Bachelorarbeit

im Studiengang
Engineering Business Information Systems

zur Erlangung des Grades eines
Bachelor of Science (B.Sc)

Thema:
Entwicklung und Implementierung einer API zur Steuerung virtueller Maschinen (VMs) und Containern

vorgelegt von:

Daniel Bilic
geb. am 30.10.1999 in Frankfurt/Main
Brunnenweg 92, 63071 Offenbach/Main
Matrikel-Nr.: 1306444

im Sommersemester 2023
am Fachbereich 2: Informatik und Ingenieurwissenschaften
der Frankfurt University of Applied Sciences

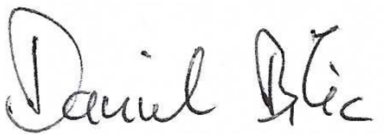
Erstprüfer: Prof. Dr. Christian Baun
Zweitprüfer: Lukas Atkinson

Abgabedatum: 22.06.2023

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit hat in gleicher Form noch keiner anderen Prüfbehörde vorgelegen.

Ort, Datum: Offenbach am Main, 22.06.2023

Unterschrift: 

Danksagung

Hiermit bedanke ich mich zum einen bei meiner Familie und meinen Freunden, die mich durch mein Studium begleitet haben und mir stets zur Seite standen. Zum anderen gehört mein Dank Herrn Prof. Dr. Christian Baun, sowie dem DaaS-Projektteam, bestehend aus Lukas Atkinson und Johannes Bouche, für die Unterstützung bei der Themenfindung. Alle drei standen mir bei Fragen zur Seite.

Zusammenfassung

Ziel und Aufgabe der vorliegenden Bachelorthesis waren die Entwicklung und Implementierung einer API zur Steuerung virtueller Maschinen (VMs) und von Containern in der Programmiersprache Python. Die VMs und Container enthalten unveränderte Desktop-Anwendungen für die Betriebssysteme Linux und Windows und zudem eine geeignete Lösung zur Bereitstellung eines Remote-Desktop-Protokolls. Die API realisiert die grundlegenden Funktionen zur Steuerung der VMs und Container, wie das Starten oder Stoppen einer Instanz. Ziele waren zudem, dass die realisierte Lösung leichtgewichtig und performant funktioniert sowie das Bereitstellen einer Sicherheitsfunktionalität.

Abstract

The goal and task of this bachelor thesis was to develop and implement an API for controlling virtual machines (VMs) and containers in the Python programming language. The VMs and containers contain unmodified desktop applications for the Linux and Windows operating systems and also a suitable solution for providing a remote desktop protocol. The API implements the basic functions for controlling the VMs and containers, such as starting or stopping an instance. The goals were also to make the realized solution lightweight and performant, as well as to provide a security functionality.

Inhaltsverzeichnis

Abkürzungsverzeichnis

VM	=	Virtuelle Maschine
KVM	=	Kernel-based Virtual Machine
QEMU	=	Quick Time Emulator
GUI	=	Graphical User Interface
CLI	=	Command Line Interface
API	=	Application Programming Interface
REST	=	Representational State Transfer
HTTP	=	Hypertext Transfer Protocol
SDK	=	Software Development Kit
URL	=	Uniform Resource Locater
URI	=	Uniform Resource Identifier

Abbildungsverzeichnis

1. Einleitung

Die Technologie hinter der Virtualisierung ist schon einige Dekaden alt, aber hat bis heute nichts an ihrer Aktualität eingebüßt. Sie findet Nutzen in nahezu allen Disziplinen der Informatik und ist weiterhin Gegenstand der Forschung. Daraus und mit Bezug auf das Forschungsprojekt der fra-aus leitet sich eine Relevanz für diese Arbeit ab. Die entwickelte Lösung soll die Möglichkeit bieten in eine neuartige Desktop-as-a-Service (DaaS)- Anwendung eingebunden und weiterentwickelt zu werden. Denn gerade in Zeiten vermehrter Tätigkeit im Homeoffice und in der Remote-Arbeitsumgebung ist der Einsatz von flexiblen und bedarfsgerechten Virtualisierungsumgebungen mehr denn je von entscheidender Bedeutung.

1.1. Zielsetzung

Das Ziel dieser Arbeit ist die Entwicklung und Implementierung einer API zur Steuerung von Docker Containern und Virtuellen Maschinen (VMs) des Typs qemu/kvm. Folgende funktionale Anforderungen muss die API erfüllen:

- Starten von Containern und Virtuellen Maschinen
- Stoppen von Containern und Virtuellen Maschinen
- Pausieren von Containern und Virtuellen Maschinen
- Neustarten von Containern und Virtuellen Maschinen
- Löschen von Containern und Virtuellen Maschinen
- Abrufen einer Liste aller laufenden Instanzen
- Abrufen detaillierter Informationen über eine Instanz

Zusätzlich sollen folgenden nicht-funktionale Anforderungen umgesetzt werden:

- Authentifizierung und Autorisierung
- Benutzerfreundlichkeit und Best Practices
- Performanz

Die API soll über eine REST-Schnittstelle ansprechbar sein. Weiterführend wird das Konzept der API in dem 3. Kapitel „Design“ ausgeführt.

1.2. ZIM-Kooperationsprojekt DaaS-Design

Diese Arbeit flankiert das ZIM-Kooperationsprojekt DaaS-Design zwischen der Frankfurt University of Applied Sciences und der Nuromedia GmbH. Aufgabe und Ziel des ZIM-Kooperationsprojektes DaaS-DESIGN ist die Entwicklung und prototypische Implementierung eines neuartigen Desktop-as-a-Service (DaaS) mit Anwendungsvirtualisierung zum Betrieb beliebiger Linux- und Windows-Anwendungen in verschiedensten Einsatzszenarien. Die Benutzer werden in der Lage sein, ihre Anwendungen hochzuladen und über einen Browser zu bedienen, während der DaaS die Anwendungen in Containern betreibt und eine Proxy-Lösung verwendet. Die Lösung kann von Benutzern selbst betrieben oder als Dienst von Kunden gemietet werden. Im Vergleich zu anderen Desktops und Bereitstellungsmethoden bietet diese Lösung eine hohe Flexibilität bei der Verwendung von Anwendungen, minimiert Datenschutz- und Datensicherheitsprobleme und ermöglicht die Nutzung auf allen Arten von Client-Geräten und Betriebssystemen, da nur ein Browser benötigt wird. Dieses Projekt wird durch die aktuellen Entwicklungen in Unternehmen, Verwaltungen, Schulen, Hochschulen und Forschungseinrichtungen motiviert und findet in diesem Markt Bedarf. [1]

2. Stand der Technik

2.1. Virtualisierung

Virtualisierung bedeutet die Erstellung einer virtuellen Version von etwas, das normalerweise physisch vorhanden ist. Im Bereich der IT-Technologie bezieht sich Virtualisierung auf die Erstellung von virtuellen Ressourcen wie Betriebssystemen, Servern, Speichergeräten, Netzwerkressourcen und Anwendungen.

Virtualisierung ermöglicht es, mehrere virtuelle Ressourcen auf einem physischen Computer oder Server zu erstellen und zu nutzen, indem die Hardware-Ressourcen des physischen Geräts in mehrere virtuellen Maschinen oder Umgebungen aufgeteilt werden. Dies geschieht durch eine Abstraktionsschicht, welche zwischen physischer Hardware und Betriebssystem, bzw. Betriebssystem und Anwendungsprogramm, liegt. Mit der Abstraktion physischer IT-Ressourcen wie Hardware, Software, Speicher und Netzwerkkomponenten will man Ressourcen auf virtueller Ebene bereitstellen und diese flexibel und bedarfsgerecht an verschiedene Abnehmer verteilen. Das wird durch die Orchestrierung realisiert. Dabei werden mehrere Virtualisierungen durch ein Orchestrierungstool automatisiert und verwaltet. Es wird die effizientere Nutzung von Ressourcen, das Ausführen mehrerer Betriebssysteme auf einer Hardware-Plattform und die vereinfachte Verwaltung von Anwendungen und Daten ermöglicht.

Virtualisierung wird oft in Rechenzentren, Cloud-Computing-Umgebungen und bei der Bereitstellung von Software-as-a-Service (SaaS)-Anwendungen eingesetzt. Der Prozess der Provisionierung stellt virtualisierte Hardware bereit und bildet die Grundlage skalierbarer Cloudplattformen wie AWS oder Azure. Es gibt verschiedene Arten von Virtualisierung, zum Beispiel Server-Virtualisierung, Desktop-Virtualisierung, Anwendungs-Virtualisierung und Netzwerk-Virtualisierung.

Zwei spezielle Formen der Virtualisierung sind für diese Arbeit von besonderem Interesse [2]:

1. **Vollvirtualisierung**, ein Spezialfall der Hardware-Virtualisierung: Jeder laufenden Instanz (virtuelle Maschine) wird eigene virtualisierte Hardware zugewiesen
2. **Virtualisierung auf Betriebssystemebene**, ein Spezialfall der Software-Virtualisierung: Jede laufende Instanz (Container) erhält Zugriff auf ein virtualisiertes Betriebssystem mit begrenzten Ressourcen

Als Sonderform sei die Paravirtualisierung genannt. Der Hypervisor stellt im Gegensatz zur Vollvirtualisierung keine eigene virtuelle Hardware-Umgebung, sondern lediglich eine API zur Verfügung, über die die Gastbetriebssysteme direkt auf die physische Hardware des Hosts zugreifen können.Beide Virtualisierungstypen, VMs und Container, werden in den nächsten Abschnitten definiert und anhand bestehender Lösungen miteinander verglichen.

2.2. Virtuelle Maschinen

Eine virtuelle Maschine (VM) ist eine logische Partition der Ressourcen des Host-Systems und kann ein Betriebssystem isoliert von anderen VMs ausführen [3]. Virtuelle Maschinen greifen direkt auf die virtualisierte Hardware des Rechners zurück. Sie werden über einen sog. Hypervisor betrieben, welcher als Schnittstelle zwischen physischem Speicher und virtueller Maschine fungiert. Der Hypervisor läuft für gewöhnlich auf einem Host-Betriebssystem und emuliert ein Gast-Betriebssystem.

Nach Goldberg kann dabei in zwei Arten von Hypervisoren unterschieden werden [4]:

- **Typ-1:** native oder „bare-metal“ Hypervisoren
  (operieren auf der Hardware des Hosts)
- **Typ-2:** gehostete Hypervisoren
  (operieren auf dem Host-Betriebssystem)

Typ-1 Hypervisoren werden von VMware ESXI and The Xen Project implementiert, Typ-2 Hypervisoren von Lösungen wie Oracle VM VirtualBox, VMware Workstation und Kernel-based Virtual Machine (KVM). Bei KVM ist die Unterscheidung nicht immer leicht, da es Charakteristiken von beiden Typen besitzt.

Hypervisoren besitzen folgende grundlegende Funktionalitäten [5]:

- Isolierte Ausführung von VMs
- Emulation und Zugriffskontrolle
- Privilegierte Ausführung von Gast-VMs
- Management der VMs
- Administration der Hypervisorplattform und -software

Moderne Hardware bietet Erweiterungen, um die Implementierung der verschiedenen Funktionen eines Hypervisors zu erleichtern (siehe Intel oder AMD-Prozessoren). Lösungen,

die sich auf Hardware-Erweiterungen stützen, werden oft als Hardwareunterstützte virtuelle Maschinen (HVM) bezeichnet. Viele Plattformen (und Hypervisoren) unterstützen auch das Konzept der verschachtelten Virtualisierung. Bei diesen Systemen sind mehrere Virtualisierungsebenen möglich, d. h. ein Hypervisor wird über einen anderen Hypervisor und nicht direkt über die Hardware ausgeführt. Anwendung findet die hypevisorbasierte Virtualisierung durch virtuelle Prozessmaschinen beim Bereitstellen einer plattformunabhängigen Programmierumgebung wie Java Virtual Machines (JVM), indem sie einzelne Prozesse oder Anwendungen virtualisiert. Ein weiteres Beispiel ist die Wine-Software, mit der Windowsanwendungen unter Linux ausgeführt werden können.

Vorteile von VMs sind die schnelle Bereitstellung von Desktops und ihre Portabilität. Zudem bieten VMs eine verbesserte Datensicherheit. Der Einsatz virtueller Maschinen in der Software-Entwicklung wurde heutzutage weitgehend durch die Container-Virtualisierung verdrängt. Da sich alle auf einem Host laufenden Container ein Betriebssystem teilen, ist diese Art der Virtualisierung deutlich performanter und weniger ressourcenhungrig. Jedoch gibt es immer noch Fälle, bei denen sich das Einrichten einer VM zu Testzwecken lohnt. Etwa dann, wenn Linux-Entwicklung vom Windows-Desktop aus betrieben werden soll. Ein weiterer Vorteil beim Einsatz mancher VM-Programme liegt darin, dass diese den Zugriff auf entfernte Systeme erlauben. Bei der Desktop-Virtualisierung läuft eine VM-Software lokal, während die virtuelle Maschine auf einem entfernten Host ausgeführt wird. Eine Trennung des Betriebssystems in mehrere, voneinander isolierte Systeme ist auch unter Sicherheitsaspekten attraktiv.

Nachteile von VMs sind, dass sie speicherintensiv sind, langsam starten und die Performanz des Gesamtsystems beeinträchtigen können. Es existiert eine Vielzahl an Anbietern virtueller Maschinen wie VirtualBox, Hyper-V, KVM und VMWare. Diese Arbeit konzentriert sich auf die Implementierung mit KVM.

