Hochschule Düsseldorf
University of Applied Sciences

Fachbereich Medien
Faculty of Media

Bachelorarbeit

im Studiengang BSc. Medieninformatik PO 2010

Ein Tutorial über die Erstellung einer Microservicearchitektur in einer Continuous Delivery Umgebung basierend auf Best-Practices

vorgelegt von

Philipp-Sebastian Wolff

02. August 2019

Betreuender Prüfer:	Prof. Dr. Manfred Wojciechowski
Zweitgutachter:	Dipl.-Ing., MSc. Patrick Pogscheba
Matrikelnummer:	670760
Arbeit vorgelegt am:	02. August 2019

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich diese Arbeit - bei einer Gruppenarbeit den entsprechend gekennzeichneten Anteil der Arbeit - eigenständig verfasst und keine anderen als die angegebenen und bei Zitaten kenntlich gemachten Quellen und Hilfsmittel benutzt habe.

_____

Datum, Unterschrift

Zusammenfassung

Die Digitalisierung fordert kontinuierlich neue Services für bestehende Produkte, revolutioniert Produktionsmethoden und verändert Geschäftsmodelle. Continuous Delivery bietet in der agilen Softwareentwicklung mit kurzen Entwicklungszyklen die Antwort auf diesen Wandel. Die Verwendung einer Continuous Delivery-Umgebung (CD-Umgebung) erfordert einerseits zwar einen initialen Einrichtungsaufwand. Andererseits kann durch den Einsatz einer CD-Umgebung das gesamte Projekt an Schnelligkeit gewinnen. In Kombination mit Microservice-Architekturen lassen sich kleine Deployment-Einheiten umsetzen. Microservice-Architekturen bringen einen Mehrwert durch Flexibilität, Skalierung, Automatisierung, standardisierte Schnittstellen und Ausfallsicherheit. Microservices fungieren autark in verteilten Systemen und weisen nur sehr geringe Abhängigkeiten zu anderen Services auf. Die Kommunikation findet über einheitliche Schnittstellen statt, weshalb Implementierungsdetails der Microservices nach außen betrachtet irrelevant sind.

Für die Umsetzung der CD-Umgebung wird der web-basierte Git-Repository-Manager GitLab in Kombination mit der Container-Orchestrierungsplattform Kubernetes verwendet, da sich beide Programme einfach integrieren lassen.

Innerhalb eines Tutorials wird, basierend auf Best Practices anhand des Beispiels der HSD-Card, erläutert wie eine Microservice-Architektur mit Spring Boot, als Anwendung umgesetzt werden kann. Für die Erstellung der Microservice-Architektur legt das Tutorial Schwerpunkte auf die Verwendung von Software Design Pattern (DDD), die Struktur innerhalb Spring Boot, die Kommunikation via REST sowie die Skalierung, das Load-Balancing und die Überführung in die CD-Umgebung.

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Bachelorarbeit unterstützt und motiviert haben. Zuerst gebührt mein Dank Herrn Prof. Wojciechowski sowie Herrn Pogscheba, die meine Bachelorarbeit betreut und begutachtet haben.

Außerdem möchte ich mich bei meinem Bruder Benjamin Wolff, Theresa Opitz und Klaus Albrecht für das Korrekturlesen, für die hilfreichen Anregungen und die konstruktive Kritik bei der Erstellung dieser Arbeit herzlich bedanken.

Inhaltsverzeichnis

1. Einleitung

Agiles Arbeiten wird auch in Konzernen immer mehr zur Normalität, dies gilt auch im Bereich der Softwareentwicklung. Die Digitalisierung bewirkt in vielen Branchen einen Wandel. Sie fordert kontinuierlich neue Services für bestehende Produkte, revolutioniert Produktionsmethoden und verändert Geschäftsmodelle. Klassische Softwareprojekte haben häufig lange Release-Zyklen von mehreren Monaten bis gar zu Jahren. Eine agile Softwareentwicklung und Continuous Delivery (CD) sind die entscheidenden Erfolgsfaktoren um vor dem Hintergrund der Digitalisierung innovativ und produktiv zu bleiben. Entwickler können Konzept-, Entwicklungs-, Testphase und Release mit CD inkrementell und iterativ in kurzen Abständen durchführen. Dies ermöglicht eine schnelle Reaktion auf Veränderung am Markt und auf Änderungen der Ansprüche der Anwender. Innerhalb der CD-Pipeline kann ein hoher Automatisierungsgrad erreicht werden, wodurch es möglich wird schnell qualitativ hochwertige Software auszuliefern. Ein weiterer wichtiger Faktor für die Beschleunigung und Flexibilität eines Systems sind möglichst kleine Deployment-Einheiten. Aus diesem Grund sind Microservices von großer Bedeutung. Im Gegensatz zum Deployment-Monolithen durchlaufen Microservices die CD-Pipeline wesentlich schneller. Tritt beim Deployment ein Fehler auf, ist nicht der gesamte Release-Zyklus unterbrochen, sondern nur ein atomarer Teil.

Erfolgreich werden Microservice-Architekturen beispielsweise beim Volkswagen-Konzern eingesetzt. Der Konzern kann mit dem eigens entwickelten Lizenz-Monitoring-System mit über 200 Lizenzservern rund 50.000 Anwender bedienen. Durch die verwendete Microservice-Architektur kann das System skaliert werden um höhere Lasten, mehr Daten und Anwender zu bewerkstelligen. Die eingesparte Entwicklungszeit kommt direkt dem Produkt zugute. Der Einsatz von CD ist somit ein Erfolgsfaktor für moderne Softwareprojekte. CD erfordert jedoch eine enge Zusammenarbeit zwischen Entwicklungs- und Betriebsteam, sowie das schnelle Feedback durch den Anwender. Zudem muss der initiale Aufwand für die Einrichtung einer CD-Umgebung sowie die Wartung und Betrieb dieses Systems berücksichtig werden. vgl. (Menk, et al., 2017) & (Birk, et al., 2019).

1.1. Motivation

Die Motivation zu dieser Arbeit entstand aus meinem Interesse an einer modernen Softwarearchitektur und der dadurch insbesondere zu erreichenden schnellen Auslieferung von Software an den Endanwender. Zudem möchte ich meine persönlichen Kenntnisse in einem modernen Softwareumfeld erweitern. Die entschiedenen Vorteile von CD und Microservice-Architekturen sind die Flexibilität und der hohe Automatisierungsgrad. Durch das Darlegen von Definitionen, Zusammenhängen und der technischen Umsetzung einer CD-Umgebung möchte ich mit Hilfe eines Tutorials zeigen, wie es möglich ist, basierend auf Best Practices, eine Microservice-Anwendung zu erstellen. Weiter zeige ich, wie eine Anwendung im klassischen Sinne mit Hilfe von Software Design Patterns in Microservices zerlegt und anschließend mit einer CD-Umgebung bereitgestellt wird. Dieses Vorgehen soll durch das Tutorial nachvollzogen werden können und helfen schnellst möglichst eine Anwendung zu veröffentlichen. Ein weiterer Aspekt ist die Verfügbarkeit und die damit verbundene Skalierbarkeit, welche auf Nachfrage, also on demand, gesteigert, aber auch entsprechend vorhandener Ressourcen gesenkt bzw. angepasst werden kann.

1.2. Zielstellung

Vor dem Beginn des Tutorials, werde ich den grundsätzlichen Aufbau einer CD-Umgebung erläutern. Die CD-Umgebung wird in der Regel in einem Unternehmen von DevOps bereitgestellt und gewartet. Ein durchschnittlicher Entwickler hat mit der Konfiguration und Einrichtung meist weniger zu tun. Der Entwickler definiert die CD-Pipeline und achtet darauf, dass seine CD-Pipeline Stages erfolgreich sind. Dennoch ist der Aufbau ein grundlegender Bestandteil, welchen ich vorstellen möchte.

Die exemplarische Umsetzung einer HSD-Card als Anwendung dient als Grundlage für dieses Tutorial. Dazu wird das Anwendungsgebiet erläutert und Erweiterungsmöglichkeiten aufgezeigt.

Außerdem wird die HSD-Card Anwendung in dem Tutorial, anhand von Best Practices und Pattern, in Microservices zerlegt und in die CD-Umgebung überführt. Die Erstellung, die Kommunikation, die Skalierbarkeit, die Verfügbarkeit und das

Load-Balancing der Microservices und letztlich die Überführung in die CD-Umgebung soll in dem Tutorial aufgezeigt und nachvollziehbar werden.

2. Grundlagen

Die Definition eines Microservices und dessen Architektur, sowie die Unterschiede von CD, Continuous Integration (CI) und Continuous Deployment werden in den nachfolgenden Kapiteln näher erläutert.

2.1. Continuous Delivery

Continuous Delivery (CD) wird definiert als kontinuierliche Auslieferung von Software und stellt eine Sammlung von Techniken, Prozessen und Werkzeugen dar, welche die Umsetzung von kurzen Entwicklungszyklen mit häufigen Softwareupdates bis in das Produktivsystem ermöglichen, vgl. (Augusten, 2017).

Die Grundidee von CD ist es eine vom Kunden gewünschte Software instantan abzurufen. Dazu sind einige Arbeitsschritte notwendig. Der Kunde XY hat eine Idee für eine Software und stellt einige Anforderungen. Diese Anforderungen werden in einem Entwicklerteam diskutiert. Es entstehen Rückfragen an den Kunden, bis schließlich für jeden Entwickler einige Aufgaben feststehen. Diese Aufgaben werden in Quellcode umgesetzt. Anschließend werden alle Softwareschnipsel, die aus den einzelnen Aufgaben entstanden sind, zusammengeführt. Es folgt ein automatisierter Build Prozess, gefolgt von anschließenden funktionalen und nichtfunktionalen Tests. Wurden die Schritte erfolgreich durchlaufen, wird die Software in einer Umgebung bereitgestellt.

In Abbildung 1 werden die zuvor beschriebenen Schritte, beginnend mit der Umsetzung in den Quellcode, mit Hilfe einer Continuous Delivery Pipeline abgebildet.

Durch die Implementierung von kleinen automatisierten Tests, welche jeder Teil der Software vor der Freigabe durchläuft, wird die Kompatibilität und Lauffähigkeit der entwickelten Software gewährleistet. Diese zuvor beschriebene Vorgehensweise vereinfacht eine spätere Fehlersuche und minimiert den Aufwand zur Behebung, vgl. (Augusten, 2017).

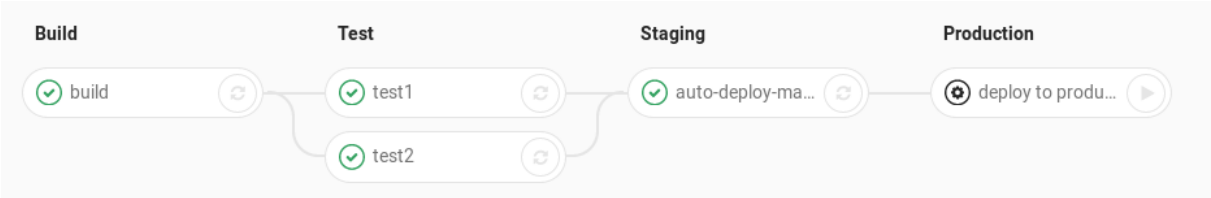


Abbildung 1 CD-Pipeline (Gitlab.com, 2019)

CD ist im eigentlichen Sinn nur einer von drei Begriffen, welcher den automatisierten Prozess der Pipeline beschreibt. Diese drei Begriffe werden im Folgenden erläutert:

- Continuous Integration (CI) umfasst alle Schritte der CD-Pipeline bis auf den des Deployments in das Produktivsystem und ist ein Teil von CD. Diese Vorgehensweise ist eine gängige Softwareentwicklungspraxis, bei dem ein Build der Software erstellt wird und durch automatisierte Tests läuft. Das Deployment in eine Testumgebung kann Bestandteil von CI sein, allerdings niemals das Deployment in ein Produktivsystem.
- Continuous Delivery (CD), wird ausschließlich in dieser Arbeit verwendet, ist ein Software-Engineering Ansatz, bei dem CI, automatisierte Tests und die schnelle Bereitstellung zuverlässig und wiederholt mit minimalem menschlichem Eingriff ausgeführt werden. Die Bereitstellung wird manuell und strategisch durchgeführt.
- Continuous Deployment ist eine Softwareentwicklungspraxis, welche eine sehr ausgereifte CD-Pipeline, mit einer hohen Testabdeckung und stabiler Infrastruktur, voraussetzt. Continuous Deployment enthält alle notwendigen Schritte von CD und erfordert kein menschliches manuelles Eingreifen um Software in das Produktivsystem zu überführen.

Vgl. ((Wolff, 2018 S. 22-24f.) (Gitlab, 2016))

Eine Studie von Perforce (2015) zeigt, dass einige Unternehmen CD Methoden für das Bereitstellen ihrer Produkte verwenden:

> „*The study indicates that Continuous Delivery has really taken off: 65% say their companies have migrated at least one project/team to Continuous Delivery practices.*

> *80% of SaaS companies are doing Continuous Delivery, compared to 51% of non-SaaS companies (like boxed or on-premise software, embedded systems or hardware, industrial goods, etc.)*
>
> *Nearly everyone agrees on the vital role of the collaboration platform (version management, build automation, code review, etc.) in achieving Continuous Delivery. 96% said it's important and 40% said it's critical. No argument here.“* (Perforce, 2014)

2.2. Microservice-Architektur

Nicht nur herkömmliche Softwarearchitekturen können von CD profitieren, sondern insbesondere Microservice-Architekturen, die überwiegend Gegenstand dieser Ausarbeitung sind. Microservice-Architekturen haben den entscheidenden Vorteil, besonders autark zu fungieren. Sie weisen im Gegensatz zu anderen Softwarearchitekturen nur sehr geringe Abhängigkeiten zu anderen Services auf, vgl. ((Wolff, 2018), (Calcott, 2018)).

Eine Anwendung besteht meistens aus vielen zusammenhängenden Funktionen, welche in Abhängigkeit zueinanderstehen und modularisiert sind. Erst beim Build Prozess ergibt sich eine vollständige Anwendung, welche getestet werden kann, siehe Abbildung 2.

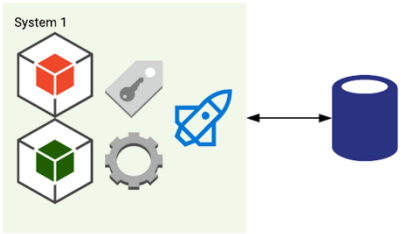


Abbildung 2 Monolithische Anwendung (Weerasinghe, 2018)

In einer Microservice-Architektur werden diese Funktionalitäten, durch später beschriebene Pattern und Methoden, in mehrere autark funktionierende Microservices geschnitten, siehe Abbildung 3. Nach Möglichkeit soll ein Microservice nur eine Aufgabe erfüllen und so klein wie möglich sein. Ein Microservice zeichnet sich durch seine geringe Größe und leichte Ersetzbarkeit aus. Microservices können in verschiedenen Programmiersprachen erstellt werden, da die Kommunikation zu anderen Microservices über eine standardisierte Schnittstelle

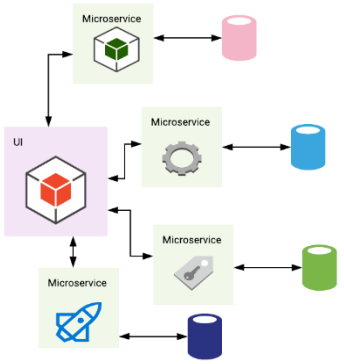


Abbildung 3 Microservice Anwendung (Weerasinghe, 2018)

wie z.B. REST stattfindet. Microservices verfolgen einen dezentralen Ansatz und sind in der Lage, physikalisch oder virtuell voneinander getrennt, miteinander interagieren.

E. Wolff sieht Microservices als ein Modularisierungskonzept, welches dazu dient, große Softwaresysteme aufzuteilen und welches die Organisation und Entwicklungsprozesse beeinflusst. Daraus ergeben sich drei Unterteilungen:

- **Stil/Konzept:** Ein Konzept für die Modularisierung von Software, das Wert auf eine bestimmte Organisation und Vorgehensweise legt.
- **Komponente:** Der einzelne Microservice, der eigenständig existiert und seine fachlich orientierte Aufgabe erledigt.
- **Architektur:** Microservice-Architektur entsteht als großes Bild über alle einzelnen Microservices hinweg.

Vgl. ((Wolff, 2018 S. 3 f.), (Wolff, 2016 S. 10 f.))

2.3. Zusammenwirken Continuous Delivery und Microservices

In einer CD-Umgebung wird jeder Microservice als eigenständiges Projekt mit einer CD-Pipeline betrachtet. Dementsprechend hat jeder Microservice eigene Tests, mit denen Funktionen geprüft werden. Um ein gutes Zusammenarbeiten der einzelnen Microservices in der Anwendung zu gewährleisten, empfiehlt es sich alle Microservices möglichst gleich zu implementieren. Damit ist der Aufwand für die Anpassung der CD-Pipeline geringer. Die CD-Pipeline lässt sich durch die Verwendung von Umgebungsvariablen leichter wiederverwerten.

3. Konzeption der Continuous Delivery-Umgebung

Das nachfolgende Kapitel verschafft einen Überblick über die Mittel und Wege, die notwendig sind, um eine CD-Umgebung aufzubauen. Dabei werden zunächst das grundlegende Prinzip einer Softwareversionsverwaltung dargelegt und die wichtigsten Praktiken erläutert. Im nächsten Schritt werden die Verwendung der Softwareversionsverwaltung und die aus der Verwendung hervorgehenden Eigenschaften in einem konkreten Beispiel erläutert.

Eine CD-Umgebung besteht aus mehreren Programmen und kann sehr vielseitig aufgebaut werden. Durch diesen Umstand kann die Komplexität sehr hoch sein.

Daher kann es unter Umständen mehrere Jahre dauern, bis sich ein Entwickler in Gänze auskennt. Nachfolgend werden Definitionen, Prozesse und Konzepte der am Markt üblicherweise in Kombination verwendeten Programme für eine CD-Umgebung vorgestellt, vgl. (Buehrle, 2018).

In diesem Kapitel grau hinterlegte Befehle werden auf der Konsole und im Browser ausgeführt.

3.1. Softwareversionsverwaltung

Eine Softwareversionsverwaltung (VCS engl. Version Control System) ist ein System zur Versionierung von Dateien. Durch das Erstellen und Verwalten verschiedener Versionen einer Datei sind Fehler einfacher rückgängig zu machen und Änderungen besser nachvollziehbar. Jeder Entwickler erhält mit seinem individuellen Benutzeraccount den Zugriff zu diesem System. Jede Version einer Datei wird mit einem Zeitstempel und einem Kommentar versehen. Gespeichert werden nur die Änderungen um Speicherplatz zu sparen. Das Herzstück eines VCS ist das Repository. Dieses dient als Archiv. In dem Archiv werden alle Änderungen samt den dazugehörigen Meta-Daten hinterlegt. Die Entwicklung findet statt indem jeder Entwickler einen Entwicklungszweig (Branch) des Hauptprojekts (Master) erstellt. Dort werden die Änderungen der entwickelten Dateien eingestellt. Bei Fertigstellung des gewünschten Ergebnisses wird dieser Entwicklungszweig wieder in das Hauptprojekt überführt. Diese Vorgehensweise ermöglicht ein koordiniertes und effizientes Arbeiten von mehreren Entwicklern zeitgleich an einem Projekt. Durch die Protokollierung der Dateien, lassen sich Änderungen bei der Zusammenführung (Merge) in den Master von anderen Entwicklern besser nachvollziehen. Die Verwendung eines VCS schafft eine Transparenz über dazukommende Funktionen oder Dateien eines Entwicklungszweiges.

Um den Verlust von Informationen durch paralleles Arbeiten von mehreren Entwicklern durch z.B. Überschreiben zu verhindern, gibt es zwei unterschiedliche Ansätze im Aufbau eines Repositories des VCS, siehe Abbildung 4, vgl. (Blöchinger, 2019).

- Bei der zentralen Versionsverwaltung, wie z.B. Subversion (SVN), werden alle Dateien über ein zentrales Repository auf einem Server verwaltet. Jeder

Benutzer hat neben dem Zugriff auf dieses Repository eine lokale Arbeitskopie (Working Directory). Diese beinhaltet allerdings nur die lokalen Änderungen. Eine vollständige Änderungshistorie und die neuste Version des Projektes sind nur in dem Repository vorhanden.

- Bei der dezentralen Versionsverwaltung, wie z.B. bei Git, hat jeder Benutzer neben der lokalen Arbeitskopie ein eigenes lokales Repository mit der kompletten Änderungshistorie. Änderungen in anderen Repositorys sind jedoch nicht direkt ersichtlich und müssen vorab synchronisiert werden (Pull),vgl. (Blöchinger, 2019).

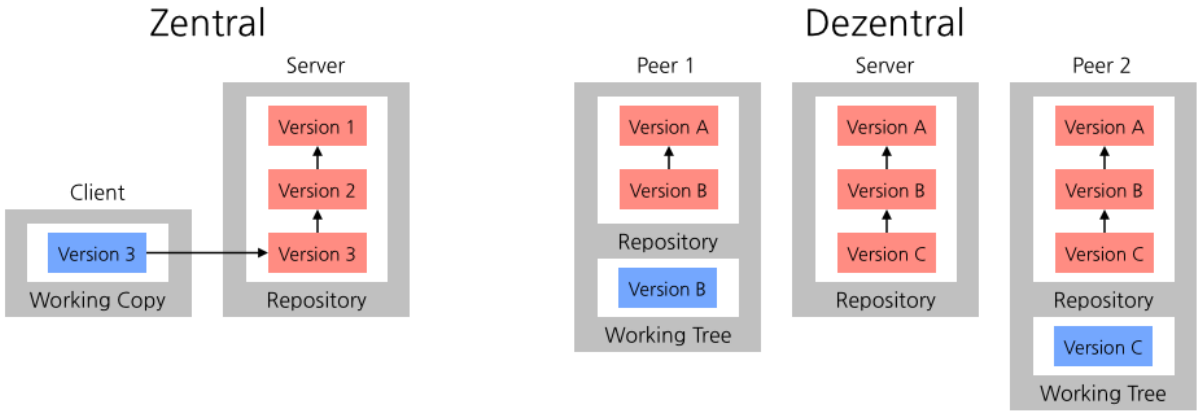


Abbildung 4 Unterschied zentrales/dezentrales Repository (Helmich, 2018)

Bei der Arbeitsweise eines VCS gibt es zwei verschiedene Konzepte:

- Lock Modify Write: In dieser Arbeitsweise, auch pessimistische Versionsverwaltung genannt, wird die zu bearbeitende Datei für andere Zugriffe gesperrt und erst dann wieder für alle freigegeben, wenn eine erfolgreiche Änderung durchgeführt wurde.
- Copy Modify Merge: Diese Arbeitsweise, auch optimistische Versionsverwaltung genannt, lässt eine gleichzeitige Bearbeitung einer Datei zu, indem eine lokale Arbeitskopie erstellt wird. Anschließend wird diese automatisch oder manuell (Merge) mit der Originaldatei zusammengeführt. Problematisch wird es bei Binärdateien, da diese ohne weitere Hilfsmittel nicht zusammengeführt werden können. Alle modernen de- oder zentralen VCS verwenden Copy Modify Merge.

Vgl. (Wikipedia, 2019).

In dieser Arbeit wurde als Grundlage das Programm Git als VCS verwendet. Ein häufig verwendeter Workflow ist der Gitflow, siehe Abbildung 5. Dieser funktioniert wie folgt:

Der Master-Branch des Projekts beinhaltet die gesamte Änderungshistorie und jeder „Commit“ erhält als „Tag“ die Versionsnummer. Der Stand des Master-Branches ist der Stand, welcher im Produktivsystem zu sehen ist.

Für die Entwicklung wird ein Develop-Branch erstellt. Dieser dient als Integrations-Branch für Feature Branches. Jeder Entwickler zweigt also für seine Aufgabe von dem Develop- als Parent-Branch ab und entwickelt für sich in einem Feature-Branch. Anschließend wird das fertig entwickelte Feature wieder mit dem Develop-Branch zusammengeführt (Merge). Ein Feature-Branch interagiert niemals direkt mit dem Master-Branch.

Haben sich eine Reihe von fertigen Features im Develop-Branch angesammelt, findet ein sogenannter „Fork“ statt. Dies bedeutet, dass von dem Develop-Branch ein Release-Branch abgezweigt wird. Die fortlaufende Entwickelung kann weiter auf dem Develop-Branch vorgenommen werden, während in dem Release-Branch weitergearbeitet werden kann. Ergänzende Dokumentation, Bug Fixes oder Ähnliches werden in einem Release-Branch hinzugefügt, bevor dieser zum nächsten Rollout Termin mit dem Master-Branch zusammengeführt wird.

Sollte nach einem Rollout festgestellt werden, dass es Fehler oder Probleme mit dem Produktivsystem gibt, wird ein sogenannter Hotfix-Branch erstellt. Dieser Branch wird direkt vom Master-Branch abgezweigt. Ein Hotfix-Branch dient als Wartungszweig, indem der Entwickler das Problem behebt oder einen vorherigen Stand des Master-Branches wiederherstellt. Anschließend wird der Hotfix-Branch in den Master- und Develop-Branch überführt, vgl. (Atlassian, 2019).

