Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Ein Buildsystem für Shopware-basierte Shops

Björn Matthäs
Matrikelnummer: 3985060
matthaes@zedat.fu-berlin.de

Betreuer: Prof. Dr. Lutz Prechelt
Eingereicht bei: Prof. Dr. Lutz Prechelt

Berlin, 30. Oktober 2019

Zusammenfassung

Diese Arbeit befasst sich mit der Erstellung einer Automatisierung des Buildprozesses für Shopware-basierte Shops auf Basis einer kontinuierlichen Integration und Deployment, um das Zusammenspiel von Shopware-Features zu testen und die Qualität der Programmierung zu erhöhen, als auch halbautomatisiert zu verteilen. Des Weiteren wird eine eigene Notation und Schnittstelle eingeführt um die Unabhängigkeit vom Continuous Integration Server zu gewährleisten.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

30. Oktober 2019

Björn Matthäs

Inhaltsverzeichnis

1 Einführung

Die Bachelorarbeit beschäftigt sich mit der Frage, wie ein Buildprozess eines Shopware-basierten Shop aussehen kann. Bereits bestehende Entwicklungsprozesse sollen hinsichtlich der Erhöhung von Codequalität und der zeitlicher Fertigstellung verbessert werden. In diesem Kapitel wird auf die Problemstellung eingegangen, welche Motivation dahinter steht und welches Ziel erreicht werden soll. Im zweiten Kapitel werden notwendige Begriffe und Definitionen eingeführt, welche für das Verständnis relevant sind. Das anschließende Kapitel beschäftigt sich mit der Erhebung von Anforderungen und Infrastrukturen, während Kapitel 4 die Vorgehensweise zur Erstellung eines Buildprozess für Shopware-basierte Shops beschreibt.

1.1 Problemstellung

Das Unternehmen „Portaltech Reply“ hat sich auf den Bereich des E-Commerce spezialisiert und ist ein mittelständisches IT-Beratungsunternehmen. Hier werden für Kunden unter anderem Onlineshops erstellt oder Erweiterungen für bestehende Onlineshops umgesetzt. Es werden gleichzeitig viele unterschiedliche Shops auf Basis von Shopware in vielen kleinen Teams realisiert. Über das Kundenfeedback unterschiedlichster Projekte, sowohl bei Neuerstellungen von Onlineshops als auch bei System- und Featureupdates, kristallisierte sich heraus, dass Releases sich oft zeitlich verzögern. Häufig genannte Gründe sind die späte Integration aller Implementierungen oder auch notwendige Rollbacks der Releases nach Livestellung, da nicht alle möglichen Fehlerquellen während der Entwicklung erfasst und getestet wurden. Während der Entwicklung werden die Tests manuell ausgeführt, sowohl von Entwicklern, Projektmanager, als auch von Kunden. Meist werden diese gegen Ende der Projektphase ausgeführt. Dies bringt die Problematik mit sich, dass Fehler sehr spät oder gar nicht entdeckt werden und sich dadurch die Dauer des Projektes verlängert, was häufig in Nacht- und Wochenendeinsätzen endet. Darüber hinaus werden die Tests auf unterschiedlich konfigurierten System (lokal, interne Staging- / Testsysteme und für Kunden bereitgestellte Staging- / Testsysteme) ausgeführt. Ein daraus entstehende Schwierigkeit ist, dass Entwicklungen auf einem System funktionieren, jedoch nicht auf einem anderen, was die Fehlersuche schwierig und langwierig macht. Aufgrund zu erwartender Problemen nach Liveschaltung von Onlineshops und sich damit einhergehenden hohen Entwicklungskosten werden zudem selten Shopsystem- und Feature-Updates durchgeführt.

Somit stellen sich durch das Kundenfeedback und dem Betrachten der Vorgehensweise der Erstellung / Updates von Entwicklungen nun zwei hauptsächliche Problemstellungen heraus: Die Integration und Qualität der einzelnen Entwicklungen sowie die Dauer bis zur Fertigstellung eines Releases.

1.2 Motivation

Das Bedürfnis nach einer höheren Qualität der Codebasis und einer schnelleren Auslieferung von Onlineshops soll als Motivation für einen automatisierte Abbildung des Entwicklungs-, Test- und Auslieferungsprozess dienen.

1.3 Ziel der Arbeit

Ziel soll es sein, eine Automatisierung der Prozesse für das kontinuierliche Integrieren und Ausliefern (Deployment) von Onlineshopsystemen zu entwickeln, um das Zusammenspiel der einzelnen Features zu testen und die Qualität der Programmierung zu erhöhen. Darüber hinaus soll ein schnelles Deployment der Shopprojekte mit deren zum Teil unterschiedlichen Anforderungen erzielt werden. Es soll eine Möglichkeit gefunden werden, die eine Art Schablonensystem implementiert und die Projekte automatisiert auf Continuous Integration-Server (CI-Server) erstellt, baut und testet sowie anschließend auf einem Staging- / Testsystem ausliefert. Dies soll exemplarisch für einen Shopware-basierten Shop in die Continuous Integration (CI) - / Continuous Delivery (CD) - Pipeline mittels Containerisierung durchgeführt und getestet werden.

2 Grundlagen

2.1 Definitionen

2.1.1 Shopware

Shopware ist eine Onlineshoplösung auf Basis der Programmiersprache PHP und wurde von dem gleichnamigen deutschen Unternehmen im Jahre 2004 entwickelt. Das Onlineshopsystem ist modular aufgebaut und bietet die Möglichkeit aus einer Vielzahl von Erweiterungen das Bedürfnis der Nutzer individuell anzupassen. Darüber hinaus verfügt Shopware (SW) über viele Content-Management-Funktionalitäten. Es werden sowohl eine Open-Source-Version, als auch eine kommerzielle Version angeboten. Zum Ende des Jahres 2019 wird eine neue Majorversion (Shopware 6) erscheinen. Diese wird auf einer vollständig neuen technologischen Basis entwickelt werden. [3]

2.1.2 Continuous Integration

Continuous Integration (CI) beschreibt einen Prozess, in dem in regelmäßigen Abständen die von Mitgliedern eines Entwicklerteams Arbeitsstände in eine Gesamtcodebasis zusammengeführt, getestet und zu einer funktionierenden Applikation gebaut wird. Dieses Konzept hat seinen Ursprung im „Extreme Programming“ [5]. Es beschreibt einen Prozess regelmäßiger Zusammenführung einzelner Komponenten in eine Anwendung. Dieses Konzept wurde unter anderem von M. Fowler um den Gedanken des automatisierten Build erweitert. „ Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible“ [9], was auch von Humble und Farley [12] unterstrichen wird „Continuous Integration that every time sombody commits any change, the entire application is built and a comprehensive set of automated tests is run against it“ .