2.2.1. KVM

KVM steht für „Kernel-based Virtual Machine“ und ist eine Virtualisierungstechnologie für Linux. Es ermöglicht die Erstellung und Verwaltung von VMs auf einem Host-System. KVM basiert auf der Virtualisierungserweiterung (Hardware-Virtualisierung) moderner x86-Prozessoren, wie Intel-VT oder AMD-V. Die Geschichte von KVM begann im Jahr 2006 und seitdem ist sie ein Teil der Linux-Kernel-Version geworden. KVM verfügt über Funktionen, wie z. B. die Kombination aus verbesserter Sicherheit (SELinux und sVirt), Live-Migration, Zeitplanung und Ressourcenkontrolle, geringere Latenz und höhere Priorisierung. [6]

KVM ist als Kernelmodul in Linux enthalten und fungiert als Hypervisor, es führt aber keine Emulation durch. Der Quick-Emulator (QEMU) ist der zuständige Emulator der Hardware und enthält die Userspace-Komponenten von KVM. Er unterstützt die Paravirtualisierung für einige Geräte. Die Schnittstelle „/dev/kvm“ richtet den Adressraum der virtuellen Maschine ein und speist die simulierte Eingabe/Ausgabe über QEMU [2], das heißt, KVM arbeitet nicht eigenständig. Es ist eine API, die vom Kernel für den Userspace bereitgestellt wird. Als Linux-Kernel hat KVM weder eine GUI noch eine CLI. Endbenutzer verwenden KVM also in der Regel über QEMU (wo es als Beschleunigungsmethode vorhanden ist). Dadurch laufen die VMs mit fast nativer Geschwindigkeit. Zudem ermöglicht Qemu das Emulieren anderer Prozessorarchitekturen wie PowerPC oder ARM und verschiedener Arten von Laufwerken. Als Virtualisierungsbibliothek kann libvirt verwendet werden, welcher als „Wrapper“ für KVM und QEMU fungiert und APIs wie *„virsh“* oder *„virt-manager“* für die Verwendung durch andere Programme bereitstellt. KVM wird häufig in Rechenzentren und virtualisierten Umgebungen eingesetzt, um virtuelle Infrastrukturen, private Clouds und Hosting-Dienste bereitzustellen. Es bietet eine flexible und leistungsstarke Virtualisierungslösung für verschiedene Anwendungsfälle und unterstützt eine Vielzahl von Betriebssystemen und Anwendungen.

2.3. Container

Bei der containerbasierten Virtualisierung genannt (im Folgenden „Container“) partitioniert ein einziges Betriebssystem die Ressourcen eines Hosts und sorgt für die Isolierung zwischen diesen Partitionen. Das heißt die Container abstrahieren Prozesse und Anwendungen auf Betriebssystemebene, sie laufen auf demselben Betriebssystemkern wie die Hostmaschine [7]. Die Idee der Containervirtualisierung geht maßgeblich auf Pike et al. zurück [8], der „Function Control Groups“ (cgroups) und Linux namespaces als Grundlage aufzeigt. Da Container auf Ebene des Betriebssystems arbeiten, ist keine zusätzliche Schicht zur Verwaltung der Host-Ressourcen erforderlich. Dies führt zu einer geringeren Komplexität, effizienteren Ressourcennutzung und einem geringeren Overhead als bei VMs. Zudem lässt sich eine höhere Dichte an einzelnen virtuellen Instanzen erreichen, da jeder Container geringe Prozessor- und Speicherressourcen verbraucht. Ein Container verpackt eine Anwendung mit all ihren Abhängigkeiten, sodass sie auf jedem Betriebssystem ausgeführt werden kann. Diese Portabilität eignet sich gut für Webanwendungen und Microservices. Die gemeinsame Nutzung des Betriebssystemkerns bedeutet jedoch, dass die gebotene Isolierung nicht so stark ist wie bei

der Vollvirtualisierung. Des Weiteren haben Container einen erhöhten Netzwerk- und E/A-Overhead und eine schlechtere Mehrmandantenfähigkeit bzw. -sicherheit im Vergleich zu Hypervisoren. Laut Casalicchio unterscheidet man bei der Virtualisierung vier Anwendungen: Applikations- und Systemcontainer, Containermanager und Containerorchestrierung. Forschungsschwerpunkte von Containervirtualisierungen finden sich in absteigender Häufigkeit zu den Subgebieten der Orchestrierung, Anwendungen, Security und Performanz [9]. Diese Themen werden im Rahmen dieser Arbeit allerdings nur am Rande betrachtet. Es existieren eine Vielzahl an Containerlösungen. Beispiele dafür sind Linux-VServer, OpenVZ, LXC, Docker und Rocket. Diese Arbeit wird sich auf Docker beschränken.

2.3.1. Docker

Docker ist eine Open-Source-Plattform für die Containerisierung von Anwendungen. Es ermöglicht die Erstellung und Verwaltung von Containern, um Anwendungen und ihre Abhängigkeiten in einer isolierten Umgebung ausführen. Im Gegensatz zu LXC handelt es sich bei Docker um eine High-Level Lösung, die über die Jahre mehrere Abstraktionsschichten erhalten hat, um dem größeren Open-Source-Ökosystem gerecht zu werden. Genau wie LXC und andere Container-Technologien basiert Docker auf den Linux-Kernelfunktionen von cgroups und Namespaces. Cgroups ermöglicht die Zuweisung und Kontrolle von Ressourcen wie CPU, Speicher und Netzwerkbandbreite für jeden Container. Namespaces hingegen isoliert Container von anderen Containern und dem Hostsystem. Darauf aufbauend bietet Docker Tools zum Builden und Packen von Anwendungen in portable Umgebungen. Ab Version 0.9 stellte Docker die Unterstützung für Linux als Standard-Ausführungsumgebung ein und ersetzte diese durch eigene Implementierungen namens libcontainer und schließlich runC. Derzeit sind die beiden wichtigsten Docker-Engine-Komponenten containerd und runC. Des Weiteren besteht die Docker-Architektur aus folgenden Komponenten [10]:

- Docker Engines:
  Eine Client-Server-Anwendung, die auf dem Hostrechner mit den folgenden Komponenten installiert wird:
  (a) Docker-Daemon: läuft auf dem Host, ist für die Erstellung, den Aufbau und die Ausführung von Anwendungen zuständig
  (b) Eine REST-API wird für die Kommunikation mit dem Docker-Daemon verwendet.
  (c) Ein Client sendet über das Terminal eine Anfrage an den Docker-Daemon, um auf die Operationen zuzugreifen. [11]

- Images:

  Schreibgeschütztes Template zur Erstellung von Containern. Im Allgemeinen bilden Basis-Images die Grundlage für jedes Image. Zudem ist es möglich, eine Docker-Datei zu erstellen, die alle Anweisungen zur Erstellung eines Docker-Images enthält. Wenn der Befehl „docker build“ im Terminal ausgeführt wird, wird das Image mit allen Abhängigkeiten erstellt, die in der Docker-Datei aufgeführt sind. [11]
- Container

  Eine lauffähige Instanz eines Docker Images. Durch den Befehl „docker run“ wird ein Container erstellt und gestartet. Ein Container kann über Befehle wir start, stop und pause über den Docker-Daemon mit einer eindeutigen ID angesprochen werden.
- Registrierung

  Eine private und öffentliche Registrierung von Docker Images
- Services

  Ein Scheduling-Service namens Swarm, der eine Multi-Host- und Multi-Container-Bereitstellung ermöglicht. Swarm wurde in der Version 1.12 eingeführt

Abbildung 1 verschafft ein Überblick über die Docker-Komponenten:

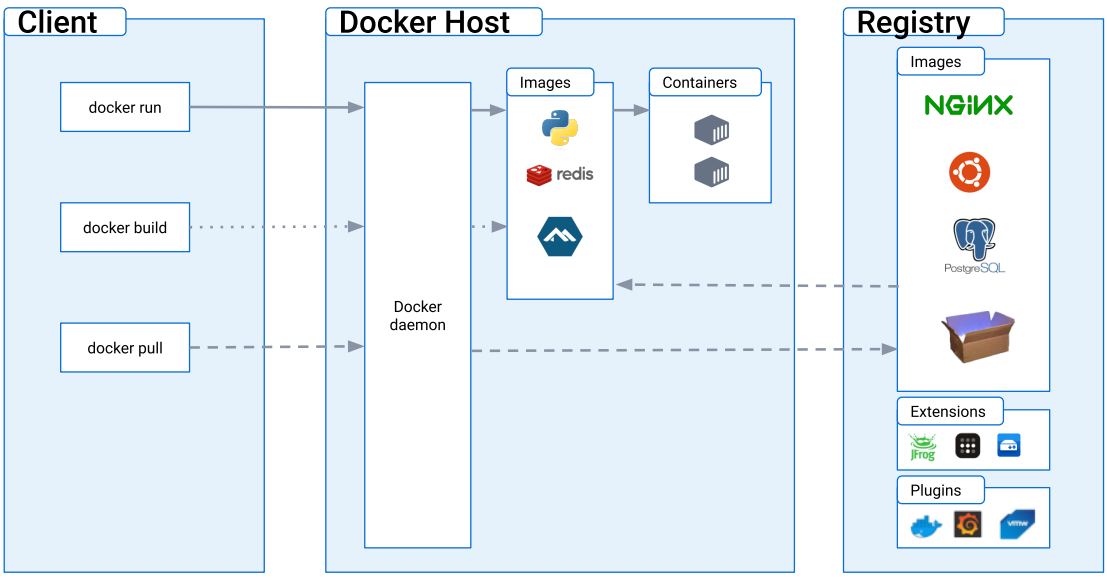


Abbildung 1: Docker Überblick, Quelle: [10]

2.4. Application Programming Interface

Ein Application Programming Interface (API) ist eine Schnittstelle, die es verschiedenen Anwendungen ermöglicht, miteinander zu kommunizieren und Daten auszutauschen. APIs werden häufig von Software-Entwicklern genutzt, um Anwendungen oder Dienste miteinander zu verbinden, so dass sie nahtlos zusammenarbeiten können.

APIs stellen in der Regel eine Reihe von Programmierschnittstellen und Bibliotheken zur Verfügung, die es Entwicklern ermöglichen, Anwendungen und Dienste auf einfache Weise zu integrieren. Die meisten APIs sind standardisiert, so dass Entwickler sie leicht verstehen und nutzen können.

Es gibt verschiedene Arten von APIs, die je nach Anwendungsfall und Zweck eingesetzt werden. Einige der wichtigsten Arten von APIs sind:

- Web APIs: Auch bekannt als HTTP-APIs oder RESTful APIs, sind sie am häufigsten und ermöglichen es, auf Webressourcen wie Datenbanken, Dateien, E-Mails und andere Webdienste zuzugreifen.
- Bibliotheks-APIs: Diese Art von API ist in eine Bibliothek eingebettet, die Entwickler verwenden können, um auf bestimmte Funktionen und Datenstrukturen innerhalb der Bibliothek zuzugreifen.
- Betriebssystem-APIs: Betriebssystem-APIs bieten Entwicklern eine Schnittstelle, um auf Funktionen und Ressourcen des Betriebssystems zuzugreifen, wie beispielsweise das Dateisystem, das Netzwerk oder die Benutzeroberfläche.
- Hardware-APIs: Diese APIs ermöglichen es Entwicklern, auf die Funktionen von Hardwarekomponenten wie Kameras, Sensoren oder GPS zuzugreifen.
- Daten-APIs: Daten-APIs ermöglichen den Austausch von Daten zwischen verschiedenen Anwendungen und Systemen.
- Messaging-APIs: Messaging-APIs ermöglichen die Kommunikation zwischen Anwendungen oder Systemen über Nachrichtenprotokolle oder WebSocket.
- Media-APIs: Media-APIs ermöglichen die Integration von Audio-, Video- und Grafikfunktionen in Anwendungen und Dienste.

Diese Arbeit wird als Web API realisiert, daher bedarf es einer genaueren Einordnung der zugrundliegenden Technologien. Das ist zum einen das REST-Prinzip und zum anderen das JSON-Dateiformat, zudem wird auf das HTTPS-Protokoll eingegangen.

2.4.1. Representational State Transfer (REST)

REST (Representational State Transfer) ist ein Architekturstil für die Entwicklung von Webanwendungen, der von Roy Fielding in seiner Dissertation aus dem Jahr 2000 vorgestellt wurde. REST definiert ein Set von Designprinzipien, die eine einfache, skalierbare und leichtgewichtige Interaktion zwischen Client und Server ermöglichen.

REST basiert auf der Verwendung von HTTP-Protokollmethoden (wie GET, POST, PUT, DELETE) und Ressourcen-URLs, um auf Ressourcen in einer Webanwendung zuzugreifen. Der Client sendet eine Anfrage an den Server, die aus einer HTTP-Methode, einer Ressourcen-URL und optionalen Parametern besteht. Der Server antwortet mit einer HTTP-Antwort, die den Status der Anfrage und gegebenenfalls Daten zurückgibt.

Ein weiteres wichtiges Konzept in REST ist die Verwendung von eindeutigen Ressourcenidentifikatoren (URI) für jede Ressource in der Anwendung, um eine klare und konsistente Struktur für die Anfragen und Antworten zu schaffen. Durch die Verwendung dieses Architekturstils können Webanwendungen flexibler und leichter zu skalieren sein, da sie weniger Zustand speichern und die Verarbeitung auf mehrere Server verteilt werden kann.

Der REST-Architekturstil zeichnet sich durch folgende Eigenschaften aus [12]:

- Einheitliche Schnittstelle
  Durch folgende Einschränkungen wird eine einheitliche Schnittstelle bestimmt:
  - Identifikation von Ressourcen
  - Manipulation von Ressourcen durch Darstellung
  - Selbstschreibenden Nachrichten
  - Hypermedia als Motor des Anwendungszustandes
- Client-Server
  Client- und Serverkomponenten arbeiten unabhängig voneinander.
- Zustandslosigkeit
  Jede Anfrage des Clients muss eigenständig funktionieren und Informationen über den Sitzungsstatus enthalten.
- Cachefähig
  Explizite Kennzeichnung der Antwort nach Cachefähigkeit, um dem Client mitzuteilen, ob dieser Antworten für redundante Abfragen cachen kann.
- Mehrschichtiges System

Aufbau einer Architektur aus hierarchischen Schichten.

- Code-on-Demand (optional)
  Möglichkeit des Herunterladens oder Ausführens von Code in Applets oder Skripten

2.4.2. JavaScript Object Notation (JSON)

JSON (JavaScript Object Notation) ist ein leichtgewichtiges Datenaustauschformat, das auf der Syntax von JavsScript-Objekten basiert. Es wird häufig in Webanwendungen als Alternative zu XML verwendet, da es einfacher zu lesen und zu schreiben ist und eine geringere Datenübertragungsgröße aufweist.

JSON verwendet eine Textformatierung, um Daten zu speichern und zu übertragen. Es besteht aus einer Sammlung von Schlüssel-Werte-Paaren, die in geschweiften Klammern { } eingeschlossen sind. Die Schlüssel sind Strings, die den Namen der Datenfelder darstellen, während die Werte die eigentlichen Daten enthalten. Die Daten können primitive Datentypen wie Strings, Zahlen und Booleans oder komplexe Datentypen wie Arrays und verschachtelte Objekte sein [13].

