamische Strukturen, die zur Laufzeit existieren und wie sie zusammenwirken. (vgl. Starke, 2018, Seite 158)

Verteilungssicht (Infrastruktursicht)

Die Verteilungssicht (oder auch als Infrastruktursicht bezeichnet) zeigt die Infrastruktur des Systems aus Betreibersicht. Dokumentiert werden die Hardwarekomponenten, Netztopologien und -protokolle sowie sonstige Bestandteile der physischen Systemumgebung. (vgl. Starke, 2018, Seite 158)

2.3.2 Architekturdokumentation mit arc42

arc42 basiert auf praktischen Erfahrungen mit vielen Systemen in verschiedenen Bereichen und bietet eine Vorlage zur Dokumentation und Kommunikation von Softwarearchitekturen. arc42 gibt Antworten darauf, was und wie dokumentiert und kommuniziert werden sollte, um eine geeignete Architekturdokumentation zu erstellen. (vgl. Starke)

2.4 GitLab

GitLab ist eine Platform, die aus zwei Hauptbestandteilen besteht. Erstens ist GitLab ein sogennantes Repository; eine serverseitige Anwendung, auf der mit der Versionsverwaltung Git die gesamte Änderungshistorie eines Softwareprojektes gespeichert werden kann. Zweitens bietet GitLab die Möglichkeit, CI/CD Pipelines einzurichten und Anwendungen automatisch zu deployen.

2.5 Docker

2.5.1 Docker-Container

Mit Containern oder auch virtuellen Maschinen kann Software von der Umgebung auf der sie läuft isoliert werden. Dadurch wird das Problem gelöst, dass Software meist auf einem Rechner entwickelt und getestet wird, und auf diesem funktionstüchtig ist, aber Probleme auf anderen Rechnern bereitet. Der bekannteste Anbieter einer Container-Runtime ist Docker. Dabei ist die Containertechnologie nichts Neues, wurde durch Docker jedoch durch die einfache Benutzbarkeit populär.

Container werden von Docker wie folgt beschrieben:

> „A **container** is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.“ What is a Container

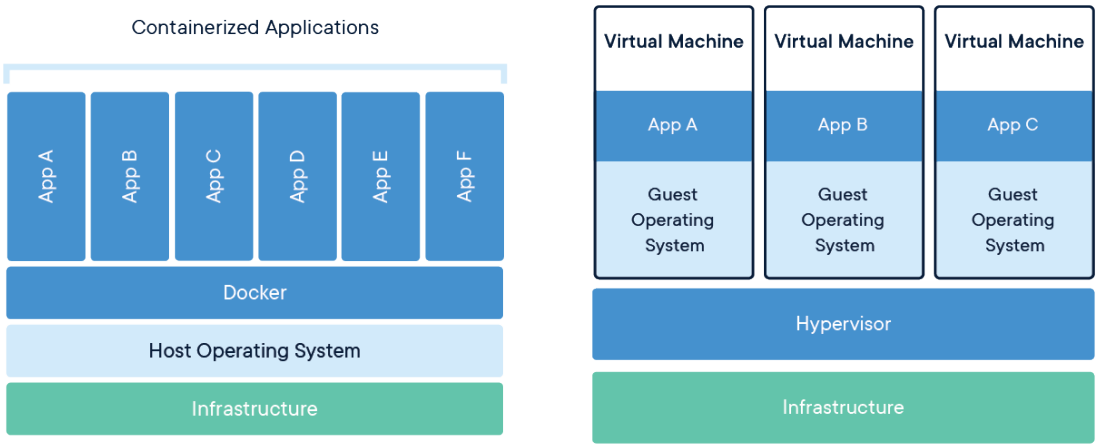

Abbildung 2.6: Vergleich zwischen Containern und Virtuellen Maschinen. (Quelle: What is a Container)

Container sind eine Abstraktion auf der Anwendungsebene, die Code und Abhängigkeiten zusammenfassen. Wie in Abbildung 2.6 zu sehen, können mehrere Container auf demselben Computer ausgeführt werden und den Betriebssystemkern mit anderen Containern gemeinsam nutzen, wobei jeder Container von Docker als isolierte Prozesse ausgeführt werden. Dadurch belegen Container weniger Speicherplatz als Virtuelle Maschinen, die für jede Virtualisierung ein komplettes Betriebssystem benötigen. (vgl. What is a Container)

2.5.2 Docker Image

> „A **Docker container image** is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.“ What is a Container

Container-Images werden zur Laufzeit zu Containern. Im Fall von Docker werden Images zu Containern, wenn sie auf der Docker Engine laufen. (vgl. What is a Container)

2.5.3 Docker Engine

Die Docker Engine ist eine client-server Anwendundung. Dadurch können Docker-Container nicht nur auf dem eigenen Rechner gesteuert werden, sondern auch auf entfernten Servern. Dabei besteht die Docker-Engine aus den folgenden Hauptkomponenten:

- Ein Client in Form eines Kommandlineinterfaces (das `docker` Kommado).
- Ein Server auf dem ein daemon-Prozess läuft.
- Eine REST API für den Server, um dem daemon-Prozess steuern zu können.

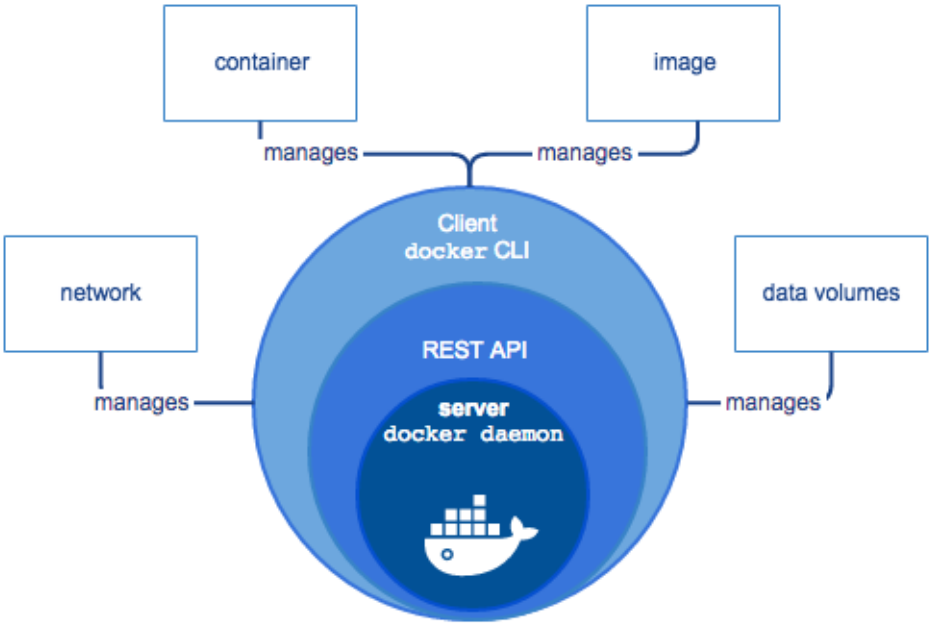

Abbildung 2.7: Aufbau der Docker-Engine.
(Quelle: Docker overview)

(Siehe Docker overview)

2.5.4 Dockerfile

In Docker werden Images durch das Auslesen von Instruktionen aus einer `Dockerfile` gebaut. Diese Textdatei enthält alle Befehle, die zum Erstellen eines bestimmten Images erforderlich sind. (vgl. Dockerfile reference)

Listing 2.1: Beispiel einer Dockerfile

```
1 FROM ubuntu:18.04
2 COPY . /app
3 RUN make /app
4 CMD python /app/app.py
```

Wie im Beispiel 2.1 zu sehen, wird in der `Dockerfile` als erstes ein Basis Image mit dem Schlüsselwort `FROM` angegeben. In diesem Fall ist es das `ubuntu` Image in der Version `18.04`. Möchte man jedoch z.B. nur Bytecode ausführen, kann das leere Image `scratch` verwendet werden. Durch das angeben des Basis-Images ubuntu, wäre das resultierende Image bereits lauffähig. Mit dem Befehl `COPY` wird im Beispiel alles aus dem aktuellen Verzeichnis vom Hostrechner in einen neuen Ordner `/app` im Container kopiert. Durch `RUN` können Kommandozeilenbefehle ausgeführt werden, womit in diesem Fall die Anwendung gebaut wird. `CMD` gibt an, was beim Start des Containers ausgeführt werden soll. Im Beispiel wird die Anwendung `app.py` gestartet. (vgl. Dockerfile reference)

Um einen lauffähigen Container mit einer eigenen Anwendung zum Laufen zu bringen, sind also folgende Schritte notwendig:

1. Das Erstellen einer Dockerfile.
2. Mit dem `docker build` Kommando wird aus dieser Datei ein Docker-Image gebaut.
3. Mit `docker run` lässt sich nun aus dem Image ein Docker-Container starten.

2.6 Kubernetes

Kubernetes ist eine erweiterbare Open-Source-Plattform, die die Verwaltung und Automatisierung von containerisierten Anwendungen erleichtert. Dabei wird deklarative Konfiguration verwendet. Bei deklarativer Konfiguration wird das erwartete Resultat

definiert und nicht die hierfür benötigten Schritte. Es ist mit Kubernetes also möglich, Container in einem Cluster zusammenfassen und dieses auf einfache und effiziente Weise zu verwalten. (vgl. What is Kubernetes)

Google hat das Kubernetes-Projekt 2014 als Open-Source-Projekt zur Verfügung gestellt und baut auf anderthalb Jahrzehnten Erfahrung auf, die Google mit der Ausführung von Produktions-Workloads im großem Maßstab hat. (vgl. What is Kubernetes)

Um den gewünschten Status des Clusters zu beschreiben, können über die Kubernetes-API Objekte übergeben werden. Über diese Objekte lässt sich unter anderem definieren, welche Anwendungen oder Containerimages verwendet werden sollen. Und es lässt sich damit die Anzahl der Replikate und Ressourcen Limits festlegen. Sobald der gewünschte Zustand eingestellt ist, wird Kubernetes dafür sorgen, dass der aktuelle Zustand des Clusters mit dem gewünschten übereinstimmt. Zu diesem Zweck führt Kubernetes verschiedene Aufgaben automatisch aus, z. B. Starten oder Neustarten von Containern oder das Skalieren der Anzahl der Repliken einer bestimmten Anwendung.

2.6.1 Architektur des Kubernetesclusters

Ein Kubernetescluster besteht aus mehreren Nodes. Jeder Node kann eine physische oder virtuelle Maschine sein. Jeder Node enthält die für den Betrieb von Pods notwendigen Dienste und wird von den Master-Komponenten verwaltet. (vgl. Kubernetes Nodes)

- Der **Kubernetes Master** bestehet aus drei Prozessen, die auf einem einzelnen Node in Ihrem Cluster ausgeführt werden, der als Master-Node bezeichnet wird. Diese Prozesse sind:
  - **kube-apiserver** - REST Schnittstelle zum Cluster
  - **kube-controller-manager** - Überwacht den Status des Clusters und sorgt dafür, dass Elemente immer in den gewünschten Zustand zurückkehren (vgl. kube-controller-manager)
  - **kube-scheduler** - Der Scheduler entscheidet, wann und auf welcher Node Pods gestartet werden (vgl. kube-scheduler)
- Jeder einzelne Node im Cluster, welcher nicht der Master ist, führt zwei Prozesse aus:

- **kubelet** - Kommuniziert mit dem Kubernetes Master
- **kube-proxy** - Ein Netzwerk-Proxy, der die Netzwerkdienste von Kubernetes auf jedem Node darstellt.

(vgl. kubernetes concepts)

2.6.2 Ressourcen in Kubernetes

Eine Ressource ist ein Endpunkt in der Kubernetes-API, der eine Sammlung von API-Objekten einer bestimmten Art speichert.
Kubernetes bietet von sich aus eine Vielzahl von Ressourcen. Die Ressourcen, welche für diese Arbeit wichtig sind, werden in den folgenden Abschnitten beschrieben.

Namespaces

Namespaces sind eine Art virtuelle Unterteilung des Kubernetes Clusters, dabei sind Namespaces logisch voneinander getrennt. Namespaces sind für die Verwendung in Umgebungen mit vielen Benutzern vorgesehen, die über mehrere Teams oder Projekte verteilt sind. Namespaces bieten einen Anwendungsbereich für Namen. Ressourcennamen müssen innerhalb eines Namespaces eindeutig sein, jedoch nicht über diese hinweg. (vgl. Namespaces)

Pods

> „Pods are the smallest deployable units of computing that you can create and manage in Kubernetes. Pods

Ein Pod ist eine Gruppe von Containern, mit geteilten Speicher- und Netzwerkressourcen. Inhalte eines Pods werden immer in einem Kontext ausgeführt. Ein Pod modelliert einen anwendungsspezifischen "logischen Host": Er enthält einen oder mehrere Anwendungscontainer, die relativ eng miteinander gekoppelt sind. (vgl. Pods)

Deployment

Deployments stellen eine Gruppe mehrerer identischer Pods ohne eindeutige Identitäten dar. Ein Deployment führt mehrere Replikate einer Anwendung aus und ersetzt automatisch alle Instanzen, die fehlschlagen oder nicht mehr reagieren. Auf diese Weise sorgen Deployments dafür, dass eine oder mehrere Instanzen einer Anwendung [...] verfügbar sind. Deployments werden vom Kubernetes-Deployment-Controller verwaltet. (Kubernetes Deployment)

ConfigMap

Mit ConfigMaps kann die Konfiguration von der Anwendungen entkoppeln werden, um containerisierte Anwendungen portabel zu halten. Daten werden als Schlüssel-Wert Paare gespeichert. ConfigMaps bieten keine Funktionalität zur Geheimhaltung oder Verschlüsselung. (vgl. Config Maps)

Secrets

Mit Kubernetes Secrets können vertrauliche Informationen wie Kennwörter, Token und SSH-Schlüssel gespeichert und verwalten werden. Das Speichern vertraulicher Informationen in einem Secret ist sicherer und flexibler als das Speichern von Klartext in einer Pod-Definition oder in einem Container-Image. (vgl. Secrets)

Job

Ein Job erstellt einen oder mehrere Pods und wird diese solange wiederholt ausführen, bis eine spezifizierte Anzahl Erfolgreichen terminieren. Jobs verfolgen die erfolgreiche Beendigung und starten Pods bei nicht erfolgreicher Ausführung erneut. Dass Löschen eines Job, löscht auch die Pods die dieser erstellt hat. (vlt. Kubernetes Jobs)

Service

Eine abstrakte Möglichkeit, eine Anwendung, die als eine Reihe von Pods läuft, über das Netzwerk als Dienst bereitzustellen. Kubernetes gibt den Pods ihre eigenen IP-Adressen und einen eindeutigen DNS-Namen für einen Satz von Pods und kann eine Lastverteilung über diese Pods durchführen. (vgl. Kubernetes Services)

Ingress

Eine API Ressource, die externen Zugriff zu einem Service in Kubernetes verwaltet; typischerweise über HTTP oder HTTPS. Genau wie bei Services kann auch hier eine Lastverteilung durchgeführt werden. (vgl. Kubernetes Ingress)

Role Based Access Controll (RBAC)

Role Based Access Control (RBAC) ist eine Methode zur Regulierung von Zugriff auf Ressourcen in Kubernetes. Dabei werden Berechtigungen in Rollen zusammengefasst, welche dann an Nutzer vergeben werde können. (vgl. RBAC)

Roles und ClusterRoles

Roles und ClusterRoles enthalten Regeln, die einen Satz von Berechtigungen darstellen. Die Berechtigungen sind rein additiv; es existieren keine verbietende Berechtigungen. Eine Role legt immer Berechtigungen innerhalb eines bestimmten Namespace fest. Eine ClusterRole hingegen ist nicht an einen Namespace gebunden und ist im ganzen Cluster gültig. (vgl. RBAC)

Service Account

Service Accounts sind für Prozesse vorgesehen, die in Pods ausgeführt werden (vgl. Service Accounts). Um dem Service Account Berechtigungen zuzuweisen, muss dieser an eine Role oder ClusterRole gebunden werden.

2.6.3 Benutzerdefinierte Ressourcen in Kubernetes

Benutzerdefinierte Ressourcen (Custom Resources) sind Erweiterungen der Kubernetes-API. Custom Resources können im laufenden Cluster hinzugefügt und entfernt werden. Sobald eine Custom Resource durch eine Custom Resource Definition (CRD) definiert wurde, können Nutzer auf diese genauso über die API zugreifen, wie auf alle anderen schon bestehenden Ressourcen im Cluster. (vgl. custom resources)

Custom Controller

Controller prüfen in Kubernetes kontinuierlich, durch Ressourcen definierte, gewünschte Zustände und setzten diese um. Durch das Hinzufügen von Custom Controllern, ist es möglich, dass eigene Custom Ressourcen interpretiert und dann umgesetzt werden. (vgl. custom resources)

2.7 Golang

Golang (Go) ist eine Programmiersprache von Google und wurde als Lösungsansatz zu bestehenden Problemen in der Software Infrastruktur entworfen. Die heutige Umgebung, in denen Anwendungen ausgeführt werden, hat fast nichts mit der Umgebung zu tun, in der die verwendeten Sprachen, hauptsächlich C++, Java und Python, erstellt wurden. Die Probleme, die durch Multicore-Prozessoren, vernetzte Systeme, massive Rechencluster und das Webprogrammiermodell verursacht wurden, wurden umgangen und nicht direkt angegangen. Darüber hinaus hat sich die Skalierung geändert: Die heutigen Serverprogramme umfassen zig Millionen Codezeilen, werden von Hunderten oder sogar Tausenden von Programmierern bearbeitet und täglich aktualisiert. Um die Sache noch schlimmer zu machen, haben sich die Erstellungszeiten selbst in großen Kompilierungsclustern auf viele Minuten oder sogar Stunden ausgedehnt.