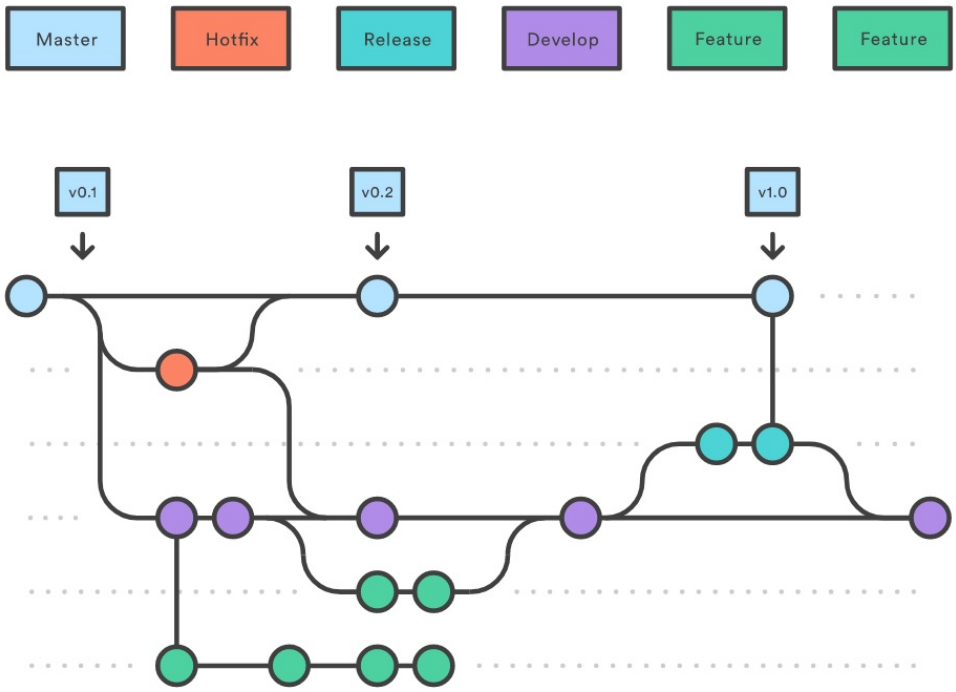


Abbildung 5 Gitflow (Atlassian, 2019)

3.2. Continuous Delivery-Umgebung

Die CD-Umgebung ist eines der Kernstücke dieser Bachelorarbeit und wird nachfolgend in Worten, sowie mit der Abbildung 6 beschrieben. Das Grundprinzip von CD wurde bereits in Kapitel 2 ff. beschrieben. Für die technische Umsetzung werden die Programme GitLab, Docker und Kubernetes verwendet.

Der Arbeitsablauf der CD-Umgebung ist wie folgt:

1. Der Quellcode gelangt gemäß dem Gitflow durch einen Commit (lokales Repository) und Push (zentrales Repository) in das Git-Repository von GitLab. Es kann jede Entwicklungsumgebung Verwendung finden, die Git unterstützt. Im Rahmen dieser Bachelorarbeit wurde IntelliJ verwendet.
2. GitLab sucht nach der gitlab-ci.yml im Root Verzeichnis des Projektes. Anhand dieser Konfiguration wird die CD-Pipeline aufgebaut und ausgeführt. Mit Hilfe des Dockerfiles wird ein Docker Image erstellt, welches durch die Definition eines Kubernetes Manifests der deployment.yml in Kubernetes bereitgestellt wird.
3. Beim erfolgreichen Ausführen der CD-Pipeline ist das Projekt über eine URL (vorausgesetzt der Discovery Service Kubernetes Ingress ist implementiert) im Kubernetes Cluster erreichbar und kann getestet werden.

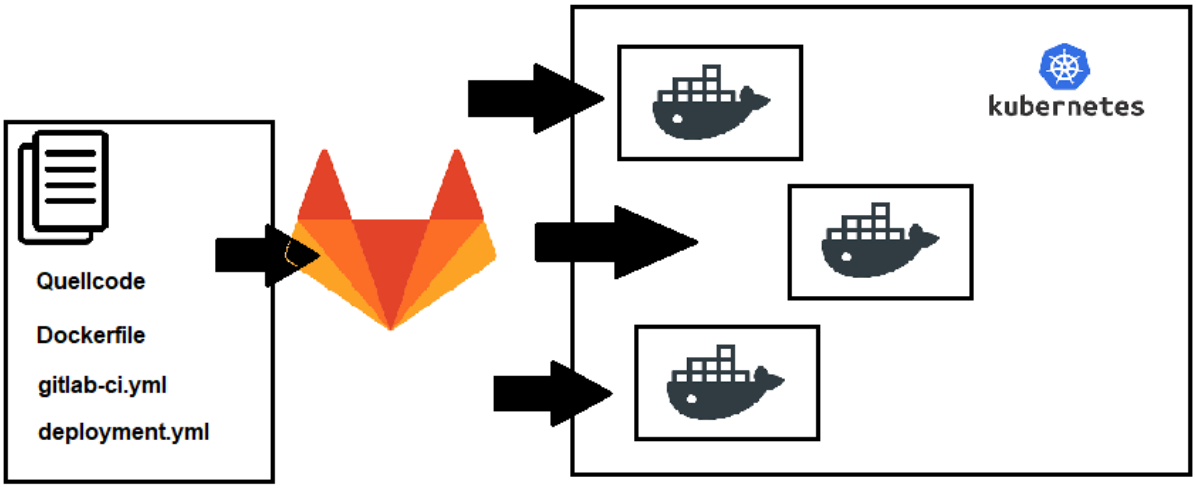


Abbildung 6 Aufbau CD-Umgebung

3.2.1. GitLab

GitLab ist ein web-basierter Open-Source Git-Repository-Manager und wurde in der Community Edition mit einer MIT-Lizenz ausgegeben.

GitLab wird in dieser Arbeit mit Hilfe einer Docker-Compose Datei, siehe Anhang B.16, definiert und direkt auf dem Host-System und mit „*docker-compose up*“ gestartet. Mit Docker-Compose ist es möglich gleich mehrere Docker Container, welche für GitLab notwendig sind zu starten, siehe auch (Wolff, 2018 S. 39 f. Kap. 4.1.5).

Im eigentlichen GitLab Container werden Parameter konfiguriert wie z.B. die URL oder das „Mounten“ (engl. anbringen) von „Volumes“ (Speicherressource) um z.B. die Daten der Projekte auf dem Host zu speichern. Nach dem Starten kann mittels „*docker ps*“ eine Übersicht der gestarteten Container angezeigt werden. Die Weboberfläche ist über https://gitlab.ba-hsd.de:30080 erreichbar. Beim erstmaligen Starten von GitLab muss das Administrator-Konto erstellt werden. Dieses verfügt über die gesamte Bandbreite an Berechtigungen, mit denen GitLab vollständig als Teil der CD-Umgebung konfiguriert werden kann.

Wie bereits in Abschnitt 3.2 kurz erwähnt, werden GitLab CI / CD- Pipelines mithilfe einer YAML-Datei, *siehe Anhang B.1* konfiguriert und in jedem Projekt aufgerufen. Beschreibung des vollständigen Aufbaus der CD-Pipeline folgt in 3.2.2.

Die dort definierten Schritte auszuführen ist Aufgabe des Gitlab-Runners (kurz Runners). Der GitLab-Runner ist nicht standardmäßig in GitLab enthalten und muss separat definiert werden. Ein Runner ist isoliert und kann eine virtuelle Maschine, einen Virtual Private Server (VPS), eine physikalische Maschine, einen Docker Container oder sogar ein Cluster von Container sein. GitLab und der Runner kommunizieren über eine API. Deshalb ist es wichtig, dass der Runner über eine Netzwerkverbindung zum GitLab-Server verfügt. Ein Runner kann einem bestimmten Projekt zugeordnet werden oder als Shared Runner für alle Projekte verfügbar sein, vgl. (GitLab, 2019). Einem Runner können Tags zugeordnet werden z.B. docker, auf welchen der Runner ausgelöst wird. Ist in der CD-Pipeline derselbe Tag für eine Stage definiert, führt der Runner diese Stage aus.

Das Menü auf der linken Seite von GitLab, siehe Abbildung 7, bietet eine Vielzahl von Funktionen zur Verwaltung des Projekts.

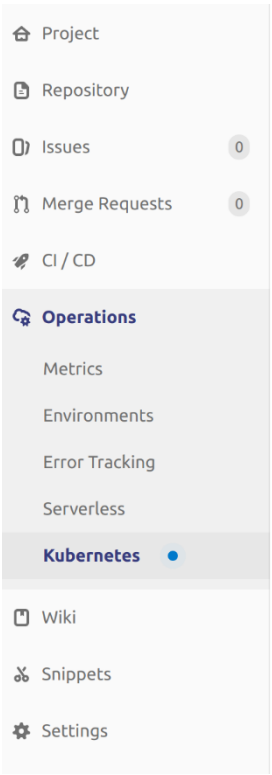


Abbildung 7 GitLab Projekt Menü

1. **Project:** Zeigt allgemeine Optionen zum Projekt.
2. **Repository:** Zeigt Informationen zu Dateien, Branches, sowie die komplette Änderungshistorie. Auch einzelne Dateien können erstellt oder verändert werden.
3. **Issues:** Anzeige und Erstellung von Anforderungen
4. **Merge Requests:** Zeigt Änderungen und Ergänzungen des Projekts für ein fertiges Feature an. Diese Änderungen können in einem Code Review von anderen Benutzern kommentiert werden. Sind keine Beanstandungen vorhanden, wird der Merge Request genehmigt und der Entwickler oder andere Entwickler dieses Projekts berechtigen den Branch, die Änderungen oder Ergänzungen in den Develop-Branch einzufügen. Dies erhöht die Qualität des Codes und wird in Unternehmen oft praktiziert.
5. **CI / CD:** Zeigt alle Jobs der Pipeline an, siehe 3.2.2.
6. **Operations:** Dient zum Konfigurieren von zusätzlichen Diensten, wie z.B. Kubernetes.
7. **Wiki:** Erklärungen und Dokumentationen zum Projekt werden hier abgelegt.

8. **Snippets:** Zeigt Code- und Textteile, die mit anderen Benutzern geteilt werden können.
9. **Settings:** Zeigt allgemeine Einstellungen zum Projekt.

Speziell für die Integration mit Kubernetes hat GitLab ein paar Funktionen implementiert. Dies ermöglicht es, ein Kubernetes Cluster direkt mit GitLab zu verbinden und einen Runner in dem Cluster zu installieren. Benötigt werden dazu folgende Informationen aus dem Kubernetes Cluster, siehe Abbildung 8:

1. **Kubernetes cluster name:** Kann frei gewählt werden.
2. **API URL:** Zeigt die IP-Adresse samt Port des Master Nodes
3. **CA Certificate:** Dient zur Authentifizierung mit dem Kubernetes Cluster
4. **Token:** Dient zur Autorisierung
5. **Project namespace:** Optional kann Namespace in Kubernetes angelegt werden.

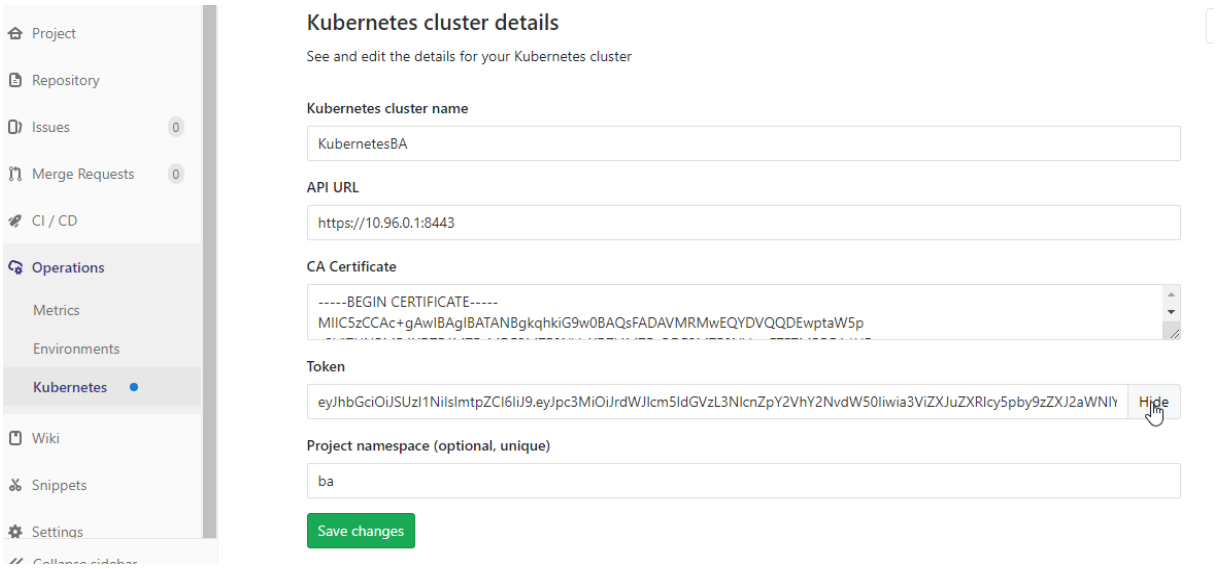


Abbildung 8 Verbindung zwischen GitLab und Kubernetes aufbauen

Wie der Zugriff zu diesen Informationen erhalten werden kann, wird in Abschnitt 3.2.4 gezeigt.

Konnte sich GitLab erfolgreich mit Kubernetes verbinden, können nun weitere Dienste wie Helm Tiller (Kubernetes Packet Manager, zwingend zum Installieren aller anderen Funktionen benötigt), Ingress, sowie der GitLab Runner direkt über

die Weboberfläche auf dem Kubernetes Cluster installiert werden, siehe Abbildung 9.

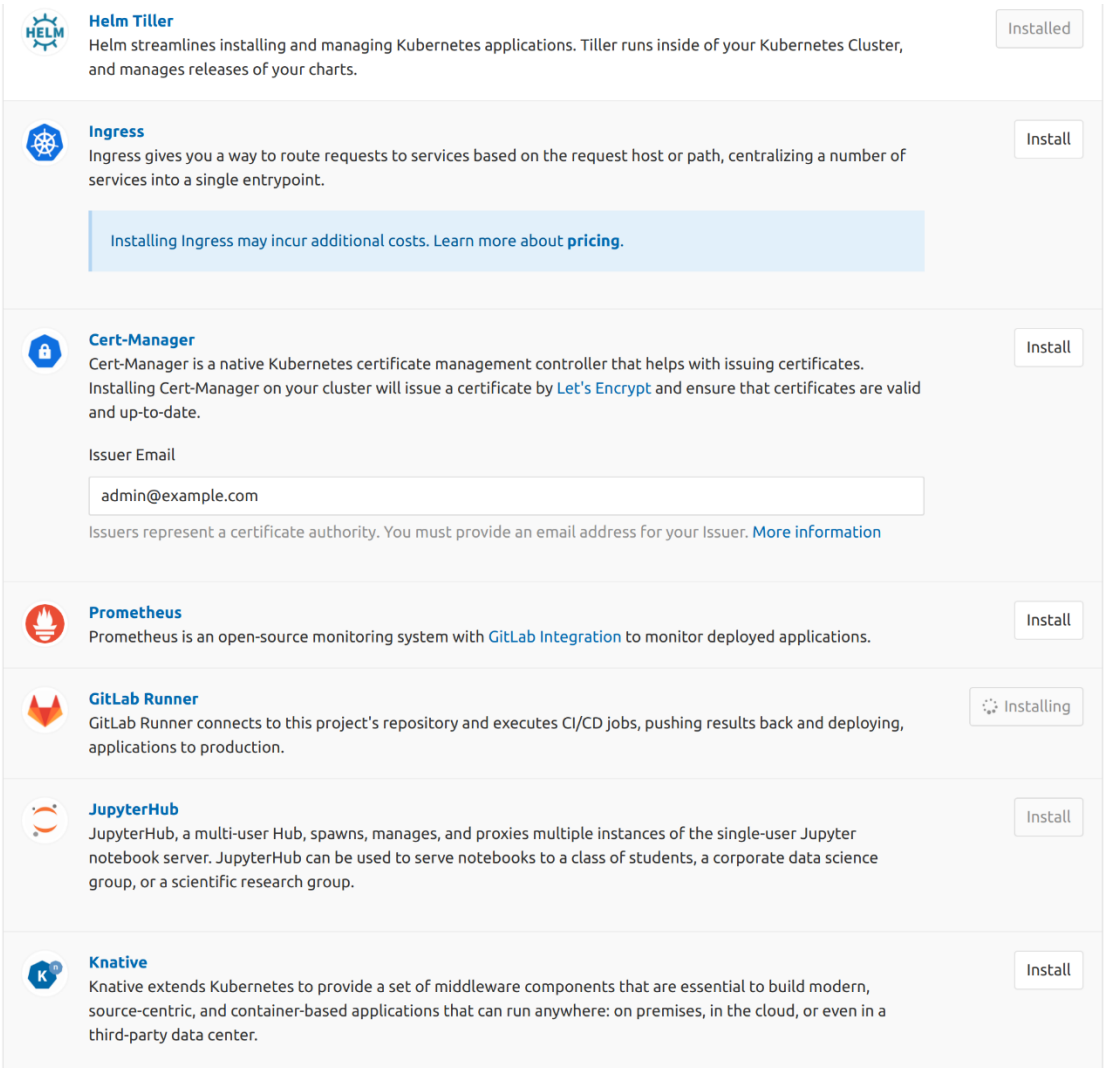


Abbildung 9 Installieren des GitLab-Runners auf dem Kubernetes Cluster

3.2.2. CD-Pipeline

Die CD-Pipeline wird in der gitlab-ci.yml, siehe Anhang B.1, definiert und muss im Stammverzeichnis des Projektes liegen. Durch jeden Commit bzw. Push wird ein neuer Job erstellt, welcher dann vom Gitlab-Runner ausgeführt wird. Unter dem Menüpunkt CI / CD können alle Jobs anzeigt werden. Die einzelnen Schritte werden in GitLab als Stages bezeichnet und werden in der gitlab-ci.yml vorab aufgezählt und entsprechend der Reihenfolge definiert. Die populärsten Stages, welche nicht fehlen sollten sind Build, Test und Deploy. Ergänzend dazu gibt es noch reservierte Schlüsselwörter, welche die Stages an Bedingungen knüpfen kann, wie z.B. das manuelle Deployment, entsprechend der Idee von CD. Stages gleicher Art wie z.B.

Tests werden parallel ausgeführt. Schlägt eine Stage oder Teile dieser fehl, wird der Job beendet. Die nächste Stage wird nur ausgeführt, wenn die vorherige Stage komplett erfolgreich war, siehe Abbildung 10.

Jede Stage ist mit einem Icon versehen, welches den aktuellen Status dieses Schritts anzeigt, siehe Tabelle 1.

Icon	Bedeutung
	Stage wird ausgeführt
	Stage erfolgreich durchlaufen
	Stage teilweise erfolgreich durchlaufen
	Stage nicht erfolgreich durchlaufen, Job wurde abgebrochen
	Job wartet auf die Ausführung
	Job erneut ausführen
	Manuell deployen
	Job wurde abgebrochen
	Stage konnte nicht erreicht werden, da der Jobs in der vorherigen Stage fehlgeschlagen ist.
	Laufenden Job abbrechen

Tabelle 1 Bedeutung der Icons der CD-Pipeline

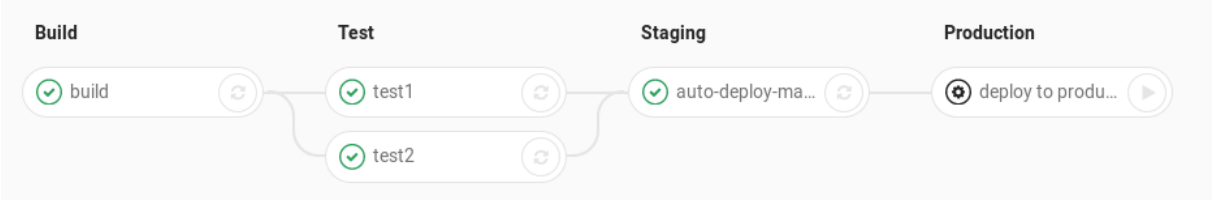


Abbildung 10 CD-Pipeline in GitLab

Während und nach einem durchlaufenden Pipeline-Schritts kann die Konsolenausgabe des GitLab-Runners durch einfaches Klicken auf das Icon beobachtet werden, siehe Abbildung 11.

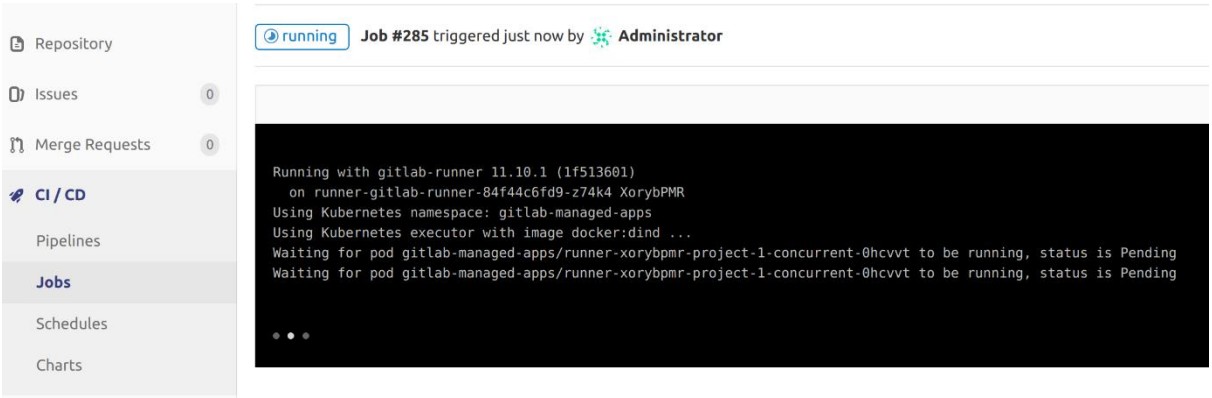


Abbildung 11 Konsolenausgabe eines laufenden Jobs der CD-Pipeline

Vgl. (GitLab, 2019)

GitLab bietet „out of the box“ die Verwendung von vordefinierten Umgebungsvariablen an. Eine Umgebungsvariable ist ein dynamisch benannter Wert, der das Verhalten von Prozessen auf einem Betriebssystem beeinflussen kann. So ist es z.B. möglich den Projektnamen, den Namen des Branches oder viele andere Werte in die lokale Umgebung des GitLab-Runners generisch einfließen zulassen. Zusätzlich besteht die Möglichkeit selbst Variablen zu definieren unter Administrator > Projektname > CI / CD Settings, vgl. (GitLab, 2019)

Die gitlab-ci.yml für diese Arbeit kann im Anhang B.1 eingesehen werden.

3.2.3. Docker

Eine leichtgewichtige Alternative zur kompletten Virtualisierung des Projekts bietet Docker. In Docker Containern, können die Ressourcen des Hosts optimal genutzt und schnell auf einem Node im Kubernetes Cluster gestartet werden. Die dafür notwendige Konfiguration wird in einem Dockerfile hinterlegt. Das Dockerfile befindet sich im Root Verzeichnis des Projektes, vgl. (Wolff, 2018 S. 38 ff. Kap. 4.1.5)

Ein Dockerfile ist ein Textdokument, welches automatisch zum Erstellen eines Docker Images von dem Docker Deamon ausgeführt wird. Bei dem Befehl `docker build`, innerhalb der CD-Pipeline, wird das Docker Image aus dem Dockerfile und dem Buildkontext, also dem lokalen Verzeichnis des vorherigen Buildjobs erstellt. Beim `docker build` wird der Kontext rekursiv abgearbeitet und an den Docker Deamon gesendet, vgl. (Docker, 2019)

Im CD-Pipeline Schritt Staging passiert Folgendes, siehe auch Anhang A.1:

1. docker info – Gibt Details zum Docker Deamon auf dem GitLab-Runner aus.
2. docker login – Zugang zur externen Docker Registry.
3. docker build – Der Docker Deamon arbeitet wie oben beschrieben die Schritte im Dockerfile ab.
    - Herunterladen einer Laufzeitumgebung.
    - Angabe der aus dem Buildjob entstandenen Datei über eine Umgebungsvariable.
    - Ebenso wird das Home Verzeichnis über Umgebungsvariablen gesetzt.
    - Setzen des Ports über den der Container erreichbar sein soll.
    - Setzen des Home Verzeichnisses.
    - Kopieren der aus dem Buildjob entstandenen Datei in das Home Verzeichnis.
    - Ausführen dieser Datei mit der Laufzeitumgebung.
4. docker –t tagname Docker Image wird getagged.
5. docker push Docker Image wird in die externe Docker Registry abgelegt.

Die Docker Registry ist eine zustandslose, stark skalierbare serverseitige Anwendung, mit der Docker-Images gespeichert und verteilt werden können. Die Registry ist Open Source und unterliegt der Apache-Lizenz. Für die Umsetzung in dieser Arbeit wurden viele Ansätze der Umsetzung verfolgt, welche im Kapitel 5.2 näher beschrieben werden. Letztlich wurde die Docker Registry auf einen externen Server mit einem „Lets Encrypt“-Zertifikat einer Domain ausgelagert. Für das Abrufen des Docker Images aus der Docker Registry innerhalb der Pipeline, bzw. vom GitLab-Runner wurden in der Konfiguration von Kubernetes die DNS-Einstellungen angepasst.

Details sind aus dem Dockerfile und der gitlab-ci.yml für diese Arbeit zu entnehmen, siehe im Anhang B.1 & B.2.

3.2.4. Kubernetes

Kubernetes ist eine Open-Source-Plattform, die den Betrieb von Linux-Containern automatisiert. In einigen Quellen, darunter (redhat, 2019), wird Kubernetes auch als

k8s (beginnt mit „k“, darauf folgen 8 Zeichen und endet mit „s“) oder „kube“ abgekürzt. Kubernetes ermöglicht es Gruppen von Hosts, die auf dem Linux-Container laufen, in einem Cluster zusammenzufassen. Diese Cluster können Hosts aus Public-, Private- oder Hybrid-Clouds haben und werden effizient über Kubernetes verwaltet. Kubernetes wurde von Google entwickelt und konzipiert und resultiert aus Googles Vorgängerplattform Borg.

Die meisten Microservices-Anwendungen erstrecken sich über mehrere Container. Einzelne Microservices werden in der Regel auf mehreren Servern bereitgestellt. Kubernetes liefert die Orchestrierungs- und Managementfunktionen, die für die Container-Bereitstellung notwendig sind. Durch die Kubernetes-Orchestrierung lassen sich Anwendungen, die sich über mehrere Container erstrecken, in einem Cluster planen und skalieren. Zudem kann der Zustand dieser Anwendungen langfristig überwacht werden. Mit Kubernetes lässt sich eine Infrastruktur aufbauen, welche komplett Container-basiert ist und das Ziel verfolgt operative Aufgaben zu automatisieren. Bei klassischen Anwendungen wird dieser Zweck (Monitoring, Ressourcenadministration etc.) durch andere Anwendungsplattformen erreicht.

