HAW HAMBURG

Bachelorarbeit

Marco Köpcke

Verwaltung von Entwicklungsinstanzen für Webanwendungen über Containervirtualisierung

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Marco Köpcke

Verwaltung von Entwicklungsinstanzen für Webanwendungen über Containervirtualisierung

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

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

Eingereicht am: 27. Mai 2019

Marco Köpcke

Thema der Arbeit

Verwaltung von Entwicklungsinstanzen für Webanwendungen über Containervirtualisierung

Stichworte

Infrastructure as Code, Webentwicklung, Container, Docker, virtuelle Maschine

Kurzzusammenfassung

Ziel dieser Arbeit ist die Implementation eines Systems, das die Entwicklung an Webanwendungen über Containervisualisierung ermöglicht. Wichtiger Bestandteil ist eine auf Webanwendungen optimierte Definitionssprache, um diese über Container starten zu können. Die entwickelte Anwendung soll die Einrichtung einer Entwicklungsumgebung beschleunigen und die Einschränkungen von isolierten Container-Systemen für den Anwender überbrücken. Das System wurde in einer Agentur für Webentwicklung evaluiert.

Marco Köpcke

Title of Thesis

Managing development environments for web applications via container-based virtualization

Keywords

Infrastructure as Code, web development, container, Docker, virtual machine

Abstract

The goal of this thesis is the implementation of a system to develop web applications via container virtualization. An important component is a definition language, optimized on running web applications as containers. The application should accelerate the setup of a development environment and bridge the restrictions of isolated container systems for the user. The system was evaluated in a web development agency.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

1 Einleitung

Containervirtualisierung hat in den letzten Jahren die Entwicklung und Auslieferung von Softwareanwendungen stark beeinflusst. Durch Lösungen wie Docker ist es möglich geworden, ohne virtuelle Maschinen Microservice-Architekturen effektiv umzusetzen. Bestandteile und Abhängigkeiten einer Anwendung lassen sich unabhängig voneinander definieren, kapseln und können plattformunabhängig ausgeführt werden. Außerdem ist Wartung, Einspielung von Updates, und das Austauschen einzelner Dienste möglich, ohne die anderen direkt zu beeinflussen (Wolff, 2018, S. 9-10). Container bieten dabei große Vorteile gegenüber virtuellen Maschinen: „Container sind viel schneller aufgesetzt als virtuelle Maschinen, lassen sich leichter auf verschiedenen Entwicklungssystemen replizieren, [und] beanspruchen weniger Ressourcen“ (Öggl und Kofler, 2018, S. 9).

Es ist möglich, die einzelnen Dienste in Definitionsdateien, beispielsweise über Docker Compose (Docker, 2019g), festzuhalten. Wird Infrastruktur in Code-Dateien festgehalten, so nennt sich dies *Infrastructure as Code* (IaC) (Morris, 2016).

1.1 Motivation

Es gibt verschiedene Möglichkeiten Entwicklungsinstanzen für Webanwendungen einzurichten. Bei einer manuellen Einrichtung benötigen Entwickler Fachwissen oder Unterstützung von Systemadministratoren. Für eine automatische Einrichtung müssen Definitionen für Entwicklungsinstanzen geschaffen werden und geeignete Werkzeuge definiert werden. Bei der automatischen Einrichtung ist Wissen über Infrastructure as Code notwendig.

Manuelle Einrichtung führt zu Problemen, da jeder Entwickler sein System unterschiedlich einrichtet und eine Standardisierung oder Dokumentation von Prozessen schwierig

wird. Es wird viel Zeit benötigt, um Systeme vollständig einzurichten. Ohne standardisierte Prozesse kann es insbesondere für Entwickler, die einem Team neu beitreten, sehr umständlich sein, ihre Entwicklungsinstanz einzurichten.

Automatisierung ist herausfordernd, weil automatische Systeme ebenfalls eingerichtet und gewartet werden müssen. Außerdem müssen die Anforderungen aller beteiligten Entwickler erfüllt sein, um Akzeptanz zu finden (Morris, 2016).

Zudem sind konkrete Lösungen für IaC zumeist auf einzelne Projekte zugeschnitten, da diese andere Anforderungen an Infrastruktur, Sicherheit und Verteilung stellen. Für Entwickler bedeutet dies jedoch eine Herausforderung, insbesondere dann, wenn sie mit vielen verschiedenen Anwendungen und Projekten arbeiten müssen. Es gibt zwar Möglichkeiten IaC über mehrere, ähnliche, Systeme zu vereinheitlichen. Dennoch ist für die Wartung Fachwissen, über das IaC-Framework und gegebenenfalls Konzepte wie Docker, notwendig.

Durch die Verwendung von Docker und IaC Werkzeugen, wie Docker Compose, können Entwickler Systeme schnell hochfahren und alle Abhängigkeiten sind gekapselt. Die Verwendung von Containern verlangt von Entwicklern allerdings ein Verständnis über die Eigenheiten von Virtualisierung und Container. Entwickler müssen manuell mit Containern interagieren, statt direkt mit Werkzeugen und Anwendungen, wie es bei einer manuellen Einrichtung möglich wäre.

Um diese und andere Probleme der Containervirtualisierung zu lösen, gibt es einige anwendungsspezifische Abstraktionswerkzeuge. Ziel dieser Werkzeuge ist es, dass sich der Benutzer weniger mit Docker oder anderen benutzten Konfigurationswerkzeugen auskennen muss. Diese Lösungen sind allerdings meist nur für eine bestimmte Version einer einzelnen Anwendung einsetzbar.

Ein zusätzliches Problem ist dass, für die Definition der IaC-Dateien Fachwissen benötigt wird. IaC-Dateien, die von Anwendungsentwicklern direkt verwendet werden sollen, müssen verständlich und anpassbar sein. Das größte Problem für Entwickler im Umgang mit IaC ist der Syntax, also das fundamentale Verständnis über die Benutzung der Definitionssprachen (Rahman u. a., 2018, S.5). Tiefer liegende Konzepte, wie beispielsweise Volumes von Docker, sollten in so einem Werkzeug für Webanwendungen abstrahiert werden, sodass sie von Entwicklern einfacher verstanden werden.

1.2 Zielsetzung

Im Rahmen dieser Arbeit soll eine Anwendung entworfen und implementiert werden, die es erlaubt, Entwicklungsinstanzen für Webanwendungen in IaC-Dateien festzuhalten und diese zu verwalten.

Die Anwendung greift zum Starten von Komponenten einer Anwendung auf eine Container-Engine, wie Docker, zurück. Es wird ein Proxy-Server entwickelt, über den sich die Container der Projekte auflisten und aufrufen lassen. Kommandozeilen-Werkzeuge für Projekte sollen ebenfalls definierbar sein und der Entwickler soll die Möglichkeit haben, diese über seine Konsole ausführen zu können.

Es wird eine Definitionssprache konzipiert, deren Ziel es ist, einfach, verständlich und optimiert für den Einsatz in Entwicklungssystemen für Webanwendungen zu sein. Es soll möglich sein, hierarchisch Projekte, Anwendungen und Dienste zu definieren und diese auch zu verteilen.

1.3 Gliederung der Arbeit

Die Arbeit ist in acht Kapitel unterteilt. Nach der Einleitung folgt im zweiten Kapitel eine Einführung in die wichtigsten Grundlagen. Das dritte Kapitel definiert Akteure und Anforderungen der Akteure an ein System zur Einrichtung von Entwicklungsinstanzen. Das vierte Kapitel vergleicht bestehende Lösungen zur Virtualisierung von Entwicklungsinstanzen und sammelt Vorteile der einzelnen Lösungen, um diese im fünften Kapitel als Teil der Systemkonzeption nutzen zu können. Im fünften Kapitel werden sämtliche Funktionen des Systems konzipiert, im Hinblick auf die Anforderungen. Das sechste Kapitel beschreibt einige wichtige Implementationsdetails. Im siebten Kapitel werden Qualitätssicherungsmaßnahmen definiert und das System wird evaluiert. Im achten Kapitel wird diese Arbeit zusammengefasst und ein Ausblick gegeben.

2 Grundlagen

In diesem Kapitel werden Grundlagen zum Verständnis dieser Arbeit dargelegt.

2.1 Containervirtualisierung

Container sind eine Kapselung von Software-Anwendungen und deren Abhängikeiten über Mechanismen, die von Betriebsystemkernen (Kernel) bereitgestellt werden (Mouat, 2016, S. 3ff.). Die Sicht auf Hardware-Ressourcen wie Netzwerkkommunikation, Dateisystemzugriff, Geräte- und CPU-Zugriff können dabei vom Kernel für einzelne Anwendungen abgetrennt werden. Diese Art der Virtualiserung wird auch als Containervirtualisierung bezeichnet.

Frühe Implementation der Containervirtualisierung unter Unix-Systemen ist der „chroot-Käfig“. Der chroot Befehl erlaubt es, für Unterprozesse das Wurzelverzeichnis des Dateisystems zu verändern, wodurch die Anwendungen auf Teile des Dateisystems keinen Zugriff mehr haben. Diese Lösung trennt nur das Dateisystem ab und kann mit Root-Rechten umgangen werden (Hogg, 2014). Moderne Betriebssysteme erlauben mit Kernel-Funktionen weitaus mehr Ressourcen abzutrennen, daraus sind Projekte wie Docker entstanden.

Im Unterschied zur herkömmlichen Virtualisierung wird kein ganzes Betriebssystem virtualisiert (Mouat, 2016). Es wird kein Hypervisor benötigt, der verschiedene Betriebssysteme direkt (Typ 1) oder über ein Host-Betriebssystem (Typ 2) mit der Hardware verknüpft, da das Betriebssystem selbst nicht virtualisiert wird (siehe Abbildung 2.1).

Da keine Hardwarezugriffe virtualisiert werden müssen, ist Containervirtualisierung grundsätzlich weitaus performanter als herkömmliche Virtualisierung. Es entsteht kein Overhead durch die Übersetzung von Maschinenbefehle durch einen Hypervisor, sondern der Zugriff findet direkt statt.

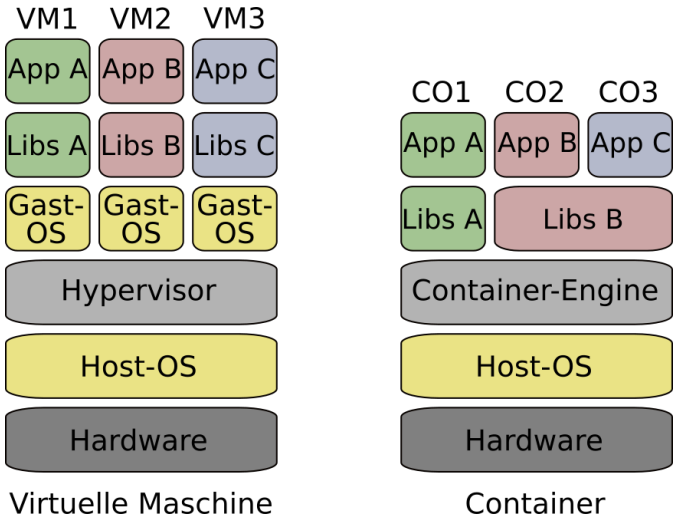


Abbildung 2.1: Vergleich Hypervisor (Typ 2) und Containervirtualisierung (nach Figure 1-1 und Figure 1-2 aus Mouat (2016), S. 5)

Container lassen sich zudem weitaus schneller starten und stoppen als virtuelle Maschinen (VMs), da nicht erst ein virtuelles System inklusive Betriebssystem hochfahren muss. Es lassen sich auch mehr Container auf einem Hostsystem ausführen als VMs, da für die Virtualisierung keine Resourcen, wie Arbeitsspeicher, reserviert werden müssen und kein zusätzliches Betriebssystem Ressourcen verbraucht.

Aus diesem Grund eignen sich Container ideal für die automatische Verteilung von Anwendungen. Wird die Infrastruktur eines Containers in Dateien festgehalten (vgl. *2.3 Infrastructure as Code* und *2.2.1 Images*), lassen sich diese automatisch, schnell und ohne viel Overhead auf andere Systeme verteilen. Da alle Systeme, bis auf den Betriebssystemkern, die gleiche Software-Infrastruktur für die Anwendung besitzen, werden zudem unerwartete Nebenwirkungen dadurch ausgeschlossen (Mouat, 2016).

Einzelne Bestandteile der Anwendung werden dabei üblicherweise nach Microservice-Ansätzen in einzelne Container verpackt. Eine Anwendung, die aus einem Webserver und einer Datenbank besteht, bestünde also aus jeweils einem Container für Webserver und einem für die Datenbank. Beide Container können mit ihren eigenen Abhängigkeiten definiert werden und sind, abgesehen von den Anwendungsschnittstellen, unabhängig voneinander und können bei Bedarf einfach ausgetauscht werden.

Ein Nachteil ist jedoch, dass Anwendungen weniger voneinander isoliert sind und Lücken im Betriebssystem zu Sicherheitsproblemen führen können. Da keine Betriebssysteme

oder Architekturen virtualisiert werden, können zudem nur Anwendungen ausgeführt werden, die auch ohne Containervirtualisierung auf dem Betriebssystem ausgeführt werden könnten.

2.2 Docker

Docker ist eine 2013 veröffentlichte Open Source Technologie zur Containervirtualisierung. Docker ist der Hauptgrund, dass Containervirtualisierung über die letzten Jahre an Beliebtheit gewonnen hat (Mouat, 2016).

Docker baut auf Funktionen im Linux Kernel auf und war ursprünglich eine Erweiterung des Linux Containers (LXC) Projektes, welches Kernel-Technologien wie CGroups, Kernel Namespaces und chroot zusammenführte, um Containervirtualisierung anzubieten.

Es setzt sich von früheren Lösungen durch einfachere Bedienung für Benutzer ab. Viele Konzepte von Docker wurden Basis für Standardisierungen. Die 2015 gegründete Open Container Initiative entwickelt diese Standards.

Dieser Abschnitt behandelt, falls nicht anders angegeben, die Ausführung von Linux Containern unter Linux.

2.2.1 Images

Ein Docker Image ist ein verteilbares Dateisystem, welches die Basis für Container darstellt. Es enthält grundlegende Elemente eines Linux-Dateisystems, alle benötigten Abhängigkeiten der Anwendung des Containers und die Anwendung selbst. Ein Image enthält zudem Informationen über die Ausführung des Containers, wie den Startbefehl und die Definition von Umgebungsvariablen.

Docker Images lassen sich auf verschiedene Arten herstellen, üblicherweise basieren sie auf einer Dockerfile. Eine Dockerfile besteht aus einer Reihe Anweisungen, die Docker mitteilen, aus welchen Dateien das Image besteht und wie Container basierend auf dem Image ausgeführt werden sollen. Quellcode 2.1 (übernommen aus Docker (2019b)) zeigt den Aufbau einer Dockerfile mit einigen Befehlen.

```
1 FROM python:2.7-slim
2 WORKDIR /app
3 COPY . /app
4 RUN pip install --trusted-host pypi.python.org -r requirements.txt
5 CMD ["python", "app.py"]
```

Quellcode 2.1: Beispiel Dockerfile

„FROM“ definiert das Basis-Image des Images. Images können aufeiner aufbauen, alle Befehle der Dockerfile werden in diesem Beispiel also auf dem Image mit dem Tag (Namen) „python:2.7-slim“ ausgeführt. „WORKDIR“ definiert das Arbeitsverzeichnis des Bauvorgangs und auch der späteren Anwendung. „COPY“ kopiert während des Bauvorgangs Dateien vom Host in das Image. „RUN“ führt in einem Container, basierend auf dem aktuellen Image, einen Befehl. „CMD“ setzt den Befehl, der beim Start des Images ausgeführt wird.

2.2.2 Container

In einem Container wird eine Anwendung isoliert zum Rest des Hostsystems ausgeführt. Diese Anwendung ist im Image als Kombination aus dem ENTRYPOINT und CMD definiert. Bei beiden handelt es sich um die Definiton einer ausführbaren Anwendung im Container, mit Argumenten. Der ENTRYPOINT wird von Docker gestartet, dieser sollte den CMD starten. Gibt es keinen ENTRYPOINT startet Docker direkt den CMD (Öggl und Kofler, 2018). Die dadurch gestartete Anwendung besitzt im Container die Prozess ID 0. Wird diese Anwendung beendet, beendet sich auch der Container.

Das Dateisystem des Containers basiert aus mehreren Schichten. Auf der untersten Ebene befindet sich das nur-lesbare Dateisystem aus dem Image. Änderungen, die im Container geschrieben werden verändern das Image nicht, sie beeinflussen nur die Laufzeit des Containers, da diese Änderungen als Schicht über dem Image liegen. Zusätzlich können zu einem Container noch Volumes hinzugefügt werden, welche Daten von außerhalb des Containers in das Dateisystem des Containers einhängen.

Unix-basierte Betriebssysteme besitzen Benutzer und Gruppen, welche durch Nummern repräsentiert werden. Innerhalb von Containern entspricht eine Benutzer-/Gruppennummer der des Hostsystems. Optional kann Docker über User Namespaces

die Zuordnung zwischen Hostsystem und Container verändern (Docker, 2019d). Die Zuordnung von Benutzer- und Gruppennummern zu Namen werden unter Linux in den Dateien „/etc/passwd“ bzw. „/etc/group“ durchgeführt. Container besitzen ihre eigenen Versionen dieser Dateien, daher können Benutzer innerhalb eines Containers andere Namen haben oder nicht existieren, im Sinne dessen, dass sie keinen Eintrag in der „/etc/passwd“ haben.

Der Benutzer 0 (root) hat unter Linux zudem erweiterte Rechte, dies gilt auch für root in Containern. Die Rechte sollten dort zwar durch Isolationsmechanismen insofern eingeschränkt sein, als dass Prozesse mit root-Rechten in Containern nicht ausbrechen können. Sollte ein solcher Prozess jedoch ausbrechen, so hat dieser Vollzugriff auf das gesamte System (Mouat, 2016).

Container können beim Starten Umgebungsvariablen bekommen, die an die gestarte Anwendung übergeben werden. Über Umgebungsvariablen können Einstellungen in der Anwendung gesteuert werden.

2.2.3 Volumes

Um Teile des Dateisystems zwischen Hostsystem und Container auszutauschen, gibt es zwei Arten von Volumes (Docker, 2019i).

Bind Mounts hängen direkt einen Teil des Dateisystems vom Host in einem Docker-Container ein. Vorteil dieser Variante ist die einfache Integration in das Hostsystem, Anwender können Dateien direkt bearbeiten und die Änderungen sind sofort im Container sichtbar.

Named Volumes hingegen werden komplett von Docker verwaltet. Sie sind flexibler als Bind Mounts, da hinter ihnen nicht direkt ein Host-Pfad stehen muss. Der Speicherort des Volumes ist also unabhängig vom Hostsystem. Volume Driver ermöglicht zusätzlich, dass die Quelldateien des Volumes nicht vom Hostsystem selbst, sondern aus dem Netzwerk kommen oder dass die Dateien zwischenverarbeitet werden, beispielsweise durch Verschlüsselung.

2.2.4 Netzwerkkommunikation

Docker verwaltet die Netzwerkkommunikation von Containern in Form von „Netzwerken“. Die Netzwerke können je nach Typ („Treiber“) unterschiedliche Netzwerkfunktionen für den Container zur Verfügung stellen (Docker, 2019f).

Standardtreiber ist „bridge“. Dieser Treiber erstellt ein virtuelles IP-Netzwerk zwischen allen Containern, die Teil des Netzwerks sind und bietet DNS Auflösung anhand definierter Namen für Container im Netzwerk an. Gateway in diesem Bridge-Netzwerk ist das Hostsystem. Container in Bridge-Netzwerken können auf TCP und UDP Ports lauschen, diese sind unabhängig vom Host-System.

Ein weiterer Treiber ist „host“, welcher die Isolation zwischen Hostnetzwerk und Containernetzwerk aufhebt. Beginnt eine Anwendung in einem Container auf TCP Port 8080 zu lauschen, so geschieht dies direkt auf dem Hostsystem und die Anwendung lässt sich über den Port über alle IP-Adressen des Hosts erreichen.

Standardmäßig legt Docker für alle Container ein virtuelles Netzwerk ein (Name „bridge“, Treiber „bridge“), über das alle Container miteinander kommunizieren können (Öggl und Kofler, 2018). Zusätzlich gibt es die Möglichkeit, weitere Netzwerke anzulegen und Containern einzeln diesen zuzuordnen.

Um auf TCP und UDP von Containern Ports zuzugreifen, lassen sich Container-Ports beim Starten eines Containers an Host-Ports binden (Port-Binding). Beispielsweise kann Docker angewiesen werden den Port 8080/tcp eines Containers an den Port 80/tcp am Host zu binden. Netzwerktraffic, der auf dem Hostsystem auf Port 80 eingeht, wird dann an Port 8080 im Container weitergeleitet (Mouat, 2016).

2.2.5 Docker unter macOS und Windows

Docker ist ursprünglich nur für Linux verfügbar, da es auf Funktionen des Linux-Kernels aufbaut. In den letzten Jahren wurden von Docker allerdings mehrere Produkte veröffentlicht, um Docker mit Linux Containern auf Mac und Windows einzusetzen.

Aktuell gibt es die Produkte „Docker for Windows“ und „Docker for Mac“, welche jeweils über Virtualisierungsfunktionen von Windows und macOS eine virtuelle Maschine auf dem Hostsystem starten, in der eine Linux-Distribution mit Docker Daemon läuft (Docker, 2019c).

Auf diesen Betriebssystemen gibt es einige wichtige Unterschiede zur nativen Ausführung auf Linux:

- Da zwischen Hostsystem und Container eine virtuelle Maschine liegt, funktioniert die Netzwerkkommunikation zwischen Host und Container anders. Unter Linux können Container theoretisch über ihre Bridge-Netzwerk IP-Adresse direkt vom Hostsystem angesprochen werden. Dies ist nicht möglich unter Mac und Windows, Port-Binding ist hier zwingend notwendig. Container können den Host nur über den DNS Namen „host.docker.internal“ erreichen (Docker, 2019e).

- Bind Mount Volumes müssen erst in die virtuelle Maschine eingehängt werden, dabei müssen Funktionen der Host- und Gastdateisysteme übersetzt und virtualisiert werden, weswegen auf beiden Betriebssystemen Bind Volumes wesentlich langsamer sind als unter Linux. Unter Mac stehen Funktionen zur Verfügung, um via Einsparungen in der Konsistenz zwischen Host und Container die Performance von Volumes zu erhöhen (Docker, 2019h).

  Unter Mac und Windows müssen Teile der Dateisysteme für die VM freigegeben werden, bevor Bind Mounts für diese Pfade eingesetzt werden können.

- Aufgrund der Hypervisor-basierten Virtualisierung treten entsprechende Einbußen in der Performance und dem Ressourcenverbrauch auf (vgl. *2.1 Containervirtualisierung*).

- Windows besitzt ein eigenes System für Benutzer und Gruppen, dass sich nicht direkt auf Linux-Berechtigungen übertragen lässt. Innerhalb eines Containers besitzen alle Dateien und Verzeichnisse in einem Bind Mount Volume daher die Benutzer- und Gruppennummer 0, nur root kann im Container lesen und schreiben. Auf dem Windows Host gelten die festgelegten Berechtigungen für den Ordner und dessen Inhalt. Dies ist nicht in der offiziellen Dokumentation dokumentiert, sondern wurde unter Docker 18.09 festgestellt.

Unter Windows gibt es zusätzlich die Möglichkeit Windows-Container auszuführen. Es werden dafür Funktionen des Windows-Kernels eingesetzt. In diesem Modus wird keine virtuelle Maschine gestartet, sondern ein Docker Daemon läuft direkt auf dem Windows-Host, es lassen sich allerdings nur Windows-Container nativ ausführen (Öggl und Kofler, 2018).

2.3 Infrastructure as Code

Infructure as Code bezeichnet einen Prozess, in dem Infrastruktur-Elemente in Definitionsdateien festgehalten werden, um die Ausspielung von Änderungen der Infrastruktur zu automatisieren (Morris, 2016). Der Begriff Infrastruktur umfasst dabei unter anderem die Bereitstellung virtueller Server und Installation von Bibliotheken.

Die Definitionsdateien enthalten Anweisungen, die sich auf verschiedene Systeme anwenden lassen. Ziel dieser Anweisungen ist es, Software zu installieren und die Systeme insgesamt zur Ausführung von Software vorzubereiten. Gute Infrastruture as Code Definitionen können sowohl neue Systeme vorbereiten, als auch bestehende Systeme bei Änderungen auf den neusten Stand bringen.

Die Definitionen werden behandelt wie Software-Quellcode und können ebenso wie diese in Versionskontrolle (VCS) versioniert werden. Es ist möglich, Tests für die Definitionen zu schreiben und es lassen sich moderne Entwicklungsstrategien wie Continuous Integration (CI) und Continuous Delivery (CD) damit verwirklichen.

Gegenüber klassischer, manuell verwalteter, Infrastruktur lassen sich so weitaus schneller Systeme auf den neusten Stand bringen, da bei Änderung der Definitionen die Änderungen schnell automatisch auf mehrere Systeme verteilt werden können. Die verwalteten Systeme sind zudem konsistent und einheitlich auf einem bekannten Stand, der durch die aktuellen Definitionsdateien nachvollzogen werden kann.

Durch die erhöhte Geschwindigkeit und Konsistenz lassen sich auch Ausfallzeiten und mögliche Nebeneffekte reduzieren, wodurch das Risiko, dass mit einer Infrastruktur-Änderung einhergeht, sinkt. Dadurch können häufiger und mit weniger Aufwand Änderungen an Infrastruktur durchgeführt werden. Bei fehlerhafter Konfiguration kann die Infrastruktur durch Anwendung alter Definitionsdateien zurückgesetzt werden (Rollback).

Definitionssprachen

Infrastructure as Code verwendet in der Regel für die Definitionsdateien deklarative domänenspezifische Sprachen. In dieser Arbeit wird der Begriff „Definitionssprache“ für diese Art von Sprachen verwendet.

Bei domänenspezifischen Sprachen (DSL) handelt es sich um formale Sprachen, die speziell für den Einsatz mit bestimmten Anwendungen (Domänen) spezifiziert sind. Sie bauen üblicherweise auf bestehenden Sprachen wie XML, JSON oder YAML auf.

In deklarativen Sprachen können gewünschte Zielstände definiert werden („System soll Z sein“). Klassische Programmiersprachen hingegen sind prozedural, bei solchen Sprachen werden konkrete Arbeitsschritte beschrieben („erst mache A, dann B“).

Vorteil von deklarativen Sprachen für Infrastruktur-Definition ist, dass kein Anfangszustand vorausgesetzt wird. Es wird nur definiert, wie das System aussehen soll. Die Anwendung, die die Sprache interpretiert muss basierend darauf bestehende Systeme von ihrem aktuellem Zustand in den Zielzustand überführen (Morris, 2016).

```
1 version: '3'
2 services:
3   web:
4     image: 'beispiel'
5     ports:
6     - '3000:80'
```

Quellcode 2.2: Beispiel Docker Compose

Quellcode 2.2 zeigt die Definitionssprache von Docker Compose, welche als allgemeine Markup-Sprache YAML verwendet. Das Beispiel startet in Verbindung mit Docker Compose einen Docker Container basierend auf dem Image „beispiel“ und bindet den Port 3000 im Container an den Port 80 auf dem Host.

2.4 Bestandteile von Entwicklungsinstanzen für Webanwendungen

Entwickler, die an Webanwendungen arbeiten, benötigen verschiedene Werkzeuge und Programme, um ihre Arbeit durchführen zu können. In dieser Arbeit werden bestehende Lösungen verglichen, um diese Bestandteile automatisiert zur Verfügung zu stellen. Es

wird danach eine Anwendung konzipiert, um diese über Containervirtualisierung bereitzustellen. Aus diesem Grund werden in diesem Abschnitt kurz die wichtigsten Bestandteile und ihre Funktionen beschrieben.

Prorgammiersprache, Compiler, Intepreter, Bibliotheken

Zunächst werden die Intepreter und/oder Compiler für die in der Entwicklung eingesetzten Programmiersprachen benötigt. Diese müssen jeweils in einer Version vorliegen, die von der Anwendung benötigt wird. Die eingesetzten Anwendungen und Frameworks bringen Anforderungen für Bibliotheken in bestimmten Versionen mit, die ebenfalls installiert werden müssen.

Webserver

Es gibt Anwendungen und Frameworks, die so kompiliert oder interpretiert werden, dass sie selbst einen Webserver ausführen, wie beispielsweise Webanwendungen basierend auf Java Spring, Python Tornado oder Node.js basierte Webserver.

Es gibt allerdings auch Anwendungen, welche die Ausführung eines zusätzlichen Webservers (Nginx, Apache) benötigen. Diese Anwendungen basieren auf Scriptsprachen wie PHP. Diese Sprachen benötigen im Webserver ein Modul oder eine Schnittstelle für das Common Gateway Interface (CGI), um bei Ausspielung ihrer Scriptdateien diese durch den Sprachinterpreter verarbeiten zu lassen. CGI ist ein Protokoll für Webserver, um externe Interpeter anzuschließen (Robinson und Coar, 2004). Werden solche Programmiersprachen eingesetzt, benötigen Entwickler neben der Installation der Programmiersprache auch die Installation des eigenständigen Webservers, welcher mit dem Interpreter verbunden werden muss.

Zusätzliche Dienste

Neben den Werkzeugen der eingesetzten Programmiersprachen benötigt die Anwendung möglicherweise zusätzliche externe Anwendungen, wie beispielsweise Datenbankserver oder Caching-Server. Diese müssen in den jeweilig unterstützten Versionen installiert werden.

CLI- und GUI-Schnittstellen

Während der Entwicklung müssen Entwickler mit der Webanwendung interagieren. Neben der Anwendung selbst, die in der Regel über einen Webbrowser oder andere HTTP-Clients bedient werden kann, stehen für die Anwendung möglicherweise weitere Werkzeuge zur Verfügung. Diese stehen entweder in Form graphischer (GUI) oder konsolenbasierter (CLI) Schnittstellen bereit. Für Node.js Anwendungen sind dies beispielsweise Werkzeuge wie „node“ (der Interpreter selbst), „npm“ (die Paketverwaltung) oder „gulp“ (ein Automatisierungswerkzeug). Zudem bringen Anwendungen und Frameworks gegebenenfalls ihre eigenen Werkzeuge mit, wie beispielsweise das „magento“ CLI-Werkzeug von Magento 2 oder „ng“ für Angular Anwendungen.

Debugger

Um Fehler während der Entwicklung zu finden und Vorgänge im Inneren der Anwendung nachvollziehen zu können, werden zudem Debugger eingesetzt. Debugger erlauben es, die Ausführung des Programmcodes anzuhalten, aktuelle Variablenbelegungen anzuzeigen, eigene Befehle auszuführen und die Ausführung des Programmcodes schrittweise fortzuführen. Der Debugger für die jeweilige Programmiersprache muss installiert und eingerichtet sein, um benutzt zu werden.

Weitere Werkzeuge

Unter dem Begriff *Entwicklungsinstanz* werden in dieser Arbeit nur Werkzeuge, Anwendungen und Bibliotheken gefasst, die mit der direkten Ausführung der Anwendung zusammenhängen. Neben diesen gibt es weitere Werkzeuge, die ein Entwickler zum effektiven Arbeiten benötigt. Dazu gehören unter anderem Werkzeuge zur Versionsverwaltung (VCS), ein Texteditor oder eine integrierte Entwicklungsumgebung (IDE), ein Webbrowser und eine Konsole (Terminal). Für die Lösungen, die in dieser Arbeit vorgestellt und konzipiert werden, wird angenommen, dass solche Werkzeuge bereits installiert sind.

2.5 Eingesetzte Webtechnologien

Als Teil der in dieser Arbeit konzipierten Anwendung wird ein HTTP Reverse Proxy-Server entwickelt. In diesem Abschnitt werden dafür verwendete Technologien kurz vorgestellt.

2.5.1 HTTP Reverse Proxy

Ein HTTP (Forward) Proxy-Server ist ein Server, der zwischen Anwender und Website steht, um für den Anwender Anfragen zum Server auszuführen. Der Anwender muss auf seinem System konfigurieren, diesen Proxy zu nutzen. Einsatzgebiet solcher Proxies ist innerhalb von Firmen, um interne Systeme durch den Proxy mit Webservern zu verbinden. Der Proxy kann dabei auch Techniken wie Caching einsetzen, um den Traffic im Netzwerk für die Benutzer des Proxies zu reduzieren.

Ein HTTP Reverse Proxy (oder Gateway) hingegen erscheint dem Anwender wie ein normaler Webserver. Der Reverse Proxy ist konfiguriert, um eingehende Anfragen an einen oder mehrere HTTP Server weiterzuleiten und die Antwort zurück an den Benutzer zu schicken. Reverse Proxies werden eingesetzt, um mehrere Backend-Server hinter einem Webserver zu vereinen (Load-Balancer). Sie können auch anhand von Regeln, wie dem HTTP Hostnamen, an verschiedene Content-Server weiterleiten (Apache, 2019).

2.5.2 WebSockets

WebSockets erlauben bi-direktionale Kommunikation von Client und Server im Browser. Klassischerweise ist dies im HTTP Protokoll nicht direkt möglich. Der Client schickt eine HTTP Anfrage an den Server und erhält daraufhin eine Antwort. Es war nicht möglich für den Server eine Nachricht an einen Client zu schicken, ohne dass Dieser direkt eine Anfrage geschickt hat.

Eine Lösung für das Problem ist HTTP Polling, bei dem der Client wiederholt den Server nach neuen Nachrichten fragt. Diese Lösung ist umständlich und bringt eine Reihe Nachteile mit sich. Aus diesem Grund wurde das WebSocket Protokoll geschaffen (Fette und Melnikov, 2011).

Das WebSocket Protokoll beschreibt, wie über HTTP Verbindungen der Client eine bidirektionale Verbindung zwischen Server und Client herstellen kann. Über diese Verbindung können Client und Server dann frei Nachrichten austauschen.

Für JavaScript gibt es in modernen Browsern eine Standardimplementierung und für etliche Server-Frameworks stehen ebenfalls Implementierungen zur Verfügung.

2.6 Namensauflösung und *hosts*-Datei

Hostnamen, wie *www.haw-hamburg.de*, können verwendet werden, um Server und Webseiten über das Internet zu erreichen. Diese Hostnamen werden dafür in IP-Adressen übersetzt (aufgelöst), um eindeutig den Traffic an einen Server zu routen.

Um Hostnamen aufzulösen gibt es das Domain Name System (DNS), welches Hostnamen im Internet eindeutig Systemen zuordnen kann.

Neben DNS gibt es auf modernen Betriebssystemen eine *hosts*-Datei. Diese Datei enthält eine Zuordnung von Hostnamen zu IP-Adressen und wird vom Betriebssystem in der Regel vor dem DNS angefragt, um Hostnamen aufzulösen (Linux, 2017). Unter Linux liegt diese Datei unter */etc/hosts*. Diese Datei wird unter anderem von Webentwicklern eingesetzt, um auf lokalen Entwicklersystemen Hostnamen für die Entwicklung auf ihre eigene IP-Adresse umzuleiten.

Einige Protokolle beinhalten die Angabe des Hostnamens als Teil des Protokolls. Dies führt bei HTTP beispielsweise dazu, dass Reverse Proxies anhand eines Hostnamen auf verschiedene Backend-Server weiterleiten können (Fielding u. a., 1999, Abschnitt 14.23).

3 Anforderungsanalyse

In diesem Kapitel werden Anforderungen an Entwicklungssysteme für Webanwendungen definiert.

Es wird festgelegt, welche Akteure an der Einrichtung von Entwicklungsinstanzen beteiligt sind und es werden bestehende Lösungen zum Einrichten eines Systems betrachtet und anhand der Anforderungen verglichen.

Die in diesem Kapitel erarbeiteten Anforderungen und bestehenden Lösungen bilden die Grundlage für die Anwendung, die in den darauf folgenden Kapiteln konzipiert wird.