Die folgende Grafik verdeutlich den Zyklus, welcher durchlaufen wird bei einem Continuous Integration-Prozess:

Um CI erfolgreich umsetzen zu können, werden einige Praktiken als essentiell betrachtet [12]. Sie stellen sicher, dass eine schnelle Fehlerbehebung und Zeitersparnis durch die Automatisierung realisiert werden kann. Diese Praktiken sind:

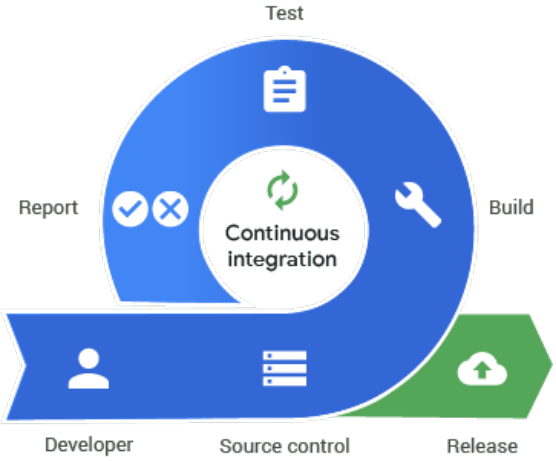

Abbildung 1: „https://codefluegel.com/de/continuous-integration/“

- Es soll kein Quellcode auf fehlgeschlagene Builds eingecheckt werden.
- Führe alle Tests im Vorfeld lokal aus, bevor eingecheckt wird.
- Warte darauf, dass alle Tests erfolgreich sind, bevor die Weiterentwicklung fortgeführt wird.
- Gehe niemals bei einem fehlgeschlagenen Build nach Hause.
- Kommentiere niemals fehlgeschlagene Tests aus.
- Übernehme die Verantwortung für alle Brüche im Quellcode, welche durch die eigenen Änderungen resultieren.

2.1.3 Deployment

Deployment bezeichnet den Prozess in der eine fertig bestehende Software in einer geeigneten Form verteilt und installiert wird. So kann dies zum Beispiel eine Auslieferung der Software auf einem Datenträger sein oder wie bei Webanwendungen üblich auf ein Server installiert und online gestellt wird.

2.1.4 Continuous Delivery

Continuous Delivery (CD) beschreibt eine Methode aus der Softwareentwicklung. Eine Software soll in der Form gebaut werden, so dass diese zu jedem Zeitpunkt in Produktion gehen kann. [8] Hierbei wird die im Buildprozess gebaute Software nur halbautomatisiert erstellt. Dies bedeutet, dass eine Software nur in den Produktionsmodus übergeht, wenn man explizit den Build auf ein System ausrollt. Die vollautomatisierte Form des Prozesses nennt man Continuous Deployment. Hierbei wird ohne Eingriff durch eine berechtigte externe Person die Software in den Produktionsmodus ausgerollt.

Für CD ist es unabdingbar, dass der Prozess Continuous Integration schon implementiert und ausgeführt wird. Folgende Abbildung verdeutlicht den Unterschied zwischen Continuous Delivery und Continuous Deployment:

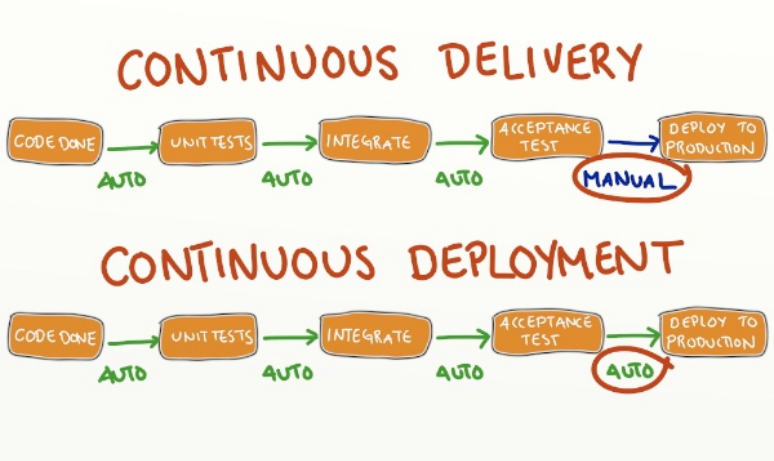

Abbildung 2: „https://sdtimes.com/automation/guest-view-continuous-delivery-vs-continuous-deployment-whats-difference/“

2.1.5 Virtualisierung / Containerisierung

Bei einer Virtualisierung ist es möglich Hardware gegenüber einem Nutzer als mehrere Maschinen darzustellen. Dabei wird die physisch vorhandene Hardware auf die unterschiedlichen Gastsysteme bzw. Maschinen verteilt. In diesem Gastsystem ist ein komplettes Betriebssystem installiert. Somit ist es möglich, dass ein Nutzer seine Anwendung in diesem Gastsystem laufen lassen kann. Nachtteil der Virtualisierung ist, dass diese Lösung von der Hardware unterstützt werden muss und sie sehr schwergewichtig ist. Diesen Nachteil versucht die Containerisierung durch den Ansatz entgegen zu wirken, dass es nicht mehr notwendig wird ein vollständiges Betriebssystem in jedem Kontext bereit zu stellen, sondern nur nur als Basis und weiter durch unterschiedliche Schichten konfiguriert. Auch in diesem Ansatz kann die ein Anwender seine Applikation laufen lassen. Ein sehr bekannter Vertreter der Containerlösung ist die Software Docker. Mit dieser Software ist es möglich Container zu erstellen und zu starten. Docker [14]stellt ein Tool bereit, welches docker-compose heisst. [13] Mit diesem Tool ist es möglich in einer einzelnen Datei die unterschiedliche Services zu definieren und zu konfigurieren. Hierbei stellt ein Service ein Container dar. So ist es zum Beispiel möglich alle notwendigen Komponenten einer Webapplikation einfach zu orchestrieren und erstellen.

3 Anforderungsanalyse

3.1 Interviews

Um ein Buildsystem für einen Shopware-basierten Shop aufzubauen, bedarf es einer genauen Ermittlung der Anforderungen. Hierbei soll der aktuelle Prozess beim Entwickeln (in einem Team) eines neuen Onlineshops oder eines Features erfasst werden, um einen IST-Zustand festzusetzen. Darüber hinaus wird gleichzeitig die bestehende Infrastruktur aufgenommen. Das daraus entstehende Ergebnis soll als Grundlage dienen, um ein Buildsystem, die konkreten Einzelschritte, notwendige Tools und aus-

zuführende Befehle abbilden zu können. Des Weiteren sollen auch die Wünsche der Entwickler und aktuelle Probleme aufgenommen werden. Als Vorbereitung auf die Interviews wurde ein kurzer Fragebogen entwickelt. Dieser beinhaltet die im Folgenden aufgelisteten Fragen:

- Welche Kundenprojekte hat der Entwickler oder Projektmanager?
- Welche Technologien werden benötigt und / oder genutzt (Datenbanksystem, Mailsystem, verwendetes Suchsystem...)?
- Welche PHP-Version wird genutzt?
- Welche Version des Shopwaresystems wird verwendet?
- Welche Umgebungsvariablen werden für die Lauffähigkeit des Shopsystems benötigt?
- Was sind notwendige Kommandozeilenbefehle beziehungsweise Shopwarekommandobefehle?
- Welche Vorgehensweisen und Strategien werden bei den Projekten bezüglich des Versionskontrollsystems (VCS) verwendet?
- Welche Webanwendung wird für das Versionskontrollsystem genutzt?
- Wie und wohin wird der Onlineshop zum Testen für den Kunden ausgerollt?
- Wird eigenes Hosting oder Kundenhosting genutzt?
- Wird Containerisierung genutzt (Docker, Vagrant oder ähnliches)?