Go wurde entwickelt, um die Arbeit in dieser Umgebung produktiver zu gestalten. Neben den bekannteren Aspekten wie der integrierten Parallelität und der Speicherbereinigung umfassen die Entwurfsüberlegungen von Go ein strenges Abhängigkeitsmanagement, die Anpassungsfähigkeit der Softwarearchitektur mit dem Wachstum der Systeme und die Robustheit über die Grenzen zwischen Komponenten hinweg. (vgl. Rob Pike)

2.7.1 Structs in Golang

In Go gibt es keinen Typen der als Objekt bezeichnet wird, aber es gibt einen Typen, der der gleichen Definition einer Datenstruktur entspricht, die sowohl Code als auch Verhalten vereint. Dieser wird in Go als „Struct“ bezeichnet. (vgl. Steve Francia, Objects In Go)

Go wurde absichtlich ohne jegliche Vererbung entwickelt. Dies bedeutet nicht, dass Objekte (Strukturwerte) keine Beziehungen haben. Stattdessen haben die Go-Autoren beschlossen, einen alternativen Mechanismus zu verwenden, um Beziehungen zu kennzeichnen. Anstelle der Vererbung folgt Go strikt dem Prinzip „composition over inheritance“. Go erreicht dies sowohl durch Subtypisierung (is-a) als auch durch Objektzusammensetzung (has-a) zwischen Strukturen und Interfaces.
Der Mechanismus, mit dem Go das Prinzip der Objektzusammensetzung implementiert, wird als eingebettete Typen bezeichnet. In Listing 5.5 ist ein Beispiel für ein Struct in Go, bei dem eine Person mit einer Adresse definiert wird. (vgl. Steve Francia)

Listing 2.2: Beispiel für ein Struct in Golang (Angeleht an Steve Francia)

```
1  type Person struct {
2      Name string
3      Address Address
4  }
5
6  type Address struct {
7      Number string
8      Street string
9      City   string
10     State  string
11     Zip    string
12 }
```

3 Anforderungsanalyse

Um die Vollständigkeit der spezifizierten Anforderungen an das System zu gewährleisten, wurden in diesem Kapitel Stakeholder analysiert und aus diesen Anforderungen an das System abgeleitet. Zudem wurden Anforderungen an die Qualität des Systems erfasst. Um Anforderungen aus Sicht des Nutzers erfassen zu können, wurde das Gesamtsystem betrachtet und nicht nur der in dieser Arbeit zu entwickelnde Teil. Also von der Aktion des Nutzers in GitLab bis hin zu der Multi-Tenancy in Kubernetes. Die Unterteilung des Clusters für die Nutzer wird des Weiteren als „Arbeitsbereich“ bezeichnet, damit keine Implementierungsdetails in der Anforderungsanalyse festgelegt werden.

3.1 Stakeholder

In diesem Abschnitt wird erläutert, wie mit dem System interagierende Stakeholder analysiert und daraus Nutzerrollen abgeleitet wurden. Dies erfolgt in den in 2.2.1 beschriebenen Schritten: Brainstorming, Organisieren, Konsolidieren und Verfeinern.

3.1.1 Brainstorming

Durch Brainstorming ergaben sich folgende Nutzerrollen:

- Student
- wissenschaftlicher Mitarbeiter
- Professor
- GitLab Nutzer
- GitLab Gruppenleiter
- GitLab Gruppenmitglied
- Administrator
- Projektleiter

- Gruppenleiter
- DevOps
- Support
- Entwickler

Die Nutzerrollen kommen aus verschiedenen Anwendungsbereichen. *Student*, *wissenschaftlicher Mitarbeiter* und *Professor* sind die wohl wahrscheinlichsten Nutzer, von denen die Anwendung in der internen Cloud der Hochschule genutzt wird. Falls die Anwendung jedoch mal in einem Unternehmen betrieben werden sollte, wurden hier auch Nutzerrollen aus diesem Kontext berücksichtigt. Dazu gehören unter anderem *Projektleiter*, *Gruppenleiter* und *Support*. Die Nutzerrollen *Administrator*, *DevOps* und *Entwickler* passen sowohl zum Unternehmenskontext als auch zur Hochschule.
Als *Entwickler* wird hier eine Nutzerrolle beschrieben, die den Tenant-Integrator weiter entwickelt und nicht irgendeine andere Anwendung die in Kubernetes deployed werden soll. Da die zu entwickelnde Anwendung Nutzer aus GitLab abbilden soll, werden auch Nutzerrollen aus GitLab aufgeführt.

3.1.2 Organisieren

Nach dem Sammeln der Nutzerrollen, wurden die Nutzerrollen so organisieren, dass deren Beziehungen untereinander verdeutlicht werden. Umso ähnlicher die Nutzerrollen, desto stärker die Überlappung der Karten.

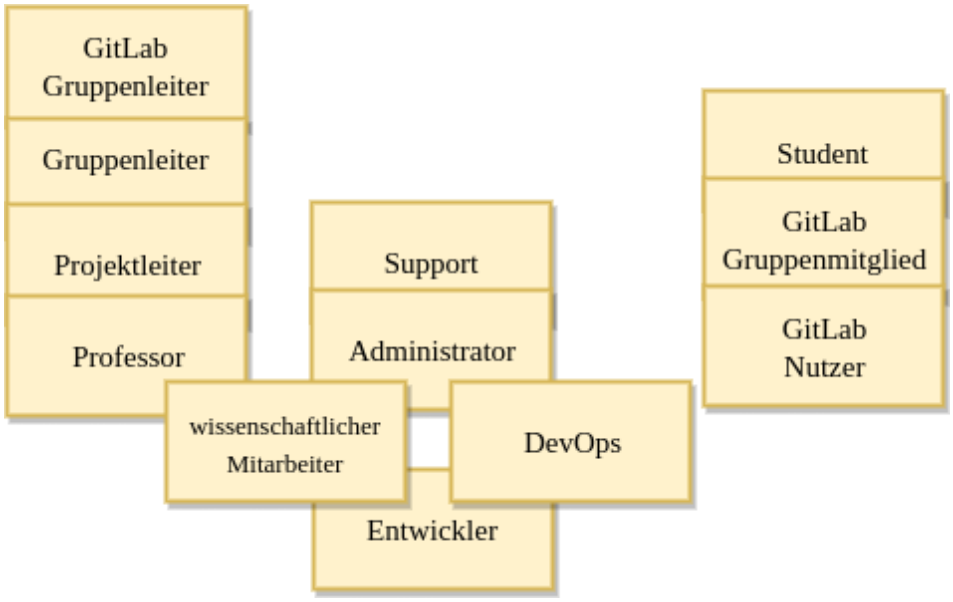

Abbildung 3.1: Organisation der Nutzerrollen.

Wie in Abbildung 3.1 zu sehen, wurden *Student*, *GitLab Gruppenmitglied* und *GitLab Nutzer* zusammengefasst, da sie alle nur eigene Arbeitsbereiche in Kubernetes für ihre Projekte benötigen.
(GitLab) Gruppenleiter, *Projektleiter* und *Professor* bringen zusätzlich die Anforderung mit, dass sie alle geteilte Arbeitsbereiche verwalten möchten, also den Zugriff von anderen Nutzern auf ihre Projekte verwalten.
Im Gegensatz zu den vorherigen Nutzerrollen, sind der *Support* und der *Administrator* eher nicht daran interessiert, eigene Arbeitsbereiche zu erhalten, sondern haben die Aufgabe, die Arbeitsbereiche anderer Nutzer zu verwalten.
Der *Entwickler* bringt verschiedene Qualitätsanforderungen an die Anwendung mit, da er die Anwendung möglichst ohne umständliche Anpassungen mit neuen Funktionalitäten erweitern möchte.
DevOps sind hier eine Mischung aus Administrator und Entwickler, da es wahrscheinlich ist, dass sie kleinere Anpassungen an der Anwendung tätigen müssen, z.B. Änderungen am Deployment, aber auch Tätigkeiten eines Administrators übernehmen.
Der *wissenschatliche Mitarbeiter* nimmt hier eine Sonderrolle ein und kann eigentlich zu allen Gruppen gehören, da er genauso gut an eigenen Arbeitsbereichen in Kubernetes interessiert sein kann, als auch an der Verwaltung von anderen Arbeitsbereichen, oder der Weiterentwicklung der Anwendung.

3.1.3 Konsolidieren

In diesem Abschnitt wurden konkrete Nutzerrollen aus den gruppierten Karten ausgearbeitet.

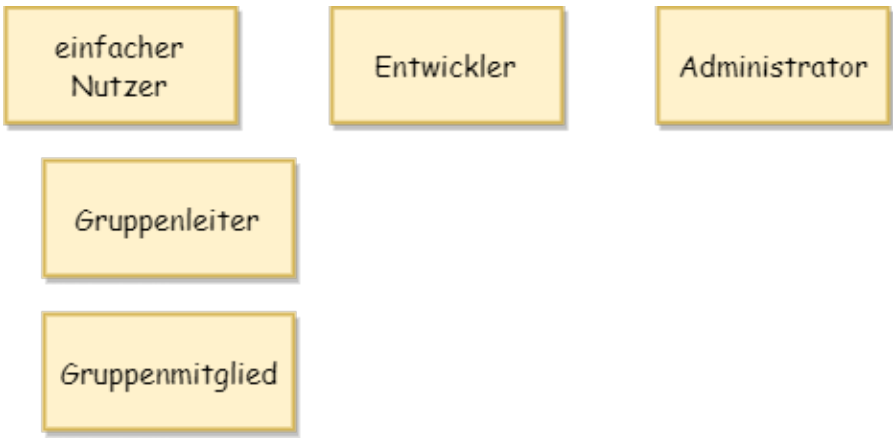

Abbildung 3.2: Konsolidieren der Nutzerrollen.

Wie in Abbildung 3.2 zu sehen, wurden *Support*, *Administrator* und *DevOps* aus der Organisationsphase als *Administrator* zusammengefasst. Der Grund dafür ist, dass diese drei Nutzerrollen alle die Anforderung mitbringen Arbeitsbereiche zu verwalten. Nur bei *DevOps* überschnitten sich die Anforderungen mit denen des Entwicklers. Da sich allerdings dazu entschieden wurde den *Entwickler* zu behalten, und so die zusätzliche Anforderung der DevOps bereits von diesem abgedeckt werden, konnte die DevOps-Karte entfernt werden.
Nun bleibt noch der Stapel mit den leitenden Rollen (*Gruppenleiter* usw.) und der Stapel mit dem *Student*, welche nur ihren eigenen Arbeitsbereich für ihre Projekte benötigen. Da beide Gruppen eigene Arbeitsbereiche benötigen, wurde diese Anforderung als *einfacher Nutzer* zusammengefasst. Die zusätzliche Anforderung der leitenden Rollen kann als eine Spezialisierung des einfachen Nutzers umgesetzt werden und wurde hier als *Gruppenleiter* festgehalten. Genauso wurde auch die zusätzliche Anforderung mit einer Gruppe in einem Arbeitsbereich zusammenzuarbeiten als Spezialisierung *Gruppenmitglied* unter dem einfachen Nutzer festgehalten.

3.1.4 Verfeinern

Nachdem ein Verständnis für die Zusammenhänge der Nutzerrollen geschaffen wurde, wurden in diesem Abschnitt alle Eigenschaften zusammengefasst, die eine Nutzerrolle von anderen unterscheidet.

Einfacher Nutzer Möchte sein(e) Projekt(e) in seinem Arbeitsbereich betreiben und nichts von anderen Projekten sehen oder irgendwelche Seiteneffekte durch andere Nutzer des Clusters haben.

- **Gruppenmitglied** Möchte zusätzlich mit anderen Gruppenmitgliedern in einem Arbeitsbereich arbeiten.
- **Gruppenleiter** Verwaltet zusätzlich seine Gruppenmitglieder und deren Arbeitsbereiche und Ressourcen.

Entwickler Möchte den Tenant-Integrator an neue Informationsquellen anschließen können, die Anwendung mit möglichst geringem Aufwand erweitern und Fehler schnell identifizieren können.

Administrator Verwaltet das gesamte Cluster, mit den sich darin befindenden Arbeitsbereichen und deren Nutzern. Darüber hinaus verwalten sie die dazugehörigen Hardware-Ressourcen, die den Nutzern zur Verfügung stehen.

3.2 Funktionale Anforderungen

Aus den Nutzerrollen wurden funktionalen Anforderungen abgeleitet. Diese werden in diesem Kapitel behandelt. Dazu wurden aus den Nutzerrollen zuerst klar erkennbare Anforderungen in Form von Stories notiert.

3.2.1 User Stories

Zu Anfang wurde eine Grundmenge von Stories für den *Backbone* benötigt. Die Lücken in der Geschichte wurden danach, wie im nächsten Abschnitt 3.2.2 beschrieben, durch das Story Mapping ausgearbeitet. Wie in den Grundlagen 2.2.2 dargestellt, werden zu den einzelnen Karten Details beschrieben und diese als Akzeptanztests festgehalten. Es wird darauf verzichtet, Akzeptanztests für Stories zu definieren, die nicht in dieser Arbeit umgesetzt werden. Welche Stories alle in dieser Arbeit umgesetzt werden, wird im ersten Release der User Story Map 3.3 dargestellt.

3.2.2 User Story Mapping

Im Folgenden, wurden die gesammelten User Stories auf eine User Story Map übertragen, um eventuelle Lücken in der Story zu identifizieren. Dadurch hat sich der grundlegende *Backbone* gebildet. In Abbildung 3.3 durch die grauen Karten dargestellt. Es wurde eine weitere Abstraktionsebene geschaffen, indem die *Epics* mit *Activities* unterteilt wurden.

An einigen Stellen wurde bewusst von der Vorlage zum Schreiben der User Stories abgewichen, da diese nicht immer einen Mehrwert mit sich bringt. Unter anderem wurde der Zweck des Ziels weggelassen, wenn sich dieser bereits mit Sicherheit aus dem Ziel der Rolle schließen lässt.

Nach Ergänzung der Lücken in der Story Map, hat sich die Story vervollständigt, womit es möglich war, die Story in Releases zu unterteilen. Dabei kommen alle Stories, die Teil dieser Arbeit sind, in den ersten Release.
In Abbildung 3.3 wurden Stories, die nach der Bildung des *Backbones* dazu gekommen sind, grün hervorgehoben.

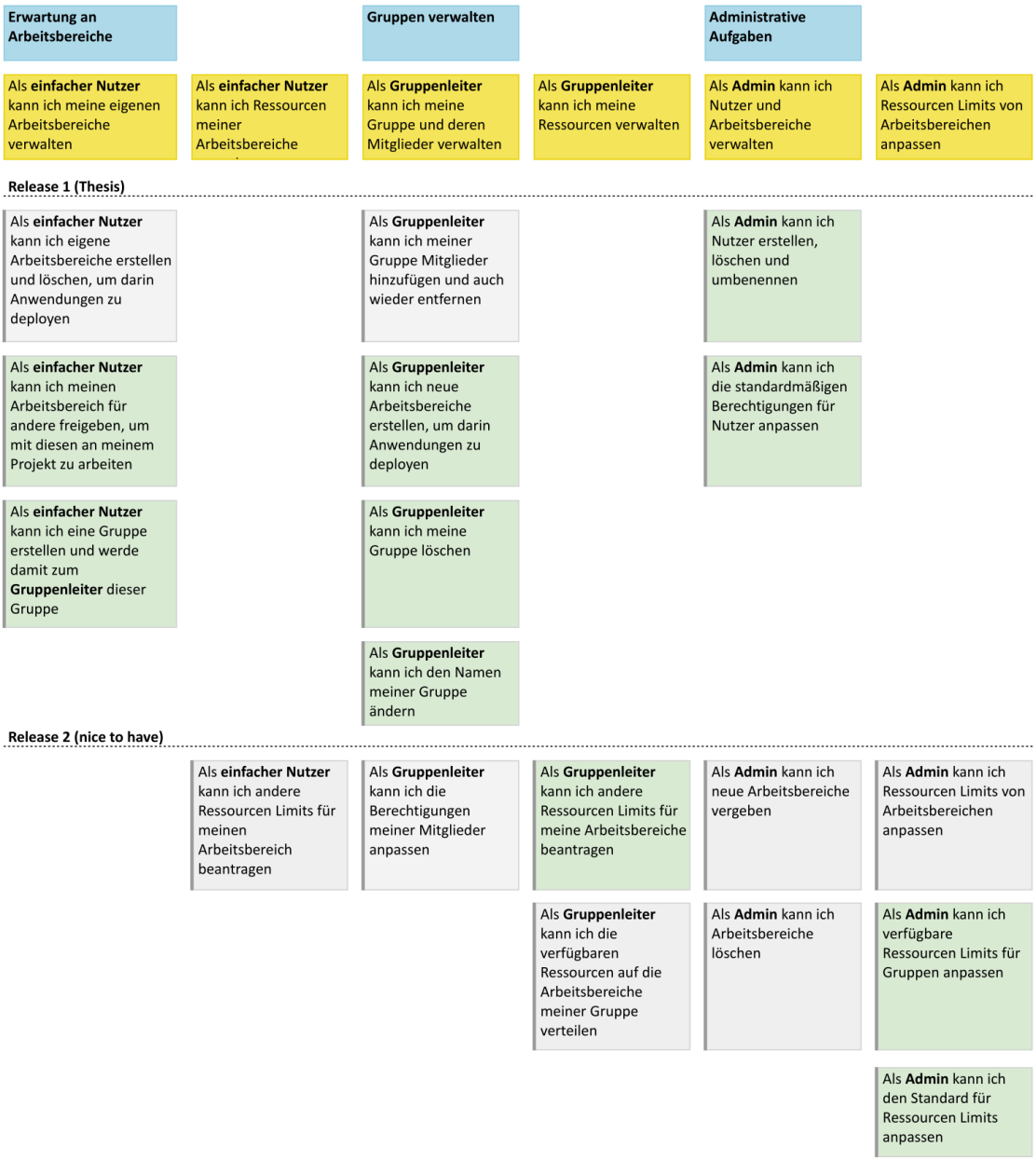

Abbildung 3.3: Vervollständigte User Story Map.