Abbildung 2 zeigt ein JSON-Objekt eines Docker-Containers:


```
{
    "Id": "658c167a479d",
    "Created": "2023-04-11T09:46:31.273724508Z",
    "Path": "docker-entrypoint.sh",
    "Args": [
      "redis-server"
    ],
    "State": {
      "Status": "running"
    }
}
```


Abbildung 2: JSON-Objekt, Quelle: eigenes Bild

JSON wird häufig in RESTful Web Services verwendet, um Daten zwischen Client und Server auszutauschen. Die Daten können über HTTP-Anfragen übertragen werden, indem sie als Payload im Body der Anfrage oder als Antwort zurückgegeben werden. JSON wird auch in der Webprogrammierung verwendet, um Daten im Browser zu verarbeiten, da es leicht in JavaScript-Objekte umgewandelt werden kann.

2.4.3. Hypertext Transfer Protocol (HTTP)

HTTP steht für „Hypertext Transfer Protocol“. Es ist ein Protokoll, das zur Übertragung von Daten über das Internet verwendet wird. HTTP ermöglicht die Kommunikation zwischen Webservern und Webbrowsern und bildet die Grundlage für die Übertragung von Webseiten, Bildern, Videos und anderen Inhalten des World Wide Web. Es ist ein zustandsloses Protokoll, das bedeutet, das jeder Request (Anfrage) an den Server unabhängig von vorherigen Anfragen ist und keine Verbindung zwischen den Anfragen aufrechterhalten wird. HTTP basiert auf der Client-Server-Architektur und ist das am häufigsten verwendete Protokoll im Web.

Im Gegensatz zum unverschlüsselten HTTP verwendet HTTPS eine Verschlüsselungsschicht, um die Daten während der Übertragung zu schützen. Als Protokoll dafür wird TLS/SSL verwendet. Die Verschlüsselung erfolgt durch den Einsatz von digitalen Zertifikaten, die von zertifizierten Zertifizierungsstellen (CAs) ausgestellt werden. Diese Zertifikate werden verwendet, um die Identität des Servers zu authentifizieren und die Datenübertragung zu verschlüsseln. Durch die Verwendung von HTTPS wird sichergestellt, dass die übertragenen Daten vor unautorisierten Zugriffen und Manipulationen geschützt sind. Es wird insbesondere bei der Übertragung sensibler Daten wie Passwörtern, Kreditkarteninformationen oder persönlichen Daten empfohlen. Viele Websites setzen heutzutage auf HTTPS als Standardprotokoll, um die Sicherheit der Nutzerdaten zu erhöhen und die Privatsphäre der Nutzer zu schützen. [14]

2.5. Bestehende APIs zur Kommunikation mit Containern und VMs

Im Folgenden Abschnitt werden existierende APIs erläutert, welche eine Kompatibilität zu Docker oder KVM aufweisen. Diese werden zum einen nach relevanten Kriterien verglichen und Funktionslücken für den speziellen Use-Case dieser Arbeit aufgezeigt. Relevante Kriterien sind der Funktionsumfang der API, die Anforderungen an die Entwicklungsumgebung und die Kompatibilität zu Docker und KVM. Zum anderen werden die APIs auf Tauglichkeit zur Verwendung als SDK/Bibliothek für die Entwicklung dieser Arbeit untersucht.

2.5.1. Docker Engine API

Die Docker Engine API ist eine RESTful API, die es ermöglicht, mit der Docker-Engine zu interagieren und Docker-Container zu verwalten. Die Docker-Engine API bietet eine Schnittstelle für Entwickler und Administratoren, um Docker-Container, Images, Netzwerke und Volumes über HTTP-Anfragen zu steuern.

Mit der Docker-Engine API kann man Docker-Container erstellen, starten, stoppen und löschen sowie Docker-Images und Netzwerke verwalten. Sie können auch Docker-Events überwachen und Docker-Container-Logs abrufen. Die API ermöglicht zudem die Integration von Docker-Containern in andere Anwendungen und Systeme.

Die Docker-Engine API ist eine wichtige Komponente in der Docker Infrastruktur und wird von vielen Werkzeugen und Plattformen genutzt, die auf Docker aufbauen. Zum Beispiel verwenden Container-Orchestrierungstools wie Kubernetes und Docker Swarms die Docker-Engine API, um Docker-Container zu orchestrieren und zu verwalten. Die meisten Befehle des Clients werden direkt auf API-Endpunkte abgebildet (z. B. ist docker ps GET /containers/json) [15].

Die Docker Engine API bietet zudem eine SDK für Python an. Die im Hintergrund laufende Docker-Engine lässt sich in Python ansprechen und Operationen an Containern, Images und Netzwerken lassen sich in Python durchführen [16].

2.5.2. Portainer API

Portainer ist eine Open-Source-Verwaltungsplattform für Docker-Container. Portainer verwaltet Container und Container-Cluster über eine grafische Oberfläche, ohne Docker-Befehle auf der Kommandozeile verwenden zu müssen. Die Plattform bietet eine intuitive Benutzeroberfläche zur Verwaltung von Containern, Volumes und Netzwerken sowie zum Starten, Stoppen und Neustarten von Containern.

Portainer ist eine Docker-Anwendung, die auf jedem Docker-Host oder in einem Docker-Cluster laufen kann. Sie bietet eine zentralisierte Verwaltungsschnittstelle, die es Benutzern ermöglicht, mehrere Docker-Hosts und -Cluster zu verwalten, ohne auf jedem Host eine separate Installation durchführen zu müssen.

Portainer unterstützt auch die Verwaltung von Swarms-Clustern und Kubernetes-Clustern, was es zu einer flexiblen Verwaltungsplattform für die Container-Orchestrierung macht. Es bietet außerdem erweiterte Funktionen wie die Möglichkeit, benutzerdefinierte Container-Images zu erstellen und zu verwalten, die Überwachung der Container-Performanz und des Zustands sowie die Integration mit LDAP- und OAuth-Authentifizierungsdiensten [17].

2.5.3. Libvirt API

Die libvirt API ist eine Open-Source-Bibliothek, die eine standardisierte Programmierschnittstelle für die Verwaltung von virtuellen Maschinen und virtuellen Umgebungen bereitstellt. Die Bibliothek bietet eine plattformübergreifende Abstraktionsschicht, die es Entwicklern ermöglicht, virtuelle Umgebungen zu erstellen und zu verwalten, ohne sich um die spezifischen Unterschiede zwischen den Hypervisoren kümmern zu müssen. Aus Sicht des Endbenutzers bietet es CLI-Tools wie virsh, virt-manager und virt-install.

Die libvirt API unterstützt eine Vielzahl von Virtualisierungstechnologien wie Qemu/kvm, Xen, VMware und Hyper-V. Sie bietet eine Vielzahl von Funktionen wie das Erstellen, Starten, Stoppen, Pausieren und Löschen von virtuellen Maschinen, sowie das Konfigurieren von Ressourcen wie CPU, Speicher und Netzwerk. Darüber hinaus bietet die Bibliothek Funktionen zur Verwaltung von virtuellen Netzwerken, Storage-Pools, Snapshots und vielen anderen Aspekten der Virtualisierung [18].

Die libvirt API wird von vielen Virtualisierungs-Management-Tools, Cloud-Service-Providern und Hosting-Unternehmen genutzt, um ihre Virtualisierungslösungen zu verwalten. Die libvirt API ist in der Programmiersprache Python als „libvirt-python“ Modul verfügbar [19].

2.5.4. Proxmox VE API

Proxmox VE ist eine Open-Source-Software-Plattform, die es ermöglicht, virtuelle Maschinen und Container zu erstellen, zu verwalten und bereitzustellen. Die Proxmox VE API ist eine RESTful API, die es Entwicklern erlaubt, auf die Funktionen von Proxmox VE zuzugreifen und diese zu automatisieren. Die API bietet eine breite Palette von Funktionen, einschließlich der Möglichkeit, virtuelle Maschinen und Container zu erstellen, zu starten, zu stoppen und zu löschen. Darüber hinaus ermöglicht die API die Verwaltung von Speicher und Netzwerk sowie die Überwachung der Gesamtleistung des Proxmox VE-Umgebung. Allerdings arbeitet Proxmox ausschließlich mit VMs auf Basis von KVM/Qemu und LXC- Containern.

Die Proxmox VE API kann über verschiedene Programmiersprachen wie Python, Perl und Ruby verwendet werden und bietet eine leistungsstarke und flexible Möglichkeit, Proxmox VE zu automatisieren und zu verwalten [20].

2.5.5. Vergleich bestehender APIs

Im Vergleich der APIs lässt sich zusammenfassen, dass die Docker Engine API und die libvirt API am nahesten an den ursprünglichen Befehlen der Daemons laufen und diese nachahmen. Die Docker API ist laut Docker „die API, die der Docker-Client für die Kommunikation mit der Engine verwendet, sodass alles, was der Docker-Client tun kann, mit der API getan werden kann“ [15].

Portainer setzt an der Docker Engine an auf und erweitert diesem um Funktionalitäten wie Orchestrierung und Sicherheit und damit um eine Abstraktionsebene zu den Docker-Diensten. Die libvirt-API lehnt sich ebenso direkt an die Qemu/KVM-Dienste an und bietet durch *„virsh“*, *„virt-manager“* und *„virt-install“* ein umfangreiches Toolkit zum Verwalten von Ressourcen wie VMs, Netzwerken und Snapshots, ermöglicht dabei eine granulare Konfiguration von Ressourcen wie CPU und Speicher. Es stellte sich heraus, dass libvirt zudem eine gute Dokumentation besitzt und auf gewachsen Erfahrung vieler Entwickler zurückgreifen kann. Die Proxmox API arbeitet ebenfalls mit Linux-Virtualisierungen Qemu/KVM und LXC und bietet, wenn es um das grundlegende Verwalten dieser Technologien geht, keinen Mehrwert gegenüber libvirt. Folgende Tabelle stellt die besprochene API-Lösungen gegenüber:

Tabelle 1: Vergleich von API-Lösungen

Dienst	**Kategorie**	**Virtualisierung**	**Dokumentation**	**SDK**
Docker Engine API	Docker Service	Container (Docker)	Umfänglich	Python (Docker-py), Go
Portainer	Containerorchestrierung	Container (Docker)	Ausreichend	Python / Go SDK
Libvirt	Virtualisierungsschnittstelle von Linux	Qemu/KVM, LXC, Xen, VMWare ESX	Umfänglich	C, Python, Perl, Go, …
Proxmox VE API	Orchestrierung von VMs und Containern	Qemu/KVM, LXC	Knapp	Python, Java, PHP, C#, Go

Quelle: Eigene Darstellung

3. Design

Im folgenden Abschnitt wird das Design der Anwendung näher erläutert und auf die Konzeption der Anwendung eingegangen. Zudem wird der Einsatz von Technologien und Bibliotheken, welche bereits im vorherigen Kapitel beleuchtet wurden, begründet und in Bezug zu den Anforderungen an die zu entwickelnde API gebracht. Dazu werden mögliche Lösungen beschrieben und auf Grundlage dessen eine eigene Lösung designt. Die funktionalen und nicht funktionale Anforderungen an die zu entwickelnde API werden aus Kapitel 1 aufgegriffen und in ein Design ausgearbeitet. Dabei sollen die Best-Practice Prinzipien einer REST-API, ein einheitliches Interface und Interoperabilität, umgesetzt werden.

Bei der API soll es sich um eine private Implementierung handeln, das heißt sie wird nur innerhalb eines Unternehmens verwendet werden und wird dementsprechend nicht ins Internet kommunizieren. Die Lösung soll in die Unternehmensarchitektur integriert und beispielsweise an existierende Datenbanken und Netzwerkschnittstellen angebunden werden, um in ersterem Beispiel Nutzerdaten zu beziehen. Entsprechend dieses Ansatzes wird eine rudimentäre Sicherheitsfunktionalität implementiert, da keine Bedrohung innerhalb der Datenübermittlung erwartet wird, welche Anmeldedaten abgreifen oder Szenarien wie Session Hijacking auslösen könnten.

Das Konzept der API ist es, diese weitestgehend modular zu gestalten. Das heißt, dass zum Bespiel die Methoden der Docker- bzw. libvirt-Klasse Typen und Objekte zurückgibt, die ohne entsprechendes Docker- bzw. Libvirt-Modul verständlich sind. So soll ermöglicht werden, entsprechende Klasse auch in anderen Anwendungen einsetzen zu können, um z. B. eine CLI-API zu schreiben. Es soll als eine Art Werkzeugkasten verstanden werden.

Die REST-Schnittstelle soll Container und VMs insoweit abstrahieren, dass sie als Bezeichnung *„/resource“* in der URL der API-Endpunkte zu sehen sind. Das heißt es wird für Funktionen, die sich für Docker Container und VMs überschneiden (list, info, start, stop, restart, delete) einen einzigen Endpunkt geben.