3.1 Akteure

An der Einrichtung und Ausführung einer Entwicklungsinstanz sind mehrere Akteure beteiligt. *Akteur* beschreibt in diesem Zusammenhang eine Gruppe von Benutzern, die bestimmte Erwartungen und Anforderungen gegenüber dem Prozess haben. Dieser Abschnitt definiert verschiedene Akteure und grenzt diese voneinander ab.

Als zusätzliche Akteure können Kunden und Projektmanager von Softwareprojekten gesehen werden. Diese haben auch Anforderungen an Entwicklungsinstanzen. Sie möchten, dass das Softwareprojekt erfolgreich entwickelt werden kann und die Anwendung am Ende des Projektes im vollen Umfang funktioniert. Da diese Akteure allerdings nicht direkt mit Entwicklungsinstanzen interagieren und ihre Anforderungen sich in denen der Entwickler und Systemadministratoren wiederspiegeln, werden sie nicht weiter behandelt.

3.1.1 Entwickler

Entwickler sind diejenigen, die mit den Entwicklungsinstanzen arbeiten müssen, um Webanwendungen zu entwickeln. Sie erwarten, dass die Anwendung zur Verfügung steht, inklusive all ihrer Abhängigkeiten, zusätzlicher Dienste und gängiger Hilfswerkzeuge zur Entwicklung (Hüttermann, 2012).

Entwickler kennen sich mit der Anwendung aus, aber nicht unbedingt mit der Installation einer Entwicklungsinstanz. Entwickler können zwar Abhängigkeiten, in Form von Bibliotheken für die Anwendung, installieren und aktualisieren, sind aber nicht detailliert mit der Einrichtung zusätzlicher Dienste, wie Datenbank oder Webserver vertraut.

Es ist möglich, dass die Anwendung durch Änderungen am Code während der Entwicklung nicht korrekt funktioniert. Entwickler sind in der Lage solche Fehler zu beheben, erwarten jedoch, dass die Anwendung und ihre Hilfsmittel darüber hinaus grundsätzlich funktionieren und ihm ermöglichen, effektiv an der Anwendung zu entwickeln.

Sie erwarten von Systemadministratoren, dass Prozesse bereitstehen, mit denen sie schnell ihre Entwicklungsinstanzen aufsetzen können. Die Kernaufgabe von Entwicklern ist das Entwickeln an der Webanwendung, nicht die Einrichtung des Systems.

Entwickler können auch selbst die Rolle von Systemadministratoren für ihre eigenen Entwicklungssysteme einnehmen. Dies kann ein ungewollter, gezwungener, Nebeneffekt sein, da im Team keine Systemadministratoren für Entwicklungsinstanzen vorgesehen sind. Es kann allerdings auch im Prozess beabsichtigt sein, beispielsweise in DevOps Prozessen. DevOps bezeichnet einen Prozess, in dem die Übergänge zwischen Softwareentwicklung und Systemadministration verschwimmen (Hüttermann, 2012). Einige Entwickler in solchen Entwickler-Teams besitzen zum Teil auch das Wissen und die Aufgaben von Systemadministratoren.

3.1.2 Systemadministratoren

Systemadministratoren kennen sich mit der Einrichtung eines Systems für die Webanwendung aus. Sie können die Anwendung mit ihren Abhängigkeiten einrichten und kennen sich detailliert mit den Einstellungsmöglichkeiten der beteiligten Anwendungen aus (Hüttermann, 2012).

Systemadministratoren entwickeln nicht an der Anwendung selbst mit. Ihre Aufgabe ist es, Systeme, auf denen die Anwendung ausgeführt wird, einzurichten. Dazu können auch Entwicklungsinstanzen gehören. Die Arbeit von Systemadministratoren stellt die Grundlage dar, unter der Entwickler arbeiten können.

Systemadministratoren möchten nicht einzelne Systeme manuell einrichten oder sich mit den Besonderheiten einzelner Arbeitscomputer auseinander setzen (Morris, 2016), sondern erwarten Möglichkeiten, geordnete Prozesse zur Einrichtung solcher Systeme definieren und steuern zu können.

3.2 Anforderungen an Entwicklungsinstanzen für Webanwendungen

Die Akteure Entwickler und Systemadministrator besitzen Anforderungen an Entwicklungsinstanzen, die die Basis für alle weiteren Analysen und Konzeptionen bieten. Diese Anforderungen werden in diesem Abschnitt in einzelnen User-Stories aufgeschlüsselt.

Die hier definierten Anforderungen sind bewusst kurz und allgemein gehalten. Konkrete Anforderungen können unter anderem vom Entwickler-Team, einzelnen Entwicklern und der eingesetzten Anwendung abhängen. Diese allgemeinen Anforderungen fassen allerdings grundlegende Ziele zusammen, die notwendig sind für gut funktionierende Entwicklungsinstanzen.

3.2.1 Ausführung und Bedienung

Entwickler besitzen an ihre Entwicklungsinstanz gewisse Anforderungen zur *Ausführung und Bedienung*. Ihr oberstes Ziel ist es, ihre Anwendung weiter zu entwickeln. Sie müssen daher mit der Entwicklungsinstanz im vollen Umgang arbeiten können, ohne dass diese sie bei ihrer Arbeit behindert.

A1.1 Einrichtungsaufwand. Als Entwickler möchte ich eine Entwicklungsinstanz mit möglichst wenig Aufwand aufsetzen können, um schnell mit meiner Arbeit beginnen zu können.

A1.2 Funktionsumfang der Anwendung. Als Entwickler möchte ich, dass meine Entwicklungsinstanz mir alle Funktionen der Anwendung, an der ich arbeite, bereit stellt, damit ich in vollem Umfang an ihr arbeiten kann.

A1.3 Erreichbarkeit der Anwendung via HTTP. Als Entwickler möchte ich via Browser oder HTTP-Client auf meine Webanwendung zugreifen können, um mit ihr interagieren zu können.

A1.4 Verfügbarkeit externer Dienste. Als Entwickler möchte ich auf Schnittstellen externe Dienste, wie Datenbanken oder Caching-Server über TCP und/oder UDP zugreifen können, um diese im Rahmen meiner Arbeit zu verwalten.

A1.5 Verwaltung von Daten und Dateien. Als Entwickler möchte ich Daten und Dateien meiner Anwendung verwalten können. Dazu gehören Daten aus angeschlossenen Datenbanken, welche ich beispielsweise importieren, exportieren oder manipulieren können möchte. Dazu gehören ebenfalls Quellcode-Dateien, Bilder oder andere Dateien, die Teil der Anwendung sind.

A1.6 Debugging. Als Entwickler möchte ich den Debugger der eingesetzten Programmiersprache verwenden können, um Fehler in der Anwendung einfacher nachvollziehen zu können und Lösungen für Probleme zu finden.

A1.7 Hilfswerkzeuge. Als Entwickler möchte ich zur Anwendung passende GUI- und CLI-Hilfsanwendungen nutzen, um mir die Arbeit zu erleichtern.

A1.8 Logdateien. Als Entwickler möchte ich Einblick auf Logdateien (Protokolle bestimmter Aktionen und Fehlerfälle) meiner Anwendung und aller beteiligten Systeme haben, damit diese mich bei der Entwicklung und Fehlerbehandlung unterstützen.

A1.9 Wechsel zwischen verschiedenen Versionsständen. Als Entwickler möchte ich verschiedene Versionsstände von Code und Datenbanken anwenden können, um zwischen verschiedenen Anwendungsversionen zu wechseln, sodass ich diese vergleichen oder an mehreren Funktionen unabhängig voneinander entwickeln kann.

A1.10 Arbeit an mehreren Projekten. Als Entwickler möchte ich an mehreren Projekten gleichzeitig arbeiten können und die Arbeit an verschiedenen Entwicklungsinstanzen schnell wechseln können. Die verschiedenen Projekte besitzen dabei möglicherweise Anforderungen an verschiedene Versionen von Programmiersprachen oder Bibliotheken. Meine Entwicklungsinstanzen müssen mir für alle Anwendungen die jeweils passenden Versionen bereitstellen.

A1.11 Kompatibilität mit Betriebssystemen. Als Entwickler möchte ich, dass die Entwicklungsinstanz auf meinem Betriebssystem funktioniert, damit ich damit arbeiten kann.

3.2.2 Definition und Installationsprozess

Systemadministratoren besitzen Anforderungen an *Definition und Installationsprozess* der Entwicklungsinstanzen. Ihr oberstes Ziel ist es, Entwicklungsinstanzen geordnet und wohldefiniert Entwicklern zur Verfügung zu stellen. Änderungen am Prozess und den Instanzen müssen sich unkompliziert und schnell durchführen lassen, da nicht funktionierende Entwicklungsinstanzen Entwickler von ihrer Arbeit abhalten (Morris, 2016).

A2.1 Dokumentation. Als Systemadministrator möchte ich die Einrichtung von Entwicklungsinstanzen dokumentieren können, damit ich und andere Benutzer der Instanzen diese nachvollziehen können.

A2.2 Anwendbarkeit auf verschiedene Systemen. Als Systemadministrator möchte ich, dass sich der Einrichtungsprozess für Entwicklungsinstanzen auf die verschiedenen Betriebssysteme und Arbeitsumgebungen der Entwickler einheitlich anwenden lässt, um einen einfachen, einheitlich und nachvollziehbaren Prozess einrichten zu können.

A2.3 Generalisierung verschiedener Projekte. Als Systemadministrator möchte ich Gemeinsamkeiten verschiedener Projekte zentral vereinheitlichen, um Gesamtaufwand der Definition für viele verschiedene Entwicklungsinstanzen für verschiedene Projekte möglichst gering zu halten.

A2.4 Automatisierung. Als Systemadministrator möchte ich, dass von mir definierte Entwicklungsinstanzen sich möglichst automatisiert anwenden lassen, um Zeit zu sparen und sicherzustellen, dass sich keine menschlichen Fehler in den Prozess einschleichen.

A2.5 Anwendung von Änderungen. Als Systemadministrator möchte ich einfach Änderungen an Entwicklungsinstanzen definieren und bei Entwicklern anwenden können, um Aufwand zu sparen und schnell auf Änderungswünsche eingehen zu können.

A2.6 Fehlerbehandlung. Als Systemadministrator möchte ich in Fehlerfällen nachvollziehen können, warum Entwicklungsinstanzen nicht korrekt funktionieren, um Probleme möglichst schnell zu lösen, sodass Entwickler schnell wieder an ihren Systemen arbeiten können.

A2.7 Reproduktion des Produktivsystems. Als Systemadministrator möchte ich, dass die Entwicklungsinstanzen die eingesetzten Produktivumgebungen möglichst exakt wiedergeben. Dies soll verhindern, dass Unterschiede in der Infrastruktur ungewollte Auswirkungen haben, wenn eine Änderung an der Anwendung auf das Produktivsystem übertragen wird.

3.3 Manuelles Einrichten von Entwicklungsinstanzen

In diesem Abschnitt wird die manuelle Einrichtung von Entwicklungsinstanzen betrachtet. Anhand dieser Beschreibung sollen die Anforderungen aus dem vorherigen Abschnitt und Probleme, die beim Einrichten einer Entwicklungsinstanz auftreten können, verdeutlicht werden. Dabei wird zunächst davon ausgegangen, dass Entwickler ihre Instanzen selbst einrichten und kein Systemadministrator.

Soll eine Entwicklungsinstanz auf dem Computer eines Entwicklers manuell eingerichtet werden, so müssen von Hand alle Anwendungen, Werkzeuge und Bibliotheken installiert und eingerichtet werden.

Der Entwickler kann hierfür einer Dokumentation folgen. Diese muss allerdings auf die von ihm eingesetzte Version der Anwendung, ihrer Abhängigkeiten und dem Betriebssystems des Entwicklers zugeschnitten sein. Unerwartete Einstellungen am System des Entwicklers, die bei der Erstellung der Dokumentation nicht berücksichtigt wurden, können

dabei zu Problemen führen. Insgesamt ist daher ein hohes Verständnis des Entwicklers über die betroffenen Anwendungen und seinen Computer notwendig.

Die Dokumentation für die Einrichtung zu erstellen und aktuell zu halten ist aus den genannten Gründen ebenfalls aufwendig. Durch projektspezifische Unterschiede und der verschiedenen Systeme einzelner Entwickler ist es schwierig, einheitliche Prozesse zu definieren. Es ist nur möglich die Dokumentation und damit den Prozess zu testen, indem die gesamte Anleitung zum Installieren einer Entwicklungsinstanz bei Änderungen manuell auf allen Systemen getestet wird.

Da die verschiedenen Entwicklersysteme auf unterschiedlichen Bibliotheken und Betriebssystemen basieren können, ist es auch möglich, dass entwickelte Funktionen zwar auf den Systemen einzelner Entwickler funktionieren, aber nicht auf anderen Testsystemen oder dem Produktivsystem.

Ähnliche Probleme schildert Morris (2016) im Kapitel *Challenges with Dynamic Infrastrcuture* für die Einrichtung von Cloud-Servern. Manuelle Server-Einrichtung führt zu *Schneeflocken-Servern*. Schneeflocken-Server sind Server, deren Konfigurationen sich vom ursprünglich definierten Standard unterscheiden, beispielsweise weil für einzelne Server Optimierungen durchgeführt oder Fehler behoben wurden. Schneeflocken-Server führen zu *fragiler Infrastruktur*. Fragile Infrastruktur beschreibt Infrastruktur, die nicht leicht verändert oder gewartet werden kann. Grund dafür sind die vielen voneinander abweichenden Schneeflocken-Server. Bei manuell eingerichteten Entwicklungsinstanzen handelt es sich um Schneeflocken-Server und die gesamte Infrastruktur der Entwickler ist damit fragil.

Erfahrene Entwickler, die sich gut mit den Bestandteilen für das Aufsetzen einer Entwicklungsinstanz auskennen, können sich Entwicklungsinstanzen einrichten, die ihren persönlichen Anforderungen genügen. Weniger erfahrene Entwickler sind allerdings davon abhängig mit Hilfe erfahrenerer Entwickler oder Systemadministratoren ihre Instanzen aufzusetzen.

Einige dieser Probleme könnten gelöst werden, indem Systemadministratoren auf festgelegten Betriebssystemen Entwicklungsinstanzen für die Entwickler einrichten und testen. Dies setzt allerdings voraus, dass Änderungen am System nur Systemadministratoren durchführen. Nur so kann sichergestellt werden, dass diese jederzeit über den Gesamtzustand der Entwicklungsinstanzen Bescheid wissen und Support geben können. Entwickler

sind dann auf die Systeme angewiesen, wie sie die Administratoren einrichten. Entwickler müssen in Absprache mit Systemadministratoren Änderungen durchführen. Da alle Änderungen von Hand durchgeführt werden, entsteht bei großen Entwickler-Teams viel Arbeit für Systemadministratoren. Bei Änderungen an den Entwicklungsinstanzen müssen Systemadministratoren sicherstellen, dass alle Entwicklersysteme auf den neusten Stand gebracht werden. Solange sind die Entwickler an ihrer Arbeit behindert. Es entsteht dadurch insgesamt viel Aufwand für die Wartung der Entwicklungsinstanzen und minimale Flexibilität für die Entwickler.

In einem manuellen Einrichtungsprozess ist es nur sehr schwer möglich Anforderungen an Definition und Installationsprozess zufriedenstellend zu erfüllen. Es handelt sich um einen Prozess, der sich schwer dokumentieren und vereinheitlichen lässt. Je nach Projekt kann er zudem sehr komplex sein. Bei Fehlern ist es schwer eine Ursache zu finden, da diese auch stark vom System des einzelnen Entwicklers abhängig sind. Es ist schwer Aufwände zu planen, da diese von Erfahrung und Präferenzen des Entwicklers und der Aktualität der bestehenden Dokumentation abhängen.

3.4 Automatisierung der Einrichtung von Entwicklungsinstanzen

Eine Lösung, um Anforderungen an Definition und Installationsprozess besser zu genügen, ist das Einrichten von Entwicklungsinstanzen zu automatisieren und damit zu standardisieren. Für diesen Zweck gibt es viele Werkzeuge, einige davon nutzen Virtualisierung oder Containervirtualisierung, während andere die Installation von Software direkt auf den Systemen der Entwicklern durchführen.

Diese Werkzeuge (Tools) erlauben es nach Mustern von *Infrastructure as Code* die Entwicklungsumgebungen in Dateien zu definieren. Entwickler können mittels der Werkzeuge ihre Entwicklungsinstanzen basierend auf den Dateien einrichten. Die Definitionsdateien können von Systemadministratoren oder DevOps-Entwicklern verändert werden und Entwicklern zugespielt werden. Die Tools können dann die bestehenden Entwicklungsinstanzen anhand der neuen Definitionen auf den aktuellen Stand bringen.

Automation Fear

Einen automatisierten Prozess einzusetzen kann kompliziert sein, wenn die verschiedenen Entwickler sich ihre eigenen Systeme nach dem Prinzip der Schneeflocken-Server bisher manuell eingerichtet haben. Grund dafür ist, dass eine Umstellung und Vermischung von manuellen und automatischen Prozessen zu *Automation Fear* führt (Morris, 2016). Dabei handelt es sich um die Angst, die bestehenden manuellen Prozesse durch vollständig automatische Prozesse zu ersetzen.

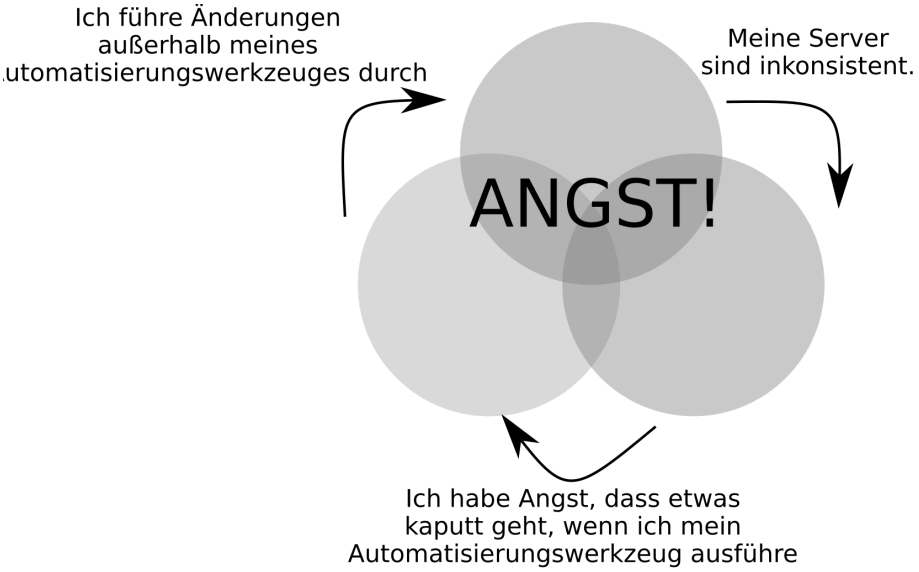


Abbildung 3.1: Kreislauf der *Automation Fear* (nach Figure 1-1 aus Morris (2016), S. 9)

Die vielen unterschiedlichen Erwartungen, die sich Entwickler an ihre eigenen Systeme gesetzt haben, führen dazu, dass Automatisierungsprozesse nur teilweise und individuelle Anpassungen manuell angewendet werden. Da die Systeme dadurch inkonsistent zueinander werden, besteht dann die Angst, dass die Ausführung der Automatisierungswerkzeuge zu Fehlern führt (siehe Abbildung 3.1). Automation Fear führt zu schlecht einsetzbaren Automatisierungsprozessen, die am Ende keine der Anforderungen zufriedenstellend erfüllen können und die Entwickler bei ihrer Arbeit behindern.

Die Einrichtung eines automatischen Prozesses zum Einrichten von Entwicklungsinstanzen bedeutet daher auch, sich von manuellen Prozessen und Schneeflocken-Servern zu trennen. Es ist notwendig, dass die Entwickler einige Kompromisse in ihren eigenen Anforderungen an Ausführung und Bedienung eingehen. Das Team, dass einen solchen Prozess integrieren soll, muss den Wünschen des Entwickler-Teams entgegen kommen.

Das Werkzeug zur Automatisierung muss flexibel genug sein, um sich den individuellen Anforderungen eines Entwickler-Teams anzupassen. Es ist wichtig, dass sich mit dem Werkzeug möglichst viele Anforderungen an Ausführung und Bedienung einfach umsetzen lassen, da es ansonsten schwer ist, Akzeptanz im Entwickler-Team zu finden und der Automation Fear entgegen zu wirken.

4 Vergleich bestehender Lösungen

In diesem Kapitel werden bestehende Lösungen zur Automatisierung von Infrastruktur basierend auf der Anforderungsanalyse verglichen. Stärken und Schwächen der Lösungen werden gesammelt und bieten die Grundlage für die Spezifikation des in dieser Arbeit konzipierten Systems.

4.1 Allgemeine Lösungen zur Automatisierung

Die in diesem Abschnitt vorgestellten Lösungen sind allgemeine Lösungen zur Automatisierung von Infrastruktur. Alle, bis auf Vagrant, sind nicht für die Ausführung von Entwicklungsinstanzen optimiert, sondern sind eher für den Einsatz von Staging- und Produktivumgebungen konzipiert. Je nach Endanwendung kann es aufwendig sein, diese Tools so zu optimieren, dass sie auch für Entwicklungsumgebungen zufriedenstellend einsetzbar sind.

4.1.1 Ansible

Ansible ist eine 2012 entwickelte Open-Source Software zum Konfigurationsmanagement, automatischem Deployment (Verteilen) von Anwendungen und zur Provisionierung (Vorbereitung) von Servern (Coca u. a., 2019).

Ansible erlaubt es als Teil sogennanter *Playbooks* Server zu konfigurieren. Bei Playbooks handelt es sich um YAML Dateien, die Anweisungen in einer Definitionssprache enthalten.

Die einzelnen Anweisungen in Playbooks basieren auf *Modulen*. Ein Modul kann eine bestimmte Art von Ressource verwalten. So gibt es beispielsweise Module, um Paketmanager von Linux Distributionen, Git Repositories oder Dateien zu verwalten. Einige

Module sind vom Betriebssystem unabhängig einsetzbar, wohingegen andere bestimmte Betriebssysteme oder Software vorausetzen.

Die Definitionen im Playbook beschreiben einen gewünschten Zustand des Systems. Diese Beschreibungen sind idempotent: Ein Playbook lässt sich mehrmals mit der gleichen Konfiguration ausführen und muss dann das System immer wieder in den gleichen Zustand bringen.

Die Liste der zu verwaltenden Server lassen sich im *Inventory* verwalten. Für bestimmte Server und Umgebungen lassen sich Variablen festlegen, die nur für diese Umgebungen und Server gelten. Variablen lassen sich an vielen Stellen in Playbooks definieren und einsetzen. Der Syntax für die Benutzung von Variablen basiert auf Jinja, einer bekannten Templating-Engine. Es lassen sich ganze Konfigurations-Dateien für Anwendungen als Teil eines Playbooks in Templates definieren und von Jinja auswerten. So kann beispielsweise für das Deployment eines Nginx Webservers, die Webserver-Konfiguration als Template im Playbook abgelegt werden. Im Template können dann Ansible Variablen verwendet werden, die beim Deployment ausgefüllt werden.

Playbooks lassen sich in *Rollen* aufteilen. Rollen sind Unteraufgaben, die auch über mehrere Playbooks verteilt eingesetzt werden können. Sie erlauben somit die Zerlegung von Playbooks in mehrere Bausteine.

Aufgrund der Flexibilität und des Funktionsumfangs von Ansible lassen sich mit gut definierten Playbooks die meisten Anforderungen an Entwicklungsinstanzen erfüllen.

Diese Flexibilität ist allerdings auch ein Nachteil. Es ist notwendig, dass die Playbooks sorgfältig angefertigt und sinnvoll in Rollen zerlegt werden, um auf längere Sicht wartbar zu sein. Je nach konkreten Anforderungen der Entwickler-Teams kann es aufwendig sein, entsprechende Playbooks zu definieren.

Ein potentielles Problem sind insbesondere die Anforderungen A2.2 und A2.7. Da Ansible grundsätzlich ohne Virtualisierung oder Containervirtualisierung arbeitet, werden alle Änderungen direkt auf den Computern der Entwickler ausgeführt. Arbeiten die Entwickler mit vielen verschiedenen Betriebssystemen, so ist es schwer, Playbooks zu definieren, die auf allen Systemen funktionieren. Ansible Playbooks sind zudem eigentlich für den Einsatz auf Servern ausgerichtet. Server sollten sich in der Regel, bis auf die Anwendung der Playbooks, kaum ändern. Entwickler-PCs werden aber konstant durch Entwickler bedient. Kleinere Anpassungen der Entwickler an ihren Systemen können dazu führen, dass Playbooks nicht vollständig angewendet werden können.

Da keine Virtualiserung eingesetzt wird, können Probleme auftreten, wenn an mehreren Projekten gerarbeitet wird. Die verschiedenen Projekte könnten verschiedene Versionen der eingesetzten Programmiersprache vorausetzen. Die Ansible Playbooks müssen so konzipiert sein, dass sie in solchen Fällen die Installation von verschiedenen Versionen erlauben. Da die eingesetzten Programmiersprachen und Bibliotheken dies eventuell grundsätzlich nicht erlauben, müssen Hilfsmittel eingesetzt werden, was die Komplexität erhöht.

Mit Ansible lassen sich Entwicklungsinstanzen automatisiert installieren. Aufgrund der fehlenden Virtualisierung müssen die Playbooks für alle eingesetzen Entwickler-PCs angepasst werden. Die Einrichtung und Wartung guter Playbooks kann aufgrund der Flexibilität von Ansible und den Anforderungen der Entwickler aufwendig sein.

Neben Ansible gibt es weitere ähnliche Werkzeuge wie Puppet, Chef und Salt, die ähnlich funktionieren wie Ansible.

Ansible und ähnliche Werkzeuge können auch in Kombination mit den anderen, hier vorgestellten, Lösungen verwendet werden. Es gibt Ansible Module zur Verwaltung von Docker Containern, Kubernetes Clustern und Vagrant VMs.

4.1.2 Vagrant

Vagrant ist ein Open-Source Softwareprojekt zum Verwalten von Virtuellen Maschinen für Entwicklungsinstanzen (Roberts u. a., 2019). Die Entwicklung an Vagrant begann im Jahr 2010.

Vagrant erlaubt es virtuelle Maschinen zu definieren und diese basierend auf Basis-Abbildern (*Boxes*) zu bauen und zu provisionieren (durch Scripte vorzubereiten).

Die Konfigurationen werden projektweise definiert , in einer Datei mit dem Namen *Vagrantfile*. Die Interaktion mit den virtuellen Maschinen findet primär über eine Konsole, basierend auf dem Secure Shell Protocol (SSH), statt. Über einen Befehl der Vagrant CLI können Entwickler sich in die Konsole einer virtuellen Maschine einwählen.

Darüber hinaus können Ordner mit der VM geteilt werden, um beispielsweise den Quellcode der Anwendung der VM bereitzustellen. Es können Ports an das Hostsystem weitergeleitet werden, sodass beispielsweise TCP Port 3000 der VM über Port 80 des Hostsystems erreichbar ist. Dies kann zu Problemen führen, wenn mehrere Projekte auf densel-

ben Host-Port binden wollen. Dadurch können Projekte dann nicht gleichzeitig gestartet sein.

Über *Vagrant Share* lassen sich laufende VMs mit Personen im Internet teilen.

Vagrant lässt sich über Plugins erweitern und mit Werkzeugen wie Ansible einsetzen.

Mit Vagrant lassen sich viele der Anforderungen erfüllen. Da vollständig mit virtuellen Maschinen gearbeitet wird, sind Nebeneffekte durch verschiedene Entwicklersysteme größtenteils ausgeschlossen und die Anforderungen A2.2 und A2.7 lassen sich dadurch gut umsetzen. Da die einzelnen VMs verschiedener Projekte unabhängig sind, können keine Probleme wegen Versionskonflikten eingesetzter Bibliotheken und Anwendungen auftreten. Wie gut die Entwickler mit den VMs arbeiten können hängt davon ab, wie gut die VMs für ihre bestimmten Anforderungen definiert wurden.

Großer Nachteil von Vagrant ist die Performance der Virtualisierung, insbesondere im Vergleich zur Containervirtualisierung. Ein weiterer Nachteil ist die Verwaltung vieler ähnlicher Projekte, da Vagrant direkt keine Möglichkeit bietet Konfiguration von Projekten aufzuteilen und über mehrere Projekte zu verteilen, abgesehen von der Definition eigener Boxes. Dieser Nachteil lässt sich durch den kombinierten Einsatz mit Ansible beheben, dadurch wird aber wiederum die Komplexität erhöht. Der Einsatz von Vagrant ist nicht nahtlos, Entwickler müssen direkt mit VMs interagieren.

4.1.3 Docker Compose

Docker Compose ist ein in Docker integriertes Werkzeug, um die Definition und Ausführung von Docker Containern zu vereinfachen (Docker, 2019g).

Dienste einer Anwendung werden in Form von Container-Definitionen in einer YAML-Definitionsdatei festgehalten und können über ein CLI Tool gestartet und gestoppt werden. Es erlaubt, mehrere Container für die einzelnen Dienste eines Projektes zu starten. Diese können über Netzwerke miteinander kommunizieren. Mehrere Projekte lassen sich gleichzeitig starten und Änderungen an den Definitionsdateien lassen sich einfach anwenden. Die einzelnen Projektdateien sind unabhängig voneinander, die Container-Definitionen lassen sich nicht in mehreren Projekten benutzen. Die Images, auf denen Container basieren, lassen sich allerdings projektübergreifend einsetzen.

Über Docker Compose lassen sich die Anforderungen an Entwicklungsinstanzen grundsätzlich erfüllen, vorausgesetzt es bestehen gut definierte Images und Container. Durch die Containervirtualisierung sind Nebeneffekte durch verschiedene Entwicklersysteme größenteils ausgeschlossen. Die Anforderungen A2.2 und A2.7 lassen sich also ebenfalls umsetzen.

In einem Entwicklungs-Kontext ist es wichtig, dass die Dienste unter Linux und Mac mit derselben Benutzer- und Gruppennummer gestartet sind, da es ansonsten unweigerlich zu Berechtigungsproblemen kommt, wenn Container und Entwickler versuchen Dateien zu verändern. Benutzer- und Gruppennummer von Containern sind in Images definiert, lassen sich allerdings in Docker Compose Dateien überschreiben. Das Image muss dies aber grundsätzlich unterstützen.

Im Gegensatz zu Vagrant ist Docker Compose nicht primär für Entwicklungsinstanzen gedacht. Es gibt keine direkten Möglichkeiten beim Starten oder Stoppen von Containern Provisionierung in Form von Scripten durchzuführen. Die Images der Container müssen daher bereits vollständig für die Entwicklung bereit sein oder Start-Scripte zur Provisionierung enthalten.

Die Interaktion zwischen Entwickler und Entwicklungsinstanz findet, wie auch bei Vagrant, über Konsolen-Zugriff auf Containern und Definition von Port-Bindings statt. Die Port-Bindings führen auch hier zum Problem, dass nur ein Projekt gleichzeitig an einen Host-Port gebunden werden kann.

Es ist möglich, sich via „docker[-compose] exec“, in laufende Container einzuwählen und dort Befehle auszuführen. Zusätzlich ist es möglich für Hilfsprogramme oder Konsolen neue Container im selben Netzwerk auszuführen. Diese können in zusätzlichen Docker-Compose Scripten definiert sein oder manuell über den „docker run“ Befehl gestartet werden.

Docker Compose bietet einfache Möglichkeiten Docker Container zu definieren und zu verteilen. Um für Entwicklungsinstanzen eingesetzt werden zu können, müssen einige Einschränkungen beachtet werden. Docker Compose bietet keine nahtlose Integration für Entwickler und setzt daher voraus, dass diese sich grundsätzlich mit Containern auskennen und mit diesen interagieren können.

4.1.4 Kubernetes

Kubernetes ist eine 2014, ursprünglich von Google veröffentliche, Open-Source Software zur Orchestrierung von Containern. Container-Orchestrierung bezeichnet die Konfiguration, Koordination und Ausführung von Containern auf verschiedenen Systemen, basierend auf Container-Definitionen und weiteren Regeln. Kubernetes setzt als Container-Engine standardmäßig auf Docker, erlaubt aber auch die Ausführung anderer Container-Engines.

Kubernetes erlaubt es mehrere physische Systeme (*Nodes*) zu einem gemeinsamen *Cluster* zusammenzufassen, auf dem Container gestartet werden. Kubernetes bietet zur Konfiguration verschiedene Bausteine, die als *Objekte* bezeichnet werden (Hasegawa u. a., 2019).

Einfachste Grundeinheit ist der *Pod*. Ein Pod besteht aus einem oder mehreren Containern. Pods können beispielsweise von *Deployments* erzeugt werden. Deployments definieren, wie Pods auf verschiedene Nodes verteilt werden sollen und stellen sicher, dass eine gewisse Anzahl an Pods verfügbar ist. Neben Deployments gibt es weitere Abstraktionen über Pods, wie beispielsweise *Jobs*, welche einen Pod starten und bis zum Ende durchlaufen lassen.

Neben Pods und weiteren Objekten die Pods verwalten, gibt es viele weitere Objekte zur Konfiguration eines Clusters. Pods können *Services* zugeordnet werden. Services sind im Kubernetes Netzwerk über IP-Adresse und DNS Namen erreichbar und verteilen über Verfahren wie Round-Robin die Last der Anfragen auf ihnen zugeordnete Pods.

Ingress-Objekte erlauben Services und Pods einfacher über HTTP geroutet zu werden, indem einem Ingress Services und Hostnamen zugeordnet werden. Ein Ingress-Controller, basierend auf Webservern und Reverse Proxies wie Nginx oder Haproxy, nimmt eine HTTP-Anfrage entgegen und routet diese basierend auf dem Hostnamen an einen Service. Der Service leitet die Anfrage wiederum zu den jeweiligen Pods weiter. Ingress erlaubt es damit einfach Container über HTTP zu erreichen.

Kubernetes bietet insgesamt viele Möglichkeiten, die insbesondere für den Betrieb auf Cloud-Servern interessant sind. Für die Entwicklung ist insbesondere das Ingress-Feature interessant, da es erlaubt viele Projekte über einen gemeinsamen Proxy Server zu erreichen. Port Binding Probleme, wie bei Vagrant und Docker Compose, lassen sich so komfortabler lösen.

Aufgrund der Optimierung für Server-Systeme bringt Kubernetes allerdings einige Hindernisse für die lokale Entwicklung mit sich. Pods können nur gestartet oder komplett gelöscht werden. Es ist nicht möglich sie kurz zu stoppen und fortzusetzen. Das bedeutet das beim Start eines Projektes erst gewartet werden muss, bis Kubernetes einen komplett neuen Pod einrichtet und startet. Aufgrund der Funktionsweise von Kubernetes Scheduling kann es außerdem passieren, dass Pods plötzlich gelöscht und neu erstellt werden, um Last zu verteilen.

Die Installation und der Betrieb von Kubernetes kommt außerdem, aufgrund der Abhängigkeiten von Kubernetes, mit einem sehr großen Overhead. Zur einfacheren Einrichtung auf Entwicklungs-Systemen gibt es daher Hilfsmittel wie Minikube. Minikube erlaubt es einen Kubernetes Cluster mit einer Node zu installieren. Minikube nutzt dafür allerdings VMs, in denen der Cluster betrieben wird. Kubernetes lässt sich auch manuell direkt auf den Entwickler-Systemen installieren, dies ist aufgrund der vielen Abhängigkeiten aber ein komplexer Vorgang. Die Scheduling-Algorithmen von Kubernetes bieten keine Vorteile für Entwickler, sondern stellen eher eine Behinderung dar. Insgesamt ist der Betrieb eines Kubernetes Clusters für die Entwicklung daher nicht sehr geeignet.