Rund um den Arbeitsbereich des **einfachen Nutzers** wurden folgende Stories gesammelt:

1. *Als **einfacher Nutzer** kann ich eigene Arbeitsbereiche erstellen, umbenennen und löschen, um darin Anwendungen zu deployen.*

Im Kontext der HAW bedeutet dies, dass jeder Student automatisch Arbeitsbereiche in Kubernetes erhalten soll, sobald er in GitLab Projekte erstellt. Die in GitLab angelegten Anwendungen können dann in diesem Arbeitsbereich deployed werden.

Akzeptanztests:

- Nach dem Erstellen eines Projektes in GitLab steht ein Arbeitsbereich in Kubernetes für dieses Projekt zur Verfügung.
- Nach dem Löschen eines Projektes wird auch der Arbeitsbereich gelöscht.
- Nur der Nutzer, der das Projekt erstellt hat, hat auch Zugriff auf den zugehörigen Arbeitsbereich.
- Der Arbeitsbereich ist nur für den Nutzer sichtbar, dem das zugehörige Projekt gehört.

2. *Als **einfacher Nutzer** kann ich andere Ressourcen Limits für meinen Arbeitsbereich beantragen.*

   Da den vergebenen Arbeitsbereichen nicht unendlich Ressourcen zur Verfügung stehen können, ist es notwendig, dass es Nutzern ermöglicht wird, höhere Limits zu beantragen.

   Diese Story könnte durch einen einfachen Kommunikationskanal wie Slack o.ä. abgedeckt werden, an deren Ende ein Administrator sitzt, der befähigt ist, an den Ressourcen der Tenants Änderungen vorzunehmen.

3. *Als **einfacher Nutzer** kann ich meinen Arbeitsbereich für andere freigeben, um mit diesen an meinem Projekt zu arbeiten.*

   Nutzern sollte es möglich sein, mit anderen in einem Arbeitsbereich zu arbeiten, indem sie Nutzer zu ihrem Projekt hinzufügen. Genauso sollten sie diesen Zugriff wieder entziehen können.

   Akzeptanztests:

   - Nutzer, die dem Projekt hinzugefügt wurden, erhalten Zugriff auf den zugehörigen Arbeitsbereich in Kubernetes.
   - Abhängig von den Berechtigungen im Projekt wird dem Nutzer beim Hinzufügen

die korrekte Rolle in Kubernetes zugewiesen.
- Beim Entfernen des Nutzers aus dem GitLab Projekt wird auch der Zugriff auf den Arbeitsbereich wieder entzogen.

4. *Als **einfacher Nutzer** kann ich eine Gruppe erstellen und werde damit zum **Gruppenleiter** dieser Gruppe*

   Eine Gruppe erhält in Kubernetes einen Arbeitsbereich für alle Mitglieder dieser Gruppe.

   Akzeptanztests:
   - Tenant für die Gruppe wurde in Kubernetes erstellt.
   - Abhängig von der Konfiguration wird für die Gruppe ein Arbeitsbereich für Testzwecke ohne zugehöriges Projekt erstellt.

Rund um die Gruppenverwaltung durch den **Gruppenleiter** wurden die folgenden Stories festgehalten:

1. *Als **Gruppenleiter** kann ich meiner Gruppe Mitglieder hinzufügen und auch wieder entfernen.*

   Es soll möglich sein, dass mehrere Nutzer zusammen in einem Arbeitsbereich arbeiten. Für in GitLab erstellte Gruppen wird ein Arbeitsbereich bereitgestellt, der für alle Mitglieder dieser Gruppe sichtbar ist und auf den sie Zugriff haben.
   In GitLab gibt es zu normalen Gruppen auch Untergruppen. Bei diesen werden Berechtigungen von übergeordneten Gruppen auf die unteren vererbt. Auch diese Erwartungen der Benutzer wird in den Akzeptanztests festgehalten.

   Akzeptanztests:
   - Wenn einem Nutzer Zugriff zu der Gruppe in GitLab gewährt wurde, bekommt dieser auch Zugriff auf die Arbeitsbereiche der Gruppe in Kubernetes. Besitzt die Gruppe weitere Untergruppen, bekommt der Nutzer auch Zugriff auf die Arbeitsbereiche der untergeordneten Gruppen.
   - Beim Entfernen des Nutzers von der Gruppe wird auch der Zugriff auf Arbeitsbereiche der Gruppe und ihrer untergeordneten Gruppen wieder entzogen.
   - Abhängig von den Berechtigungen im Projekt und der Konfiguration des Tenant Integrators wird dem Nutzer beim Hinzufügen die korrekte Rolle in Ku-

bernetes zugewiesen.

2. *Als **Gruppenleiter** kann ich neue Arbeitsbereiche erstellen, um darin Anwendungen zu deployen.*

   Beim Erstellen eines Projektes in einer Gruppe soll allen Mitgliedern ein geteilter Arbeitsbereich in Kubernetes zur Verfügung stehen.

   Akzeptanztests:
   - Alle Gruppenmitglieder haben Zugriff auf den Arbeitsbereich des geteilten Projektes.
   - Wird das Projekt in einer Gruppe mit übergeordneten Gruppen erstellt, müssen auch die Gruppenmitglieder der übergeordneten Gruppen Zugriff auf den erstellten Arbeitsbereich erhalten.
   - Abhängig von den Berechtigungen der Gruppenmitglieder und der Konfiguration des Tenant Integrators wird den Gruppenmitgliedern die korrekte Rolle in Kubernetes, für den Arbeitsbereich, zugewiesen.

3. *Als **Gruppenleiter** kann ich meine Gruppe löschen.*

   Akzeptanztests:
   - Tenant der Gruppe ist auch in Kubernetes nicht mehr vorhanden nachdem die Gruppe gelöscht wurde.
   - Alle zu der Gruppe gehörenden Arbeitsbereiche in Kubernetes sind gelöscht.
   - Tenants von übergeordneten Gruppen existieren noch.
   - Auch für Tenants von übergeordneten Gruppen wurden die Arbeitsbereiche der soeben gelöschten Untergruppe entfernt.

4. *Als **Gruppenleiter** kann ich den Namen meiner Gruppe ändern.*

   Akzeptanztests:
   - Bezeichnung des Tenants wird in Kubernetes geändert.
   - Alle Arbeitsbereiche der Gruppe bleiben mit denselben Zugriffsberechtigungen erhalten.

Zu den Aufgaben eines Administrators wurden folgende Stories identifiziert:

1. *Als **Administrator** kann ich neue Arbeitsbereiche vergeben.*

Es sollte Administratoren möglich sein, zusätzlich zu den automatisch erstellen Arbeitsbereichen des Integrators, Arbeitsbereiche für Nutzer bereit zu stellen.

2. *Als **Administrator** kann ich Arbeitsbereiche löschen.*

   Es sollte Administratoren möglich sein, zusätzlich erstellte Arbeitsbereiche wieder zu löschen.

3. *Als **Administrator** kann ich die Ressourcen Limits von Arbeitsbereichen anpassen.*

   Da es Nutzern möglich sein soll, neue Ressourcen Limits zu beantragen, ist es notwendig, dass Administratoren diese anpassen können.

4. *Als **Administrator** kann ich Nutzer erstellen, löschen und umbenennen.*

   Akzeptanztests:
   - Beim Erstellen eines Nutzers wird für diesen ein Tenant in Kubernetes angelegt.
   - Je nach Konfiguration, steht dem erstellen Nutzer ein Arbeitsbereich in Kubernetes zur Verfügung.
   - Beim Löschen eines Nutzers wird auch der Tenant in Kubernetes gelöscht.
   - Beim Umbenennen eines Nutzers wird auch der Tenant und alle seine Arbeitsbereiche in Kubernetes umbenannt.

Da sich aus den Qualitätsanforderungen aus Kapitel 3.3 ergab, dass die Anforderungen des Entwicklers durch die Szenarien 3.2 und 3.6 bereits abgedeckt werden, wird der Entwickler nicht weiter als Nutzerrolle aufgeführt werden. Der Vollständigkeit halber werden die identifizierten Stories hier noch einmal aufgeführt:

1. *Als **Entwickler** ist es mir möglich den Integrator lokal zu debuggen.*

   Um die Weiterentwicklung des Integrators möglichst einfach zu gestalten, soll ein lokales Debugging und Testen möglich sein.

   Akzeptanztests:
   - Anwendung lässt sich lokal im Debug Modus starten.

2. *Als **Entwickler** ist es mir möglich, den Integrator lokal zu testen.*

Der Integrator soll über eine Testsuite verfügen, die lokal ausführbar ist.

Akzeptanztests:

- Anwendung verfügt über eine Testsuite.
- Testsuite lässt sich lokal ausführen.

3.3 Qualitätsanforderungen

In diesem Abschnitt werden alle Qualitätsanforderungen des zu entwickelnden Systems festgehalten, die die Architekturentscheidungen im nächsten Kapitel 4 Systementwurf maßgeblich beeinflussen können.
Die Qualitätsanforderungen werden tabellarisch aufgeführt. Dabei wird für jede Qualitätsanforderung ein Geschäftsziel definiert, durch ein oder mehrere Beispiele gestützt, und die entsprechende Anforderung an das zu entwickelnde System festgehalten.

3.3.1 Änderbarkeit

Geschäftsziel(e):	Ermögliche die Konfiguration des Systems in Bezug auf das Standardverhalten.
Beispiel:	Es ist nicht mehr gewünscht, dass jeder Nutzer automatisch ein Arbeitsbereich für Testzwecke erhält bevor dieser überhaupt einem Projekt angehört.
	Gruppenleiter aus GitLab sollen mehr Berechtigungen in den Arbeitsbereichen der Gruppe erhalten als die Mitglieder der Gruppe.
Anforderung:	Administratoren sind in der Lage das Standardverhalten des Systems zu konfigurieren.

Tabelle 3.1: **Szenario**: Das Standardverhalten des Systems soll geändert werden.

Geschäftsziel(e):	Ermögliche Entwicklern das isolierte Testen und Entwickeln.
Beispiel:	Entwickler möchte die Anwendung ohne Seiteneffekte durch Anwender oder andere Entwickler weiterentwickeln und seine Änderungen testen.
Anforderung:	Anwendung und Tests der Anwendung müssen lokal lauffähig sein.

Tabelle 3.2: **Szenario**: Entwickler möchten an dem System möglichst komfortabel und ohne Seiteneffekte Änderungen vornehmen können.

Geschäftsziel(e):	Ermögliche als Informationsquelle für die Multi-Tenancy ein gänzlich anderes System zu nutzen.
Beispiel:	Administratoren könnten einen Client erhalten, mit dem sie die Multi-Tenancy des Clusters komfortabel über die Konsole verwalten könnten.
	Es könnte sich dazu entschieden werden von GitLab auf eine andere Informationsquelle zu wechseln.
Anforderung:	Entwickler sind in der Lage, auf Basis vordefinierter Schnittstellen das Grundgerüst des Systems zu erweitern.

Tabelle 3.3: **Szenario**: Entwickler können neue Informationsquellen anschließen.

Geschäftsziel(e):	Entwickler sollen einen einfach Einstieg in das System haben und damit einfacher Änderungen vornehmen können (verbessert auch die gefühlte Benutzbarkeit für Entwickler).
Bespiel:	Neuer Mitarbeiter soll Änderung am System vornehmen.
Anforderung:	Durch ausreichende Dokumentation in der README, soll es Entwicklern unmittelbar möglich sein, das System zu starten, deployen, testen und Änderungen vorzunehmen.

Tabelle 3.4: **Szenario**: Developer Experience.

3.3.2 Benutzbarkeit

Geschäftsziel(e):	Verbessere die Benutzbarkeit (und gefühlte Zuverlässigkeit). Ermögliche Benutzern, zu Fehlern führende Kombinationen von Eingabedaten zu korrigieren, ohne dass das System abstürzt.
Beispiel:	Das System gibt eine Fehlermeldung zurück, wenn der Name des Tenant Sonderzeichen enthält.
Anforderung:	Das System nimmt korrekte Eingabedaten an, weist fehlerhafte Eingabedaten zurück und gibt diese im Log aus.

Tabelle 3.5: **Szenario**: Validierung von Benutzereingaben.

3.3.3 Effizienz

Geschäftsziel(e):	Risikoarme Änderungen und Erweiterungen.
Beispiel:	Entwickler führt eine Änderung am Quellcode durch und startet anschließend die automatisierte Testsuite.
Reaktion:	Das Testframework führt sämtliche Testfälle aus und berichtet die Testergebnisse an den Benutzer.
Anforderung:	Sämtliche Testfälle sind in weniger als 3 Minuten ausgeführt

Tabelle 3.6: **Szenario**: Ausführung von Tests in weniger als 3 Minuten.

Geschäftsziel(e):	Keine Neuversuche durch zu lange Antwortzeiten.
Beispiel:	Durch Timeouts und daraus resultierende Neuversuche in der Informationsquelle werde Operationen möglicherweise mehrfach ausgeführt, weil die Informationsquelle nicht wissen kann das die erste Operation noch ausgeführt wird.
Reaktion:	Das System verarbeitet und antwortet schnell genug, um keine Neuversuche auszulösen.
Anforderung:	Verarbeitung und Antwort einer Anfrage sollen in unter einer Sekunde erfolgen.

Tabelle 3.7: **Szenario**: Verarbeitung sollen innerhalb einer Sekunde erfolgen.

3.3.4 Zuverlässigkeit

Geschäftsziel(e):	Fähigkeit, Fehler in der Verarbeitung schnell zu identifizieren, lokalisieren und zu beheben.
Beispiel:	Verarbeitung einer Anfrage schlägt fehl.
Reaktion:	System sammelt die für Fehlerdiagnose und -behebung relevanten Informationen (Art des Fehlers, betroffene Datensätze, Zeit usw.).
Anforderung:	Relevante Informationen werden nach Auftreten des Fehlers im Log ausgegeben.

Tabelle 3.8: **Szenario**: Fehler in der Verarbeitung schnell identifizieren.

3.3.5 Funktionalität

Geschäftsziel(e):	System ist in- und außerhalb von Kubernetes lauffähig.
Beispiel:	Anwender möchte den Integrator zuerst außerhalb von Kubernetes betreiben und dann langsam auf Kubernetes umsteigen und diesen dort betreiben.
Anforderung:	System ist in- und außerhalb von Kubernetes lauffähig.

Tabelle 3.9: **Szenario**: Der Integrator lässt sich in- und außerhalb von Kubernetes betreiben.

4 Systementwurf

Im folgenden Kapitel wird ein Konzept des Systems entworfen und auf die wichtigsten Entwurfsentscheidungen eingegangen. Dieses Kapitel orientiert sich an arc42, was in Kapitel 2.3.2 vorgestellt wurde.

4.1 Kontextabgrenzung

Der Tenant Integrator synchronisiert alle relevanten Zustandsänderungen, die ein Nutzer vornimmt, aus GitLab. Über die Kubernetes API werden Tenants als Ressourcen in Kubernetes erstellt, gelöscht und aktualisiert.
Der Tenant Manager soll als Controller in Kubernetes laufen und setzt die Multi-Tenancy in Kubernetes um, indem die von der Tenant-Ressource vorgegebenen Zustände umgesetzt werden. In der aktuellen Version „v1alpha1“ erzeugt der Tenant Manager, für die vom Tenant Integrator erstellen Tenant Ressourcen, Namespaces in Kubernetes. Dafür wird von dem Tenant Manager eine CRD vorgegeben.

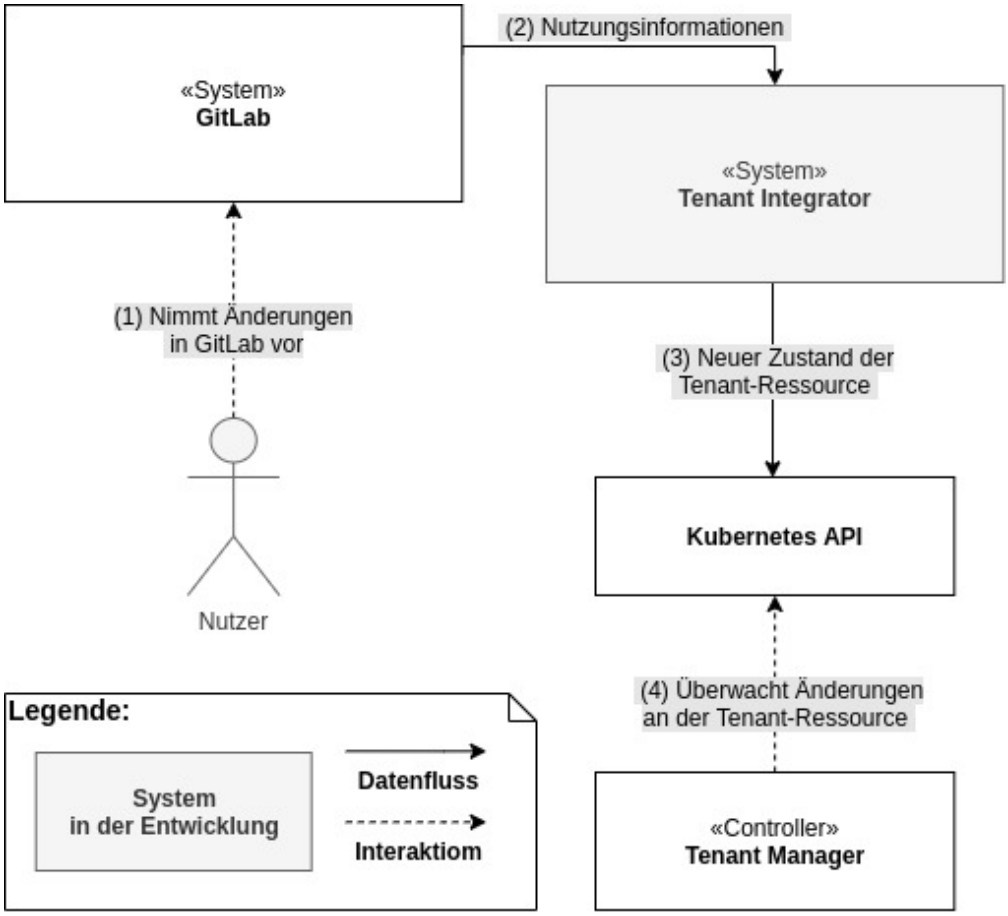

Abbildung 4.1: Kontextabgrenzung zu den Nachbarsystemen.