Folgende Best-Practice Prinzipien einer REST-API können ausgemacht und auf diese Arbeit angewandt werden [21]:

1. JSON als Datenformat für Anfragen und Antworten
2. Nomen anstatt Verben in URL-Endpunkten verwenden

REST arbeitet mit Ressourcen und deren „Collections“ (Sammlungen von Ressourcen. Die Art des Zugriffs wird über HTTP-Methoden unterschieden. Das heißt die URL enthält nur den Namen der Ressourcen und ggf. die spezifische ID: *„/resources/123“*

3. Name von Collections im Plural
4. Verschachtelung von Ressourcen
5. Fehlerbehandlung mit verständlicher Fehlerbeschreibung und HTTP-Codes
6. Filterung, Sortierung und Auswahl von Feldern

Abbildung 3 zeigt den schematischen Aufbau der API und der beteiligten Komponente:

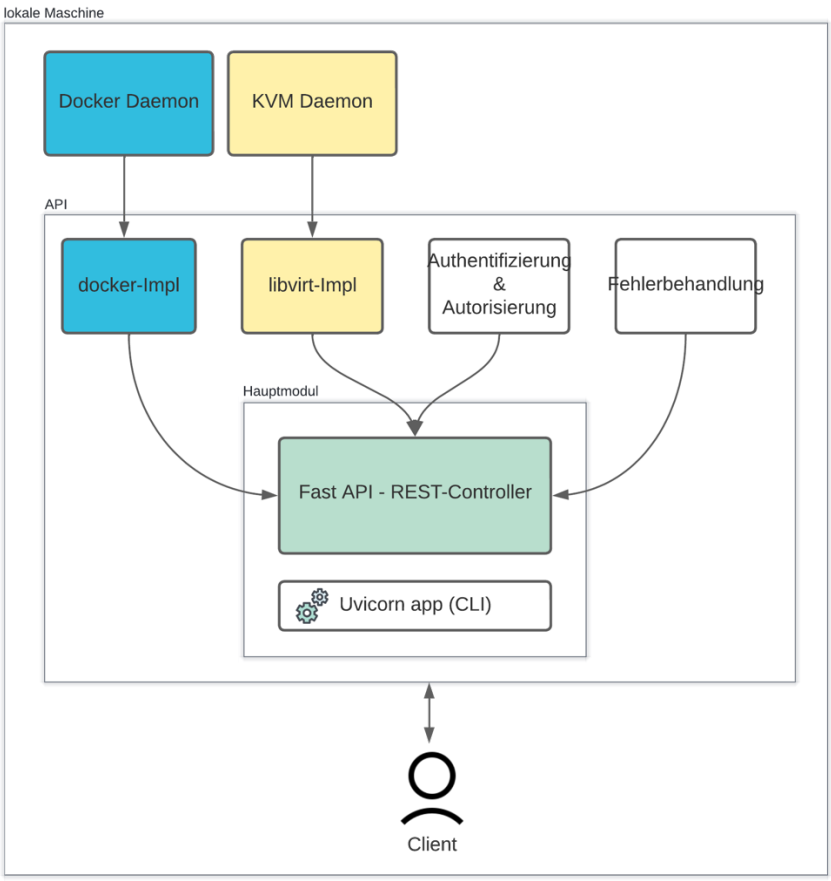


Abbildung 3: Schema der Anwendung, Quelle: eigenes Bild

3.1. APIs zur Kommunikation mit Docker und KVM

Sowohl Portainer als auch Proxmox bieten keine gut dokumentierte und umfängliche SDK in Python an. Proxmox ist zudem nicht mit Docker kompatibel. Es lässt sich sagen, dass der Funktionsumfang der meisten APIs über die einfache Verwaltung von Containern und

virtuellen Maschinen hinausgeht. Für diese Arbeit wird eher eine leichtgewichtigere Lösung bevorzugt. Zudem vereinigt keine der genannten APIs die Kommunikation mit Docker Containern und KVM-Maschinen. Die Docker Engine API setzt direkt auf den Docker Daemon der Hostmaschine auf und „wrappt“ die bekannten CLI-Befehle als SDK in Python, was die Entwicklung einfacher und intuitiver gestaltet. Die libvirt API besitzt ebenfalls eine gut dokumentierte Python-SDK.

Deshalb kommt dieser Vergleich zu dem Schluss, dass eine eigene Lösung auf Basis der Docker Engine- und der libvirt-API in Python entwickelt wird. Die gewählten Lösungen bieten nichtsdestotrotz ein Entwicklungspotential z. B. bezüglich der Implementierung von Netzwerken und der Orchestrierung von VMs und Containern.

3.2. REST-Schnittstelle

Als Framework wurde FastAPI gewählt. Es macht die Implementierung einer REST-Schnittstelle möglich, mit welcher in der Entwicklung lokal kommuniziert wurde. Zusätzlich bietet das Framework eine grafische Oberfläche mit einer Übersicht aller API-Endpunkten und entsprechender REST-Methode. Durch diese Oberfläche kann man die API-Endpunkten ansprechen und damit testen. Dort sind die formalen Einschränkungen an die Anfrage festgehalten. Zudem findet sich dort das festgelegte Schema des Bodys wieder. Die grafische Oberfläche dient der API daher gleichzeitig als Dokumentation. Dies macht die API übersichtlicher für spätere Entwickler und den Einsatz in Produktivumgebungen. Das Framework kommt zudem mit einigen Funktionen an Board wie Authentifizierung, Proxy-Einstellungen und die Möglichkeit, eine Datenbank anzubinden. [22]

3.3. Docker Container

Die API soll Funktionen enthalten, welche Docker Container und zum Teil Docker Images ansprechen. Die Python-Implementierung wird mit dem Docker-Daemon des Hostsystems kommunizieren. Die Möglichkeit zum Erstellen von Images soll in dieser Ausarbeitung nicht berücksichtigt werden. Es wird davon ausgegangen, dass zu verwendende Images bereits existieren. Des Weiteren sollen bei der Erstellung eines Containers nicht alle Konfigurationsmöglichkeiten berücksichtigt werden.

3.4. Libvirt VM

Genau wie bei Docker sollen die Funktionen zum Steuern der VMs in ein gesondertes Python-Modul ausgelagert werden, um das Designprinzip der Modularität zu erfüllen. Die Handhabung von virtuellen Maschinen gestaltet sich im Gegensatz zu Docker umständlicher. Es bestehen vielfältige Möglichkeiten eine VM zu konfigurieren, da die Interaktion mit der Hardware vielschichtiger ist als bei Docker Containern. Zum einen unterscheidet libvirt eine VM in persistent und transient. Transiente VMs existieren nur bis zum Herunterfahren der VM oder dem Neustart des Hostsystems. Persistente VMs existieren unbegrenzt. Letztere werden in dieser Arbeit beim Erstellen von VMs implementiert.

Im Folgenden werden mögliche Zustände einer VM in libvirt erläutert:

- Undefiniert: Ist der Grundzustand einer VM. Dieser Zustand soll in der API nicht abgedeckt werden.
- Definiert oder gestoppt: Eine VM, nachdem sie heruntergefahren wurde.
- Laufend: VM läuft aktiv auf dem Hypervisor des Hosts.
- Pausiert: Die Ausführung der VM auf dem Hypervisor wurde ausgesetzt und ihr Zustand vorübergehend gespeichert, bis die VM wieder gestartet wird. Dabei verbraucht die VM zwar RAM-Speicher, aber keine Prozessorressourcen. Die VM hat keine Kenntnis darüber, ob sie angehalten wurde oder nicht. Dieser Zustand soll durch den „stop“-Endpunkt der API abgedeckt werden.
- Gespeichert: Ähnlich wie der angehaltene Zustand, aber der Zustand der VM wird in einem dauerhaften Speicher abgelegt. Auch in diesem Zustand kann die VM wiederhergestellt werden, ohne dass sie merkt, dass Zeit vergangen ist.

3.4.1. Erstellen einer virtuellen Maschine

XML wird als Dateiformat zum Speichern der Konfigurationen aller Elemente in libvirt verwendet, einschließlich VMs, Netzwerken, Speichern und Snapshots. Beim Erstellen von VMs werden RAM und CPU-Ressourcen zugeteilt, sowie Speichervolumen und Netzwerkschnittstellen. Für diese API ist vorgesehen, dass der Benutzer beim Erstellen einer VM bestimmte Spezifikationen wie RAM und CPUs selbst definieren soll. Zudem müssen bestimmte Pfade, wie die der Installationsdatei, übergeben werden. Die Konfiguration von Speichergeräten ist hingegen definiert und beschränkt sich auf die Erstellung einer qcow2-Festplatte über „vda“. Dieser Speichertyp verwendet als Treiber „virtio“ und implementiert

eine Paravirtualisierung, durch welche ein direkter Zugriff auf die Hardware durch sogenannte Hypercalls gewährleistet wird. Dieser Ansatz ist performanter gegenüber der Verwendung von „hda“-Treibern. Im Gegensatz zu ersteren, den paravirtualisierenden Treibern, emuliert ein „hda“-Treiber die Speichergeräte und Systemaufrufe des Hostsystems und erhöht somit den Verwaltungsaufwand, was gerade bei einer steigenden Anzahl virtueller Maschinen auf einem Host als negativ zu bewerten ist. Auch die Netzwerkschnittstelle ist fest definiert und soll eine Standardeinstellung bereitstellen, was in der Regel für die meisten emulierten Betriebssysteme reicht.

3.4.2. Herunterfahren/Neustarten einer VM

Eine VM reagiert nicht zuverlässig auf das Herunterfahren von außen, da der Befehl zunächst an den Gast weitergereicht werden muss. Der Befehl zum Neustart einer VM verhält sich ebenso. Libvirt sendet ein ACPI-Kommando zu Herunterfahren/Neustarten der VM an die VM. Wenn diese den Befehl nicht verstehen kann, passiert nichts. Zuerst muss auf der VM die Software ACPID installiert und gestartet werden. Um dem Client trotzdem eine Möglichkeit zum Herunterfahren der VM zu geben, soll ein zwanghaftes Herunterfahren der VM implementiert werden.

3.4.3. Snapshots

Zusätzlich zu den ursprünglichen Befehlen soll eine Funktion zur Erstellung von Snapshots der VMs implementiert werden. Ein Snapshot ist eine Ansicht des Betriebssystems einer virtuellen Maschine und all ihrer Anwendungen zu einem bestimmten Zeitpunkt. Ein wiederherstellbarer Snapshot einer virtuellen Maschine ist eine grundlegende Funktion der Virtualisierungslandschaft. Snapshots ermöglichen es den Benutzern, den Zustand ihrer virtuellen Maschine zu einem bestimmten Zeitpunkt zu speichern und zu diesem Zustand zurückzukehren. Snapshots werden im Rahmen dieser Arbeit vor allem aus dem Grund implementiert, da die Methoden zum Speichern/Herstellen von VMs nur den Zustand des Arbeitsspeichers speichern, jedoch nicht den des Plattenspeichers. Wenn der Gast wiederhergestellt wird, muss sich der zugrundeliegende Gastspeicher also genau in demselben Zustand befinden wie beim ersten Speichern des Gastes. Das bedeutet das ein Gast nur einmal aus demselben gespeicherten Zustand wiederhergestellt werden kann. Damit ein Gast mehrmals aus demselben gespeicherten Zustand wiederhergestellt werden kann, muss ein Schnappschuss des Gastspeichers zum Zeitpunkt der Speicherung erstellt werden und bei Wiederherstellung

explizit auf diesen Speicherschnappschuss zurückgegriffen werden. Des Weiteren soll die Möglichkeit bestehen, erstellte Snapshots zu löschen [19].

3.5. Sicherheitsfunktionen

3.5.1. Authentifizierung und Autorisierung

Die Authentifizierung und Autorisierung eines Benutzers (Clients) kann auf mehreren Wegen geschehen. Erstens über die HTTP-Basisauthentifzierung. Sie basiert auf Base64-Codierung und verwendet HTTP-Header zur Authentifizierung. Die zweite Methode ist die eines API-Schlüssels. Sie eignen sich für Anmeldungen, bei denen viele Benutzer Zugang benötigen, zudem sind sie sicherer und einfacher zu handhaben. Drittens gibt es die Methode OAuth mit OpenID. OAuth ist für die Autorisierung und OpenID für die Authentifizierung zuständig. Seit OAuth2 ist die Authentifizierung des Benutzers über einen Drittanbieter möglich.

Der Ansatz des API-Schlüssels findet in dieser Arbeit Verwendung. Dazu werden die eingebauten Plugins von FastAPI verwendet, um eine proprietäre Authentifizierung zu gewährleisten. Dazu wird ein JSON Web Token (JWT) genutzt. Bei einem JWT-Token handelt es sich um einen Standard, mit dem sich ein JSON-Objekt in eine lange, dichte Zeichenkette ohne Leerzeichen kodieren lässt. Der Token ist nicht verschlüsselt, so dass jeder die Informationen aus dem Inhalt wiederherstellen kann. Aber er ist signiert, das heißt, der Ursprung des Tokens kann verifiziert werden.

Eine Autorisierung soll ebenfalls realisiert und durch OAuth-Scopes implementiert werden. Diese Scopes sind weniger als Rollen zu verstehen, sondern eher als Geltungsbereiche, welche Bereiche bzw. Funktionalitäten einer Anwendung abgrenzen. [22]

Eine Datenbank ist in der ersten Konzeption im Zuge dieser Arbeit nicht vorgesehen, kann in nächste Entwicklungsschritte jedoch problemlos über das FastAPI-Framework integriert werden. Dementsprechend werden Nutzerdaten in dem Code der Anwendung gespeichert. Das Passwort soll als Hash-Wert soll als Hash-Wert hinterlegt und über den Hashingalgorithmus *„bcrypt“* codiert werden.

3.5.2. CSRF-Sicherheit

Cross Site Request Forgery (CSRF), auch bekannt als Ein-Klick-Angriff oder Session Riding, ist ein Angriff auf eine Webanwendung. Er tritt auf, wenn eine bösartige Website, eine E-Mail oder ein Webforum den Webbrowser eines Opfers veranlasst, eine unerwünschte Aktion auf einer vertrauenswürdigen Website durchzuführen.

Aufgrund der zustandslosen Natur von HTTP kann die Sitzungsverwaltung über Cookies erfolgen. Ein Browser kann Statusinformationen in einem Cookie speichern. Cookies werden in den heutigen Webanwendungen in der Regel als Authentifizierungsmethode verwendet. Jede HTTP-Anfrage, die denselben Cookie-Wert der jeweiligen Webanwendung enthält, gilt als authentifiziert. Es wird bei jeder Anfrage an die Webanwendung zurückgeschickt, ohne dass der Benutzer eingreifen muss [23]. In der Implementierung dieser API ist keine Speicherung von Sitzungsinformationen in Cookies vorgesehen. Aus diesem Grund kann auf eine Sicherheitsfunktionalität gegenüber CSRF-Attacken verzichtet werden.

3.5.3. Cross-Origin Resource Sharing (CORS)

Es soll eine Cross Origin Resource Sharing(CORS) Handhabung implementiert werden. CORS ist ein Mechanismus, der es ermöglicht, dass eine Web-Anwendungen im Browser auf Ressourcen einer anderen Domain zugreifen kann. Zugriffe dieser Art werden normalerweise vom Browser durch die Same-Origin-Policy (SOP) untersagt. CORS ist ein Kompromiss zugunsten größerer Flexibilität im Internet unter Berücksichtigung möglichst hoher Sicherheitsmaßnahmen. Mit der Verwendung von CORS ist es Webentwicklern möglich, normale XMLHttpRequests bzw. die JavaScript Fetch API zu benutzen. Dies ist von Bedeutung, wenn ein Browser über JavaScript mit der API kommunizieren will. Daher wird diese Funktionalität eingebaut.

3.6. Ausnahmebehandlung

Die Idee hinter einem Excepection-Handler ist es, eigene Ausnahmen zu formulieren, welche die passenden Fehlerbeschreibungen enthalten. Dadurch sollen mehrfach vorkommende Fehler aufgefangen und behandelt werden. Das sind Fehler wie ein Abbruch der Verbindung zu den Docker- bzw. libvirt-Diensten oder ein Fehler des Benutzers, wenn eine aktive Ressource angesprochen werden soll, diese aber bereits inaktiv ist. Die genaue Implementierung dieser und vorangegangener Designentscheidungen werden in dem nächsten Kapitel dargelegt.