4.2 Anwendungsspezifische Lösungen

Um den spezifischen Anforderungen an Entwicklungsinstanzen gerecht zu werden, gibt es, aufbauend auf allgemeinen Lösungen, anwendungsspezifische Lösungen. Diese versprechen automatisierte Entwicklungsinstanzen für bestimmte Anwendungen und Frameworks bereitzustellen.

Vorteil solcher Lösungen ist, dass sie bereits auf den Entwicklungseinsatz optimiert sind und in der Regel von Teams gewartet werden, die ebenfalls Entwickler für die jeweiligen Anwendungen sind. Die Tools passen damit besser zu den Anforderungen der jeweiligen Entwickler. Anwendungsspezifische Lösungen stellen in der Regel auch Hilfsmittel bereit, um häufig durchgeführte Tätigkeiten nahtloser, ohne direkte Interaktion mit Containern, ausführen zu können.

Nachteil dieser Lösungen ist, dass sie sich nur für bestimmte Anwendungen eignen. Teilweise sind sie dabei von bestimmten Versionen der Anwendung abhängig. Weiteres Problem ist, dass der Support für die Lösungen nur eingeschränkt vorhanden ist, da es sich um weniger verbreitete Tools als die allgemeinen Lösungen handelt, an denen meistens

eine kleine Gruppe Entwickler arbeitet. Entwickler-Teams machen ihre Entwicklungsinstanzen bei Nutzung einer solchen Lösung abhängig von dem Support dieser Lösungen.

In diesem Abschnitt werden einige Lösungen (basierend auf Docker) vorgestellt und Vor- und Nachteile dieser Tools herausgearbeitet.

4.2.1 docker-django (für Django)

docker-django ist eine Zusammenstellung einer Docker Compose Konfiguration für Python Django. Es kommt mit benötigten zusätzlichen Konfigurationsdateien und einer Dokumentation (Bruins Slot, 2019). Django ist ein beliebtes Web-Framework für die Programmiersprache Python.

docker-django baut auf Docker Compose auf und erweitert es nicht programmatisch. Es stellt lediglich eine Vorlage bereit, um einfach mit Django und Docker Compose starten zu können.

Die Dokumentation listet die getesteten Versionen aller benötigten Abhängigkeiten. Die Bedienung findet über Docker Compose statt. Es bietet eine Anleitung, um zusätziche Werkzeuge über „docker exec“ in einem der Container auszuführen.

Die Konfiguration lässt sich über eine Reihe von Dateien im Ordner „config“ durchführen und über Umgebungsvariablen, die sich in der „docker-compose.yml“ einstellen lassen.

Die Konfiguration definiert drei Container:

- **db.** Eine Postgres Datenbank, für die ein Standard Image eingesetzt wird. Diese kann von der Django-Anwendung benutzt werden.
- **webserver.** Ein Nginx Webserver, für den ein Standard Image eingesetzt wird. Der Nginx-Server fungiert als Reverse Proxy vor der eigentlichen Django Anwendung. Er wird über Umgebungsvariablen gesteuert und über Template-Dateien die als Volume eingehängt werden. Der Startbefehl des Containers ist so verändert, dass die Templates über den Befehl „envstubst“ vor dem Starten des eigentlichen Nginx-Servers verarbeitet werden. Die verarbeiteten Dateien stellen dann die Konfiguration des Nginx dar.

- **webapp.** Die Django-App. Der Code wird aus einem angegebenen Verzeichnis gelesen und als Volume eingehängt. Über eine Datei lassen sich zusätzliche Umgebungsvariablen definieren, die an die Django-Anwendung übergeben werden.

docker-django erlaubt es Entwicklern von Django Anwendungen einfach Projekte basierend auf Docker Compose, aufzusetzen. Wichtige Aspekte, wie die Konfiguration der beteiligten Container, sind dabei berücksichtigt. Komplexere Konfigurationsdateien, wie die Nginx Konfiguration, werden über Templating mit Werten von Umgebungsvariablen befüllt, sodass Anwender nur noch einzelne Umgebungsvariablen setzen müssen, ohne sich im Detail mit Docker Compose oder Nginx auszukennen.

Probleme, die auftreten können, werden in der Dokumentation beschrieben. Es gibt eine Anleitung, um über Befehle mit der Anwendung zu interagieren.

docker-django erlaubt es somit Teams, die sich nicht viel mit Docker auskennen, schnell Entwicklungsumgebungen aufzusetzen. Da die Lösung als Vorlage fungiert und erweitert werden kann, können Entwickler-Teams die Lösung weiter an ihre eigenen Bedürfnisse anpassen.

4.2.2 wp-local-Docker (für Wordpress)

wp-local-Docker (Version 1) ist ein Projekt, um eine Wordpress-Entwicklungsinstanz, basierend auf Docker Compose, einzurichten (Marslender u. a., 2019). Wordpress ist eine Software zum Erstellen eigener Blogs und Webseiten, basierend auf PHP. PHP benötigt zur Ausführung einen eigenständigen Webserver, wie Nginx oder Apache.

Ähnlich wie auch docker-django baut es direkt auf Docker Compose auf. Es wird zusätzlich eine Dokumentation zur Verfügung gestellt und weitere Scripte, um die Einrichtung der Wordpress Installation zu vereinfachen. Hilfsbefehle werden als Scripte und CLI Aliase bereitgestellt und können so von Entwicklern ausgeführt werden, ohne dass diese sich direkt in die Container einwählen müssen. Entwickler müssen sich dadurch weniger mit Docker Compose auskennen, und die Arbeit ist für sie nahtloser.

Die Dienste sind auf zwei Docker Compose Projekte aufgeteilt. Die „docker-compose.yml“ enthält grundlegende Dienste für die Ausführung einer Wordpress Installation, wie den PHP Interpreter, einen Nginx Webserver, eine MySQL Datenbank. Es ist auch ein Mailcatcher enthalten, welcher erlaubt, in einer isolierten Entwicklungsumgebung Emails über

Wordpress versenden und empfangen zu können. Log-Dateien der Dienste werden in einem zentralen Ordner auf dem Host gesammelt.

Zusätzlich wird ein Dienst mit dem Namen „WPSnapshots“ gestartet, der von den Autoren des Projektes entworfen wurde, um Wordpress Instanzen mit anderen Team-Mitgliedern zu teilen (vergleichbar mit *Vagrant Share*).

Die „admin-compose.yml“ stellt weitere hilfreiche Dienste bereit, wie PhpMyAdmin, eine Web-Oberfläche zur Administration des MySQL Servers.

Einige der benutzten Images basieren auf Standard-Images aus der Docker Registry, andere sind von den Autoren des Projektes erstellt, speziell für den Einsatz mit wp-local-Docker.

Wie auch bei docker-django findet die Konfiguration der einzelnen Dienste über Umgebungsvariablen und Konfigurationsdateien statt.

Über wp-local-Docker können Entwickler-Teams schnell, basierend auf Docker, Entwicklungsinstanzen aufsetzen. Da das Projekt als Vorlage fungiert, können es Entwickler-Teams an ihre Anforderungen anpassen.

Version 2

Aktuell wird eine neue Version von wp-local-Docker entwickelt, die eine eigenständige Anwendung ist. Sie basiert auf der JavaScript Laufzeitumgebung Node.js. Diese Lösung kommuniziert über CLI-Schnittsellen und APIs mit Docker und Docker Compose.

Das Tool ermöglicht es, über eine einzelne zentrale CLI-Anwendung („10updocker“) Wordpress Installationen zu verwalten.

Einzelne Wordpress Entwicklungsinstanzen werden in sogenannten *Environments* (Umgebungen) verwaltet. Beim Erstellen einer Umgebung wird der Benutzer interaktiv abgefragt, welche Anforderungen er benötigt. Die Einrichtung ist dabei simpel und beschränkt sich auf wenige Befehle.

Die Einrichtung einer Umgebung kann entweder auf einer bestehenden Wordpress-Installation oder einer Neuen basieren. Das Tool installiert dabei nicht Wordpress selbst. Der Anwender muss den Wordpress-Code selbstständig herunterladen und die eigentliche Installation über die Installationsroutinen von Wordpress durchführen.

Es können mehrere Umgebungen gleichzeitig gestartet werden, da wp-local-Docker den Traffic über einen zentralen Gateway Reverse Proxy leitet, welcher ebenfalls als Container gestartet wird und an Port 80 und 443 an den Host gebunden wird. Einzelne Umgebungen werden über einen Hostnamen unterschieden. Dieser Hostname wird vom Tool in die hosts-Datei eingetragen, um auf dem Entwickler-PC direkt nach der Installation routebar zu sein. Dies macht die Einrichtung und Installation mehrerer Projekte sehr einfach für Entwickler, sie müssen sich nicht mit Docker oder Netzwerkkonfiguration auskennen.

Über das Tool lassen sich Umgebungen starten und stoppen und Konsolen auf die Container öffnen. Um mit der Wordpress-Installation über CLI zu interagieren ist also das Verbinden via Konsole notwendig. Es lassen sich über den Befehl auch Logdateien von Containern auslesen und das Tool integriert WPSnapshots.

Diese Neufassung von wp-local-Docker vereinfacht die Ausführung von Wordpress Installationen, durch einfache interaktive Abfragen und vollautomatische Netzwerkkonfiguration. Mit diesem Werkzeug können Wordpress Entwickler schnell Entwicklungsinstanzen aufsetzen. In der Dokumentation sind allerdings keine Hilfsscripte oder Aliase, um häufige CLI Befehle auszuführen, so wie beim Vorgänger. Zusätzlich kommt hinzu, dass Anwender von wp-local-Docker abhängig von der neuen Node.js Anwendung sind. Da diese allerdings intern mit Docker Compose Dateien arbeitet, ist eine Migration auf Docker Compose möglich.

4.2.3 Magedev (für Magento)

Magedev ist ein Werkzeug zur Verwaltung von Entwicklungsinstanzen für die E-Commerce Software Magento (Leers und Nuß, 2018). Basis für Magedev ist Docker. Magedev besteht aus einer, in PHP geschriebenen, CLI-Anwendung. Die Anwendung kommuniziert mit der Docker API, um Images zu bauen und Container zu starten.

Projekte werden in einer „magedev.json“-Datei definiert und können darüber erweitert werden. Das Tool definiert für verschiedene Versionen von Magento jeweils Standard-Container und Standard-Images. Die Standard Container können programmatisch über PHP Klassen erweitert und der „magedev.json“ hinzugefügt werden. Das Tool kann Images für Docker Container bauen. Dafür werden Bau-Anweisungen in einer Syntax ähnlich zur Dockerfile, in eigenen PHP-Klassen hinterlegt. Projekt-Einstellungen und Port-Bindings werden über die „magedev.json“ verwaltet.

Um neue Projekte anzulegen, muss eine minimale „magedev.json“ angelegt werden, in der die eingesetzte Magento Version und der gewünschte Hostname für den Shop eingetragen werden müssen. Magento benötigt in seiner Konfigurationsdatenbank die Einstellung des Hostnamen. Diese Einstellung wird von Magedev beim Start eines Magento-Shop in die Datenbank geschrieben, sodass der Entwickler dafür nichts weiter einstellen muss.

Magedev erlaubt bestehende Projekte, mit bestehenden Mediendateien, Code-Ständen und Datenbanken zu importieren. Pfade für Datenbank-Abbild und Mediendateien lassen sich dabei konfigurieren. Magedev bietet auch Möglichkeiten, einen komplett neuen Shop zu installieren. Bei Import und Neuinstallation werden gängige und notwendige Befehle ausgeführt, um den Shop in Betrieb nehmen zu können.

Magedev besitzt eine Dokumentation, die Entwicklern die Benutzung des Tools erklärt. Magedev besitzt weitere praktische Funktionen für die Entwicklung, wie die Einrichtung des Debuggers und die Möglichkeit Einstellungen zu definieren, die beim Starten in die Datenbank des Magento Systems eingespielt werden.

Magedevs CLI-Interface erlaubt es Entwicklern, sich auf die Konsole der Container einzuwählen. Desweitern bietet das Tool allerdings auch Befehle, die gängige CLI-Operationen abkürzen. Dadurch müssen sich Entwickler nicht erst in einen Container einwählen und können so nahtloser mit der Entwicklungsinstanz interagieren.

Magedev bringt keinen eigenen Reverse Proxy-Server als Gateway mit, es kann nur ein Projekt gleichzeitig gestartet sein.

Magedev erlaubt es Magento Entwicklern, einfach Entwicklungsinstanzen mit Docker aufzusetzen. Es setzt sich insbesondere durch Funktionen, wie die nahtlose Integration zusätzlicher Befehle und der Import-Funktionen, von anderen Lösungen ab. Nachteil von Magedev ist die große Abhängigkeit an das Tool, da Container und Images direkt innerhalb von PHP-Code in der Anwendung definiert werden müssen.

5 Konzeption der System-Architektur

In diesem Kapitel wird ein System zur Verwaltung von Entwicklungsinstanzen für Webanwendungen konzipiert, basierend auf den Ergebnissen der vorherigen Kapitel.

5.1 Grundlegende Konzeption basierend auf den Analyseergebnissen

Entwickler besitzen Anforderungen an ihre Entwicklungsinstanzen (A1.1 - A1.11), um ihre Arbeit zufriedenstellend durchführen zu können.

Die manuelle Einrichtung von Entwicklungsinstanzen erlaubt Entwicklern zwar, ihre Systeme so einzurichten, dass sie damit selbst am besten arbeiten können, führt allerdings auf lange Sicht auch zu mangelnder Dokumentation und Bugs aufgrund von Abweichungen der einzelnen Systeme. Automatisierungslösungen helfen, um diese Probleme zu lösen und die Anforderungen an Definition und Installationsprozess (A2.1 - A2.7) zu erfüllen. Sie erlauben eine Kontrolle über die Konfiguration der Entwicklungsinstanzen und damit eine Verringerung der Aufwände und unerwarteter Nebeneffekte.

Erfüllt ein Automatisierungswerkzeug nicht alle Anforderungen des Entwickler-Teams kann es zur Automation Fear kommen. Entwickler korrigieren einzelne Aspekte der automatisierten Lösung von Hand, und es ist schwer für das Team diese in die Automatisierungslösung einzuarbeiten. Es ist dadurch längerfristig nicht mehr möglich, nur noch mit den Automatisierungen zu arbeiten, wodurch jegliche Vorteile wieder hinfällig werden.

Allgemeine Lösungen (wie Docker Compose, Ansible und Vagrant) sind, um alle Anforderungen von Entwickler-Teams zu erfüllen, aufwendig einzurichten, und viele von ihnen sind nicht optimiert für die Ausführung von Entwicklungsinstanzen. Keines der auf Virtualisierung basierenden Lösungen erlauben es dem Entwickler nahtlos zu arbeiten, sie setzen direkte Interaktion des Entwicklers mit VMs oder Containern voraus.

Aus diesen Gründen werden anwendungsspezifische Lösungen entwickelt. Diese sollen den Entwicklungsprozess optimieren und die Interaktion mit Containern für den Entwickler nahtloser gestalten. Sie erlauben es schnell Entwicklungsinstanzen für die jeweiligen Anwendungen zu starten. Sie werden allerdings nur für bestimmte Versionen bestimmter Anwendungen entwickelt und sind abhängig vom Support der Projektbetreuer.

Es fehlt ein System, das die Vorteile der anwendungsspezifischen Lösungen mit den Vorteilen der allgemeinen Lösungen verbindet. Ein solches System wäre per Design auf die Anforderungen von Webentwicklern an ihre Systeme optimiert und würde für Systemadministratoren einfache Möglichkeiten schaffen Entwicklungssysteme zu definieren.

Ziel dieses Kapitel ist es, die Architektur eines solchen Systems zu konzipieren. Die Anwendung soll die definierten Anforderungen an Entwicklungsinstanzen erfüllen. Es soll dabei für Entwickler eine möglichst nahtlose Integration bieten, um Probleme der Automation Fear zu vermeiden.

Entwickler sollen sich möglichst wenig mit Virtualisierung auskennen müssen, sondern lediglich mit der Anwendung, an der sie arbeiten. Systemadministratoren sollen einfache Möglichkeiten haben, Definitionen für Entwicklungsinstanzen zu schreiben und diese über mehrere Projekte zu Verteilen.

Das konzipierte System trägt den Namen *Riptide*.

5.1.1 Umsetzung der Anforderungen

Das System (Riptide) soll alle Anforderungen, die in *3.2 Anforderungen an Entwicklungsinstanzen für Webanwendungen* definiert wurden, erfüllen. Nachfolgend wird für jede Anforderung aufgeschlüsselt, wie das Riptide System diese erfüllt.

A1.1 *(Einrichtungsaufwand)*

Für die Entwickler soll der Einrichtungsaufwand möglichst gering gehalten werden. Es soll lediglich die Installation der Container Engine und von Riptide selbst (plus ggf. die Laufzeitumgebung der Programmiersprache(n)) nötig sein. Projektdateien definieren die Container zur Ausführung (vgl. *5.5.2 Projekte*). Diese Dateien sollen mit dem Quellcode der Anwendungsprojekte verteilt werden. Riptide stellt einen Befehl zur Ersteinrichtung zur Verfügung (vgl. *5.7.1 Ersteinrichtung eines Projektes*). Über diesen Befehl werden die Entwickler durch die Ersteinrichtung geführt, danach können sie ihre Arbeit beginnen.

A1.2 *(Funktionsumfang der Anwendung)*
Das System arbeitet mit einer Definitionssprache, die es erlaubt Container für eine Anwendung zu definieren. Diese Container können denden vollen Funktionsumfang einer Anwendung abdecken (vgl. *5.5 Konfigurations-Entitäten* und *5.6 Definitionssprache für Entwicklungsumgebungen*). Darüber hinaus ist diese Anforderung von der Nutzung entsprechender Images abhängig (vgl. *Abgrenzungen*).

A1.3 *(Erreichbarkeit der Anwendung via HTTP)*
Für Container können HTTP-Ports definiert werden (vgl. *5.5.4 Dienste*, Schlüssel „port“). Dieser Port ist über den Proxy-Server erreichbar (vgl. *5.3.5 Proxy-Server*).

A1.4 *(Verfügbarkeit externer Dienste)*
Weitere Anwendungs-Dienste, wie Datenbank-Server, lassen sich als Container definieren und werden zusätzlich gestartet. TCP- und UDP-Ports für diese Dienste lassen sich definieren und sind für Entwickler zugänglich (*5.7.3 Zugriff auf zusätzliche Dienst-Ports*).

A1.5 *(Verwaltung von Daten und Dateien)*
Quellcode der Anwendungen werden Containern über Volumes zur Verfügung gestellt. Der Entwickler kann diese Dateien über das Host-System direkt bearbeiten. Datenbanken können verwaltet werden (vgl. *5.3.4 Datenbank-Treiber*) und das System bietet Schnittstellen zum Import zusätzlicher Dateien (vgl.*5.7.1 Ersteinrichtung eines Projektes*).

A1.6 *(Debugging)*
Schnittstellen für Debugger lassen sich definieren. Container lassen sich im Vordergrund starten (vgl. *5.3.2 CLI-Anwendung*, Befehl „start-fg“). Konsolen und Entwicklungsumgebungen (IDEs) können damit Umgebungsvariablen setzen, um Debugger-Verbindungen herzustellen.

A1.7 *(Hilfswerkzeuge)*
Zusätzliche Hilfsprogramme auf Container-Basis werden bereitgestellt (vgl. *5.5.5 Kommandos* und *5.7.2 Kommandos von Anwendungen*).

A1.8 *(Logdateien)*
Logdateien werden dem Entwickler bereitgestellt (vgl. *5.7.5 Logging*).

A1.9 *(Wechsel zwischen verschiedenen Versionsständen)*
Das System bietet Werkzeuge zur Verwaltung mehrerer Datenbank-Versionen (vgl.

5.3.4 Datenbank-Treiber). Versionskontrolle des Quellcodes der Anwendung ist nicht Teil des Systems (vgl. *Abgrenzungen*).

A1.10 *(Arbeit an mehreren Projekten)*
Das System erlaubt, dass mehrere Projekte gleichzeitig gestartet sind und erlaubt Zugriff auf die Systeme über den Proxy-Server (vgl. *5.3.5 Proxy-Server*).

A1.11 *(Kompatibilität mit Betriebssystemen)*
Für die Implementierung wird berücksichtigt, dass das System auf Linux, macOS und Windows einsetzbar ist, insbesondere für die Wahl des Container-Backends (vgl. *5.3.3 Container-Backend*).

A2.1 *(Dokumentation)*
Riptide erlaubt es, die Entwicklungsinstanzen in Definitionsdateien festzuhalten. Die Definitionssprache unterstützt Kommentare. Riptide besitzt zudem selbst eine Dokumentation (vgl. *5.9 Dokumentation*).

A2.2 *(Anwendbarkeit auf verschiedene Systemen)*
Die Entwicklungssysteme werden in vom Betriebssystem unabhängigen Container-Umgebungen verwaltet, welche auf eindeutigen Definitionsdateien basieren.

A2.3 *(Generalisierung verschiedener Projekte)*
Die Definitionssprache ist aus einer Hierarchie von Konfigurations-Entitäten (vgl. *5.5 Konfigurations-Entitäten*) aufgebaut. Anwendungen können beispielsweise verschiedene Dienste beinhalten, die die eigentlichen Container repräsentieren. Diese Dienste können in verschiedenen Anwendungen und Projekten eingesetzt werden. Einstellungen von Entitäten lassen sich auf Projektebene für einzelne Projekte anpassen. Dies wird ermöglicht, durch eine Verteilung einzelner Entitäten auf Repositories (vgl. *5.6.3 Repositories*) und durch die Eigenschaften der Definitionssprache (vgl. *5.6 Definitionssprache für Entwicklungsumgebungen*).

A2.4 *(Automatisierung)*
Alle Aktionen von Riptide lassen sich über CLI-Befehle steuern und automatisieren.

A2.5 *(Anwendung von Änderungen)*
Eine Änderung von Definitionsdateien kann auf alle Entwickler-Systeme verteilt werden, da Projektdateien mit dem Projekt-Quellcode verteilt werden und weitere Definitionsdateien in Versionskontrollsystemen versioniert werden (vgl. *5.6.3 Repositories*).

A2.6 *(Fehlerbehandlung)*
Die Riptide-CLI bricht bei ungültigen Definitionsdateien ab und gibt Fehlerbeschreibungen über Problemfälle, welche Systemadministratoren benutzen können, um Fehler zu finden.

A2.7 *(Reproduktion des Produktivsystems)*
Riptide arbeitet mit Containern, welche auf beliebigen Images basieren. Diese Images können auch auf Produktivsystemen eingesetzt werden, wodurch eine gewisse Reproduktion des Produktivsystems möglich ist. Riptide selbst ist nicht für den Einsatz auf Produktivsystemen konzipiert.

5.1.2 Abgrenzungen

Die von Riptide gestarteten Container basieren auf Docker Images. Riptide kann nicht zur Erstellung von Images für Container-Engines genutzt werden. Für die Erstellung und Verteilung der Images können bestehende Werkzeuge, wie Dockerfiles und Docker Registries verwendet werden.

Riptide bringt keine Versionskontrolle für den Quellcode von Anwendungsprojekten mit. In der Konzeption wird davon ausgegangen, dass Teams ihren Anwendungsquellcode in Versionskontrollsystemen (VCS) wie Git versionieren.

5.2 Architektur des Systems

Riptide ist unterteilt in mehrere Komponenten, welche als eigenständige Software-Pakete veröffentlicht werden und über definierte Schnittstellen miteinander kommunizieren.

Architektur von Softwaresystemen kann nach dem 4+1-Sichten-Modell aus verschiedenen Sichten betrachtet werden (Kruchten, 1995). Dieses Modell teilt die Sicht auf Architektur in Logische Sicht (Logical View), Ablaufsicht (Process View), Verteilungssicht (Physical View), Entwicklungssicht (Development View) und Szenarien auf.

Dieser Abschnitt gibt einen Einblick über die einzelnen Sichten auf die Gesamt-Architektur. Details zu den Sichten auf einzelne Komponenten werden im Kapitel *5.3 Komponenten* erläutert.

Logische Sicht

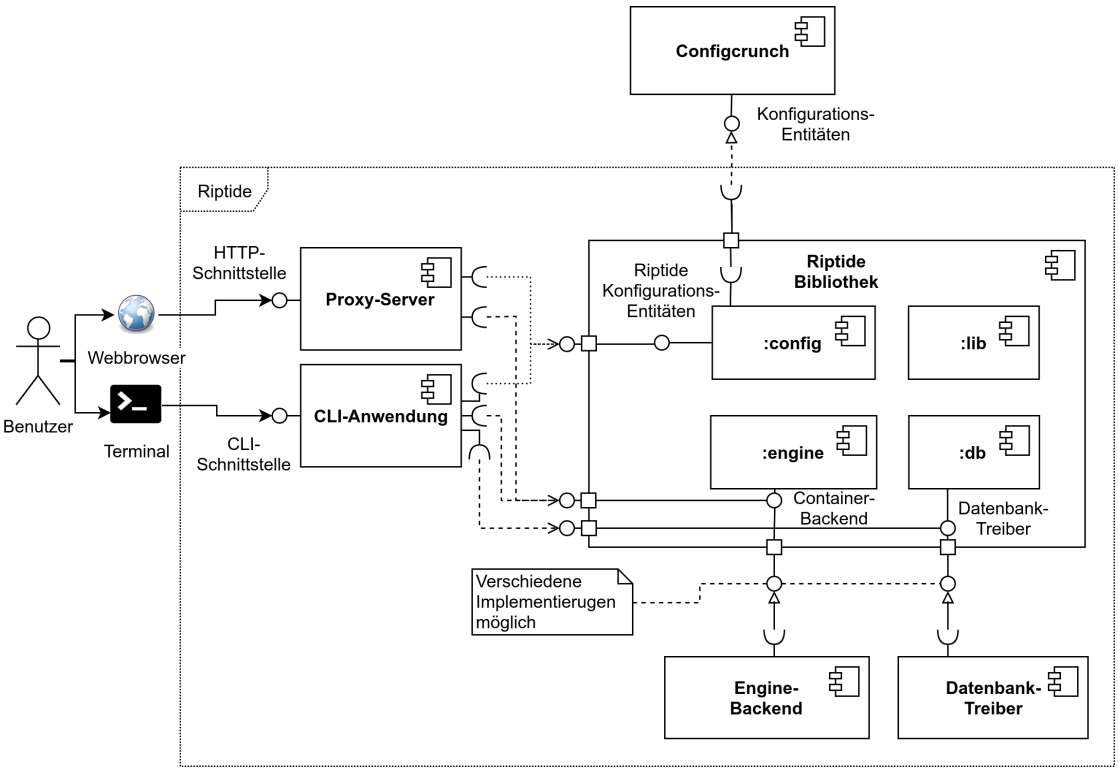


Abbildung 5.1: Überblick über die Architektur von Riptide (UML Komponentendiagramm)

Abbildung 5.1 und die nachfolgende Auflistung der Komponenten zeigt die Entwurfssicht (Development View).

Das System besteht aus einer zentralen Bibliothek (*Riptide Bibliothek*) und zwei Komponenten mit denen der Benutzer interagiert (*CLI-Anwendung* und *Proxy-Server*). Das System besteht zudem aus mindestens einer Implementation eines *Container-Backends*, welche als Aufgabe haben die Software-Container zu starten und zu verwalten. Zur Verwaltung von Datenbankmanagementsystemen (DBMS) stellt das System *Datenbank-Treiber*-Komponenten bereit. Die Schnittstellen für Container-Backends und Datenbank-Treiber sind Teil der Riptide Bibliothek.

Die Konfiguration des Systems wird über Konfigurations-Entitäten geregelt (siehe Kapitel 5.5). Diese Entitäten sind Bestandteil einer Definitionssprache (siehe *5.6 Definitionssprache für Entwicklungsumgebungen*). Das Einlesen und Verhalten der Definitionssprache

wird durch die Bibliothek *Configcrunch* bereitgestellt. Diese Bibliothek ist von Riptide grundsätzlich unabhängig und kann auch von anderen Systemen verwendet werden.

Ablaufsicht

Beispiele für die Ablaufsicht (Process View) werden in den Abschnitten *5.3.2 CLI-Anwendung* (Abbildung 5.4) und *5.3.3 Container-Backend* (Abbildung 5.5) gezeigt.

Diese Sichten auf die dynamische Laufzeit des Systems zeigen beispielhaft den Startprozess der Container für ein Projekt. In den Abbildungen wird auch auf die Schnittstellen zwischen diesen Komponenten während dieses Prozesses eingegangen. Der Proxy-Server ist ebenfalls in der Lage Projekte zu starten. Der Ablauf findet entsprechend der Abbildung der CLI-Komponente statt.

Damit der Startprozess ausgeführt werden kann, muss die Konfiguration des Systems und des aktuellen Projektes eingelesen werden. Dieser Vorgang wird im Abschnitt *5.3.1 Riptide Bibliothek* (Abbildung 5.3) erläutert.

Weitere Details zu der Ablaufsicht einzelner Komponenten werden im Abschnitt *5.3 Komponenten* erläutert.

Verteilungssicht

Abbildung 5.2 zeigt die Verteilungssicht (Physical View). Eine verteilte Riptide Installation besteht aus einer Reihe Entwickler PCs (in diesem Beispiel einer) auf denen alle zentralen Riptide Komponenten (Lib, CLI, Proxy) und mindestens eine Container-Engine installiert sind. Auf diesen Entwickler-Rechnern ist eine Container Engine (im Beispiel Docker) installiert. Pro Projekt wird ein isoliertes Container Netzwerk gestartet, in denen einzelne Softwarecontainer gestartet sind. Diese Softwarecontainer werden nach Definitionen in der Projektdatei gestartet.

Die Projektdateien werden jeweils mit dem Quellcode der einzelnen Projekte aus einem Versionskontrollsystem geladen. Weitere Entitäten werden aus zusätzlichen VCS-Repositories geladen (vgl. *5.6.3 Repositories*). Andere Verteilungsmechanismen außer Versionskontrollsysteme sind für Projekte und Projektdateien ebenfalls möglich, nicht

aber für zusätzliche Entitäten. Die Container auf den Entwickler-PCs basieren auf Definitionen in Form von Images. Die Images werden von Container Registry-Servern heruntergeladen.

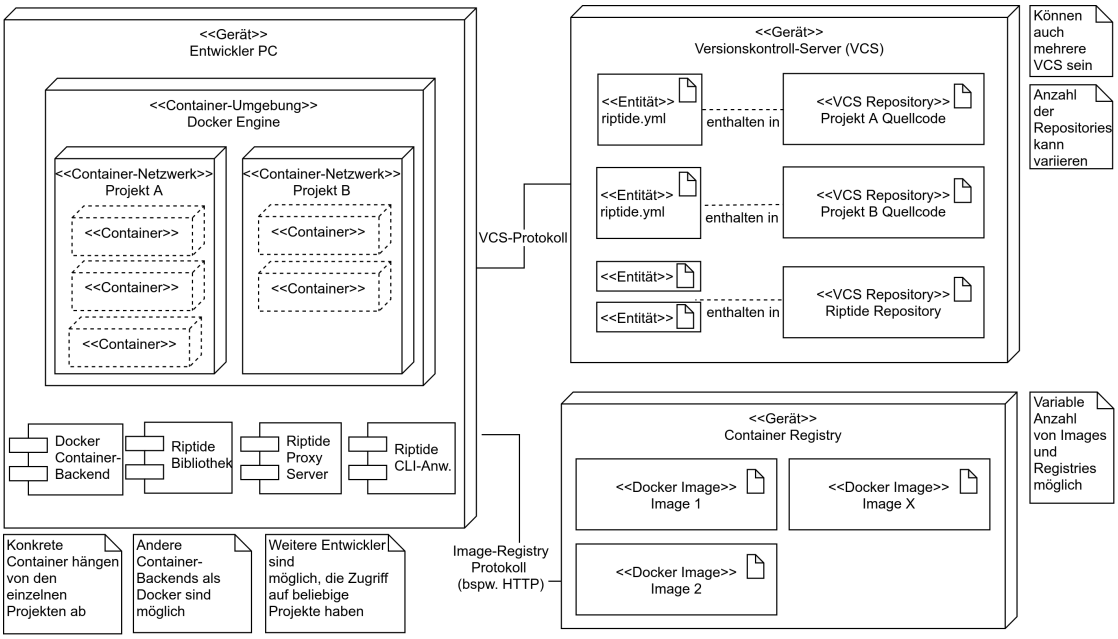


Abbildung 5.2: Verteilungssicht auf die Architektur von Riptide (UML Verteilungsdiagramm)

Entwicklungssicht und Szenarien

Auf die Entwicklungssicht der einzelnen Komponenten wird im Abschnitt *5.3 Komponenten* eingegangen. Die Szenarien (Anwendungsfälle) leiten sich aus den User Stories ab (vgl. *3.2 Anforderungen an Entwicklungsinstanzen für Webanwendungen*).

5.3 Komponenten

Dieser Abschnitt listet Details zu den einzelnen Komponenten des Riptide System und erläutert, wie diese miteinander interagieren.

Die Bibliothek zur Auswertung der Definitionssprache (*Configcrunch*) wird ebenfalls als Teil dieser Arbeit konzipiert. In diesem Kapitel wird Configcrunch allerdings nicht erläutert, da es sich hierbei um eine vom Riptide-System unabhängige Bibliothek handelt.

Details zur Auswertung der Definitionssprache und zu Configcrunch sind im Kapitel *5.6 Definitionssprache für Entwicklungsumgebungen* beschrieben.

5.3.1 Riptide Bibliothek

Zentraler Bestandteil von Riptide ist die Bibliotheks-Komponente. Die Riptide Bibliothek stellt grundlegende Funktionen von Riptide bereit. Alle anderen Komponenten greifen auf diese zentrale Bibliothek zurück.

Ablaufsicht

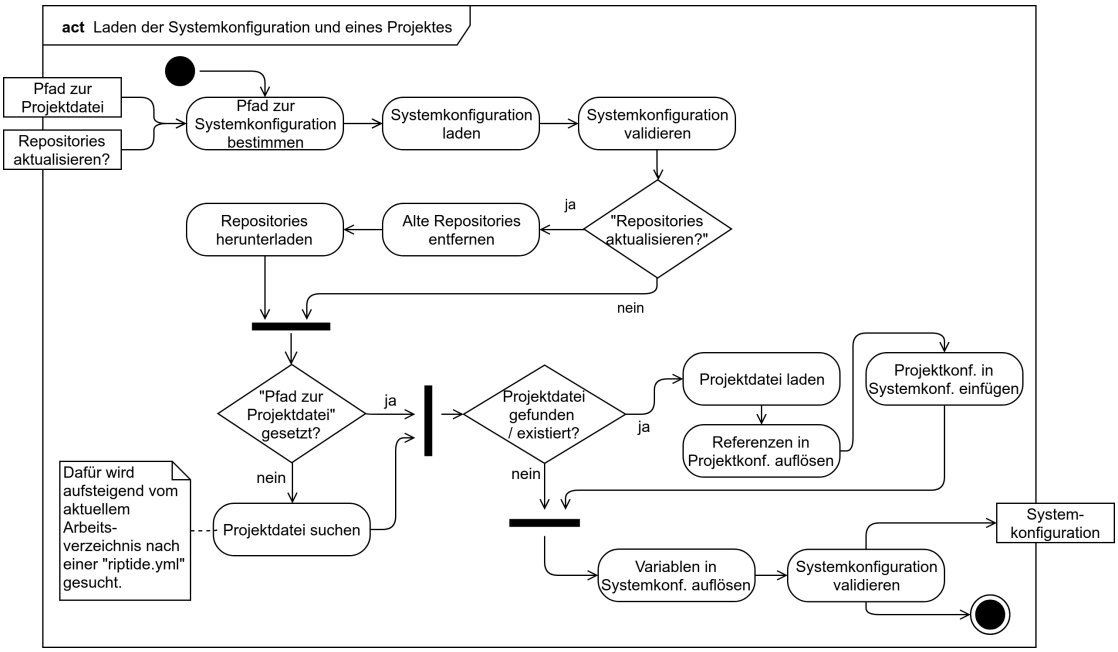


Abbildung 5.3: Ablaufsicht auf den Lade-Prozess der Konfiguration (UML Aktivitätsdiagramm)

Die Riptide Bibliothek wird von den anderen Komponenten während verschiedener Prozesse benutzt. Zentrale Aufgabe der Bibliothek ist das Laden der Systemkonfiguration und der Projekte. Abbildung 5.3 zeigt diesen Vorgang.