3.2 Auswertung

Im Anschluss an die Interviews werden die Antworten verglichen und kategorisiert nach Wichtigkeit und nach Verteilung der Häufigkeit zum Beispiel bei der genutzten Version von Shopware und PHP. Diese Ergebnisse der Interviews geben einen Einblick in die Anforderungen. Für die Beantwortung der Fragen bezüglich der Infrastruktur wurde zusätzlich die verantwortliche Abteilung für Infrastruktur und Hosting zu Rate gezogen, welche Anwendungen im Einsatz sind und zur Verfügung stehen. Geprüft wurde, ob Anwendungen zur statischen Codeanalyse oder bestehende Continuos Integration-Server existieren und wie das Hosting aufgestellt (eigene Server, Cloud) ist.

3.3 IST-Zustand

Dieser Abschnitt befasst sich mit dem aktuellen Zustand des Entwicklungsprozesses basierend auf den Interviews und etwaigen vorhandenen Buildprozessen in der Firma Portaltech Reply.

3.3.1 Architektur eines Shopware-basierten Shops

Es hat sich durch die Interviews herauskristallisiert, dass es zum Teil unterschiedliche Basen bei den Shopwaresystemen und auch dem verwendeten System für Erweiterungen (im Folgenden Pluginsystem) gibt. Zu unterscheiden ist, welche Shopwareversion genutzt wird und welches daraus resultierende Pluginsystem verwendet wird.. Shopwaresysteme kleiner als Version 5.2 nutzen das sogenannte „alte“ Pluginsystem von Shopware [1]. Shopwareversionen größer 5.2 nutzen das neue Pluginsystem [2]. Relevant ist dabei, dass beide Pluginsysteme unterschiedliche Pfade zu den Erweiterungen nutzen, welche gegebenenfalls abgebildet werden müssen. Des Weiteren besteht ein Unterschied in der gelegentlichen Basis des Aufbaus eines Shopwarebasierten Shops. Es wird unterschieden zwischen einem sogenannten Composer-Setup und einem Legacy-Setup. Während beim Legacy-Setup alle Shopwarekomponenten im Basisverzeichnis platziert sind, wird beim Composer-Setup Shopware selbst als Bibliothek eingebunden [7].

Da die meisten Projekte bei Portaltech Reply mit dem Composer-Setup arbeiten oder zukünftig arbeiten werden, wird in dieser Arbeit das Legacy-Setup betrachtet.

Ein typischer Aufbau eines Shopware-basierte Shop besteht aus den Komponenten PHP-Interpreter [17], eine Datenbank wie Mysql [**?**] für die Speicherung der Daten und einen Webserver, wie NGINX [16] welcher ein Frontend über das Internet bereitstellen kann.

3.3.2 Infrastruktur

Die vorhandene Infrastruktur der Firma Portaltech Reply, sieht folgendermaßen aus: Es existieren unterschiedliche Serversysteme für das interne Testing und für die Entwickler sowie Systeme für Kunden zum Testen und Systeme für den produktiven Einsatz. Der Einfachheit halber werden diese Umgebungen DEV, INTEGRATION, und PRODUKTION genannt. Das Produktivsystem wird im Folgenden außer Acht gelassen, da die meisten bestehenden Kundenprojekte in Zukunft erst in die firmeneigene Cloud umgezogen werden. Der Buildprozess wird somit nur halbautomatisiert (Continuous Delivery) abgebildet. Weiterhin wird in fast allen Kundenprojekten der Firma Portaltech Reply das Versionskontrollsystem Git verwendet und die Repositorys mittels GitLab (1) als Weboberfläche verwaltet. Containerisierung der Shopware-basierten Shops wird größtenteils zur lokalen Entwicklung benutzt und basiert auf der Softwarelösung Docker und dem dazugehörigen Tool „docker-compose“. Portaltech Reply verfügt bereits über einen existierenden Jenkins (2) CI-Server, allerdings bietet auch Gitlab die Möglichkeit, als CI-Server zu fungieren.

3.3.3 Buildprozess und Deployment

Der aktuelle Buildprozess sieht wie folgt aus: Es wird bei der Entwicklung ein Gitflow-basiertes Modell genutzt [4]. Sobald neu umgesetzte Entwicklungen in die Kundentestumgebung integriert werden sollen, müssen die Änderungen in einem durch das jeweilige Team bestimmten Branch zusammengeführt werden und manuell auf den internen Testserver heruntergeladen und ausgecheckt werden (git checkout), mittels

Composer werden zudem die Bibliotheken installiert und notwendige Konfigurationen im Shop für die neuen Entwicklungen getätigt. Da diese auf Plugins basieren, werden diese manuell installiert, aktiviert und konfiguriert. Nun wird manuell getestet und anschließend bei „Erfolg“ mit der gleichen Prozedur auf dem Kundentestsystem installiert und freigegeben. Zur Hauptkommunikation setzt die Firma auf das Chatsystem „Slack“, zusätzlich wird mittels E-Mail kommuniziert.

3.4 SOLL-Zustand

Der Buildprozess soll (halb-) automatisiert ablaufen. Nach dem „Einchecken“ der neuesten Änderungen muss der Onlineshop sowohl auf einen Test- als auch Kundentestsystem heruntergeladen, Bibliotheken installiert und, getestet werden sowie ausgerollt werden. Funktionale Anforderungen sind:

- Automatisierung der Installationsschritte
- Ausführung von automatisierten Unittests
- Prüfen auf Kompatibilität der genutzten PHP-Versionen
- Statische Codeanalyse der Entwicklung
- Benachrichtigung per E-Mail und Slack über den Status der Builds (Erfolg / Misserfolg)
- Ausführung der allgemeinen Shopwaretests
- Setzen von Umgebungsvariablen

Nichtfunktionale Anforderungen sind:

- Zukunftssicherheit des Buildprozesses und Unabhängigkeit zum CI-Server
- Einfache Erstellung eines standardisierten Buildprozesses
- Gleichartige Testumgebung

3.5 Probleme

Beim Erheben der Anforderungen und beim Sichten des Quellcodes der unterschiedlichsten Kundenprojekte (und durch die Interviews) wurde deutlich, dass in den Projekten und Teams sehr wenige Tests geschrieben werden. Fast alles wird manuell nach Checklisten getestet. Problematisch hierbei ist, dass der automatisierte Testprozess im Continuous Integration und Continuous Delivery Prozess wichtig ist. Das manuelle Testen ist somit ein zeitfressender Faktor. Des Weiteren kann ohne automatisiertes Testing keine verlässliche Auskunft über den Zustand der Anwendungen gegeben werden.