4. Implementierung

In diesem Abschnitt wird auf die Implementierung eingegangen. Dazu werden die wichtigsten Entwicklungsschritte aufgezeigt und auf entstandene Probleme eingegangen. Der Quellcode zu dieser Implementierung ist im Internet unter https://github.com/DanB1999/Virtualization_API abrufbar und findet sich im Anhang dieser Arbeit wieder.

4.1. Installation

Die API wurde auf einem Linux-Betriebssystem entwickelt, genauer gesagt auf der Distribution Ubuntu 22.04.2 LTS. Im ersten Schritt musste Python auf dem Betriebssystem installiert werden [24]. Als Entwicklungsumgebung wurde Visual Studio Code genutzt.

Zunächst musste sichergestellt werden, dass die Docker Engine auf dem System installiert ist. Der Installationsprozess gestaltete sich unter Ubuntu etwas umständlicher. Um KVM bzw. QEMU nutzen zu können, mussten mithilfe des apt – Paketverwaltungssystems folgende Pakete installiert werden: „bridge-utils“, „cpu-checker“, „libvirt-clients“, „libvirt-daemon“, „qemu“ und „qemu-kvm“. Mit *„kvm-ok“* konnte getestet werden, ob kvm zur Verfügung steht. Anschließend wurden folgende Python-Module über den Befehl *„pip install“* installiert: „fastapi“, „passlib[bcrypt]“, „docker“ und „libvirt“. Daraufhin wurde mit der Entwicklung der Docker-Komponente begonnen.

4.2. REST-Controller

Folgender Abschnitt beschreibt den Hauptteil der Anwendung. Dort wird die FastAPI-Instanz definiert, welche für die REST-Funktionalität zuständig ist und die Endpunkte bereitstellt. Zudem werden die Docker und libvirt-Klasse dort importiert.

Es wurde eine CORS-Handhabung implementiert. Dazu wurde eine Middleware zu der API hinzugefügt, welche bei Anfragen die Adresse des Clients mit einer Liste von Adressen erlaubten Ursprungs abgleicht und die Kommunikation in der Antwort an den Browser autorisiert, sollte dieser in der Liste stehen.
Diese Liste muss in Zukunft mit entsprechenden Adressen ergänzt werden. Zudem sind nach jetziger Konfiguration die Übermittlung von Anmeldeinformationen zugelassen und es wurden keine Einschränkungen bei erlaubten HTTP-Methoden oder dem Header vorgenommen, was durch das Wildcard-Symbol „*“ zu erkennen ist (siehe Abbildung 4).

```python
origins = [
    "http://localhost.tiangolo.com",
    "https://localhost.tiangolo.com",
    "http://localhost",
    "http://localhost:8080",
]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Abbildung 4: CORS-Implementierung, Quelle: eigenes Bild

Nachfolgend wurden in der main.py-Datei die API-Endpunkte definiert. Dies ist über den *„@app.get/put/post/delete“*- Befehl umgesetzt. Der angehängte Pfad gibt der FastAPI-App zu verstehen, unter welcher URL der Endpunkt aufzurufen ist. Die ersten beiden Endpunkte *„/token“* und *„/users/me“* dienen der Authentifizierung bzw. Autorisierung und dem Nutzermanagement. Danach wurden die Endpunkte für allgemeine Operationen auf Ressourcen realisiert, wie das Starten und Stoppen von Containern und VMs, und im Anschluss daran spezifische Operationen, wie das Erstellen von Container und VMs. Abbildung 5 zeigt beispielhaft die Implementierung eines solchen Endpunktes an dem Befehl zum Starten einer Ressource.

```python
@app.put("/resources/{id}/start")
async def start_resource(
    current_user: Annotated[User, Security(get_current_active_user, scopes=["advanced"])],
    id: str,
    revertSnapshot = None
):
    try:
        if get_resource(id) == "docker":
            return docker.start_container(id)
        elif get_resource(id) == "kvm-qemu":
            return vm.start_vm(id, revertSnapshot)
    except APIError as e1:
        raise HTTPException(status_code=status.HTTP_409_CONFLICT, detail=e1.message)
    except ResourceAlreadyRunning as e2:
        raise HTTPException(status_code=status.HTTP_409_CONFLICT, detail=e2.message)
```

Abbildung 5: Beispiel-Endpunkt, Quelle: eigenes Bild

Es werden Fehler aus den Funktionen entgegengenommen und eine HTTP-Ausnahme ausgelöst. Je nachdem um welchen Fehler es sich handelt, weicht der HTTP-Statuscode ab. Es handelt sich vorwiegend um 400er Fehler, sprich um Fehler des Benutzers bzw. der Clientseite, wie z. B. die Übermittlung eines falschen Images oder eines fehlerhaften Arguments.

Die Funktion *„get_resource“* sucht nach der übergebenen ID und gibt zurück, ob es sich um eine VM oder einen Container handelt, um nachfolgend zu wissen, welche weiteren Funktionen aufzurufen sind.

4.3. Docker

Um auf die Docker-Dienste im Hintergrund zuzugreifen, bedarf es des Objekts *„client“* vom Typ *„DockerClient“*. Diese Klasse wird durch den Befehl *„docker.from_env“* instanziiert, welcher einen aus Umgebungsvariablen konfigurierten Client zurückgibt. Mithilfe dieses Clients kann auf Docker-Ressourcen wie Container und Images des Hostsystems zugegriffen werden.

Die erste Funktion *„list_containers“* gibt eine Liste aus JSON-Objekten zurück, die alle Container-Instanzen enthält. Der Parameter *„all“* gibt an, dass sowohl laufende als auch inaktive Container zurückgegeben werden sollen. Um einzelne Container anzusprechen, wird das Objekt *„containers“* der vorher definierten client-Variable angesprochen. Die list-Methode gibt lediglich eine Liste von Container-Objekten zurück. Die Objekte werden nun durchlaufen und es werden JSON-Objekte für entsprechende Container gebaut, welche die ID, den Namen, die Bezeichnung des Images und einen Status enthalten. Bei der ID handelt es sich um eine verkürzte ID von Docker, welche dem Client der API als eindeutige Identifizierung eines Containers zur Verfügung steht. Diese Objekte werden einer Liste hinzugefügt, welche nach Durchlaufen der Funktion zurückgegeben wird.

Die Funktion *„get_container_info“* gibt Information über einen Container zurück, welcher durch die ID angesprochen wird. Das spezifische Container-Objekt wird durch den Aufruf der Funktion *„get_container“* mithilfe der ID zurückgegeben. Die aufgerufene Funktion ruft dabei wiederrum die Docker-Methode *„containers.get“* auf, um auf das Docker-Objekt zuzugreifen. Sollte kein Container gefunden werden, fängt das *„try“*-Statement den Fehler des Typs „docker.errors.NotFound“ ab. Dieser Aufruf und seine Fehlerbehandlung finden sich in den folgenden Funktionen immer dann wieder, wenn eine spezifische Container-Instanz ermittelt werden muss.

Wenn ein Container gestartet werden soll, wird die Funktion *„start_container“* aufgerufen. Durch die Docker-Methode *„container.start“* wird dann ein Container gestartet.

Analog dazu wird zum Stoppen eines Containers die Funktion „stop_container“ aufgerufen. Durch die Docker-Methode *„container.stop“* wird der Container gestoppt.

Die Funktion *„restart_container“* ist dafür zuständig, den Container neuzustarten.

Soll ein Container entfernt werden, wird die Funktion *„remove_container“* aufgerufen. Es wird die Docker-Methode *„container.remove“* dafür verwendet. Sollte der Container noch laufen, wird eine Fehlermeldung zurückgegeben, da nur ein Container gelöscht werden kann, wenn er inaktiv ist. Bei allen drei Funktionen wird nach erfolgreicher Operation ein String mit einer Erfolgsmeldung zurückgegeben.

Die nächste Funktion *„prune_containers“* wird aufgerufen, wenn alle inaktiven Container entfernt werden sollen (docker prune [10]). Sie gibt bei erfolgreicher Entfernung eine Erfolgsmeldung und eine Liste aller entfernten Container zurück.

Als letzte Funktion ist das Erstellen und Starten eines Containers implementiert (docker run [10]). Das geschieht durch die Docker-Methode *„containers.run“,* welcher zwei Parameter übergeben werden. Zum einen wird ein String namens *„image“* übergeben, welches die Bezeichnung des Docker Images ist, zum anderen Parameter wie Ports oder Namen des Containers. Letzteres geschieht in Form eines JSON-Objektes, welches an die Funktion übergeben wird. Dieses Objekt ist von der Klasse *„ContainerObj“* und erbt von der BaseModel-Klasse. Dort sind einige Felder mit Typendeklaration und Standardwerten definiert. Mit der Klasse *„Config“,* innerhalb des *„ContainerObj“,* ist es möglich bestimmte Einstellungen vorzunehmen, wie die Erstellung eines Beispiel-Objektes *„schema_extra“.* Der Ausdruck *„extra = Extra.allow“* erlaubt das Hinzufügen zusätzlicher Felder. Damit die Docker-Methode die Parameter verwenden kann, muss das *„ContainerObj“* zuerst in ein Python-Dictionary konvertiert werden. Es können mehrere Fehler auftreten. Das Image kann durch Schreibfehler nicht gefunden werden. Es kann ein Fehler in den Parametern vorliegen, dass bspw. ein Port bereits von einer anderen Anwendung verwendet wird. Solche Fehler werfen eine Docker *„APIError“*-Ausnahme. Des Weiteren kann es passieren, dass ein Parameter falsch bezeichnet wurde, was als *„TypeError“* auftaucht und ebenfalls behandelt wird (siehe Abbildung 6).

```
95      def runContainer(self, image, attr):
96          try:
97              obj = self.client.containers.run(image, **attr.dict())
98              return {"Following container sucessfully created": {"Id" : obj.attrs.get('Id'), "Name": obj.attrs.get('Name')},
99                      "info": "For further parameters visit: https://docker-py.readthedocs.io/en/stable/containers.html"}
100         except errors.APIError as e1:
101             if e1.status_code == 404:
102                 raise ImageNotFound()
103             elif e1.status_code >= 409:
104                 raise APIError(str(e1.explanation))
105         except TypeError as e2:
106             raise ArgumentNotFound(e2.args[0])
```

Abbildung 6: Docker Run, Quelle: eigenes Bild

4.4. VM-Klasse

In diesem Abschnitt geht es um die Implementierung der Funktionalitäten zur Steuerung von Virtuellen Maschinen des Typs kvm/qemu. Bei dem ersten Use Case handelt es sich wie bei Docker auch um die Auflistung aller VM-Instanzen. Dies wird durch die Funktion *„list_vms“* umgesetzt. In der Funktion wird, wie in allen anderen Funktionen der Klasse VM, zunächst die Funktion *„libvirt_connect“* aufgerufen. Es wird mit dem Befehl *„libvirt.open(‘qemu:///system‘)“* eine Verbindung zu den Diensten von kvm/qemu geöffnet, ist dies nicht möglich, weil die libvirt-Dienste nicht erreichbar sind, wird ein Ausnahme des Typs *„ConnectionFailed“* abgefangen. Bei erfolgreicher Ausführung wird ein Objekt zurückgegeben, über welche libvirt-Ressourcen angesprochen werden können. Mithilfe der libvirt-Methode *„listAllDomains“* wird eine Liste aller Domains (VMs) zurückgegeben, welche analog zur Auflistung von Containern durchlaufen wird und als JSON-Objekt einer Python-List hinzugefügt wird (siehe Abbildung 7).

```
26  class VM():
27      def list_vms(self):
28          conn = self.libvirt_connect()
29          domains = {}
30          domains = conn.listAllDomains()
31          jsonList = []
32          for dom in domains:
33              jsonList.append({"uuid": dom.UUIDString(),
34                               "name": dom.name(),
35                               "isActive": dom.isActive(),
36                               "status": self.get_vm_status(dom).get("desc"),
37                               "isPersistent": dom.isPersistent()})
38          return jsonList
```

Abbildung 7: Auflisten der VMs, Quelle: eigenes Bild

Die Funktion *„list_snapshots“* gibt auf gleiche Weise alle Snapshots einer virtuellen Maschine zurück. Des Weiteren werden durch die Funktion *„list_storage_vol“* alle Storage Volumen als Liste ausgegeben. Dafür muss zunächst der Storage Pool namens *„default“* abgefragt werden. Das Pool-Objekt enthält eine Methode „listVolumes“, über welche alle Volumes dieses Pools zurückgegeben werden.

Die folgende Funktion *„get_vm_info“* gibt Informationen über eine VM-Instanz zurück. Die Funktion „get_vm“ gibt das der ID entsprechende VM-Objekt zurück. Dazu wird libvirt-Methode *„lookupByUUIDString“* aufgerufen. Sollte keine VM gefunden werden, wird der Fehler durch die Ausnahme „ResourceNotFound“ abgefangen. Über das ermittelte VM-Objekt werden Attribute wie Status und Hardware-Informationen ermittelt.

Um eine VM-Instanz zu starten, muss die Funktion *„start_vm“* aufgerufen werden. Zunächst wird geprüft, ob eine VM über ein Snapshot wiederhergestellt werden soll. Ist dies der Fall, wird der Snapshot unter der übergebenen Bezeichnung gesucht. Dann wird über *„revertToSnapshot“* der Snapshot gestartet. Sollte der Snapshot nicht vorhanden sein, wird dieser Fehler durch die generische „APIError“-Ausnahme abgefangen. Es wird der Status der VM abgefragt. Wenn sich die VM im Status 3 befindet, wurde sie pausiert, dann soll der Befehl *„dom.resume“* ausgeführt werden, um die VM zu starten. Sollte die VM allerdings heruntergefahren worden sein, wird sie mit dem Befehl *„dom.create“* wieder hochgefahren. Wenn sie beim Herunterfahren gespeichert wurde (->managedSave [18]), dann wird sie durch letzteren Befehl aus dem gespeicherten Zustand wieder hochgefahren. Sollte die VM bereits laufen, wird eine Ausnahme der Klasse *„DomainAlreadyRunning“* zurückgegeben.

Soll eine laufende VM-Instanz gestoppt bzw. pausiert werden, wird die Funktion *„stop_vm“* aufgerufen. Es wird analog zum Starten der VM der Status überprüft. Sollte die VM im Status 1 (aktiv) sein, wird der Befehl *„dom.suspend“* ausgeführt, andernfalls die Ausnahme *„DomainNotRunning“* zurückgemeldet. Hier ist zu erwähnen, dass *„suspend“* eine VM pausiert und damit den Speicherzustand des Gastes vorübergehend speichert. Zu einem späteren Zeitpunkt ist es möglich, den Gast wieder in seinen ursprünglichen Zustand zu versetzten und die Ausführung dort fortzusetzen, wo sie aufgehört hat. Der Befehl *„suspend“* speichert kein dauerhaftes Abbild der VM, hierfür muss die VM explizit gespeichert werden (->managedSave).

Die Funktion *„reboot_vm“* führt einen Neustart der VM durch. Wenn sich die VM im laufenden Zustand befindet, wird die libvirt-Methode *„dom.reboot“* ausgeführt und ein Kommando zum Neustart an die VM gesendet. Es ist nun von der VM und der installierten Software abhängig, ob sie auf das Kommando reagiert [19].

Zum Herunterfahren einer VM muss die Funktion *„shutdown_vm“* ausgeführt werden. Sie übergibt zum einen die ID der herunterzufahrenden VM und zum anderen eine boolesche

Variable, die besagt, ob die VM gespeichert werden soll. Wenn ja, wird die libvirt-Methode *„managedSave“* ausgeführt. Dadurch wird im Gegensatz zu save das Speichern der VM automatisiert, beim nächsten Start der VM verwendet libvirt den letzten Speicherstand der VM. Soll die VM nicht gespeichert werden, wird das Herunterfahren durch *„destroy“* erzwungen. Ansonsten wird die VM standardmäßig heruntergefahren, das heißt analog zum Neustart wird ein Kommando an die VM gesendet (siehe Abbildung 8).