Kubernetes wurde für diese Arbeit komplett auf einem Host neben GitLab realisiert. Üblich ist es, das Kubernetes Cluster auf mehrere physikalische oder virtuelle Maschinen aufzuteilen. Um diesem Cloud-Native Ansatz zu folgen, wurden der Kubernetes Master und Node jeweils in einer virtuellen Maschine auf dem Host realisiert, siehe Abbildung 25.

Nachfolgend sind die Funktionsweisen der einzelnen Bestandteile eines Kubernetes Clusters und die wichtigsten Begriffe erklärt:

- **Kubernetes Master:** Die Maschine, die alle Kubernetes Nodes kontrolliert und über die Kommandos und Aufgaben ausgeführt und verteilt werden.
- **Kubernetes Node:** Die Maschine, die vom Kubernetes Master kontrolliert wird und die abzuarbeitenden Anweisungen über die API erhält.
- **Pod:** Ein Zusammenschluss aus einem oder mehreren Containern und die kleinste administrative Einheit, innerhalb des Clusters. Alle Container in einem Pod teilen sich eine IP-Adresse, einen Hostname und weitere Ressourcen. Durch die Pods lassen sich Netzwerk und Speicherressourcen

vom zugrundeliegenden Container abstrahieren. Dies ermöglicht es Container einfacher innerhalb eines Clusters oder von einem Cluster in ein Cluster zu verschieben.

- **Replication Controller:** Dieses Tools kontrolliert, dass identische Kopien eines Pods auf dem Cluster laufen. Diese Funktion wird über ein Kubernetes Manifest vom Typ Replication Controller definiert und für die Skalierung gebraucht.
- **Service:** Pods und Arbeitsdefinitionen werden dadurch entkoppelt. Ein Kubernetes Service wird ebenfalls über ein Manifest definiert und leitet Serviceanfragen automatisch an den richtigen Pod. Ein solcher Service wurde z.B. für den GitLab-Runner, welcher ebenfalls in einem Pod läuft im Cluster integriert.
- **Kubelet:** Dieser Dienst läuft auf dem Node. Er liest die Container-Manifeste aus und startet daraufhin den Container basierend unter Verwendung der Manifest Konfiguration.
- **Kubectl:** Ist ein Befehlszeilen-Kommando Tool für Kubernetes.

Wie bereits oben erwähnt, setzen Kubernetes Master und Node jeweils auf einer virtuellen Maschine mit Linux-Betriebssystem auf. Das Betriebssystem interagiert mit den Containern und den Pods. Dafür muss auf der physikalischen oder virtuellen Maschine Docker installiert sein. Über den Kubernetes Master werden von den Administratoren (Dev-Ops) Befehle abgesetzt. Der Kubernetes Master leitet diese Befehle an die untergeordneten Nodes weiter. Eine Vielzahl von Services entscheidet automatisch, welcher Node am besten für diese Aufgabe geeignet ist und führt diese aus. In der vorliegenden Arbeit gibt es nur einen Node, aus diesem Grund wird immer an diesen weitergeleitet. Wenn Kubernetes einen Pod auf einem Node startet, wird über kubelet Docker die Anweisung zum Starten eines Containers übermittelt. Außerdem übermittelt kubelet den Status und andere Informationen des Containers an den Kubernetes Master, siehe Abbildung 12. Anstatt diese Informationen manuell einzuholen, werden sie vom System automatisch von Docker abgefragt, vgl. (redhat, 2019).

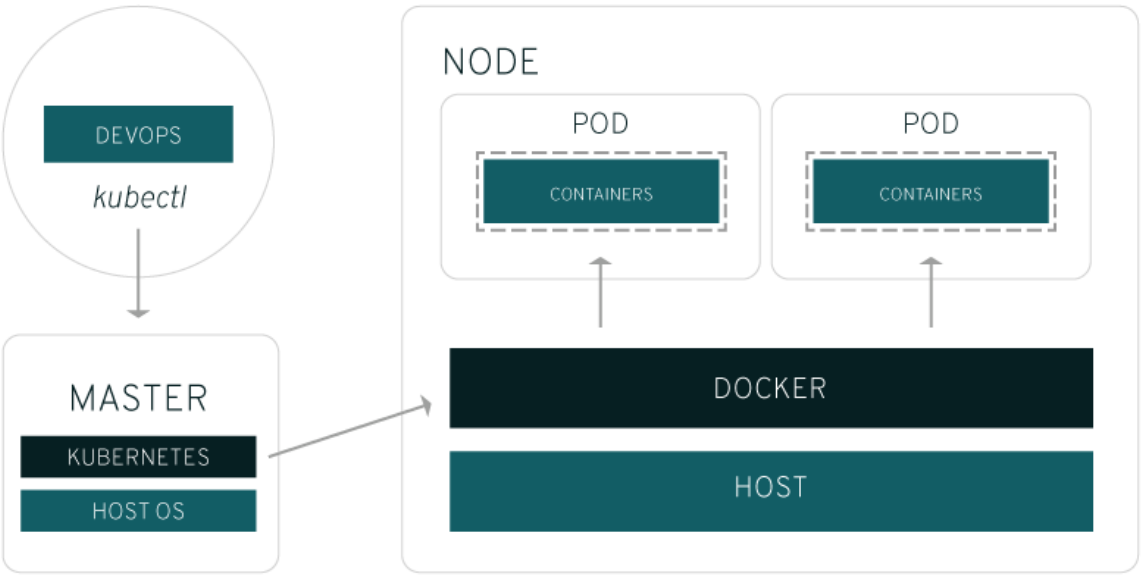


Abbildung 12 Kubernetes Infrastruktur (redhat, 2019)

Neben dem Erstellen der virtuellen Maschinen und einigen Vorbereitungen zum Installieren eines Kubernetes Clusters, ist der wichtigste Befehl zum Erstellen des Kubernetes Masters der Folgende: `sudo kubeadm init --apiserver-advertise-address=192.168.XXX.XXX --pod-network-cidr=10.244.0.0/16`. Dieser Befehl gibt an unter welcher IP-Adresse der Master Node erreichbar ist und welche IP-Range als Pod-Network genutzt werden soll.

In der Arbeit wurde als Software-Defined-Network(SDN) Flannel benutzt, welches die IP-Range vorgibt. Flannel ist ein sehr simples Overlay-Netzwerk, welches die Anforderung von Kubernetes erfüllt, vgl. (kubernetes, 2019).

Ist der Kubernetes Master mit allen Services installiert, wird der Befehl zum Hinzufügen der Nodes in das Cluster eingeblendet. Dieser Befehl soll auf den virtuellen Maschinen, die als Node fungieren, ausgeführt werden, siehe Abbildung 13, vgl. (Ellis, 2017).

```
....
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy

Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 192.168.25.110:6443 --token 2hcl3s.tc9nufjeewlos0nk \
    --discovery-token-ca-cert-hash sha256:252928d861475d389ad4fdb790411403da872f1048b3900fc5514f9c0cd2145e
```

Abbildung 13 Erfolgreiche Installation des Kubernetes Master mit Anweisung zum Hinzufügen von Kubernetes Nodes

Die Integration von GitLab in Kubernetes wurde im Abschnitt 3.2.1 aus der Sicht von GitLab bereits gezeigt. Nun werden die Schritte gezeigt, welche erforderlich sind um die notwendigen Informationen in Kubernetes zu erhalten und GitLab erfolgreich mit Kubernetes zu verbinden, siehe Anhang A.2.

- Erstellen eines Service Accounts für GitLab um GitLab zu berechtigen als Cluster-Admin Aktionen auszuführen, siehe Anhang B.17.
- Ausführen der Service Account Definition mit „kubectl apply -f gitlab-admin-service-account.yaml“ auf dem Kubernetes Master.
- Die IP-Adresse des Kubernetes Masters und weitere Informationen anzeigen: „*kubectl cluster-info*“.
- Das Token für den Gitlab-Admin-Service-Account einsehen: „*kubectl -n kube-system describe secret $(kubectl -n kube-system get secret | grep gitlab-admin | awk '{print $1}')*“
- Um das Kubernetes CA-Zertifikat ausfindig zu machen, sollte zunächst die Übersicht der Secrets ausgeben werden: „*kubectl get secrets*“.
- Das CA-Zertifikat befindet sich im default-token-XXXXX, hier: „*default-token-g6kdt*“. Das Zertifikat ist base64-codiert und muss im JSON-Format hinterlegten Secret gefunden und decodiert werden. „*kubectl get secret default-token-g6kdt -o jsonpath="{['data']['ca\.crt']}" | base64 –decode*“

Durch das Verbinden von GitLab und Kubernetes und die Installation von Helm Tiller (Kubernetes Paketmanager) wird in Kubernetes der Namespace „gitlab-managed-apps“ erstellt. In diesem befinden sich die Pods für z.B. den GitLab-Runner, Helm

Tiller und weitere Apps, die über die Oberfläche von GitLab hinzugefügt wurden, vgl. (GitLab, 2019).

Das Kubernetes Dashboard dient dem Monitoring und wird mit folgendem Befehl installiert:

„*kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/master/aio/deploy/recommended/kubernetes-dashboard.yaml*“

Diese Definition des Kubernetes Manifests für das Dashboard beinhaltet:

- **Secret:** Definition des Tokens für das Dashboard.
- **Service Account:** Definiert unteranderem, in welchem Namespace das Dashboard liegt.
- **Role Bindung:** Definition von „Rules“ und Berechtigungen.
- **Deployment:** Definition zum Starten des Containers inkl. Ressourcen etc.
- **Service:** Definition der Erreichbarkeit z.B. der Ports.

Wurde das Manifest erfolgreich ausgeführt, kann mit dem Befehl: „kubectl proxy“ das Dashboard wie unterliegend vom Kubernetes Master aufgerufen werden: *http://localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/*

Das Login, siehe Abbildung 14, kann über ein Token erfolgen, welches über das erstellte Secret, ähnlich wie bei dem Verbinden mit GitLab, wie folgt ausgelesen werden kann: „*kubectl get secret $(kubectl get serviceaccount dashboard -o jsonpath="{.secrets[0].name}") -o jsonpath="{.data.token}" | base64 –decode*“, vgl. (kubernetes, 2019)

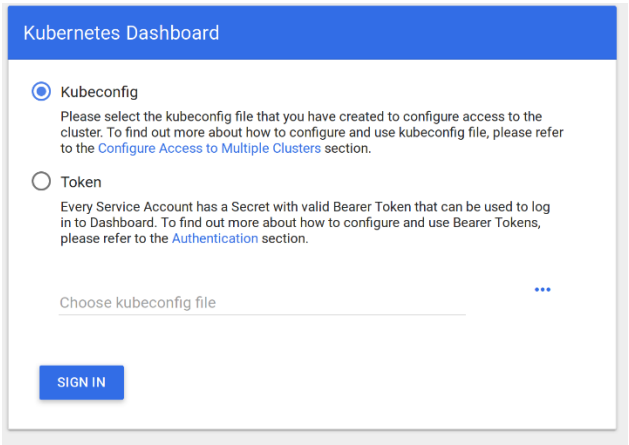


Abbildung 14 Kubernetes Dashboard Login mit Token

3.2.5. Workflow

Die Informationen zu den einzelnen Komponenten der CD-Umgebung haben einen Überblick über die Infrastruktur gegeben. Zusammenfassen lässt sich der Workflow, welcher zum Teil in der CD-Pipeline abgebildet wird, wie folgt:

- Entwicklung der Anwendung/Microservices in einer Entwicklungsumgebung.
- Pushen des Quellcodes in GitLab mittels Git.

Folgende Schritte werden nun durch den GitLab-Runner auf dem Kubernetes Cluster, welcher den Kubernetes Executor verwendet, siehe Abbildung 11, innerhalb der CD-Pipeline ausgeführt:

- Erstellen des Builds und Ablegen des entstandenen Artefakts.
- Ausführen automatisierter Tests.
- Erstellen des Docker Images unter Verwendung des Artefakts.
- Pushen des Docker Images in die externe Docker Registry.
- Starten eines Docker Containers (Pods etc.) anhand eines definierten Kubernetes-Manifests, welches unter anderem den Namen des Dockers Images und die Docker Registry angibt.
- Deployment im Kubernetes Cluster. Der Status kann über das Kubernetes Dashboard eingesehen werden.

3.3. Microservice-Architektur

Eine Microservice-Architektur lässt sich mit der beschriebenen CD-Umgebung folgendermaßen umsetzen. Es wird jeder Microservice in GitLab als eigenständiges Projekt betrachtet. Jeder Microservice muss, wie Abbildung 6 zeigt, mindestens die Definitionen des Dockerfiles, der gitlab-ci.yml und die des Kubernetes-Manifests (deployment.yml) im Root Verzeichnis beinhalten. Die Definitionen dieser Dateien sind im Anhang zu finden, siehe Anhang B.1, B.2, B.3. Jeder Microservice wird mittels Spring Boot umgesetzt und benötigt:

- **Controller:** RestController für die Kommunikation mit anderen Microservices und die Umsetzung von Logik.

- **Repository:** In einer Microservice-Architektur wird meist ein zentraler Service als Präsentation Layer verwendet, für direkte Aufrufe wird ein Repository verwendet.
- **Model:** Kommunikation mit der Datenhaltungsebene.

Mehrschichtige Systemarchitekturen weisen eine gute Skalierbarkeit auf, weil die einzelnen Schichten logisch voneinander getrennt sind. Ein gutes Beispiel dafür ist die Drei-Schichten-Architektur, wie in Abbildung 15 dargestellt.

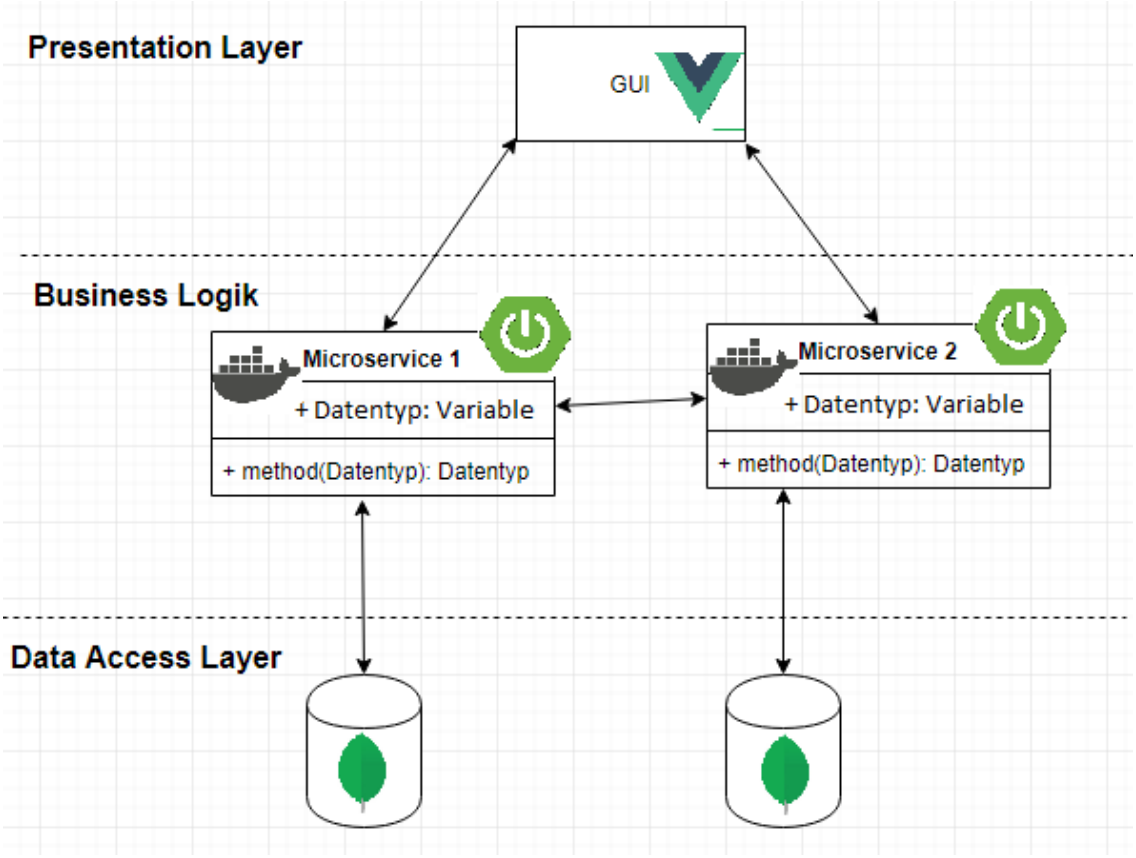


Abbildung 15 dreischichtige Microservice-Architektur

So können z. B. bei verteilten Systemarchitekturen wie einer Microservice-Architektur die Daten im Data Access Layer in mehreren getrennten Containern liegen. Auf dem Business Layer werden mehrere Microservices betrieben. Diese kommunizieren mit der darunter (Data Access Layer) und der darüber liegenden Schicht (Presentation Layer). Der Presentation Layer bereitet die von der Business Logik empfangenen Daten auf. Diese können clientseitig mit z.B. Vue.js gerendert werden, siehe Abbildung 15.

3.4. Fazit

Die Konzeption und die Implementierung einer CD-Umgebung gestalten sich vielseitig, da es eine große Auswahl an Programmen auf dem Markt gibt. Jedoch heben sich einige Programme durch Funktionen und die Breite der Dokumentation ab. Dadurch, dass eine CD-Umgebung äußerst komplex sein kann, muss kontinuierlich an der Weiterentwicklung der CD-Umgebung gearbeitet werden. Diese Weiterentwicklung ist ein iterativer Lernprozess und Teil des Konzepts von CD.

Die CD-Umgebung basiert auf einem VCS, dass es ermöglicht, gemeinsam an einem Projekt zu arbeiten. Durch die Einhaltung von Workflows, wie z.B. dem Gitflow, wird der Arbeitsablauf erleichtert. Nicht viele Programme bieten die Möglichkeit Git und CD miteinander zu vereinen. Deshalb hebt sich GitLab als web-basierter Open-Source Git-Repository-Manager durch die Integration einer CD-Pipeline hervor. Zudem bietet GitLab eine umfassende Dokumentation. Durch die Definition einer CD-Pipeline mittels der *gitlab-ci.yml* können einzelne Schritte (Stages) umgesetzt werden. Nichtdestotrotz ist es notwendig sich mit der entsprechenden Syntax und der Verwendung von Umgebungsvariablen vertraut zu machen.

Durch die leichtgewichtige Alternative zur kompletten Virtualisierung mittels Docker können Ressourcen des Hostsystems ideal genutzt werden. Die Umsetzung innerhalb der CD-Pipeline erfordert ein Grundverständnis über Dockerfiles, Docker Images, Docker Registry und Docker Container. Erläuterungen hierzu werden in der offiziellen Docker Dokumentation beschrieben.

Eine weitverbreitete Container-Orchestrierung und von Google entwickelte Software ist Kubernetes. Diese ermöglicht es Container über ein verteiltes System zu einem Cluster zusammenzuführen. Durch die Abstrahierung von Containern und Ressourcen, wie z.B. der Speicher, ist das problemlose Verschieben von Containern auf andere Hosts möglich. Mit Kubernetes lässt sich eine CD-Umgebung einfach aufbauen und verwalten, da das Ausführen operativer Aufgaben auf Ebene von Containern zugelassen ist. Auch der vom System verwaltete Docker Deamon, welcher Container startet und dessen Status abruft, erleichtert die Handhabung. Allerdings müssen für das volle Verständnis von Kubernetes eine Reihe von

Fachbegriffen und ebenso eine eigene Syntax beherrscht werden. Ein weiterer entscheidender Vorteil ist die Integration von Kubernetes in GitLab. Über die Oberfläche lassen sich durch die Eingabe von einigen Informationen aus dem Kubernetes Cluster beide Programme miteinander verbinden. Erst dieser Zusammenschluss und die Integration aus beiden Programmen ermöglicht die Erstellung einer vollständigen CD-Umgebung. Denn beide Programme sind darauf konzipiert einen Workflow nach einmaliger Konfiguration wiederholt automatisch zu durchlaufen.

Ein Microservice muss mindestens alle, in Abbildung 6 gezeigten, beschriebenen Dateien beinhalten um in einer CD-Umgebung automatisiert, gebaut und als Docker Container gestartet zu werden. Der Zusammenschluss aus mehreren Microservices bildet eine Microservice-Architektur. Diese kann, zusammen mit einem „Graphic User Interface“ (GUI) und einer oder mehreren Datenbanken, eine dreischichtige Softwarearchitektur und somit eine vollständige Anwendung bilden.

4. Continuous Delivery und Microservices in der Anwendung

In diesem Kapitel wird anhand eines Tutorials basierend auf Best Practices gezeigt, wie aus einer monolithischen Anwendung eine Microservice Anwendung erstellt und diese Microservice Anwendung in die in Kapitel 3 beschriebene CD-Umgebung integriert werden kann.

In diesem Kapitel werden grau hinterlegte Ausdrücke als Konsoleneingaben oder Ausdrücke in Java gekennzeichnet.

4.1. Anwendung: Multifunktionaler elektronischer Studentenausweis HSD-Card

Als Beispiel für die monolithische Anwendung dient der multifunktionale elektronische Studentenausweis, die HSD-Card. Es wird im Rahmen dieses Tutorials angenommen, dass die HSD-Card als Anwendung, gemäß (Thorsten Ebert, 2014), umgesetzt wurde und schematisch folgende Funktionen aufweist, siehe Abbildung 16.

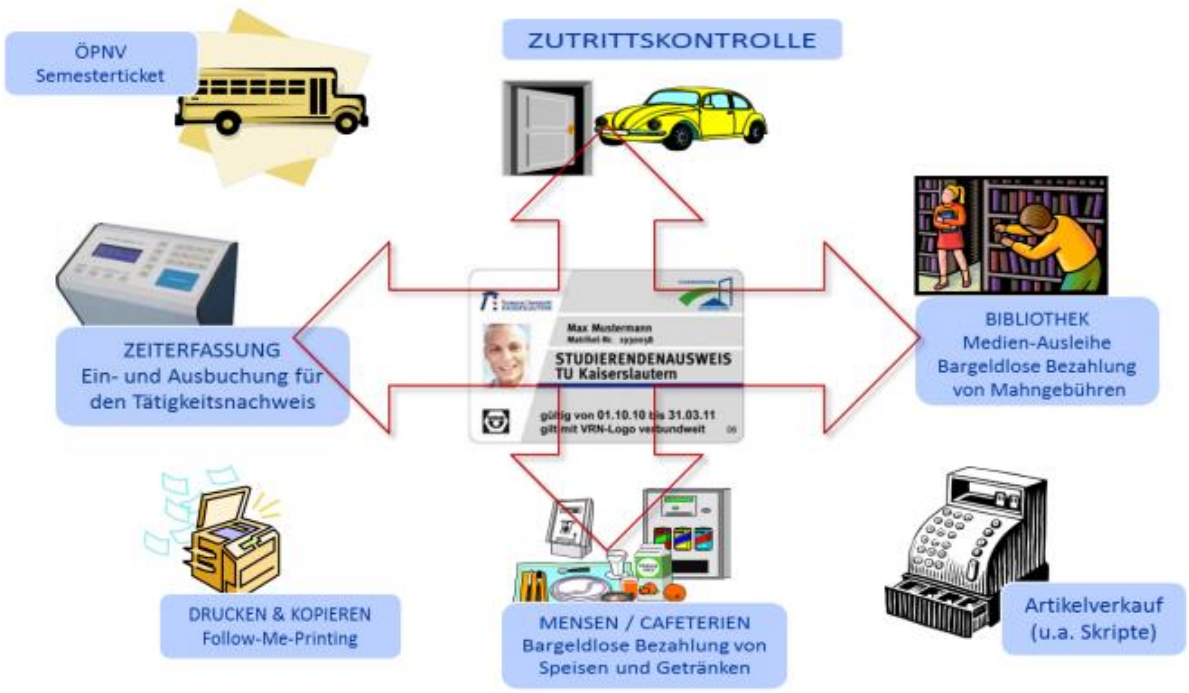


Abbildung 16 Übersicht der Anwendung der HSD-Card (Thorsten Ebert, 2014)

Es wird zudem davon ausgegangen, dass die folgenden Daten in einer Datenbank gespeichert sind bzw. werden. Von der Anwendung werden die folgenden Daten abgerufen und verändert, wenn der Studierende die HSD-Card verwendet:

1. Der Produktkatalog der Mensa/Cafeteria
2. Die Bücher der Bibliothek
3. Die Räumlichkeiten für die Zutrittskontrolle
4. Das Semesterticket für den ÖPNV
5. Die Drucker/Kopierer für das Abarbeiten der Druckaufträge
6. Die Daten der Studierenden
7. Die Bestellungen/Ausleihen eines Studierenden
8. Die Zeiterfassungen

4.2. Tutorial

Dieses Tutorial zeigt exemplarisch anhand der Funktionen der HSD-Card, wie eine Microservice Anwendung schrittweise erstellt werden kann. Beginnend mit dem Schneiden der Anwendung in Microservices über die Kommunikation bis zur Integration in eine CD-Umgebung mit Discovery Service, Skalierung sowie Load Balancing.

4.2.1. Methoden zur Erstellung einer Microservice-Architektur

Domain Driven Design, kurz DDD, ist eine Sammlung von Software Design Pattern, mit der die Transparenz komplexer Software Projekte erhöht werden kann. Durch DDD wird die Gesamtkomplexität der Anwendungsdomäne in verschiedene Fachdomänen unterteilt, strukturiert, modelliert und umgesetzt. Für das Verständnis von Microservices ist das Domain Driven Design grundlegend, vgl. (Wolff, 2016 S. S. 45).

Jede Fachdomäne wird als ein Bounded Context dargestellt. Jeder Bounded Context kann in einem oder, wenn die Komplexität zunimmt oder über Abteilungsgrenzen hinausgeht, in mehrere Microservices unterteilt werden.