4 Vorgehensweise

4.1 Vorüberlegungen

Um den Buildprozess wie in den funktionalen Anforderungen automatisiert ablaufen zu lassen, ist es notwendig, einen sogenannten CI-Server zu nutzen. Dieser soll die erforderlichen Schritte des Bauens und dem Ausführen der Tests übernehmen sowie das Ausrollen auf den verschiedenen Umgebungen automatisieren. Wie die Interviews gezeigt haben, besitzt Portaltech Reply bereits zwei Arten von CI-Servern, welche bisher jedoch nicht genutzt werden. Nach Recherche zur Handhabung und Nutzung der beiden Systeme fiel die erste Wahl auf Jenkins, da dieser CI-Server über eine Fülle von Dokumentation und Anleitungen verfügt. Um gleichartige Testumgebungen abbilden zu können, wird auf die Nutzung der Containerisierungslösung Docker gesetzt. Ein weiterer Punkt für eine Containerbasierte Umgebung ist, dass zum Ende des Jahres eine neue Majorversion von Shopware (SW6) erscheint, welche einen API-First-Ansatz und Microservice-orientiertes Design nutzen wird. Weitere Überlegungen bezüglich der schlecht aufgestellten Testbasis bei Kundenprojekten ist es, die von Shopware eigens ausgelieferten Tests ausführen zu lassen, um zumindest die Kernfunktionalität mit veränderter Logik durch Plugins sicherstellen zu können. Hierbei bezieht sich die Kernfunktionalität auf die Nutzung des digitalen Einkaufskorbs bis hin zum Bestellabschluss und die vorhandenen grundlegenden Artikeldaten. Sollten hier durch selbsterstellte Plugins die Bereiche des Bestellprozessen beeinträchtigt und notwendigen Artikeldaten beschädigt sein, so wäre dies bereits durch Shopware-eigene Tests abgefangen.

Zukunftssicherheit des Buildprozesses und die Unabhängigkeit des CI-Servers kann durch die Einführung einer Schnittstelle und eigener Notation für die Konfiguration erreicht werden. Somit ist es notwendig im Vorfeld eine eigene Notation zu definieren, wobei diese durch die Schnittstelle in geeigneter Weise interpretiert und umgesetzt werden muss, um den dahinter stehenden CI-Server ansprechen zu können.

4.2 Eigene Notation

Bei der Definition der Notation für die Schnittstelle wurde im ersten Schritt geprüft, welche Informationen notwendig sind, um einen Onlineshop zu erstellen und lauffähig zu machen. Darunter fallen zum Beispiel die Zugangsdaten für Datenbanken, welcher Branch ausgecheckt werden soll, welche Umgebungsvariablen genutzt werden sollen und über welche Kommunikationskanäle ein Feedback des Buildprozesses versendet werden soll. Folgende Notation ist im ersten Schritt entstanden:

```
notification:
  slack:
    channel: '#feedbackchannel'
    private: 'max.mustermann'
  mail:
    to:
    - admin@mail.com
    - dev@mail.com
```

```
deploy:
  integration:
    branch: integration
    url: 'integration.example.com'
    variables:
    credentials:
    db:
    - DB_HOST=integration
    - DB_DATABASE=integration
    - DB_USERNAME=root
    - DB_PASSWORD=root
    - DB_PORT=3306
    env: integration
  staging:
    branch: dev
    url: 'staging.example.com'
    variables:
    credentials:
    db:
    - DB_HOST=staging
    - DB_DATABASE=staging
    - DB_USERNAME=root
    - DB_PASSWORD=root
    - DB_PORT=3306
    env: testing
   ...
```

Diese Notation soll der Einfachheit halber unter dem Dateinamen „ci.yml“ im Basisverzeichnis des Onlineshops existieren. Hier wird auf das Format Yaml gesetzt, da dies für Entwickler schnell zu erstellen, zu verstehen und lesbar ist.

4.3 Interface / Schnittstelle

Wie bereits in Abschnitt 4.1 erwähnt, ist es sinnvoll eine Schnittstelle einzuführen. Diese soll den CI-Server vom eigentlichen Buildprozess abkoppeln. Der Vorteil dieser Lösung ist, dass nun der CI-Server austauschbar ist. Sollte sich die Firma Portaltech Reply entschließen, aus lizenzrechtlichen, finanziellen oder anderen Gründen eben diesen zu tauschen, muss nur noch die Schnittstelle angepasst werden, um ein anderes System zu unterstützen. Dies erhöht die Wartbarkeit und die Zukunftssicherheit. Damit ist es möglich, dass ein Entwickler(-team) die Schnittstelle anpassen kann und die DevOps-Abteilung für Server und Konfiguration verantwortlich sein kann. Wie in Abschnitt 4.2 zur Notation erwähnt, muss die Schnittstelle, die definierte Notation als Eingabe verarbeiten und in geeigneter Weise den CI-Server ansprechen können. Nach Recherche der Dokumentation zu Jenkins und Gitlab steht fest, dass beide Systeme mithilfe einer Konfigurationsdatei mit Anweisungen zum Buildprozess gesteuert werden können. Bei Jenkins lautet diese Datei „Jenkinsfile“, bei Gitlab „.gitlab-ci.yml“. Somit steht fest, dass die Schnittstelle als Ausgabe eine solche Datei erzeugen muss. Beide Konfigurationen liegen üblicherweise im Basisverzeichnis der zu bauenden Anwendung. In vorliegendem Fall ist dies das Basisverzeichnis des Onlineshops. Es liegt

nahe, die Schnittstelle in der Programmiersprache zu entwickeln, welche bei Portaltech Reply standardmäßig eingesetzt wird, dies ist PHP sowie das darauf basierende Frameworks Symfony. Dies ist ein sehr verbreitetes Framework und bietet aufgrund der Modularität alle Möglichkeiten, die Schnittstelle umzusetzen. Da Symfony eine Konsolenkomponente bereitstellt, kann die Anwendung einfach auf der Konsole gestartet werden. Im ersten Schritt wird der Prozess für die CI-Server-Datei einfach gehalten. Somit sieht der angedachte Prozess folgendermaßen aus:

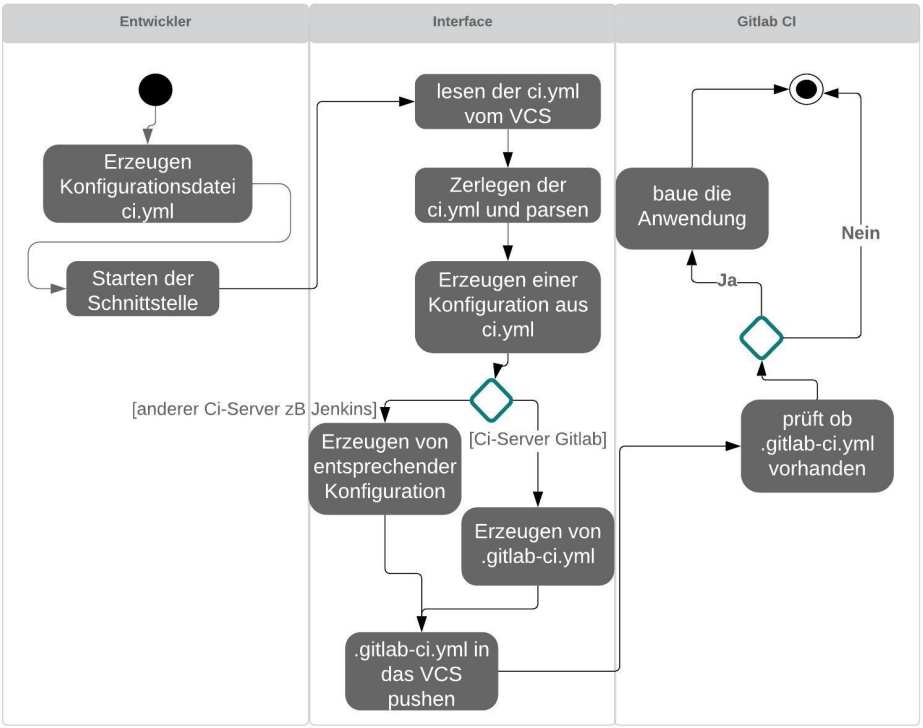

Abbildung 3: „Aktivitätsdiagramm“

Folgender Ablauf:

1. Schnittstelle liest die Datei ci.yml vom GitLab-System ein
2. Schnittstelle zerlegt die Konfiguration (parsen)
3. Schnittstelle entscheidet anhand eines Parameters, welche CI-Konfiguration erzeugt werden soll
4. Schnittstelle schreibt die CI-Konfiguration als Datei zurück in das GitLab-System
5. Gitlab startet bei Vorhandensein einer „.gitlab-ci.yml“ den Buildprozess

Anhand des oben genannten Ablaufes ist es möglich, die Schnittstelle in die einzelnen Komponenten aufzuteilen. Die Schnittstelle benötigt eine Komponente, die die zuvor definierte Notation zerlegen kann, eine Komponente, die daraus eine Konfiguration bauen kann und eine, die sich mit einem Versionskontrollsystem verbinden, lesen und schreiben kann. Somit ergeben sich folgende Bereiche:

- Connector (BoundaryObject, ControlObject)
- Parser (ControlObject)
- Reader, Writer (ControlObject)
- Schnittstelle schreibt die CI-Konfiguration als Datei zurück in das GitLab-System
- Configuration (EntityObject)
- CLI-Komponente (BoundaryObject)

Die erzeugte CI-Konfiguration muss am Ende auch für den richtigen CI-Server erstellt werden. Dazu wird beim Start der Schnittstelle um einen weiteren Parameter zum Beispiel „jenkins“ oder „gitlab“ ergänzt. In der Schnittstelle ist für das Erzeugen der Ausgabedatei das Strategie-Muster [10] genutzt worden, welche mithilfe des übergebenden Parameters die konkrete Strategie (Algorithmus) auswählt. Damit ist es möglich, zusätzliche „Writer“ im Nachhinein zu entwickeln, die einen anderen CI-Server unterstützen. Ähnlich wurde die Komponente „Connector“, welche die Verbindung mit dem Versionskontrollsystem im konkreten Fall das firmeneigene GitLab zuständig ist, gebaut. Um beide Seiten der Verbindung zu realisieren und auch um eine einfache CI-Konfiguration zu testen, sollte nun der gewählte Jenkins zum Einsatz kommen. Dabei stellte sich heraus, dass die verantwortliche Abteilung für Infrastruktur den vorhandenen Jenkins nicht mehr warten wird. Die weitere Nutzung des Jenkins würde hier somit gegen die nicht funktionale Anforderung verstoßen. Es macht deshalb wenig Sinn weiterhin auf den Jenkins in der Entwicklung zu setzen. Aus diesem Grund wird die in der Vorüberlegung getroffene Wahl revidiert und stattdessen GitLab als CI-Server genommen. Für einen ersten Prototyp kann die Architektur folgendermaßen skizziert werden:

Die Schnittstelle übernimmt die Aufgaben, eine definierte Konfigurationsdatei aus dem GitLab zu holen, zu zerlegen, die entsprechende Ausgabedatei zu erzeugen und als eigenständigen „Commit“ zurück zu schreiben beziehungsweise zu pushen. Zu diesem Zeitpunkt ist es noch nicht möglich, sich mit GitLab zu verbinden, zu lesen oder zu schreiben. Dies liegt an der fehlenden Authentifizierung. GitLab selbst bietet ein Interface, im folgenden GitLab-API genannt. Dies ermöglicht, sich über diese API zu authentifizieren, mittels eines sogenannten „AccessToken“. Dieser kann für einen beliebigen User erzeugt werden. Des Weiteren muss dem User GitLab-API Rechte gewährt werden, um in einem Repository lesen und schreiben zu können. Der Access-Token wird beim Start in der Kommandozeile als Parameter übergeben. Zusätzlich muss der Schnittstelle mitgeteilt werden, in welches Repository sie schreiben soll (das eigentliche Kundenshop-Repository). Dazu wird auch beim Start die URL des Repositories zusätzlich übergeben. Die URL wird in dem Connector zulegt und später in die GitLab-API-URL transformiert. Eine bereits existierende CI-Server-Konfiguration wird gegebenenfalls geupdated. Mit der Schnittstelle wurde das VCS auf der einen Seite und dem CI-Server auf der anderen Seite verbunden. Darüber hinaus wurde ein einfacher Controller in die Schnittstelle eingebaut, um den Status der Applikation wiederzugeben. Dieser Status zeigt an, ob die Anwendung läuft oder nicht. Hierbei wird ein einfacher JSON-String zurückgegeben. Während der Entwicklungszeit wurde darauf geachtet, diese mit Unittests zu versehen. Im PHP-Umfeld hat sich das

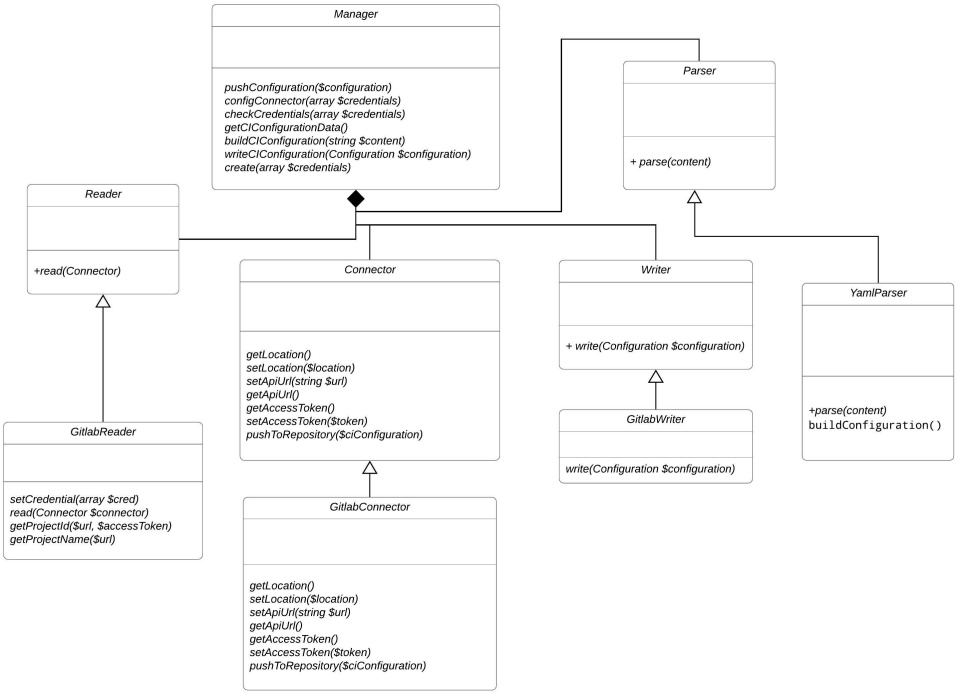

Abbildung 4: „Skizzierung des Prototyps“

Testframework PHPUnit [6] als de facto Standard durchgesetzt. Das zugrunde liegende Framework Symfony bietet die Integration der Testumgebung auf einfache Weise an. Die Schnittstelle kann mit dem Konsolenbefehl `"./bin/phpunit"` aufgerufen werden. Das Testframework wird auch bei den Shopware-basierten Shops auf ähnliche Weise aufgerufen.