```
135     def shutdown_vm(self, id, save, force):
136         conn = self.libvirt_connect()
137         dom = self.get_vm(id)
138         try:
139             state = self.get_vm_status(dom).get("state")
140             if state == 1:
141                 if save:
142                     dom.managedSave()
143                     return "VM sucessfully saved"
144                 elif force:
145                     dom.destroy()
146                     return "VM was forced to shutdown"
147                 else:
148                     dom.shutdown()
149                     return "VM sucessfully shutdown"
150             else:
151                 raise ResourceNotRunning()
152         except libvirtError as e:
153             raise APIError(str(e))
```

Abbildung 8: Herunterfahren einer VM, Quelle: eigenes Bild

Eine VM wird gelöscht, indem die Funktion *„delete_VM“* aufgerufen wird. Wenn es sich um eine laufende VM-Instanz handelt, wird eine Fehlermeldung zurückgegeben, man solle die laufende Instanz vorher herunterfahren. Sollte dies nicht der Fall sein, wird die VM über den Befehl *„dom.undefine“* gelöscht. Eine Ausnahme besteht darin, dass das Löschen einer VM nicht zugelassen wird, wenn diese vorher gespeichert wurde.

Die Funktion *„delete_snapshot“* löscht einen Snapshot einer VM. Die Funktion *„delete_storage_vol“* löscht das Storage Volume einer VM. Die Funktion wird aufgerufen, wenn eine VM gelöscht werden soll. Die VM muss dafür allerdings heruntergefahren worden sein, sonst wird ein Fehler zurückgegeben.

Zum Starten einer VM werden zwei Ansätze implementiert. Beim ersten Ansatz werden die Konfigurationen durch den Client direkt im XML-Format übermittelt, es wird die Funktion *„run_vm_xml“* ausgeführt. Diese nimmt die XML-Anfrage und formatiert sie. Im Anschluss wird das XML mithilfe der Funktion *„toprettyxml“* dargestellt, sodass es durch darauffolgenden Befehl *„defineXMLFlags“* verstanden werden kann. Letzterer Aufruf sorgt dafür, dass ein VM-Objekt mit der definierten Konfiguration instanziiert wird. Dieses VM-

Objekt wird mit *„dom.create“* gestartet. Bei erfolgreichem Durchlauf gibt die Funktion eine Erfolgsmeldung zurück, wenn nicht, wird ein APIError geworfen. Der zweite Ansatz übermittelt die Konfiguration über JSON als Schlüssel-Wertepaare. Dies ist für Anfragen gedacht, welche keine zusätzlichen Konfigurationen für die VM benötigen. Die Funktion *„run_vm_json“* wird durch den API-Endpunkt aufgerufen, welche im Gegensatz zum XML-Ansatz ein JSON-Objekt übergeben bekommt. Das JSON-Objekt wird in ein Python-Dictionary konvertiert und die Werte in eine XML-Vorlage eingefügt. Bevor die VM erstellt wird, wird zunächst ein entsprechendes Storage Volume mit dem Namen der VM erstellt. Diese Volume wird in der XML-Konfiguration als *„<disk>“*-Element spezifiziert. Es wird zudem eine standardmäßige Netzwerkschnittstelle definiert. Die XML-Vorlage wird an die libvirt-Methode *„defineXMLFlags“* übergeben.

Die Funktion *„create_snapshot“* erstellt einen Snapshot einer bestimmten VM. Dazu wird eine Konfiguration spezifiziert, welche lediglich den Namen des zu erstellenden Snapshots enthält. Die Funktion „create_storage_vol“ erstellt ein Storage Volume und wird bei der Erstellung einer VM aufgerufen. Die übergebene „Flag“ garantiert, dass der Befehl keine Festplatten ändert, es sei denn der gesamte Satz von Änderungen kann atomar durchgeführt werden, was die Wiederherstellung nach einem Ausfall vereinfacht. Storage Volumes sind Abstraktionen von physischen Partitionen, logischen LVM-Volumes, dateibasierten Disk-Images und anderen von libvirt verwalteten Speichertypen. Speicher-Volumes werden virtuellen Gastmaschinen als lokale Speichergeräte präsentiert, unabhängig von der zugrundeliegenden Hardware. Das Volume wird mit Standardeinstellung erstellt, d.h. als ein Plattenspeicher in qcow2-Format. Übergeben wird lediglich der Name der VM.

4.5. Security Klasse (Authentifizierung und Autorisierung)

Im folgenden Abschnitt ist die Implementierung der Authentifizierungs- und Autorisierungsfunktionen der API beschrieben. Grundlegend sind zwei Geltungsbereiche (nachfolgend Scopes) zur Autorisierung implementiert. Zum einen *„basic“,* mit der Berechtigung zum Lesen von Ressourcen, sprich für get-Abfragen. Zum anderen findet sich die Berechtigung *„advanced“,* welche auf die restlichen Abfragen zum Manipulieren von Containern und VMs zugreifen darf. Es reicht nicht nur *„advanced“* zu besitzen, um auf alles zugreifen zu können, man braucht gleichzeitig auch die *„basic“* – Berechtigung um Lesen zu können. Dazu wird ein OAuth2PasswordBearer – Objekt definiert, welches die URL des *„/token“*-Endpunkts übergeben bekommt. Dadurch weiß das FastAPI-Frontend, über welchen

Endpunkt Benutzername und Passwort übermittelt werden sollen. Die eben beschriebenen Scopes werden dort ebenfalls definiert, wie in Abbildung 9 zu sehen ist.

```
53    oauth2_scheme = OAuth2PasswordBearer(
54        tokenUrl="token",
55        scopes={"basic": "Basic read permission on ressources",
56                "advanced": "Write and run permission on ressources"
57        }
58    )
```

Abbildung 9: OAuth-Implementierung, Quelle: eigenes Bild

Die Speicherung von Nutzerdaten geschieht über ein Python-Dictionary, das stellvertretend für eine Datenbankanbindung implementiert wurde. Die Passwörter der Benutzer liegen als Hash-Werte vor, um Vertraulichkeit und Integrität zu gewährleisten.

Um sich als Benutzer zu authentifizieren, muss der *„/token“*-Endpunkt aufgerufen werden. Dieser übermittelt einen generierten JWT-Token an den Benutzer, welcher zur Authentifizierung für weitere Anfragen dient. Der *„/token“*-Endpunkt ruft dazu zunächst die *„authenicate_user“*-Funktion auf, welche sich der Funktion *„get_user“* bedient, um die provisorische Datenbank nach dem Benutzer zu durchsuchen und diesen zurückzugeben (siehe Abbildung 10).

```
73    def authenticate_user(fake_db, username: str, password: str):
74        user = get_user(fake_db, username)
75        if not user:
76            return False
77        if not verify_password(password, user.hashed_password):
78            return False
79        return user
```

Abbildung 10: Benutzer-Auth., Quelle: eigenes Bild

In einem zweiten Schritt wird das übergebene Passwort verifiziert, indem es mit dem gehashten Passwort aus der „Datenbank“ abgeglichen wird. Der Abgleich verwendet dabei eine Variable des Typen *„CryptContext“*. Dies ist ein Objekt des passlib-Moduls, das für das Hashen von Passwörtern zuständig ist. Mit *„schemes“* spezifiziert man einen Algorithmus, welcher für die Berechnung verwendet werden soll, in diesem Fall *„bcrypt“*. Wenn das Passwort korrekt ist, gibt die Funktion den gefundenen Benutzer zurück (siehe Abbildung 11).

```python
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def verify_password(plain_password, hashed_password):
    return pwd_context.verify(plain_password, hashed_password)

def get_password_hash(password):
    return pwd_context.hash(password)

def get_user(db, username: str):
    if username in db:
        user_dict = db[username]
        return UserInDB(**user_dict)
```

Abbildung 11: Passwort-Verifizierung, Quelle: eigenes Bild

Nachfolgend wird ein Token generiert. Das geschieht durch den Aufruf der Funktion *„create_access_token“*. Es werden der Benutzername, die Scopes und die Wirkungsdauer des Tokens übergeben. Anschließend werden die Benutzerdaten durch einen konstanten Schlüssel als JWT-Token kodiert. Dieser JWT-Token wird zurückgegeben (siehe Abbildung 12).

```python
def create_access_token(data: dict, expires_delta: timedelta | None = None):
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() + timedelta(minutes=15)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt
```

Abbildung 12: Erstellung des Token, Quelle: eigenes Bild

Der Benutzer der API besitzt nun einen API-Token mit einer Wirkungsdauer von 30 Minuten. Dieses Token muss bei der nächsten Anfrage an die API in dem Header-Feld *„Authorization“* hinterlegt sein, um auf Ressourcen der API zugreifen zu dürfen. Nach Ablauf der Wirkungsdauer muss sich der Benutzer erneut über den *„/token“*-Endpunkt authentifizieren.

Greift der Benutzer auf einen API-Endpunkt zu, wird sein Token jedes Mal verifiziert. Dies geschieht durch die Abhängigkeit zur *„get_current_active_user“*-Funktion der Security-Klasse. Um Abhängigkeiten zu injizieren, wird der Befehl *„Depends“* verwendet. Wenn zusätzlich ein Scope übergeben werden soll, muss auf *„Security“* zurückgegriffen werden, wie man in folgendem Beispiel aus main.py erkennt. Folgenden Endpunkt der Abbildung 13 kann nur ein authentifizierter Client aufrufen, welcher den Scope *„basic“* besitzt:

```python
@app.get("/users/me/", response_model=User)
async def read_users_me(
    current_user: Annotated[User, Security(get_current_active_user, scopes=["basic"])],
):
    return current_user
```

Abbildung 13: Benutzer-Endpunkt, Quelle: eigenes Bild

Die Funktion *„get_current_active_user“* greift das übermitteltete JWT-Token ab und besitzt die Aufgabe, den passenden Benutzer zurückzugeben. Analog zum letzteren Beispiel besitzt diese Funktion ebenfalls eine Abhängigkeit, und zwar zum Objekt *„oauth2_scheme“* vom Typ *„OAuth2PasswordBearer“*, welcher das Token an die Funktion übergibt. Zunächst dekodiert die Funktion das erhaltene Token. Daraus kann es zum einen den Benutzernamen lesen. Wenn dieser leer ist, wird eine HTTP-Antwort mit Fehlercode 401 zurückgegeben. Wenn ein Fehler beim Lesen des JWT-Tokens oder bei der formalen Validierung des „TokenModel“ mit Pydantic aufkommt, löst die letztere HTTP-Ausnahme ebenfalls aus. Zudem wird überprüft, ob ein Benutzer mit diesem Benutzernamen existiert, wenn nicht, wird die HTTP-Ausnahme ausgelöst. Als letztes wird überprüft, ob alle abhängigen Scopes in dem empfangenen Token enthalten sind, das heißt, ob der Benutzer autorisiert ist. Besitzt er nicht die notwendigen Scopes, wird eine HTTP-Ausnahme ausgelöst (siehe Abbildung 14).