Eine Methode zur Unterteilung in Fachdomänen ist die Erstellung einer Context Map (Kontextübersicht). Dabei wird zunächst die Kerndomäne festgelegt. Die Kerndomäne ist im vorliegenden Fall der Studierende, da dieser der Hauptkonsument der Anwendung ist. Dieser Bounded Context (Entität) wird zunächst eingekreist und benannt. Nachfolgend werden alle Funktionen dieses Bounded Contexts in den Kreis geschrieben. Durch diese Vorgehensweise kann überprüft werden, ob die Funktionen, wie z.B. die zum Drucken & Kopieren ein Teil des Kontexts des Studierenden sind. Ein weiterer Bounded Context ist das Inventar, das das gesamte Inventar der Hochschule enthält, welches der Studierende konsumieren/benutzen kann. In diesem Kontext findet die Verwaltung und Pflege der Produkte, Räume, Drucker/Kopierer und der anderen Gegenstände, wie z.B. Bücher der Bücherei, statt. Die letzte Fachdomäne ist die Buchhaltung, mit der die Studierenden die konsumierten Leistungen bezahlen. Dieser Bounded Context enthält Funktionen für das Hinzu oder Abbuchen von liquiden Mitteln vom Studierendenkonto. Außerdem ist es möglich Rechnungen zu erstellen.

Insgesamt ergeben sich drei verschiedene Fachdomänen. In der Context Map werden Verbindungen in Form von Doppelpfeilen eingezeichnet um die Abhängigkeiten der einzelnen Fachdomänen aufzuzeigen. Die führende Fachdomäne, die von einer anderen Fachdomäne aufgerufen wird, erhält am Pfeilende die Bezeichnung „up“ und am Pfeilbeginn „down“. Dies zeigt, welche Fachdomäne von der anderen abhängig ist. Das bedeutet, dass bei der Änderung

der Datenstruktur der führenden Fachdomäne die abhängige Fachdomäne ihre Schnittstelle entsprechend an diese Datenstrukturänderung anpassen muss. Die vollständige Context Map ist in Abbildung 17 dargestellt.

Ein weiterer Punkt ist die Datenhaltung. Nach Möglichkeit werden Abhängigkeiten soweit es geht vermieden. Besitzen die Daten der einzelnen Fachdomänen zu wenige Überschneidungen, sollten die Daten separat für jede Fachdomäne verwaltet werden. Im Fall der HSD-Card würde es z.B. keinen Sinn ergeben die Konsumdaten des Studierenden in der Inventar-Domäne zu speichern oder umgekehrt die Produkte der Mensa in der des Studierenden.

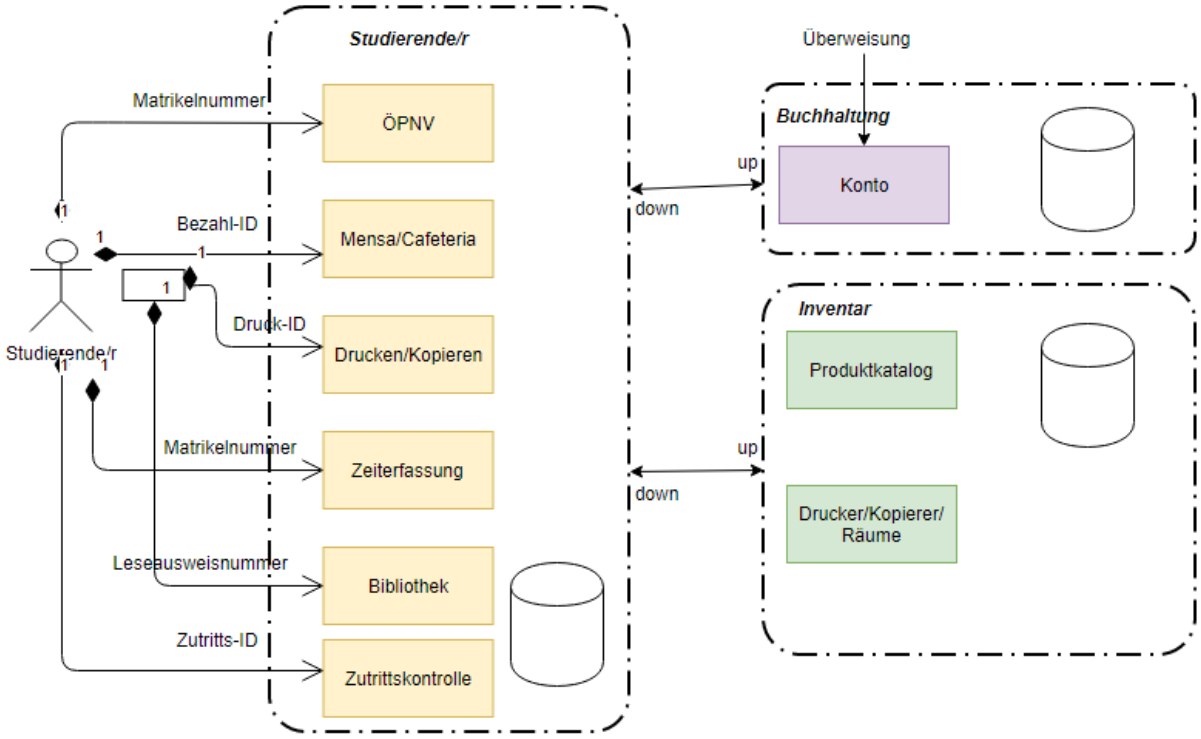


Abbildung 17 Context Map HSD-Card

Ein weiterer Teil von DDD ist die Obiquituous Language (allgemeingültige Sprache). Die Idee ist, dass auf allen Ebenen genau dieselben Begriffe genutzt werden. Dies soll die Kommunikation innerhalb des Unternehmens verbessern und helfen Missverständnisse zu vermeiden. Auch das Übersetzen vom Deutschen ins Englische ist ein Verstoß gegen Obiquituous Language. So muss beispielsweise bei der Erstellung von Microservices der Drucker in jedem Microservice als „Drucker“ und nicht als „Printer“ oder anders bezeichnet werden, vgl. (Wolff, 2016 S. S. 46 f.).

4.2.2. Erstellen eines Microservices

Um ein Microservice Projekt zu erstellen, wird in der Entwicklungsumgebung ein neues Projekt erstellt. Die Microservices in dieser Arbeit sind mit Spring Boot umgesetzt worden. Ein neues Spring Boot Projekt kann mit dem „Spring Initializr“ über die IDE erstellt werden. Nachdem die Java Version, der Name und weitere Parameter vergeben wurden, erscheint die Auswahl der „Dependencies“. Für einen Microservices, welcher später auf Kubernetes bereitgestellt wird, werden folgende „Dependencies“ benötigt, siehe Anhang A.3:

1. Spring Web Starter
2. Spring Reactive Web
3. Rest Repositorys
4. Spring Data MongoDB

Nachdem das Projekt erstellt wurde und alle Dependencies geladen wurden, wird die Struktur unter src>main>java>packagename in folgender Weise erstellt:

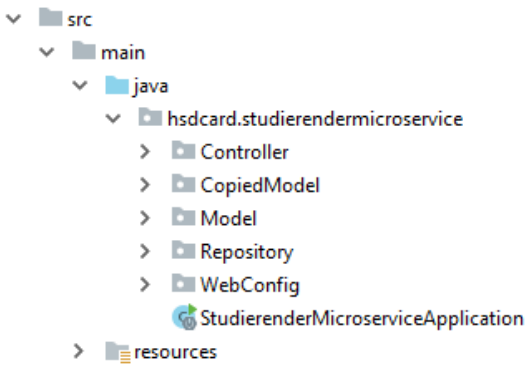


Abbildung 18 Ordnerstruktur des Microservices

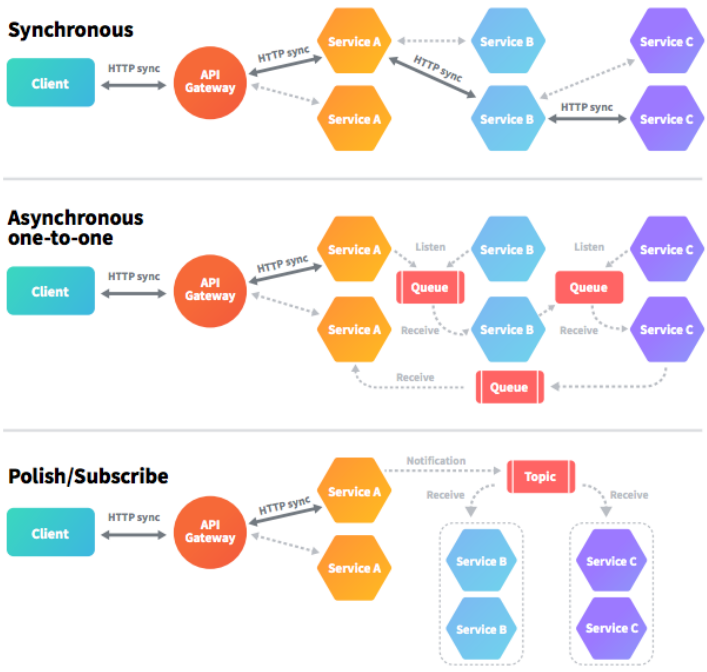
1. **Controller**: Dieser dient als Schnittstelle zu anderen Microservices oder dem GUI und bildet die Logik ab, die bei einem REST-Aufruf abgearbeitet werden soll, siehe Anhang B.4.
2. **CopiedModel:** Hier werden Models abgelegt, die für die Aufrufe der Daten anderer Microservices benötigt werden. Dies ist eine gängige Vorgehensweise und führt im vorliegenden Fall zu einer gewünschten Codeduplikation, siehe Abschnitt 4.2.3 & Anhang B.5. Da dieser Microservice das abhängige System (down) ist, muss er bei Änderungen im führenden System (up) auch angepasst werden, siehe Abbildung 17.
3. **Model:** Hier werden die Models des Microservices definiert, welche in identischer Struktur in die Datenbank geschrieben oder gelesen werden, siehe Anhang B.6.
4. **Repository:** Das Repository ist ein Interface, welches die Verknüpfung zur Datenbank (MongoDB) herstellt und vom Controller implementiert wird.

Zusätzlich können hier Methoden zum Abrufen von Daten implementiert werden, die für die Logik innerhalb des Controllers notwendig sind, siehe Anhang B.7.

5. **WebConfig:** Hier wird die Konfiguration des Servlets vorgenommen, siehe Anhang B.8, z.B. Cross-Origin Resource Sharing (CORS), damit Ressourcen anderer Microservices über den Browser genutzt werden dürfen, was notwendig für ein GUI ist, vgl. (Mozilla, 2019).
6. **microservicenameApplication:** Dies ist die Main Klasse des Projektes. In dieser werden beispielweise auch Java Beans definiert, siehe Anhang B.9.
7. **Resources:** Dies ist ein automatisch bei Projekterstellung generierter Ordner. Dieser enthält die application.properties Datei. Diese dient dazu globale Paramenter für den Microservice zu definieren, z.B. den Port des Microservices oder den Hostname der Datenbank, siehe Anhang B.10.

4.2.3. Kommunikation und Schnittstellen

Da die Microservice-Architektur in verteilten Systemen umgesetzt wird, benötigt es eine Art der Kommunikation. Dazu muss jeder einzelne Microservice eine Schnittstelle anbieten. Die Schnittstelle kann mit verschiedenen Technologien wie REST mit JSON oder Messaging umgesetzt werden. Außerdem ist entscheidend, ob diese synchron oder asynchron ist, vgl. (Wolff, 2018 S. S.15).



Eine synchrone Schnittstelle wird mittels REST/SOAP über das HTTP Protokoll realisiert. Es ist ein synchrones, zustandsloses Protokoll und sehr populär. Bei einer synchronen Kommunikation sendet der Client einen Request und

Abbildung 19 Synchrone und asynchrone Kommunikation zwischen Microservices (Mińkowski, 2017)

wartet auf eine Response vom Service (Microservice). Der Client selbst geht mit diesem Protokoll asynchron um. Das bedeutet, dass kein Thread blockiert wird, bis die Response vom Service eintrifft, vgl. *(Mińkowski, 2017)*.

Bei einer asynchronen Schnittstelle wird ein sogenannter Message Broker verwendet. Der Client wartet nicht auf eine Antwort, sondern nur auf eine Empfangsbestätigung des Brokers, dass der Request eingegangen ist. Der Broker arbeitet die eingegangen Request nach dem Prinzip „First in First out“ (FIFO) ab und sendet entsprechend die Response mit den Daten an den Client, vgl. (Mińkowski, 2017).

Für Schnittstellenänderungen gibt es das Konzept Consumer-Driven Contracts. Es klärt das Verhältnis von Service und Nutzer und spezifiziert die Schnittstelle in einem Vertrag. Nutzer können dadurch besser mit Aufrufen dieser Schnittstelle umzugehen. Annahmen, die ansonsten durch den Entwickler getroffen werden, werden ausgeschlossen, vgl. (Vitz, 2016).

In dieser Arbeit wird eine synchrone Schnittstelle verwendet, da der benötigte Aufwand geringer ist und die synchrone Schnittstelle für die Größe der Anwendung ausreichend ist.

Um über REST in Spring Boot zu kommunizieren war bisher die Verwendung von Methoden der Klasse „RestTemplate“ üblich. Allerdings sollte diese Klasse nicht mehr verwendet werden: „…The `RestTemplate` will be deprecated in a future version...“ (spring.io, 2019).

Stattdessen sollte wie in dem vorliegenden Tutorial der „WebClient.builder“ verwendet werden. Dieser wird zunächst in der „microservicenameApplication“ mit der @Bean Annotation als Java Bean definiert, siehe Anhang B.9. Eine Java Bean (Producer) folgt dem Singleton Pattern. Es wird also genau eine Instanz erzeugt, welche von mehreren Klasse innerhalb der Anwendung verwendet werden kann.

Die Controller Klassen sind die Klassen (Consumer), welche den „WebClient.builder“ benötigen um REST Aufrufe durchzuführen. Mittels Dependency Injection wird diese Instanz unter Verwendung der @Autowired Annotation den entsprechenden Klassen zur Verfügung gestellt, siehe Anhang B.4.

Da in Java „Strong Typing“ herrscht, ist ein gleicher Datentyp entscheidend für die Dependency Injection.

Die Controller Klasse ist mit der Annotation @RestController in der Lage REST Aufrufe durchzuführen. Die Art des REST Aufrufs wird ebenfalls mit einer Annotation über die Methode definiert, siehe Abbildung 20.

```
43      /**
44       * Gibt eine Liste von Druckern des Inventar-Services zurück
45       * @return Flux<DruckerKopierer>
46       */
47      @GetMapping("/drucker")
48      public Flux<DruckerKopierer> getAllDruckerKopierer(){
49
50          Flux<DruckerKopierer> druckerKopierer = webClientBuilder.build() WebClient
51                  .get() RequestHeadersUriSpec<capture of ?>
52                  .uri( s: "http://"+Hostname+":8082/drucker/all") capture of ?
53                   .accept(MediaType.APPLICATION_JSON) capture of ?
54                  .retrieve() ResponseSpec
55                  .bodyToFlux(DruckerKopierer.class);
56
57          return druckerKopierer;
58
59      }
```

Abbildung 20 REST-Aufruf über den WebClient.builder

Beim Erstellen des Microservices wurde als Dependency „Spring Reactive Web“ ausgewählt. Diese Abhängigkeit enthält Klassen und Methoden des Spring WebFlux Frameworks. Dieses verwendet intern Project Reactor, vgl. (baeldung, 2019).

Project Reactor ist dafür konzipiert worden „non blocking io“ für HTTP (einschließlich Websockets) innerhalb einer Microservice-Architektur zu bieten. Außerdem kann es hohe Durchsatzraten in der Größenordnung von bis zu 10 Millionen Requests pro Sekunde verwalten, vgl. (projectreactor, 2019).

In dieser Arbeit wird speziell der Typ „Flux“, für eine Sammlung von Ressourcen, und der Typ „Mono“ für eine einzelne Ressource des Project Reactors verwendet. Der „WebClientBuilder“ für den Typ „Mono“ ist vom syntaktischen Aufbau ähnlich dem des Typs „Flux“. Deshalb wird nur der Typ „Flux“ näher betrachtet. In Zeile 48, siehe Abbildung 20, wird der Rückgabewert der Methode vom Typ „Flux“ mit einem generischen Typ, in diesem Fall „DruckerKopier“, definiert. Der REST Aufruf findet durch „webClientBuider“ statt. Durch das Ausführen von aufeinander folgenden

Methoden wird der REST Aufruf nach und nach vervollständigt, siehe Zeile 50 f., Abbildung 20. Dabei werden beispielhaft folgende Methoden aufgerufen:

- **.get():** Legt die Art des REST Aufrufs fest, siehe Zeile 51.
- **.uri(„http://...):** Die URI sollte so dynamisch wie möglich aufgebaut sein, um bei Änderungen flexibel reagieren zu können. Typischerweise wird die Domain des Discovery Services (siehe Abschnitt 4.2.5) angegeben, siehe Zeile 52.
- **.accept(Datenformat):** Datenformat der erwarteten Daten, siehe Zeile 53.
- **.retrieve():** Die Response, siehe Zeile 54.
- **.bodyToFlux(Klasse):** Der Response Body wird ausgelesen und das JSON in ein Objekt der angegebenen Klasse überführt. Diese Klasse muss der Microservice vorhalten. Aus diesem Grund wurde in CopiedModel das Model des aufzurufenden Microservices hinterlegt, siehe Zeile 55.

Anschließend wird ein Objekt des Typs "Flux“, welches sämtliche Drucker/Kopierer enthält, zurückgegeben, siehe Zeile 57.

4.2.4. Implementierung in die CD-Umgebung

Um eine Kommunikation zwischen Microservice und Datenbank (MongoDB) zu ermöglichen, müssen dem Microservice einige Eigenschaften in Form von Variablen in der „application.properties“ im „resources“ Ordner mitgeteilt werden, siehe Abbildung 18 und Anhang B.10. Diese Eigenschaften werden dem Microservice beim Starten zugeschrieben und sind global verfügbar. Voraussetzung ist, dass die richtigen Dependencies in der POM.xml eingefügt wurden. Für das Bestimmen des Verhaltens in der CD-Umgebung unter Kubernetes wurde nachträglich folgende Dependency hinzugefügt, da diese nicht über den Spring Initializr gesucht werden kann, siehe Abbildung 21.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-kubernetes-config</artifactId>
    <version>0.3.0.RELEASE</version>
</dependency>
```

Abbildung 21 Dependency Spring Cloud Kubernetes (Ardiansyah, 2018)

Im Abschnitt 3.2.2 wurden CD-Pipeline Stages behandelt und erläutert, welche Dateien benötigt werden. Zudem wurde erklärt, wie GitLab mit Kubernetes

verbunden wird und mit Hilfe des GitLab-Runners die CD-Pipeline Stages ausführt. In der letzten Stage (Deploy) wird entweder auf eine Test- oder Produktionsumgebung deployed. Hier wird in der „gitlab.ci.yml“ definiert, welches Image zum Ausführen genommen wird, welche Umgebung Verwendung findet und welches Skript ausgeführt werden soll. Der Skriptteil ist für die Implementierung wichtig. Dort werden zunächst die Umgebungsvariablen aus GitLab in das Deployment Manifest, welches auch das Service Manifest beinhaltet, gesetzt. Anschließend werden die Manifest Dateien einzeln mit „kubectl apply –f MANIFEST.yaml“ ausgeführt, siehe Anhang B.1. In diesem Schritt werden unteranderem auch die Manifest Dateien für die MongoDB Datenbank ausgeführt.

In Kubernetes werden Ressourcen voneinander entkoppelt um diese flexibler zu gestalten. Aus diesem Grund müssen die Ressourcen einzeln über Manifest Dateien definiert werden, siehe Anhänge B.12, B.13, B.14, B.15 & Abbildung 22. Um eine MongoDB auf Kubernetes zu betreiben, sollten die Manifests in folgender Reihenfolge erstellt werden:

1. **PersistentVolume:** Ressourcen vom Typ PersistentVolume dienen zur Verwaltung von persistenten Speicher im Kubernetes Cluster. Im Gegensatz zu Volumes wird der Lebenszyklus in Kubernetes verwaltet. Außerdem bringen Volumes den Nachteil mit sich, dass sie an einen Container geknüpft sind. Wenn der Container aus irgendeinem Grund unfreiwillig beendet wird, gehen die Daten verloren, (cloud.google, 2019).
2. **PersistentVolumeClaim:** Ist der Anspruch auf eine PersistentVolume-Ressource. Pods verwenden diesen Typ als Volume. Der Kubernetes Master prüft diesen Anspruch um das gebundene Volume zu finden und stellt dieses dem Pod zur Verfügung, (cloud.google, 2019).
3. **ReplicationController:** Verwaltet die Anzahl der Pods durch die Angabe der Replicas Anzahl. Der ReplicationController startet exakt die genannte Anzahl an Pods und verwaltet diese. Sollte der Node mit einem oder mehreren Pods ausfallen, werden die ausgefallenen Pods auf einem anderen Node gestartet, damit die Gesamtanzahl wiedererreicht ist. Aus diesem Grund sollte auch bei einem Pod ein ReplicationController verwendet werden. Der

Pod bzw. der sich darin befindende Container referenziert entsprechend auf ein Volume.

4. **Service:** Der Service definiert z.B. über welche Ports externe Anfragen vom Service über den ReplicationController an den Pod weitergeleitet werden.

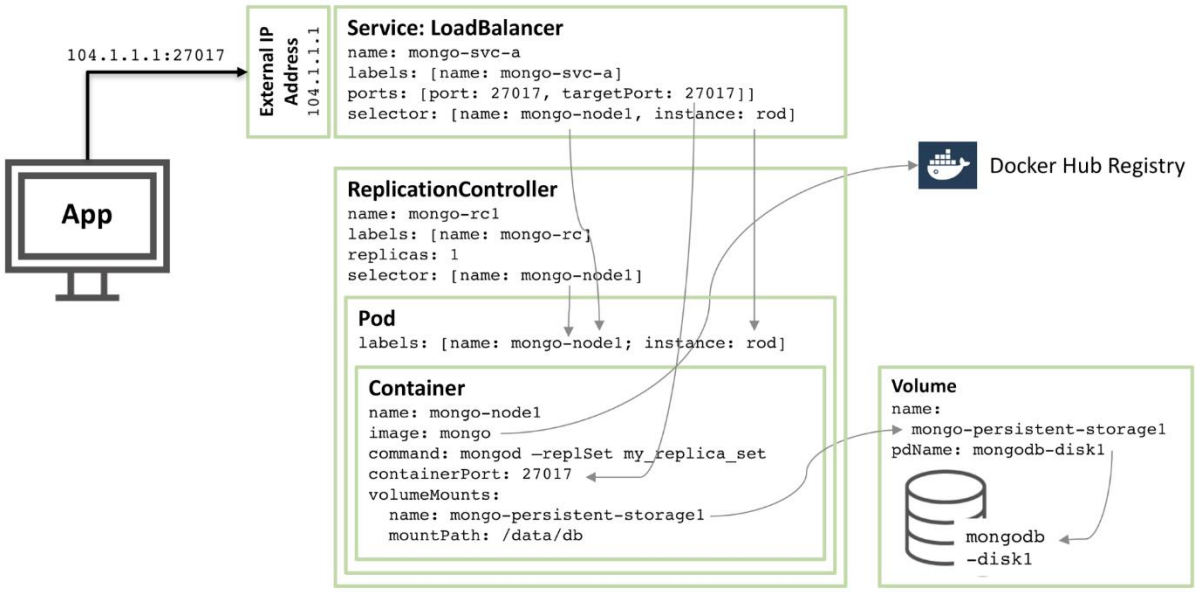


Abbildung 22 MongoDB Replica Set, dass als Kubernetes Pod konfiguriert und als Dienst verfügbar gemacht wurde (Morgan, 2016)

Die Authentifizierung des Microservices mit der Datenbank findet über ein Kubernetes Secret statt. In einem Kubernetes Secret werden vertrauliche Informationen wie Kennwörter oder Tokens gespeichert und verwaltet. Auf diese Weise vertrauliche Information einzufügen ist sicherer und flexibler als diese im Klartext in die Manifest eines Pods zu schreiben. Passwörter oder Benutzernamen werden beispielsweise als base64-codierter Hash in einem Secret abgelegt. In der „application.properties“ des Microservices sind ebenso Variablen als Platzhalter für diese Daten zur Authentifizierung mit der Datenbank hinterlegt. Diese Variablen werden durch das Deployment Manifest des Microservices gesetzt, wobei im Manifest explizit definiert ist, welches Secret verwendet werden soll und wie die entsprechenden Variablen des Secrets benannt sind, siehe Anhänge B.10 & B.11.

Innerhalb der CD-Pipeline sollten Manifests für PersistentVolume und PersistentVolumeClaim nicht ausgeführt werden, da es zu Fehlern kommen kann oder die bisherigen Daten verloren gehen, weil der Speicher neu angelegt wird.

4.2.5. Verfügbarkeit (Discovery Service & Load-Balancing, Skalierbarkeit, Ausfallsicherheit)

Da Microservices auf verteilten Systemen betrieben werden, in denen häufiger Kommunikations- oder anderen Problemen auftreten als in einem kohärenten System, ist es besonders wichtig diese Probleme durch Steigerung der Verfügbarkeit und Ausfallsicherheit zu kompensieren, vgl. (Fernuni-Hagen, 2019).

Es gibt verschiedene Methoden einen Microservice intern und extern zur Verfügung zu stellen. In jeder Microservice-Architektur gibt es für diesen Fall einen Discovery Service. Beim typischen Aufbau einer Microservice-Architektur unter Spring Cloud wird für diesen Zweck einen Discovery Service mittels Eureka implementiert. Diese dient als Registrierungsstelle bzw. als API-Gateway für alle Dienste, vgl. (Wolff, 2018 S. S.45).

In Kubernetes übernimmt das Service Manifest eines Microservices die Aufgabe des Discovery Services. Der Service verteilt den Datenverkehr auf die darunterliegenden Pods. Insgesamt gibt es vier verschiedene Arten in Kubernetes, mit denen Pods in einem Kubernetes Cluster zur Verfügung gestellt werden.

- **ClusterIP:** Dies ist der Standard-Kubernetes-Dienst und funktioniert ausschließlich bei internen Zugriffen innerhalb des Clusters. Dieser Dienst arbeitet über den Kubernetes-Proxy und verteilt somit den Datenverkehr an den Service. In dieser Art und Weise arbeitet beispielweise das Kubernetes-Dashboard. Mit dem Befehl „kubectl proxy“ auf dem Kubernetes Master wird der Proxy aktiviert. Der Proxy leitet Anfragen vom „localhost“ in das Kubernetes Cluster weiter. Bei dieser Methode muss „kubectl“ als authentifizierter Benutzer ausgeführt werden, vgl. (Dinesh, 2018).
- **NodePort:** Es wird ein Port definiert der, auf allen Knoten geöffnet wird. Über diesen Port werden die Daten an den Service geleitet, welcher die Daten wiederum an die Pods weiterleitet. Diese Methode wird typischerweise zu Testzwecken und nicht im Produktivsystem verwendet.
- **LoadBalancer:** Der Loadbalancer Service ist die Standardmethode um einen Dienst für extern bereitzustellen und wird deshalb auch in dieser Arbeit verwendet, siehe Abbildung 23 & Anhang A.4. Für die Implementierung wird der Typ „LoadBalancer“ angegeben und eine IP-Adresse, über die der

Service erreichbar ist. Die Einrichtung des Load-Balancers kann entweder in der Service Manifest erfolgen, siehe Anhang B.15, oder der Service kann nachträglich über die CLI wie folgt gepatcht werden:

```
kubectl patch svc mircoservicename –n namespace –p '{"spec": {"type": "LoadBalancer“, "externalIPs": ["192.168.25.110"]}}‘
```

Die angegebene IP-Adresse ist die des Kubernetes Masters. Der Port des Services wird in der Manifest mit z.B. “ - **port**: 8080“ angegeben.

Bei dieser Methode gibt es keine Filterung, kein Routing etc. Dadurch kann fast jede Art von Datenverkehr gesendet werden, z.B. HTTP, TCP, UDP, Websockets uvm.

- **Ingress:** Im Gegensatz zu den anderen Methoden ist Ingress kein Service, sondern eine Art Router, welcher sich als Eintrittspunkt zum Cluster vor den Services befindet, siehe Abbildung 23. Ingress funktioniert pfadbasiert, wodurch ein subdomänenbasiertes Routing direkt zu den Services der einzelnen Microservices möglich ist. Diese Funktionsweise hat den Vorteil, dass nur ein Load-Balancer mit gleicher IP-Adresse und gleichem Port verwendet wird. Im Gegensatz dazu wurde in der vorher beschriebenen Methode vor jedem Service ein Load-Balancer vorgeschaltet, für den nicht zwingend eine andere IP-Adresse, aber zumindest immer ein anderer Port vergeben werden muss. Außerdem bietet Ingress noch zahlreiche Erweiterungsmöglichkeiten, wie z.B. den Cert-Manager um SSL-Zertifikate für Domains bereitzustellen, vgl. (Dinesh, 2018).
  Ingress ist die leistungsstärkste der vier Methoden, mit denen Microservices nach außen zur Verfügung gestellt werden können. Ingress ist jedoch auch die komplizierteste und umfassendste Methode, weshalb eine Implementierung im Rahmen dieser Arbeit nicht möglich war. Die Google Kubernetes Engine (GKE) oder andere Cloud Anbieter bieten eine eigene fertige Implementierung an. Eine einfache Alternative, mit der Kubernetes lokal betrieben werden kann ist „Minikube“. Innerhalb von „Minikube“ ist es möglich den Ingress Controller über das Aktivieren eines Addons zu implementieren. Eine Anleitung befindet sich in der Kubernetes Dokumentation, siehe (Documentation Kubernetes, 2019). Letztlich muss

nur noch die Ingress Manifest Datei des Microservices ausgeführt werden, siehe Anhang B.18.

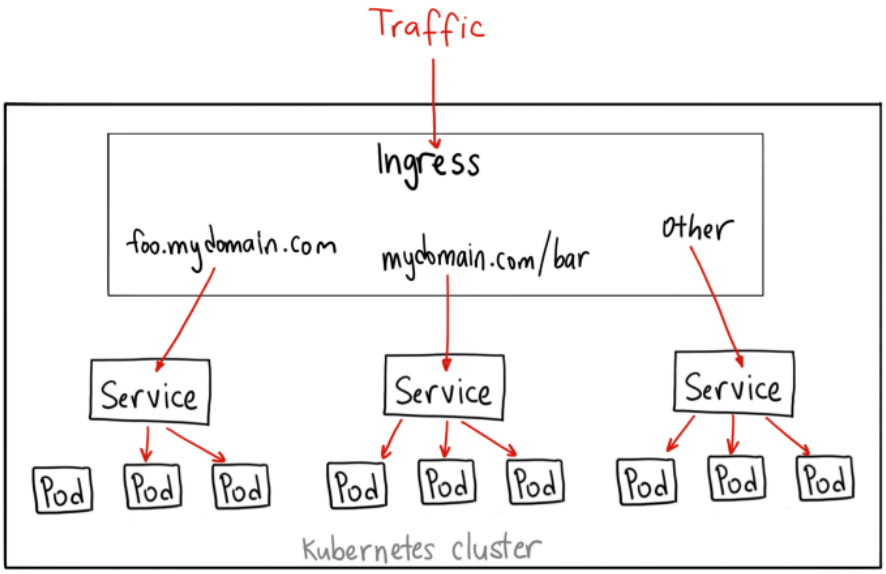


Abbildung 23 Load-Balancing über Ingress (Dinesh, 2018)

Ein Load-Balancing ist nur dann sinnvoll, wenn der Microservice skalierbar ist. Über die grafische Oberfläche des Kubernetes-Dashboards lässt sich die Anzahl der Pods dadurch einstellen, dass beim Deployment über das Aktionsmenü rechts „Scale“ ausgewählt und die Anzahl der Pods eingegeben wird. Mit dem Bestätigen innerhalb des Modal-Fensters skaliert Kubernetes automatisch die Anzahl der Pods und startet zusätzliche Pods oder stoppt vorhandene Pods um die gewünschte Anzahl zur Verfügung zu stellen, siehe Abbildung 24. Ein andere Möglichkeit der

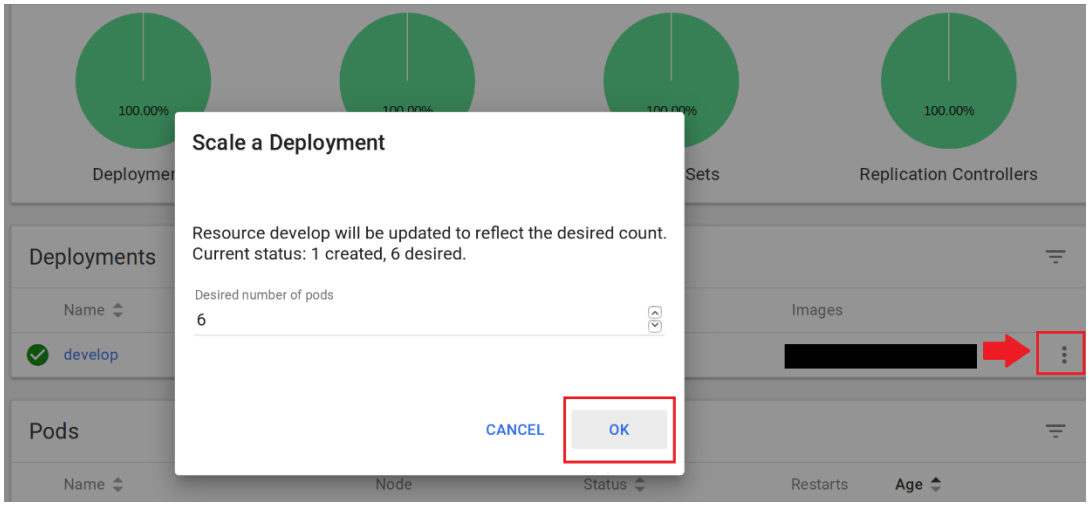


Abbildung 24 Skalierung des Deployments

Skalierung besteht darin, die Anzahl der Pods direkt im Deployment Manifest des Microservices unter „`replicas: 1`“ festzulegen, siehe Anhang B.3. Auf diese Weise

wird direkt bei Ausführung des Manifests die entsprechende Anzahl von Pods gestartet.

Auch über andere Arten von Manifests (Pods, Replication Sets oder Replication Controller) lassen sich Pods skalieren. Dabei werden allerdings nur neue Pods hinzugefügt.

4.3. Fazit

Der multifunktionale elektronische Studentenausweis, die HSD-Card, wurde mit ihren Funktionen schematisch dargestellt und mit weiteren Annahmen wie Daten angereichert um die Darstellung als Anwendung zu ermöglichen. Die HSD-Card dient als Ausgangspunkt für das Tutorial zur Erstellung einer Microservice-Architektur.

Mit dem Domain Driven Design, einer Sammlung von Software Design Pattern, wird innerhalb des Tutorials gezeigt, wie eine Anwendung domänenbasiert in einzelne Microservices separiert und schematisch in einer Context Map dargestellt wird.

Mit dieser Grundlage lässt sich durch Spring Boot ein Microservice umsetzen. Der Aufbau des Microservices innerhalb Spring Boots ist aus Best Practices entstanden und wird im Detail erläutert.

Die Auswahl der richtigen Art der Kommunikation über eine Schnittstelle ist von Bedeutung. Die populärste Art der Kommunikation ist die Implementierung einer synchronen Schnittstelle über REST/HTTP mit JSON als Datenformat. Mit Hilfe des WebClientBuilders, einer neuen Methodik für REST Aufrufe innerhalb Spring Boot, ist es möglich die genaue Art des REST Aufrufs, das erwartete Datenformat, die URI und weitere Parameter anzupassen. Diese Methodik wird von Spring Reactive Web implementiert und beinhaltet zusätzlich das Framework Project Reactor. Dieses Framework wurde dazu konzipiert „non blocking io“ für HTTP (einschließlich Websockets) innerhalb einer Microservice-Architektur zu bieten. Außerdem kann es hohe Durchsatzraten in der Größenordnung von bis zu 10 Millionen Requests pro Sekunde problemlos verwalten.

Die Implementierung in die CD-Umgebung wird durch die Datei „gitlab-ci.yml“ festgelegt, welche sämtliche Stages der CD-Pipeline beinhaltet. Auf die letzte Stage wird näher eingegangen. In dieser Stage werden die notwendigen Kubernetes Manifest Dateien deployed. Diese legen die eigentliche Infrastruktur im Kubernetes Cluster fest. Darunter fällt bespielweise das Deployment Manifest, welches vorgibt, welches Docker-Image für den Microservice geladen werden soll. Das Deployment Manifest startet darauf basierend Pods (Container). In Manifest-Dateien werden ebenso ein Discovery Service, der Replication Controller oder eventuell benötigter Speicher sowie Passwörter definiert und anschließend in Kubernetes verwaltet.

Microservices werden auf verteilten Systemen betrieben, in denen häufiger Kommunikationsprobleme auftreten. Die Kommunikationsprobleme müssen durch eine hohe Verfügbarkeit kompensiert werden. Kubernetes bietet verschiedene Arten um Microservices intern und extern zur Verfügung zu stellen und den Datenverkehr auf einzelne Instanzen von Microservices zu verteilen.

Ein Load-Balancer, welcher vor den Service geschaltet wird, verteilt automatisch den Datenverkehr an verfügbare Instanzen des einzelnen Microservices. Diese Arbeitsweise wird über den Service im Manifest implementiert oder kann nachträglich gepatcht werden. Eine komplexere, dementsprechend auch leistungsstärkere Methode bietet Kubernetes Ingress, welches pfadbasiert funktioniert und somit auch ein subdomänenbasiertes Routing unterstützt.

Ein Load-Balancing ist nur dann sinnvoll, wenn der Microservice skalierbar ist. Über die grafische Oberfläche des Kubernetes-Dashboard lässt sich die Anzahl der Pods einstellen. Dabei ist zu beachten, dass die Anzahl nur beim Deployment der Gesamtanzahl entspricht, bei anderen Arten wird die eingebende Anzahl an Pods zusätzlich zur bestehenden Anzahl ergänzt.

5. Zusammenfassung

In diesem Kapitel wird zusammenfassend erläutert wie die technischen Rahmenbedingungen des gesamten Projekts sind und wie diese umgesetzt wurden. Darüber hinaus ist auch die im Folgenden dargestellte Findungsphase der richtigen Programme von Bedeutung.

Zudem gab es zahlreiche Herausforderungen und Probleme bei der technischen Umsetzung. Innerhalb der Retrospektive wird gezeigt, welche Herausforderungen und Probleme gelöst werden konnten und bei welchen Lösungen noch Optimierungsbedarf besteht. Weitere Maßnahmen zur Verbesserungen und Funktionen der HSD-Card werden gezeigt. Abschließend folgen Ausführungen über generelle Zukunftsaussichten für weitere Projekte dieser Art.

5.1. Technische Aspekte der CD-Umgebung

Das Kapitel 3 zeigt bereits einige Komponenten der CD-Umgebung im Detail. Ganzheitlich betrachtet gibt Abbildung 25 einen Überblick über alle in der CD-Umgebung verwendeten Programme. Dabei ist die Wahl der Entwicklungsumgebung unbedeutend, denn die meisten modernen IDEs unterstützen die Integration von GIT als Versionskontrollsystem. Für die Integration einer CI/CD-Pipeline wurden einige Programme miteinander verglichen, darunter auch Jenkins, welches auch oft mit Spring Boot Anwendungen zum Einsatz kommt. GitLab ist ein web-basierter Open-Source Git-Repository-Manager und kann Git-Repositorys intern verwalten, wogegen Jenkins auf ein externes Repository zugreift. Zudem bietet GitLab ab Version 8.8 eine Integration zur Container-Orchestrierungsplattform Kubernetes an, welches ein wichtiger Bestandteil einer Microservice-Architektur ist, wenn diese auf verteilten Systemen in Linux-Containern bereitgestellt werden.

Der gesamte Aufbau der Abbildung 25 soll eine cloudbasierte Infrastruktur simulieren, in der jeder fettgedruckte schwarze Rahmen, innerhalb des Hostsystems, einen eigenen Server in der Cloud darstellen soll. Dementsprechend wurde GitLab nach dem Prinzip „Platform as a Service“, kurz „PaaS“ mittels Docker-Compose umgesetzt, zur Umsetzung siehe Abschnitt 3.2.1 & Anhang B.16. GitLab greift zur Ablage der fertigen Docker-Images auf eine private Docker Registry zu. Diese kann direkt bei Docker Hub eingerichtet werden oder auf einer anderen Maschine, in der Docker installiert ist.

Das Kubernetes Cluster wurde innerhalb von zwei Ubuntu VMs, bestehend aus einem Master und einem Node, installiert, zur Umsetzung siehe Abschnitt 3.2.4. Anschließend wurde GitLab mit dem Kubernetes Cluster verbunden und der GitLab-

Runner, welcher zur Ausführung der CD-Pipeline notwendig ist, innerhalb des Kubernetes Clusters installiert, zur Umsetzung siehe Abschnitt 3.2.1.

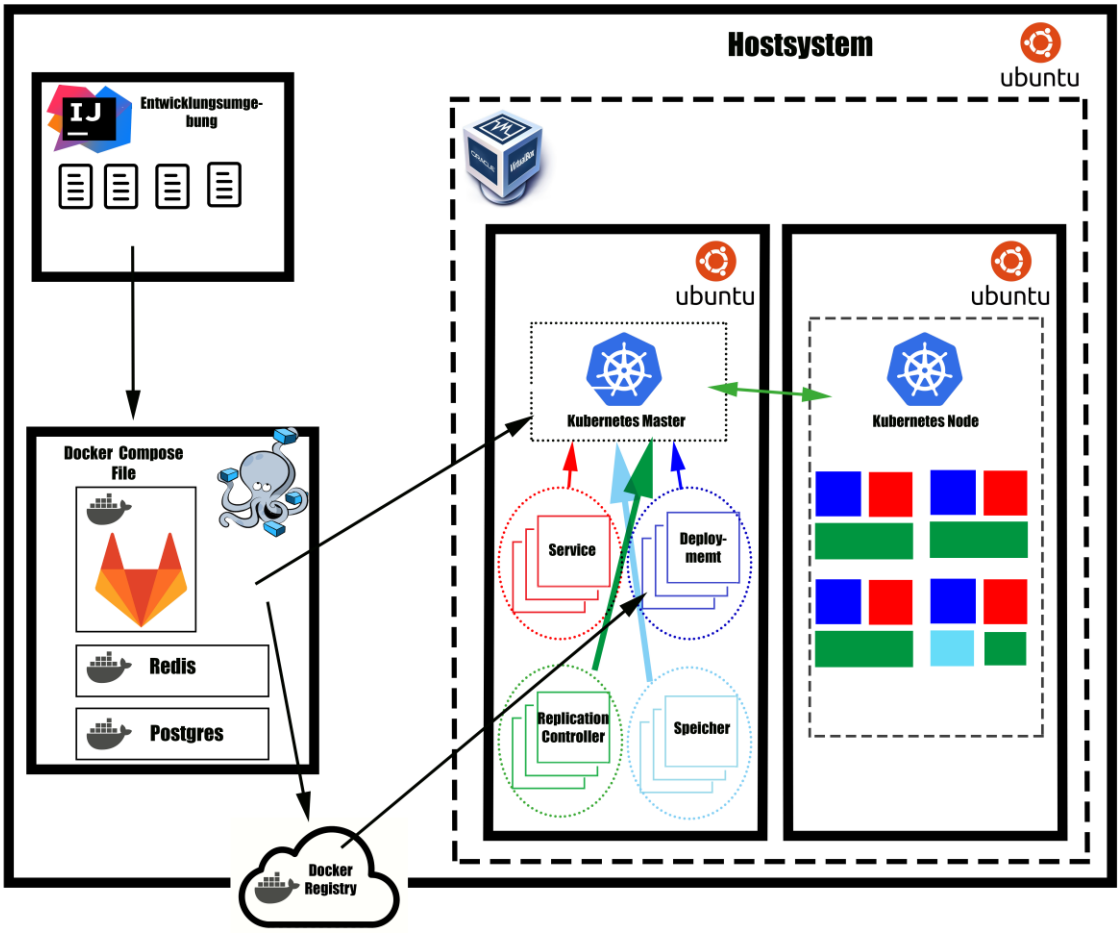


Abbildung 25 Gesamtaufbau der Infrastruktur

Für die Orchestrierung der Container wurden mehrere Ansätze der technischen Umsetzung verfolgt.

Darunter auch fertige Lösungen, die innerhalb einer VM betrieben werden und bereits eine große Anzahl von Funktionen integrieren. Eine Lösung wird von Kubernetes bereitgestellt und nennt sich Minikube. Diese VM beinhaltet einen Kubernetes Master und Node sowie die Kubernetes CLI „*kubectl*“. Die Aktivierung von Addons, wie z.B. Ingress, kann mittels eines CLI-Befehls vorgenommen werden. Ebenfalls konnte eine Verbindung zu GitLab, exakt wie in Abschnitt 3.2.1 beschrieben, durchgeführt werden. Jedoch sind die Möglichkeiten der internen Kubernetes Konfiguration und der Speicherverwaltung sowie die gesamte

Performance sehr begrenzt, weshalb Minikube getestet wurde, es jedoch nicht für den Zweck dieser Arbeit ausreichend war. Für das Testen eines einzelnen Microservices auf dem lokalen Rechner eines Entwicklers ist die Variante vollkommen ausreichend.

Mit den Standard-Einstellungen wird Minikube mit zwei CPUs und einem GB RAM gestartet. Um diese Einstellungen zu ändern, wird empfohlen, Minikube zu stoppen, zu löschen und anschließend mit mehr Ressourcen zu starten, vgl. (skhatri, 2018).

Eine ähnliche Lösung bietet Red Hat mit Minishift, welches ein Openshift Cluster ist und auf einen Fork des Kubernetes Projekts basiert. Minishift wirkt anfangs durch eine grafische Oberfläche, über die einzelne Pods verwaltet werden können, gefällig. Auch das Monitoring ist wesentlich besser als das des Kubernetes Dashboards, da es sich automatisch aktualisiert. Demgegenüber gibt es keine direkte Integration seitens GitLab mit Openshift. Dies bedeutet, dass der GitLab-Runner eigenständig über eine Manifest Datei definiert und nach extern zur Verfügung gestellt werden muss, damit der GitLab-Runner von der CD-Pipeline aus angesprochen werden kann. Die eigene CLI „*oc*“ muss zusätzlich installiert und konfiguriert werden. Dies erfordert einen enormen Konfigurationsaufwand, da Minishift ebenfalls in einer VM betrieben wird.

Beide Lösungen, sowohl Minishift als auch Minikube, haben zusätzlich gemeinsam, dass es zu unerwarteten Fehlern kommen kann, welche ausschließlich durch Löschen und Neustarten der VM-Instanz behoben werden können. Daher sollte ein Entwickler seine Konfigurationsschritte genau dokumentieren.

Aus diesen Gründen empfiehlt es sich ein Kubernetes Cluster von Scratch aufzubauen. Einen Einstieg in die Kuberneteswelt bietet „katacoda.com“ mit vielen Tutorials.

Die für dieses Projekt verwendete Hardware beinhaltet einen 8 Kernprozessor und 24 GB Arbeitsspeicher.

5.2. Herausforderung und Probleme bei der Umsetzung

Die größte Herausforderung, welche bewältigt werden mussten, bestand in der Integration einer Orchestrierungsplattform. Minikube und Minishift schienen im ersten Augenblick die idealen Lösungen ohne viel Konfigurationsaufwand für einen schnellen Einstieg zu sein, doch sprachen die im Folgenden dargestellten Probleme gegen diese Entscheidung.

Sollte sich für Minishift/Openshift entschieden werden, ist Folgendes zu beachten.

- Die Integration von Minishift in GitLab ist nicht trivial, da der GitLab-Runner als Template selbst definiert werden muss, siehe Anhang A.5.
- Damit Container gestartet werden dürfen sollte sich als Systemadmin eingeloggt und eine Rolle mit ausreichend Rechten definiert werden. Es ist erforderlich Container im „Privileg Mode“ zu starten, siehe Anhang A.6.

Sollte sich für die Integration mit Minikube/Kubernetes entschieden werden, ist Folgendes zu beachten:

- Es kann vorkommen, dass Container nicht mit der Kubernetes API kommunizieren können. Für diesen Fall sollten die Firewall Einstellungen überprüft werden, siehe Anhang A.7.
- Die Steigerung der verfügbaren Ressourcen können nur durch Stoppen, Löschen und anschließendes Neuerstellen samt Parameter durchgeführt werden, ansonsten stehen standardmäßig beim Start zwei CPU und ein GB RAM zur Verfügung.

Probleme, die innerhalb GitLab aufgetreten sind:

- Wenn die entsprechenden Informationen für die Integration von Kubernetes/Openshift in GitLab gefunden worden sind, siehe Abschnitt 3.2.1, ist darauf zu achten, dass unter den administrativen Einstellungen (Admin Area>Settings>Network>Outbound Request) Requests zur API innerhalb eines lokalen Netzwerkes erlaubt sind, siehe Anhang A.8.
- Bei der Installation von GitLab Add-Ons unter Openshift ist zu beachten, dass unter „Base Domain“ die IP zur API eingetragen wird, siehe Anhang A.9. Ansonsten kann das Installieren von AddOns fehlschlagen.

- Der GitLab-Runner kann standardmäßig keine Docker Befehle ausführen. Deshalb ist es notwendig einige Einstellungen in der Definition der CD-Pipeline innerhalb der entsprechenden Stage, welche Docker-Befehle verwendet, vorzunehmen. Als Service werden „docker in docker“ (dind) sowie ein Einstiegspunkt angegeben. Außerdem benötigt es Variablen, die den Docker Treiber und den Host angeben, siehe Anhang B.1.
- Jedes Projekt muss mit dem Kubernetes Cluster verbunden werden. Die GitLab-App Helm Tiller muss installiert werden. Der GitLab-Runner oder andere Apps werden geteilt und können einmal pro Cluster installiert.
- GitLab bietet seit der Version 8.8 eine eigene Container Registry, welche eine Docker Registry ersetzen bzw. integrieren soll. Dazu soll laut der GitLab Dokumentation eine Docker Registry eigenständig als Container aufgesetzt werden, mit dem Verweis auf die Docker Dokumentation, siehe (Gitlab.com, 2019). In der „gitlab.rb“ Konfigurationsdatei kann die Container Registry aktiviert werden mit einem Verweis auf die Docker Registry. Anschließend erscheint im Menü der GitLab Oberfläche der Eintrag „Container Registry“. Die Authentifizierung bei der Docker Registry findet über den GitLab Benutzer statt. Über die CLI ist dies nachweislich möglich. Jedoch erscheinen in der Oberfläche keine Docker Images und über die CD-Pipeline kann nicht zugegriffen werden, siehe Anhang A.10 & A.11. Zudem wurde die Docker Registry extern ausgelagert, auf einen Server mit gültigem SSL-Zertifikat.

Herausforderungen und Probleme, die in Kubernetes aufgetreten sind:

- Um einen eigenen DNS-Server für Pods zu verwenden, muss der Kubernetes DNS angepasst werden. Der Kubernetes DNS ist im Namespace „Kube-System“ zu finden und kann über das Kubernetes-Dashboard direkt angepasst werden. Dabei gibt es zwei verschiedene Arten von DNS, der „Kube-DNS“ und der „Core-DNS“. In dieser Arbeit wurde „Core-DNS“ verwendet. Über dessen Config-Map kann das „Corefile“ angepasst werden, siehe (Kubernetes-Customizing-DNS-Service, 2019) & Anhang A.12.
- Sollte das Deployment melden „no basic auth credenials“ und kann das Image nicht aus der Docker Registry laden, dann kann dies daran liegen,

dass kein Secret für die Docker Registry in diesem Namespace hinterlegt wurde. Oder es wurde im Deployment nicht angegeben, welches Secret verwendet werden soll, siehe Anhang B.3 & A.13.

- Das SSL-Zertifikat der Docker Registry ist nicht gültig oder abgelaufen. Dieses muss über den Server, auf dem die Docker Registry betrieben wird, erneuert und anschließend die Docker Registry neu gestartet werden. Eine Einstellung in GitLab oder Kubernetes ist hier nicht notwendig, siehe Anhang A.14.

5.3. Retrospektive

Jedes Projekt ist einzigartig und läuft über einen gewissen Zeitraum. Es gibt einen definierten Start- und Zielzeitpunkt, zwischen denen die Anforderungen erfüllt werden sollen. Eine Retrospektive ist eine gängige Projektmanagementmethode, um über positive und negative Aspekte zu reflektieren.

Über dieses Projekt können folgende Aussagen getroffen werden:

Pro:

- Durch die Implementierung von CD war es möglich Konzept-, Entwicklungs-, Testphase und Release der HSD-Card Microservice-Anwendung inkrementell und iterativ in einem kurzen Abstand durchführen.
- Bei der praktischen Ausarbeitung einer Microservice-Architektur und dem Anwenden von Pattern des Domain Driven Designs sind kleine Deployment-Einheiten entstanden. Dies ist ein wichtiger Faktor für die Beschleunigung und Flexibilität eines Systems und grenzt sich von einer monolithischen Anwendung ab.
- Die Strukturierung einer Microservice-Anwendung und die verwendeten Technologien, basierend auf Best Practices, dienten als gutes Vorbild für das Tutorial und können wiederholt angewendet werden.
- Der hohe Automatisierungsgrad und die Steigerung der Qualitätssicherung durch CD erwiesen sich als Erfolgsfaktor für das Projekt.
- Durch das Simulieren einer Cloud-Infrastruktur konnte ein standardisiertes System, das skalierbar und zuverlässig ist, implementiert werden.

Kontra:

- Die Findungsphase der passenden Orchestrierungsplattform. Fehlgeschlagene Implementierungen von Minikube und Minishift kosteten viel Zeit.
- Der initiale Aufwand für die Einrichtung einer CD-Umgebung sowie Wartung und Betrieb dieses Systems sollten bei der Planung berücksichtigt werden.
- Die GitLab Container Registry und die Beschreibung in der Dokumentation sind nicht ausreichend. Aus diesem Grund konnte sie nicht für dieses Projekt verwendet werden.
- Eine Docker Registry ist für den Aufbau der CD-Pipeline zwingend erforderlich, leider bietet Docker Hub über den kostenlosen Account nur ein privates Repository an. Deshalb war es notwendig eine eigene Docker Registry zu implementieren, was wiederum zu Problemen geführt hat.

5.4. Zukunftsaussichten

Die Möglichkeiten und Zukunftsaussichten der HSD-Card innerhalb einer Microservice-Architektur sind beinahe grenzenlos. Zum einem können die Logik und das Datenmodell der bestehenden Microservices weiter verbessert und erweitert werden. Zum anderen können noch weitere Funktionen als neue Microservices hinzukommen.

In dem Dokument der Hochschule werden folgende zusätzliche Funktionen angegeben, (Thorsten Ebert, 2014):

- Steuerung der Gebäudeelektronik (Licht, Heizung, Belüftung, Präsentationstechnik)
- Ausweis für die Prüfungsanmeldung
- Ausweis für die Authentifizierung an Servern und Systemen

Das Meistern all dieser Herausforderungen einer Microservice-Architektur erfordert die Koordination und Kommunikation verschiedener Teams. Demnach ist ein Team zwar ein der Regel für einen Microservice und die entsprechende Pipeline verantwortlich, jedoch funktioniert die gesamte Anwendung nur, wenn die Entwicklungskultur teamfähig bleibt. Entscheidend sind ein solides Design und die Analyse der Domäne. Die Wahl aus einer Vielfalt an Technologien, die Modellierung

der Daten und die Gestaltung der APIs sollten mit der nötigen Sorgfalt durchgeführt werden.

Mit den Trends hin zur agilen Entwicklung und DevOps, zu der Verlagerung von RZ-Infrastrukturen in die Public Cloud sowie zu der Etablierung moderner Microservice-Architekturen, werden Container zum unverzichtbaren Technologie-Baustein, siehe Abbildung 26 vgl. (Velten, 2019).

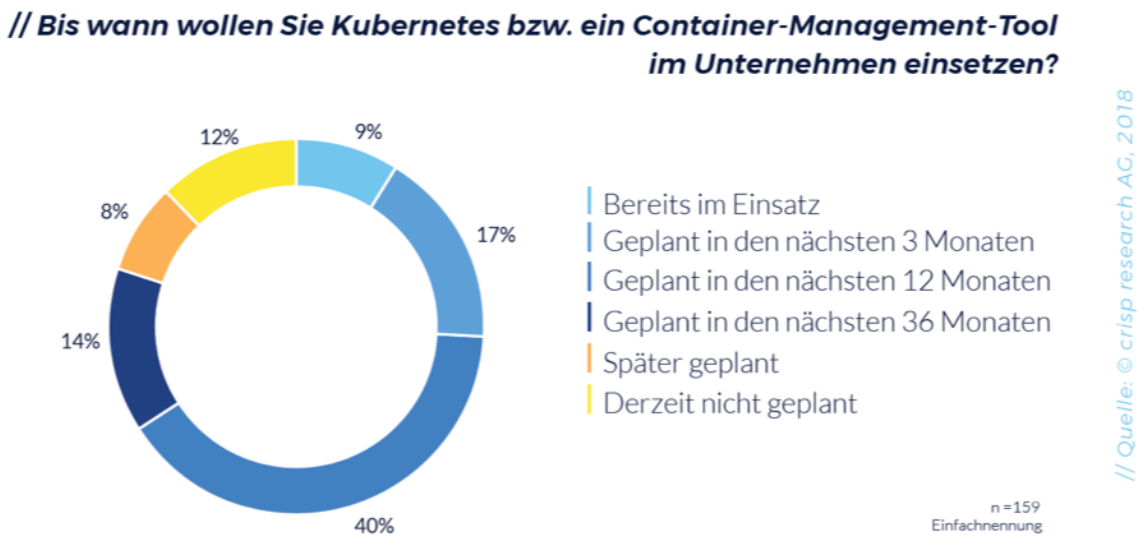


Abbildung 26 Umfrage Kubernetes (Velten, 2019)

5.5. Fazit