4.2 Fachlicher Kontext

Um Zustandsänderungen in Kubernetes nachvollziehen zu können, sind in der folgenden Tabelle 4.1 alle relevanten Zustandsänderungen von GitLab mit dem geplanten Resultat in Kubernetes aufgeführt:

Aktion in GitLab	Aktion des Integrators
Nutzer Events	
Nutzer erstellt	Tenant erstellen
Nutzer gelöscht	Tenant löschen
Nutzer umbenannt	Tenant umbenennen
Gruppen Events	
Gruppe erstellt	Tenant erstellen
Gruppe gelöscht	Tenant löschen
Gruppe umbenannt	Tenant umbenennen
Untergruppen Events	
Untergruppe erstellt	Tenant erstellen und alle Nutzer, die Zugriff auf den Tenant der übergeordneten Gruppe haben, übernehmen
Untergruppe gelöscht	Tenant löschen
Untergruppe umbenannt	Tenant umbenennen
Projekt Events	
Nutzer/Gruppe hat Projekt erstellt	Dem Tenant Arbeitsbereich hinzufügen
Nutzer/Gruppe hat Projekt gelöscht	Arbeitsbereich vom Tenant löschen
Nutzer/Gruppe hat Projekt umbenannt	Arbeitsbereich des Tenants umbenennen
Nutzer/Gruppe hat Projekt verschoben	Arbeitsbereich vom Tenant entfernen und neuen erstellen
Projektmitglieder Events	
Nutzer zu Team hinzugefügt	Tenant Zugriff auf Arbeitsbereich geben
Nutzer aus Team entfernt	Tenant Zugriff auf Arbeitsbereich entziehen
Gruppenmitglieder Events	
Nutzer zur Gruppe hinzugefügt	Nutzer dem Tenant der Gruppe hinzufügen
Nutzer aus Gruppe entfernt	Nutzer aus dem Tenant der Gruppe löschen

Tabelle 4.1: Relevante Zustandsänderungen von GitLab und das geplante Resultat an der Tenant Ressource in Kubernetes

4.3 Technischer Kontext

Im Folgenden wird der Datenaustausch mit den Nachbarsystemen aus technischer Sicht beschrieben. Dazu gehören, die Art der Daten die ausgetauscht werden, die verwendeten Schnittstellen, und Protokolle.

4.3.1 Technischer Kontext zu GitLab

Die Synchronisation des Zustandes von GitLab wird von dem Tenant Integrator durch einen Webhook realisiert werden. Ein Webhook ist ein HTTP Endpunkt für sofortige Ereignisbenachrichtigung von einem anderem System. In GitLab lässt sich dafür ein Endpunkt registrieren, an den ausgewählte Ereignisse per HTTP geschickt werden. Dieser wird so konfiguriert, dass alle Ereignisbenachrichtigungen an den dafür vorgesehen Endpunkt des Tenant Integrators geschickt werden.

In der Dokumentation von GitLab (System hooks) lässt sich zu jeder Aktionen, die im fachlichen Kontext 4.2 beschrieben wurde, ein Ereignis finden. Da über den Webhook nicht immer der gesamte aktuelle Zustand sondern nur Zustandsänderungen empfangen werden, ist es notwendig, beim Start des Tenant Integrators den gesamten aktuellen Zustand von GitLab zu synchronisieren. Dies wird über die REST-API von GitLab abgewickelt werden. Diesen Teil des Tenant-Integrators wird des Weiteren als Fetcher bezeichnet. Dieser Fetcher wird beim Start alle relevanten Information über die GitLab-API auslesen. Danach wird der Webhook nur noch auf Zustandsänderungen reagieren.

4.3.2 Technischer Kontext zum Tenant Manager

Der Tenant Integrator interagiert nicht direkt mit dem Tenant Manager sondern legt nur den Soll-Zustand als Ressource in der REST-API von Kubernetes fest. Dieser soll dann von dem Tenant Manager umgesetzt werden.

Der Tenant Integrator nutzt ein Kubernetes Projekt namens „code-generator“ (code generator). Es wird genutzt, um den Kubernetes-Client zu erweitern. Durch den Code-Generator kann aus dem Tenant-CRD Quellcode generieren werden, der benötigt wird, um mit der Tenant-Ressource der Kubernetes-API arbeiten zu können.

In dem Listing 4.1 ist ein Beispiel für eine Tenant-Ressource in Kubernetes. Diese wird, anhand der in den Kommentaren festgelegten Nummerierung, in der Liste darunter erläutert:

Listing 4.1: Tenant Ressorce Beispiel

```yaml
apiVersion: tenants.cbrgm.net/v1alpha1 # (1)
kind: Tenant # (2)
metadata:
  labels:
    controller-tools.k8s.io: "1.0" # (3)
  name: tenantName # (4)
spec:
  projects: # (5)
  - name: projectName
    namespace: namespaceName
    groups: # (6)
    - name: userWithMasterRole
      members: # (6.1)
      - name: userName
      roleRef: # (6.2)
        kind: ClusterRole
        name: tenant-master-role
        apiGroup: rbac.authorization.k8s.io
```

1. Version der Tenant Ressource
2. Art der Ressource, die durch ein CRD definiert wurde.
3. Version des *controller-tools* mit dem die Ressource generiert wurde.
4. Der Name des Tenant wird dem Nutzer oder der Gruppe von GitLab entnommen.
5. Liste von Projekten eines Tenant mit zugehörigen Namespace.
6. Nach Berechtigungen gruppierte Liste von Nutzern, die Zugriffsrechte auf den Namespace des Tenants erhalten.
   6.1. Mitglieder der Gruppe
   6.2. Rolle mit den Zugriffsrechten in Kubernetes. Diese Rolle wird allen Nutzern der Gruppe zugewiesen und gilt für alle Namespaces des Tenants.

Benennung in GitLab und Kubernetes

Die Einschränkungen für die Benennung einer Ressource in Kubernetes weichen nur gering von den Einschränkungen in GitLab ab. Es gibt jedoch Unterschiede, die dazu führen, dass nicht alle Namen aus GitLab in Kubernetes als Tenant-Ressource gespeichert werden können.

1. **Erlaubte Zeichen**
   In GitLab ist es bei Bezeichnungen erlaubt, zusätzlich zu den alphabetischen Zeichen und Zahlen noch einige Sonderzeichen zu verwenden. Zu den zulässigen Sonderzeichen gehören: Minus, Unterstrich und Punkt. In Kubernetes ist es nur das Minus. Dazu kommt, dass Sonderzeichen nicht am Anfang oder am Ende eines Namens stehen dürfen.
2. **Groß und Kleinschreibung**
   In GitLab können die alphabetischen Zeichen eines Nutzer- oder Gruppennames groß oder klein geschrieben sein, allerdings darf derselbe Name nicht doppelt vorkommen und dabei wird nicht die Groß- oder Kleinschreibung beachtet. Dies bedeutet, dass alle großen Zeichen aus GitLab in Kubernetes klein dargestellt werden können, ohne dass zwei Nutzer oder Gruppen auf denselben Tenant abgebildet werden.
3. **Anzahl der Zeichen**
   In Kubernetes darf der Name einer Ressource nicht mehr als 253 Zeichen betragen. In GitLab sind es 255 Zeichen. Was dazu führt, dass Nutzer und Gruppen mit Namen zwischen 254 und 255 Zeichen zwar in GitLab erstellt, aber nicht in Kubernetes als Tenant-Ressource gespeichert werden können.

4.4 Lösungsstrategie

Für die Implementierung des Tenant-Integrators wurde die Sprache Go gewählt. Go ist eine Open Source-Programmiersprache von Google, mit der auch Kubernetes entwickelt wurde. Wie in den Qualitätsmerkmalen beschrieben, sollte die Antwortzeit von Anfragen durch GitLab nicht zulange brauchen, da GitLab sonst dieselbe Anfrage erneut schickt und eine erneute Verarbeitung starten würde. Dem kommt Go durch seine performante Leistung zugute. Go läuft direkt auf der darunter liegenden Hardware und wird nicht

wie zum Beispiel Java auf einer virtuellen Maschine ausgeführt, was bedeutet, dass Programmcode direkt zu einer Binärdatei kompiliert wird und damit schneller ist.

Für die bessere Erweiterbarkeit wird der Tenant-Integrator in zwei Module aufgeteilt werden. Dadurch wird sich der generische Teil, der nicht auf GitLab zugeschnitten ist, für jegliche Informationsquellen wiederverwenden lassen. Der generische Teil wird weiter die Bezeichnung Tenant Integrator tragen, um den Baustein Tenant-Integrator und die gleichnamige Bezeichnung des Gesamtsystems auseinanderhalten zu können, wird der technische Baustein immer als Tenant-Integrator-Baustein oder -Modul hervorgehoben werden. Das Modul, welches GitLab und den Tenant-Integrator verbindet, wird des Weiteren als GitLab-Tenant-Integrator bezeichnet. Eine Übersicht bietet das Diagramm 4.2 im nächsten Kapitel.

Wie schon im technischen Kontext zu GitLab 4.3.1 beschrieben, wird der GitLab-Tenant-Integrator zur Laufzeit einmal Daten über den Fetcher von der GitLab-API laden und danach über den Webhook auf Zustandsänderungen reagieren. Der Webhook wird als einfache Server Anwendung entwickelt werden und sowohl in als auch außerhalb von Kubernetes lauffähig sein. Zudem soll darauf geachtet werden, dass die Weiterentwicklung des Systems ohne viel Aufwand möglich ist.
Der Fetcher wird als einfache Client-Anwendung umgesetzt werden und beim Deployment als Job bereitgestellt werden. Der Fetcher wird dadurch nur einmal gestartet, und sobald alles synchronisiert wurde, wieder beendet.

In 4.3.2 wurde erläutert inwieweit Bezeichnungen aus GitLab in Tenant-Ressourcen umgesetzt werden können. Da die Einschränkungen von GitLab und Kubernetes nur gering voneinander abweichen und ein Fehlerfall eher selten vorkommen wird, wird darauf verzichtet eine Lösung für das Abbilden aller Bezeichnungen zu finden. Um zu verhindern dass durch ungültige Bezeichnungen weitere Fehler entstehen, sei es in dem System oder durch darauf folgendes Fehlverhalten von Benutzern, sollen ungültige Bezeichnungen abgefangen werden und der Fehlerfall im Log sowie in der Antwort der Anfrage erläutert werden. Damit wird das Problem auf die Qualitätsanforderung 3.5 zurückgeführt.

4.5 Bausteinsicht

Die Multi-Tenancy soll in Kubernetes, wie in Abbildung 4.2 zu sehen, durch drei Module realisiert werden.

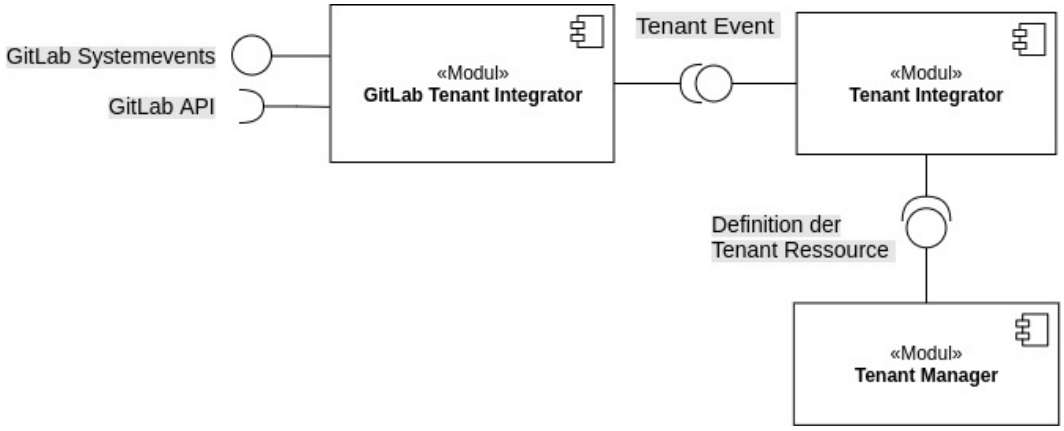

Abbildung 4.2: Komponentendiagramm: Übersicht über die Module des Gesamtsystems.

Tenant Manager Setzt die Multi-Tenancy in Kubernetes um. Zudem gibt diese Komponente vor welche Informationen über einen Tenant benötigt werden. Der Tenant-Manager wird in Kubernetes als Controller laufen und Tenant Ressourcen durch ein CRD vorgeben.

Tenant Integrator Bietet eine Schnittstelle an und erstellt, löscht und aktualisiert die Tenant Ressourcen über die Kubernetes-API.

GitLab Tenant Integrator Nutzt die Schnittstelle des generischen Tenant Integrator Bausteins und bildet über diesen Zustände von GitLab als Tenant Ressourcen in Kubernetes ab. Benötigt wird eine Schnittstelle zur GitLab-API, um den gesamten Zustand synchronisieren zu können und es wird eine Schnittstelle für laufende Zustandsänderungen als Webhook angeboten.

4.5.1 Paketstruktur

In Abbildung 4.3 wird die Aufteilung der Pakete und deren Struktur durch deren Zugriff auf andere Pakete dargestellt. Grundlegend werden dabei initiale Objekte beim Start der Anwendung im `cmd`-Paket initialisiert. Je nachdem, ob die Anwendung als *Webhook* oder *Fetcher* gestartet wird, werden dabei unterschiedliche Objekte initialisiert.

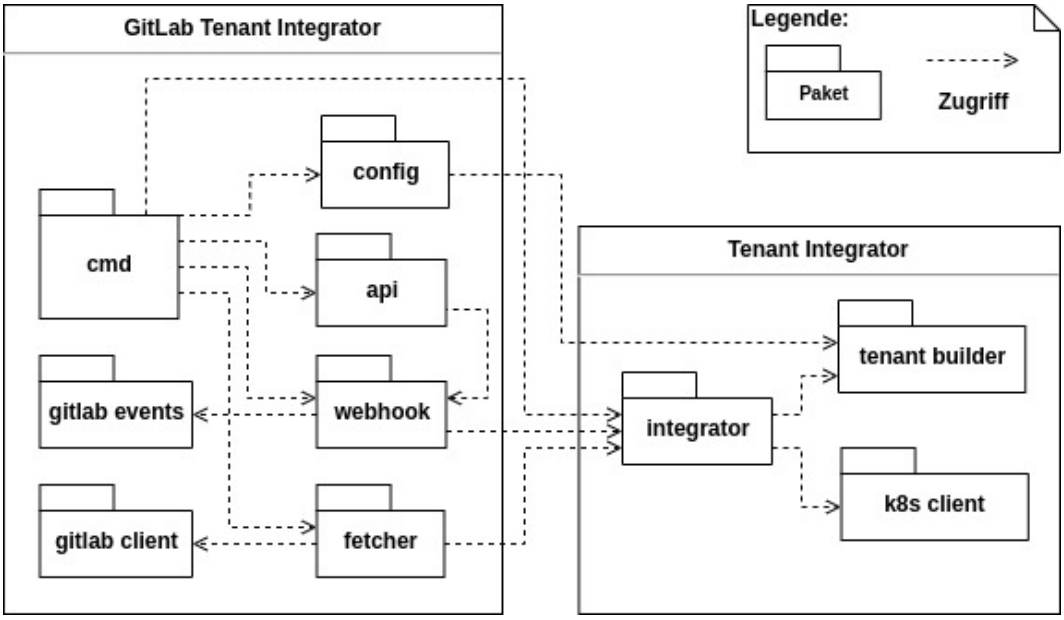

Abbildung 4.3: Übersicht über die Pakete des zu entwickelnden Systems.

In den folgenden Tabellen wird auf die Pakete aus Abbildung 4.3 eingegangen und deren Zweck und Verantwortlichkeiten erläutert. Begonnen wird von rechts nach links, um möglichst mit den Paketen zu beginnen, die auf keine Anderen zugreifen. Referenzen auf andere Pakete werden fett hervorgehoben.

Pakete des Tenant Integrator Moduls

k8s client	Client für die Kubernetes-API. Je nachdem, ob das System in oder außerhalb von Kubernetes läuft, muss es anders initialisiert werden. Für eine Verwendung außerhalb des Clusters wird eine Datei zur Authentifizierung benötigt.
tenant builder	Enthält Methoden mit denen die Tenant-Objekte vom **integrator** erstellt werden. Bei der Initialisierung des **tenant builders** muss das Standardverhalten definiert werden. Dazu gehören die Berechtigungsgruppen, in die sich die Tenants aufteilen lassen sollen, und die Namespaces, die jeder Tenant erhalten soll.
integrator	Ist das Herz des ganzen Systems. Über eine Schnittstelle werden Operation zu Tenants entgegengenommen, im **tenant builder** daraus Objekte erstellt, und über den **k8s client** als Tenant-Ressourcen in Kubernetes gespeichert.

Pakete des GitLab Tenant Integrator Moduls

gitlab client	Client für die GitLab API.
fetcher	Läd mit dem **gitlab client** den aktuellen Zustand aus GitLab und nutzt die API des **integrator**s um diesen in Tenant-Operationen abzubilden.
gitlab events	Enthält alle Events von GitLab, die von dem **Webhook** akzeptiert werden.
webhook	Nimmt HTTP-Anfragen entgegen und bildet diese auf die korrekten Methoden des **integrator**s ab, wenn es sich um korrekte Events von GitLab handelt.
api	REST Schnittstelle mit einem Endpunkt für den Webhook und einem für die health-checks von k8s.
config	Enthält alle Information über die Konfiguration des gesamten Systems. Dazu gehört auch die Konfiguration es **tenant builders**.
cmd	Commandlineinterface über das beim Start alle Komponenten initialisiert werden, die für das Grundverhalten des Systems benötigt werden.