4.4 Containerisierung

Wie in Kapitel 2 angeführt, kann mit Hilfe einer Containerisierungstechnologie ein Container erstellt und in Betrieb genommen werden. Damit ist es möglich, Anwendungen in diesen Containern zu starten und auszuführen. Docker ist eine Software, welche es ermöglicht, diese Technologie einzusetzen. Der Vorteil bei dieser Technologie ist, dass auf einem Server beliebig viele Container unabhängig laufen können. Somit kann die Anwendung in Containern nach den Bedürfnissen gebaut werden, ohne das darunterliegende Linux anpassen zu müssen. Oft ist dies auch aus Sicherheitsgründen nicht erlaubt. Um eine Anwendung in einem Container lauffähig zu machen, müssen die benötigten Softwarekomponenten und Tools nur in den Containern installiert werden. Wenn mehrere Container, welche für den Betrieb notwendig sind, (Datenbank, Webserver etc.) benötigt werden, kann das Tool docker-compose genutzt werden. Dieses Tool vereinfacht das Administrieren vieler Container. Solch eine Umgebung ist über die sogenannte `"docker-compose.yml"` definierbar. Mit dieser Lösung kann nun die Schnittstelle containerisiert werden, aber auch die Kundenonlineshops. In der Firma setzen einige Entwickler auf Docker und docker-compose für die lokale Entwicklung. Aus diesem Grund existiert bereits für die Shopware-

basierten Shops eine Containerisierung, welche die notwendigen Komponenten wie Datenbank, Webserver und PHP-Interpreter bereitstellt. Da diese Containerisierung bereits erprobt ist, kann sie für den Buildprozess genutzt werden.

Nach demselben Schema wurde eine docker-compose-Datei für die Schnittstelle erstellt. Sie besteht im Wesentlichen aus einem Webserver und einem PHP-Interpreter.

```
version: '3.7'
services:
app:
  build:
    context: .
      dockerfile: ./docker/php-fpm/Dockerfile.staging
    args:
      uid: 1000
      gid: 1000
  expose:
    - "9000"
  volumes:
    - ${APP_PATH}:/app
web:
  image: nginx:alpine
  volumes:
    - ${APP_PATH}:/app
    - ./docker/nginx/default.conf:/etc/nginx/conf.d/default.
       conf
  ports:
    - "8081:80"
  command: nginx -g "daemon off";
  depends_on:
    - app
```

In dem oben gezeigten Listing für die Containerisierung der Schnittstelle werden über das Keyword „service“ vier unterschiedliche Container definiert und konfiguriert. Hier werden ein Webserver auf Basis von NGINX und ein PHP-Container zum Ausführen der Anwendung erzeugt. In jedem dieser Services werden Ports freigegeben über „ports“ und „expose“, die für eine Erreichbarkeit notwendig sind.Über das „volumes“-Keyword wird der Quellcode der Applikation vom Hostsystem in die Container gemountet. Dies ist notwendig, sonst kann im Container keine Applikation laufen, da von seitens des Containers kein Zugriff nach aussen auf das Hostsystem erlaubt ist.

Eine docker-compose-Datei für einen Shopware-basiertes Shop sieht ähnlich aus:

```
services:
  web:
    build:
      context: .
      dockerfile: ./docker/nginx/Dockerfile
    ports:
      - 80:80
    volumes:
      - project:/app
    depends_on:
```

```
    - php
php:
  build:
    context: .
    dockerfile: ./docker/php/Dockerfile
    args:
      uid: 1000
      gid: 1000
  volumes:
    - project:/app
  depends_on:
    - db
db:
  build:
    context: .
    dockerfile: ./docker/mysql/Dockerfile
  environment:
    MYSQL_ROOT_PASSWORD: "'password"'
    MYSQL_USER:"'"user"'
    MYSQL_PASSWORD: "'password"'
    MYSQL_DATABASE: "'databasename"'
  ports:
    - 3306:3306
  volumes:
    - mysql-data:/var/lib/mysql
mailer:
  image: djfarrelly/maildev
  ports:
  - "1080:80"
```

Wie man gegenüber dem Schnittstellen docker-compose-File sehen kann, werden hier zusätzlich eine Datenbank bereitgestellt und ein Mailer. Bei dieser Version ist interessant zu sehen, wie man Environment-Variablen an den Container mit übergeben kann, siehe service „db“. Somit ist es möglich wichtige Zugangsdaten mit zu übergeben. Aufgrund der Sicherheit, ist es sinnvoll im GitLab-Repository des Kundenonlineshops die Zugangsdaten als geschützte Variablen zu hinterlegen. Der GitLab-CI-Server kann diesem dem Buildprozess übergeben und weiterreichen.

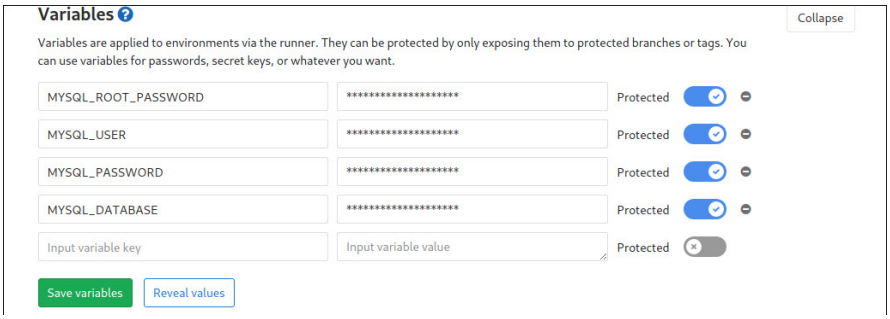

Abbildung 5: „GitLab -Settings - CICD - Variables “

4.5 Schritt Continuous Delivery

Jetzt kann man sich fragen, weshalb die ganze Mühe mit der Containerisierung und dem docker-compose. Mit `"docker-compose up -d --build"` kann die gesamte Schnittstelle lauffähig gemacht werden. Die benötigten Container werden gebaut und anschließend gestartet. Mit diesem Einzeiler kann der Buildprozess enorm vereinfacht werden. Selbiges gilt für Shopware-basierte Shops. Das heißt, man kann mit dem Befehl die gesamte Anwendung überall bauen und starten. Dies wäre in unserem Buildprozess der Schritt, um einen Shopwareshop auf einen internen Testserver und / oder auf einem Kundentestsystem zur Verfügung zu stellen.