Die technische Umsetzung der CD-Umgebung hat eine sorgfältige Analyse der Programme und dessen Funktionen erfordert. GitLab konnte sich durch seine Vielfalt an Integrationsmöglichkeiten gegenüber Jenkins durchsetzen. Die Integration von Kubernetes in GitLab war das ausschlagende Argument, diesen web-basierten Open-Source Git-Repository-Manager zu verwenden. Ganzheitlich sollte das Projekt an das Prinzip von Cloud-Native und „Plattform as a Service“ (PaaS) angeknüpft werden, weshalb GitLab in einer Containerstruktur mittels Docker-Compose und das Kubernetes Cluster in VMs umgesetzt wurden, siehe Abbildung 25. Diese Art der Umsetzung in verteilten Systemen stellt eine verbreitete Art der Bereitstellung von Webanwendungen speziell von Microservices da. Für diese Arbeit wurde eine leistungsstarke Hardware benötigt um die Infrastruktur komplett abzubilden.

Durch diverse Herausforderungen und Probleme, besonders in der Phase der Findung geeigneter Software, ist sehr viel Aufwand und Zeit in die falsche Richtung oder in unerwartete Probleme bei der Umsetzung investiert worden.

Rückblickend betrachtet hat dieses Projekt mir sehr viel technisches Verständnis und Expertise in den verwendeten Technologien vermittelt. Es gab jedoch auch Punkte, in denen die offizielle Dokumentation unzureichend schien um gewisse Funktionen der Software implementieren zu können.

Des Weiteren bietet die HSD-Card als Microservice-Architektur großes Potenzial an Erweiterungen und Verbesserungen. Die Zukunftsaussichten für eine Orchestrierung via Kubernetes sind positiv einzuschätzen. Zudem ist auch in Deutschland ein vermehrter Einsatz von Microservice-Architekturen und CD zu erwarten.

6. Verzeichnisse

6.1. Abbildungsverzeichnis

6.2. Literaturverzeichnis

Ardiansyah. 2018. Centralized Configuration Spring Application on Kubernetes. *medium.com.* [Online] 04. 11 2018. [Zitat vom: 18. 07 2019.] https://medium.com/@ard333/centralized-configuration-spring-application-on-kubernetes-4fd9e1a31f35.

Atlassian. 2019. www.atlassian.com. [Online] 2019. [Zitat vom: 20. 06 2019.] https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow.

Augusten, Stephan. 2017. dev-insider. *www.dev-insider.de.* [Online] 08. 12 2017. [Zitat vom: 29. 05 2019.] https://www.dev-insider.de/was-ist-continuous-delivery-a-664429/.

baeldung. 2019. Anleitung zu Spring 5 WebFlux. *baeldung.com.* [Online] 07. 07 2019. [Zitat vom: 18. 07 2019.] https://www.baeldung.com/spring-webflux.

Birk, Alexander und Lukas, Christoph. 2019. Eine Einführung in Continuous Delivery. *heise.de.* [Online] 08. 05 2019. [Zitat vom: 25. 07 2019.] https://www.heise.de/developer/artikel/Eine-Einfuehrung-in-Continuous-Delivery-Teil-1-Grundlagen-2176380.html?seite=all.

Blöchinger, Martin. 2019. trusted. [Online] 2019. [Zitat vom: 20. 06 2019.] https://trusted.de/versionsverwaltung.

Buehrle, Anita. 2018. Top 11 Continuous Delivery Tools for Kubernetes. *dzone.* [Online] 07. 07 2018. [Zitat vom: 29. 05 2019.] https://dzone.com/articles/top-11-continuous-delivery-tools-for-kubernetes-pa-1.

Calcott, Gary. 2018. silicon. *www.silicon.de.* [Online] 02. 03 2018. [Zitat vom: 29. 05 2019.] https://www.silicon.de/41666855/microservices-vs-monolithische-architekturen-ein-leitfaden.

cloud.google. 2019. PersistentVolumes mit nichtflüchtigem Speicher. *cloud.google.com.* [Online] 2019. [Zitat vom: 19. 07 2019.] https://cloud.google.com/kubernetes-engine/docs/concepts/persistent-volumes?hl=de.

Dinesh, Sandeep. 2018. Kubernetes NodePort vs LoadBalancer vs Ingress? When should I use what? *medium.com.* [Online] 11. 03 2018. [Zitat vom: 22. 07 2019.] https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0.

Docker. 2019. Best practices for writing Dockerfiles. *https://docs.docker.com.* [Online] 2019. [Zitat vom: 21. 06 2019.] https://docs.docker.com/develop/develop-images/dockerfile_best-practices/.

Documentation Kubernetes. 2019. Set up Ingress on Minikube with the NGINX Ingress Controller. *kubernetes.io.* [Online] 2019. [Zitat vom: 20. 06 2019.] https://kubernetes.io/docs/tasks/access-application-cluster/ingress-minikube/.

Elektronik Kompendium. 2019. TCP - Transmission Control Protocol. *elektronik-kompendium.de.* [Online] 31. 06 2019. [Zitat vom: 31. 07 2019.] https://www.elektronik-kompendium.de/sites/net/0812271.htm.

Ellis, Alex. 2017. Kubernetes on bare-metal in 10 minutes. *blog.alexellis.io.* [Online] 27. 06 2017. [Zitat vom: 27. 05 2019.] https://blog.alexellis.io/kubernetes-in-10-minutes/.

Fernuni-Hagen. 2019. 1678 Verteilte Systeme Zusammenfassung . *fernuni-hagen.de.* [Online] 2019. [Zitat vom: 21. 07 2019.] https://www.fernuni-hagen.de/FACHSCHINF/1678/1678_Zusammenfassung.pdf.

Gitlab. 2016. about.gitlab.com. [Online] 08. 05 2016. [Zitat vom: 29. 05 2019.] https://about.gitlab.com/2016/08/05/continuous-integration-delivery-and-deployment-with-gitlab/.

GitLab. 2019. Connecting GitLab with a Kubernetes cluster. *docs.gitlab.com.* [Online] 2019. [Zitat vom: 20. 05 2019.] https://docs.gitlab.com/ee/user/project/clusters/.

—. 2019. docs.gitlab.com. [Online] 2019. [Zitat vom: 21. 06 2019.] https://docs.gitlab.com/ee/ci/runners/README.html.

—. 2019. GitLab CI/CD environment variables. *docs.gitlab.com.* [Online] 2019. [Zitat vom: 21. 06 2019.] https://docs.gitlab.com/ee/ci/variables/.

—. 2019. GitLab CI/CD Pipeline Configuration Reference. *docs.gitlab.com.* [Online] 2019. [Zitat vom: 21. 06 2019.] https://docs.gitlab.com/ee/ci/yaml/.

Gitlab.com. 2019. doc.gitlab.com. [Online] 2019. [Zitat vom: 29. 05 2019.] https://docs.gitlab.com/ee/ci/pipelines.html.

—. 2019. GitLab Container Registry. *docs.gitlab.com.* [Online] 2019. [Zitat vom: 15. 06 2019.] https://docs.gitlab.com/ee/user/project/container_registry.html.

Helmich, Martin. 2018. mittwald.de. [Online] 2018. [Zitat vom: 20. 06 2019.] https://www.mittwald.de/blog/webentwicklung-design/webentwicklung/versionsverwaltung-einfuhrung-in-git-teil-1.

Kern, Bonny. 2017. Open Source Lizenzen - Die MIT-Lizenz. *wss-redpoint.com.* [Online] 15. 12 2017. [Zitat vom: 31. 07 2019.] https://wss-redpoint.com/open-source-lizenzen-die-mit-lizenz.

Kimmer, Cornelius. 2016. Was ist ein Message Broker? *blogs.sas.com/.* [Online] 25. 05 2016. [Zitat vom: 31. 07 2019.] https://blogs.sas.com/content/sasdach/2016/05/25/was-ist-ein-message-broker/.

kubernetes. 2019. Cluster-Vernetzung. *kubernetes.io.* [Online] 2019. [Zitat vom: 27. 05 2019.] https://kubernetes.io/docs/concepts/cluster-administration/networking/.

—. 2019. web-ui-dashboard. *kubernetes.io.* [Online] 2019. [Zitat vom: 27. 05 2019.] https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/.

Kubernetes-Customizing-DNS-Service. 2019. Customizing DNS Service. *kubernetes.io.* [Online] 2019. [Zitat vom: 06. 06 2019.] https://kubernetes.io/docs/tasks/administer-cluster/dns-custom-nameservers/.

Luber, Stefan und Augsten, Stephan. 2017. Was ist eine API? *dev-insider.de.* [Online] 08. 03 2017. [Zitat vom: 30. 07 2019.] https://www.dev-insider.de/was-ist-eine-api-a-583923/.

—. 2017. Was ist eine IDE? *dev-insider.de.* [Online] 10. 05 2017. [Zitat vom: 31. 07 2019.] https://www.dev-insider.de/was-ist-eine-ide-a-600703/.

Martincevic, Nino. 2016. DDD: Context is King – Kein Context, keine Microservices. *informatik-aktuell.de.* [Online] 22. 11 2016. [Zitat vom: 30. 07 2019.] https://www.informatik-aktuell.de/entwicklung/methoden/ddd-context-is-king-kein-context-keine-microservices.html.

Menk, Dr. Christoffer, Bindick, Dr. Sebastian und Augsten, Stephan. 2017. Maximale Flexibilität durch Microservices. *dev-insider.* [Online] 26. 10 2017. [Zitat vom: 11. 05 2019.] https://www.dev-insider.de/maximale-flexibilitaet-durch-microservices-a-654800/.

Mińkowski, Piotr. 2017. Kommunikation zwischen Microservices. *dzone.com.* [Online] 19. 12 2017. [Zitat vom: 01. 07 2019.] https://dzone.com/articles/communicating-between-microservices.

Morgan, Andrew. 2016. Running MongoDB as a Microservice with Docker and Kubernetes. *mongodb.com.* [Online] 04. 04 2016. [Zitat vom: 28. 06 2019.] https://www.mongodb.com/blog/post/running-mongodb-as-a-microservice-with-docker-and-kubernetes.

Mozilla. 2019. Cross-Origin Resource Sharing (CORS). *developer.mozilla.org.* [Online] 08. 07 2019. https://developer.mozilla.org/de/docs/Web/HTTP/CORS.

2015. Perforce. [Online] 2015. [Zitat vom: 29. 05 2019.] https://www.perforce.com/.

Perforce. 2014. *Continuous Delivery Report.* 02 2014.

projectreactor. 2019. *projectreactor.io.* [Online] 2019. [Zitat vom: 19. 07 2019.] https://projectreactor.io/.

redhat. 2019. what-is-kubernetes. *redhat.com.* [Online] 2019. [Zitat vom: 25. 06 2019.] https://www.redhat.com/de/topics/containers/what-is-kubernetes.

Schmitz, Peter. 2017. Was ist ein Hash? *security-insider.de.* [Online] 23. 08 2017. [Zitat vom: 30. 07 2019.] https://www.security-insider.de/was-ist-ein-hash-a-635712/.

skhatri. 2018. Insufficient memory / CPU? *github.com.* [Online] 15. 01 2018. [Zitat vom: 05. 05 2019.] https://github.com/kubernetes/minikube/issues/567.

spring.io. 2019. Class RestTemplate. *docs.spring.io.* [Online] 2019. [Zitat vom: 18. 07 2019.] https://docs.spring.io/spring/docs/current/javadoc-api/index.html?org/springframework/web/client/RestTemplate.html.

Thorsten Ebert, IT Beratung. 2014. hs-duesseldorf.de. [Online] 30. 04 2014. [Zitat vom: 27. 06 2019.] https://www.hs-duesseldorf.de/hochschule/verwaltung/gebaeudemanagement/neubau-dokumentation/Documents/5Chipkarte.pdf.

Velten, Carlo. 2019. crisp-research.com. *Kubernetes & Cloud Native Trends 2019.* [Online] 24. 01 2019. [Zitat vom: 24. 07 2019.] https://www.crisp-research.com/kubernetes-und-cloud-native-trends-2019-das-jahr-de/.

Vitz, Michael. 2016. Consumer-Driven Contracts – Testen von Schnittstellen innerhalb einer Microservices-Architektur. *innoq.com.* [Online] 26. 09 2016. [Zitat vom: 17. 07 2019.] https://www.innoq.com/de/articles/2016/09/consumer-driven-contracts/.

Weerasinghe, Sidath. 2018. dzone.com. [Online] 03. 09 2018. [Zitat vom: 10. 05 2019.] https://dzone.com/articles/monolithic-to-microservices.

Wikipedia. 2019. Hypertext Transfer Protocol. *Wikipedia.* [Online] 2019. [Zitat vom: 31. 07 2019.] https://de.wikipedia.org/wiki/Hypertext_Transfer_Protocol.

—. 2019. Model View Controller. *Wikipedia.* [Online] 2019. [Zitat vom: 31. 07 2019.] https://de.wikipedia.org/wiki/Model_View_Controller.

—. 2019. Repository. *Wikipedia.* [Online] 2019. [Zitat vom: 31. 07 2019.] https://de.wikipedia.org/wiki/Repository.

—. 2019. User Datagram Protocol. *Wikipedia.* [Online] 2019. [Zitat vom: 31. 07 2019.] https://de.wikipedia.org/wiki/User_Datagram_Protocol.

—. 2019. WebSocket. *Wikipedia.* [Online] 2019. [Zitat vom: 31. 07 2019.] https://de.wikipedia.org/wiki/WebSocket.

—. 2019. Wikipedia. [Online] 20. 06 2019. [Zitat vom: 20. 06 2019.] https://de.wikipedia.org/wiki/Versionsverwaltung.

Wolff, Eberhard. 2016. *Microservices - Grundlagen flexibler Softwarearchitekturen.* 1. Auflage. Heidelberg : d.punkt Verlag, 2016.

Wolff, Philipp-Sebastian. 2018. *Wissenschaftliche Vertiefung Continuous Delivery mit Microservices.* Düsseldorf : s.n., 2018.

6.3. Glossar

Begriff	Bedeutung
API	Die Abkürzung API steht für Application Programming Interface und bezeichnet eine Programmierschnittstelle. Die Anbindung erfolgt auf Quelltext-Ebene. APIs kommen in vielen Anwendungen zum Einsatz und werden im Webumfeld in Form von Web-APIs genutzt, siehe (Luber, et al., 2017).
Branch	Ein Branch in Git ist nichts anderes als ein simpler Zeiger auf einen Commit.
Bounded Context	Die Bedingungen, unter denen ein bestimmtes Modell definiert, klar und anwendbar ist. Primär ist der Bounded Context eine sprachliche Begrenzung. Eine Änderung in der Sprache der Domäne ist gleichzeitig eine Änderung des Modells, siehe (Martincevic, 2016).
Build	Eine Ansammlung von Schritten, wie etwa dem Kompilieren und Analysieren des Source Codes. Häufig gilt jeder Build als neue Version.
Bug Fix	Übersetzt bedeutet "Bug" soviel wie "Käfer". Als Bug bezeichnet man einen unbemerkten Fehler in einem Programm. Der Fix behebt diesen Fehler.
CA (Certificate)	CA (Certification Authority) ist in der Informationssicherheit eine Zertifizierungsstelle die digitale Zertifikate herausgibt.
CLI	CLI steht für Command Line Interface und bezeichnet die Kommandozeile.
Cluster	Cluster leitet sich vom englischen Wort für Bündel ab. In der Informatik steht ein Cluster für eine logische Zusammenfassung.
Code Review	Manuelle Prüfung der Arbeitsergebnisse der Softwareentwicklung durch eine andere Person.
Commit	Hinzufügen der letzten Änderungen am Source Code zum Version Control System.
CD-Pipeline	Wird innerhalb CI / CD in mehreren Schritten, sogenannten Stages, durchlaufen.
Deamon	Programm welches im Hintergrund abläuft und bestimmte Dienste zur Verfügung stellt.
Deployment	Das Übertragen von Software Artefakten in eine Produktivumgebung des Kunden oder das anderweitige Bereitstellen der Software für den Endnutzer.
Monolith	Bezeichnet in der IT eine als untrennbare gestaltete Einheit.

Begriff	Bedeutung
DevOps	Ein Kunstwort aus den Begriffen Development und Operations und bezeichnet Systemadministratoren in der Softwareentwicklung.
Discovery Service	Dient als Registrierungsstelle bzw. als API-Gateway für alle Dienste und kennt wie ein Telefonbuch alle Namen der Dienste und ihre IP-Adresse.
Fork	Das Abspalten einer Software mit separater Entwicklung weg vom eigentlichen Ursprung durch ein anderes Team.
Git-Repository-Manager	Dient zur Verwaltung von Git- Repositorys und bieten meist nützliche Zusatzfunktionen.
GUI	Ein Graphical User Interface, kurz GUI, beschreibt eine grafische Benutzeroberfläche mit deren Hilfe der Nutzer mit der Software interagieren kann. Dabei kann es sich beispielsweise um eine Webseite handeln, die als Schnittstelle für den Nutzer dient um mit dem dahinterliegenden Server interagieren zu können.
Hash	Als Hash oder Hashwert bezeichnet die Informatik die Ausgabe einer Hashfunktion, als Hash aber auch einen listenartigen Datentyp, bei dem der Zugriff auf die Elemente über deren Hashwert erfolgt, die Hashtabelle, siehe (Schmitz, 2017).
HSD-Card	Ein multifunktionaler elektronischer Studentenausweis der Hochschule Düsseldorf in Form einer Chip-Karte, dessen Einführungstermin noch unbekannt ist.
HTTP	Das Hypertext Transfer Protocol (HTTP, englisch für Hypertext-Übertragungsprotokoll) ist ein zustandsloses Protokoll zur Übertragung von Daten auf der Anwendungsschicht über ein Rechnernetz, siehe (Wikipedia, 2019).
IDE	Eine integrierte Entwicklungsumgebung oder IDE stellt Programmierern eine Sammlung der wichtigsten Werkzeuge zur Softwareentwicklung unter einer Oberfläche zur Verfügung. Die Arbeit für die Erstellung von Programmen wird dadurch vereinfacht, siehe (Luber, et al., 2017).
Image	Image (von engl. image für Bild, Abbild) bezeichnet in der Informationstechnik ein Abbild eines Systems oder einer Software.
Load-Balancing	Ein Verfahren zur regulierten Verteilung von Datenverkehr um Ressourcen besser zu nutzen.
Master	In der Informatik bezeichnet Master ein führendes System zur Steuerung anderer Systeme.

Begriff	Bedeutung
Merge	Zusammenführung von zwei unterschiedlichen Softwareversionen in einem VCS.
Message Broker	Ein Message Broker wird als Middle Ware bezeichnet und hat primär die Aufgabe, Nachrichten zu empfangen und an einen oder mehrere Empfänger weiterzuleiten, siehe (Kimmer, 2016).
MIT	Bei der MIT-Lizenz handelt es sich um eine der freizügigsten Open-Source-Lizenzen, weshalb sie kaum Beschränkungen oder Verpflichtungen für Nutzer enthält, siehe (Kern, 2017).
Mounten	(Engl. mount für befestigen) Bezeichnet das Einbinden oder Aktivieren eines Dateisystems an einer bestimmten Stelle im System.
MVC	Model View Controller (MVC, englisch für Modell-Präsentation-Steuerung) ist ein Muster zur Unterteilung einer Software in die drei Komponenten Datenmodell (englisch model), Präsentation (englisch view) und Programmsteuerung (englisch controller),siehe (Wikipedia, 2019).
Platform as a Service (PaaS)	Ein Cloud-Modell, in dem Anbieter Entwicklungstools, Datenverwaltung und ein Betriebssystem zur Verfügung stellen. Darunterliegende Hardware, Netzwerksicherheit und Fläche im Rechenzentrum wird vom Anbieter verwaltet.
Push	Hinzufügen eines Softwareabbilds oder einer Softwareänderung in ein entferntes System (Server).
REST	REST steht für „Representational State Transfer“, orientiert sich am Verhalten des World Wide Web (WWW) und beschreibt einen Ansatz für die Kommunikation zwischen Client und Server in Netzwerken.
Repository	Ein Repository (englisch für Lager, Depot oder auch Quelle; Plural: Repositorys), auch – direkt aus dem Lateinischen entlehnt – Repositorium (Pl. Repositorien), ist ein verwaltetes Verzeichnis zur Speicherung und Beschreibung von digitalen Objekten für ein digitales Archiv, siehe (Wikipedia, 2019).
Rollout / Release	Ein Rollout / Release ist die Veröffentlichung oder die Zurverfügungstellung einer Software für den Nutzer.
Runner	Bezeichnet einen Agenten zum Ausführen von Stages innerhalb der CD-Pipeline.

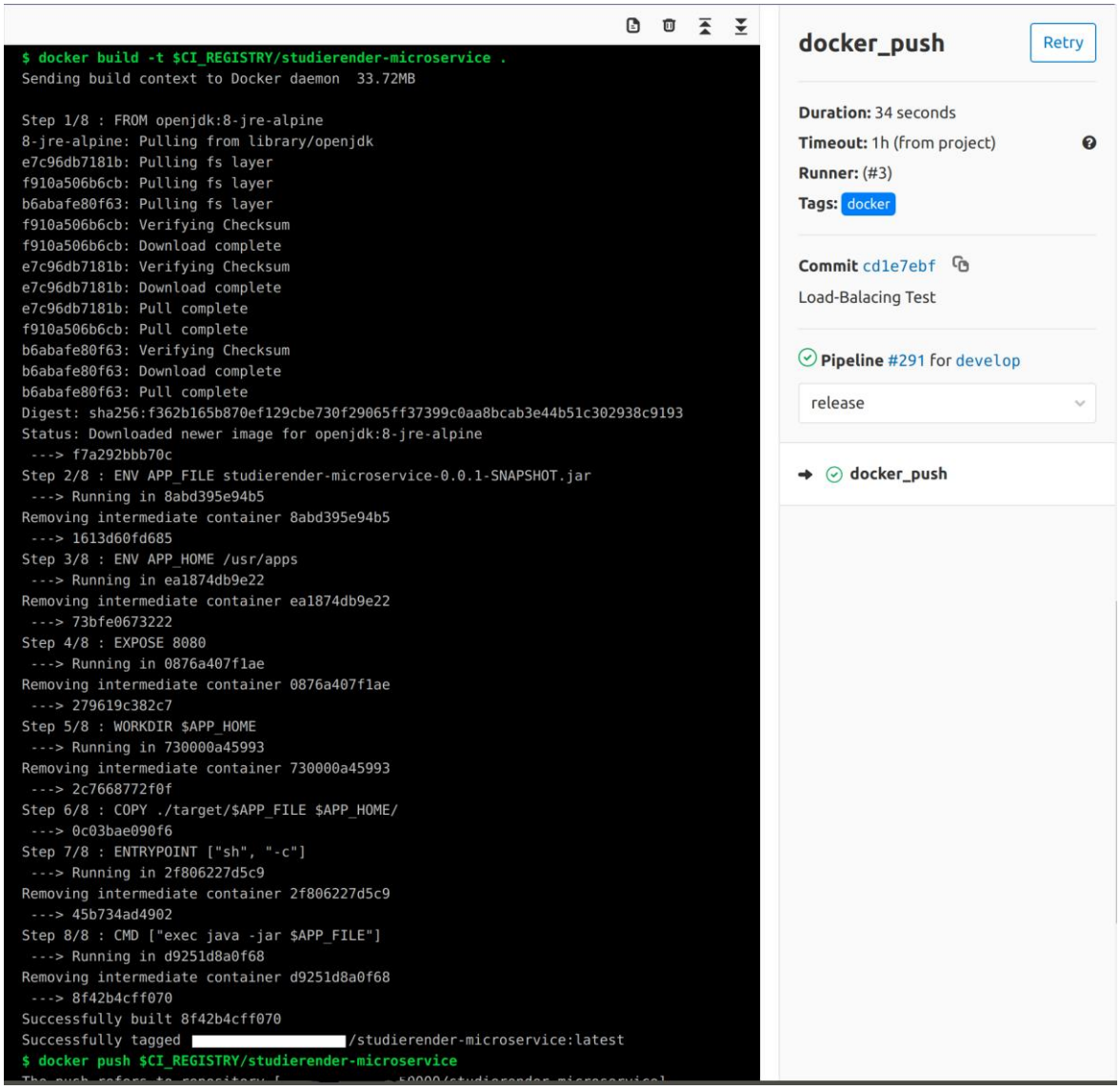
Begriff	Bedeutung
Servlet	Bezeichnet Java-Klassen, deren Instanzen Anfragen von Clients im Web-Umfeld entgegen nehmen.
Stage	Bezeichnet einen Schritt in der CD-Pipeline.
Strong Typing	Bedeutet strenge oder starke Typisierung und ist ein Prinzip in Programmiersprachen, bei denen Variablen exakt einen Datentyp besitzen müssen.
Software Design Pattern / Pattern	Entwurfsmuster, welche als bewährte Lösungsschablonen für wiederkehrende Probleme genutzt werden können.
Spring Boot	Spring Boot gehört zum Spring Framework für Java-Plattformen und wird oft im Web-Umfeld eingesetzt.
Tag	Markiert einen Datenbestand oder einen Teil einer Software mit zusätzlichen Informationen.
TCP	Das Transmission Control Protocol, kurz TCP, ist Teil der Protokollfamilie TCP/IP. TCP ist ein verbindungsorientiertes Protokoll und soll maßgeblich Datenverluste verhindern, Dateien und Datenströme aufteilen und Datenpakete den Anwendungen zuordnen können, siehe (Elektronik Kompendium, 2019).
Token	Folge generierter zusammengehöriger Zeichen und wird meist zur Authentifizierung genutzt.
UDP	Das User Datagram Protocol, kurz UDP, ist ein minimales, verbindungsloses Netzwerkprotokoll, das zur Transportschicht der Internetprotokollfamilie gehört. UDP ermöglicht Anwendungen den Versand von Datagrammen in IP-basierten Rechnernetzen, siehe (Wikipedia, 2019).
Version Control System (VCS)	Ein System zum Aufzeichnen von Änderungen an einer Datei oder einem Satz von Dateien über die Zeit. Somit kann auf die komplette Historie zugegriffen werden. Diese dient der Dokumentation der Änderungen und stellt die Möglichkeit bereit auf ältere Versionen der Dateien zuzugreifen.
Volume	Gemeint ist ein Speichervolumen als identifizierbare Einheit zur Datenspeicherung. Es kann physisch als auch virtuell ausgetauscht werden.
Websocket	Das WebSocket-Protokoll ist ein auf TCP basierendes Netzwerkprotokoll, das entworfen wurde, um eine bidirektionale Verbindung zwischen einer Webanwendung und einem

Begriff	Bedeutung
	WebSocket-Server bzw. einem Webserver, der auch WebSockets unterstützt, herzustellen, siehe (Wikipedia, 2019).
Working Directory	Meint das lokale Verzeichnis, in dem ein Prozess im Dateisystem arbeitet.

6.4. Anhang

A. Screenshots

A.1. Stage Package Docker Build & Push



Screenshot 1 Stage Release Ausführen des DockerFiles durch docker build und anschließendem docker push in die Docker Registry

A.2. Kubernetes Information

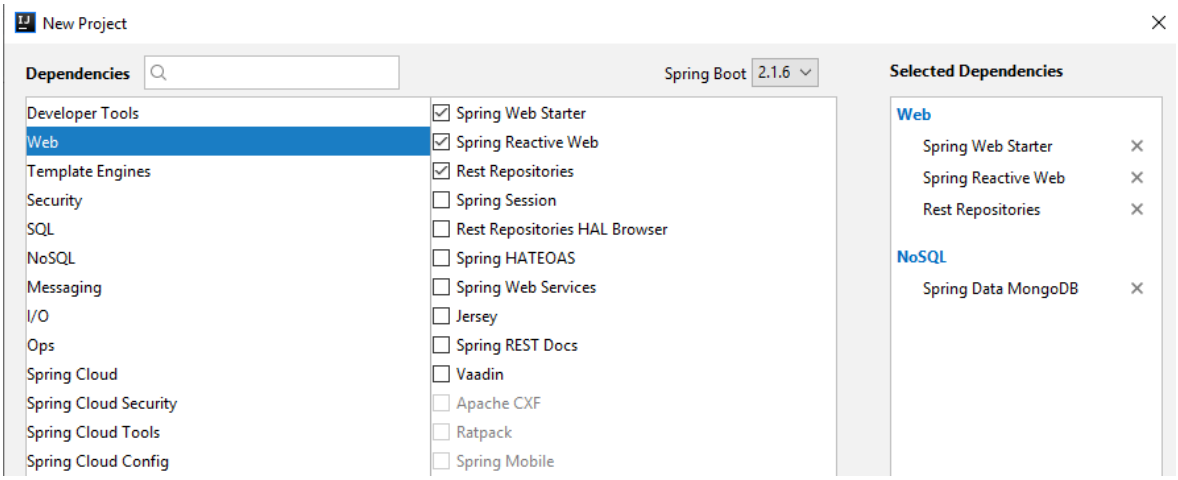
```
ubuntu@ubuntu-desktop:~$ kubectl apply -f Schreibtisch/gitlab-admin-service-account.yaml
Warning: kubectl apply should be used on resource created by either kubectl create --save-config or kubectl apply
serviceaccount/gitlab-admin configured
Warning: kubectl apply should be used on resource created by either kubectl create --save-config or kubectl apply
clusterrolebinding.rbac.authorization.k8s.io/gitlab-admin configured
ubuntu@ubuntu-desktop:~$  kubectl -n kube-system describe secret $(kubectl -n kube-system get secret | grep gitlab-admin | awk '{print $1
}')
Name:         gitlab-admin-token-p9r7f
Namespace:    kube-system
Labels:       <none>
Annotations:  kubernetes.io/service-account.name=gitlab-admin
              kubernetes.io/service-account.uid=30071d26-70d9-11e9-9a57-0800270ec843