```python
async def get_current_active_user(
    security_scopes: SecurityScopes, token: Annotated[str, Depends(oauth2_scheme)]
):
    if security_scopes.scopes:
        authenticate_value = f'Bearer scope="{security_scopes.scope_str}"'
    else:
        authenticate_value = "Bearer"
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": authenticate_value},
    )
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        if username is None:
            raise credentials_exception
        token_scopes = payload.get("scopes", [])
        token_data = TokenData(scopes=token_scopes, username=username)
    except (JWTError, ValidationError):
        raise credentials_exception
    user = get_user(fake_users_db, username=token_data.username)
    if user is None:
        raise credentials_exception

    for scope in security_scopes.scopes:
        if scope not in token_data.scopes:
            raise HTTPException(
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail="Not enough permissions",
                headers={"WWW-Authenticate": authenticate_value},
            )
    if user.disabled:
        raise HTTPException(status_code=400, detail="Inactive user")
    return user
```

Abbildung 14: Benutzer verifizieren, Quelle: eigenes Bild

4.6. Fehlerbehandlung

Im Sinne der REST-Architekturprinzipien wird eine Fehlerbehandlung implementiert, welche dem Client eine unmissverständliche Fehlermeldung zurückgibt. Zudem werden selbstschreibende Nachrichten umgesetzt, welche dem Client eine mögliche Lösung des Fehlers übermitteln. Dazu wurden Klassen definiert, welche von der „Exception"-Klasse erben. Zudem macht es den Code übersichtlicher und effizienter, wenn die Fehlerbehandlung redundant vorkommender Fehler nur einmal im Code definiert werden muss. In einigen Ausnahmen sind Fehlermeldungen festgeschrieben, in anderen werden sie beim Aufrufen übergeben.

Folgende Klassen zur Fehlerbehandlung wurden implementiert:

- RessourceNotFound: Angefragter Container oder VM wurde unter der übermittelten ID nicht gefunden.
    - Übermittelter Identifier fehlerhaft
    - Ressource nicht mehr vorhanden
- ArgumentNotFound: Argument oder Parameter ist nicht korrekt, z. B. bei der Erstellung von Containern
    - Übermitteltes Argument fehlerhaft
- ImageNotFound: Bezeichnung eines Docker Image ist nicht korrekt.
    - Übermittelte Bezeichnung fehlerhaft
- APIError: Allgemeine Fehler aus Konflikten innerhalb von Docker oder KVM, z. B. wenn ein Container gestoppt wird, obwohl er schon inaktiv war.
- ConnectionFailed: Fehler in der Verbindung zum Daemon.
- ResourceNotRunning: Angefragte VM oder angefragter Container läuft nicht.
    - VM oder Container soll neugestartet, gestoppt oder heruntergefahren werden, obwohl diese nicht mehr aktiv sind.
- ResourceAlreadyRunning: Angefragte VM oder angefragter Container läuft bereits.
    - VM oder Container soll gestartet werden, aber befindet sich bereits im Status laufend.
- RessourceRunning: Angefragte Ressource läuft.
    - VM oder Container soll gelöscht werden, läuft aber noch. Der Nutzer muss die Ressource vorher stoppen bzw. herunterfahren.

4.7. Testen der API

Die API wurde auf Funktionalität getestet. Dazu wurde die Software Postman verwendet. Postman macht es möglich, REST-APIs zu testen. Über selbst erstellte Anfragen kann man die Endpunkte der API ansprechen und diese auf Fehler testen. Die Testabfragen finden sich unter folgendem Link: https://github.com/DanB1999/Virtualization_API/tree/main/tests. Abbildung 15 zeigt das Testen der API in Postman anhand des Startens einer VM.

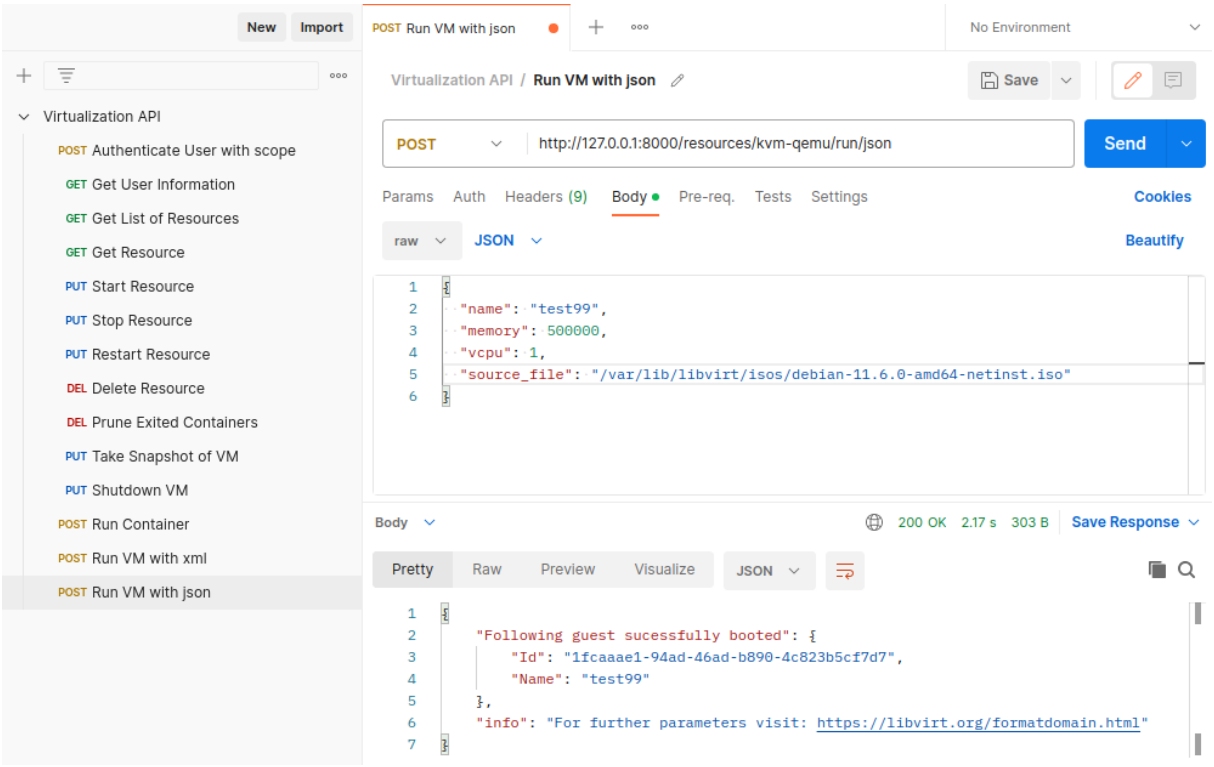


Abbildung 15: API-Testcase, Quelle: eigenes Bild

5. Fazit

Die Entwicklung und Implementierung einer API zur Steuerung von Containern und VMs konnte erfolgreich umgesetzt werden. Die definierten, funktionalen Zielsetzungen konnten nahezu alle implementiert und getestet werden. Das Herunterfahren und Neustarten einer VM ist beim Testen fehlgeschlagen. Diese Operationen erfordern ein bestimmtes Kommando, welches an den Gast gesendet wird. Dazu muss auf dem Gast Software installiert werden, welche dieses Kommando entgegennimmt und ausführt. Zusätzlich zu den in Kapitel 1 definierten Zielsetzungen konnten einige Funktionen wie das Erstellen von Snapshots von VMs realisiert werden. Die Anforderung dazu ergab sich im Laufe der Implementierung und Auseinandersetzung mit der libvirt-API, da eine Möglichkeit benötigt wird, Festplattenspeicher persistent zu sichern. Die nicht-funktionalen Anforderungen der Usability konnte anhand des REST-Prinzips und dessen Best-Practices in die Tat umgesetzt werden. Ebenso wurde eine rudimentäre Sicherheitsfunktion implementiert. Die Performanz der Anwendung wurde im Zuge dieser Arbeit nicht berücksichtigt, lässt sich jedoch rückblickend als irrelevant für diese Art der implementierten Software bewerten. Herausforderungen traten bei der Einarbeitung zur Konfiguration von VMs auf, da diese viele unterschiedliche Einstellungen bieten. Des Weiteren gestaltete sich die Implementierung der libvirt-Funktionalitäten teilweise als umständlich, da öfters auf das XML-Format zurückgegriffen werden musste.

5.1.5.1 Ausblick

In der vorgeschriebenen Zeit konnten alle Anforderungen realisiert werden, doch es besteht noch Weiterentwicklungspotenzial. Zum Beispiel ließe sich das Konfigurieren einer VM weiter ausgestalten, indem verschiedene Netzwerk- und Speicherkomponenten hinzugefügt werden. Zudem bestehe die Möglichkeit, eine VM nach Erstellung zu bearbeiten und beispielsweise die CPU- und RAM-Zuweisung zu erhöhen. Die Docker-Komponente könnte man insoweit verbessern, dass man Images eigenhändig erstellen kann. Die Docker Engine API lässt Spielraum für weitere Funktionen wie Netzwerke und Orchestrierung über Swarms, sodass sich die hier erarbeitete Lösung in Zukunft durch entsprechende Funktionalitäten erweitern lässt. Die Ergebnisse dieser Arbeit eröffnen die Möglichkeit der weiteren Integration in andere Python-Projekte oder in die Forschungsarbeit. So wäre es denkbar, die API als Command Line Interface (CLI) zu implementieren. Das FastAPI-Framework würde eine Bereitstellung (Deployment) als Webservice unterstützen. So ließe sich eine API entwickeln, die unabhängig von einem Host agiert und Container und VMs über ein Netzwerk ansprechen kann.

6. Anhang

6.1. main.py

```python
from datetime import  timedelta
from typing import List, Annotated
from fastapi import Depends, FastAPI, HTTPException, Query, Request, Security, status
from fastapi.security import OAuth2PasswordRequestForm
from pydantic import BaseModel
from Docker import Docker, ContainerObj
from VM import VM, DomainObj
from Security import (
    User, Token, fake_users_db,
    authenticate_user, create_access_token,
    get_current_active_user, ACCESS_TOKEN_EXPIRE_MINUTES
)
from fastapi.middleware.cors import CORSMiddleware
from exceptions import (
    APIError, ArgumentNotFound, ResourceAlreadyRunning,
    ResourceNotFound, ResourceNotRunning, ImageNotFound, ResourceRunning
)

docker = Docker()
vm = VM()

app = FastAPI()

origins = [
    "http://localhost.tiangolo.com",
    "https://localhost.tiangolo.com",
    "http://localhost",
    "http://localhost:8080",
]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

@app.post("/token", response_model=Token)
async def login_for_access_token(
    form_data: Annotated[OAuth2PasswordRequestForm, Depends()]
):
    if len(form_data.scopes) == 1:
        scopes = str(form_data.scopes[0]).split('+')
    else:
        scopes = form_data.scopes
    user = authenticate_user(fake_users_db, form_data.username, form_data.password)
    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )
    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={"sub": user.username, "scopes": scopes},
        expires_delta=access_token_expires
    )
    return {"access_token": access_token, "token_type": "bearer"}
```

```python
@app.get("/users/me/", response_model=User)
async def read_users_me(
    current_user: Annotated[User, Security(get_current_active_user, scopes=["basic"])],
):
    return current_user

@app.get("/resources")
async def get_list(
    current_user: Annotated[User, Security(get_current_active_user, scopes=["basic"])],
    type: Annotated[str, Query(description="docker container, docker images, kvm-qemu vms, kvm-qemu volumes")]
):
    if type in "docker container":
        return docker.list_containers()
    elif type in "docker images":
        return docker.list_images()
    elif type in "kvm-qemu vms":
        return vm.list_vms()
    elif type in "kvm-qemu volumes":
        return vm.list_storage_vol()

@app.get("/resources/{id}")
async def get_info(
    current_user: Annotated[User, Security(get_current_active_user, scopes=["basic"])],
    id: str,
    filter: Annotated[bool, Query(description="List Snapshots of VM")] = False
):
    if get_resource(id) == "docker":
        return docker.get_container_info(id)
    elif get_resource(id) == "kvm-qemu":
        if filter:
            return vm.list_snapshots(id)
        else:
            return vm.get_vm_info(id)

@app.put("/resources/{id}/start")
async def start_resource(
    current_user: Annotated[User, Security(get_current_active_user, scopes=["advanced"])],
    id: str,
    revertSnapshot = None
):
    try:
        if get_resource(id) == "docker":
            return docker.start_container(id)
        elif get_resource(id) == "kvm-qemu":
            return vm.start_vm(id, revertSnapshot)
    except APIError as e1:
        raise HTTPException(status_code=status.HTTP_409_CONFLICT, detail=e1.message)
    except ResourceAlreadyRunning as e2:
        raise HTTPException(status_code=status.HTTP_409_CONFLICT, detail=e2.message)

@app.put("/resources/{id}/stop")
async def stop_resource(
    current_user: Annotated[User, Security(get_current_active_user, scopes=["advanced"])],
    id: str
):
```

```python
    try:
        if get_resource(id) == "docker":
            return docker.stop_container(id)
        elif get_resource(id) == "kvm-qemu":
            return vm.stop_vm(id)
    except APIError as e1:
        raise HTTPException(status_code=status.HTTP_409_CONFLICT, detail=e1.message)
    except ResourceNotRunning as e2:
        raise HTTPException(status_code=status.HTTP_409_CONFLICT, detail=e2.message)

@app.put("/resources/{id}/restart")
async def restart_resource(
    current_user: Annotated[User, Security(get_current_active_user, scopes=["advanced"])],
    id: str
):
    try:
        if get_resource(id) == "docker":
            return docker.restart_container(id)
        elif get_resource(id) == "kvm-qemu":
            return vm.reboot_vm(id)
    except APIError as e1:
        raise HTTPException(status_code=status.HTTP_409_CONFLICT, detail=e1.message)
    except ResourceNotRunning as e2:
        raise HTTPException(status_code=status.HTTP_409_CONFLICT, detail=e2.message)

@app.delete("/resources/{id}/delete")
async def remove_resource(
    current_user: Annotated[User, Security(get_current_active_user, scopes=["advanced"])],
    id: str,
    deleteStorageVol: Annotated[bool, Query(description="Delete asociated storage volume")] = True,
    deleteSnapshot: Annotated[str, Query(description="Delete snapshot instead of vm")] = None,
):
    try:
        if get_resource(id) == "docker":
            return docker.remove_container(id)
        elif get_resource(id) == "kvm-qemu":
            if deleteSnapshot:
                return vm.deleteSnapshot(id, deleteSnapshot)
            if len(vm.get_vm_snapshots(id)) == 0:
                if deleteStorageVol:
                    vm.delete_storage_vol(id)
                return vm.delete_vm(id)
            else:
                raise HTTPException(status_code=status.HTTP_409_CONFLICT,
                                    detail="Cannot delete inactive domain with "
                                        + str(len(vm.getDomainSnapshots(id)))+ " snapshots")
    except APIError as e1:
        raise HTTPException(status_code=status.HTTP_409_CONFLICT, detail=e1.message)
    except ResourceRunning as e2:
        raise HTTPException(status_code=status.HTTP_409_CONFLICT, detail=e2.message)

@app.delete("/resources/docker/prune")
async def prune_containers(
    current_user: Annotated[User, Security(get_current_active_user, scopes=["advanced"])]
):
    try:
        return docker.prune_containers()
    except APIError as e1:
        raise HTTPException(status_code=status.HTTP_409_CONFLICT, detail=e1.message)
```

```python
@app.put("/resources/{id}/snapshot")
async def take_snapshot(
    current_user: Annotated[User, Security(get_current_active_user, scopes=["advanced"])],
    id: str,
    snapshot_name: str
):
    get_resource()
    try:
        return vm.create_snapshot(id, snapshot_name)
    except APIError as e1:
        raise HTTPException(status_code=status.HTTP_409_CONFLICT, detail=e1.message)

@app.put("/resources/{id}/shutdown")
async def shutdown_vm(
    current_user: Annotated[User, Security(get_current_active_user, scopes=["advanced"])],
    id: str,
    save: Annotated[bool, Query(description="Save VM for later use, priority over force command")] = False,
    force: bool = False,
):
    get_resource()
    try:
        return vm.shutdown_vm(id, save, force)
    except APIError as e1:
        raise HTTPException(status_code=status.HTTP_409_CONFLICT, detail=e1.message)
    except ResourceNotRunning as e2:
        raise HTTPException(status_code=status.HTTP_409_CONFLICT, detail=e2.message)