4.6 Laufzeitsicht

Um darzustellen, wie Zustandsänderungen aus GitLab als Tenant abgebildet und in Kubernetes als Ressource gespeichert werden, wurde ein Zustandsdiagramm erstellt. Die Abbildungen soll dabei helfen, dass alle Informationen aus GitLab korrekt verarbeitet werden. Zudem lassen sich daraus Testfälle ableiten. Die Diagramme werden in der Realisierung in Kapitel 5.6 verwendet um Spezialfälle darzustellen.

Im Gegensatz zu einem einfachen Zustandsautomaten lassen sich für die Tenant-Ressource durch zum Beispiel das Hinzufügen von Gruppenmitgliedern unendlich viele Zustände definieren. Da durch wiederkehrende Aktionen jedoch keine neuen Informationen gewonnen werden, wurden diese im Diagramm weggelassen. Zudem werden die Zustände im YAML-Format dargestellt. Dies hilft dabei, die Zustände und ihre Eigenschaften und Einschränkungen besser zu verstehen. Berechtigungsgruppen wurden im Diagramm nicht dargestellt, da sie die Zustände nur um eine weitere Dimension verkomplizieren, aber keinen Mehrwert mit sich bringen.

4.7 Infrastruktursicht

Da über den Webhook nicht der gesamte Zustand, sondern nur Zustandsänderungen empfangen werden, ist es notwendig, den Tenant-Integrator in zwei Komponenten aufzuteilen. Beim Start des Integrators wird daher ein Job ausgeführt werden, der die aktuellen Information aus GitLab synchronisiert. Dieser Job wird des Weiteren als **Tenant-Fetcher** bezeichnen.

5 Realisierung

In diesem Kapitel wird die Umsetzung des GitLab Tenant Integrators und des Tenant Integrators erläutert, welche in Kapitel 4 konzipiert wurden. Dabei wird nur auf die wichtigsten Teile des realisierten Endproduktes eingegangen werden.

5.1 Generierung des Kubernetes Clients

Wie im ersten Absatz unter 4.3.2 beschrieben, nutzt der Tenant Manager das code generator Projekt, um den Kubernetes Client zu erweitern. Dieses Projekt basiert auf dem gengo Projekt, mit dem Quellcode durch Go-Structs generiert werden können. Für die Generierung eines Clients wurde, wie in Listing 5.2 zu sehen, der Tenant als Struct in Go beschrieben und mit Anweisungen für den Generator versehen.

Listing 5.1: Globale Angaben für den Generator aus dem Tenant Manager. (`docs.go` Zeile 17-23. Unwichtige Angaben wurden dabei beim Beispiel ausgelassen)

```
1 // +k8s:deepcopy-gen=package
2 // +k8s:defaulter-gen=TypeMeta
3 // +groupName=tenants.cbrgm.net
4 package v1alpha1
```

Listing 5.2: Das Tenant Struct aus dem Tenant Manager. (`tenant_types.go` Zeile 59-73)

```
1 // +genclient
2 // +genclient:nonNamespaced
3 // +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.
    Object
4
5 // Tenant is the schema for the tenant API
```

```
6  // +kubebuilder:printcolumn:name="Namespaces",type="integer",
       JSONPath=".status.Namespaces",description="The number of
       namespaces owned by the tenant"
7  // +kubebuilder:printcolumn:name="Age",type="date",JSONPath=".
       metadata.creationTimestamp"
8  type Tenant struct {
9   metav1.TypeMeta   `json:",inline"`
10  metav1.ObjectMeta `json:"metadata,omitempty"`
11
12  Spec   TenantSpec   `json:"spec,omitempty"`
13  Status TenantStatus `json:"status,omitempty"`
14 }
```

Im Folgenden wird auf die Listings 5.1 und 5.2 eingegangen und es werden alle Kommentarzeilen, die mit einem Plus starten, erläutert, da diese Zeilen Befehle zur Code-Generierung für das Tenant Struct darunter angeben.

`+k8s:deepcopy-gen` Generiert für jeden Typen standardmäßig die `DeepCopy`-Methode

`+groupName` Bestimmt den vollständigen Namen der API Ressource.

`+genclient` Markiert das Objekt für den Generator und gibt an, dass ein Client für die Kubernetes API generiert werden soll. Dabei werden die Standard Verb-Funktionen des Client generiert (`create`, `update`, `delete`, `get`, `list`, `update`, `patch`, `watch` und abhängig davon, ob das `.Status` Feld im Struct existiert wird der client mit `updateStatus` generiert)

`+genclient:nonNamespaced` Alle Verb-Funktionen können ohne die Angabe eines Namespaces genutzt werden. Die Ressource ist also nicht an einen Namespace gebunden und steht im ganzen Cluster zur Verfügung.

`+kubebuilder:printcolumn:name` in Zeile 7 und 8 geben an, welche Felder mit `kubetcl get` angezeigt werden sollen.

`+k8s:deepcopy-gen:interface` Bestimmt den Rückgabetypen der `DeepCopy`-Methode. Notwendig, da alle API Typen dieses Interface teilen.

Zur finalen Generierung des Clients wird das in Listing 5.3 dargestellt Script verwendet:

Listing 5.3: Script für die Generierung des Clients. (`update-codegen.sh`)

```
1 #!/usr/bin/env bash
2
```

```
3  set -o errexit
4  set -o nounset
5  set -o pipefail
6
7  # workaround for go modules
8  go mod vendor
9  chmod u+x ./vendor/k8s.io/code-generator/generate-groups.sh
10
11 SCRIPT_ROOT=$(dirname "${BASH_SOURCE[0]}")/..
12 CODEGEN_PKG=${CODEGEN_PKG:-$(cd "${SCRIPT_ROOT}"; \
13   ls -d -1 ./vendor/k8s.io/code-generator 2>/dev/null || \
14   echo ${GOPATH}/src/k8s.io/code-generator)}
15
16 "${CODEGEN_PKG}"/generate-groups.sh "deepcopy,client" \
17   github.com/kubetenancy/tenant-integrator/pkg/generated \
18   github.com/cbrgm/tenant-manager/pkg/apis \
19   tenants:v1alpha1 \
20   --output-base "$(dirname "${BASH_SOURCE[0]}")/../../../.." \
21   --go-header-file "${SCRIPT_ROOT}"/hack/boilerplate.go.txt
```

Im Listing 5.3 werden den Zeilen 11 und 12 zuerst Hilfsvariablen deklariert. `SCRIPT_ROOT` bestimmt den aktuellen Pfad aus dem das Skript aufgerufen werden soll. Dies entspricht dem Hauptverzeichnis des Projektes. Die `update-condegen.sh` Datei liegt dabei im Ordner `hack`.

`CODEGEN_PKG` bestimmt den Ort des Code-Generators. Da für Go Module kein bestimmter Ort für Abhängigkeiten festgelegt ist, ist es notwendig die Abhängigkeiten über den `go mod vendor` Befehl in Zeile 8 in den `vendor` Ordner zu speichern. Zudem müssen, wie in Zeile 9 zu sehen, die Berechtigungen für den aktuellen Nutzer so angepasst werden, dass er das Skript unter `./vendor/k8s.io/code-generator/generate-groups.sh` auch ausführen kann. In Zeile 16 wird das Skript zur Generierung des Clients dann ausgeführt. Über den ersten Parameter `"deepcopy,client"` wird dabei angegeben, was genau generiert werden sollen. Der zweite Parameter in Zeile 17 gibt an, wo die generierten Dateien landen sollen und der dritte in Zeile 18 gibt an, wo sich die Angaben an den Generator befinden. Die schon in 5.1 und 5.2 gezeigt wurden.

5.2 Konfiguration

Wie in 3.1 beschrieben, soll das Verhalten des GitLab Tenant Integrators konfigurierbar sein, um initiale Berechtigungsgruppen und Namespaces in den Tenant Ressourcen anpassen zu können. In Listing 5.4 ist ein Beispiel für eine Konfigurationsdatei. Dabei sind grundlegend drei Hauptkategorien zu konfigurieren. Die Konfigurationen für jeden anzulegenden Tenant sind in der Datei `properties.yaml` unter `integrator` zu finden. Die anderen beiden Kategorien sind jeweils spezifische Konfigurationen für den `webhook` und den `fetcher`.
Für eine genauere Erläuterung werden nun Kernpunkte im Listing nummeriert und es wird darunter darauf eingegangen:

Listing 5.4: Beispiel einer Konfiguration Datei (properties.yaml) des GitLab Tenant Integrators. (`root`-Verzeichnis)

```yaml
integrator:
  defaultNamespaces: # (1)
    - "default"
  roleGroups: # (2)
  - name: "master"
    roleRef: "tenant-master-role"
  - name: "member"
    roleRef: "tenant-member-role"
webhook:
  token: "mySecretToken" # (3)
  port: "8080" # (4)
gitlab:
  api:
    url: "http://localhost/api/v4" # (5)
    token: "RYukddr3-V5fPyXZHJp3" # (6)
log.level: "Debug"
```

1. Suffix für den Namespace, den jeder Nutzer initial erhält. Dieser Namespace oder Namespaces, wenn mehrere Suffixe angegeben werden, stehen dem Nutzer also schon zu Verfügung bevor er überhaupt einem Projekt in GitLab angehört. In diesem Fall würde also ein Namespace mit dem Suffix `default` für jeden Nutzer erstellt werden und für Testzwecke zur Verfügung stehen.

2. Über `roleGroups` wird festgelegt, welche Berechtigungsgruppen in den Tenant Ressourcen auswählbar sind. Wie im Beispiel dargestellt, könnte ein Gruppenleiter in GitLab der Berechtigungsgruppe `master` in der entsprechenden Tenant Ressource hinzugefügt werden und dadurch der `tenant-master-role` in k8s zugewiesen werden.
3. Token der von GitLab unter dem HTTP Header `X-Gitlab-Token` mitgeschickt werden muss.
4. Port auf dem der Webhook Events empfängt.
5. URL zur GitLab API.
6. Token der vom GitLab Tenant Integrator zur GitLab API unter dem HTTP Header `Private-Token` mitgesendet werden muss.

Die Konfigurationsdatei `properties.yaml` wird für das lokale Ausführen der Anwendung verwendet. Damit können Entwickler die Anwendung starten, ohne erst alle Umgebungsvariablen definieren zu müssen. Zudem wird damit auch die Qualitätsanforderung 3.2 erfüllt, bei dem es Entwicklern ermöglicht werden soll, das System möglichst komfortabel und von anderen isoliert zu testen. Im Cluster ist es eher üblich die Konfiguration mit Umgebungsvariablen zu übergeben. Dies entkoppelt Konfiguration von der Anwendung.

Das Laden der Konfiguration über die Umgebungsvariablen sowie der Konfigurationsdatei wurde mit der Bibliothek viper umgesetzt. Die geladene Konfiguration wird dabei von der Bibliothek in ein `Struct` geladen. Im Listing 5.5 ist die vollständige erste Ebene der Konfiguration in Go sowie Beispielhaft eine zweite Ebene dargestellt. Über die Kennzeichnung `mapstructure` werden für jedes Feld des Structs die passenden Bezeichnungen für die Konfiguration angegeben.

Listing 5.5: Zwei Ebenen des Structs für die Konfiguration (config-Paket)

```
1 type properties struct {
2  Gitlab      gitlabProperties      `mapstructure:"gitlab"`
3  Kubernetes  kubernetesProperties `mapstructure:"kubernetes"`
4  Webhook     webhookProperties     `mapstructure:"webhook"`
5  Integrator  integratorProperties `mapstructure:"integrator"`
6  Log         logProperties         `mapstructure:"log"`
7 }
8
9 type webhookProperties struct {
```

```
10  Port   string `mapstructure:"port"`
11  Token string `mapstructure:"token"`
12 }
```

Damit viper alle Umgebungsvariablen lädt, müssen diese erst registriert werden. Dies ist bei viper über eine Konfiguration, das Setzten von Standardwerten oder über Flags in der Kommandozeile möglich, jedoch nicht über ein Struct.
Aus diesem Grund wurde eine Hilfsmethode entworfen, um die benötigten Variablen bei viper zu registrieren. Diese ist in Listing 5.6 dargestellt.

Listing 5.6: Registrierung der benötigten Umgebungsvariablen. (`config`-Paket)

```
1  // keys have to be set or env-vars doesn't get loaded
2  func workaroundForEnvVars() {
3   var keys []string
4   extractTags(reflect.TypeOf(properties{}), []string{}, &keys)
5   for _, key := range keys {
6    viper.SetDefault(key, "")
7   }
8  }
9
10 func extractTags(from reflect.Type, pre []string, result *[]
       string) {
11  for i := 0; i < from.NumField(); i++ {
12   pre := pre
13   field := from.Field(i)
14   tag := field.Tag.Get("mapstructure")
15
16   if tag != "" {
17    pre = append(pre, ".", tag)
18    // if field is struct go into recursion
19    if field.Type.Kind() == reflect.Struct {
20     extractTags(field.Type, pre, result)
21    } else { // if not add to result
22     newResult := strings.Join(pre[1:], "")
23     *result = append(*result, newResult)
24    }
```

```
25    }
26   }
27 }
```

Im Folgenden wir das Registrieren von Umgebungsvariablen aus Listing 5.6 erläutert. Dabei wird anhand der im Listing dargestellten Zeilennummern referenziert werden. Diese Zeilennummern entsprechen nicht den Zeilennummern aus dem originalen Quellcode.

Der grundlegende Ansatz ist, dass alle Felder des `properties`-Structs aus Listing 5.5 rekursive ausgelesen werden. Beim rekursiven Aufruf ist jeder Knoten ein Struct, über dessen Felder iteriert werden. Die Werte von `mapstructure` werden von Knoten zu Knoten weitergegeben, bis ein Feld gefunden wird, welches kein Struct ist. Sobald dieses Feld gefunden wurde, werden die gesammelten Werte von `mapstructure` bis zu diesem Feld konkateniert und zurückgegeben. Die zurückgegebenen Werte werden dann bei viper bekannt gemacht, indem diese mit leeren Standardwerten registriert werden.

In Zeile 10 ist die Definition der Methode `extractTags` zu sehen, welche in Zeile 4 aufgerufen wird. Die Methode nimmt unter anderem einen Typen über den Parameter `from` entgegen, bei dem die Werte von `mapstructure` ausgelesen werden sollen. Dazu kommt der Parameter `pre` über den die ausgelesenen Werte von `mapstructure` rekursive weitergeben werden. In Zeile 17 wird in `pre` der Wert von `mapstructure` des aktuellen Feldes an die Werte der Structs, die zu diesem geführt haben, angehängt. Zwischen jedes angehängte Zeichen kommt ein Punkt als Trennzeichen. Der Parameter `pre` ist ein Array von Strings, da die Werte erst zum Ende konkatenieren werden.
In dem letztem Parameter `result` werden alle konkatenierten Werte gesammelt. Dieser Wert ist ein Pointer, damit er in allen Wurzeln auf den gleichen Speicherbereich referenziert. Da in Zeile 17 auch der zuerst gefundene Wert mit einem Punkt davor gespeichert wird, wird dieser in Zeile 22 beim Konkatenieren des Ergebnisses übersprungen.

5.3 Integrator mit dem Tenant Builder

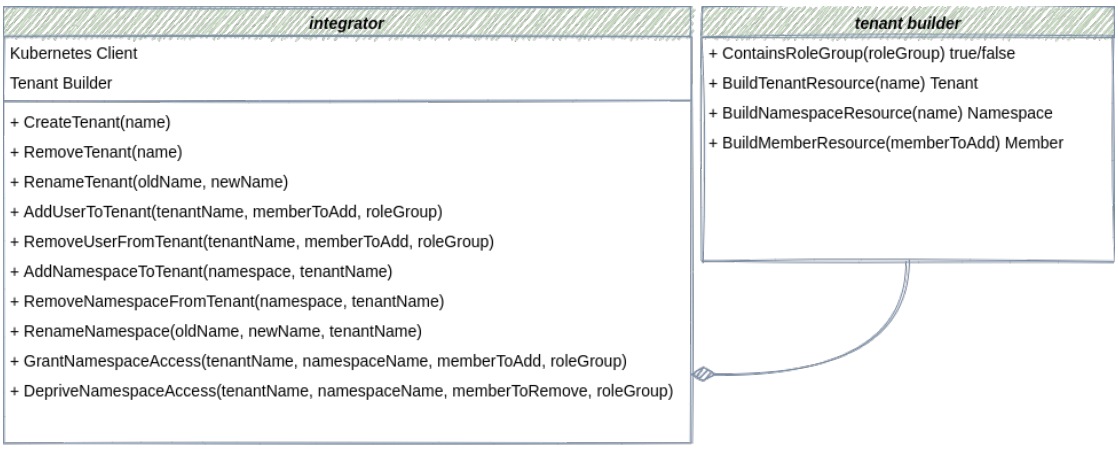

Abbildung 5.1: Ausschnitt eines Klassendiagrammes des `integrators` mit dem `tenant builder`.

Der `integrator` bietet alle Funktionalitäten, die zur Modifikation von Tenants möglich sind. Für das Erstellen und Modifizieren der Tenant-Ressource wird der `tenant builder` genutzt. Dieser vereinfacht den Umgang mit Tenant-Ressourcen, indem die Änderungsmöglichkeiten abstrahiert und auf das Wesentliche reduziert werden.

5.4 GitLab Events des Webhooks

Aus der Dokumentation von GitLab wurden die benötigten Events für den System-hook entnommen und als Structs in Go definiert. Welche Events benötigt werden, wurde bereits in 4.1 erläutert. Jedes Event erhält eine Funktion, über die nach der Deserialisierung die korrekte Funktion des Integrators aufgerufen wird.