Type:  kubernetes.io/service-account-token

Data
====
namespace:  11 bytes
token:      eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1l
c3BhY2UiOiJrdWJlLXN5c3RlbSIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VjcmV0Lm5hbWUiOiJnaXRsYWItYWRtaW4tdG9rZW4tcDlyN2YiLCJrdWJlcm5ldGVzL
mlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC5uYW1lIjoiZ2l0bGFiLWFkbWluIiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZXJ2aWNlLWFjY291bnQudW
lkIjoiMzAwNzFkMjYtNzBkOS0xMWU5LTlhNTctMDgwMDI3MGVjODQzIiwic3ViIjoic3lzdGVtOnNlcnZpY2VhY2NvdW50Omt1YmUtc3lzdGVtOmdpdGxhYi1hZG1pbiJ9.duOLcU
xddAPIqiZEtgjMdJTSa7vrBV0U24uSwJdSJlwjLFV6sWB6YnmGGBl47BCzIWUtaoxN3L-fWaZlO-4BZdM0udtH5Z9nMHzjwwuZBxhJ6eOa3xDTsjkgA_7j2_rmwlXkQLREjzGwVmJ
2l3TArP5Nmnx1UWQnZtYOvUviROaBWcrC9PU3lPjkG0EBDN9ybmQguEuLL1YYw3RGk3LvpTeXgMPvKrTAu22EtTYoGuPrHsU4GIHciXqlG6D83ib62AdCOIYfSb3vnQBzjZtaquCV
srMSCNhy2X9Pe5qpqD_3V5hZeQKMIRu9uxL6UdLIKJWNGVSCSAgj28UP4v5YFA
ca.crt:     1066 bytes
ubuntu@ubuntu-desktop:~$ kubectl get secrets
NAME                  TYPE                                  DATA      AGE
default-token-g6kdt   kubernetes.io/service-account-token   3         151d
gitlab-token-4tkq5    kubernetes.io/service-account-token   3         141d
mongodb               Opaque                                2         2h
ubuntu@ubuntu-desktop:~$ kubectl get secret default-token-g6kdt -o jsonpath="{['data']['ca\.crt']}" | base64 --decode
-----BEGIN CERTIFICATE-----
MIIC5zCCAc+gAwIBAgIBATANBgkqhkiG9w0BAQsFADAVMRMwEQYDVQQDEwptaW5p
a3ViZUNBMB4XDTE4MTEyMDE3MTE0NVoXDTI4MTExODE3MTE0NVowFTETMBEGA1UE
AxMKbWluaWt1YmVDQTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALTx
4gI9Tgek0cCvQsS0nuQ8n3kYGMcxi0fsv7fjRawdWuleGy/HJCCxDi3hrKGpXqYD
EDMNIALHI+4KEQplJQkD2mFgULZkekSsUOgSXipKXiZWGyXzG1QHEwpgk01p9ZXS
z8FGIQGes6ec31K6PQOPgErxGoAIawU6lsG/jwJj4BPzoT5UnctrE/mQe7ItnWzt
iV2k5lg2IMf/DyG07o5UmNQPFj6ukrsT7hG2f9F1mEMpOyGpU90WIK0eRuo8GE0h
itiVCIs9CUYSWco68Xod+rCngk3BOeMUkpfOxSAOZhc8HZHTIb+ZUyw8FIJm1GtG
mJBsuQX25/btESRVOAECAwEAAaNCMEAwDgYDVR0PAQH/BAQDAgKkMB0GA1UdJQQW
MBQGCCsGAQUFBwMCBggrBgEFBQcDATAPBgNVHRMBAf8EBTADAQH/MA0GCSqGSIb3
DQEBCwUAA4IBAQCfWgnEYcTFUwER9R7WN5P/++HV9ghGjf/gCxHVXTjBKV+gR79b
OQyoG6eZjHUswDPBB5IADyydRANT0dmFg3Ol+GNiVd9pts5MEoSC5x2AxmmdPOlq
kzrotmVgdHvBnbhcFAneN6wzI+8Uuyaf9YuH9V9TVLlI2sXlMKLGvYQfpmecQX+l
KVkt8c4uBUgPGmp80kqOw4NiIu5LM9rL6EzhnjtDH28xXtlNJZ2PPE2Hu3yNm2qH
1rK23UtdKLgQDSGIcL6ZU/cDUSzErnsMKt4VbrRGofzs2PxFAGWYh18NdZs+EbSt
Bfy7F/K35THCOOf39WrPvNXtTqsqkrbDpme+
-----END CERTIFICATE-----
```

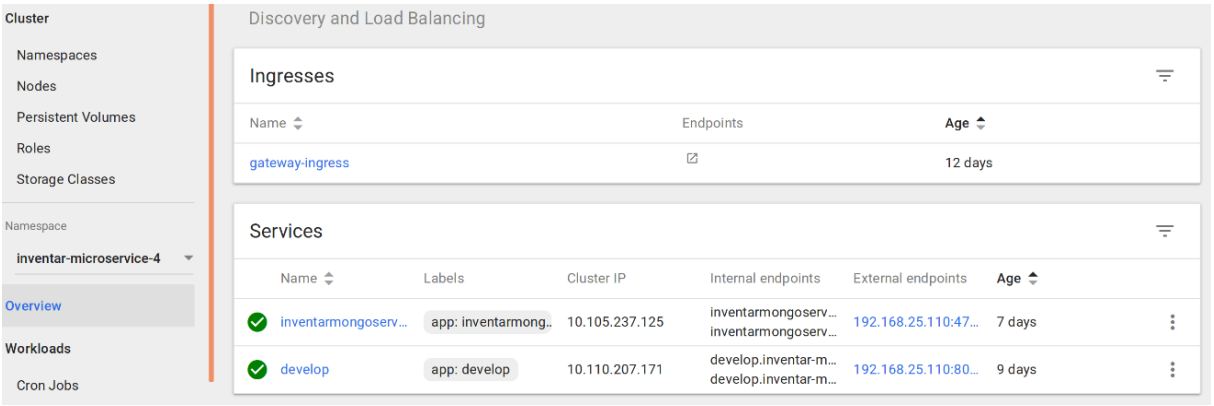
Screenshot 2 Informationen aus Kubernetes zum Verbinden mit GitLab

A.3. Spring Boot Dependencies



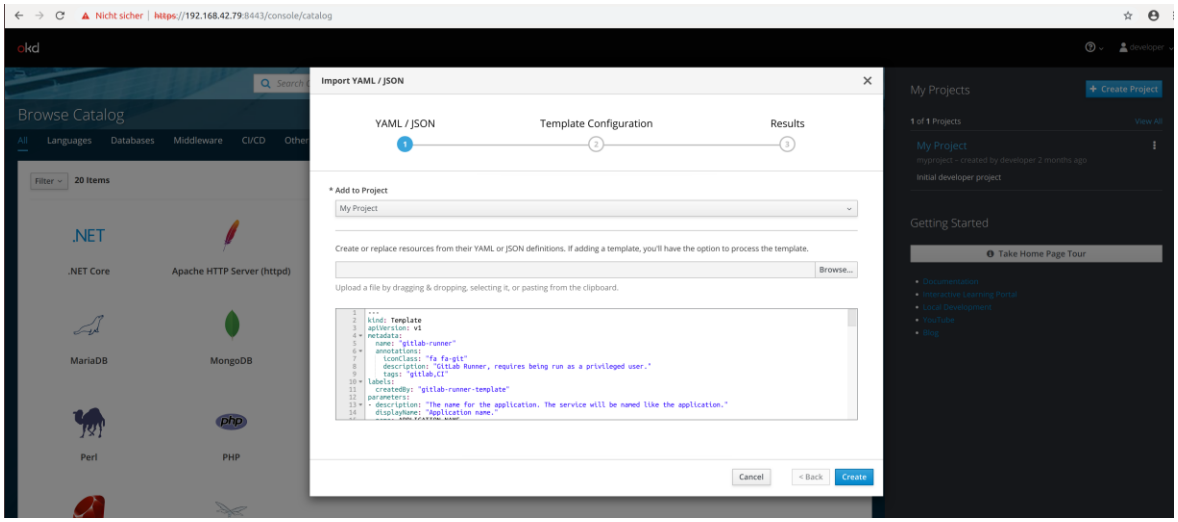
Screenshot 3 Spring Boot Dependencies

A.4. Service Discory & Load-Balancing



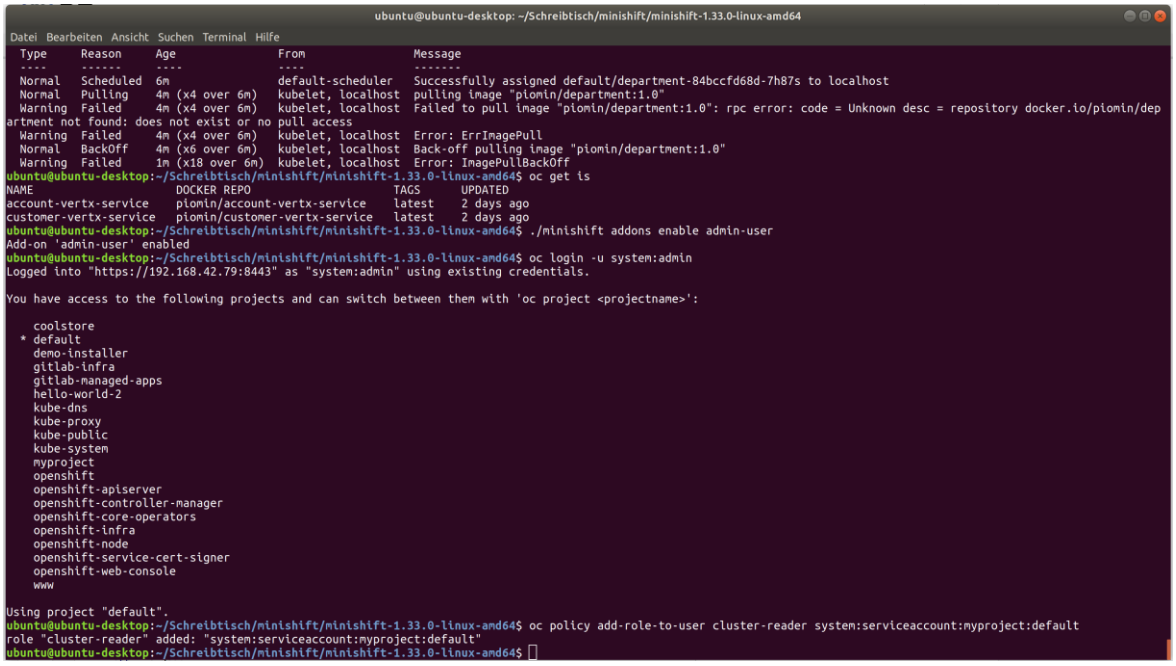
Screenshot 4 Discovery Service & Load-Balancing

A.5. GitLab-Runner Template Opfenshift



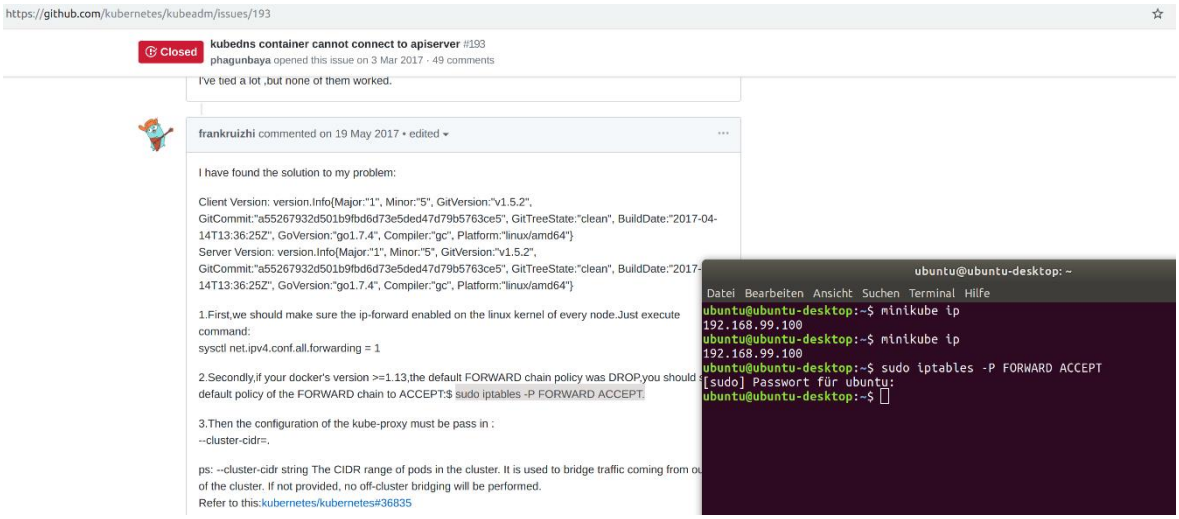
Screenshot 5 Erstellen eines GitLab-Runner Templates in Openshift

A.6. Definition Admin-Rolle für Privilige Mode in Openshift



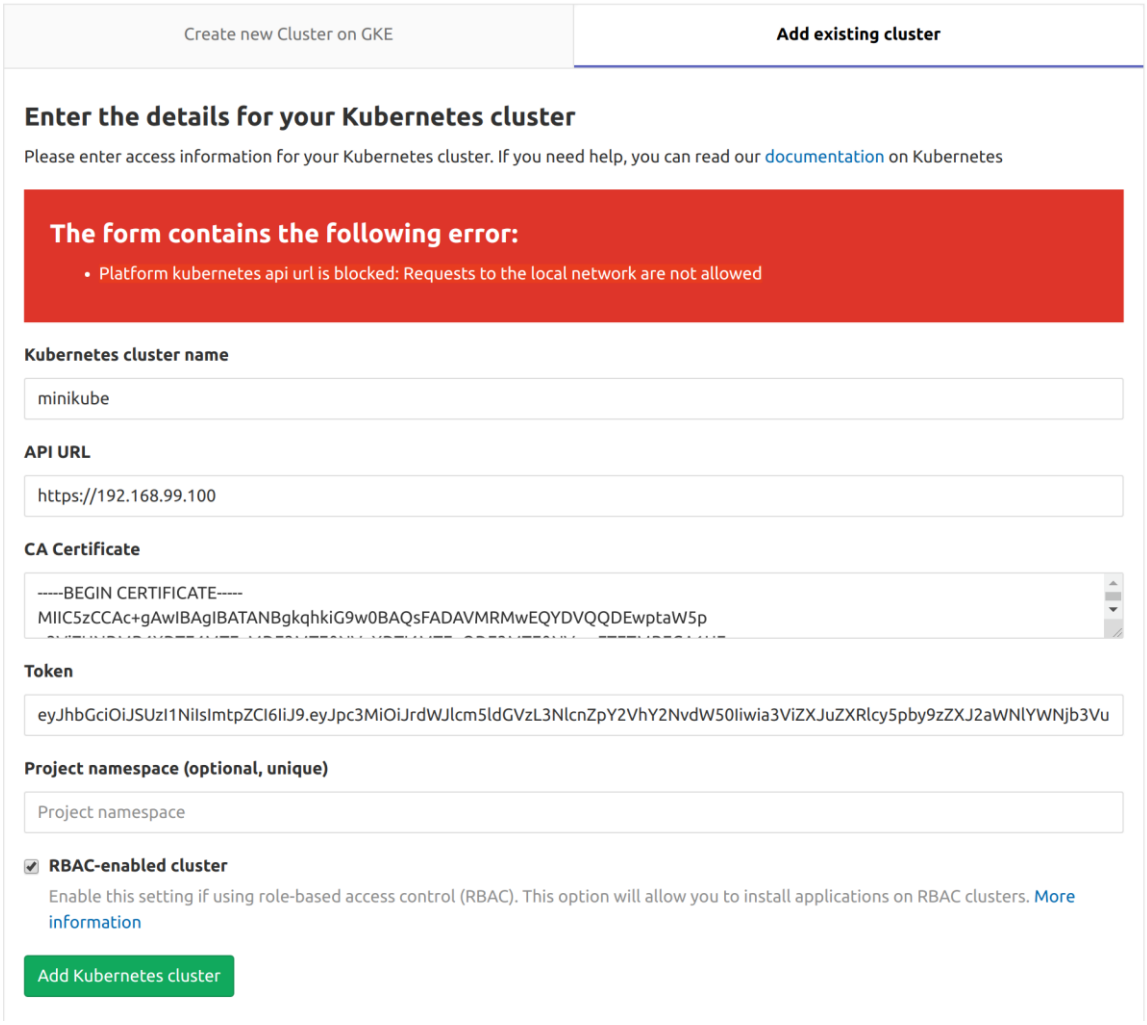
Screenshot 6 Vergeben einer Admin-Rolle in Openshift um in Container im Privilege Mode zu starten

A.7. Firewalleinstellungen



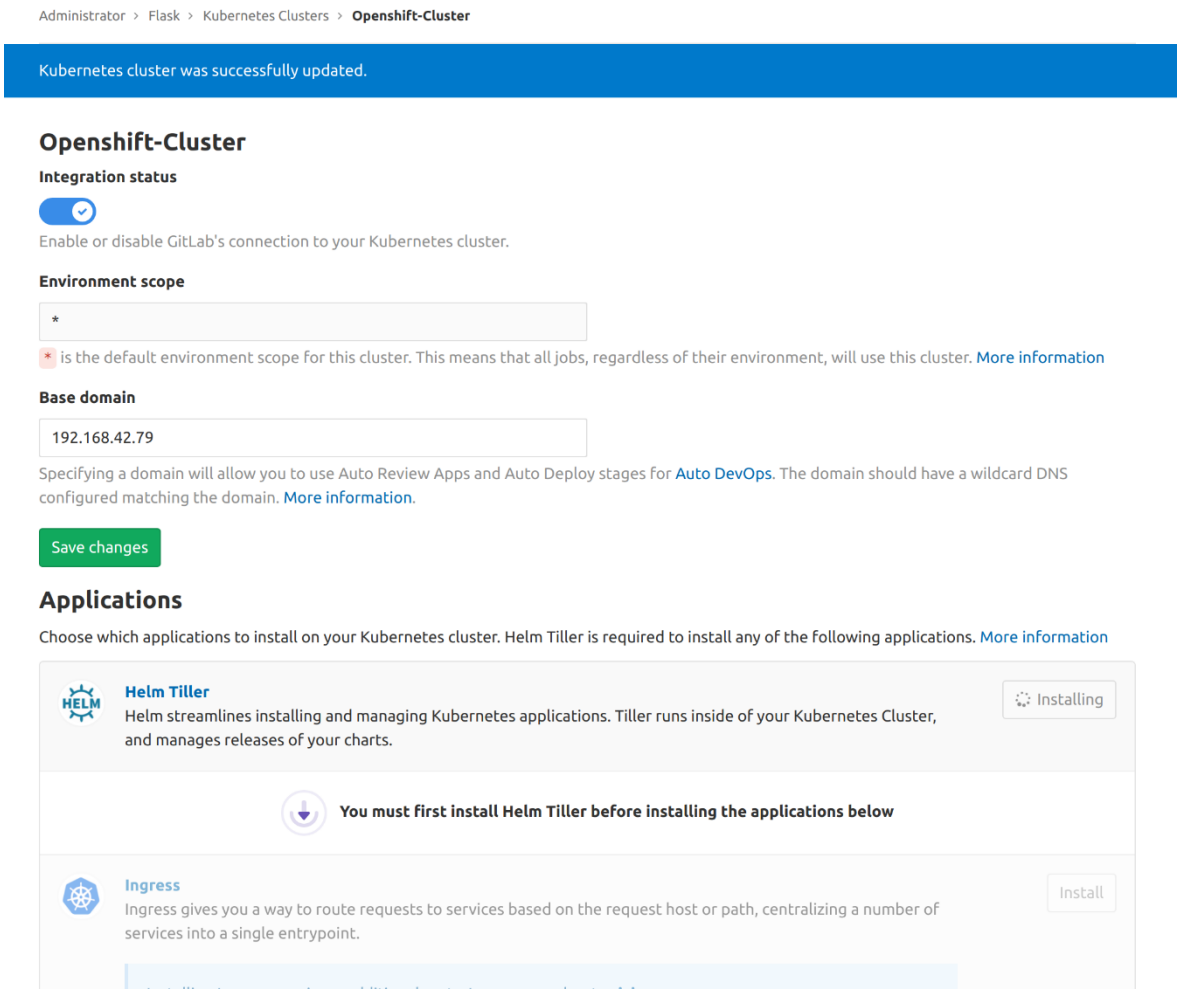
Screenshot 7 Firewalleinstellungen anpassen damit mit der Kubernetes API kommuniziert werden kann

A.8. Kubernetes API URL wird im lokalen Netwerk geblockt



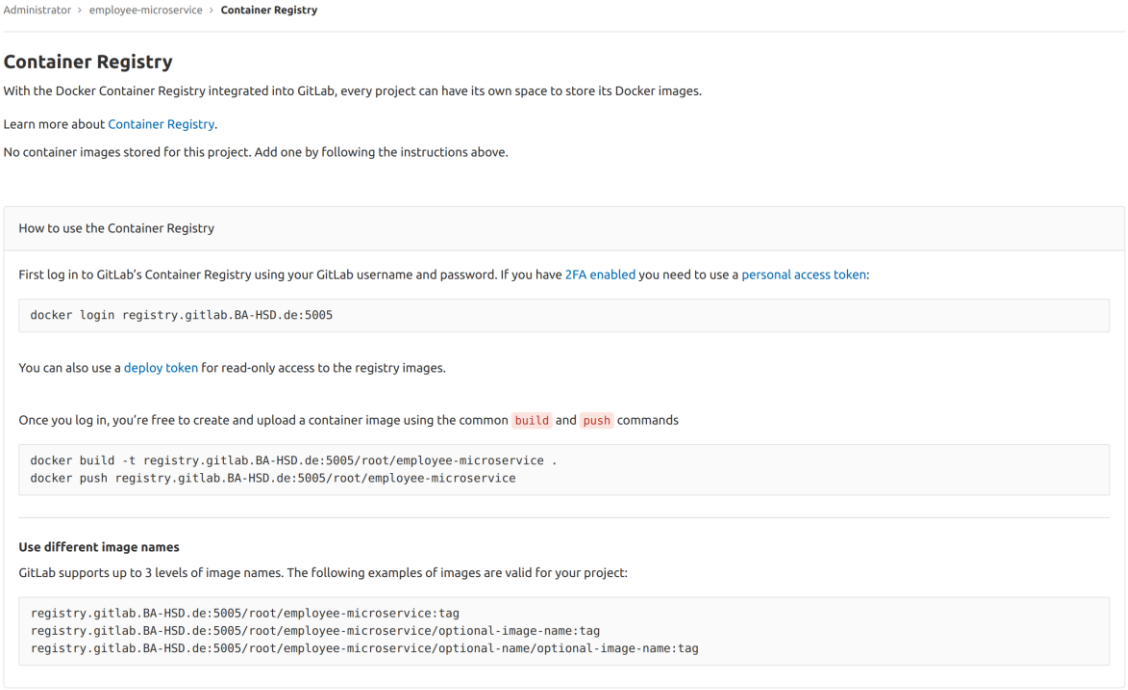
Screenshot 8 Kubernetes API URL wird im lokalen Netzwerk blockiert

A.9. Eingabe der Base Domain für Openshift



Screenshot 9 Eine Base Domain (API IP) sollte für Openshift eingetragen werden.

A.10. GitLab Container Registry



Screenshot 10 GitLab-Container Registry über Web-Oberfläche

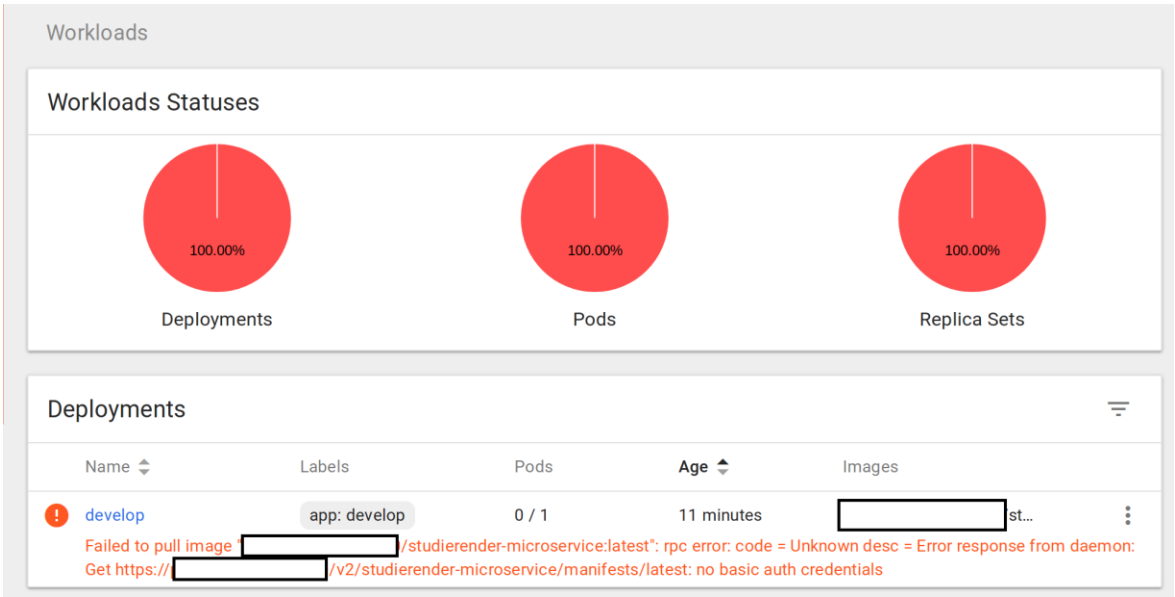
A.11. Gitlab.rb

Screenshot 11 gitlab.rb Konfigurationsdatei

A.12. Kubernetes Core-DNS Config-Map Corefile

Screenshot 12 Corefile der Config-Map des Core-DNS bearbeitet mit JSON-Editor-Online

A.13. No basic auth credentials



Screenshot 13 "no basic auth credentials" erscheint wenn im Deployment das Secret nicht angeben wurde

A.14. SSL-Zertifikat ungültig oder abgelaufen

Screenshot 14 CI Job schlägt fehlt wenn das Zertifikat der Docker Registry abgelaufen oder ungültig ist

B. Listings

B.1. Gitlab-ci.yaml