Beim Laden der Konfiguration wird zunächst die Systemkonfigurations-Datei von einem vordefinierten Pfad eingelesen und validiert. Beim Starten des Vorgangs kann angegeben werden, ob die Repositories (siehe Abschnitt 5.6.3) aktualisiert werden sollen. Ist dies

gewünscht, werden nun zunächst die Repositories aus der Systemkonfiguration heruntergeladen und alte Repositories, die nicht mehr Teil der Konfiguration sind, entfernt.

Beim Starten des Vorgangs kann der Pfad zu einer Projektdatei angegeben werden. Wurde keine Projektdatei angegeben, sucht das System automatisch nach einer Projektdatei, indem aufsteigend vom aktuellen Arbeitsverzeichnis nach einer „riptide.yml“-Datei gesucht wird. Es wird überprüft, ob die angegebene oder ermittelte Projektdatei existiert. Falls sie existiert, wird sie geladen und Referenzen innerhalb der Projektdatei werden aufgelöst (siehe *5.6.2 Algorithmus zum Einlesen von Definitionsdateien*). Die Projektdatei wird danach validiert und in die Systemkonfiguration als Knoten eingefügt. Zum Schluß, unabhängig davon, ob eine Projektdatei geladen wurde, werden die Variablen in der Konfiguration aufgelöst, die Konfiguration wird validiert und als Ergebnis des Vorgangs zurückgegeben.

Bei Fehlern während dem Einlesen von Dateien oder der Validierung wird ein entsprechender Fehler geworfen.

Entwicklungssicht

Die Riptide Bibliothek besteht aus vier Unter-Komponenten, deren Implementationsdetails im Kapitel *6.1 Implementation der Riptide Bibliothek* erläutert werden.

Logische Sicht und Schnittstellen

Die Schnittstellen der Riptide Bibliothek sind die Module zum Laden der Konfiguration (vgl. *Ablaufsicht*), die Konfigurations-Entitäten (siehe Kapitel 5.5), sowie weitere Module zur Verwaltung von Dateien und Konstanten. Weiterhin befinden sich in der Bibliothek die Schnittstellen-Definitionen für Datenbank-Treiber und Container-Backends.

5.3.2 CLI-Anwendung

Die CLI-Anwendung ist die zentrale Schnittstelle zwischen Riptide und dem Benutzer. Sie stellt das ausführbare CLI-Programm *riptide* zur Verfügung.

Benutzer verwenden die CLI-Anwendung, um Projekte und Dienste (siehe Kapitel 5.5) zu starten und zu verwalten. Die CLI-Anwendung stellt einen Assistenten zur Ersteinrichtung von Projekten zur Verfügung (siehe Kapitel 5.7.1) und wird verwendet, um Container für Kommando-Entitäten zu starten (siehe Kapitel 5.7.2)

Ablaufsicht

Die CLI-Anwendung führt auf Befehl des Benutzers verschiedene Aufgaben aus. Vor Ausführung jedes Befehls wird die Riptide Konfiguration initialisiert und ggf. ein Projekt geladen. Nach Ausführung der Aufgabe beendet sich die Anwendung. Nachfolgend wird als Beispiel für die Ablaufsicht der Start-Vorgang erläutert. Der Abschnitt *Logische Sicht und Schnittstellen* erläutert den Ablauf weiterer Befehle.

Der „start“-Befehl weist Riptide an, die in der Anwendung des Projektes definierten Dienste als Container zu starten (vgl. *5.5.3 Anwendungen* und *5.5.4 Dienste*).

Abbildung 5.4 zeigt den Ablauf des „start“ Kommandos. Dem Benutzer wird der aktuelle Fortschritt dieses Befehls über Fortschrittsbalken angezeigt. Gibt es beim Start eines Dienstes ein Problem, so wird der Benutzer darüber informiert.

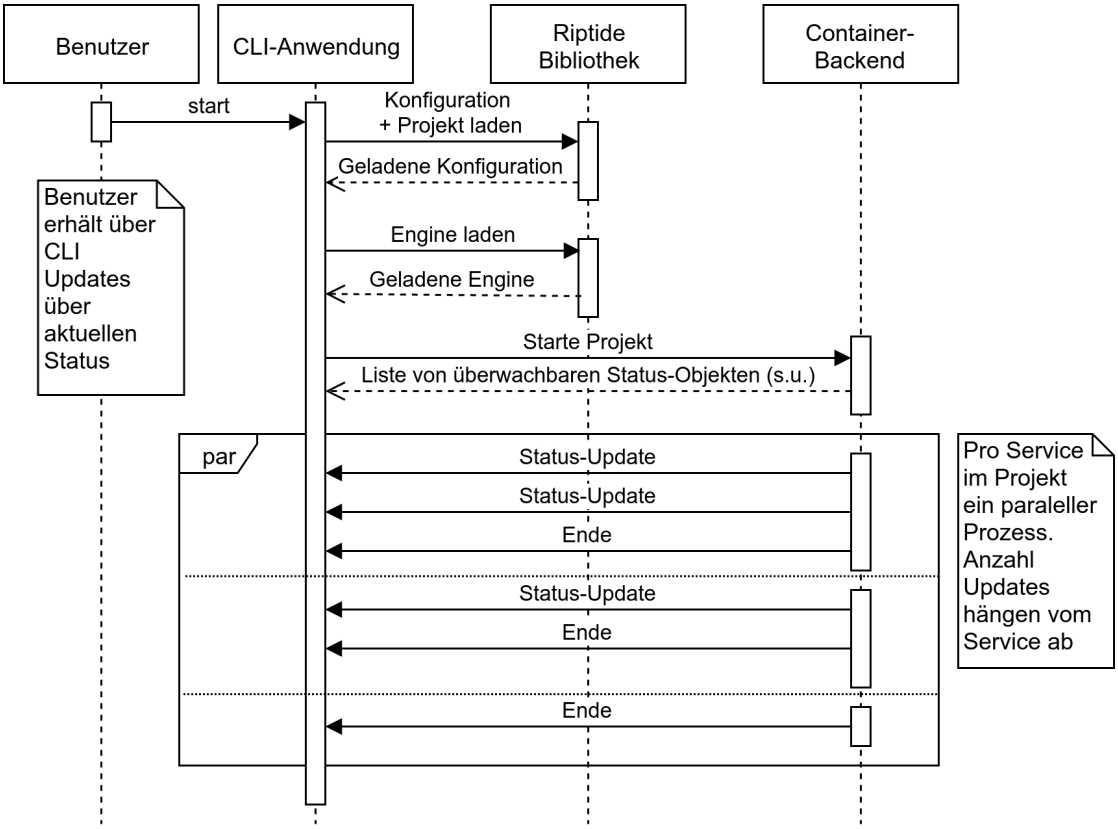


Abbildung 5.4: Ablaufsicht auf den Start-Prozess über die CLI-Schnittstelle (UML Sequenzdiagramm)

Die Abbildung zeigt die Interaktion der CLI-Anwendung mit der Riptide Bibliothek und der Container-Backend-Schnittstelle. Über die Riptide Bibliothek lädt die CLI-Anwendung die aktuelle Systemkonfiguration, das aktuelle Projekt und die Instanz der Engine-Schnittstelle entsprechend der Einstellung in der Systemkonfiguration (vgl. *5.5.1 Systemkonfiguration*). Die CLI-Anwendung weist daraufhin die Engine an, das Projekt zu starten und der CLI-Anwendung Status-Änderungen mitzuteilen.

Entwicklungssicht

Die CLI-Anwendung besteht aus einer Reihe Module, welche die einzelnen CLI-Befehle zur Verfügung stellen.

Als Bibliothek wird Click (`https://click.palletsprojects.com/en/7.x/`) eingesetzt. Click ist eine Python-Bibliothek, welche es erlaubt über Annotations („Anmerkungen“) an Python-Funktionen CLI-Befehle zu definieren. Dabei lassen sich Optionen, Argumente und Beschreibungstexte definieren.

Logische Sicht und Schnittstellen

Schnittstellen der CLI-Anwendung sind die Unter-Kommandos, welche verschiedene Aufgaben erfüllen. Je nach Kontext stehen andere Kommandos zur Verfügung. Der Kontext hängt davon ab, ob eine Projektdatei geladen ist und welche Funktionen für das Projekt konfiguriert sind. Tabelle 5.1 zeigt eine Liste aller Befehle mit einer kurzen Beschreibung.

Die CLI-Anwendung benutzt Schnittstellen der Riptide Bibliothek, um Projekte zu laden und mit Container-Backend und Datenbank-Treibern zu kommunizieren.

Update-Modus. Neben den Befehlen in Tabelle 5.1 kann die CLI-Anwendung mit der Option „--update“ (kurz „-u“) gestartet werden. Wenn diese Option gesetzt ist, aktualisiert Riptide vor der Ausführung des eigentlichen Befehls die Repositories (vgl. *5.6.3 Repositories*) und die im aktuellen Projekt eingesetzten Container Images.

Tabelle 5.1: Liste aller Befehle der Riptide-CLI

	Befehl	**Beschreibung**
Dienst-Befehle *Befehle für die Verwaltung von Diensten in Projekten. Außer „status“ nur verfügbar, wenn ein Projekt geladen ist.*		
	start [--services;-s]	Startet die Container aller oder der angegeben (--services) Dienste.
	stop [--services;-s]	Stoppt die Container aller oder der angegeben (--services) Dienste.
	restart [--services;-s]	Stoppt die Container aller oder der angegeben (--services) Dienste und startet sie danach.
	start-fg [--services;-s] DIENST	Startet alle unter „--services“ angegebenen Dienste und danach DIENST im Vordergrund, verbunden mit der Konsole. Dabei gelten einige Einschränkungen, welche mit der Option „--help“ erläutert werden.
	status	Zeigt den Status aller laufenden Dienste (ob diese gestartet sind oder nicht). Zeigt zusätzlich ggf. die Adresse, unter der der Dienst erreichbar ist und zusätzliche Ports (vgl. *5.7.3 Zugriff auf zusätzliche Dienst-Ports*).
Projekt-Befehle *Befehle für die allgemeine Verwaltung von Projekten. Nur verfügbar, wenn ein Projekt geladen ist.*		
	setup [--force] [--skip]	Startet die interaktive Projekt-Einrichtung, falls diese noch nicht ausgeführt wurde (siehe *5.7.1 Ersteinrichtung eines Projektes*). „--skip“ überpringt die Einrichtung und „--force“ erzwingt sie, falls sie bereits durchgeführt wurde.
	notes	Zeigt Hinweistexte, die am Projekt für die Benutzung und Ersteinrichtung hinterlegt sind (siehe *5.7.1 Ersteinrichtung eines Projektes*).

CLI-Befehle *Befehle, um sich in Dienst-Container einzuwählen oder CLI-Kommandos auszuführen. Nur verfügbar, wenn ein Projekt geladen ist.*		
	cmd NAME ARGS...	Führt das Kommando (vgl. *5.5.5 Kommandos* mit dem Namen NAME aus. Falls NAME nicht gegeben ist, werden Kommandos des Projektes aufgelistet. Für weitere Information siehe *5.7.2 Kommandos von Anwendungen*. ARGS werden an das Kommando als Argumente übergeben.
	exec [--root] DIENST	Öffnet eine interaktive Konsole für den Container für DIENST. Benutzer ist ohne Angabe von „--root“ der aktuelle Benutzer, sonst root.
Import-Befehle *Befehle, um Dateien oder Datenbanken zu Importieren. Nur verfügbar, wenn ein Projekt geladen ist.*		
	import-files SCHLÜSSEL PFAD	Importiert Dateien unter PFAD nach dem Pfad der in der Projektdatei unter SCHLÜSSEL konfiguriert ist. Details zur Import-Funktion sind unter *5.7.1 Ersteinrichtung eines Projektes* erläutert, dieser Befehl importiert einen einzelnen Pfad manuell. Nur verfügbar, wenn mindestens ein Dateipfad zum Import im Projekt konfiguriert ist.
	import-db	(Alias für db-import)

Datenbank-Befehle *Befehle zur Verwaltung der Datenbank via Datenbank-Treiber. Nur verfügbar wenn, ein Projekt geladen ist und ein Dienst mit der Rolle „db“ im Projekt konfiguriert ist. Details zu der Datenbank-Verwaltung sind unter 5.3.4 Datenbank-Treiber erläuert.*		
	db-switch NAME	Ändert die aktuelle Datenbank-Umgebung auf NAME, falls NAME existiert.
	db-list	Listet alle Datenbank-Umgebungen auf.
	db-status	Zeigt den Namen der aktuellen Datenbank-Umgebung und den Status des Datenbank-Dienstes.
	db-new [--stay;-s] NAME	Erstellt eine neue Umgebung NAME und wechselt zu dieser Umgebung. Falls „--stay“ gesetzt ist, wechsle nicht.
	db-copy [--stay;-s] ALT NEU	Kopiert die Umgebung ALT nach NEU und wechselt zur Umgebung NEU. Falls „--stay“ gesetzt ist, wechsle nicht.
	db-drop NAME	Löscht die Umgebung NAME, kann nicht die aktuelle sein.
	db-import DATEI	Importiert ein Datenbank-Abbild DATEI in die aktuelle Umgebung. Format hängt vom Datenbank-Treiber ab.
	db-export DATEI	Export ein Datenbank-Abbild der aktuellen Umgebung nach DATEI. Format hängt vom Datenbank-Treiber ab.
Konfigurations-Befehle *Befehle zum Verändern und Auslesen der geladenen Konfiguration.*		
	config-edit-user [--factoryreset]	Bearbeitet die Systemkonfiguration im konfigurierten Editor (vgl. *5.5.1 Systemkonfiguration*). Falls --factoryreset gesetzt ist oder keine Konfiguration existiert, so wird eine neue angelegt.
	config-edit-project [--factoryreset]	Bearbeitet die Projektkonfiguration im konfigurierten Editor (vgl. *5.5.2 Projekte*). Falls --factoryreset gesetzt ist oder keine Konfiguration existiert, so wird eine neue angelegt.
	config-dump	Gibt ein YAML-Abbild der aktuellen vollständigen Systemkonfiguration aus. In die Systemkonfiguration ist der Knoten „project“ eingefügt, darunter befindet sich ein Abbild der Projektkonfiguration.

5.3.3 Container-Backend

Die Riptide Bibliothek besitzt eine Schnittstellendefinition für Container-Backends. Ein Container-Backend implementiert das Starten, Stoppen und allgemeine Verwalten von

Containern über eine Container-Engine, wie beispielsweise Docker. Container werden von Riptide eingesetzt, um Dienste und Kommandos (vgl. Kapitel 5.5) auszuführen.

Das System setzt dabei eine Container-Engine voraus, welche nach dem Vorbild von Docker Konzepte wie Volumes, Images, Port-Bindings und Umgebungsvariablen unterstützt. Als Container-Engine ist jede Virtualisierungslösung möglich, die diese Konzepte bereitstellen kann. Für die Definition von Images in den Definitionsdateien werden Docker Images verwendet. Andere Container-Engines müssen entweder mit Docker Images arbeiten können, oder das Container-Backend muss entsprechende Adapter bereitstellen, um diese Images verwenden zu können. Theoretisch wären auch Virtualisierungslösungen über klassische virtuelle Maschinen möglich, da auch klassische VMs auf Abbildern basieren, Port-Weiterleitungen erlauben und in der Regel Möglichkeiten bereitstellen, um Dateien mit dem Host-System zu teilen.

Als Teil dieser Arbeit wird eine Referenz-Implementation für ein Container-Backend entwickelt. Da Docker eine weitverbreite Lösung für Software-Container ist und nativ Docker Images unterstützt, wird für die initiale Umsetzung des Systems ein Container-Backend auf Basis von Docker umgesetzt.

Ablaufsicht

Das Backend wird über Aufruf seiner Backend-Schnittstellen angewiesen eine Aufgabe zu erfüllen. Eine der komplexesten Aufgaben ist der Start-Befehl für Dienst-Container von Projekten. Dieser wird nachfolgend genauer erläutert.

Das Backend erhält die Namen der zu startenden Dienste und das Projekt-Objekt. Jeder Dienst wird parallel gestartet. Der Aufrufer bekommt einen Rückgabewert, über den er über Änderungen während des Start-Prozesses informiert wird. Falls ein Dienst-Container bereits läuft, so wird für diesen Dienst keine Aktion durchgeführt. Andernfalls wird er anhand der Definition des Dienstes gestartet. Falls „pre_start“ (vgl. Kapitel 5.5.4) gesetzt ist, werden für jeden Eintrag vor dem Start des Dienst-Containers diese Befehle in separaten Containern ausgeführt. „post_start“-Befehle werden nach dem Start des Dienst-Containers in diesem ausgeführt.

Abbildung 5.5 zeigt das Verhalten von Container-Backends beim Start-Prozess. Dieser ganze Prozess ist in seinen Implementierungsdetails im Kapitel *6.3 Start-Vorgang von Dienst-Containern* ausführlicher erläutert.

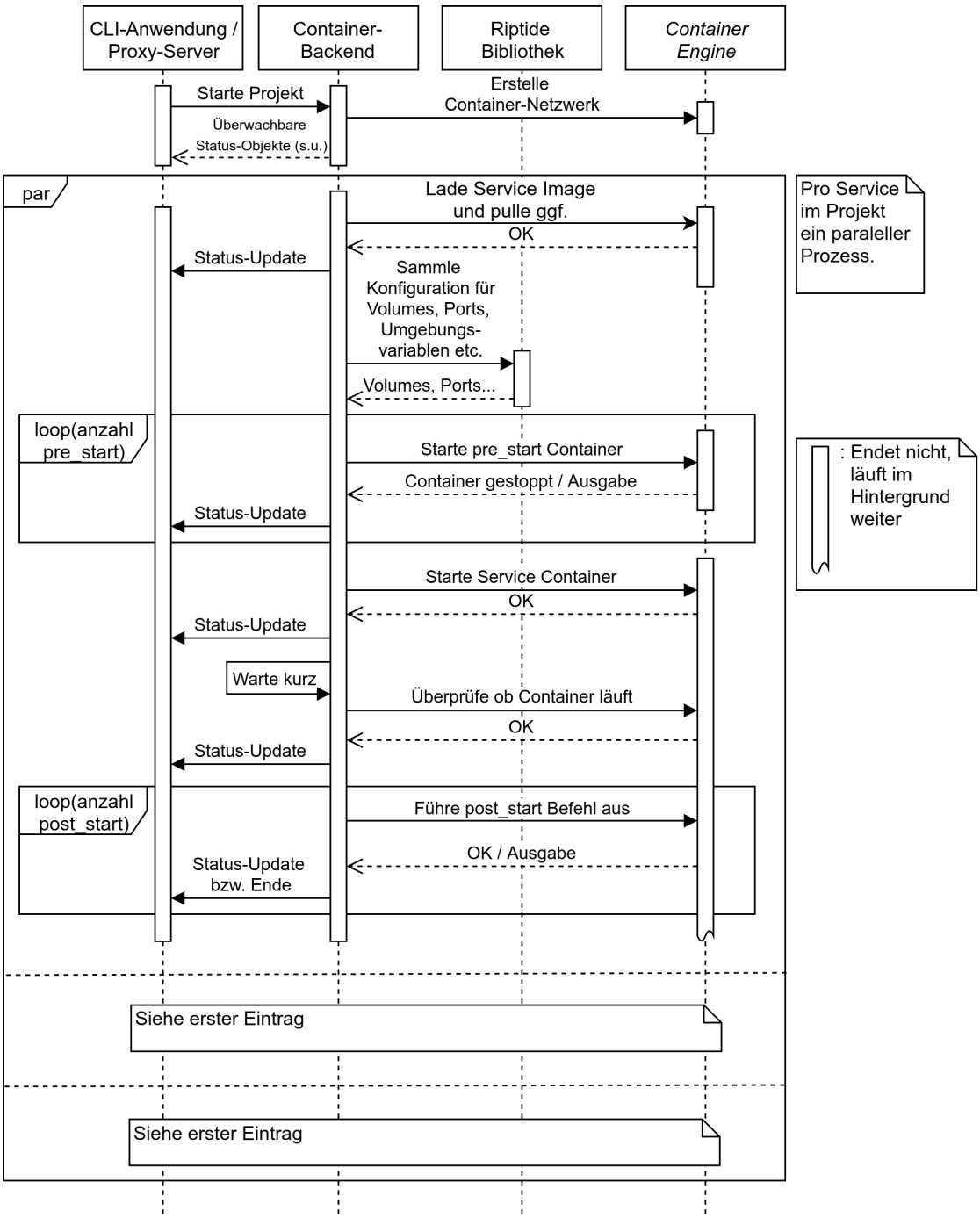


Abbildung 5.5: Ablaufsicht auf den Start-Prozess, beginnend an der Container-Backend-Schnittstelle (UML Sequenzdiagramm)

Entwicklungssicht

Container-Backends müssen die Container-Backend-Schnittstelle der Riptide Bibliothek implementieren und sich über Werkzeuge der Python Paketverwaltung als Plugin für Riptide registrieren. Implementationsdetails zur Docker Implementation finden sich im Kapitel *6.2 Implementation des Docker Container-Backends*.

Logische Sicht und Schnittstellen

Container-Backends implementieren die Container-Backend-Schnittstelle der Riptide Bibliothek. Nachfolgend findet sich eine Liste der Kern-Funktionen dieser Schnittstelle.

Projekte (Dienste) starten
: Diese Schnittstelle wird genutzt, um Dienste in Projekten zu starten. Details sind unter *Ablaufsicht* beschrieben.

Projekte (Dienste) stoppen
: Analog zum Start-Prozess stoppt ein Aufruf dieser Schnittstelle Projekt-Dienste.

Status von Dienst-Containern abrufen
: Gibt für eine Liste von Diensten in einem Projekt zurück, ob Container für diese Dienste gestartet sind oder nicht.

Netzwerk-Konfiguration von Dienst-Containern abrufen
: Gibt für einen Dienst in einem Projekt die IP-Adresse und den Port zurück, unter dem der Dienst über HTTP erreichbar ist. Falls der Container nicht läuft oder „port“ im Dienst nicht gesetzt ist (vgl. Kapitel 5.5.4), wird kein Wert zurückgegeben.

Kommando-Entität ausführen
: Führt eine Kommando-Entität eines Projektes als Container aus. Siehe *5.7.2 Kommandos von Anwendungen*. Kann nur in Konsolen ausgeführt werden. Die Kommando-Entität wird direkt interaktiv in der Konsole ausgeführt. Es gibt eine zusätzliche Schnittstelle, um Kommando-Objekte im Hintergrund (nicht interaktiv) auszuführen.

Interaktive Konsole für Dienst-Container öffnen
: Öffnet eine interaktive Konsole für einen Dienst-Container. Kann daher nur in Konsolen ausgeführt werden.

Neben der Container-Backend-Schnittstelle bieten Konfigurations-Entitäten der Dienste und Kommandos (vgl. Kapitel 5.5) Schnittstellen, um Daten wie Volumes, Umgebungsvariablen und Port-Bindings abzurufen. Die Container-Backend-Implementationen rufen diese Schnittstellen auf, um diese entsprechenden Daten zu sammeln und Container entsprechend zu starten.

5.3.4 Datenbank-Treiber

Datenbank-Treiber implementieren die Datenbank-Treiber-Schnittstelle der Riptide Bibliothek. Über diese Schnittstelle werden dem Benutzer von Riptide Funktionen bereitgestellt, um Datenbank-Management-Systeme (DBMS) zu Verwalten, falls das Projekt eine Datenbank bereitstellt. Für einzelne DBMS muss es jeweils eine Implementation dieser Schnittstelle geben.

Ablaufsicht

Der Datenbank-Treiber wird über Aufruf seiner Backend-Schnittstellen angewiesen eine Aufgabe zu erfüllen. Eine dieser Aufgaben ist das Einspielen eines Datenbank-Abbildes. Dazu startet der Treiber über das Container-Backend einen Container, welcher einen Befehl ausführt, um das übergebene Abbild in die Datenbank des gestarteten Datenbank-Dienstes einzuspielen. Das Format des Abbildes wird vom Treiber vorgegeben.

Entwicklungssicht

Datenbank-Treiber müssen die Datenbank-Treiber-Schnittstelle der Riptide Bibliothek implementieren und sich über Werkzeuge der Python Paketverwaltung als Plugin für Riptide registrieren. Durch diesen Mechanismus werden in der Dienst-Definition eingetragene Namen den Datenbank-Treibern zugeordnet (siehe *5.5.4 Dienste*, Schlüssel „driver“).

Logische Sicht und Schnittstellen

Die CLI-Anwendung bietet die Möglichkeit DBMS über Datenbank-Treiber zu verwalten. Zunächst muss im Projekt ein Dienst mit der Rolle „db“ existieren (vgl. *5.5.4*, Schlüssel „roles“). Ein solcher Dienst muss zunächst einen Schlüssel „driver“ gesetzt haben. Dieser

Schlüssel besteht aus Schlüssel-Wert-Paaren (Map). In dieser Map muss der Name des Datenbank-Treibers und seine Konfiguration hinterlegt werden.

Die Datenbank-Treiber-Schnittstelle bietet die benötigten Import- und Exportfunktionen und erlaubt dem Treiber das Dienst-Objekt mit zusätzlichen Ports oder Volumes zu erweitern, die für den Betrieb der Datenbank notwendig sind (vgl. Schlüssel unter *5.5.4 Dienste*).

Neben der Import- und Exportfunktionen stellt die Riptide CLI Befehle bereit, um die aktuelle Datenbank-Umgebung zu verwalten. Eine Datenbank-Umgebung besteht aus sämtlichen Daten, die das DBMS nutzt. Der Anwender kann neue Datenbank-Umgebungen erstellen, bestehende kopieren und zwischen ihnen wechseln. Das erlaubt Entwicklern mit mehreren Versionen der Datenbank zu arbeiten und zwischen diesen zu wechseln.

5.3.5 Proxy-Server

Das System bietet einen HTTP(S) Reverse-Proxy Server, welcher dem Benutzer ermöglicht auf HTTP-Schnittstellen von Diensten zuzugreifen und nicht gestartete Dienste automatisch starten zu lassen.

Ablaufsicht

Greift der Benutzer auf den Reverse Proxy zu, so wertet dieser den Namen des angefragten Projektes und Dienstes aus. Ist „riptide.local“ als Adresse für den Proxy Server eingetragen, so würde beispielsweise für „projektname–dienstname.riptide.local“ der Proxy-Server versuchen den Dienst „dienstname“ von „projektname“ aufzulösen. Um diese Auflösung durchzuführen, benötigt der Proxy-Server eine Liste von Projekten.

Die Riptide CLI-Anwendung schreibt eine Zuordnung von Projektnamen zu Projektdateien-Pfad in eine zentrale Datei. Der Proxy-Server lädt diese Datei und versucht zu „projektname“ eine Projektdatei zu finden. Findet er eine Projektdatei, so versucht er den Dienst „dienstname“ zu finden. Ist kein Dienstname angegeben, versucht der Proxy-Server stattdessen den ersten Dienst mit der Rolle „main“ zu finden.

Sobald ein Dienst gefunden wurde, wird die Container-Backend Schnittstelle nach der IP-Adresse und Port des Dienstes gefragt. Meldet die Container-Schnittstelle, dass der Dienst läuft und gibt diese Daten zurück, so startet der Proxy-Server eine HTTP-Anfrage.

Dabei leitet der Proxy alle Header aus der Anfrage an den Container weiter. Er setzt außerdem zusätzlichen Header. Diese zusätzlichen Header teilen der Endanwendung mit, welche IP-Adresse ursprünglich die Anfrage gestellt hat und welches Protokoll ursprünglich benutzt wurde (HTTPS oder HTTP). Diese Header sind unter anderem Header, die von der IETF aktuell im RFC7239 (`https://tools.ietf.org/html/rfc7239`) standardisiert werden (wie X-Forwarded-For und X-Forwarded-Proto). Das Ergebnis der Anfrage wird dann an den Benutzer weitergeleitet, inklusive dem HTTP Status Code.

Meldet der Dienst-Container bei der HTTP-Anfrage einen Fehler, so wird dieser direkt an den Benutzer weitergeleitet. Gibt es in der Kommunikation mit dem Dienst-Container einen Fehler, so wird eine Fehlerseite des Proxy-Servers mit entsprechender Fehlermeldung angezeigt. Wird ein Dienst oder Projekt nicht gefunden, so wird ebenfalls eine Fehlerseite ausgespielt.

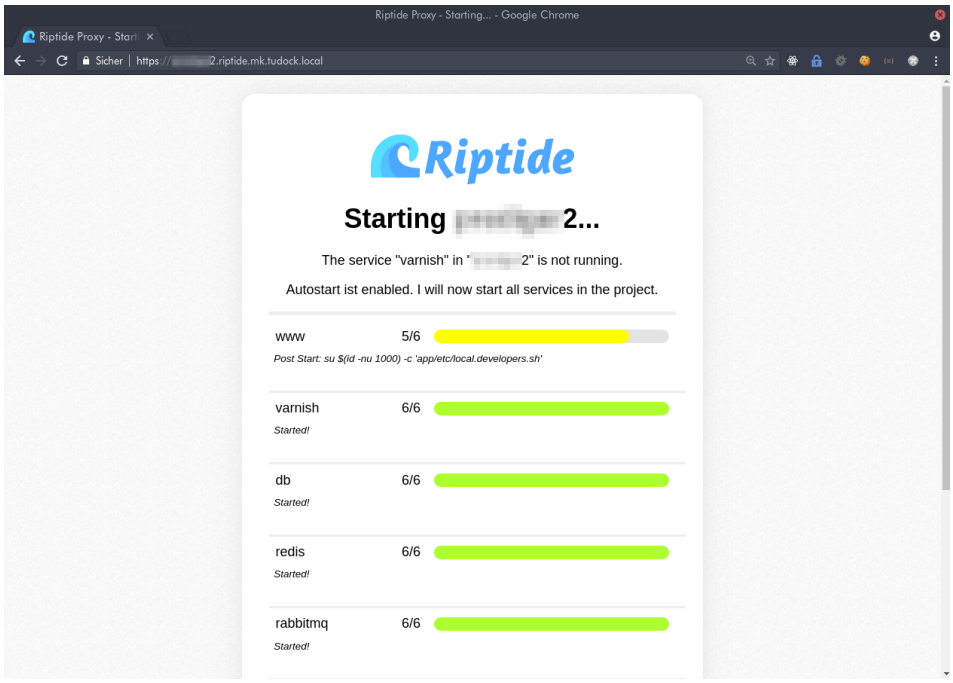


Abbildung 5.6: Auto-Start Funktion des Proxy-Servers, aufgerufen im Google Chrome Browser. Projektname wurde zensiert.

Falls ein Dienst nicht gestartet ist und in der Systemkonfiguration „proxy.autostart“ aktiviert ist, so startet der Proxy-Server über die Riptide Bibliothek alle Dienste im Projekt. Dem Benutzer wird dabei Rückmeldung in Form von Fortschrittsbalken gegeben (siehe Abbildung 5.6). Der Auto-Start Prozess entspricht dem Start-Prozess über die

CLI-Anwendung (vgl. Abbildung 5.4). Sobald der Auto-Start Prozess abgeschlossen ist, leitet der Proxy-Server zur angefragten Adresse im laufenden Dienst-Container weiter.

Ist kein Projektname oder Dienstname angegeben, so zeigt der Proxy-Server eine Startseite mit einer Liste aller registrierten Projekte und Dienste.

Neben der HTTP-Proxy Funktion bietet der Proxy-Server eine WebSocket-Proxy Funktion. Viele moderne Webdienste benötigen WebSockets, um korrekt zu funktionieren. Der Proxy-Server nimmt WebSocket Anfragen an und leitet sie entsprechend an Dienst-Container weiter.

Entwicklungssicht

Der Proxy-Server nutzt als Grundlage das Python Web-Framework Tornado (`https://www.tornadoweb.org/en/stable/`). Tornado bietet einen HTTP-Server, HTTP-Client, WebSocket-Server und WebSocket-Client an und stellt somit alle benötigten Grundfunktionen des Proxy-Servers zur Verfügung. Tornado erlaubt es, mehrere Anfragen gleichzeitig, nicht-blockierend, zu verarbeiten. Weitere Details zur Entwicklungssicht sind im Kapitel *6.4 Implementationsdetails der Proxy-Server-Komponente* beschrieben.

Logische Sicht und Schnittstellen

Das Verhalten des Proxy-Servers wird über die Systemkonfiguration gesteuert (vgl. 5.5.1, Schlüssel „proxy“).

Die Basis-URL des Proxy-Servers („proxy.url“) ist die Domain über die der Proxy-Server erreichbar ist. Dienste sind über Subdomains dieser Domain erreichbar. Die Einstellungen unter „proxy.ports“ erlauben die Angabe eines HTTP und HTTPS Ports, unter welchem der Proxy Server auf eingehende Verbindungen lauscht. Der Proxy Server nutzt auf dem HTTPS Port ein selbstsigniertes SSL/TLS-Zertifikat. Dem Anwender ist es möglich dieses Zertifikat als vertrauenswürdig einzustufen, sodass er den Proxy-Server ohne SSL-Warnungen oder Zugriffsfehler nutzen kann.

5.4 Wahl der Programmiersprache und Laufzeitumgebungen

Als Programmiersprache für die Implementierung aller Riptide Komponenten wurde Python gewählt. Python ist eine weit verbreitete Programmiersprache, mit der sich alle benötigten Komponenten umsetzen lassen. Python wird üblicherweise interpretiert.

Funktionen der Sprache, wie Asyncio für asynchrone Programmierung, bieten zusätzlich Möglichkeiten, schnell einfach lesbaren und testbaren Programmcode zu erzeugen. Docker bietet eine Bibliothek zur Interaktion mit der Docker API und verbreitete Web Frameworks wie Tornado erlauben es, für wichtige Bestandteile der Anwendung auf fundierte Bibliotheken zurückzugreifen.

Python bietet zudem standardisierte Möglichkeiten, um Software-Pakete zu Verteilen und Plugins für Software-Pakete zu erstellen („entry_points“ in der „setup.py“ eines Paketes). Python bietet zudem einfache Möglichkeiten CLI-Befehle zu definieren, die nach der Installation eines Python-Paketes direkt im „PATH“ des Benutzers sind und sich somit systemweit ausführen lassen.

Als Programmiersprache wurde für die CLI-Anwendung PHP in Betracht gezogen, da es in dem Unternehmen, in dem diese Arbeit geschrieben wurde, als primäre Programmiersprache eingesetzt ist. Viele Web-Projekte verwenden PHP, wodurch potentielle Benutzer die Programmiersprache bereits installiert haben. Für PHP gibt es außerdem die Bibliothek Symfony Console (`https://symfony.com/doc/current/components/console.html`), welche es einfach erlaubt sehr benutzerfreundliche CLI-Anwendungen zu erstellen. Für Python wurde keine Bibliothek gefunden, die insgesamt einen solchen Funktionsumfang bietet. PHP erlaubt allerdings nicht ohne Weiteres die Ausführung von asynchronem Code oder zusätzlicher Threads. Außerdem benötigt PHP zusätzlich einen Webserver zum Einsatz über HTTP, wodurch es sich nicht gut als Basis für den Proxy-Server eignet. Aus diesem Grund wurde PHP als Programmiersprache für das System verworfen.

Als Programmiersprache für den Proxy-Server wurde JavaScript mit Node.js als Laufzeitumgebung in Betracht gezogen, da Node.js eine sehr verbreitete Lösung ist, um Web-Server aufzusetzen. Es gibt Server-Implementierungen für WebSocket und bestehende Bibliotheken für Reverse Proxies. Da mit Asyncio und dem Tornado Web Framework

für Python jedoch vergleichbare Lösungen existieren, wurde sich auch hier für Python entschieden.