4.6 Schritt Continuous Integration

In Abschnitt 2.1.2 wurde CI definiert. Um dies sowohl für die Schnittstelle als auch Shopware-basierte Shops umzusetzen, muss im Buildprozess die jeweilige Anwendung / Onlineshop getestet werden. Wie dies für die Schnittstelle aussieht, wurde bereits in der Umsetzung gezeigt. Ähnlich läuft dies für SW-Shops. Hier wird auch auf das Testingframework PHPUnit zurückgegriffen. Der ausführende Befehl lautet hier:

```
"./vendor/bin/phpunit"
```

PHPUnit muss im Vorfeld installiert werden, dies gilt auch für die Schnittstelle. In Kapitel 3 wurde bereits über Composer geschrieben, welches Bibliotheken für eine PHP-Anwendung installiert. Von diesem Tool machen wir Gebrauch und können mit dem Befehl „composer require –dev“ das Framework als Bibliothek installieren. Alternativ kann die Bibliothek auch über die sogenannte „composer.json“ hinzugefügt werden und mit dem Befehl „composer install“ installiert werden. Letzerer Befehl ist auch notwendig, um die Shopware-basierten Shops zu bauen und deren Abhängigkeiten zu installieren. Eingangs wurde erläutert, dass Shopware einerseits aus Plugins und andererseits aus dem Shopwarekern besteht. Um die Shopware-eigenen Tests auszuführen, reicht es im Basisverzeichnis ein einfaches „./vendor/bin/phpunit“ zu schreiben. Die Plugins müssen anders getestet werden. Dazu muss der gerade dokumentierte Befehl um ein „custom/project“ erweitert werden, da die Plugins nicht im Shopwarekern integriert sind, sondern über ein anderes Verzeichnis eingebunden sind. Die Befehle zum Testen und zum Containerisieren müssen in eine separate Datei zusammengefasst werden. Diese Datei schreibt den gesamten Buildprozess und heißt wie bereits erwähnt „.gitlab-ci.yml“. Diese Datei besteht aus mehreren Abschnitten.

Da Shopware-basierte Shops und die Schnittstelle mit Hilfe von Docker und docker-compose gebaut werden soll beziehungsweise auch auf Test- / Kundentestsystemen deployed benötigigen wir ein Container mit einem vorinstallierten Docker. In der Dokumentation empfiehlt GitLab diese Variante. [15]

```
image: docker:19:03
services:
  - docker:dind
```

Im ersten Teil der Datei wird im Abschnitt „stages“ definiert, welche Aufgaben ausgeführt werden sollen. Dazu werden in einer Liste frei wählbare Namen definiert. Auf diese werden in den einzelnen Aufgabenabschnitten referenziert. Des Weiteren

wird hier mit Hilfe des Abschnitts „cache“ ein globale flüchtiger Cache für den aktuellen Buildprozess definiert. Der „vendor“-Ordner ist bei dem Composer-Setup von Shopwareshops der Pfad zu den Bibliotheken. Dieser wird während des gesamtem Prozesses allen anderen Jobs zur Verfügung stehen.

```
stages:
  - prepare
  - test
  - deploy
  - notify
```

```
cache:
  untracked: true
  key: ${CI_PROJECT_NAME}-${CI_COMMIT_REF_SLUG}
  paths:
    - vendor/
  policy: pull
```

Wie im letzten Kapitel beschrieben, müssen alle Abhängigkeiten auch im Buildprozess installiert werden. Dies kann man über einen separaten „Job“ ausführen. Diese werden über den sogenannten „Cache“ in die anderen Jobs übertragen. Problematisch bei der Erstellung des Caches und GitLab ist in Kombination mit asynchron ausgeführten Aufgaben, dass dieser immer überschrieben wird und leer ist [11]. Für die Schnittstelle und den Shopwareshops würde dies bedeuten, in jedem Job die Abhängigkeiten erneut zu installieren. Dies kostet sehr viel Zeit und ist inperformant. Als Lösung wurde der gerade definierte Cache ersetzt durch Artefakte. Diese wurden mit einer Ablaufzeit von 10 Minuten definiert. Der Grund ist, dass Artefakte nicht flüchtig sind und manuell entfernt werden müssen. Somit können die Abhängigkeiten als Artefakte in jedem Job genutzt werden und ein erneutes Installieren ist nicht mehr notwendig. Dieses Problem tritt nur auf, wenn die sogenannten „gitlab-runner“ mit mehr als einem Thread und als Executor Docker ausgeführt werden. Jeder Job definiert, was in dem Buildprozess getan werden soll.

Im „script“-Abschnitt werden zusätzlich Bibliotheken installiert, welche nur im Buildprozess benötigt werden. Wichtig hierbei ist, dass wir diese Befehle im Kontext des aktuellen Users laufen lassen, da die Container standardmäßig unter dem User Root laufen. Somit hätten wir in den anschliessenden Jobs keine Berechtigung. Den aktuellen User und die Gruppe kann man mit `--user=$(id -u):$(id -g)` übergeben.

```
build_deps:
  stage: prepare
  cache:
    untracked: true
    key: ${CI_PROJECT_NAME}-${CI_COMMIT_REF_SLUG}
    paths:
    - vendor/
    policy: pull-push
  before_script:
    - mkdir -p $CI_PROJECT_DIR/.composer/cache
    - export COMPOSER_CACHE_DIR=$CI_PROJECT_DIR/.composer/cache
```

```
  script:
    - sudo docker run --user=$(id -u):$(id -g) --rm --volume
       $PWD:/app composer require  wapmorgan/php-code-fixer -q
    - sudo docker run --user=$(id -u):$(id -g) --rm --volume
       $PWD:/app composer require  squizlabs/php_codesniffer -q
    - sudo docker run --user=$(id -u):$(id -g) --rm --volume
       $PWD:/app composer require  sebastian/phpcpd -q
    - sudo docker run --user=$(id -u):$(id -g) --rm --volume
       $PWD:/app composer install
  artifacts:
    paths:
    - vendor/
    expire_in: 10 minutes
  tags:
  - build
```

Nachfolgend wurden mehrere Jobs definiert, die eine statische Codeanalyse durchführen. Unter dem Job mit dem Name „compatibility“ wird geprüft ob die Anwendung unter verschiedenen PHP-Versionen läuft.

```
compatibility:
  stage: test
  image: php:7.3-cli
  dependencies:
    - build_deps
  script:
    - ./vendor/bin/phpcf -t 7.0 ./src
    - ./vendor/bin/phpcf -t 7.1 ./src
    - ./vendor/bin/phpcf -t 7.2 ./src
    - ./vendor/bin/phpcf -t 7.3 ./src
  tags:
    - tests
```

Dieser Job testet auf dem von Portaltech Reply als Standard definierten Format beim Quellcode und ob prüft auf Duplikate mittels eines Copy & Paste-Detector.

```
codestyle:
  stage: test
  image: php:7.3-cli
  dependencies:
    - build_deps
  script:
    - ./vendor/bin/phpcpd ./src
    - ./vendor/bin/phpcs --colors --standard=PSR2 ./src
  tags:
    - tests
```

Um die Unittests der Anwendung auszuführen muss in diesem speziellen Fall im Container bei Ausführung das Tool Composer installiert. Notwendig ist, da Symfony das Testframework über einen individuellen Pfad aufruft und dieser die benötigten Abhängigkeiten in eben diesen Pfad installiert.

```
unittests:
```

```
  stage: test
  image: php:7.3-cli
  dependencies:
    - build_deps
  before_script:
    - apt-get -qq update
    - apt-get install -qq -y git
    - curl -sS https://getcomposer.org/installer | php
  script:
    - php bin/phpunit
  allow_failure: false
  tags:
    - tests
```