```
stages:
  - build
  - test
  - release
  - deploy

# Wenn dind verwendet wird, ist es ratsam den Overlayfs-Treiber für
verbesserte Leistung zu verwenden.

variables:
  MAVEN_OPTS: "-Dmaven.repo.local"
  DOCKER_DRIVER: overlay2
  DOCKER_HOST: "tcp://localhost:2375"
  INSTALL_DIR: $CI_PROJECT_DIR/maven
cache:
  paths:
    - ./.m2/repository
  key: "$CI_BUILD_REF_NAME"
```

```
maven-build:
  image: maven:3-jdk-8
  stage: build
  script: "mvn clean install"
  artifacts:
    paths:
     - target/*.jar
  tags:
    - docker

test:
  stage: test
  image: maven:latest
  script:
    - mvn $MAVEN_CLI_OPTS test

services:
  - name: docker:dind
    entrypoint: ["dockerd-entrypoint.sh"]
    command: ["--insecure-registry", "gitlab.ba-hsd.de"]

docker_push:
  stage: release
  image: docker:dind
  script:
    - docker info
    - docker login -u root -p $password $CI_REGISTRY
    - echo $CI_REGISTRY_USER $CI_REGISTRY_PASSWORD $CI_REGISTRY test
    - docker build -t $CI_REGISTRY/studierender-microservice .
    - docker push $CI_REGISTRY/studierender-microservice
  dependencies:
    - maven-build
  tags:
    - docker

deploy_test:
  image: lachlanevenson/k8s-kubectl:latest
  stage: deploy
  environment:
    name: studierender-microservice
    url: https://192.168.25.110:6443
  when: manual
  script:
    - kubectl version
    - sed -i "s/__CI_BUILD_REF_SLUG__/${CI_BUILD_REF_SLUG}/"
studierender-deployment.yaml
    - sed -i "s/  VERSION  /${CI COMMIT REF NAME}/" studierender-
deployment.yaml
    - echo ${CI_PROJECT_NAME}  __VERSION__ ${CI_COMMIT_REF_NAME}
${CI_ENVIRONMENT_SLUG}
    - (kubectl create -f studierender-deployment.yaml || (kubectl delete
-f studierender-deployment.yaml && kubectl apply -f studierender-
deployment.yaml))
    - kubectl apply -f ./kubernetes/mongoDBsecret.yaml
    - (kubectl create -f ./kubernetes/studentMongoDBController.yaml ||
(kubectl delete -f ./kubernetes/studentMongoDBController.yaml && kubectl
apply -f ./kubernetes/studentMongoDBController.yaml))
    - kubectl apply -f ./kubernetes/studentMongoDBService.yaml
  tags:
```

```
    - kubernetes

# Wenn eine Produktivumgebung vorhanden wäre, dann würde diese Stage
zusätzlich ausgeführt werden.
#deploy_prod:
#  image: lachlanevenson/k8s-kubectl:latest
#  stage: deploy
#  environment:
#    name: studierender-microservice
#    url: FQDN der Produktion
#  when: manual
#  script:
#    - kubectl version
#    - sed -i "s/__CI_BUILD_REF_SLUG__/${CI_BUILD_REF_SLUG}/"
studierender-deployment.yaml
#    - sed -i "s/  VERSION  /${CI COMMIT REF NAME}/" studierender-
deployment.yaml
#    - echo ${CI_PROJECT_NAME}  __VERSION__ ${CI_COMMIT_REF_NAME}
${CI_ENVIRONMENT_SLUG}
#    - (kubectl create -f studierender-deployment.yaml || (kubectl delete
-f studierender-deployment.yaml && kubectl apply -f studierender-
deployment.yaml))
#    - kubectl apply -f ./kubernetes/mongoDBsecret.yaml
#    - (kubectl create -f ./kubernetes/studentMongoDBController.yaml ||
(kubectl delete -f ./kubernetes/studentMongoDBController.yaml && kubectl
apply -f ./kubernetes/studentMongoDBController.yaml))
#    - kubectl apply -f ./kubernetes/studentMongoDBService.yaml
#  tags:
#    - produktion
```

Listing 1 gitlab-ci.yml (studierender-microservice)

B.2. Dockerfile

```
FROM openjdk:8-jre-alpine
ENV APP_FILE studierender-microservice-0.0.1-SNAPSHOT.jar
ENV APP_HOME /usr/apps
EXPOSE 8080
WORKDIR $APP_HOME
COPY ./target/$APP_FILE $APP_HOME/
ENTRYPOINT ["sh", "-c"]
CMD ["exec java -jar $APP_FILE"]
```

Listing 2 Dockerfile (Studierender-Microservice)

B.3. Deployment.yaml (studierender-microservice)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: __CI_BUILD_REF_SLUG__
  labels:
    app: __CI_BUILD_REF_SLUG__
spec:
  replicas: 1
  selector:
    matchLabels:
      app: __CI_BUILD_REF_SLUG__
  template:
    metadata:
      labels:
```

```
        app: __CI_BUILD_REF_SLUG__
    spec:
      imagePullSecrets:
        - name: registry-secret
      containers:
      - name: studierender
        image: docker-registry:5000/studierender-microservice:latest
        ports:
        - containerPort: 8080
        env:
          - name: MONGO_USERNAME
            valueFrom:
              secretKeyRef:
                name: mongo-secret
                key: user
          - name: MONGO_PASSWORD
            valueFrom:
              secretKeyRef:
                name: mongo-secret
                key: password
---
apiVersion: v1
kind: Service
metadata:
  name: __CI_BUILD_REF_SLUG__
  labels:
    app: __CI_BUILD_REF_SLUG__
spec:
  type: LoadBalancer
  ports:
  - port: 8080
    protocol: TCP
    nodePort: 30081
  externalIPs: [192.168.25.110]
  selector:
    app: __CI_BUILD_REF_SLUG__
```

Listing 3 Deployment.yaml (studierender-deployment.yaml)

B.4. Controller – Bestellcontroller.java

```
package hsdcard.studierendermicroservice.Controller;

import com.fasterxml.jackson.databind.ObjectMapper;
import hsdcard.studierendermicroservice.CopiedModel.Artikel;
import hsdcard.studierendermicroservice.Model.Bestellung;
import hsdcard.studierendermicroservice.Repository.BestellRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.http.converter.ResourceHttpMessageConverter;
import org.springframework.web.bind.annotation.*;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.util.Date;
import java.util.List;

@RestController
```

```
@RequestMapping("/bestellung")
public class BestellController {
    private BestellRepository bestellRepository;

    @Autowired
    private WebClient.Builder webClientBuilder;

    @Value("${spring.data.mongodb.host}")
    private String Host;

    public BestellController(BestellRepository bestellRepository) {
        this.bestellRepository = bestellRepository;
    }

    @GetMapping("/all")
    public List<Bestellung> getAll(){
        List<Bestellung> Bestellungen = this.bestellRepository.findAll();
        return Bestellungen;
    }

    @GetMapping("/matrikelnummer")
    public List<Bestellung>
getByMatrikelNummer(@PathVariable("matrikelnummer") String
matrikelnummer){
        List<Bestellung> bestellungen =
this.bestellRepository.findByMatrikelNummer(matrikelnummer);
        return bestellungen;
    }

    /**
     * Gibt eine Liste von Artikeln des Inventar-Services zurück
     * @return Flux<Artikel>
     */
    @GetMapping("/artikel")
    public Flux<Artikel> getAllArtikel(){

        Flux<Artikel> artikelFlux = webClientBuilder.build()
                .get()
                .uri("http://"+Host+":8082/artikel/all")
                .accept(MediaType.APPLICATION_JSON)
                .retrieve()
                .bodyToFlux(Artikel.class);

        return artikelFlux;
    }

    /*
    Erstellen einer neuen Bestellung
    Ruft dafür den bestellten Artikel auf um diesen in die Bestellung auf
zunehmen
     */
    @PostMapping("/{artikelID}/{matrikelNummer}/{anzahl}")
    public void setNeueBestellung(@PathVariable("artikelID")String
artikelID, @PathVariable("matrikelNummer") String matrikelNummer,
    @PathVariable("anzahl")int anzahl){
        Bestellung bestellung = new Bestellung();

        Mono<Artikel> artikelMono = webClientBuilder.build()
                .get()
                .uri("http://"+Host+":8082/artikel/"+artikelID)
```

```
                .accept(MediaType.APPLICATION_JSON)
                .exchange()
                .flatMap(clientResponse ->
clientResponse.bodyToMono(Artikel.class));

        bestellung.setAnzahl(anzahl);
        bestellung.setArtikelID(artikelID);
        bestellung.setPreis((artikelMono.block().getPreis() * anzahl));
        bestellung.setMatrikelNummer(matrikelNummer);
        bestellung.setStatus("offen");
        bestellung.setDatum(new Date().toString());

       this.bestellRepository.insert(bestellung);

    }
}
```

Listing 4 BestellController.java (Studierender-Microservice)

B.5. CopiedModel – Artikel.java

```
package hsdcard.studierendermicroservice.CopiedModel;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Document(collection = "artikel")
public class Artikel {
    @Id
    private String artikelID;
    private String  artikelName;
    private float preis;
    private String kategorie;

    public Artikel(){}

    public Artikel(String artikelID, String artikelName, float preis,
String kategorie) {
        this.artikelID = artikelID;
        this.artikelName = artikelName;
        this.preis = preis;
        this.kategorie = kategorie;
    }

    public String getArtikelID() {
        return artikelID;
    }

    public String getArtikelName() {
        return artikelName;
    }

    public float getPreis() {
        return preis;
    }

    public String getKategorie() {
        return kategorie;
    }
}
```

Listing 5 CopiedModel Artikel.java

B.6. Model Bestellungen.java

```
package hsdcard.studierendermicroservice.Model;

import org.springframework.data.mongodb.core.mapping.Document;
import org.springframework.format.annotation.DateTimeFormat;

import java.util.Date;

@Document(collection = "bestellung")
public class Bestellung {
    private String artikelID;
    private String matrikelNummer;
    private float preis;
    private String status;
    @DateTimeFormat(iso= DateTimeFormat.ISO.DATE_TIME)
    private String datum;
    private int anzahl;

    public Bestellung(){}

    public Bestellung(String artikelID, String matrikelNummer, float
preis, String datum, String status, int anzahl) {
        this.artikelID = artikelID;
        this.matrikelNummer = matrikelNummer;
        this.preis = preis;
        this.datum = datum;
        this.status = status;
        this.anzahl = anzahl;
    }

    public int getAnzahl() {
        return anzahl;
    }

    public String getStatus() {
        return status;
    }

    public String getArtikelID() {
        return artikelID;
    }

    public String getMatrikelNummer() {
        return matrikelNummer;
    }

    public float getPreis() {
        return preis;
    }

    @DateTimeFormat(iso= DateTimeFormat.ISO.DATE_TIME)
    public String getDatum() {
        return datum;
    }

    public void setArtikelID(String artikelID) {
        this.artikelID = artikelID;
    }
```

```
    public void setMatrikelNummer(String matrikelNummer) {
        this.matrikelNummer = matrikelNummer;
    }

    public void setPreis(float preis) {
        this.preis = preis;
    }

    public void setStatus(String status) {
        this.status = status;
    }

    public void setDatum(String datum) {
        this.datum = datum;
    }

    public void setAnzahl(int anzahl) {
        this.anzahl = anzahl;
    }
}
```

Listing 6 Model Bestellung.java

B.7. Repository BestellRepository.java

```
package hsdcard.studierendermicroservice.Repository;

import hsdcard.studierendermicroservice.CopiedModel.Artikel;
import hsdcard.studierendermicroservice.Model.Bestellung;
import org.springframework.data.mongodb.repository.MongoRepository;

import java.util.List;

public interface BestellRepository extends
MongoRepository<Bestellung,String> {
 List<Bestellung> findByMatrikelNummer(String matrikelnummer);
 Artikel  findByArtikelID(String artikelID);
}
```

Listing 7 Repository BestellRepository.java

B.8. WebConfig.java

```
package hsdcard.studierendermicroservice.WebConfig;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**");
    }
}
```

Listing 8 WebConfig.java, notwendig für CORS

B.9. ProjectnameApplication.java

```
package hsdcard.studierendermicroservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.reactive.function.client.WebClient;

@SpringBootApplication

public class StudierenderMicroserviceApplication {

@Bean
public WebClient.Builder getWebClientBuilder(){
    return WebClient.builder();
}

    public static void main(String[] args) {
        SpringApplication.run(StudierenderMicroserviceApplication.class,
args);
    }

}
```

Listing 9 StudierenderMicroserviceApplication.java Instanziert die Bean

B.10. Application.properties

```
spring.cloud.kubernetes.reload.enabled=true
spring.cloud.kubernetes.secrets.name=mongo-secret
spring.data.mongodb.host=192.168.25.110
spring.data.mongodb.port=47863
spring.data.mongodb.database= studentDB
spring.data.mongodb.username=${MONGO_USERNAME}
spring.data.mongodb.password=${MONGO_PASSWORD}
```

Listing 10 Festlegen von globalen Variablen in der application.properties

B.11. Kubernetes Secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: mongo-secret
data:
  user: cm9vdA==
  password: Q0R1c2VyMTUwMQ==
```

Listing 11 Secret für MongoDB

B.12. Kubernetes PersistentVolume.yaml

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: student-pv-volume
  labels:
    type: local
spec:
  storageClassName: student-pv
```

```
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 1Gi
  hostPath:
    path: /data/mongo2
    type: DirectoryOrCreate
```

Listing 12 PersistentVolume für MongoDB (StudentDB)

B.13. Kubernetes PersistentVolumeClaim.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: student-pv-claim
  labels:
    app: mongopv
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: student-pv
  resources:
    requests:
      storage: 1Gi
```

Listing 13 PersistentVolumeClaim bindet PersistentVolume (StudentDB)

B.14. Kubernetes ReplicationController.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: studentmongodb-controller
  labels:
    name: student
spec:
  replicas: 1
  template:
    metadata:
      labels:
        name: student
    spec:
      containers:
        - name: studentmongodb
          image: mongo:latest
          ports:
            - containerPort: 27017
              hostPort: 47863
          volumeMounts:
            - name: mongo-persistent-storage
              mountPath: /data/db
      volumes:
        - name: mongo-persistent-storage
          persistentVolumeClaim:
            claimName: student-pv-claim
```

Listing 14 Replication Controller mountet PersistentVolumeClaim als Volume

B.15. Kubernetes Service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: studentmongoservice
  labels:
    app: studentmongodb
spec:
  ports:
    - port: 47863
      protocol: TCP
      targetPort: 47863
  selector:
    appdb: student
```

Listing 15 Discovery-Service für MongoDB

B.16. Docker-Compose.yaml

```yaml
---
services:
  gitlab:
    container_name: GitLabBA-HSD
    environment:
      GITLAB_OMNIBUS_CONFIG: |
        postgresql['enable'] = false
        gitlab_rails['db_username'] = "gitlab"
        gitlab_rails['db_password'] = "gitlab"
        gitlab_rails['db_host'] = "postgresql"
        gitlab_rails['db_port'] = "5432"
        gitlab_rails['db_database'] = "gitlabhq_production"
        gitlab_rails['db_adapter'] = 'postgresql'
        gitlab_rails['db_encoding'] = 'utf8'
        redis['enable'] = false
        gitlab_rails['redis_host'] = 'redis'
        gitlab_rails['redis_port'] = '6379'
        gitlab_rails['gitlab_shell_ssh_port'] = 30022
    hostname: gitlab.BA-HSD.de
    image: "gitlab/gitlab-ce:latest"
    ports:
      - "30080:30080"
      - "30022:22"
    volumes:
      - "./data_gitlab_config:/etc/gitlab"
      - "./data_gitlab_logs:/var/log/gitlab"
      - "./data_gitlab_data:/var/opt/gitlab"
  postgresql:
    container_name: PostgresBA-HSD
    environment:
      - POSTGRES_USER=gitlab
      - POSTGRES_PASSWORD=gitlab
      - POSTGRES_DB=gitlabhq_production
    image: "postgres:latest"
    restart: always
    volumes:
      - "./data_postgresql:/var/lib/postgresql:rw"
  redis:
    container_name: RedisBA-HSD
```

```
    image: "redis:3.0.7-alpine"
    restart: always
version: "3"
```

Listing 16 Docker-Compose.yaml für die GitLab-Infrastruktur

B.17. Gitlab-Service-Account.yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: gitlab-admin
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: gitlab-admin
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
  - kind: ServiceAccount
    name: gitlab-admin
    namespace: kube-system
```

Listing 17 GitLab Service Account

B.18. Kubernetes Ingress.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: gateway-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  backend:
    serviceName: ingress-nginx-ingress-default-backend
    servicePort: 80
  rules:
  - host: microservices.hsdcard
    http:
      paths:
      - path: /student
        backend:
          serviceName: studierender-microservice-2
          servicePort: 8080
```

Listing 18 Beispielhafte Umsetzung eines Kubernetes Ingress Manifests (Studierender-Microservice)