Die Wahl einer zusätzlichen Programmiersprache für den Proxy-Server hätte zudem das Problem, dass die Anwender entweder zusätzliche Programmiersprachen installieren müssten, oder das System über Docker Container diesen Starten müsste. Dafür müsste das System allerdings die Funktionalitäten der Riptide Bibliothek als API-Server bereitstellen, damit der Proxy-Server im Container die Möglichkeit hat, Konfigurations-Entitäten zu Laden.

Alle Riptide Komponenten sind mit Python 3.6 und 3.7 (neuste zum Veröffentlichungszeitpunkt) kompatibel.

5.5 Konfigurations-Entitäten

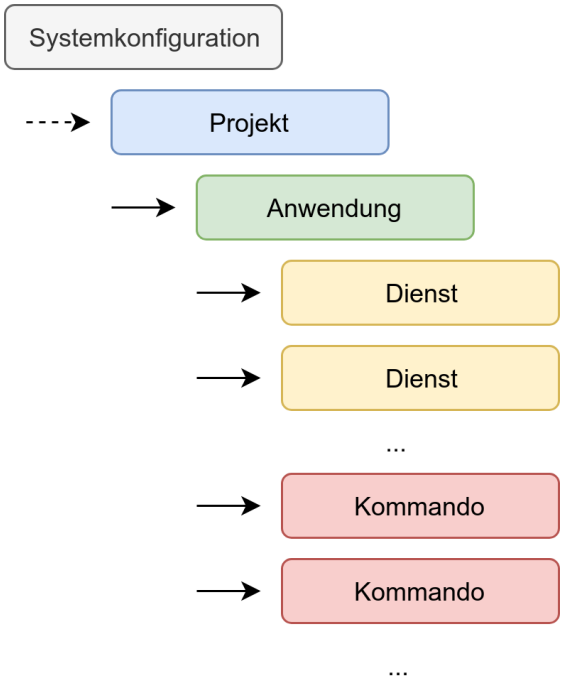


Abbildung 5.7: Hierarchische Struktur der Konfigurations-Entitäten

Zentraler Bestandteil des Systems sind die Konfigurations-Entitäten. Dabei handelt es sich um Repräsentationen der *Systemkonfiguration* und der Konfiguration der einzelnen *Projekte* und deren Bestandteile.

Diese Entitäten können von Benutzern in Form einer auf YAML basierenden Definitionssprache in Dateien geschrieben werden (siehe *5.6 Definitionssprache für Entwicklungsumgebungen*). Die Konfigurations-Entitäten beschreiben, woraus Projekte bestehen, welche Container gestartet werden müssen und welche Interaktionsmöglichkeiten der Benutzer mit dem Projekt hat.

Die einzelnen Entitäten stehen hierarchisch miteinander in Beziehung (siehe Abbildung 5.7). Die Systemkonfiguration wird aus einer zentralen Datei eingelesen. Die Projektdatei wird aus einer zusätzlichen Datei eingelesen. Jedes Projekt besitzt eine eigene Projektdatei. Aus der Projektdatei ergeben sich die weiteren geladenen Entitäten.

Entitäten bestehen aus einer Reihe von Schlüssel-Wert-Paaren (*Maps*). Die Schlüssel und ihre Werte bestimmen das Verhalten der einzelnen Riptide Komponenten. Aus diesem Grund wird die Bedeutung der einzelnen Schlüssel in den folgenden Abschnitten beschrieben. Es lässt sich so ein Überblick über die Funktionsweise von Riptide definieren.

5.5.1 Systemkonfiguration

Die *Systemkonfiguration* (*Config*) beinhaltet zentrale Einstellungen für alle Komponenten von Riptide. Pro Benutzer auf dem Anwender-PC gibt es eine Systemkonfiguration.

Schlüssel:

proxy.url
: Definiert die Basis-URL, unter der Projekte mit dem Proxy-Server erreichbar sein sollen.

proxy.ports
: Definiert den HTTP und HTTPS Port, auf dem der Riptide Proxy-Server lauschen soll.

proxy.autostart
: Aktiviert oder deaktiviert die Autostart-Funktion des Proxy-Servers (siehe *5.3.5 Proxy-Server*).

update_hosts_file
: Aktiviert oder deaktiviert, dass Projekt-Hostnamen in die System *hosts*-Datei (vgl.

2.6 Namensauflösung und hosts-Datei) eingetragen werden, um einfaches Routing des Proxy-Servers ohne DNS zu ermöglichen.

engine
: Name des Container-Backends, siehe *5.3.3 Container-Backend*.

repos
: Liste von Repositories, siehe *5.6.3 Repositories*.

5.5.2 Projekte

Projekte (*Projects*) repräsentieren die einzelnen Anwendungsprojekte, an denen die Entwickler arbeiten. Projekte werden in Projektdateien defininiert. Alle Unterordner eines Ordners, in dem eine Projektdatei vorkommt, werden als Teil dieses Projektes betrachtet. Die Konzeption geht davon aus, dass die Projektdatei zusammen mit dem Quellcode eines Projektes bereitgestellt wird.

Schlüssel:

name
: Name des Projektes, dieser wird zur eindeutigen Identifikation eines Projektes und einer Projektdatei benutzt.

src
: Pfad zu dem Quellcode des Projektes, relativ zur Projektdatei. Containern werden Dateien innerhalb dieses Pfades zur Verfügung gestellt.

app
: Anwendung des Projektes.

5.5.3 Anwendungen

Anwendungen (*Apps*) repräsentieren jeweils eine Webanwendung. Die Anwendungen bestehen aus einer Reihe von Diensten, die zur Ausführung benötigt werden. Dazu können Webserver, Datenbanken oder die Webanwendung selbst gehören. Anwendungen können eine Reihe von Kommandos definieren.

Schlüssel:

name
: Name der Anwendung.

notices
: Hinweistexte, die dem Benutzer bei der Einrichtung der Entwicklungsinstanz angezeigt werden (siehe *5.7.1 Ersteinrichtung eines Projektes*). Optional.

import
: Definition von Pfaden, die für diese Anwendung benutzt werden können, um Dateien zu importieren. Der Benutzer wird bei der Einrichtung der Entwicklungsinstanz gefragt, ob er Dateien importieren möchte. Siehe *5.7.1 Ersteinrichtung eines Projektes*. Optional.

services
: Liste von Diensten für die Anwendung. Die Liste besteht aus Maps. Schlüssel sind die Namen der Dienste, Werte die Dienste selbst. Optional.

commands
: Liste von Kommandos für die Anwendung. Die Liste besteht aus Maps. Schlüssel sind die Namen der Kommandos, Werte die Kommandos selbst. Optional.

5.5.4 Dienste

Dienste (*Services*) repräsentieren einzelne „Unter-Anwendungen“ einer Anwendung. Ein Dienst entspricht einem Container. Die Konfiguration von Diensten ist an die Container-Definiton von Docker Containern angelehnt.

Schlüssel:

roles
: Liste von Rollen, die der Dienst erfüllt. Optional. Kann beliebige Schlüsselwörter beinhalten. Einige Schlüsselwörter sind vordefiniert:

  main Der Dienst ist der Haupt-Einstiegspunkt für das Projekt. Er ist direkt über den Proxy-Server erreichbar (bspw. „projektname.riptide.local“), wohingegen

für die anderen Dienste der Name des Dienstes in der URL angegeben werden muss (bspw. „projektname–dienstname.riptide.local“).

db Dieser Dienst ist die zentrale Datenbank des Projektes. Ein Datenbank-Treiber (*driver*) wird beim Dienst angegeben.

src Der Dienst benötigt Zugriff auf den Quellcode der Anwendung und bekommt diesen als Volume zur Verfügung gestellt.

image

Docker Image, dass für die Ausführung des Dienstes benutzt werden soll.

command

Kommando, das ausgeführt werden soll, wenn der Container startet. Entspricht Docker CLI „--command“ Option. Optional, Standard ist im Image definiert.

working_directory

Arbeitsverzeichnis des Kommandos im Container. Entspricht Docker CLI „--workdir“ Option. Optional, Standard ist im Image definiert.

port

Port auf dem im Container auf HTTP-Traffic gelauscht wird. Der Benutzer wird über den Proxy-Server mit diesem Container-Port verbunden. Optional, wenn nicht gesetzt, dann ist der Container über den Proxy-Server nicht erreichbar.

logging

Konfiguration für das Logging, siehe *5.7.5 Logging*. Optional.

pre_start

Kommandos, die ausgeführt werden sollen, bevor der Container für den Dienst startet. Diese Kommandos werden ebenfalls in Containern ausgeführt, die Konfiguration dieser Container basiert auf der Konfiguration des Dienstes. Optional.

post_start

Kommandos, die nach dem Start des Dienst-Containers im Dienst-Container ausgeführt werden sollen. Optional.

environment

Zusätzliche Umgebungsvariablen für den Container. Entspricht Docker CLI „--env“ Option. Optional.

config

Definition von Konfigurationsdateien, die als Volumes dem Container zur Verfügung gestellt werden. Siehe *5.7.4 Konfigurationsdateien von Diensten*. Optional.

additional_ports

Liste von TCP- und UDP-Ports, die der Dienst zusätzlich zur Verfügung stellt. Siehe *5.7.3 Zugriff auf zusätzliche Dienst-Ports*. Optional.

additional_volumes

Liste von zusätzlichen Dateien oder Ordnern, welche Containern als Volumes zur Verfügung gestellt werden. Entspricht Docker CLI „--volume“ Option. Optional.

driver

Datenbank-Treiber, falls die „db“-Rolle gesetzt ist. Siehe *5.3.4 Datenbank-Treiber*.

run_as_current_user

Im Normalfall werden Container mit der Benutzer- und Gruppennummer des Benutzers ausgeführt, der Riptide bedient. Ist diese Option deaktiviert, so wird der Container stattdessen mit den Benutzer- und Gruppennummern ausgeführt, die im Image definiert sind. Optional.

5.5.5 Kommandos

Kommandos (*Commands*) sind CLI-Anwendungen, welche Entwicklern erlauben mit der Anwendung zu interagieren. Riptide stellt über die CLI-Schnittstelle Möglichkeiten bereit, ein Kommando im aktuellen Arbeitsverzeichnis auszuführen. Der Entwickler kann einem Kommando Argumente übergeben. Die CLI-Anwendung wird dann mit diesen Argumenten gestartet. Siehe *5.7.2 Kommandos von Anwendungen*.

Aliase. Es können zusätzlich Aliase für Kommandos erstellt werden. Führt ein Entwickler ein Alias-Kommando aus, so wird stattdessen das zugeordnete Kommando ausgeführt. Alias-Kommandos besitzen nur einen Schlüssel *aliases*, welcher den Namen des referenzierten Kommandos beinhaltet.

Schlüssel:

image
: Docker Image, dass für die Ausführung des Kommandos benutzt werden soll.

command
: Kommando, das ausgeführt werden soll, wenn der Kommando-Container startet. Entspricht Docker CLI „--command“ Option. Optional, Standard ist im Image definiert.

environment
: Zusätzliche Umgebungsvariablen für den Container. Entspricht Docker CLI „--env“ Option. Optional.

additional_volumes
: Liste von zusätzlichen Dateien oder Ordnern, welche Containern als Volumes zur Verfügung gestellt werden. Entspricht Docker CLI „--volume“ Option. Optional.

5.6 Definitionssprache für Entwicklungsumgebungen

Die Konfigurations-Entitäten werden vom System aus Definitionsdateien gelesen. Inhalt der Dateien ist eine Definitionssprache basierend auf der Auszeichnungssprache YAML. Die Definitionsdateien werden nachfolgend auch als *Dokumente* bezeichnet.

Jede Definitionsdatei enthält ein einzelnes YAML-Dokument. Das YAML-Dokument enthält eine assoziative Liste (unsortierte Schlüssel-Wert-Paare; *Map*) mit einem Schlüssel, der den Typ der Konfigurations-Entität angibt. Der Wert hinter diesem Schlüssel ist eine Map. Der Inhalt dieser Map basiert auf dem Schema der Konfigurations-Entität. Definitionsdateien müssen dem Schema ihrer entsprechenden Konfigurations-Entität entsprechen. Das Schema der Entitäten basiert jeweils auf den im vorherigen Kapiteln aufgelisteten Schlüsseln. Quellcode 5.1 zeigt eine gültige Definitionsdatei für eine *Systemkonfiguration* Entität.

YAML wurde als Basis gewählt, da es eine Auszeichnungssprache ist, die sowohl für Menschen, als auch für Computer einfach lesbar ist. YAML wird als Auszeichnungssprache auch für ähnliche Lösungen, wie Ansible, Docker Compose und Kubernetes eingesetzt.

```yaml
# Dies ist ein YAML-Kommentar (wird nicht ausgewertet)
riptide:
  proxy:
    url: riptide.local
    ports:
      http: 80
      https: 443
    autostart: true
  update_hosts_file: true
  repos: []
  engine: docker
```

Quellcode 5.1: Beispiel für eine Riptide Systemkonfiguration

Für die Umsetzung, der in diesem Abschnitt beschriebenen Eigenschaften und dem Einlesen solcher Definitionsdateien, wird eine eigene Bibliothek (*Configcrunch*) geschrieben. Details zu dieser Bibliothek sind im Kapitel *5.6.4 Bibliothek zur Verarbeitung der Sprache (Configcrunch)* erläutert.

5.6.1 Eigenschaften der Sprache

Die Definitionssprache bietet die Möglichkeit, Dokumente mit anderen Dokumenten zu überlagern und Unter-Dokumente einzubinden. Darüber hinaus können Benutzer Variablen in der Definitionssprache verwenden, die von Riptide ausgewertet werden. Diese Funktionalitäten der Sprache werden in den folgenden Abschnitten erläutert.

Überlagerung von Dokumenten

Dokumente können mit Dokumenten des gleichen Typs überlagert werden. Wird ein Dokument vom System geladen (*Dokument A*), so wird im Dokument der Schlüssel „*$ref*“ gesucht. Wird dieser gefunden, wird versucht ein anderes Dokument anhand des Wertes zu laden. Der Wert enthält einen Pfad zu einem anderen Dokument, relativ zu einer definierten Liste von zu durchsuchenden Ordnern (siehe *5.6.3 Repositories*).

Das Dokument am angegeben Pfad wird geladen (*Dokument B*) und als Basis verwendet. Enthält dieses Dokument „$ref“-Einträge werden diese zunächst, wie in diesem Abschnitt beschrieben, aufgelöst.

Danach folgt die Überlagerung: Alle Einträge in Dokument B werden mit Einträgen aus Dokument A überschrieben. Die Zuordnung findet anhand des Schlüssels statt. Ist ein Eintrag in Dokument A nicht enthalten, so bleibt der Eintrag aus Dokument B erhalten. Maps werden rekursiv nach diesem Prinzip überlagert. Listen in Dokument A werden stattdessen mit Einträgen aus Dokument B ergänzt. Der „$ref“-Eintrag wird bei der Überlagerung entfernt.

Stößt der Algorithmus auf dem Weg im Dokument A auf den Wert „*$remove*“, so werden Schlüssel und Wert stattdessen aus Dokument B entfernt.

Sind mehrere zu durchsuchende Ordner angegeben, so werden alle Ordner durchsucht. Wird das Dokument in mehreren Ordnern gefunden, so werden diese nach definierter Reihenfolge zunächst untereinander überlagert. Bei diesem Überlagrungs-Prozess werden auch „$ref“-Einträge der Dokumente in niedriger priorisierten Ordnern mit denen höher priorisierter Ordner überschrieben. Erst nachdem diese Dokumente untereinander überlagert wurden, wird der „$ref“-Eintrag ggf. aufgelöst.

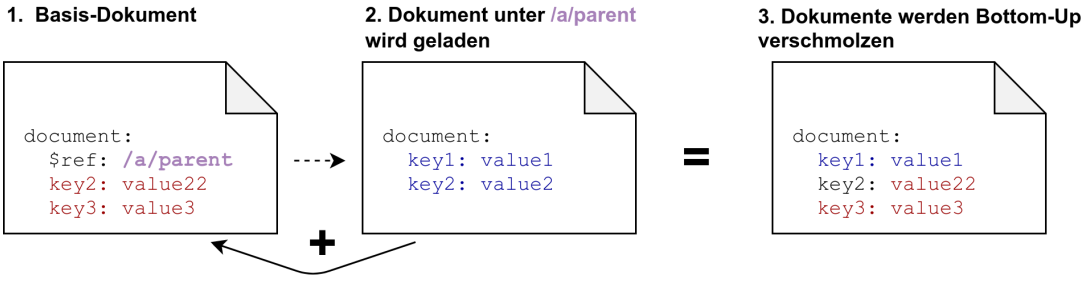


Abbildung 5.8: Beispiel für die Überlagerung von Dokumenten

Der Algorithmus arbeitet Bottom-Up: es werden zunächst rekursiv alle Referenzen durchgearbeitet und dann werden die Dokumente beginnend mit dem letzten „$ref“-Eintrag überlagert. Abbildung 5.8 zeigt ein einfaches Beispiel.

Der Algorithmus hat potenziell die Möglichkeit in eine Endlosschleife zu laufen, falls es Zirkelbezüge bei den „$ref“-Einträgen gibt. Das System merkt sich aus diesem Grund bereits geladene Dokumente und wirft einen Fehler, wenn versucht wird ein Dokument mehrfach mit demselben Dokument zu überlagern.

Die Überlagerung von Dokumenten erlaubt es, Teile von Dokumenten aufzuteilen und zu verteilen. Projektkonfigurationen lassen sich somit auf mehrere Projekte generalisieren: Ein Projektdatei referenziert ein gemeinsames Dokument, in dem allgemeine Projekteinstellungen für mehrere Projekte enthalten sind. In der Projektdatei selbst werden einige dieser Einstellungen überschrieben oder ergänzt.

Hierarchische Einbindung von Unter-Dokumenten

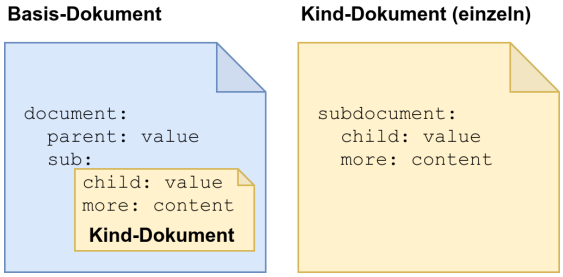


Abbildung 5.9: Beispiel für die Einbindung von Unter-Dokumenten

Dokumente können Unter-Dokumente beinhalten (siehe Abbildung 5.9). Unter-Dokumente sind vollwertige Dokumente und können beispielsweise ihrerseits „$ref“-Einträge beinhalten und überlagert werden (siehe auch 5.6.2). Unter-Dokumente unterliegen einem eigenem Schema und können separat von ihren Eltern-Dokumenten verwendet und validiert werden.

Eltern-Dokumente können in ihrem Validierungs-Schema angeben, dass an einer bestimmten Stelle Unter-Dokumente erwartet werden. Es ergibt sich somit einer Hierarchie von verschiedenen Dokumenten-Typen.

Die Aufteilung in Unter-Dokumente erlaubt es (in Kombination mit der Überlagerung von Dokumenten) einzelne Teile von Projekten (Dienste, Kommandos) in separate Dateien aufzuspalten und diese somit unabhängig von einzelnen Projekten zu definieren und einsetzen zu können.

Auswertung von Variablen

Dokumente können als Werte Zeichenketten beinhalten. Diese Zeichenketten können Variablen in Form von Schablonen-Zeichenketten beinhalten. Der Syntax der Schablonen

wird definiert durch die Python Templating-Engine Jinja2 (`http://jinja.pocoo.org/docs/2.10/templates/`). An dieser Stelle werden nur einfache Schablonen zum Einfügen von Variablen und Aufrufen von Funktionen definiert und betrachtet.

Die in den Schablonen referenzierten Variablen und Funktionen werden im Kontext des Dokumentes ausgewertet. Eine referenzierte Variable wird beim Auswerten statt der Schablone eingefügt. Eine referenzierte Funktion wird ausgeführt.

Die Auswertung beginnt immer bei der Wurzel des Dokuments. Beispiel: ein Dokument besitzt eine Map am Schlüssel „ebene1“. Diese besitzt den Schlüssel „ebene2“ („ebene1.ebene2“). Um den Wert hinter „ebene1.ebene2“ zu benutzen, muss überall im Dokument die Schablone „{{ebene1.ebene2}}“ verwendet werden.

Funktionen sind als Methoden an Klassen hinterlegt, die den jeweiligen Dokumenten-Typ repräsentieren (siehe auch Implementationsdetails, 5.6.4). Es wird standardmäßig eine Funktion „parent“ bereitgestellt, welche das Eltern-Dokument eines Unter-Dokuments zurückgibt. So können auch Werte von Eltern-Dokumenten abgerufen werden. Riptide definiert zusätzliche Funktionen, wie „domain“ für Dienste, um die vollständige Domain eines Dienstes abzurufen. Eine vollständige Liste von Hilfsfunktionen befindet sich in der angehängten Dokumentation (siehe 5.9).

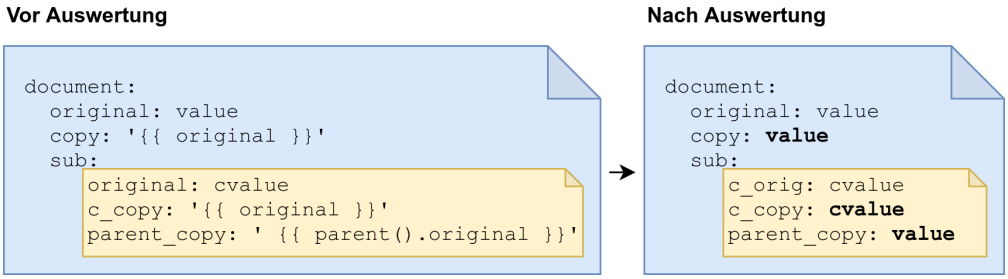


Abbildung 5.10: Beispiel für die Auswertung von Variablen

Für den Algorithmus werden alle Zeichenketten im Dokument nach Schablonen durchsucht und ausgewertet. Da Variablen Referenzen zu anderen Variablen enthalten können, wird der Algorithmus danach solange wiederholt, bis alle Schablonen abgearbeitet wurden, also keine Änderungen im Durchlauf des Algorithmus durchgeführt wurden. Abbildung 5.10 zeigt ein einfaches Beispiel des Algorithmus.

Es ist nicht möglich mit der „parent“-Hilfsfunktion Felder des Eltern-Dokumentes zu referenzieren, die ebenfalls Variablen beinhalten. Grund dafür ist, dass diese referenzierten

Felder noch nicht ausgewertet wurden, da immer zuerst Unter-Dokumente ausgewertet werden. Dadurch wird die Schablone des Eltern-Dokumentes direkt in das Unter-Dokument übernommen. Da die Schablone allerdings für ein anderes Dokument gedacht war, lässt sie sich im Kontext des Unter-Dokumentes nicht korrekt auswerten. Die Spezifikation und Implementation des Algorithmus könnte dahingehend in späteren Versionen des Systems verbessert werden.

Variablen erlauben es, andere Werte, insbesondere Werte in Unter-Dokumenten oder Basis-Dokumenten für die Überlagerung, zu referenzieren und zu verwenden. Dadurch entsteht eine größere Flexibilität, da zentrale Einstellungen an einer Stelle geändert werden können und an anderen Stellen übernommen werden. Beispiel: Das Projekt setzt eine MySQL Datenbank ein. Unter dem Pfad „services.db.driver.config.password“ kann im Projekt das Datenbank-Passwort gesetzt werden. Soll das Datenbank-Passwort in einem anderen Dienst verwendet werden (beispielsweise in Umgebungsvariablen), so kann ein anderer Dienst den Wert referenzieren: „{{parent().services.db.driver.config.password}}“.

5.6.2 Algorithmus zum Einlesen von Definitionsdateien

Die Kombination aus Überlagerung von Dokumenten, Einbindung von Unter-Dokumenten und der Auswertung von Variablen ergeben den finalen Algorithmus.

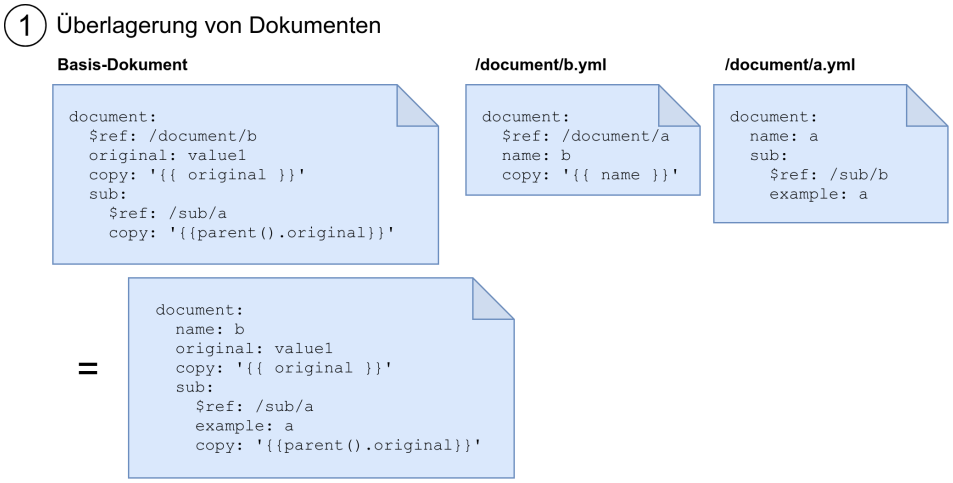


Abbildung 5.11: Algorithmus zum Einlesen (1/3): Überlagerung von Dokumenten

Im ersten Schritt wird das eingelesene Dokument mit anderen Dokumenten überlagert (siehe Abbildung 5.11).

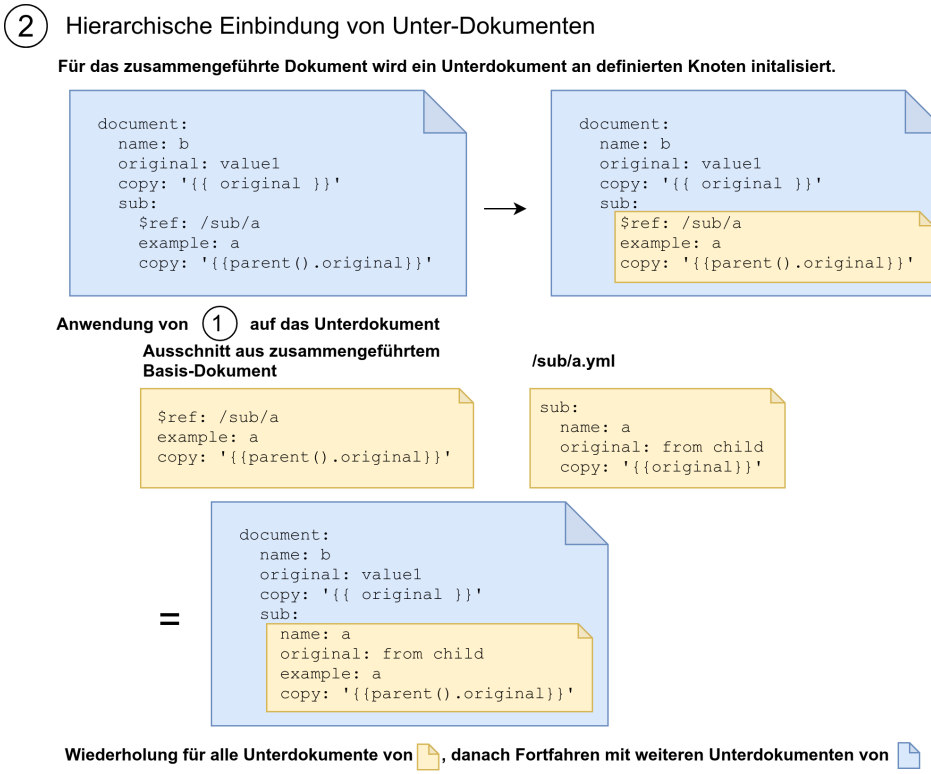


Abbildung 5.12: Algorithmus zum Einlesen (2/3): Einbindung von Unter-Dokumenten

Im nächsten Schritt werden Unter-Dokumente des Dokumentes initialisiert. Danach wird der Algorithmus für dieses Dokument von vorne begonnen (Überlagerung und Initialisierung von Unter-Unter-Dokumenten). Bei der Überlagerung werden eigenständige Dokumente des Typs vom Unter-Dokument eingelesen (siehe Abbildung 5.12).

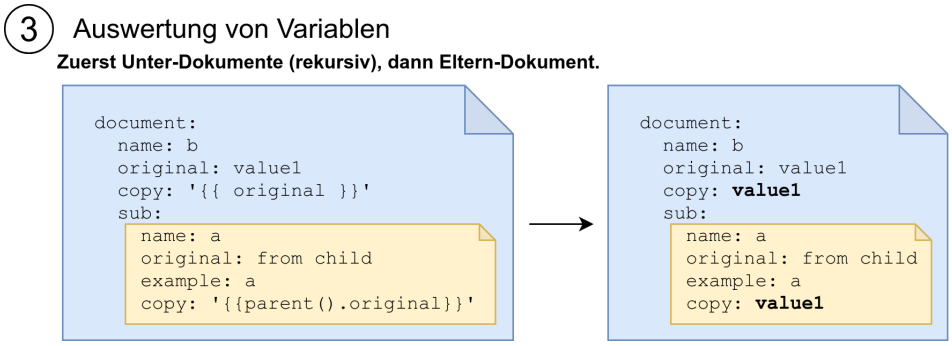


Abbildung 5.13: Algorithmus zum Einlesen (3/3): Auswertung von Variablen

Nachdem alle Überlagerungen und Unter-Dokumente ausgewertet wurden, werden die Variablen ausgewertet (siehe Abbildung 5.13). Es werden zunächst alle Variablen von Unter-Dokumenten und danach vom Eltern-Dokument aufgelöst. Dieser Vorgang arbeitet rekursiv und arbeitet daher auch mit Unter-Dokumenten von Unter-Dokumenten.

5.6.3 Repositories

Um Dokumente zu überlagern müssen Pfade angegeben werden, die durchsucht werden, um Dokumente zu finden. Um diese Ordner zu verteilen, erlaubt es Riptide Git Repositories in der Systemkonfiguration anzugeben (vgl. 5.5.1, Schlüssel „repos“). Git ist ein Versionskontrollsystem, welches eingesetzt wird, um Software zu versionieren und Quellcode in Entwickler-Teams zu verteilen. Die Repositories werden von Riptide herunterladen.

Dieser Mechanismus ermöglicht Teams, Teile ihrer Infrastruktur und Konfiguration über mehrere Projekte zentral zu verwalten. Da die Definitionssprache auch das Durchsuchen mehrerer Pfade erlaubt, ist es möglich mehrere Git Repositories anzugeben. Es ergibt sich daraus eine Hierarchie und einzelne Git Repositories können sich gegenseitig überlagern.

Es wurde ein öffentliches (Open Source) Riptide Git Repository erstellt (*Riptide Community Repository*), in dem die Gemeinschaft der Riptide Anwender für verschiedenste Anwendungsfälle Definitionen bereitstellen kann. Dieses Repository enthält Definitionen für Anwendungen, Dienste und Kommandos, die Nutzer des Systems direkt verwenden können, ohne sich vollständig mit den Details das Systems und den Konfigurations-Entitäten auskennen zu müssen. Das Repository enthält unter anderem Definitionen für

die E-Commerce Plattform Magento (vgl. *7.2 Kompatibilität mit verschiedenen Webanwendungen und Frameworks*). Alle Entitäten sind nach einem einheitlichem Schema dokumentiert, um dem Anwender die Nutzung der Entitäten zu vereinfachen (vgl. *5.9 Dokumentation*)

Entwickler-Teams können das Riptide Community Repository für ihre Arbeit einsetzen. Sie haben zusätzlich die Möglichkeit, es mit ihrem eigenem Repository zu überlagern, um an einigen Definitionsdateien Team-spezifische Änderungen durchzuführen. Auf Projektebene können sie dann Änderungen in der Projektdatei durchführen.

5.6.4 Bibliothek zur Verarbeitung der Sprache (Configcrunch)

Die vorherigen Abschnitte beschreiben das Verhalten der Definitionssprache, die vom System eingesetzt werden soll. Da das System in Python implementiert wird, ist eine Bibliothek erforderlich, die aus YAML-Dateien Konfiguration, entsprechend der Definition der Sprache, in Python-Objekte einlesen kann. Prinzipiell wären auch andere Auszeichnungssprachen denkbar, solange sich mit der Bibliothek die drei zentralen Eigenschaften der Sprache (Überlagerung, Einbindung von Unterdokumenten und Auswertung von Variablen und Hilfsfunktionen) umsetzen lassen.

Zu diesem Zweck wurden verschiedene Bibliotheken untersucht. Nicht alle Eigenschaften der Sprache standen zum Beginn der Untersuchung fest. Eeinige Details der Sprach-Definition sind von den Eigenschaften dieser Bibliotheken inspiriert.

Python wird standardmäßig mit dem Modul *configparser* (`https://docs.python.org/3/library/configparser.html`) ausgeliefert. Dieses Modul erlaubt es, Konfiguration aus INI-Dateien einzulesen. Es wird keine Überlagerung von Dokumenten unterstützt. Eine Einbindung von Unter-Dokumenten ist ebenfalls nicht direkt möglich. Die Auswertung von Variablen wird allerdings unterstützt, nicht jedoch die Ausführung von Hilfsfunktionen.

Die Python-Bibliothek *Confire* (`https://pythonhosted.org/confire/`) unterstützt Überlagerung von Dokumenten und erlaubt es, Konfigurationswerte durch Umgebungsvariablen zu überschreiben. Es gibt einen Mechanismus, um Unter-Dokumente einzulesen („Nested Configurations“). Es gibt allerdings keinen Mechanismus, um in den Dateien Variablen aufzulösen oder Hilfsfunktionen aufzurufen.

Die Bibliothek *hiyapyco* (`https://pythonhosted.org/hiyapyco/`) unterstützt Überlagerung und Variablen-Auflösung ähnlich zur Definition in dieser Arbeit. *Templated YAML* (`https://nazarethcollege.github.io/templated-yaml/build/html/index.html`) unterstützt zusätzlich die Definition von Hilfsfunktionen. Beide unterstützen nicht die Einbindung von Unter-Dokumenten.

config (`https://www.red-dove.com/config-doc/`) unterstützt Überlagerung, Variablen-Auflösung und auch die Einbindung von Unter-Dokumenten. Bei der Einbindung von Unter-Dokumenten werden Eltern- und Kind-Dokument allerdings vollständig verschmolzen, sodass Auflösung von Variablen relativ zum Unter-Dokument nicht möglich ist. Die Definition von Hilfsfunktionen wird nicht unterstützt.

Nach Untersuchung dieser (und ähnlicher) Lösungen wurde in Betracht gezogen, *config* als Basis zu nehmen oder mehrere andere Lösungen zu kombinieren. Aufgrund verschiedener Eigenheiten der einzelnen Lösungen, die eine Kombination der Lösungen umständlich macht, und der Tatsache das *config* seit 2010 nicht mehr gewartet wird, wurde jedoch entschieden stattdessen eine neue Bibliothek zu schreiben. Diese Bibliothek trägt den Namen *Configcrunch*. Die Bibliothek liest YAML-Dateien über *pyyaml* (`https://pyyaml.org/`) ein und implementiert den Algorithmus zum Einlesen von Definitionsdateien.

Configcrunch ist von Riptide grundsätzlich unabhängig und kann auch von anderen Projekten eingesetzt werden. Die Bibliothek stellt eine abstrakte Klasse bereit (*YamlConfigDocument*), in der das Verhalten der Sprache implementiert ist. Die Riptide Bibliothek enthält Unterklassen dieser Klasse, welche jeweils eine Entität repräsentieren (siehe *6.1 Implementation der Riptide Bibliothek*).

5.7 Zusätzliche Funktionen

Das Riptide-System bietet einige weitere Funktionen an, die an dieser Stelle detailliert beschrieben werden.