Über die API des Webhooks werden alle HTTP-Anfragen mit dem HTTP-Verb `POST` empfangen, die an den Endpunkt `/webhook` gesendet werden. Damit eine Anfrage von dem Webhook angenommen wird, müssen drei HTTP-Header richtig gesetzt sein:

- `Content-Type` muss den Wert `application/json` haben,
- `X-Gitlab-Event` den Wert `System Hook` und
- `X-Gitlab-Token` muss dem Wert des Tokens entsprechen, der beim Start der Anwendung über die Konfiguration gesetzt wurde.

Ist einer dieser Header nicht gesetzt, wird mit dem HTTP-Status `400` und einer Fehlermeldung geantwortet. Ist die Anfrage allerdings korrekt, muss das System-Hook-Event aus dem Body deserialisiert werden. Dies wird in der Funktion `requestToEvent` abgewickelt, die im folgenden Listing 5.7 dargestellt wird:

Listing 5.7: Deserialisierung der GitLab Events. (`pkg/webook/webhook.go`)

```
1  func requestToEvent(request *http.Request) (eventName string,
       event *events.GitlabEvent, err error) {
2   body, err := ioutil.ReadAll(request.Body)
3   if err != nil {
4    return "unknown", nil, err
5   }
6
7   genericEvent := new(events.GitlabEventGeneric)
8   err = json.Unmarshal(body, genericEvent)
9   if err != nil {
10   return "unknown", nil, err
11  }
12
13  event, err = events.From(genericEvent.EventName)
14  if err != nil {
15   return genericEvent.EventName, nil, err
16  }
17
18  err = json.Unmarshal(body, event)
19  if err != nil {
20   return genericEvent.EventName, nil, err
21  }
22
23  return genericEvent.EventName, event, err
24 }
```

In Zeile 2 wird der gesamte Body der Anfrage als Bytes ausgelesen. Die Information, um welches Event es sich genau handelt, wird im Body der Anfrage mitgegeben. Dies führt dazu, dass die Information zwei mal deserialisiert werden müssen. Als ersten Schritt wird ein allgemeines Objekt deserialisiert, über das der Wert des Feldes `event_name`

ausgelesen werden kann (Zeile 7). Durch diesen Wert wird die Art des Events bestimmt und die damit zu erwarteten Informationen. In Zeile 13 geschieht dies über über die Funktion `events.From(string)`. Dabei wird über den Namen des Events das entsprechende Schema aus einer Map ausgelsesen und von der Funktion als leeres Struct zurückgegeben. In Zeile 19 wird das Schema dann mit Werten befüllt, indem der gesamte Body deserialisiert wird.

5.5 GitLab Client des Fetchers

Über den Fetcher werden die Ressourcen `users`, `groups` und `projects` der GitLab-API ausgelesen, deserialisiert und an den `integrator` (Siehe Paketstruktur 4.5.1) weitergegeben. Im Gegensatz zum `Webhook` können Ressourcen direkt deserialisiert werden, da eindeutig ist, um welche Art von Ressource es sich handelt.

5.6 Spezialfälle bei der Abbildung von Gruppen

In den folgenden Abschnitten wird auf die Implementierung einiger spezieller Fälle eingegangen.

5.6.1 Workaround für Projektmitglieder in Gruppen

Bei der Entwicklung fiel auf, dass die vorgegebene Spezifikation eines Tenants nicht ausreicht, um Gruppen korrekt abzubilden. Zugriffsberechtigungen werden in Tenant Ressourcen dargestellt, indem Nutzer, die Zugriff auf einen Namespace erhalten sollen, unter diesem gespeichert werden. Dies bedeutet, dass in GitLab zuerst ein Projekt erstellt werden muss und diesem dann Nutzer hinzugefügt werden können. Bei Gruppen ist es jedoch üblich, dass Nutzer der Gruppe hinzugefügt werden, bevor Projekte vorhanden sind. Daher benötigen diese also auch Zugriff auf die Namespaces in Kubernetes, die erstellt werden sobald die Gruppe Projekte erhält. Es ist also möglich, dass zum Zeitpunkt an dem Nutzer der Gruppe hinzugefügt werden, noch keine Projekte vorhanden sind und damit auch keine definierten Namespaces in der Tenant Ressource, unter denen Nutzer gespeichert werden können.

Im Folgenden wird eine Notlösung beschrieben, mit der die aktuelle Version des Tenant Integrators auch für Gruppen nutzbar ist. Diese ist in der Abbildung 5.2 dargestellt und wird darunter erläutert. Informationen von GitLab wurden fett hervorgehoben, und neue Informationen zudem noch grün.

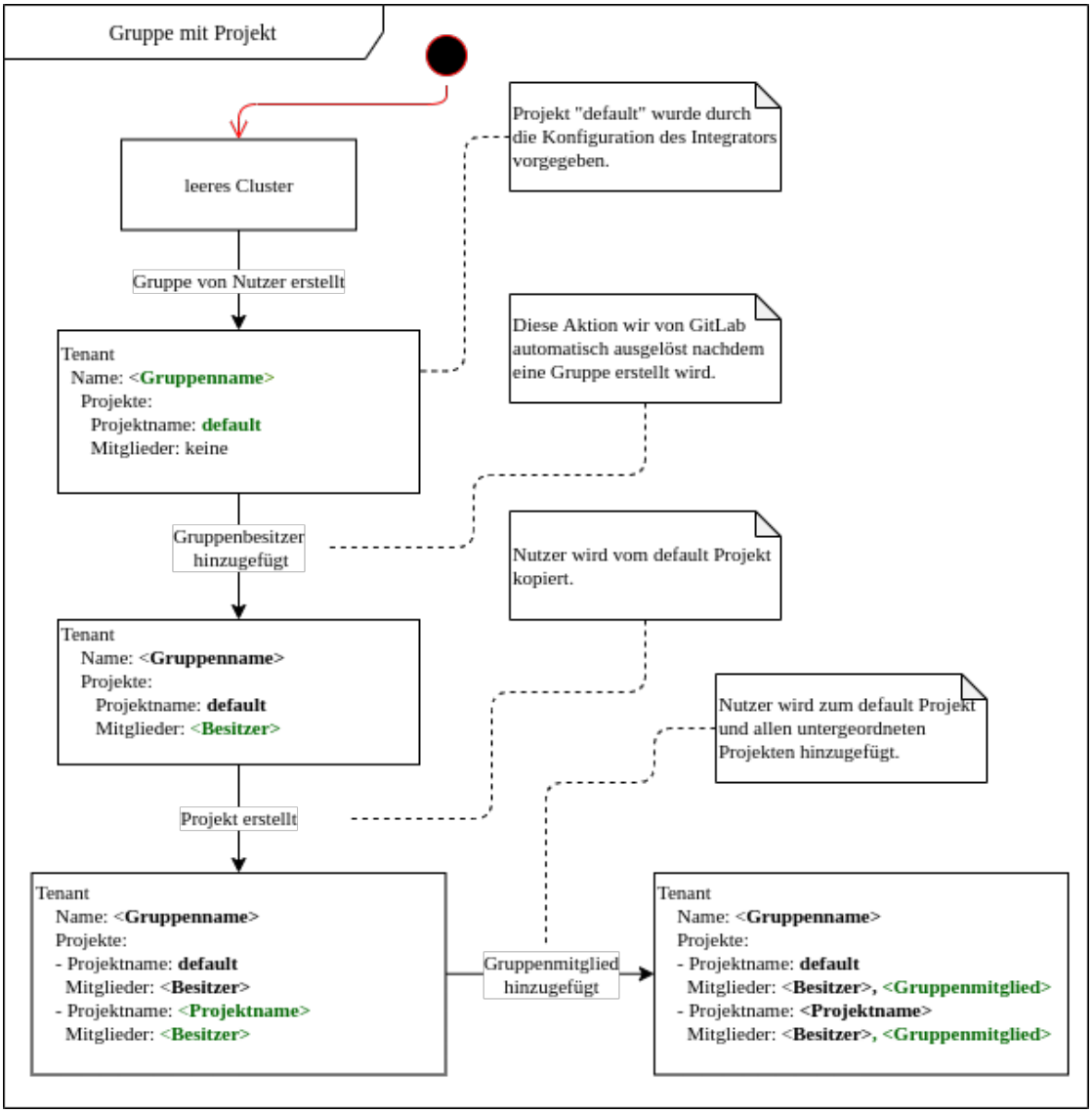

Abbildung 5.2: Workaround für neue Gruppenmitglieder in Projekten.

Wie in Abbildung 5.2 zu sehen, werden Gruppen als neue Tenant Ressourcen in Kubernetes gespeichert. Der Integrator kann so konfiguriert werden, dass jedem Tenant immer ein Namespace zur Verfügung steht. Dies wird genutzt, um die Nutzer einer Gruppe zu

speichern, bevor diese Gruppe ein Projekt in GitLab erstellt hat. In der Abbildung ist dies beispielhaft ein Namespace mit dem Namen „default“. Nachdem der Besitzer der Gruppe in GitLab hinzugefügt wurde, kann dieser unter dem Namespace „default“ gespeichert werden.
Wenn die Gruppe daraufhin Projekte erstellt, müssen die Nutzer unter dem „default“-Namespace auch zu allen neuen Namespace hinzugefügt werden. Im Beispiel ist zu sehen, wie der Besitzer vom „default“-Namespace dem Namespace des neuen Projektes zugewiesen wird.

Diese Art von Workaround hat den Vorteil, dass er nur im Tenant-Integrator implementiert werden muss und nicht im Tenant-Manager. Es wäre auch möglich gewesen die Mitglieder einer Gruppe nur in einem Namespace zu speichern ohne diese auf die neuen Projekte zu replizieren. Dazu müsste der Tenant-Manager jedoch wissen, welcher der Namespace ist unter dem die Mitglieder gespeichert sind. Dies wäre eine Lösung ohne redundante Daten, würde aber gegen die aktuelle Spezifikation der Tenant-Ressource sprechen, in der jeder Namespace alle seine Mitglieder angibt. Zudem wurde der Workaround nur im Tenant-Integrator Modul implementiert, da GitLab mit diesem Fehlverhalten nichts zu tun hat.

5.6.2 Vererbung von Gruppenmitgliedern aus übergeordneten Gruppen

In GitLab ist es möglich, untergeordnete Gruppen zu erstellen. Dabei haben alle Nutzer der übergeordneten Gruppe Zugriff auf diese. Informationen über diese Nutzer der übergeordneten Gruppe werden von GitLab jedoch nicht an den Webhook des Tenant-Integrator weitergeleitet. Es lassen sich, wie in 5.3 zu sehen, über den Pfad der Gruppe die Namen der übergeordneten Gruppen bestimmen und im Cluster sollte bereits ein Tenant für jede dieser Gruppen existieren. Dadurch können die Mitglieder auf die untergeordnete Gruppe repliziert werden. Für den Fetcher gilt dieses Problem nicht, da er über die Identifikationsnummern die übergeordneten Gruppen auslesen kann.

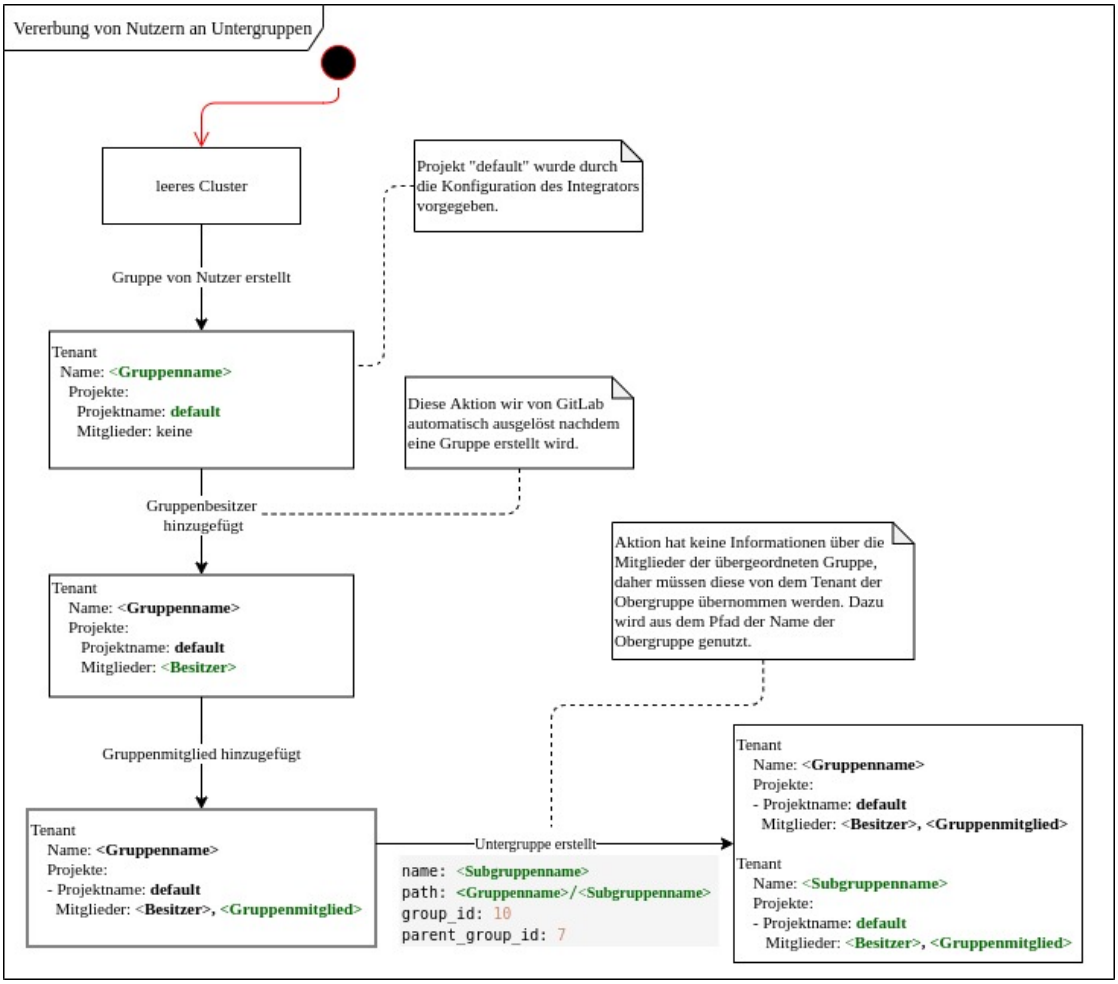

Abbildung 5.3: Workaround für die Vererbung von Gruppenmitgliedern aus übergeordneten Gruppen.

5.6.3 Hinzufügen von Mitgliedern in mehrschichtigen Gruppen

Wie im vorigen Abschnitt beschrieben, haben in Gitlab Mitglieder aus übergeordneten Gruppen Zugriff auf untergeordnete Gruppen. Zudem wurde gezeigt wie bei der Erstellung von Untergruppen, Mitglieder auf die Tenant-Ressourcen dieser Gruppen vererbt werden. Es ist aber auch möglich, dass Mitglieder der Obergruppe hinzugefügt werden, wenn Untergruppen schon existieren. Auch in diesem Fall müssen diese auf die Untergruppen vererbt werden.

Problemevaluation

In Abbildung 5.4 ist zu sehen, dass dieser Zustandsübergang aktuell vom Webhook nicht abgebildet werden kann. Beim Hinzufügen eines Nutzers wird lediglich der Pfad der Gruppe, zu dem der Nutzer hinzugefügt wurde, übergeben. Dieser Pfad besteht aus allen Obergruppen bis hin zu dem eigentlichen Gruppennamen (<Obergruppe1>/<Gruppe>). Informationen über untergeordnete Gruppen werden nicht übergeben. Dazu kommt, dass auch in den Tenant-Ressourcen keine Informationen über die Hierarchien von Gruppen gespeichert werden.

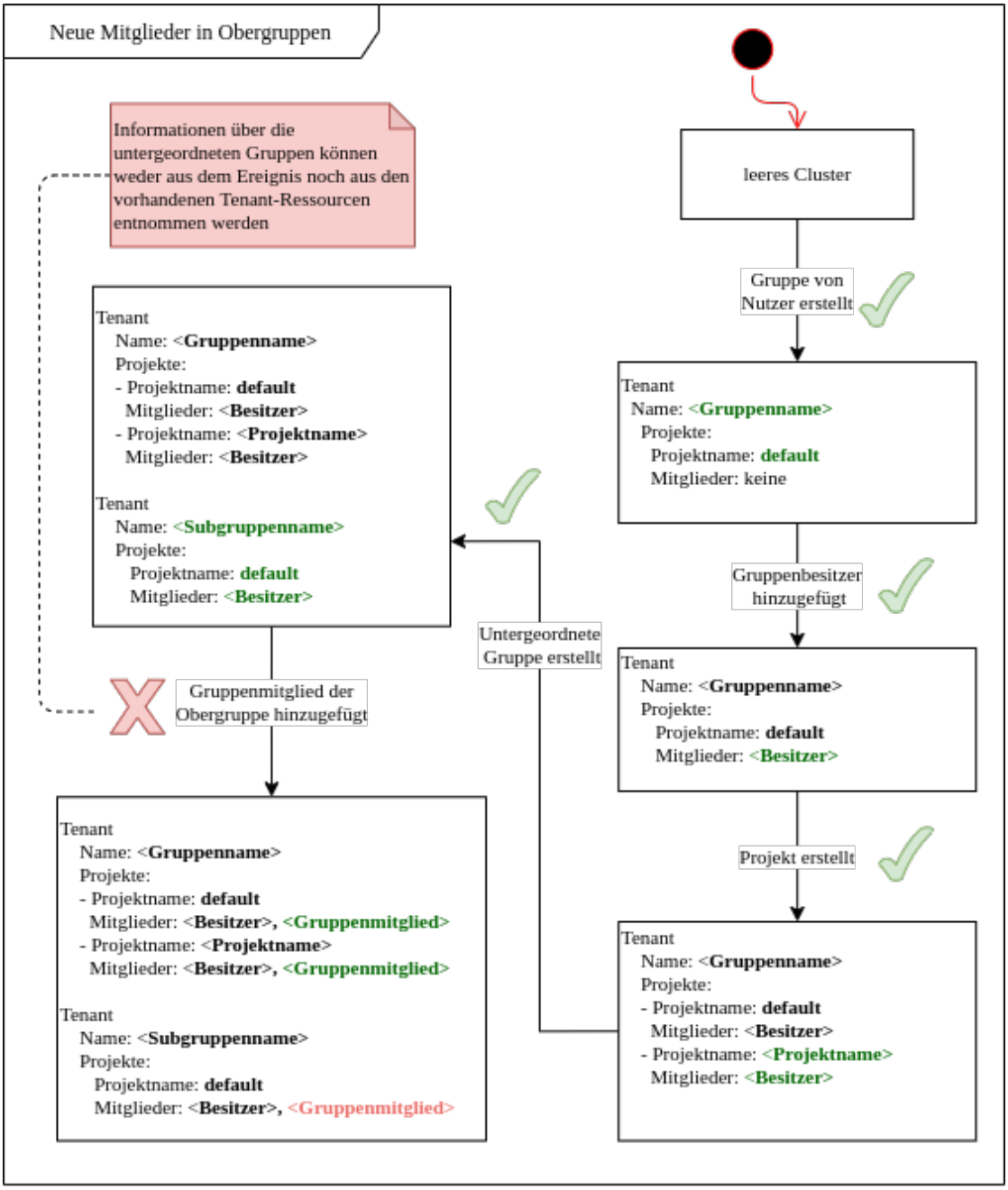

Abbildung 5.4: Problemdarstellung vom Hinzufügen neuer Mitglieder in mehrschichtigen Gruppen.