Dieser Job hat die Aufgabe den Quellcode zu mittels dem SonarQube-Scanner zu analyseren und an einen SonarQube zu übermitteln. Sonarqube ist eine Plattform für die statische Codeanalyse und zur Bewertung der technischen Qualität des Codes.

```
review:
  stage: test
  image: newtmitch/sonar-scanner
  script: >-
    /usr/local/bin/sonar-scanner \
    -Dsonar.projectKey=interfaceApp \
    -Dsonar.sources=src \
    -Dsonar.host.url=urls:9010 \
    -Dsonar.login=loginkey
  tags:
    - tests
```

In diesem Job wird das Tool docker-compose installiert. Anschliessend baut und startet docker-compose die Container wie im Kapitel 4.4 erläutert mittels der Konfiguration `"'docker-comose.yml"'`. Abschließend wir der Cache der Anwendung geleert.

```
staging:
  stage: deploy
  dependencies:
    - build_deps
  only:
    - staging
  before_script:
    - export COMPOSE_INTERACTIVE_NO_CLI=1
  script:
    - sudo apt-get install -y python-pip
    - sudo pip install docker-compose
    - sudo docker image prune -f
    - sudo docker-compose -f docker-compose.yml build
    - sudo docker-compose -f docker-cmpose.yml up -d
    - sudo docker-compose exec -T --user=$(id -u):$(id -g) app
       php bin/console cache:clear
  tags:
    - deploy
```

Jeder dieser Jobs ist mit einem Tag versehen. Diese können genutzt werden um unterschiedliche ausführende „gitlab-runner“ zu nutzen. Damit können Jobs unterschiedlich behandelt werden und in bestimmten Situationen nicht gestartet werden.

4.7 Feedback

Essentiell für die CI ist die sogenannte Feedbackschleife. Das bedeutet, wenn der oben genannte automatisierte Buildprozess in Gänze oder Teilen fehlschlägt, muss der Entwickler informiert werden. Dies ermöglicht eine schnellere Raktionszeit, wodurch Fehler schneller behoben werden können. Dieses Prinzip ist das oberste Gebot bei der Entwicklung und CI. Das Gebot besagt „Behebe erst die Fehler, bevor neue Entwicklungen getätigt werden“. Damit ein Entwickler informiert werden kann, gibt es in Gitlab die Möglichkeit Benachrichtigungen per E-Mail und auch per „Slack“ zu konfigurieren. Letzteres ist sehr sinnvoll, da Portaltech Reply das Kommunikationsmittel Slack bereits benutzt. Darüber hinaus kann auch eine angepasste Slack-Benachrichtigung im Buildfile definiert werden. Der Vorteil dieser Lösung ist, dass man nicht nur über Erfolg und Misserfolg benachrichtigt wird, sondern auch bei selbstdefinierten Ereignissen im Buildprozess. Das Listing zeigt ein exemplarische Benachrichtigung per Slack bei Miserfolg des Builds. Dieser Job dient hier nur der Vollständigkeit, da Portaltech Reply die Slack-Integration von GitLab nutzt. Der Grund für das Anführen an dieser Stelle ist, dass dieser Job individueller gestaltet und genutzt wird.

```
notify_pipeline:
  stage: notify
  script: >-
    curl -X POST -H 'Content-type: application/json' --data '
       {"text":
    "Der aktuelle Build Nr Commitnr ist fehlgeschlagen"}'
    https://hooks.slack.com/services/id/id2/key
  when: on_failure
  tags:
  - notify
```

5 Zusammenfassung

Diese Arbeit befasst sich mit der Erstellung einer Automatisierung des Buildprozesses für Shopware-basierte Shops auf Basis einer kontinuierlichen Integration und Deployment, um das Zusammenspiel von Shopware-Features zu testen und die Qualität der Programmierung zu erhöhen, als auch halbautomatisiert zu verteilen. Des Weiteren wird eine eigene Notation und Schnittstelle eingeführt um die Unabhängigkeit von Continuous Integration Server zu gewährleisten.

Es wurde im Rahmen dieser Arbeit eine Notation definiert und eine Schnittstelle in Form einer Konsolenanwendung implementiert und eingeführt Mit dieser Notation hat ein Entwickler nun eine Möglichkeit auf einfache Weise den Buildprozess zu konfigurieren, ohne sich mit dem eingesetzten CI-Server auseinander setzen zu müssen. Auch besteht mit dieser Schnittstelle die Möglichkeit weitere CI-Server anzubinden

oder auszutauschen. Dies soll die Zukunftssicherheit und die Unabhängigkeit des gesamten Buildprozesses bewahren.

Des Weiteren wurde sich mit Containerisierung befasst und sowohl die Schnittstelle, als auch einen Shopware-basierten Shop mit Hilfe von Docker und docker-compose zu einer lauffähigen Anwendung überführt. Der Ansatz mit Docker ist gewählt worden um den Buildprozess im weiteren Verlauf zu standardisieren. Ein Entwickler hat nun die Möglichkeit sein Kundenprojekt sowohl lokal als auch auf den jeweiligen Testservern laufen zu lassen und eine gleichartige Umgebung zu nutzen.

Als Mehrwert bei der Entwicklung des Buildprozesses ist ein Entwicklungs-Stack entstanden. Dieser kann bei der Weiterentwicklung genutzt und mit wenig Aufwand eingesetzt werden. So ist es möglich die Weiterentwicklung nicht nur von einem kleinen Kreis Entwickler abhängig zu machen.

Literaturverzeichnis

[1] Shopware AG. The legacy plugin system. https://developers.shopware.com/developers-guide/legacy-plugin-system/#search-results.

[2] Shopware AG. Plugin quick startup guide. https://developers.shopware.com/developers-guide/plugin-quick-start/#search-results.

[3] Shopware AG. Shopware. https://www.shopware.com/de/.

[4] Atlassian. Gitflow workflow. https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow.

[5] Kent Beck. *Extreme programming explained : embrace change*. Addison-Wesley, Boston, 2. ed., 3. printing edition, 2005.

[6] Sebastian Bergmann. Welcome to phpunit! https://phpunit.de/.

[7] Composer. A dependency manager for php. https://getcomposer.org/.

[8] Martin Fowler. Continuous delivery. https://martinfowler.com/bliki/ContinuousDelivery.html.

[9] Martin Fowler. Continuous integration, 2006. https://www.martinfowler.com/articles/continuousIntegration.html.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[11] Gitlab.com. Issuetracker. https://gitlab.com/gitlab-org/gitlab-foss/issues/54664.

[12] Jez Humble and David Farley. *Continuous Delivery - Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, Amsterdam, 1. aufl. edition, 2010.

[13] Docker Inc. Docker compose. https://docs.docker.com/compose/.

[14] Docker Inc. Modernize your applications, accelerate innovation. https://www.docker.com/.

[15] GitLab Inc. Use docker-in-docker workflow with docker executor. https://docs.gitlab.com/ee/ci/docker/using_docker_build.html#use-docker-in-docker-workflow-with-docker-executor.

[16] Inc. NGINX. Nginx. https://www.nginx.com/.

[17] PHP. Php. https://www.php.net/.