5.7.1 Ersteinrichtung eines Projektes

Bevor ein Projekt von einem Entwickler gestartet werden kann, ist eine Ersteinrichtung über den „setup“-Befehl notwendig. Dieser Befehl wird von der CLI-Anwendung bereit-

gestellt.

Der Befehl startet einen interaktiven Assistenten zur Ersteinrichtung. Der Benutzer erhält in diesem Assistenten Hinweise zur Benutzung der Anwendung, falls diese gesetzt sind (vgl. *5.5.3*, Schlüssel „notes.usage“).

Danach kann der Entwickler auswählen, ob er eine Neuinstallation für die Anwendung ausführt und die Anwendung erst einmalig installiert werden muss, oder ob er die Arbeit an einem bestehenden Projekt beginnen möchte.

Entscheidet sich der Entwickler für die Neuinstallation, so wird ihm der Hinweistext für die Neuinstallation der Anwendung angezeigt (vgl. *5.5.3*, Schlüssel „notes.installation“). Danach endet der Assistent.

Entscheidet sich der Entwickler für die Arbeit an einem bestehenden Projekt, so wird er gefragt, ob er Dateipfade importieren möchte, die an der Anwendungs-Konfiguration hinterlegt sind (vgl. *5.5.3*, Schlüssel „import“). Beispielsweise kann eine Webanwendung Mediendateien für Besucher der Webseite anbieten, welche im Quellcode-Ordner im Pfad „pub/media“ liegen müssen. Dieser Pfad könnte in der Konfiguration eingetragen sein. Der Entwickler erhält dann die Frage, ob er Dateien für diesen Pfad importieren möchte und woher. Der Entwickler kann über den CLI-Befehl „import-files“ auch nach der Ersteinrichtung noch Dateien importieren.

Ist für die Anwendung eine Datenbank konfiguriert, so wird der Entwickler beim Import eines bestehenden Projektes zusätzlich nach dem Abbild für diese Datenbank zum Import gefragt. Diese Abfrage entspricht dem CLI-Befehl „db-import“ (siehe *Datenbank-Treiber*).

Die Import-Funktionen und die Hinweis-Texte sollen die Ersteinrichtung für Entwickler vereinfachen. Werden zur Ausführung der Anwendung Dateien, Datenbank-Abbilder oder die Ausführung zusätzlicher Befehle benötigt, so wird der Entwickler darauf hingewiesen und durch den Datenimport geführt.

5.7.2 Kommandos von Anwendungen

Für die Anwendung im Projekt können CLI-Kommandos definiert werden (vgl. *5.5.3 Anwendungen* und *5.5.5 Kommandos*). Diese CLI-Kommandos starten einen Container, basierend auf einem Image und einem definierten Befehl. Der Container wird an die

Konsole des Entwicklers angebunden. Gestartete CLI-Container können vom Entwickler wie gewöhnliche CLI-Befehle bedient werden.

```
app:
  name: demo
  commands:
    npm:
      image: node:10
      command: npm
```

Quellcode 5.2: Definitionsdatei für eine minimale Riptide Anwendung, mit einem Kommando, um npm auszuführen, basierend auf Node.js 10

Der Benutzer kann ein CLI-Kommando über den „cmd“ Befehl der Riptide CLI-Anwendung ausführen. Beispiel für ein Kommando mit dem Namen „npm“, welches die gleichnamige CLI-Anwendung zur Verwaltung von Node.js Paketen zur Verfügung stellt (siehe Quellcode 5.2): „riptide cmd npm install“. Dieser Befehl lässt sich vom Entwickler genau so benutzen, als würde dieser „npm install“ direkt ausführen. Lediglich der Präfix „riptide cmd“ muss verwendet werden.

Über optionale Integrationen für die Bash- und zsh-Shells (siehe *6.5 Integration der Bash und Zsh Shells*) lassen sich Riptide CLI-Kommandos auch ohne diesen Präfix ausführen. Der Entwickler muss sich dafür im Projektordner oder einem Unterordner davon befinden. Er kann dann im Beispiel „npm install“ direkt in der Konsole ausführen.

Um eine möglichst nahtlose Integration zu bieten, erhalten gestartete Kommando-Container Zugriff auf alle Umgebungsvariablen der aktuellen Konsole (bis auf einige Ausnahmen, die die Ausführung stören würden). Zusätzlich erhalten die Kommando-Container Zugriff auf den Quellcode des Projektes via Volume (vgl. *5.5.2*, Schlüssel „src“). Das Arbeitsverzeichnis des Containers entspricht dem Arbeitsverzeichnis des Entwicklers auf dem Host-System.

5.7.3 Zugriff auf zusätzliche Dienst-Ports

Für Dienste lassen sich zusätzliche TCP- und UDP-Ports bereitstellen, die vom Anwender über das Host-System erreichbar sein sollen. Diese lassen sich als Einträge in der Map

unter dem Schlüssel „additional_ports“ für Dienste konfigurieren (vgl. 5.5.4).

Jeder Eintrag besteht aus einer Kombination eines TCP- und UDP-Ports innerhalb des Dienst-Containers („container“) und der Angabe eines Ports, der auf dem Host-System benutzt werden soll („host_start“). Beim ersten Start des Dienstes sucht Riptide beginnend bei „host_start“ nach dem Ersten freien Port auf dem Hostsystem. Frei bedeutet, dass keine Anwendung aktuell auf dem TCP-Port horcht und der Port nicht bereits von einem anderen Riptide Dienst reserviert wurde. Dieser freie Port wird dann in einer zentralen Konfigurationsdatei für diesen Dienst reserviert.

Riptide teilt der Container-Engine mit, die TCP- und UDP-Container-Ports an den gefundenen freien Port auf dem Hostsystem zu binden. Dieser freie Port wird für spätere Startvorgänge wiederverwendet. Im Ergebnis hat somit jeder Dienst eine eindeutige Port-Nummer über die sein TCP- bzw. UDP-Dienst erreichbar ist. Der Benutzer kann sich diese über die CLI-Schnittstelle anzeigen lassen.

Beispiel: Definieren mehrere Projekte MySQL-Datenbank Server, welche ihren Dienst auf einem TCP Port bereitstellen, so können Entwickler in ihrem MySQL-Verwaltungswerkzeug für jeden MySQL Dienst eine eigene Verbindung mit dem entsprechenden TCP Port angeben. Da sich dieser Port nicht ändert, müssen Entwickler den Dienst nur starten und können ihn dann über die konfigurierte Verbindung nutzen.

5.7.4 Konfigurationsdateien von Diensten

In der Map hinter dem Schlüssel „config“ lassen sich für Dienste Konfigurationsdateien definieren, welche laufenden Dienst-Containern als Volumes bereitgestellt werden sollen (vgl. 5.5.4).

Riptide sucht nach Konfigurationsdateien, die in „from“ angegeben sind. Die Pfad-Angaben dort sind relativ zu den Definitionsdateien, die benutzt wurden, um den Dienst zu laden. Nachdem das System die Konfigurationsdatei gefunden hat, wertet es den Inhalt der Datei aus. In der Datei können Schablonen, wie in Kapitel *5.6.1 Auswertung von Variablen* beschrieben, eingesetzt werden. Riptide wertet die Variablen der Schablone im Kontext des Dienstes aus und speichert das Ergebnis. Die ausgewertete Datei wird dann als Volume im Dienst-Container an dem in „to“ angegeben Pfad bereitgestellt.

Die Konfigurationsdateien erlauben es, Anwendern Konfigurationsdateien für Anwendungen bereitzustellen, dessen Inhalt abhängig von den Definitionsdateien ist, wie beispielsweise für die Definition von Datenbankverbindungen.

5.7.5 Logging

In der Map hinter dem Schlüssel „logging“ lassen sich für Dienste Einstellungen für das Logging definieren (vgl. 5.5.4). Der Zugriff auf Logging-Dateien ermöglicht es, Entwicklern Probleme in ihrer Anwendung zu identifizieren.

Die Schlüssel „stdout“ und „stderr“ lassen sich auf die booleschen Werte „wahr“ und „falsch“ setzen. Je nach Einstellung werden die entsprechenden Unix-Ausgabeströme der im Dienst-Container gestarteten Anwendung in Dateien umgeleitet. Diese Dateien werden in einem Logging-Ordner im Projektordner zur Verfügung gestellt.

Neben den Ausgabeströmen lassen sich beliebige andere Pfade im Container als Volume im Logging-Ordner bereitstellen (Einträge unter „paths“). Es lassen sich im Container Hilfs-Kommandos starten, deren Ausgabeströme ebenfalls als Dateien im Logging-Ordner bereitgestellt werden (Einträge unter „commands“).

5.8 Veröffentlichung des Systems

Der Quellcode des gesamten Riptide Systems und von Configcrunch ist als Reihe von Git Repositories auf GitHub veröffentlicht. Sämtliche Repositories sind zudem angehängt.

Insgesamt gibt es folgende Git-Repositories:

- Software-Pakete
  - riptide-lib (Riptide Bibliothek, `https://github.com/Parakoopa/riptide-lib`)
  - riptide-proxy (Proxy-Server, `https://github.com/Parakoopa/riptide-proxy`)
  - riptide-cli (CLI-Anwendung, `https://github.com/Parakoopa/riptide-cli`)

  - riptide-engine-docker (Docker Container-Backend, `https://github.com/Parakoopa/riptide-engine-docker`)
  - riptide-db-mysql (MySQL Datenbank-Treiber, `https://github.com/Parakoopa/riptide-db-mysql`)
  - configcrunch (inklusive Dokumentation, `https://github.com/Parakoopa/configcrunch`)
- Dokumentation von Riptide (`https://github.com/Parakoopa/riptide-docs`)
- Riptide Community Repository (siehe *5.6.3 Repositories*, `https://github.com/Parakoopa/riptide-repo`)
- Dockerfiles für Images, die im Riptide Community Repository eingesetzt werden (`https://github.com/Parakoopa/riptide-docker-images`)

Die Software-Projekte sind zudem über *Python Package Index* (PyPI, `https://pypi.org/`) veröffentlicht und lassen sich somit von Benutzern über *pip*, das Paketverwaltungs-Werkzeug für Python, installieren (`https://pypi.org/user/Parakoopa/`). Die Namen der Pakete entsprechen den Namen in der Auflistung oben. Weitere Informationen zum Erstellungsprozess der Pakete, die auf PyPI veröffentlicht werden, finden sich im Kapitel *7.1 Qualitätssicherung*.

Die Dokumentation ist zusätzlich im HTML-Format veröffentlicht (für die Anzeige in Webbrowsern), siehe Kapitel *5.9 Dokumentation*.

Das Riptide Community Repository ist auf einige Docker Images angewiesen, deren Dockerfiles im *riptide-docker-images* Repository liegen. Diese Images werden über den Docker Hub automatisiert gebaut und zur Verfügung gestellt (`https://hub.docker.com/u/riptidepy`).

5.9 Dokumentation

Das System wird ausgeliefert mit einer Dokumentation (englisch). Diese Dokumentation besteht aus vier Teilen.

Der erste Teil ist die Benutzerdokumentation für Entwickler („User Documentation“). Dieser Teil der Dokumentation beschreibt die Funktionen und die Installation von Riptide. Die Dokumentation ist als Anleitung konzipiert, welcher Entwickler schrittweise folgen können, um den Umgang mit Riptide zu lernen.

Der zweite Teil ist die Benutzerdokumentation für Systemadministratoren („Configuration Guide“). Dieser Teil beschreibt Konfigurations-Entitäten (vgl. 5.5), wie die Definitionssprache sich verhält (vgl. Kapitel 5.6) und enthält Anleitungen über die Erstellung von Projektdateien.

Der dritte Teil ist die Dokumentation der Entitäten aus dem Riptide Community Repository (vgl. *5.6.3 Repositories*).

Der vierte Teil ist die Moduldokumentation. Diese wird zum großen Teil direkt aus Kommentaren im Quellcode von Riptide generiert und gibt einen Einblick in die Implementation von Riptide. Hier finden sich auch Hinweise zur Erstellung neuer Engine-Backends und Datenbank-Treiber.

Die Dokumentation ist angehängt und online verfügbar unter `https://riptide-docs.readthedocs.io/`.

Es wird zusätzlich eine Dokumentation für Configcrunch (vgl. *5.6.4 Bibliothek zur Verarbeitung der Sprache (Configcrunch)*) bereitgestellt. Diese Dokumentation ist für Entwickler geschrieben, die Configcrunch einsetzen möchten und erklärt die Funktionsweise der Definitionssprache und die Implementation eigener Konfigurations-Entitäten. Die Dokumentation ist angehängt und online verfügbar unter `https://configcrunch.readthedocs.io/`.

6 Implementation

In diesem Kapitel werden einige wichtige Implementationsdetails des zuvor konzipierten Systems (Riptide) beschrieben.

6.1 Implementation der Riptide Bibliothek

Die Riptide Bibliothek ist der Systemkern aller Anwendungen und Schnittstellen (CLI, Proxy, Container-Backend, Datenbank-Treiber). Die Bibliothek besteht aus vier Python Paketen:

config
: Das „config“-Paket enthält die Konfigurations-Entitäten als Unterklasse von *„YamlConfigDocument“*, der Basis-Klasse von Configcrunch (vgl. *5.6.4 Bibliothek zur Verarbeitung der Sprache (Configcrunch)*). Es enthält zudem ein Modul zum Laden von Konfiguration und Projekten (*„config.loader“*) und ein Modul zur Verwaltung von Dateien und Speicherorten (*„config.files“*). Für die Verarbeitung von Ports (vgl. 5.7.3), Logging (vgl. 5.7.5) und Konfigurationsdateien für Dienste (vgl. 5.7.4) stehen im Paket *„config.services“* Module bereit.

engine
: Das „engine“-Paket enthält die Container-Backend-Schnittstelle als abstrakte Klasse *„AbstractEngine“* im Modul *„engine.abstract“* (anders als in anderen Programmiersprachen gibt es keine „interface“-Definitionen in Python).

  Zusätzlich enthält das Paket Module, die in der Kommunikation zwischen Container-Backend und CLI-Anwendung oder Proxy-Server eingesetzt werden. Besonders hervorzuheben ist dabei das Modul *„engine.results“*, welches die Implementierung einer Nachrichten-Warteschlange (Message Queue) beinhaltet. Diese wird

im Start-Vorgang von Diensten eingesetzt, siehe *6.3 Start-Vorgang von Dienst-Containern*.

db

Dieses Paket enthält die Datenbank-Treiber-Schnittstelle und Module zur Verwaltung von Datenbank-Umgebungen.

lib

Dieses Paket enthält kleinere Hilfsmodule und Module zur Kompatibilität mit verschiedenen Betriebssystemen. Insbesondere die Implementation des Python Paketes „pty“ (`https://docs.python.org/3/library/pty.html`) für Windows ist hierbei hervorzuheben.

„pty.spawn“ erlaubt es CLI-Anwendungen aus einer Python-Anwendung heraus zu starten und direkt mit der Konsole des Benutzers zu verbinden. Nach Beendigung dieser Anwendung wird zur Python Anwendung zurückgekehrt. Diese Funktionalität ist essentiell für die Ausführung der Kommando-Entitäten (vgl. *5.7.2 Kommandos von Anwendungen*). Unter Windows steht dieses Paket nicht zur Verfügung, da das Windows-Terminal sich grundsätzlich anders verhält als Unix-kompatible Terminals. Das Modul *„lib.cross_platform.cppty“* beinhaltet eine Windows-Implementierung, die auf der Bibliothek „winpty“ (`https://github.com/rprichard/winpty`) basiert. Aufgrund der Komplexität kommt diese Implementierung noch mit einigen offenen Problemfällen daher, sie ist allerdings grundsätzlich funktionsfähig.

6.2 Implementation des Docker Container-Backends

Für die Nutzung des Docker Container-Backends ist die Installation von Docker auf dem Hostsystem des Anwenders notwendig. Unter Mac und Windows ist die Installation von „Docker for Windows“ bzw. „Docker for Mac“ notwendig.

Für die Implementation des Docker Backends wird die für Python bereitgestellte Docker SDK für Python verwendet (`https://docker-py.readthedocs.io/en/stable/`).

Dienst-Container werden über diese SDK gestartet und gestoppt. Auch das Container-Netzwerk wird über diese SDK gestartet. Die Docker SDK bietet keine Möglichkeit interaktive, mit der Konsole des Benutzers verbundene, Container zu starten. Aus diesem Grund ist die Docker SDK für die Ausführung der interaktiven Kommando-Objekte (vgl. *5.5.5 Kommandos*) ungeeignet. Es wird an dieser Stelle stattdessen ein interaktiver Container über die Docker CLI gestartet („docker run -it ...“). Die Docker CLI wird standardmäßig mit Docker unter Linux, sowie „Docker for Windows“ und „Docker for Mac“ ausgeliefert.

Um die Ausführung von Containern über die Docker CLI und die Docker SDK zu vereinheitlichen, gibt es die zentrale Klasse „ContainerBuilder“. Diese Klasse erlaubt es über das Erbauer-Muster (Gamma u. a., 2011, S. 119ff.) Container-Objekte zu erstellen. Über Methoden lassen sich Eigenschaften wie Volumes, Port-Weiterleitungen und das gewünschte Image für den Container festlegen. Die Klasse bietet zusätzliche Methoden, um gängige Container-Eigenschaften basierend auf Kommando- oder Dienst-Entitäten zu setzen. Aus dem „ContainerBuilder“ lassen sich eine Kommandozeile für die Docker CLI und Argumente für die Docker SDK generieren.

Das Container-Backend startet alle Container mit einem eigenen Entrypoint-Script, nachfolgend „Riptide Entrypoint“ genannt (für Kommando (CMD) und ENTRYPOINT siehe *2.2.2 Container*).

Der „Riptide Entrypoint“ ist notwendig, um zu Beginn eines Container-Starts einen Benutzer und eine Gruppe mit der Benutzer- und Gruppennummer des Host-System-Benutzers im Container anzulegen. Am Ende führt der „Riptide Entrypoint“ den Original-Entrypoint mit diesem Benutzer aus, es sei denn, in der Dienst-Entität ist „run_as_current_user“ auf „falsch“ gesetzt, vgl. *5.5.4 Dienste*. Über diesen Mechanismus werden mögliche Berechtigungsprobleme mit Containern vermieden (vgl. Kapitel 2.2.2). Der „Riptide Entrypoint“ startet zudem im Container zusätzliche Hintergrundprozesse für das Logging (vgl. *5.7.5 Logging*).

Der „Riptide Entrypoint“ wird vor das Entrypoint-Script des Images und vor das gewünschte Kommando für den Container gestellt, sodass dieser „Riptide Entrypoint“ das eigentliche Entrypoint-Script startet, welcher dann das Kommando des Containers startet.

Docker sieht es nicht vor, vor dem Entrypoint des Images ein weiteres Entrypoint-Script zu definieren. Es kann standardmäßig nur einen Entrypoint und ein Kommando geben

Tabelle 6.1: Ausführung des Container-Befehls von Docker, je nach Definition von CMD und ENTRYPOINT (nach Docker (2019a)).

	Kein ENTRYPOINT	„Shell format“: ENTRYPOINT exec_entry p1_entry	„Exec format“: ENTRYPOINT [“exec_entry”, “p1_entry”]
Kein CMD	*Fehler, nicht erlaubt*	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
„Exec format“: CMD [“p1_cmd”, “p2_cmd”]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry p1_cmd p2_cmd
„Shell format“: CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

(Docker, 2019a). Um dieses Problem zu lösen, wird der Entrypoint des zu startenden Containers auf den „Riptide Entrypoint“ gesetzt. Der Wert des Kommandos (CMD) bleibt auf dem Original-Kommando aus dem Image gesetzt. Das Container Backend setzt eine Umgebungsvariable mit dem Wert des ursprünglichen Entrypoints. Je nach Definition des Entrypoints startet Docker den Container so, dass der Entrypoint ohne weitere Argumente gestartet wird („Shell form“) oder mit dem Kommando als Kette von Argumenten („Exec form“, vgl. Tabelle 6.1). Abhängig davon, wie der ursprüngliche Entrypoint definiert war, können nun verschiedene Fälle eintreten:

- Es ist kein Entrypoint im Image definiert: Der „Riptide Entrypoint“ führt das Original-Kommando aus.
- Es ist ein Entrypoint, in „Shell form“ im Image definiert: Der „Riptide Entrypoint“ führt den Original-Entrypoint aus und übergibt diesem nicht das Original-Kommando. Das Original-Kommando wird nicht ausgeführt (vgl. Tabelle 6.1)
- Es ist ein Entrypoint, in „Exec form“ im Image definiert: Der „Riptide Entrypoint“ führt den Original-Entrypoint aus und übergibt diesem das Original-Kommando. Das Original-Entrypoint führt das Original-Kommando aus.

Über diesen Mechanismus wird vor den definierten Entrypoint ein zusätzlicher Entrypoint gestellt, der sich entsprechend dem Standardverhalten von Docker verhält.

6.3 Start-Vorgang von Dienst-Containern

Projekte bestehen aus der Definition von Diensten, welche vom System in Form von Containern gestartet werden können. Dieser Start-Vorgang kann aktiv über die CLI-Anwendung vom Benutzer durchgeführt werden, oder durch den Aufruf eines gestoppten Dienstes via der Proxy-Server-Komponente.

Der Start-Vorgang besteht aus einer Schnittstelle auf Seiten der CLI- bzw. Proxy Anwendung, welche über die Riptide Bibliothek bereitgestellt wird (siehe Abbildung 5.4), und der Implementierung des Start-Vorgangs über die Container-Backend-Schnittstelle (siehe Abbildung 5.5). Aufrufer und Container-Backend kommunizieren über Nachrichten („Status-Updates“) über den aktuellen Start-Status. Der Start-Vorgang wird parallel für jeden Dienst ausgeführt und der Benutzer sieht die Status-Updates in Form von Fortschrittsbalken für jeden Dienst (siehe Abbildung 5.6).

Um die paralelle Ausführung von mehreren Start-Vorgängen zu erlauben wird Python Asyncio eingesetzt. Eine Python-Funktion kann als „async“ definiert werden (*asynchrone Funktion*). Innerhalb einer asynchronen Funktion kann vor einer Anweisung das Schlüsselwort „await“ eingesetzt werden, um die Ausführung der Funktion zu unterbrechen und zu warten, bis die Anweisung abgeschlossen ist. Dem Asyncio Event-Loop wird dieser Warte-Zustand mitgeteilt und er fährt mit der Ausführung der nächsten asynchronen Aktion fort. So lässt sich einfache asynchrone Programmierung umsetzen. Es ist allerdings noch nicht möglich, gleichzeitig mehrere Code-Stellen auszuführen, hierfür muss Asyncio mit Threads kombiniert werden.

```
# Start all services
queues = {}
loop = asyncio.get_event_loop()
for service_name in services:
    # Create queue and add to queues
    queue = ResultQueue()
    queues[queue] = service_name
    if service_name in project["app"]["services"]:
        # Run start task
        loop.run_in_executor(
            None,
            service.start,

```

```
14                 project["name"],
15                 project["app"]["services"][service_name],
16                 self.client,
17                 queue
18             )
19         else:
20             # Services not found :(
21             queue.end_with_error(ResultError("Service not found."))
22
23     return MultiResultQueue(queues)
```

Quellcode 6.1: Ausschnitt aus der Start-Funktion für Projekte des Docker Container-Backends („start_project“)

Das Container-Backend teilt Asyncio mit, die Dienst-Start-Funktion in einem Executor (Thread) auszuführen. Asyncio startet dann einen Thread für die Ausführung dieser Funktion, allerdings nur bis zu einer Maximalanzahl von Threads. Ist die Maximalanzahl erreicht, so wird mit der Ausführung gewartet, bis ein Thread frei wird. Quellcode 6.1 zeigt, wie die Event-Loop geladen wird (Zeile 3), um in ihr über einen Executor für jeden Dienst (Zeile 4) die Start-Funktion auszuführen (Zeilen 10 - 18, *„service.start“* ist die Referenz zur Start-Funktion).

Um den Status abzurufen, wird mit den Klassen *„ResultQueue“* und *„ResultMultiQueue“* aus dem Paket *„riptide.engine.results“* gearbeitet. *„ResultQueue“* implementiert eine Message Queue, welche je Dienst eingesetzt wird, um den aktuellen Status des Dienstes als Nachricht zu speichern. Diese Message Queue basiert auf der Bibliothek „janus“ (`https://github.com/aio-libs/janus`), welche eine Warteschlangen-Implementation bereitstellt, die sowohl über Asyncio „await“ gelesen und geschrieben werden kann, als auch threadsicher über mehrere Threads. Die Dienst-Start-Funktion schreibt in diese Queue Status-Updates und kann sie als beendet markieren, um zu signalisieren, dass der Start-Vorgang abgeschlossen wurde. Dabei lässt sich unterscheiden, ob der Start-Vorgang erfolgreich war oder nicht. Quellcode 6.1 zeigt die Erstellung dieser Queue pro Dienst (Zeile 6) und die Übergabe an die Start-Funktion (Zeile 17). Zudem wird das Beenden einer *„ResultQueue“* mit Fehlerzustand in Zeile 21 gezeigt.

„ResultMultiQueue“ kombiniert mehrere *„ResultQueue“*s. *„ResultMultiQueue“* nimmt als Konstruktor eine Zuordnung von *„ResultQueue“*s zu Namen an und erlaubt die Iteration über Asyncio. Für jedes Status-Update in einer der *„ResultQueue“*s gibt die asynchrone Iteration über *„ResultMultiQueue“* eine Nachricht zurück. Diese Nachrichten enthalten

den Namen der betroffenen Queue, die Status-Nachricht der Queue und die Information, ob diese Queue beendet wurde. In Quellcode 6.1 wird ein solches Objekt in Zeile 23 als Ergebnis der „start_project“-Funktion zurückgegeben, basierend auf den Queues, welche in Zeile 2 als Python Dictionary erzeugt und in Zeile 7 befüllt wurden.

```
1 async def start_project(project, services, engine):
2     """Eingabe ist das Projekt-Objekt, das Container-Backend und die Liste
      der Namen der zu startenden Dienste."""
3     async for service_name, status, finished \
4       in engine.start_project(project, services):
5         print("Status von Service %s ist: %s, ist der Prozess
      abgeschlossen?: %r" % (service_name, status, finished))
```

Quellcode 6.2: Aufruf der Schnittstelle zum Starten von Projekten, anhand eines einfachen Beispiels)

Der Start-Vorgang auf Seiten der CLI und des Proxies erhält die *„ResultMultiQueue“* (vgl. Abbildungen 5.4 und 5.5, „Liste von überwachbaren Status-Objekten“). Um dem Benutzer den Fortschritt anzuzeigen, wird via Asyncio-Feature „async for“ über die *„ResultMultiQueue“* iteriert. Quellcode 6.2 zeigt somit den gesamten Start-Vorgang aus Sicht von CLI und Proxy.

6.4 Implementationsdetails der Proxy-Server-Komponente

Dieser Abschnitt beschreibt einige Details zu den inneren Abläufen des Proxy-Servers (vgl. Kapitel 5.3.5). Der Proxy-Server nutzt als Grundlage das Python Web-Framework Tornado (`https://www.tornadoweb.org/en/stable/`).

SSL-Zertifikate

Für den Zugriff über HTTPS wird die Bibliothek certauth (`https://pypi.org/project/certauth/`) eingesetzt. Diese Bibliothek generiert ein Zertifikat für eine Zertifizierungsstelle, welche die Entwickler in ihr System importieren können, um diese als vertrauenswürdig einzustufen. Diese Zertifizierungsstelle wird genutzt, um SSL/TLS-Zertifikate für die Hostnamen des Proxy-Servers zu generieren.

Rechte und Berechtigungen

Unter Linux ist es normalen Benutzern nicht möglich auf TCP Ports unter 1024 zu lauschen. Dies kann nur root. Alternativ ist für den Prozess das zusätzliche Recht („Capability“) „CAP_NET_BIND_SERVICE“ erforderlich (Linux, 2019). Nur root hat die Autorisierung solche Rechte zu verteilen.

Da die Konfiguration von Riptide abhängig vom Benutzer ist und Container im Normalfall mit der Benutzer- und Gruppennummer dieses Benutzers gestartet werden sollen, ist es nicht sinnvoll, den Proxy-Server vollständig als root auszuführen.

Stattdessen lässt sich der Proxy-Server als root unter Angabe eines Benutzernamens starten. Der Proxy-Server gibt sich selbst das Recht „CAP_NET_BIND_SERVICE“ und ändert danach die Benutzer- und Gruppennummer seines Prozesses sowie einige weitere Einstellungen so, dass diese den Einstellungen des übergebenen Benutzers entsprechenden. Damit fällt der Proxy-Server zurück auf die Berechtigungen eines normalen Benutzers, aber plus dem Recht auf TCP Ports unter 1024 zu lauschen. Die Rechtevergabe erfolgt über die Python Bibliothek python-prctl (`https://github.com/seveas/python-prctl`).

Routing von Traffic

Der Proxy-Server enthält ein Modul zum Starten des Tornado HTTP Servers. Tornado erlaubt es eine Reihe sogenannter *Handler* einzusetzen. Die Handler befinden sich im Python Paket „riptide_proxy.server“ Ein Handler kümmert sich entweder um die Abwicklung einer HTTP- oder einer WebSocket Anfrage. Welcher Handler eingesetzt wird, bestimmen *Matcher*. Es gibt drei Paare von Matchern und Handlern:

- **HTTP-Anfragen.** Der Handler nimmt Traffic an, um diesen an Container weiterzuleiten und Status-Seiten anzuzeigen, siehe *Ablaufsicht* in Kapitel *5.3.5 Proxy-Server*.
- **WebSocket-Anfragen an eine spezielle Route (___riptide_proxy_ws).** Der Handler nimmt WebSocket Verbindungen für den Auto-Start Prozess an.
- **Alle anderen WebSocket-Anfragen.**: Der Handler stellt eine WebSocket Verbindung zum passenden Container her und leitet Nachrichten zwischen Container und Benutzer weiter. Er fungiert als WebSocket Reverse Proxy. Hostnamen und

angefragter Pfad werden beim Aufbau der Proxy-Verbindung entsprechend dem Server im Container mitgeteilt.

Die beiden Handler, die die Zuordnung zu Projekten und Diensten durchführen müssen, nutzen dafür ein gemeinsames Modul „riptide_proxy.project_loader“.

6.5 Integration der Bash und Zsh Shells

Wie im Kapitel *5.7.2 Kommandos von Anwendungen* beschrieben, erlaubt Riptide die Ausführung von definierten Kommando-Objekten.

Diese lassen sich in einer Konsole einerseits über den „cmd“-Befehl der Riptide CLI ausführen (Beispiel: „riptide cmd npm install“) oder direkt (Beispiel: „npm install“).

Für die direkte Ausführung ist die Aktivierung einer Shell-Integration notwendig, die in diesem Abschnitt erläutert wird. Die Shell ist eine Anwendung, die in einer Konsole ausgeführt wird. Sie wird zur Ausführung anderer Befehle (interaktiv) oder von Scripten (Shell-Scripte) verwendet. Unter vielen Linux-Distributionen und macOS wird standardmäßig die *Bash*-Shell als interaktive Shell verwendet. Eine andere Shell ist *zsh*. Für beide stellt das Riptide System Integrationen zur Verfügung. Für Windows-Shells stellt Riptide keine Integration bereit.

Um die direkte Ausführung von Riptide Kommandos zu erlauben, speichert die Riptide CLI beim Einlesen der Projektkonfiguration ausführbare Dateien in einem Unterordner des Projektes ab („_riptide/bin“). Bei diesen Scripten handelt es sich jeweils um Python-Code, der Riptide anweist, das jeweilige Kommando-Objekt interaktiv auszuführen. Im oben genannten Beispiel legt die Riptide CLI dort ein Script namens „npm“ ab.

Der Anwender kann nun durch Ausführung des Scripts den Befehl ausführen (Beispiel, falls sich der Anwender im Wurzelverzeichnis des Projektes befindet: „_riptide/bin/npm install“). Damit unter Unix-ähnlichen Betriebssystemen bei der Ausführung einer ausführbaren Datei keine Angabe eines Pfades notwendig ist und damit sich solche Dateien auf dem gesamten System ausführen lassen, müssen Verzeichnisse mit ausführbaren Dateien Teil der Umgebungsvariable „*PATH*“ sein. Die „PATH“-Variable enthält eine Liste von Pfaden, die zu diesem Zweck durchsucht werden.

Riptide stellt über Integrationsscripte Mechanismen bereit, die die „PATH“-Variable je nach aktuellem Arbeitsverzeichnis so verändern, dass der zum Projekt gehörende „_riptide/bin“-Ordner in den „PATH“ eingetragen wird. Um diese Integration zu aktivieren, muss der Anwender von Riptide ein Script in die Startkonfiguration seiner Bash oder zsh-Shell laden.

Die Integrationsscripte untersuchen beim Wechsel des aktuellen Verzeichnisses, ob das aktuelle Verzeichnis oder ein anderes in der Dateisystemstruktur aufsteigendes Verzeichnis, ein Riptide-Projekt ist. Falls ja, wird der Pfad zum „_riptide/bin“-Ordner in den „PATH“ eingetragen. Falls nein oder falls das Projekt gewechselt wird, wird der alte Pfad wieder ausgetragen.

Über diesen Mechanismus ist es dem Entwickler möglich, in jedem Projekt alle definierten Riptide-Kommandos direkt über seine Shell auszuführen.

7 Evaluation

In diesem Kapitel wird die Qualität des konzipierten und implementierten Systems evaluiert. Dafür werden zunächst Maßnahmen zur Qualitätssicherung aufgelistet und es wird überprüft, ob sich das System mit gängigen Webanwendungen und Frameworks einsetzen lässt. Danach wird eine Evaluationsphase des Systems in der Tudock GmbH ausgewertet. Zum Schluss wird aufgrund dieser Ergebnisse beurteilt, ob die definierten Anforderungen erfüllt werden.

7.1 Qualitätssicherung

Wie im Kapitel *5.8 Veröffentlichung des Systems* beschrieben, sind sämtliche Komponenten des Riptide Systems und von Configcrunch auf Github veröffentlicht. Beim Senden neuer Commits an die Git Repositories werden auf einem zentralen Server Tests durchgeführt und ggf. Software veröffentlicht. Der Status dieser Prozesse wird auf Github an den einzelnen Commits vermerkt. Bei diesen Prozessen handelt es sich um *Build-Pipelines* nach Continuous Integration (CI) und Continuous Delivery (CD) Prinzipien.

Die automatisierten Build-Pipelines sollen sicherstellen, dass eingesendete Code-Änderungen den festgelegten Qualitätsstandards entsprechen und mit dem neuen Code das System weiterhin vollständig lauffähig ist. Dies ist der CI-Aspekt der Build-Pipelines. Ergänzt werden die automatischen Tests durch manuelle Tests von Entwicklern und durch Rückmeldungen von Benutzern, welche auf Github über *Issues* Probleme und Wünsche zum System äußern können.

Außerdem erkennt der eingesetzte Dienst *Read the Docs* (`https://riptide-docs.readthedocs.io/en/latest/`) Änderungen an Dokumentationen automatisch bei Eingang neuer Commits. Der Dienst stellt eine HTML-Version der Dokumentationen bereit (siehe *5.9 Dokumentation*). Werden für die Software-Pakete Änderungen auf einem Git-Zweig *release* durchgeführt, so werden diese Änderungen als neue Version über

die Plattform PyPI zur Verfügung gestellt (siehe *5.8 Veröffentlichung des Systems*). Bei diesen Prozessen handelt es sich um die CD-Aspekte der Build-Pipelines.

Alle Pipelines der Software-Projekte bestehen mindestens aus dem Bauprozess der Python-Pakete und ggf. der Veröffentlichung auf PyPI. Die Veröffentlichung auf PyPI wird nur durchgeführt, wenn alle anderen Schritte der jeweiligen Pipeline erfolgreich durchlaufen wurden.