Workaround

Für den Webhook des Tenant-Integrators existiert zum Zeitpunkt dieser Arbeit kein Workaround. Es ist jedoch möglich durch manuelles Anstoßen des Fetchers den kompletten Zustand aus GitLab zu synchronisieren. Dadurch können fehlende Gruppenmitglieder aus den Obergruppen übernommen werden. Dies liegt daran, dass der Fetcher alle Gruppen durchläuft. Trifft der Fetcher auf eine Gruppe mit übergeordneten, werden auch die Mitglieder aller übergeordneten Gruppen übernommen.

Lösungsvorschläge

Des Weiteren werden einige Lösungsvorschläge vorgestellt:

1. Ausführen des Fetchers vom Webhook:
   Im Gegensatz zum Workaround ließe sich der Fetcher auch vom Webhook starten. Der Fetcher könnte in der laufenden Anwendung des Webhooks selber als extra Thread laufen oder vom Webhook als Job in Kubernetes gestartet werden.
   Bei beiden dieser Lösungen vermischen sich die Aufgabenverteilungen dieser Anwendungen und machen diese voneinander abhängiger.
2. Automatisiertes Ausführen des Fetchers:
   Anstatt den Fetcher als einfachen Job nur einmal zum Zeitpunkt des Deployments zu starten, könnte dieser auch in einem bestimmten Zeitabstand wiederholt werden. Durch diese Lösung würde nicht sofort, nach dem Hinzufügen eines Mitglied in einer Obergruppe, ein konsistenter Zustand in Untergruppen bestehen. Jedoch würde dieser nach Ablauf der gewählten Zeitspanne des Jobs eintreten.
3. Anpassung der Tenant-Ressource:
   Die vorangegangenen Lösungsvorschläge versuchen eine Lösung für die gegebene Spezifikation eines Tenant zu finden. Es wäre allerdings auch möglich, die Tenant-Ressource so anzupassen, dass diese hierarchische Gruppen korrekt abbilden könnte. Beim Erstellen einer Untergruppe müsste der Obergruppe lediglich ein Verweis auf die neu erstellte Untergruppe hinzugefügt werden. Wie in Abbildung 5.3 gezeigt, wird die Information über Obergruppen bei der Erstellung mitgegeben und könnte somit das Problem lösen.

Im Laufe dieser Arbeit wird keiner der vorgeschlagenen Lösungen umgesetzt, da es am sinnvollsten erachtet wird, die Definition der Tenant-Ressource anzupassen und diese

vom Tenant-Manager vorgegeben wird.
Der Grund dafür ist, dass es sinnvoll ist, den Zugriff auf Kubernetes zu verschachteln, um Hierarchien korrekt abbilden zu können. Ohne das Verschachteln kann einem Tenat auch immer nur ein ganzes Stück des Clusters zugeordnet werden. Die kleinste Einheit zur Unterteilung bleibt in Kubernetes ein Namespace, was einem Projekt in GitLab entspricht. Danach können diese in einem Tenant zusammengefasst werden, welche daraufhin weiter verknüpft werden müssen, um diese Stücke zusammenzufassen.

5.7 Deployment

Wie im Kontext zu GitLab 4.3.1 beschrieben, wird der Tenant Integrator in zwei verschiedenen Varianten gestartet werden, um beim Deployment zuerst den aktuellen Zustand aus GitLab mit dem `Fetcher` zu synchronisieren und dann nur noch auf Zustandsänderungen mit dem `Webhook` zu reagieren.
Für das Deployment wird der Soll-Zustand der Kubernetes Ressourcen in YAML-Dateien beschrieben, die in 2.6.2 erläutert wurden.

Das Deployment besteht aus einem Kubernetes-Deployment für den `Webhook` und einem Kubernetes-Job, über den einmalig am Start der `Fetcher` gestartet wird. Damit alle Zustandsänderungen von GitLab synchronisiert werden, sollte der `Webhook` vor dem `Fetcher` laufen, sonst könnten Zustandsänderungen verloren gehen, die zwischen dem Start der beiden Anwendungen stattfinden. Doppeltes Synchronisieren stellt jedoch kein Problem dar, da das Ergebnis identisch bleibt.

Damit die Anwendungen auch Zugriff auf die Kubernetes-API haben und Tenant-Ressourcen anlegen können, wird ein Service Account mit bereitgestellt. Durch diesen erhalten die Anwendungen im ganzen Cluster administrativen Zugriff auf die Tenant-Ressource.
Die Konfiguration der Anwendungen werden über drei ConfigMaps an die Pods übergeben. `Webhook` und `Fetcher` erhalten dabei jeweils eine eigene ConfigMap für Konfigurationen, die nur auf den aktuellen Anwendungszweck zugeschnitten sind. Der `Fetcher` muss zum Beispiel nichts über die GitLab API wissen. Dazu kommt eine gemeinsame ConfigMap für die Gruppierung von Nutzern in einer Tenant-Ressource (Siehe roleGroups im Listing 5.4).

Das Deployment wurde in einem lokalen Cluster getesten. Genutzt wurde dafür Minikube. Minikube führt ein Kubernetes-Cluster mit einem einzigen Node in einer VM aus. Um

das gesamte Deployment testen zu können, muss zuerst der Support für die Verwendung eines Ingress in Minikube aktiviert werden. Dies ist möglich über den Befehl: „`minikube addons enable ingress`“.
Minikube besitzt einen eigenen Docker-daemon. Durch diesen ist es möglich, dass lokal ein Image gebaut werden kann und damit in Kubernetes zur Verfügung steht. Durch diese Funktionalität entfällt der Umweg über ein externes Repository, in dass das Image zwischengespeichert werden müsste.

5.7.1 Dockerisierung

Im Container wird zur Laufzeit nur die auszuführende Binärdatei benötigt. Es ist jedoch von Vorteil den Bauprozess der Binärdatei im Container durchzuführen, um das Deployment so einfach wie möglich zu halten. Das Kapseln der Build- und Deployment-Logik in die Container-Definition hat den Vorteil, dass das Deployment aus Sicht des Programmes, welches das Deployment durchführt, immer gleich abläuft; die Anwendung wird containerisiert und danach gestartet. Es wird dabei kein programmiersprachenspezifisches Wissen über das Laden von Abhängigkeiten oder den Kompiliervorgang benötigt.
In Docker ist der beschriebene Vorgang durch „multi-stage-builds“ möglich. Dabei können mehrere Container hintereinander gestartet werden und Dateien von einem Container in den nächsten kopiert werden. Jeder Container stellt dabei eine „Stage“ dar. In der Dockerfile (Listing 5.8) wird dieses Vorgehen genutzt, um einen möglichst leichtgewichtigen Container mit nur einer Binärdatei des Systems zu erhalten.

Listing 5.8: Dockerfile

```
1  ### STEP 1: build executable binary
2  ARG GO_VERSION=1.12
3  FROM golang:${GO_VERSION}-alpine AS builder
4
5  ENV GO111MODULE=off
6
7  WORKDIR /go/src/github.com/kubetenancy/gitlab-tenant-integrator
8
9  COPY . ./
10
11 RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build \
12       -o /go/bin/gitlab-tenant-integrator .
```

```
13
14 ### STEP 2: build a small image
15 FROM scratch
16
17 # Copy our static executable
18 COPY --from=builder /go/bin/gitlab-tenant-integrator go/bin/
       gitlab-tenant-integrator
```

Im Listing 5.8 wird in Zeile 3 das Image `golang:1.12-alpine` als Basis verwendet und als die erste Stage mit dem Label „builder“ versehen. Auf dem Basisimage befindet sich Golang in der Version 1.12, installiert auf dem Betriebssystem Alphine Linux, einer leichtgewichtigen Version von Linux.
In Zeile 9 werden alle Dateien des Projektes in den Container kopiert und dann in Zeile 11 kompiliert. Es wird darauf verzichtet, die Abhängigkeiten im Container zu laden. Stattdessen werden die Abhängigkeiten über den `vendor`-Ordner, welcher sich im Hauptverzeichnis des Projektes befindet, direkt mit in den Container kopiert. Der Grund dafür ist, dass sich das „Tenant Manger“-Projekt zur Zeit der Arbeit in einem privaten Repository auf GitHub befindet.
In Zeile 15 beginnt die zweite Stage, indem ein leeres Image als Basis für den neuen Container verwendet wird. In Zeile 18 wird dann nur die kompilierte Binärdatei aus dem ersten Container in neuen kopiert.
Da die Anwendung in verschiedenen Modi gestartet werden soll, wird in Dockerfile kein `Entrypoint` mit angegeben. Ein `Entrypoint` gibt sonst an, welcher Befehl beim Start eines Containers standardmäßig ausgeführt wird.

5.8 Tests

In diesem Abschnitt wird beschrieben, wie die Integrations- und Akzeptanztests aufgebaut wurden und was von diesen abgedeckt wird. Auf die einzelnen Unit-Tests wird nicht eingegangen werden.

5.8.1 Integrationstests

Beim Integrationstest wird das Zusammenspiel der einzelnen Komponenten getestet, dazu gehören unter anderem Klassen und Pakete eines Moduls. Um die Wiederverwendbarkeit

des Tenant-Integrators mit anderen Informationsquellen als GitLab zu gewährleisten, werden alle Methoden der Schnittstelle durch Integrationstests abgedeckt. Damit wird die Qualitätsanforderung 3.3 verifiziert, durch die gefordert wurde, dass sich neue Informationsquellen anschließen lassen.
Die Tests rufen die Methoden der Schnittstelle auf und überprüfen das Ergebnis in Kubernetes über die Kubernetes-API. In der Qualitätsanforderung 3.2 wurde gefordert, dass Tests lokal und ohne Seiteneffekte ausgeführt werden können. Aus diesem Grund wird in den Tests ein falscher Client für Kubernetes erzeugt. Alle Zustände werden im Arbeitsspeicher gespeichert und lassen sich wie gewohnt abfragen. Dadurch ist es möglich die Tests isoliert auszuführen ohne ein Kubernetes-Cluster hochzufahren.

5.8.2 Akzeptanztests

Die Akzeptanztests wurden in den funktionalen Anforderungen aus Kapitel 3.2 definiert. Um die Korrektheit des Gitlab-Tenant-Integrators zu überprüfen, wurden Tests geschrieben, die über einen Client die REST-API des Webhooks aufrufen. Damit wie bei den Integrationstests isoliertes Testen ohne das Hochfahren eines Kubernetes-Clusters möglich ist, wurde auch in diesen Tests auf einen falschen Kubernetes-Client zurückgegriffen.
Dies bringt den Nachteil mit sich, dass der Webhook in der Testsuite initialisiert und gestartet werden muss, damit dieser Zugriff auf den falschen Kubernetes-Client hat und die Ergebnisse auch verifizieren kann. Dadurch werden, wie in Abbildung 5.5 zu sehen, die Pakete `cmd` und `config` nicht abgedeckt. Pakete, die von den Akzeptanztests abgedeckt werden, wurden grün hervorgehoben. Das `config`-Paket wird jedoch bereits durch Unit Tests abgedeckt. Das `cmd`-Paket muss durch manuelles Testen verifiziert werden. Um die Interaktion mit der Kommandozeile zu testen, müsste die Anwendung in einem eigenen Prozess gestartet werden, wodurch kein Zugriff auf den falschen Kubernetes-Client möglich wäre und damit auch nicht auf die Ergebnisse. Für den Fetcher wurden die Akzeptanztests genauso aufgebaut. Dazu kommt, dass der Client für GitLab einen falschen Client erhält, um die REST-API von GitLab zu simulieren.

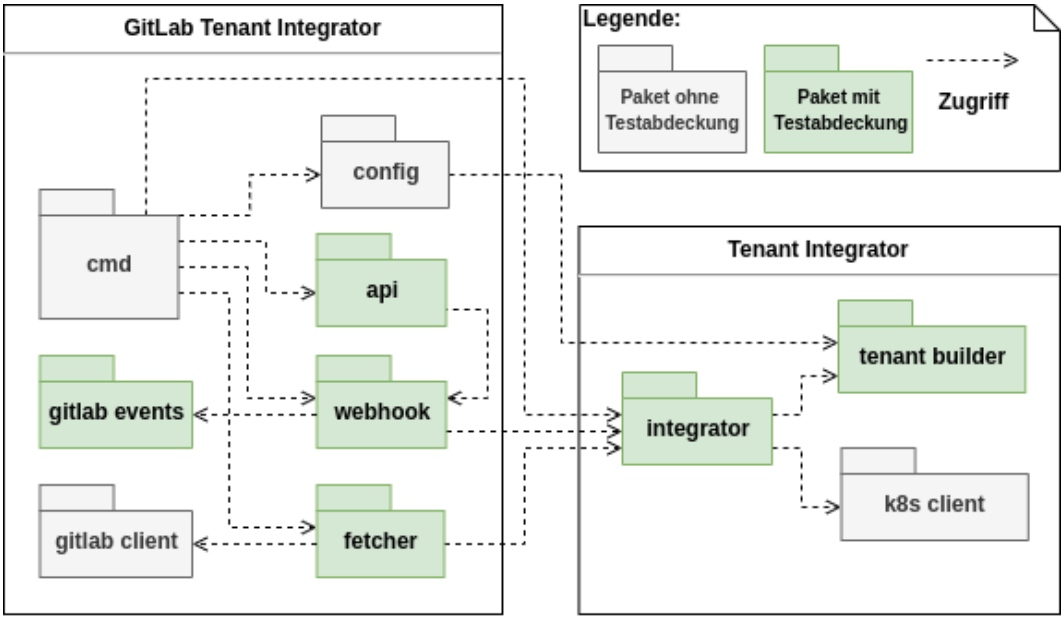

Abbildung 5.5: Übersicht über die Testabdeckung durch Akzeptanztests der Pakete.

6 Evaluation

6.1 Verifikation

In der Verifikation wird diskutiert, inwieweit das finale System der Spezifikation entspricht. Dazu werden die erarbeiteten User Stories und Qualitätsanforderungen aus der Anforderungsanalyse 3 betrachtet und anschließend mit dem Ist-Zustand verglichen.

6.1.1 Verifikation der funktionalen Anforderungen

Die Vollständigkeit der funktionalen Anforderungen wird durch die definierten Akzeptanztests aus Kapitel 3.2.1 und 3.2.2 bestimmt. Diese Test wurden wie in 5.8.2 beschrieben, umgesetzt und bestätigen damit die Vollständigkeit der Anwendung.

Die funktionalen Anforderungen wurden für den Gesamtprozess inklusive des Tenant-Managers beschrieben. Im Gegensatz zum Gesamtprozess werden vom Tenant-Integrator die Tenant-Ressourcen in Kubernetes nur als Soll-Zustand definiert und nicht umgesetzt. Dies bedeutet, dass wenn in den Akzeptanztests ein Arbeitsbereich für Nutzer gefordert ist, dieser lediglich in einer Tenant-Ressource definiert wird und noch kein Arbeitsbereich den Nutzer zur Verfügung steht. Die Umsetzung ist Aufgabe des Tenant-Managers. Genauso können einige Akzeptanztests nur von dem Tenant-Manager umgesetzt werden, wie zum Beispiel: „Der Arbeitsbereich ist nur für den Nutzer sichtbar, dem das zugehörige Projekt gehört.“ (User Stories 3.2.1)

6.1.2 Verifikation der Qualitätsanforderungen

In diesem Kapitel wird erörtert inwieweit die Qualitätsanforderungen erfüllt wurden. Zur Verständlichkeit werden noch einmal die in Kapitel 3.3 beschriebenen Szenarien und daraus resultierenden Anforderungen aufgeführt.

Änderbarkeit

Szenario: Das Standardverhalten des Systems soll geändert werden.
Anforderung: Administratoren sind in der Lage das Standardverhalten des Systems zu konfigurieren.