Für einige wichtige Module innerhalb der Riptide Bibliothek und des Docker Container-Backends gibt es zudem automatische Unit-Tests, welche einzelne Module der Software-Projekte, voneinander unabhängig auf ihre korrekte Funktion testen.

Für die Riptide Bibliothek sind zudem Integrationstests implementiert. Solche Tests kontrollieren einzelne modulübergreifende Teile des Systems. Die Riptide Integrationstests überprüfen, dass die Riptide Bibliothek korrekt Projekte laden kann und die bestehenden Container-Backends korrekt verschiedene Aspekte von Dienst-Containern abbilden können. Über die Integrationstests der Riptide Bibliothek werden sowohl die Riptide Bibliothek als auch alle installierten Container-Backends auf ihre Funktion getestet.

Für Configcrunch gibt es Akzeptanztests. Diese Tests überprüfen, ob Configcrunch YAML-Dateien korrekt einliest und wieder ausgibt, indem die Ausgaben von Configcrunch gegen erwartete Werte überprüft werden. Die Dokumentation von Configcrunch enthält zudem Doctests (`https://docs.python.org/3/library/doctest.html`). Doctests sind ein Sprach-Feature von Python und erlauben es, Code, der in Dokumentationen eingesetzt wird, zu testen. Dadurch kann Beispiel-Code in Dokumentationen nicht veralten, da in diesem Fall die Doctests nicht mehr funktionieren würden.

Für die Dokumentation von Riptide werden Funktionen des eingesetzten Dokumentations-Frameworks Sphinx (`http://www.sphinx-doc.org`) eingesetzt, um Teile der Dokumentation direkt aus den Kommentaren im Quellcode auszulesen und als Teil der HTML-Dokumentation zu veröffentlichen. Dies wird insbesondere für die Schema-Beschreibung der Konfigurations-Entitäten eingesetzt. Durch diesen Mechanismus wird sichergestellt, dass Entwickler, die das Schema ändern, auch direkt die Dokumentation aktualisieren, wodurch die Wartbarkeit und damit die Qualität der Dokumentation steigt.

Die Dokumentationen für Entitäten im Riptide Community Repository werden bei neuen Commits in die Riptide Dokumentation kopiert. So stehen diese Dokumentationen zusam-

men mit der restlichen Dokumentation zur Verfügung. Für das Community-Repository sind zusätzlich noch automatische Tests vorgesehen, welche die Entitäten im Repository validieren - diese sind allerdings noch nicht implementiert. Die Docker Images für das Riptide Community Repository werden über eine separate Build-Pipeline bei Docker Hub gebaut und veröffentlicht (siehe *5.8 Veröffentlichung des Systems*).

Die Proxy-Server-Komponente und die CLI-Anwendung werden aktuell über manuelle Tests auf ihre Funktion getestet, eine Ergänzung durch Akzeptanz-Tests ist vorgesehen. Weiterhin ist es auch hier möglich, Unit-Tests für einzelne Module zu implementieren.

7.2 Kompatibilität mit verschiedenen Webanwendungen und Frameworks

Das Riptide System wurde konzipiert, um die Definition von Projekten für beliebige Webanwendungen und Frameworks zu erlauben.

```
1 project:
2   name: magento-demo
3   src: src
4   app:
5     $ref: /app/magento2/ce/2.3
```

Quellcode 7.1: Riptide Projektdatei für ein Magento 2 Projekt, unter Referenz auf das Riptide Community Repository

Um das System zu evaluieren, wurden Projektdateien für verschiedene Anwendungen und Frameworks erstellt. Dazu gehören unter anderem PHP-basierte Anwendungen mit Nginx oder Apache Webserver sowie Node.js und Python basierte Webserver.

Es wurde zudem der Einsatz von Riptide mit der E-Commerce Plattform Magento getestet. Riptide Magento Projekte bestehen aus dem PHP Process Manager (FPM, `https://www.php.net/manual/de/install.fpm.php`) und Nginx zur Ausführung der Magento-Anwendung, einer MySQL Datenbank (`https://www.mysql.com/`

de/) und einem Redis Server (https://redis.io/) zur Speicherung der Sitzungsdaten und des Anwendungs-Caches. Weiterhin besteht die Umgebung aus einem Varnish Caching-Server (https://varnish-cache.org/), welcher von der Anwendung generierte HTML-Seiten zwischenspeichert, einer RabbitMQ Instanz (https://www.rabbitmq.com/), welche von Magento zur Verwaltung von Message Queues eingesetzt wird, und einem SMTP-Server mit Weboberfläche (Mailhog, https://github.com/mailhog/MailHog), welcher von Magento generierte Emails abfängt und dem Entwickler zur Verfügung stellt. All diese Anwendungen werden von Riptide als Dienste in einem Magento Projekt gestartet. Riptide Magento Projekte werden weiterhin mit einer Reihe Kommando-Objekte ausgeliefert, welche gängige CLI Werkzeuge, wie den PHP Paketmanager Composer (https://getcomposer.org/) zur Verfügung stellen.

Entwickler-Teams können diese getestete Magento Umgebung über das Riptide Community Repository (vgl. Kapitel 5.6.3) beziehen, indem sie eine Projektdatei, wie in Quellcode 7.1 gezeigt, anlegen. Es wurden weitere Dienste, Kommandos und komplette Anwendungsumgebungen getestet und als vollständig funktionsfähig befunden. Dazu gehören einfache Node.js Anwendungen und das Python Dokumentations-Framework Sphinx, welches auch zur Generierung der Riptide und Configcrunch Dokumentationen genutzt wurde. Während der Tests konnten fehlende Funktionen oder Befehle durch Ergänzung der Entitäts-Definitionen korrigiert werden.

Die Verwendung von Riptide mit Magento und einigen weiteren PHP oder Node.js basierten Anwendungen wurde mit einer Reihe Benutzern in der Tudock GmbH getestet (siehe Kapitel 7.3).

Alle getesteten Entitäten werden über das Riptide Community Repository angeboten. Für die Verwendung einiger Anwendungen und Frameworks befinden sich in der angehängten Riptide Dokumentation Anleitungen (Sektion „Configuration Guide“ → „Examples“).

7.3 Einsatz des Systems in der Tudock GmbH

Das Riptide System wurde in der Tudock GmbH evaluiert. Die Tudock GmbH ist eine Agentur, die primär E-Commerce-Projekte auf Basis der E-Commerce-Plattform Magento entwickelt und betreut. Das Entwickler-Team bestand im Zeitraum der Evaluation aus

zwölf Entwicklern, die mindestens eine Woche während des Evaluationszeitraums anwesend waren. Von diesen Entwicklern sind zwei neu dem Team beigetreten. Es gibt im Team keine Mitglieder, welche explizit die Rolle eines Systemadministrators nach Kapitel 3.1.2 einnehmen, diese Aufgaben werden im bestehenden Entwickler-Team verteilt.

Der Zeitraum der Evaluation umfasste den 20. März bis 7. Mai 2019.

Riptide wurde sechs Entwicklern dieses Teams zur Verfügung gestellt. Die Konfigurations-Entitäten für die Arbeit im Unternehmen fertigte der Autor an. Den Testpersonen wurde freigestellt, auch oder stattdessen eigene Projekte zu konfigurieren. Die angehängte Dokumentation für das Riptide System wurde zur Verfügung gestellt. Den Umfang der Nutzung von Riptide konnten die beteiligten Entwickler selbst festlegen. Die Evaluationsphase begann mit einer gemeinsamen Einführung des Systems und eine Führung durch den Installationsprozess.

Es wurde entschieden, Riptide als Teil der Evaluationsphase für mindestens je ein Magento 1.9 Projekt und ein Magento 2.3 Projekt einzusetzen. Es wurden Riptide Definitionen für mindestens drei Magento Projekte angelegt, fünf Entwickler haben aktiv Riptide zur Entwicklung von Magento Projekten eingesetzt (vgl. Auswertung des Fragebogens in Kapitel 7.4). Details zum Aufbau einer Magento 2 Riptide Anwendung sind im Kapitel 7.2 beschrieben.

Außerdem wurde Riptide im Rahmen eines Modernisierungsprojekts einer PHP-basierten Webanwendung eingesetzt. Diese Webanwendung ist eine Eigenentwicklung der Tudock GmbH. Die alte Version der Anwendung stellte Backend-Funktionen über PHP, sowie Frontend-Funktionen über eine JavaScript-basierte Weboberfläche zur Verfügung. Die neue Version soll Backend und Frontend in zwei separate Projekte aufteilen, das Backend soll dabei möglichst unberührt bleiben. Ziel ist es, das Frontend durch eine losgelöste Anwendung zu ersetzen, welche über eine API-Schnittstelle mit dem Backend kommuniziert und auf modernen JavaScript Technologien basiert. Es wurden Riptide Projekte sowohl für das PHP-basierte Backend, als auch für das neue Frontend geschaffen. Das neue Frontend liefert den Frontend-Code über einen eigenständigen Node.js Webserver aus. Die Modernisierung der Anwendung war zum Ende des Evaluationszeitraums nicht abgeschlossen. An diesem Projekt arbeiteten im Zeitraum der Evaluation zwei Entwickler. Beide Entwickler bekamen Riptide zur Verfügung gestellt. Der zweite Entwickler trat dem Projekt erst nach der Hälfte des Evaluationszeitraums bei.

Als Ergebnis der Evaluation im Unternehmen wurde ein Fragebogen konzipiert und ausgewertet. Die Konzeption und Auswertung ist im Kapitel 7.4 erläutert.

7.4 Messung und Auswertung: Umsetzung der Anforderungen

Im Kapitel *3.2 Anforderungen an Entwicklungsinstanzen für Webanwendungen* wurden Anforderungen für das System definiert. Nachfolgend wird evaluiert, ob das Riptide System diese Anforderungen erfüllt.

7.4.1 Definition der Qualitätsmerkmale

Um das System zu bewerten, werden die Anforderungen zunächst Qualitätsmerkmalen nach ISO 25010/2011 zugeordnet (Estdale und Georgiadou, 2018). Der Standard beschreibt Qualitätsmerkmale für Software. Die Qualität von Riptide lässt sich anhand dieser Merkmale über objektive und subjektive Kriterien messen.

Für die Zuordnung der Anforderungen wird das Gesamtsystem betrachtet, inklusive Merkmale der konzipierten Definitionssprache. Aus diesem Grund wurden beispielsweise A2.3 und A2.5 dem Merkmal Wartbarkeit zugeordnet. Einige Anforderungen wurden mehreren Merkmalen zugeordnet, da sie auf mehrere Punkte zutreffen. Die Merkmale *Compatibility* und *Portability* wurden zur Vereinfachung nachfolgend zusammengefasst. Es wurde zusätzlich die Qualität der Dokumentation als Merkmal aufgenommen.

Functional Suitability (Funktionalität)

Funktionelle Vollständigkeit und Korrektheit

A1.2 (100), A1.3 (80), A1.4 (80), A1.5 (40), A1.6 (60), A1.7 (70), A1.8 (40)

Performance Efficiency (Effizienz)

Verhältnis des Leistungsniveaus zum Einsatz der aufgewandten Ressourcen

A1.1 (100)

Compatibility (Kompatibilität) und Portability (Übertragbarkeit)

Kompatibilität mit Fremdsystemen und Übertragbarkeit in andere Umgebungen

A1.9 (50), A1.10 (80), A1.11 (80), A2.2 (100), A2.7 (60)

Usability (Benutzbarkeit)

Beurteilung der Benutzererfahrung und Einschätzung des Aufwandes der zur Benutzung aufgebracht werden muss

A1.3 (90), A1.4 (90), A1.5 (90), A1.6 (100), A1.7 (100), A1.8 (100), A2.4 (100)

Realiability (Verlässlichkeit)

Beurteilung darüber, wie sehr das System ein gewisses Leistungsniveau aufrecht erhalten kann

A2.5 (100), A2.6 (80)

Security (Sicherheit)

Schutz des Systems gegen Angriffe, die Datendiebstahl oder Manipulation erlauben

Maintainability (Wartbarkeit)

Einschätzung der Lesbarkeit und des Aufbau des Codes in Blick auf den Aufwand der zur Änderung des Codes erbracht werden muss

A2.3 (100), A2.5 (80), A2.6 (60)

Qualität der Dokumentation

A2.1 (100)

Die Zahlen in Klammern geben eine Gewichtung der Anforderungen für die Auswertung der subjektiven Metriken an. Höhere Werte stellen eine höhere Gewichtung dar. Für die Bestimmung der Werte wurde vom Autor subjektiv eingeschätzt, inwiefern einzelne Anforderungen zur Erfüllung eines Qualitätsmerkmales beitragen.

Systeminterne Aspekte der Merkmale, wie Wartbarkeit des Codes, sind durch keine Anforderungen explizit abgedeckt. Es handelt sich dabei allerdings ebenfalls um wichtige Faktoren für die Gesamtqualität des Systems, welche für die Bewertung in Betracht gezogen werden müssen. Für die Messung solcher Qualitätsmerkmale wäre eine Auswertung von Bug-Reports oder Code-Reviews möglich.

Tabelle 7.1: Objektive Metriken der Riptide Komponenten

	Riptide Bibliothek	CLI-Anwendung	Proxy-Server	Docker Container-Backend	MySQL Datenbank-Treiber	Config-crunch
Anzahl Codezeilen*	3185 / 4496	1032 / 1261	627 / 877	1683 / 1913	98 / 109	882 / 1263
Issues bei GitHub	0	0	0	0	0	0
Forks bei GitHub	0	0	0	0	0	0
Sterne bei Github	3	1	1	0	0	0
Git Commits	81	68	42	63	13	63
Anzahl Mitwirkende	1	1	1	1	1	1
Anzahl der PyPI Abhängigkeiten** (direkt / indirekt)	8 / 15	5 / 21	4 / 19	2 / 27	2 / 17	3 / 4
Anzahl der Pakete bei PyPI, die von X abhängig sind	4	0	0	0	0	5

* Python Quellcode (ohne Kommentare / mit Kommentaren)
** „Direkt“ bezeichnet die Abhängigkeiten, die in der setup.py definiert sind, „indirekt“ bezeichnet die Anzahl der Pakete, die bei der Installation installiert werden (inklusive Abhängkeiten der Abhängkeiten).

7.4.2 Objektive Metriken

Für Software-Projekte können objektive Metriken erfasst werden (Ludewig und Lichter, 2013). Dabei handelt es sich um absolute Kennzahlen, wie Anzahl der Zeilen Code oder der offenen *Issues* auf Github. Diese Metriken sind meist absolut, da für die Einordnung auf einer Skala Vergleichswerte notwendig wären.

Die objektiven Metriken können daher nicht ohne Weiteres eingesetzt werden, um die zuvor definierten Qualitätsmerkmale auf einer Skala zu messen. Werden entsprechende Vergleichswerte geschaffen, beispielsweise durch Vergleich der objektiven Metriken über mehrere Software-Iterationen, so können relative Aussagen über die Qualitätsmerkmale getroffen werden. Das Verhältnis zwischen Anstieg der automatischen Software-Tests und dem Anstieg der Anzahl von Codezeilen kann beispielsweise Rückschluss auf die *Wartbarkeit* ermöglichen. In einem wirtschaftlich betrachtetem Software-Projekt kann die Anzahl der Codezeilen zudem Rückschlüsse über die Rentabilität des Projektes geben, indem für eine Codezeile ein durchschnittlicher Kostenwert ermittelt wird (Ludewig und Lichter, 2013).

Für Riptide sind einige objektiven Metriken in Tabelle 7.1 aufgelistet. Da noch keine Vergleichswerte vorliegen, kann mit diesen Werten zu diesem Zeitpunkt noch keine Aussage über die Erfüllung der Qualitätsmerkmale getroffen werden.

7.4.3 Subjektive Metriken

Neben der Erfassung objektiver Metriken können Software-Projekte auch subjektiv beurteilt werden. Diese Beurteilung kann beispielsweise durch professionelle Gutachter erfolgen (Ludewig und Lichter, 2013). Um eine subjektive Beurteilung durchzuführen, müssen messbare Qualitätsaspekte definiert werden. Diese Qualitätsaspekte werden dann von Gutachtern betrachtet und anhand einer Skala bewertet. Anhand einer Zuordnung der Qualitätsaspekte zu Qualitätsmerkmalen und einer Auswertung der Skalen können so Aussagen über die Erfüllung einzelner Qualitätsmerkmale getroffen werden.

Fragebogen

Für die subjektive Evaluation von Riptide wurde ein Fragebogen konzipiert (siehe Anhang), der von einigen Mitarbeitern der Tudock GmbH am Ende der Evaluationsphase ausgefüllt wurde (vgl. Kapitel 7.3). Die Mitglieder der Testgruppe werden somit als Gutachter der Software eingesetzt.

Einige Fragen erlauben den Testern Textantworten einzutragen. Diese werden nicht zur Bestimmung der Qualitätswerte verwendet, geben allerdings auch Aufschluss über die Erfahrungen der Tester und die Qualität der Software.

Viele Fragen im Fragebogen wurden entweder direkt Qualitätsmerkmalen oder einzelnen Anforderungen zugeordnet. Die meisten Fragen arbeiten dabei mit einer Skala von eins bis fünf, wobei fünf jeweils der beste Wert ist. Andere Fragen sind dieser Skalierung angepasst, wobei die Anpassung jeweils angegeben ist.

Um das Ergebnis einer einzelnen Frage zu bestimmen, werden die einzelnen Antworten folgendermaßen gemittelt. Für alle Anforderungen werden zunächst alle zugeordneten Fragen gesammelt. Die Ergebnisse zu diesen Fragen werden nun entsprechend zur Gewichtung der Anforderung gewichtet gemittelt. Es ergibt sich daraus ein Gesamtwert für eine Anforderung. Im nächsten Schritt werden die Werte für die Anforderungen und die direkten Zuordnungen von Fragen zu Qualitätsmerkmalen für jedes Merkmal gewichtet

gemittelt, woraus sich ein Gesamtwert für ein Qualitätsmerkmal ergibt (Ludewig und Lichter, 2013, Kapitel 14.3.3). Dieses Verfahren wird in der *Auswertung* anhand eines Beispiels gezeigt.

Für das Merkmal Sicherheit befindet sich keine Frage im Fragebogen, daher wird dieses Merkmal für die subjektive Auswertung nicht weiter betrachtet. Bis auf diese Ausnahme wurde sichergestellt, dass jedes Qualitätsmerkmal ausreichend ausgewertet werden kann.

Auswertung

Nachfolgend werden die ausgefüllten Fragebogen ausgewertet. Fünf Testpersonen haben den Fragebogen ausgefüllt. Die konkreten Ergebnisse sind zusammen mit dem Fragebogen Teil des Anhangs.

Da es sich um eine *sehr* kleine Testgruppe handelt und der Rahmen der Benutzung des Systems nicht vorgegeben war, sind diese Ergebnisse alleine nicht aufschlussreich. Um gut auswertbare und repräsentative Ergebnisse zu erzielen, muss eine viel größere Gruppe von Testern vorliegen und der Umfang der Nutzung des Systems muss vorgegeben werden. Es handelt sich bei der nachfolgenden Auswertung daher lediglich um ein Beispiel, um den Auswertungsprozess zu demonstrieren. Zur vollständigen Begutachtung wäre zudem die Überprüfung systeminterner Merkmale wie der Code-Qualität notwendig.

Die Antworten mit Skalen werden wie unter *Fragebogen* beschrieben ausgewertet. Für das Mittel der Anforderung A1.3 ergibt sich folgende Rechnung, dabei seien a_F die jeweiligen Antwort-Werte zu den entsprechenden Fragen und g_F die definierten Gewichte der Fragen zu Anforderung A1.3:

$$
\begin{aligned}
q_{A1.3} &= \frac{g_{F4.1} \cdot a_{F4.1} + g_{F4.2} \cdot a_{F4.2} + g_{F4.3} \cdot a_{F4.3}}{g_{F4.1} + g_{F4.2} + g_{F4.3}} \\
&= \frac{20 \cdot 3,75 + 100 \cdot 4,25 + 20 \cdot 4,5}{140} \\
&\approx 4,214
\end{aligned}
$$

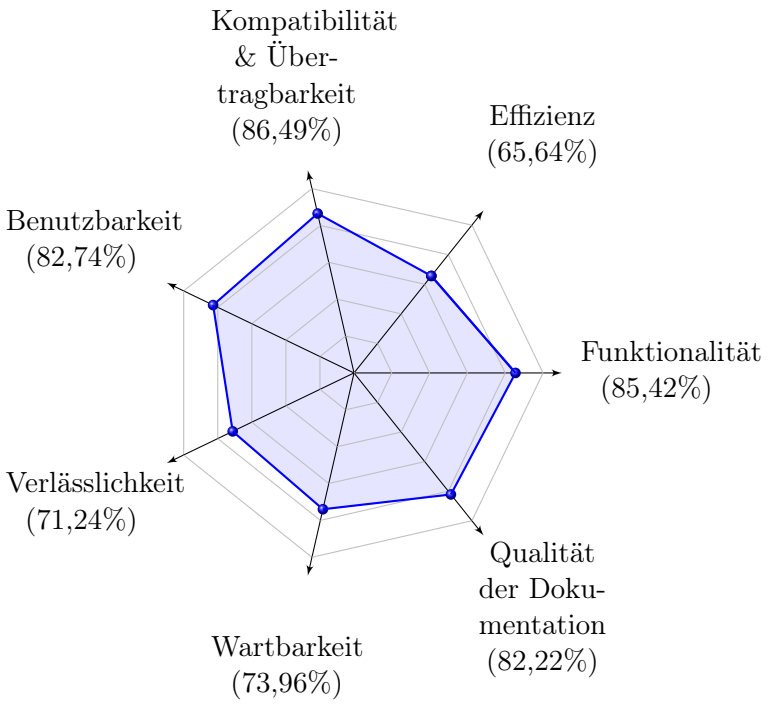


Abbildung 7.1: Werte der Qualitätsmerkmale nach Auswertung der fünf Antworten auf den Fragebogen

Für das Mittel des Qualitätsmerkmales *Effizienz* ergibt sich folgende Rechnung, dabei seien q_A die gewichteten Mittel der Anforderungen, a_F jeweilige Antwort-Werte und $g_{F/A}$ jeweils definierte Gewichte:

$$\begin{aligned} q_{Effizienz} &= \frac{g_{A1.1} \cdot q_{A1.1} + g_{F4.4} \cdot a_{F4.4} + g_{F11.1} \cdot a_{F11.1}}{q_{A1.1} + q_{F4.4} + q_{F11.1}} \\ &= \frac{100 \cdot 3,147 + 5 \cdot 4,75 + 5 \cdot 4,5}{110} \\ &\approx 3,282 \end{aligned}$$

Es sind gerundete Werte angegeben, die Rechnung wurde mit exakten Werten durchgeführt. Diese Rechnungen werden entsprechend für alle Qualitätsmerkmale durchgeführt. Es ergeben sich Endwerte, wie in Abbildung 7.1 gezeigt. 100% entspricht dabei einem Skalen-Wert von fünf und 0% einem Skalenwert von eins. Von den fünf Antworten auf den Fragebogen hat eine Person auf Frage F1.2 „0-10%“ geantwortet, und damit angegeben,

Riptide während der Evaluation nicht verwendet zu haben. Aus diesem Grund liegen für alle Fragen, die zur Bestimmung der Qualitätswerte benutzt wurden, nur vier Antworten vor.

Der Durchschnitt für alle Qualitätswerte liegt bei 78,24% (3,9112). Die Testpersonen haben das System somit grundsätzlich eher positiv bewertet. Die Testgruppe hat das System kurz nach Fertigstellung der ersten Version getestet, daher waren Fehler und Probleme zu erwarten. Daher sind auch Effizienz und Verlässlichkeit als schwächste Werte nicht überraschend.

Obwohl die Daten aufgrund der kleinen Testgruppe nicht repräsentativ sind, lassen sich dennoch erste Erkenntnisse aus den Fragebögen ziehen. Tendenziell sind die meisten Fragen von drei Testpersonen eher positiv und von einer Person eher negativ beantwortet wurden, daraus lässt sich ableiten, dass eine Person Probleme im Umgang mit Riptide hatte. Dies zeigt sich insbesondere in den Antworten auf die Fragen F2.1, F2.5 und F5.2.

Frage F2.5 ist der Hauptgrund für den relativ niedrigen Gesamtwert für das Qualitätsmerkmal Effizienz. Bei zwei Personen hat der Einrichtungsprozess mehr als vier Stunden Zeit in Anspruch genommen. Für die Installation des Systems und die Ersteinrichtung der Entwicklungsumgebungen lässt sich somit feststellen, dass der Prozess noch nicht für alle potentiellen Benutzer einfach und zuverlässig funktioniert. Einige Text-Antworten geben bereits einen ersten Einblick auf potentielle Verbesserungsmöglichkeiten, insbesondere F2.6 und F9.4. Einige Antworten stehen dabei allerdings auch scheinbar in Widerspruch zueinander. Es zeigt sich dadurch, dass verschiedene Entwickler unterschiedliche Anforderungen an eine Anleitung zur Installation des Systems haben. Um mehr Aufschluss über Verbesserungsmöglichkeiten zu erhalten muss diese Umfrage mit einer repräsentativen Testgruppe erneut durchgeführt werden.

8 Fazit

In dieser Arbeit wurde ein System zur Definition und Ausführung von Entwicklungsinstanzen für Webanwendungen konzipiert und implementiert. Ziel war es, basierend auf einer Anforderungsanalyse und dem Vergleich von bestehenden Lösungen, ein System zu schaffen, welches Entwickler-Teams ermöglicht, einfach reproduzierbare Entwicklungsinstanzen zu definieren, zu teilen und mit diesen ohne Einschränkungen entwickeln zu können. Im Folgenden werden die Ergebnisse dieser Arbeit zusammengefasst und ein Ausblick gegeben, in Hinsicht auf die Verbesserung des Systems und weiterer Möglichkeiten mit der Technologie.

8.1 Zusammenfassung

In dieser Arbeit wurde die Einrichtung und Bedienung von Entwicklungsinstanzen für Webanwendungen analysiert und basierend darauf Anforderungen definiert. Es wurden manuelle und automatische Einrichtungsprozesse verglichen. Dabei wurde festgestellt, dass die automatische Einrichtung von Entwicklungsinstanzen grundlegende Vorteile bietet, diese allerdings davon abhängig sind, wie gut und einfach die eingesetzten Werkzeuge und Definitionsdateien der Systeme die Anforderungen der Nutzer abbilden können.

Es wurden verschiedene Lösungen zur Automatisierung der Einrichtung von Entwicklungsinstanzen verglichen und Gemeinsamkeiten, Vorteile und Nachteile gesammelt.

Basierend auf den Analyseergebnissen wurde ein System konzipiert, welches den Nutzern erlaubt, möglichst einfach Entwicklungsinstanzen zu definieren und diese Definitionen über mehrere Projekte zu teilen, um den Aufwand und die Komplexität der Instanzen zu reduzieren und standardisierte Prozesse einführen zu können. Das System erlaubt Entwicklern, alle gewohnten Funktionen, die bei einer manuellen Einrichtung ohne Virtualisierung zur Verfügung stehen würden, zu nutzen.

Das System wurde so konzipiert, dass der Einsatz ohne Kenntnisse über Container oder Virtualisierung möglich ist. Mechanismen wie ein Proxy-Server, über den Dienste aller Projekte einheitlich erreichbar sind und sich automatisch bei Zugriff starten lassen, erleichtern Entwicklern die Arbeit mit dem System.

Es wurde eine Definitionssprache konzipiert, über die sich Entwicklungsinstanzen definieren lassen. Teile der Definition lassen sich in Dateien und separate Git Repositories auslagern. Projektdateien können von ausgelagerten Definitionsdateien erben und einzelne Aspekte überschreiben. So ist es möglich ein hierarchisches und einheitliches Konfigurationsmanagement über mehrere Projekte zu schaffen. Mit dem System wird ein offenes Repository zur Verfügung gestellt, in dem die Entwicklergemeinschaft Definitionen für Webanwendungen bereitstellen kann. Über diesen Mechanismus sollen Entwickler-Teams das System auch ohne Kenntnis über die Definitionssprache nutzen können.

Docker wurde als Container-Engine eingesetzt und hat sich für das System bewährt, da es unter anderem auf allen Betriebssystemen einsetzbar ist. Das System unterstützt die Installation von Implementationen für andere Virtualisierungslösungen, um auf zukünftige Entwicklungen vorbereitet zu sein. An einigen Stellen hat sich gezeigt, dass Docker noch eine relativ junge Technologie ist, da während der Entwicklung einige Fehler, ungewöhnliche Eigenheiten und Unterschiede je nach Betriebssystem aufgetreten sind, die die Entwicklung zeitweise gestört haben. Der Einsatz automatischer Tests hat die Behebung und Reduzierung von Fehlern unterstützt.

Insgesamt konnte erfolgreich ein System zur Einrichtung und Ausführung von Webanwendungen geschaffen werden, zumindest für die in dieser Arbeit getesteten Anwendungen und Frameworks. Die entwickelte Lösung ist bereits umfangreich und geht über den Umfang eines Prototyps hinaus. Dies liegt primär an dem Usability-Anspruch für Entwickler, welcher sich über unvollständige Prototypen nicht hätte vollständig umsetzen oder evaluieren lassen. Es wurde eine Evaluation mit einer kleinen Testgruppe durchgeführt. Die Testgruppe hat das System grundsätzlich positiv bewertet, ein Benutzer hatte allerdings Probleme mit der Installation des Systems und der Einrichtung von Entwicklungsumgebungen. Ein Test mit mehr Benutzern und mehr Anwendungen ist notwendig, um konkrete Aussagen über die Qualität des Systems treffen zu können.

8.2 Ausblick

Das entwickelte System kann auf verschiedene Arten erweitert und verbessert werden.

1. **Versuche mit anderen Container-Backends.** In dieser Arbeit wurde als Container-Backend für das Riptide System eine Anbindung an die Docker Engine entwickelt. Das System könnte mit weiteren Container-Backends erweitert werden. Es gibt einige neurere Container Engines, wie rkt von CoreOS (`https://coreos.com/rkt/`), für die eine Implementation spannend wären. Desweiteren wären auch Versuche mit klassischen Virtualisierungslösungen (VMWare, VirtualBox, etc.) interessant oder hybride Lösungen über eine Anbindung an Systeme wie Vagrant.

2. **Produktiveinsatz der Definitionssprache.** Die konzipierte Definitionssprache ist nur für die Definition von Entwicklungsinstanzen gedacht. Für Entwickler-Teams, die Riptide einsetzen möchten, bedeutet dies allerdings, mindestens doppelte Definitionen zu pflegen. Es müssen Definitionen für die Entwicklungsinstanzen und für die Produktiv- und Testsysteme gepflegt werden. Eine interessante Weiterführung des Projektes wäre die Möglichkeit, auf Basis von Riptide Definitionen auch klassische Serversysteme für den Test- und Produktiveinsatz zu definieren. Eine mögliche Lösung wäre eine *Custom Resource Definition* für Kubernetes (`https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/`). Über eine solche Definition wäre es eventuell möglich, Riptide Definitionen automatisch in Kubernetes Objekte zu überführen.

3. **Erstellung von Docker Images.** Das System kann aktuell keine Docker Images erzeugen. Es ließe sich um einen Mechanismus zum Bau von Images erweitern, um nicht von externen Prozessen zum Bauen und Veröffentlichen von Images abhängig zu sein.

4. **Einsetzen anderer Image-Formate.** Riptide ist aktuell von Docker Images abhängig. Container-Backends müssen Docker Images untersützen. Container-Engines wie Docker und rkt unterstützen dieses Image-Format zwar, ein Container-Backend für klassische VM-Lösungen müsste die Images allerdings zunächst transformieren, um sie einsetzen zu können. Das System könnte erweitert werden, um verschiedene Image-Formate zu unterstützen.

5. **Bessere Kompatibilität für Windows und macOS.** Riptide ist zwar mit Windows und macOS kompatibel, die eingesetzte Docker Engine für Windows und Mac bringt jedoch einige Probleme mit sich, allen voran langsamer Volume-Zugriff im Vergleich zu Linux. Des Weiteren sind viele Aspekte von Riptide aktuell für unixartige Systeme optimiert.

6. **Ausführliche Evaluation des Systems.** Die Evaluation von Riptide in dieser Arbeit wurde mit einer sehr kleinen Testgruppe durchgeführt. Eine Evaluation mit einer größeren Testgruppe und Tests mit weiteren Anwendungen und Frameworks ist sinnvoll.

Literaturverzeichnis

[Apache 2019] Apache: *Apache Module mod_proxy : Forward Proxies and Reverse Proxies/Gateways.* 2019. – URL http://httpd.apache.org/docs/current/mod/mod_proxy.html#forwardreverse. – Abruf: 2019-02-28

[Bruins Slot 2019] Bruins Slot, Jan P.: *Docker Django.* 2019. – URL https://github.com/erroneousboat/docker-django. – Abruf: 2019-03-10

[Coca u. a. 2019] Coca, Brian ; Kuratomi, Toshio ; Barker, John: *Ansible 2.7 Documentation.* 2019. – URL https://docs.ansible.com/ansible/2.7/index.html. – Abruf: 2019-05-03

[Docker 2019a] Docker: *Dockerfile reference.* 2019. – URL https://docs.docker.com/v17.09/engine/reference/builder/. – Abruf: 2019-04-26

[Docker 2019b] Docker: *Get Started, Part 2: Containers : Dockerfile.* 2019. – URL https://docs.docker.com/get-started/part2/#dockerfile. – Abruf: 2019-02-28

[Docker 2019c] Docker: *Get started with Docker for Windows.* 2019. – URL https://docs.docker.com/docker-for-windows/. – Abruf: 2019-02-28

[Docker 2019d] Docker: *Isolate containers with a user namespace.* 2019. – URL https://docs.docker.com/engine/security/userns-remap/. – Abruf: 2019-02-28

[Docker 2019e] Docker: *Networking features in Docker Desktop for Windows.* 2019. – URL https://docs.docker.com/docker-for-windows/networking/. – Abruf: 2019-02-28

[Docker 2019f] Docker: *Networking overview.* 2019. – URL https://docs.docker.com/network/. – Abruf: 2019-02-28

[Docker 2019g] Docker: *Overview of Docker Compose*. 2019. – URL `https://docs.docker.com/compose/overview/`. – Abruf: 2018-11-24

[Docker 2019h] Docker: *Performance tuning for volume mounts (shared filesystems)*. 2019. – URL `https://docs.docker.com/docker-for-mac/osxfs-caching/`. – Abruf: 2019-02-28

[Docker 2019i] Docker: *Use volumes*. 2019. – URL `https://docs.docker.com/storage/volumes`. – Abruf: 2019-02-28

[Estdale und Georgiadou 2018] Estdale, John ; Georgiadou, Elli: *Applying the ISO/IEC 25010 Quality Models to Software Product: 25th European Conference, EuroSPI 2018, Bilbao, Spain, September 5-7, 2018, Proceedings*. S. 492–503, 01 2018. – ISBN 978-3-319-97924-3

[Fette und Melnikov 2011] Fette, I. ; Melnikov, A.: The WebSocket Protocol / RFC Editor. RFC Editor, December 2011 (6455). – RFC. – URL `http://www.rfc-editor.org/rfc/rfc6455.txt`. – ISSN 2070-1721

[Fielding u. a. 1999] Fielding, Roy T. ; Gettys, James ; Mogul, Jeffrey C. ; Nielsen, Henrik F. ; Masinter, Larry ; Leach, Paul J. ; Berners-Lee, Tim: Hypertext Transfer Protocol – HTTP/1.1 / RFC Editor. RFC Editor, June 1999 (2616). – RFC. – URL `http://www.rfc-editor.org/rfc/rfc2616.txt`. – ISSN 2070-1721

[Gamma u. a. 2011] Gamma, Erich ; Helm, Richard ; Johnson, Ralph: *Entwurfsmuster : Elemente wiederverwendbarer objektorientierter Software*. München : Addison-Wesley, 2011. – Deutsche Übersetzung: Riehle, Dirk. – ISBN 978-1-4302-4569-8

[Hasegawa u. a. 2019] Hasegawa, Makoto ; Teng, Qiming ; Perry, Steve: *Kubernetes Dokumentation - Kubernetes 1.14*. 2019. – URL `https://v1-14.docs.kubernetes.io/docs/home/`. – Abruf: 2019-05-03

[Hogg 2014] Hogg, Scott: *Software Containers: Used More Frequently than Most Realize*. 2014. – URL `https://www.networkworld.com/article/2226996/software-containers--used-more-frequently-than-most-realize.html`. – Abruf: 2019-02-28

[Hüttermann 2012] Hüttermann, Michael: *DevOps for developers*. New York, NY : Apress, 2012. – ISBN 978-1-4302-4569-8

[Kruchten 1995] Kruchten, P. B.: The 4+1 View Model of architecture. In: *IEEE Software* 12 (1995), Nov, Nr. 6, S. 42–50. – ISSN 0740-7459

[Leers und Nuß 2018] Leers, Bernhard ; Nuss, Julian: *Magedev*. 2018. – URL `https://github.com/teamneusta/php-cli-magedev`. – Abruf: 2019-03-10

[Linux 2017] Linux: *HOSTS(5) Linux Programmer's Manual*, September 2017. – Verfügbar unter `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tag/?h=v5.0.5`. Kopie der Anleitungsseite abrufbar unter `http://man7.org/linux/man-pages/man5/hosts.5.html`.