Um dieser Anforderung nachzukommen, werden von der Anwendung beim Start festgelegte Konfigurationen ausgelesen. Konfigurierbar ist unter anderem, welche Berechtigungsgruppen dem Tenant zur Verfügung stehen, sowie Arbeitsbereiche, welche dem Tenant bei der Erzeugung automatisch zugeordnet werden. Ausgelesen werden festgelegte Werte aus einer Konfigurationsdatei und aus Umgebungsvariablen. Beide Varianten sind für unterschiedliche Szenarien nützlich. Für Entwickler ist es komfortabler und übersichtlicher, die Konfigurationen gebündelt in einer Datei zu verwalten, wenn sie an der Anwendung arbeiten und diese lokal ausführen. Bei dem Deployment ist es hingegen von Vorteil, die Konfiguration von der Anwendung zu trennen. Dafür werden von der Anwendung beim Start Umgebungsvariablen ausgelesen, welche die Konfigurationsdatei ersetzen. Beim Deployment werden die Umgebungsvariablen über eine ConfigMap verwaltet. Dadurch lassen sich Werte abändern, ohne die Anwendung erneut zu deployen.

Szenario: Entwickler möchten an dem System möglichst komfortabel und ohne Seiteneffekte Änderungen vornehmen können.
Anforderung: Anwendung und Tests der Anwendung müssen lokal lauffähig sein.

Aufgrund dieser Anforderung wurden Anwendung und Tests so entworfen, dass diese ohne GitLab oder Kubernetes ausführbar sind. Dies wurde in Kapitel 5.8 bereits genauer erläutert.

Szenario: Entwickler können neue Informationsquellen anschließen.
Anforderung: Entwickler sind in der Lage, auf Basis vordefinierter Schnittstellen das Grundgerüst des Systems zu erweitern.

Wie in der Bausteinsicht 4.5 beschrieben, wurde das System aufgrund dieser Anforderung in zwei Module zerteilt. Durch diese Aufteilung lässt sich ein Teil des Gesamtsystems wiederverwenden. Die gesamte Logik rund um das Verwalten der Tenant-Ressourcen befindet sich im Tenant-Integrator-Baustein und kann so wiederverwendet werden. Es muss lediglich der GitLab-Tenant-Integrator-Baustein ersetzt werden.

Szenario: Developer Experience

Anforderung: Durch ausreichende Dokumentation in der README, soll es Entwicklern unmittelbar möglich sein, das System zu starten, zu deployen, zu testen und Änderungen vorzunehmen.

Um die Weiterentwicklung des zu entwickelnden Systems zu erleichtern, wurde eine README-Datei mit ausführlichen Informationen für Entwickler beigelegt. Dazu kommt eine Makefile, durch die komplexere Kommandos zusammengefasst und vereinfacht werden konnten. Die README enthält alle Informationen, die notwendig sind, um das System in den vorgesehenen Umgebungen zu starten. Zu den verfügbaren Umgebungen gehört der lokale Rechner und Kubernetes. Wobei es möglich ist, das System als lokalen Prozess, als Container mit Docker oder in Minikube zu starten. Darüber hinaus wird beschrieben, wie eine GitLab Instanz lokal gestartet und korrekt konfiguriert werden kann.

6.1.3 Benutzbarkeit

Szenario: Validierung von Benutzereingaben.

Anforderung: Das System nimmt korrekte Eingabedaten an, weist fehlerhafte Eingabedaten zurück und gibt diese im Log aus.

Eingaben von Benutzern werden vor der Verarbeitung überprüft und bei nicht korrekten Eingaben wird eine Fehlermeldung zurückgegeben ohne das System herunterzufahren. Sollte es doch zu einem Fehlerfall kommen, der einen Ausfall des Systems bedeutet, wird das fehlerhafte System durch Kubernetes beendet und durch eine neue Instanz ausgetauscht. Dafür wurde dem Webhook ein Endpunkt hinzugefügt. Im Deployment wurde festgelegt, dass dieser Endpunkt von Kubernetes genutzt werden kann, um die Verfügbarkeit des System zu überprüfen.

Effizienz

Szenario: Ausführung von Tests in weniger als 3 Minuten.

Anforderung: Sämtliche Testfälle sind in weniger als 3 Minuten ausgeführt.

Dadurch, dass bei den Tests keine GitLab oder Kubernetes Instanzen verwendet werden, lassen sich diese in wenigen Sekunden ausführen und damit Modifikationen am System verifizieren.

Szenario: Verarbeitung sollen innerhalb einer Sekunde erfolgen.

Anforderung: Verarbeitung und Antwort einer Anfrage sollen in unter einer Sekunde erfolgen.

Einzelnen Anfragen werden innerhalb weniger Millisekunden verarbeitet. Dies hat jedoch nur geringe Aussagekraft, da es unter hoher Last zu Abweichungen kommen kann. Damit konnte die Anforderung noch nicht verifiziert werden, da noch keine Lasttests für das System vorgenommen wurden.

Zuverlässigkeit

Szenario: Fehler in der Verarbeitung schnell identifizieren.
Anforderung: Relevante Informationen werden nach Auftreten des Fehlers im Log ausgegeben.

Die Anwendung wurde so entwickelt, dass für alle bekannten Fehlerfälle an der betroffenen Stelle Logs ausgegeben werden.

Funktionalität

Szenario: Der Integrator lässt sich in- und außerhalb von Kubernetes betreiben.
Anforderung: System ist in- und außerhalb von Kubernetes lauffähig.

Das System wurde aufgrund dieser Anforderung so entwickelt, dass beim Start mehrere Modi zur Verfügung stehen. Notwendig ist dies, da die Anwendung je nach Umgebung einen anderen Client für Kubernetes nutzt. Abhängig von der gewählten Umgebung wird bei dem Client ein anderer Weg zur Authentifizierung verwendet. Hierfür wird im Cluster der Service Account aus dem Deployment genutzt. Läuft die Anwendung außerhalb von Kubernetes, muss eine Datei zur Authentifizierung beigelegt werden. Für die vereinfachte Verwendung wurde zudem noch ein Modus hinzugefügt, bei dem ein falscher Kubernetes-Client verwendet werden kann. Bei dieser Art von Client wird der Zustand im Arbeitsspeicher gehalten.

6.2 Validation

Die Validation beschäftigt sich mit der Frage, inwieweit das finale System für den geplanten Einsatzzweck geeignet ist.

Ohne ausgiebige Tests durch potenzielle Nutzer lässt sich diese Frage nicht endgültig klären. Potenzielle Nutzer sind jedoch auf die Möglichkeiten in GitLab beschränkt. Dadurch konnten Funktionalitäten des Systems bereits weitestgehend durch die Verifizierung der Tests und das Durchspielen von Szenarien in GitLab validiert werden.

Durch die aufgestellten Qualitätsanforderungen wurde das System für Entwickler aus technischer Sicht ausreichend dokumentiert und mit unterstützenden Skripten für die vereinfachte Weiterentwicklung versehen. Selbst wenn Entwickler länger nicht mit dem System arbeiten, sollte ein Wiedereinstieg schnell und einfach möglich sein.

6.3 Fazit

Das Projekt ist mit dem Ziel gestartet, eine Anwendung zu entwickeln, die automatisiert Tenant-Ressourcen in Kubernetes mit Nutzungsinformationen aus GitLab synchronisiert. Dieser Soll-Zustand eines Tenant gibt alle Informationen vor, die für eine Umsetzung einer Multi-Tenancy in Kubernetes benötigt werden. Die Erweiterbarkeit und Zuverlässigkeit des Integrators war dabei ein besonderer Fokus. Der Integrator sollte ohne Ausfälle in der ICC der HAW betrieben werden und auch in Zukunft leicht erweiterbar sein. Zudem sollte der Integrator so generisch aufgebaut sein, dass es ohne große Umstände möglich ist, diesen für neue Informationsquellen anzupassen.
Diese Ziele wurden durch das Sammeln von Qualitätsanforderungen vervollständigt. Es wurde darauf geachtet, dass das System für die zukünftige Weiterentwicklung von Entwicklern leicht verstanden, erweitert und getestet werden kann. Die erfolgreiche Umsetzung wurde in der Verifikation gezeigt.

Eine Herausforderung war das Testen und korrekte Implementieren von allen möglichen Änderungen an einer Tenant-Ressource. Aus diesem Grund wurden für diese Zustände und deren Übergänge ein Zustandsdiagramm erstellt. Das Diagramm diente als Leitfaden für die Implementierung und das Umsetzen einer ausreichenden Testabdeckung. So ergab sich besonders, dass das Abbilden von Gruppen durch den Webhook mit der aktuellen Definition eines Tenant nicht für alle Zustände möglich war. Aus diesem Grund wurde ein Workaround implementiert, bei dem Projektmitglieder in einem vordefinierten Namespace zwischengespeichert werden, bis sie auf Namespaces eines Projektes repliziert werden. Dadurch war es möglich Nutzern gleich mehreren Namespaces zuzuweisen.
Eine noch größere Herausforderung war das Abbilden vom Hinzufügen von Mitgliedern

in mehrschichtigen Gruppen. Es war beim Hinzufügen von Nutzern nicht möglich Untergruppen zu identifizieren und die Nutzer diesen Gruppen zuzuweisen. Dazu wurden jedoch Lösungsvorschläge erarbeitet und darauf hingewiesen, dass eine Hierarchie auch bei einem Tenant sinnvoll wäre.

6.4 Ausblick

Um das Projekt zu vervollständigen, sollte die Tenant-Ressource für Gruppen optimiert werden. Die Probleme der Tenant-Ressource wurden in dieser Arbeit beschrieben und Lösungsvorschläge sowie ein Workaround wurden vorgestellt. Diese können für die Weiterentwicklung verwendet werden.
Ein weiterer Teil des Tenant-Integrators mit Optimierungsbedarf ist die initiale Konfiguration. Um eine Änderung an den Berechtigungsgruppen oder den vordefinierten Namensräumen bei dem Tenant-Integrator zu ändern, ist es notwendig, nach der Anpassung alle Tenant-Ressourcen zu löschen und durch den Fetcher neu erstellen zu lassen. Der Webhook alleine würde die neue Konfiguration nur für neue Tenant-Ressourcen übernehmen. Dieser Prozess sollte für eine einfache und ausfallfreie Migration der Tenant-Ressourcen optimiert werden.
In der Verifikation der Qualitätsanforderungen wurde darauf hingewiesen, dass die schnelle Antwortzeit auf Anfragen noch nicht ausreichend bestätigt werden konnte. Hier wäre es notwendig zu zeigen, dass die Antwortzeit des Webhooks auch unter Last weniger als eine Sekunde entspricht.
Ein weiterer Punkt, der in dieser Arbeit noch nicht beachtet wurde, ist das Thema Sicherheit. Die Token für die GitLab API und den Webhook werden aktuell noch im Deployment übergeben. Hier sollte auf eine andere Lösung zurückgegriffen werden, wie etwa die in den Grundlagen beschriebenen Secrets von Kubernetes. Darüber hinaus verwendet der Integrator aktuell noch HTTP. Hier sollte durch die Nutzung eines TLS Zertifikat HTTPS verwendet werden.
Nachdem die vorgeschlagenen Optimierungen gemacht wurden, könnten Stories aus der Anforderungsanalyse umgesetzt werden, die noch nicht in dieser Arbeit behandelt wurden. Dazu gehört hauptsächlich das Definieren und Anpassen von Ressourcenlimits. Diese Limits würden dafür sorgen, dass sich hoher Ressourcenverbrauch von einem Tenant nicht auf andere auswirken kann.

Literaturverzeichnis

[Cohn 2004] Cohn, Mike: *User Stories Applied - For Agile Software Development*. Boston : Addison-Wesley Professional, 2004. – ISBN 978–0–321–20568–1

[Config Maps] Config Maps: *Kubernetes Config Maps*. https://kubernetes.io/docs/concepts/configuration/configmap/. – Accessed: 05.12.2020

[custom resources] custom resources: *Kubernetes Custom Resources*. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/. – Accessed: 25.06.2019

[Docker overview] Docker overview: *Docker Engine Overview*. https://docs.docker.com/engine/docker-overview/. – Accessed: 26.06.2019

[Dockerfile reference] Dockerfile reference: *Dockerfile Reference*. https://docs.docker.com/engine/reference/builder/. – Accessed: 27.06.2019

[code generator] generator code: *Code Generator*. https://github.com/kubernetes/code-generator. – Accessed: 02.10.2020

[gengo] gengo: *gengo*. https://github.com/kubernetes/gengo. – Accessed: 11.10.2020

[Jessie Frazelle] Jessie Frazelle: *Multi-Tenancy Design Space*. https://docs.google.com/document/d/1PjlsBmZw6Jb3XZeVyZ0781m6PV7-nSUvQrwObkvz7jg/edit#heading=h.vk6zyqy9mghw. – Accessed: 27.06.2019, Has private access restrictions

[Krebs, Momm and Kounev] Krebs, Momm and Kounev: *Architectural Concerns in Multi-Tenant SaaS Applications*. https://www.researchgate.net/publication/264942141_Architectural_Concerns_in_Multi-Tenant_SaaS_Applications. – Accessed: 27.06.2019

[kube-controller-manager] KUBE-CONTROLLER-MANAGER: *Kubernetes Controller Manager*. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager. – Accessed: 25.06.2019

[kube-scheduler] KUBE-SCHEDULER: *Kubernetes Scheduler*. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/. – Accessed: 25.06.2019

[kubernetes concepts] KUBERNETES CONCEPTS: *Kubernetes Konzept*. https://kubernetes.io/docs/concepts/. – Accessed: 25.06.2019

[Kubernetes Deployment] KUBERNETES DEPLOYMENT: *Kubernetes Deployment*. https://cloud.google.com/kubernetes-engine/docs/concepts/deployment. – Accessed: 11.11.2020

[Kubernetes Ingress] KUBERNETES INGRESS: *Kubernetes Ingress*. https://kubernetes.io/docs/concepts/services-networking/ingress/. – Accessed: 11.11.2020

[Kubernetes Jobs] KUBERNETES JOBS: *Kubernetes Job*. https://kubernetes.io/docs/concepts/workloads/controllers/job/. – Accessed: 11.11.2020

[Kubernetes Nodes] KUBERNETES NODES: *Kubernetes Nodes*. https://kubernetes.io/de/docs/concepts/architecture/nodes/. – Accessed: 28.05.2020

[Kubernetes Services] KUBERNETES SERVICES: *Kubernetes Service*. https://kubernetes.io/docs/concepts/services-networking/service/. – Accessed: 11.11.2020

[Namespaces] NAMESPACES: *Kubernetes Namespaces*. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/. – Accessed: 20.1.2021

[Patton 2015] PATTON, Jeff: *User Story Mapping*. Deutsche Ausgabe. O'Reilly, 2015. – ISBN 978–3–95875–067–8

[Pods] PODS: *Kubernetes Pods*. https://kubernetes.io/docs/concepts/workloads/pods/. – Accessed: 11.11.2020

[RBAC] RBAC: *Kubernetes RBAC*. https://kubernetes.io/docs/reference/access-authn-authz/rbac/. – Accessed: 11.11.2020

[Rob Pike] Rob Pike: *Go at Google: Language Design in the Service of Software Engineering*. https://talks.golang.org/2012/splash.article. – Accessed: 20.10.2020

[Ron Jeffries 2000] Ron Jeffries: *Extreme Programming Installed*. Addison-Wesley Longman Publishing Co, 2000 https://dl.acm.org/citation.cfm?id=557459. – ISBN 0201708426. – Accessed: 14.02.2019

[Secrets] Secrets: *Kubernetes Secrets*. https://kubernetes.io/docs/concepts/configuration/secret/. – Accessed: 05.12.2020

[Service Accounts] Service Accounts: *Kubernetes Service Account*. https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/. – Accessed: 01.12.2020

[Starke] Starke, Gernot: *arc42*. https://arc42.org/overview. – Accessed: 15.12.2019

[Starke 2018] Starke, Gernot: *Effektive Softwarearchitekturen*. Hanser, 2018. – ISBN 9783446452077. – Accessed: 30.11.2019

[Steve Francia] Steve Francia: *Is Go An Object Oriented Language?* https://spf13.com/post/is-go-object-oriented/. – Accessed: 20.10.2020

[System hooks] System hooks: *system hooks*. https://docs.gitlab.com/ee/system_hooks/system_hooks.html. – Accessed: 01.10.2020

[viper] viper: *viper*. https://github.com/spf13/viper. – Accessed: 04.09.2019

[What is a Container] What is a Container: *Docker container*. https://www.docker.com/resources/what-container. – Accessed: 25.06.2019

[What is Kubernetes] What is Kubernetes: *Kubernetes erklärt*. https://kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/. – Accessed: 25.06.2019

Glossar

API API steht für Application Programming Interface oder in deutsch Programmierschnittstelle. Sie dient dazu, Informationen zwischen einer Anwendung und einzelnen Programmteilen standardisiert auszutauschen. Mit ihr können Anwendungen unabhängig von ihrer Implementierung untereinander kommunizieren.

Bibliothek Eine Bibliothek oder auch Library ist fremder Code der Funktionen über eine Schnittstelle anbietet.

CI/CD „CI“ bedeutet Continuous Integration, also der Automatisierungsprozess für Entwickler. „CD“ bedeutet Continuous Delivery/Deployment, also die automatische Auslieferung der Anwendung nach einer Änderung
.

composition over inheritance Bei diesem Prinzip handelt es sich um ein Delegationsmuster, bei dem durch Komposition die gleiche Wiederverwendbarkeit erreicht wird wie durch Vererbung.

Git Git ist ein Tool zur Versionskontrolle von Software.

REST REST (Representational State Transfer) ist ein Architekturstile zum Datenaustausch zwischen Softwaresystemen bei dem alles in Ressourcen definiert wird.

Testsuite Eine Sammlung von Testfällen, die zum Testen einer Anwendung verwendet werden sollen, um zu bestätigen, dass diese korrektes Verhalten aufweist.

YAML YAML ist eine vereinfachte Auszeichnungssprache zur Datenserialisierung.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Konzept und Umsetzung eines Tenant Integrators zur Zustandsvorgabe einer Multi-Tenancy in Kubernetes

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original