[Linux 2019] Linux: *capabilities(7) Linux Programmer's Manual*, März 2019. – Verfügbar unter `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tag/?h=v5.0.5`. Kopie der Anleitungsseite abrufbar unter `http://man7.org/linux/man-pages/man7/capabilities.7.html`.

[Ludewig und Lichter 2013] Ludewig, Jochen ; Lichter, Horst: *Software Engineering : Grundlagen, Menschen, Prozesse, Techniken*. Heidelberg : Dpunkt.Verlag GmbH, 2013. – ISBN 978-3-86490-092-1

[Marslender u. a. 2019] Marslender, Chris ; Chantre, Benoît ; Chung, Christian: *WordPress Docker Development Environment*. 2019. – URL `https://github.com/10up/wp-local-docker`. – Abruf: 2019-03-10

[Morris 2016] Morris, Kief: *Infrastructure as Code : Managing Servers in the Cloud*. Sebastopol, CA : O'Reilly, 2016. – ISBN 978-1-4919-2435-8

[Mouat 2016] Mouat, Adrian: *Using Docker : developing and deploying software with containers*. Sebastopol, CA : O'Reilly, 2016. – ISBN 978-1-4919-1576-9

[Rahman u. a. 2018] Rahman, Akond ; Partho, Asif ; Morrison, Patrick ; Williams, Laurie: What Questions Do Programmers Ask About Configuration As Code? In: Bosch, Jan (Hrsg.) ; Fitzgerald, Brian (Hrsg.) ; Goedicke, Michael (Hrsg.): *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering (RCoSE '18)*. New York, NY, USA : ACM, 2018, S. 16–22. – URL `http://doi.acm.org/10.1145/3194760.3194769`. – Abruf: 2018-11-24. – ISBN 978-1-4503-5745-6

[Roberts u. a. 2019] Roberts, Chris ; Cain, Brian ; Vargo, Seth: *Documentation - Vagrant by HashiCorp*. 2019. – URL `https://www.vagrantup.com/docs/index.html`. – Abruf: 2019-05-03

[Robinson und Coar 2004] Robinson, D. ; Coar, K.: The Common Gateway Interface (CGI) Version 1.1 / RFC Editor. RFC Editor, October 2004 (3875). – RFC. – URL `http://www.rfc-editor.org/rfc/rfc3875.txt`. – ISSN 2070-1721

[Wolff 2018] Wolff, Eberhard: *Das Microservices-Praxisbuch : Grundlagen, Konzepte und Rezepte*. Heidelberg : Dpunkt.Verlag GmbH, 2018. – ISBN 3-86490-526-5

[Öggl und Kofler 2018] Öggl, Bernd ; Kofler, Michael: *Docker : das Praxisbuch für Entwickler und DevOps-Teams*. Bonn : Rheinwerk Verlag GmbH, 2018. – ISBN 978-3-8362-6176-0

A Anhang

Fragebogen zur Evaluation

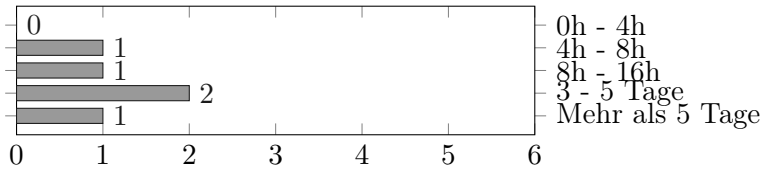
Siehe 7.4.3 Fragebogen. Falls eine Frage für die subjektive Bewertung verwendet wurde, sind die Gewichtungen angegeben. Die X-Achsen der Diagramme bilden die Anzahl der Antworten ab.

Bei allen Fragen ist jedem Teilnehmer nur eine Antwort erlaubt, es sei denn etwas anderes ist angegeben. Eine geringere Anzahl an Antworten als Teilnehmer bedeutet, dass einzelne Teilnehmer eine Frage nicht beantwortet haben.

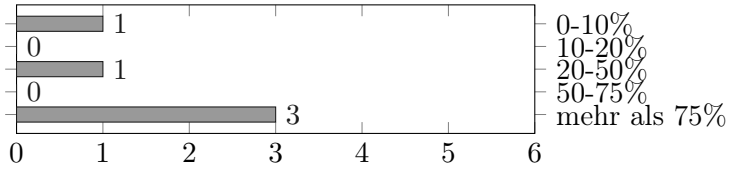
Anzahl Teilnehmer am Fragebogen: 5

Einleitung

F1.1 Wie viele Stunden arbeiten Sie pro Woche durchschnittlich an der Entwicklung von Webanwendungen? (1 Tag = 8h)

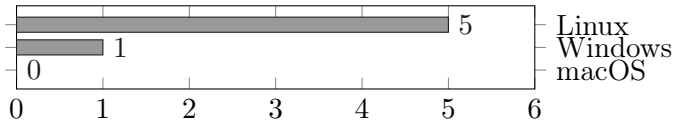


F1.2 Welchen Anteil hat die Arbeit mit Riptide im Durchschnitt in ihrer Entwicklungsarbeit ausgemacht? Dazu zählt auch Arbeit an Projekten, die über Riptide gestartet sind.



Bei Wahl der Option, „0-10%“ wird der Fragebogen nach diesem Abschnitt beendet.

F1.3 Auf welchen Betriebssystemen haben Sie Riptide eingesetzt?

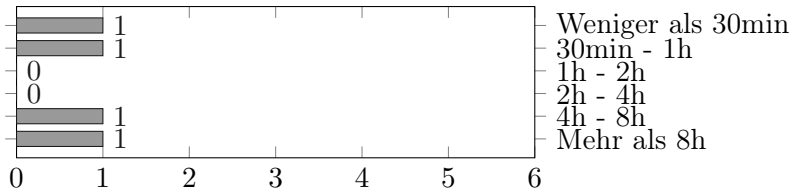


Mehrfachauswahl war möglich.

Installation von Riptide

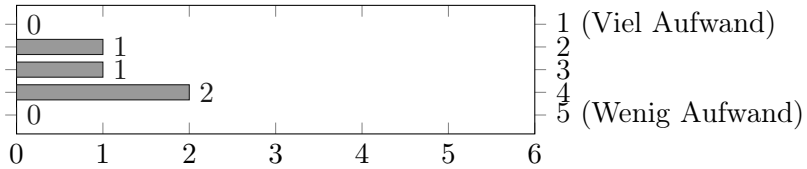
F2.1 Wie viel Zeit hat die Einrichtung von Riptide in Anspruch genommen (bis mit dem ersten Projekt gearbeitet werden konnte)?

Lineare Skala: „Weniger als 30min“ wird der Wert 5 zugewiesen, „Mehr als 8h“ der Wert 1.



Gewichtung: A1.1 (80)

F2.2 Bitte bewerten Sie die Aufwand des Installationsprozesses, falls möglich im Vergleich zur klassischen Einrichtung ihres Entwicklungssystems.



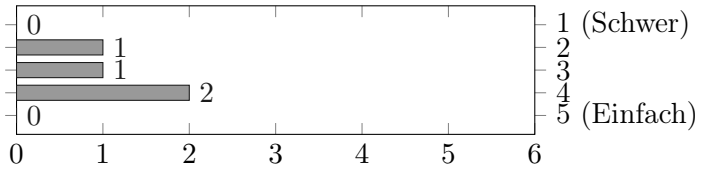
Gewichtung: A1.1 (80)

F2.3 Welche Verbesserungsvorschläge haben Sie für den Installationsprozess?

Antworten:

- „Alle Vorschläge die ich gehabt hätte wurden durch spätere Updates bereits umgesetzt. So etwas wie ein zentrales Installationsscript und eifnachere Updates usw. Aber das gibt's ja jetzt schon alles.“

F2.4 Wie einfach fanden Sie die Ersteinrichtung eines Projektes („riptide setup“)?



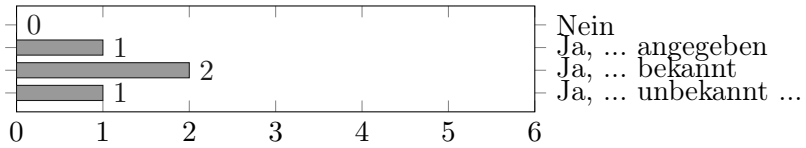
Gewichtung: A1.1 (100)

F2.5 Waren nach der Ausführung von „riptide setup“ weitere Schritte notwendig, damit das Projekt einsatzfähig war?

Falls Sie mehrere Projekte verwendet haben: Antworten Sie mit Ihrer schlechtesten Erfahrung.

Antwortmöglichkeiten:

- Nein
- Ja, aber alle zusätzlichen Schritte waren angegeben
- Ja, aber die Schritte waren bekannt
- Ja, die Schritte waren mir unbekannt, ich hatte Probleme das Projekt einzurichten



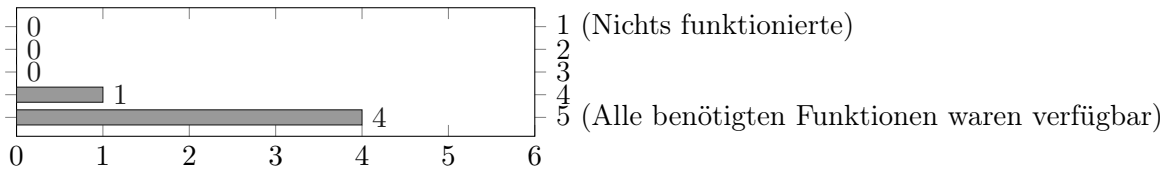
F2.6 Welche Verbesserungsvorschläge haben sie zum „riptide setup“-Prozess?

Antworten:

- „To provide more details information“
- „kürzere ToDos, sprich weniger Text, aber das liegt ja bei einem selber“

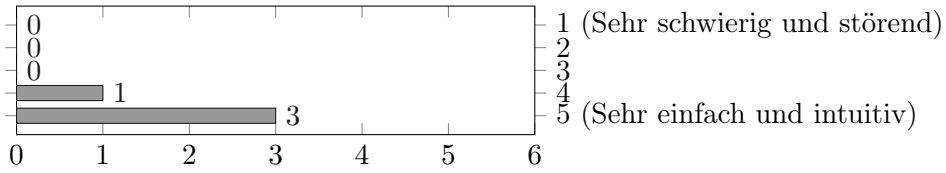
Grundfunktionen

F3.1 Bitte bewerten Sie, in welchem Umfang Ihnen Riptide die Funktionen der Webanwendungen zur Verfügung gestellt hat.



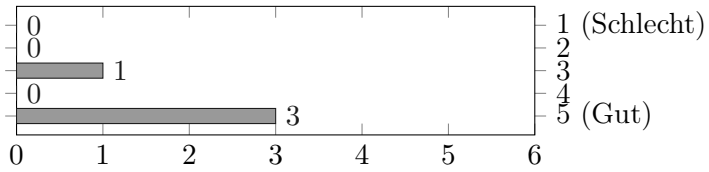
Gewichtung: A1.2 (100)

F3.2 Bitte bewerten Sie Ihre Erfahrung mit dem „riptide“-Befehl.



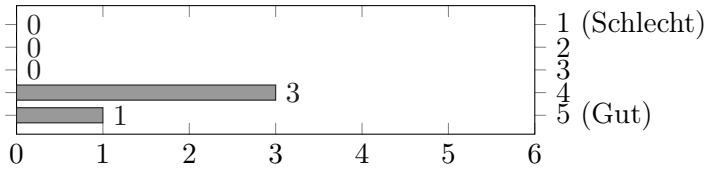
Gewichtung: Benutzbarkeit (10)

F3.3 Wie bewerten Sie ihr Erfahrung mit dem Starten und Stoppen von Projekten?



Gewichtung: Benutzbarkeit (5), Verlässlichkeit (5), A2.4 (30)

F3.4 Wie bewerten Sie die Anzeigen und Meldungen der „riptide“-CLI-Anwendung?



Gewichtung: Benutzbarkeit (5)

F3.5 Begründen Sie: Was finden Sie gut oder schlecht an den Meldungen und Anzeigen der „riptide“-CLI-Anwendung?

Antworten:

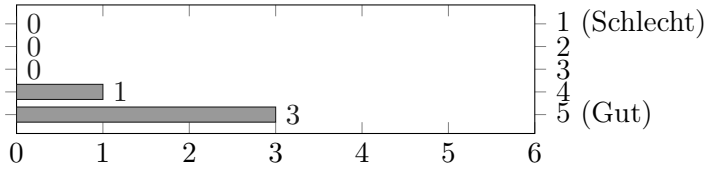
- „All messages are very clear“
- „Beispiel bei riptide -u sagt er am ende error missing command. Ja -u ist nur ein flag. Aber ich seh schon dass das Leute verwirrt. Ich würde ganz ehrlich NOCH mehr von absolut dummen Usern ausgehen.“

F3.6 Was würden Sie an der „riptide“-CLI-Anwendung verbessern?

Antworten:

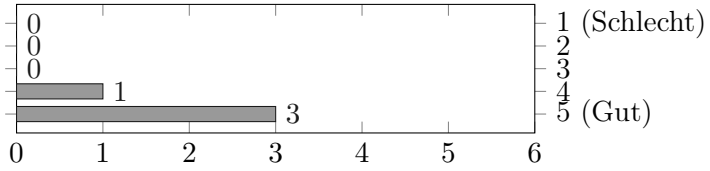
- „Understandable“

F3.7 Wie gut waren zusätzliche Dienste (bspw. Datenbanken) erreichbar?



Gewichtung: A1.4 (100)

F3.8 Wie gut ließen sich mit Riptide verschiedene Versionsstände von Code oder Datenbanken verwalten?



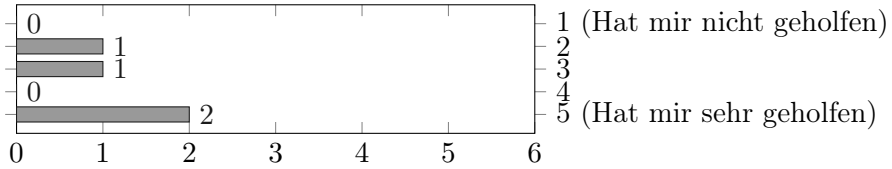
Gewichtung: A1.9 (100)

Bedienung des Proxy-Servers

Der Proxy-Server ist die Komponente von Riptide, die Ihnen erlaubt hat über den Browser auf Projekte zuzugreifen.

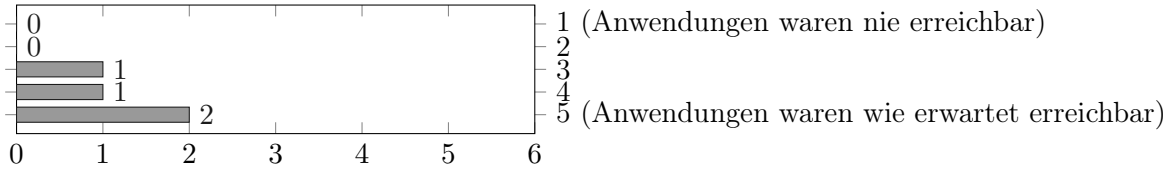
F4.1 Bitte bewerten Sie die Projektübersicht des Proxy-Servers.

Überspringen Sie die Frage, wenn sie nicht wissen was gemeint ist.



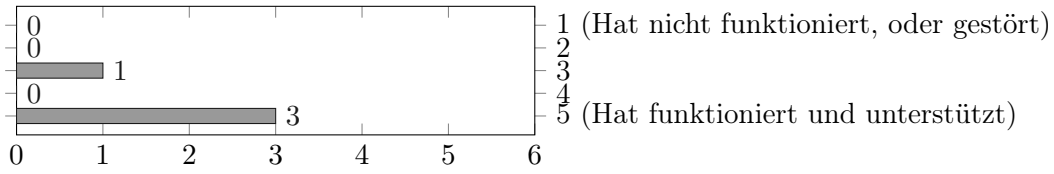
Gewichtung: A1.3 (20), Benutzbarkeit (5)

F4.2 Bitte bewerten Sie die Erreichbarkeit der Webanwendungen über den Proxy-Server.



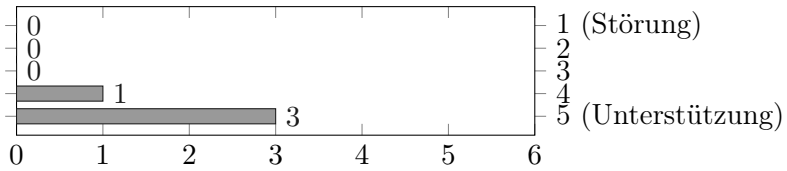
Gewichtung: A1.3 (100), A1.2 (30)

F4.3 Bitte bewerten Sie die Auto-Start Funktion des Proxy-Servers.



Gewichtung: A1.3 (20), A2.4 (30), Benutzbarkeit (5), Verlässlichkeit (5)

F4.4 Hat sie der Proxy-Server insgesamt eher gestört, oder Sie bei Ihrer Arbeit unterstützt?



Gewichtung: Benutzbarkeit (10), Verlässlichkeit (5), Effizienz (5)

F4.5 Welche Verbesserungsvorschläge haben Sie für den Proxy-Server?

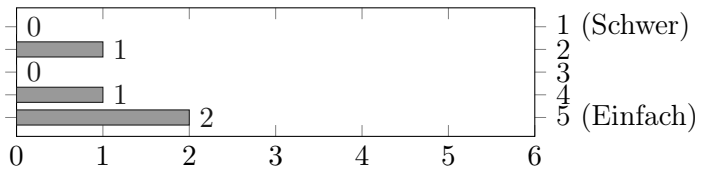
Antworten:

- „it could be easier to setup“
- „keinen, er läuft und das ist gut so“

CLI-Kommandos

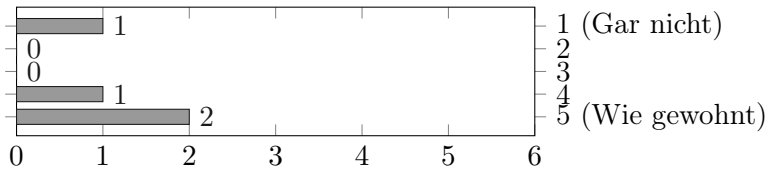
F5.1 Haben Sie die Riptide Shell-Integration genutzt?

F5.2 Wie einfach fanden Sie die Ausführung von im Projekt definierten CLI-Befehlen (npm, composer etc.)?



Gewichtung: A1.7 (100)

F5.3 In welchem Umfang konnten sie CLI-Befehle (npm, composer, etc.) nutzen, im Vergleich zur Nutzung ohne Riptide?



Gewichtung: A1.7 (70), A1.2 (30)

F5.4 Welches Verbesserungsvorschläge haben Sie zur Ausführung von CLI-Befehlen?

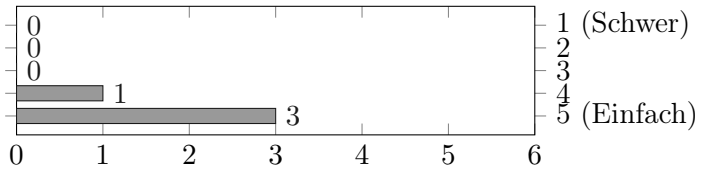
Antworten:

- „Wenn überhaupt die Anbindung von Composer. Ist echt nur ne Kleinigkeit aber wenn man vergisst "–ignore-platform-reqs„ zu nutzen dann macht das ja Probleme. Keine AHnung OB man das fixen kann. Wenn nich ist aber echt nicht schlimm."
- „Man muss darauf achten, wenn Kommandos gleich den Aliasen lauten, da der Alias Vorrang hat und man ein anderes Ergebnis erhält, als gedacht."

Arbeit mit mehreren Projekten

Bitte beantworten Sie diese Fragen nur, wenn sie mit mehreren Projekten gearbeitet haben.

F6.1 Wie einfach fanden Sie die Verwendung von mehreren Projekten mit Riptide?



Gewichtung: A1.10 (100)

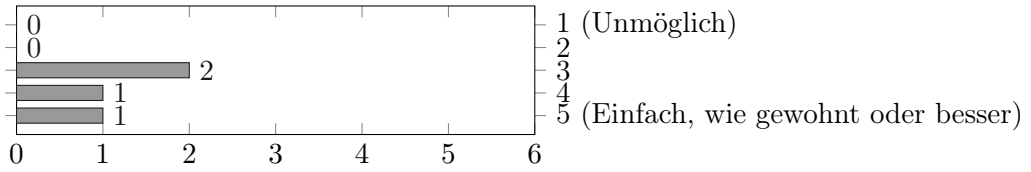
F6.2 Welches Verbesserungsvorschläge haben Sie zur Arbeit mit mehreren Projekten mit Riptide?

Antworten:

- „Easy to use and setup"
- „Die Projekte sind voneinander losgelöst und soweit sind bisher keine Probleme aufgetreten."

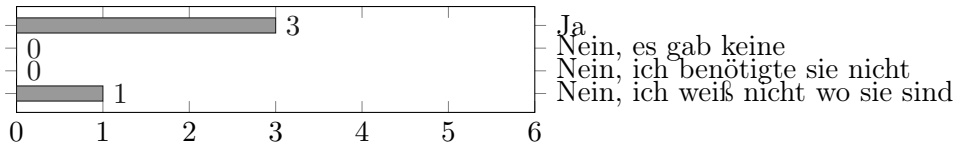
Debugging

F7.1 Wie einfachen fanden Sie die Nutzung von Debuggern mit Riptide?

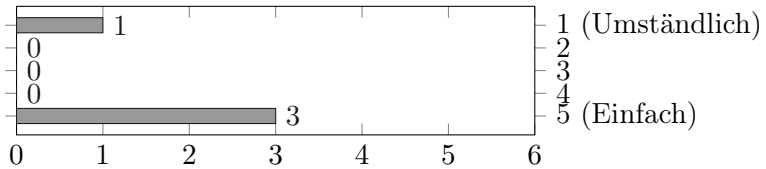


Gewichtung: A1.6 (100)

F7.2 Haben sie auf Log-Dateien zugegriffen, die Riptide Ihnen bereitgestellt hat für ihre Anwendung



F7.3 Wie einfach fanden Sie den Abruf von Logging-Dateien?



Gewichtung: A1.8 (100)

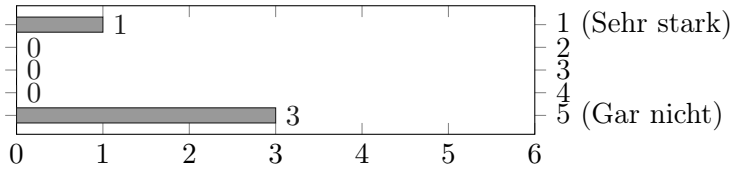
F7.4 Welche Verbesserungsvorschläge haben Sie zum Einsetzen von Debuggern oder dem Abruf von Log-Dateien mit Riptide?

Antworten:

- „Ich weiß nicht ob es derweil immer noch da ist, aber manchmal hat genervt dass auch die Nutzung von Magerun Befehlen usw. abgegriffen wurde. Das ist aber eher ein kleines Problem. Wenn man davon weiß ist es leicht zu umgehen.“
- „Die Verwendung ist in der Dokumentation erklärt und die Log-Dateien sind an den entsprechenden Orten auffindbar. Was noch nicht getestet wurde, ist der Aufruf über die Konsole. Aber hier sollte es auch kein anderes Verhalten geben.“

Dateiverwaltung und Berechtigungen

F8.1 Wie sehr haben Dateiberechtigungs-Probleme ohne „Datei nicht gefunden“-Fehler sie bei ihrer Arbeit behindert?



Gewichtung: A1.5 (100)

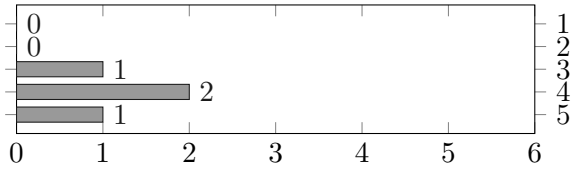
F8.2 Falls Sie Probleme bei der Verwaltung oder Bearbeitung von Dateien hatten, schildern Sie diese bitte.

Antworten:

- „Wird gemacht“

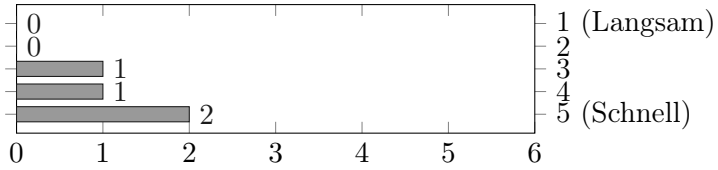
Dokumentation

F9.1 Wie bewerten Sie den Informationsgehalt der Dokumentation (auf einer Skala von 1 [Ich habe nichts gefunden, wonach ich gesucht habe] bis 5 [Ich habe alles gefunden])



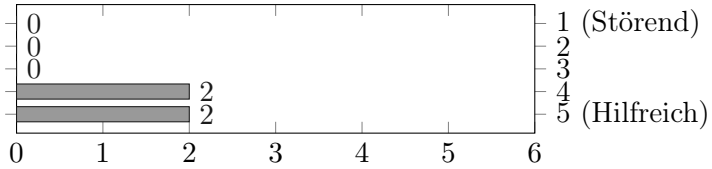
Gewichtung: A2.1 (80)

F9.2 Wie schnell haben Sie gewünschte Informationen in der Dokumentation gefunden?



Gewichtung: A2.1 (80)

F9.3 Wie bewerten Sie den Aufbau der Dokumentation?



Gewichtung: A2.1 (80)

F9.4 Welche Verbesserungsvorschläge haben Sie zur Dokumentation?

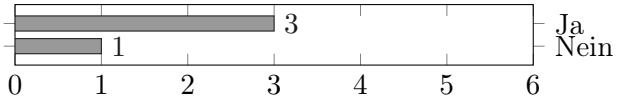
Antworten:

- „The last steps of process can be better documented “
- „I only had some issues about installing local or globally. I learned that installing without sudo installs the application locally. This info must be provided in documentation.“
- „Keinen Verbesserungsvorschlag. Ist aber viel zu lesen. :-) Daher sehr ausführlich.“

Definition von Projekten

Beantworten Sie die erste Frage. Beantworten Sie weitere Fragen nur, wenn Sie die erste Frage mit „Ja“ beantwortet haben.

F10.1 Haben sie eigene Änderungen an Projektdateien vorgenommen oder eigene App, Services oder Commands definiert?



F10.2 In welchem Umfang haben Sie eigene Projekte, Apps, Services oder Commands definiert?

Antwortmöglichkeiten:

a) Ich habe bestehende Projekte leicht geändert

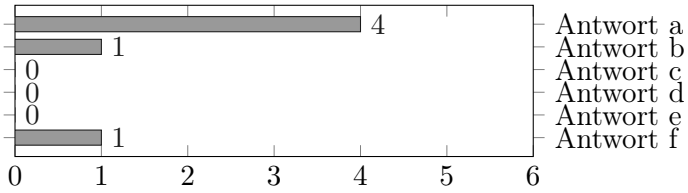
b) Ich habe neue Projekte angelegt und dabei bestehende Apps, Commands oder Services aus dem Riptide Repository genutzt

c) Ich habe neue Services oder Commands zu Projekten hinzugefügt

d) Ich habe eigene Apps definiert

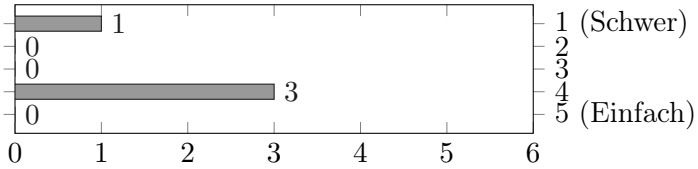
e) Ich habe eigene Services definiert

f) Ich habe eigene Commands definiert



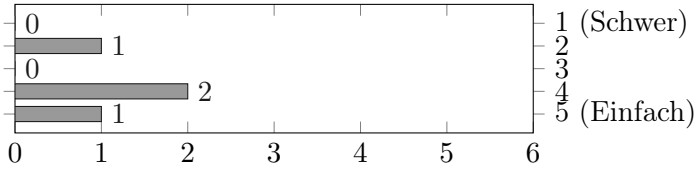
Mehrfachauswahl war möglich.

F10.3 Wie einfach fanden Sie es bestehende Projekte (oder andere Entitäten) zu ändern?



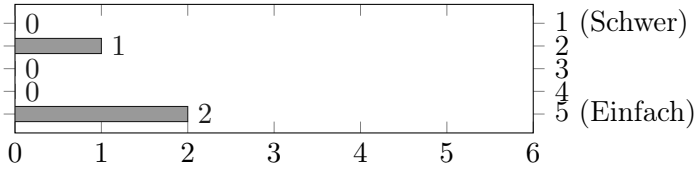
Gewichtung: A2.5 (100), A2.6 (20), A2.1 (20), A2.2 (20)

F10.4 Wie einfach nachvollziehbar fanden sie die Hierarchie der Entitäten (Projekte haben eine App, Apps haben Services und Commands, etc.)?



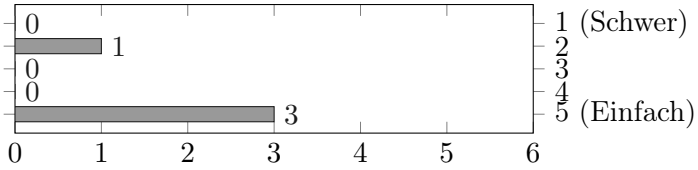
Gewichtung: A2.1 (40), A2.3 (20), A2.5 (20), A2.2 (20)

F10.5 Wie einfach nachvollziehbar fanden Sie die Einbindung von Dokumenten mittels des „$ref“-Schlüsselwortes?



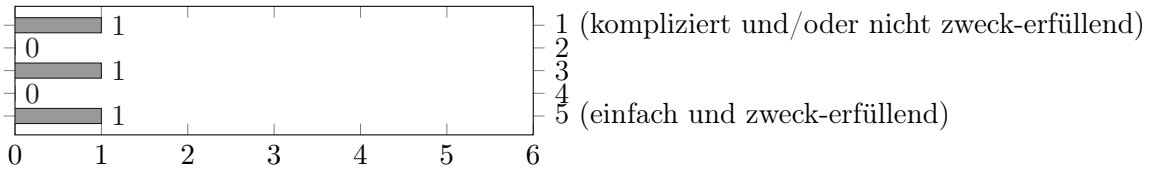
Gewichtung: A2.1 (20), A2.3 (40), A2.5 (20), A2.2 (20)

F10.6 Wie einfach fanden Sie die Arbeit mit Repositories?



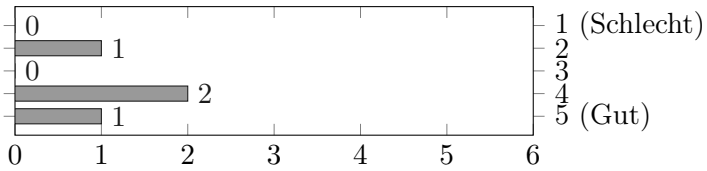
Gewichtung: A2.1 (20), A2.3 (40), A2.6 (20), A2.2 (20)

F10.7 Wie bewerten Sie die Schema der Projekte, Apps, Services, Commands?



Gewichtung: A2.1 (20), A2.5 (40), A2.3 (40), A2.2 (20)

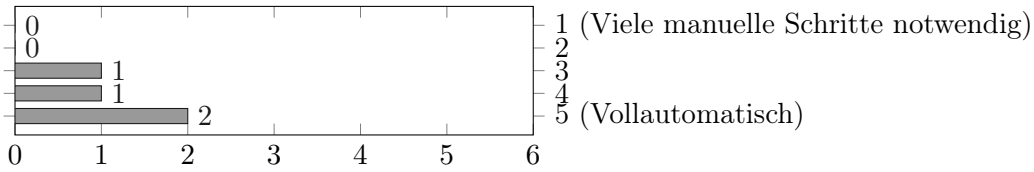
F10.8 Wie bewerten Sie das „Debuggen“ von Fehlern oder Problemen in Definitionsdateien?



Gewichtung: A2.6 (60)

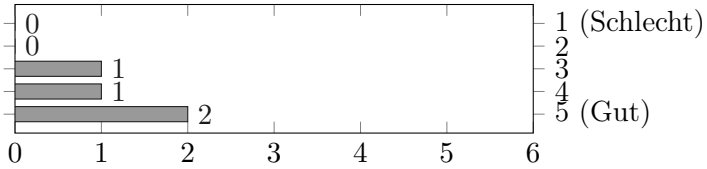
F10.9 Wie gut konnten Sie das Starten und Bedienen von Projekten automatisieren?

Wählen Sie 5, falls einmaliges „riptide setup“ für Entwickler ausreicht um ein Projekt zu nutzen und 1 falls umständliche Ausführung von zusätzlichen Befehlen für die Ersteinrichtung und nachfolgende Startvorgänge notwendig ist.



Gewichtung: A2.4 (60), A2.2 (50)

F10.10 Wie gut konnten Sie mit Riptide das Verhalten von Produktivsystemen abdecken?



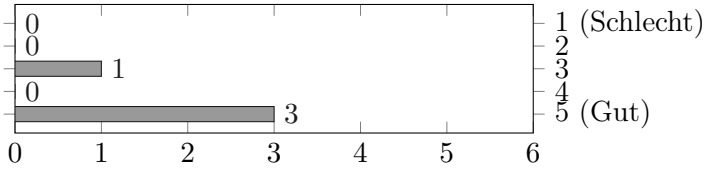
Gewichtung: A2.7 (100)

F10.11 Welche Verbesserungsvorschläge haben Sie zur Definition von Projekten, Apps, Services und Commands?

Keine Antworten.

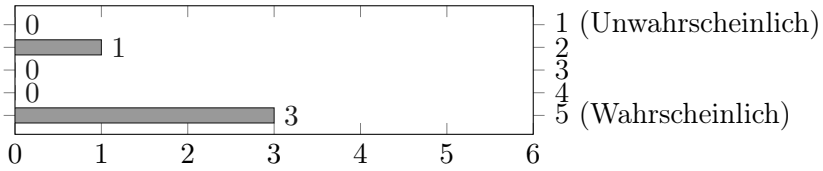
Weiteres Feedback

F11.1 Wie bewerten Sie insgesamt ihre Erfahrung mit Riptide?



Gewichtung: Funktionalität (8), Benutzbarkeit (8), Verlässlichkeit (5), Effizienz (5)

F11.2 Wie wahrscheinlich ist es, dass sie Kollegen oder Freunden Riptide weiterempfehlen?



F11.3 Welche Verbesserungsvorschläge oder Anregungen haben Sie noch?

Antworten:

- „Bisher keine, aber die kommen eventuell bei intensiver Nutzung“

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

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

Ort	Datum	Unterschrift im Original