@app.post("/resources/docker/run")
async def run_container(
    current_user: Annotated[User, Security(get_current_active_user, scopes=["advanced"])],
    obj: ContainerObj,
    image: str
):
    try:
        return docker.run_container(image, obj)
    except APIError as e1:
        raise HTTPException(status_code=status.HTTP_409_CONFLICT, detail=e1.message)
    except ImageNotFound as e2:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail=e2.message)
    except ArgumentNotFound as e3:
        raise HTTPException(status_code=status.HTTP_406_NOT_ACCEPTABLE, detail=e3.message)

class Item(BaseModel):
    name: str
    tags: List[str]

@app.post(
    "/resources/kvm-qemu/run/xml",
    openapi_extra={
        "requestBody": {
            "content": {"application/xml": {"schema": Item.schema()}},
            "required": True,
        }})
async def run_vm_xml(
    current_user: Annotated[User, Security(get_current_active_user, scopes=["advanced"])],
    request: Request
):
```

```
    content_type = request.headers['Content-Type']
    if content_type == "application/xml":
        body = await request.body()
        res = vm.run_vm_xml(body)
    else:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
                            detail=f'Content type {content_type} not supported')
    return res

@app.post("/resources/kvm-qemu/run/json")
async def run_vm_json(
    current_user: Annotated[User, Security(get_current_active_user, scopes=["advanced"])],
    obj: DomainObj
):
    try:
        vm.create_storage_vol(obj.dict().get("name"))
        return vm.run_vm_json(obj)
    except APIError as e:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail=e.message)

def get_resource(id):
    res = None
    try:
        if vm.get_vm(id):
            res = "kvm-qemu"
    except ResourceNotFound:
        try:
            if docker.get_container(id):
                res = "docker"
        except ResourceNotFound as e:
            raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail=e.message)
    return res

```

6.2. Docker.py

```python
import docker
from docker import errors
from pydantic import BaseModel, Extra
from exceptions import (
    APIError, ArgumentNotFound, ImageNotFound, ResourceNotFound,
    ResourceAlreadyRunning, ResourceNotRunning, ResourceRunning
)

class ContainerObj(BaseModel):
    name: str | None = None
    ports: object | None = None
    volumes: list[str | None ]= None
    detach: bool = True

    class Config:
        schema_extra = {
            "example": {
                "name": "Container1",
                "ports": {'5000/tcp':1000 },
                "volumes": ["/home/user1/:/mnt/vol2"],
                "detach": True
            }
        }
        extra = Extra.allow

class Docker():
    def __init__(self):
        self.client = docker.from_env()

    def list_containers(self):
        list = self.client.containers.list(all=True)
        jsonList = []
        for i in range(0 , list.__len__()):
            jsonList.append({"Id" : list[i].short_id,
                             "Name": list[i].attrs['Name'],
                             "Image": list[i].attrs['Config']['Image'],
                             "Status": list[i].attrs['State']['Status']})
        return jsonList

    def list_images(self):
        list = self.client.images.list(all=True)
        jsonList = []
        for i in range(0 , list.__len__()):
            jsonList.append({"Name": list[i].attrs["RepoTags"][0],
                             "Id" : list[i].attrs["Id"],
                             "Created": list[i].attrs["Created"],
                             "Container": list[i].attrs["ContainerConfig"]["Hostname"]
                             })
        return jsonList

    def get_container_info(self, id):
        container = self.get_container(id)
        return container.attrs

    def start_container(self, id):
        container = self.get_container(id)
        try:
            state = container.attrs["State"]["Status"]
```

```
            if state == "running":
                container.stop()
                return "Container sucessfully stopped"
            else:
                raise ResourceNotRunning()
        except errors.APIError as e:
            raise APIError(str(e.explanation))

    def restart_container(self, id):
        container = self.get_container(id)
        try:
            state = container.attrs["State"]["Status"]
            if state == "running":
                container.restart()
                return "Container sucessfully restarted"
            else:
                raise ResourceNotRunning()
        except errors.APIError as e:
            raise APIError(str(e.explanation))

    def remove_container(self, id):
        container = self.get_container(id)
        try:
            state = container.attrs["State"]["Status"]
            if state != "running":
                container.remove()
                return "Container sucessfully removed"
            else:
                raise ResourceRunning()
        except errors.APIError as e:
            raise APIError(str(e.explanation))

    def prune_containers(self):
        try:
            res = self.client.containers.prune()
            return {"Following containers sucessfully removed":res}
        except errors.APIError as e:
            raise APIError(str(e.explanation))

    def run_container(self, image, attr):
        try:
            obj = self.client.containers.run(image, **attr.dict())
            return {"Following container sucessfully created": {
                "Id" : obj.attrs.get('Id'), "Name": obj.attrs.get('Name')
                },
                "info": "For further parameters visit: https://docker-py.readthedocs.io/en/stable/
        except errors.APIError as e1:
            if e1.status_code == 404:
                raise ImageNotFound()
            elif e1.status_code >= 409:
                raise APIError(str(e1.explanation))
        except TypeError as e2:
            raise ArgumentNotFound(e2.args[0])

    def get_container(self, id):
        try:
            return self.client.containers.get(id)
        except errors.NotFound:
            raise ResourceNotFound()
```

6.3. VM.py

```python
from datetime import datetime
from typing import Union
import libvirt
from libvirt import libvirtError
from pydantic import BaseModel
import xml.dom.minidom
from lxml import etree
from exceptions import (
    APIError, ConnectionFailed, ResourceNotFound,
    ResourceAlreadyRunning, ResourceNotRunning, ResourceRunning
)

class DomainObj(BaseModel):
    name: Union[str, None] = None
    memory: int
    vcpu: int
    source_file: str

    class Config:
            schema_extra = {
                "example": {
                    "name": "vm1",
                    "memory": 500000,
                    "vcpu": 1,
                    "source_file": "/var/lib/libvirt/isos/debian-11.6.0-amd64-netinst.iso"
                }
            }

class VM():
    def list_vms(self):
        conn = self.libvirt_connect()
        domains = {}
        domains = conn.listAllDomains()
        jsonList = []
        for dom in domains:
            jsonList.append({"uuid": dom.UUIDString(),
                             "name": dom.name(),
                             "isActive": dom.isActive(),
                             "status": self.get_vm_status(dom).get("desc"),
                             "isPersistent": dom.isPersistent()})
        return jsonList

    def list_snapshots(self, id):
        conn = self.libvirt_connect()
        dom = self.get_vm(id)
        snapshots = dom.listAllSnapshots()
        jsonList = []
        for elem in snapshots:
            xmlDesc = etree.fromstring(elem.getXMLDesc())
            creationTime = datetime.fromtimestamp(int(xmlDesc.find("creationTime").text))
            jsonList.append({"name": elem.getName(),
                             "timestamp": creationTime,
                             "state": xmlDesc.find("state").text,
                             "isCurrent": elem.isCurrent(),
                             "memorySnapshot": xmlDesc.find("memory").get("snapshot")})
        return jsonList

    def list_storage_vol(self):
        conn = self.libvirt_connect()
        pool = conn.storagePoolLookupByName("default")
        if pool == None:
            raise APIError("Failed to locate any StoragePool objects")
        stgvols = pool.listVolumes()
        jsonList = []
        for stgvolname in stgvols:
```

```
            stgvol = pool.storageVolLookupByName(stgvolname)
            info = stgvol.info()
            jsonList.append({"name": stgvolname,
                             "path": stgvol.path(),
                             "Type": str(info[0]),
                             "Capacity": str(info[1]),
                             "Allocation": str(info[2])})
        return jsonList

    def get_vm_info(self, id):
        conn = self.libvirt_connect()
        dom = self.get_vm(id)
        state, maxmem, mem, cpus, cput = dom.info()
        return {"uuid": dom.UUIDString(),
                "name": dom.name(),
                "isActive": dom.isActive(),
                "status": self.get_vm_status(dom).get("desc"),
                "isPersistent": dom.isPersistent(),
                "OSType": dom.OSType(),
                "hasCurrentSnapshot": dom.hasCurrentSnapshot(),
                "hardware_info": {
                    "state": str(state),
                    "maxMemory": str(maxmem),
                    "memory": str(mem),
                    "cpuNum": str(cpus),
                    "cpuTime": str(cput)}}

    def start_vm(self, id, revertSnapshot):
        conn = self.libvirt_connect()
        dom = self.get_vm(id)
        try:
            if revertSnapshot:
                snapshot = dom.snapshotLookupByName(revertSnapshot)
                dom.revertToSnapshot(snapshot)
                return "Sucessfully reverted " + revertSnapshot
            state = self.get_vm_status(dom).get("state")
            if state == 3:
                dom.resume()
                return "VM sucessfully resumed"
            elif state == 5:
                dom.create()
                return "VM sucessfully restarted"
            elif state == 1:
                raise ResourceAlreadyRunning()
        except libvirtError as e:
            raise APIError(str(e))

    def stop_vm(self, id):
        conn = self.libvirt_connect()
        dom = self.get_vm(id)
        try:
            state = self.get_vm_status(dom).get("state")
            if state == 1:
                dom.suspend()
                return "VM sucessfully stopped"
            else:
                raise ResourceNotRunning()
        except libvirtError as e:
            raise APIError(str(e))

    def reboot_vm(self, id):
        conn = self.libvirt_connect()
        dom = self.get_vm(id)
        try:
            state = self.get_vm_status(dom).get("state")
```

```
            if state == 1:
                dom.reboot()
                return "VM sucessfully rebooted"
            else:
                raise ResourceNotRunning()
        except libvirtError as e:
            raise APIError(str(e))

    def shutdown_vm(self, id, save, force):
        conn = self.libvirt_connect()
        dom = self.get_vm(id)
        try:
            state = self.get_vm_status(dom).get("state")
            if state == 1:
                if save:
                    dom.managedSave()
                    return "VM sucessfully saved"
                elif force:
                    dom.destroy()
                    return "VM was forced to shutdown"
                else:
                    dom.shutdown()
                    return "VM sucessfully shutdown"
            else:
                raise ResourceNotRunning()
        except libvirtError as e:
            raise APIError(str(e))

    def delete_vm(self, id):
        conn = self.libvirt_connect()
        dom = self.get_vm(id)
        try:
            state = self.get_vm_status(dom).get("state")
            if state != 1:
                dom.undefine()
                return "Requested Ressource was sucessfully deleted"
            else:
                raise ResourceRunning()
        except libvirtError as e:
            raise APIError(str(e))

    def delete_snapshot(self, id, name):
        conn = self.libvirt_connect()
        dom = self.get_vm(id)
        try:
            snapshot = dom.snapshotLookupByName(name)
            snapshot.delete()
            return "Snapshot " + name + " sucessfully deleted"
        except libvirtError as e:
            raise APIError(str(e))

    def delete_storage_vol(self, id):
        conn = self.libvirt_connect()
        dom = self.get_vm(id)
        try:
            pool = conn.storagePoolLookupByName("default")
            if pool == None:
                raise APIError("Failed to locate any StoragePool objects.")
            state = self.get_vm_status(dom).get("state")
            if state != 1:
                stgvol = pool.storageVolLookupByName(dom.name()+".qcow2")
                stgvol.delete()
                return "Storage volume sucessfully created"
            else:
                raise ResourceRunning()
```

```
196         except libvirtError as e:
197             raise APIError(str(e))
198 
199     def run_vm_xml(self, body):
200         conn = self.libvirt_connect()
201         xmlconfig = xml.dom.minidom.parseString(body)
202         new_xml = xmlconfig.toprettyxml()
203         try:
204             dom = conn.defineXMLFlags(new_xml, 0)
205             dom.create()
206             return {"Following guest sucessfully booted": {"Id" : dom.UUIDString(), "Name": dom.nam
207                     "info": "For further parameters visit: https://libvirt.org/formatdomain.html"}
208         except libvirtError as e:
209             raise APIError(str(e))
210 
211     def run_vm_json(self, obj: BaseModel):
212         conn = self.libvirt_connect()
213         dict = obj.dict()
214         xmlconfig = f"""
215             <domain type='kvm'>
216               <name>{dict.get("name")}</name>
217               <memory>{dict.get("memory")}</memory>
218               <vcpu>{dict.get("vcpu")}</vcpu>
219               <os>
220                   <type>hvm</type>
221                   <boot dev='hd'/>
222                   <boot dev='cdrom'/>
223               </os>
224               <clock offset='utc'/>
225               <on_poweroff>destroy</on_poweroff>
226               <on_reboot>restart</on_reboot>
227               <on_crash>destroy</on_crash>
228               <devices>
229                   <emulator>/usr/bin/qemu-system-x86_64</emulator>
230                   <disk type='file' device='cdrom'>
231                      <source file='{dict.get("source_file")}'/>
232                      <driver name='qemu' type='raw'/>
233                      <target dev='hda'/>
234                    </disk>
235                    <disk type='file' device='disk'>
236                      <driver name='qemu' type='qcow2'/>
237                      <source file='/var/lib/libvirt/images/{dict.get("name")}.qcow2'/>
238                      <target dev='vda'/>
239                    </disk>
240                    <interface type='network'>
241                       <source network='default'/>
242                    </interface>
243                    <input type='mouse' bus='ps2'/>
244                    <graphics type='vnc' port='-1' listen='127.0.0.1'/>
245                </devices>
246             </domain>"""
247         try:
248             dom = conn.defineXMLFlags(xmlconfig, 0)
249             dom.create()
250             if dom:
251                 return {"Following guest sucessfully booted": {"Id" : dom.UUIDString(), "Name": dom
252                         "info": "For further parameters visit: https://libvirt.org/formatdomain.htm
253         except libvirtError as e:
254             raise APIError(str(e))
255 
256     def create_snapshot(self, id, snapshot_name):
257         conn = self.libvirt_connect()
258         dom = self.get_vm(id)
259         xmlconfig = f"""
```

```
260             <domainsnapshot>
261                 <name>{snapshot_name}</name>
262             </domainsnapshot>"""
263         try:
264             dom.snapshotCreateXML(
265                 xmlconfig.format(snapshot_name=snapshot_name),
266                 libvirt.VIR_DOMAIN_SNAPSHOT_CREATE_ATOMIC
267             )
268             return "Snaphot of " + dom.name() + " was sucessfully created"
269         except libvirtError as e:
270             raise APIError(str(e))
271 
272     def create_storage_vol(self, name):
273         conn = self.libvirt_connect()
274         xmlconfig = f"""
275             <volume type='file'>
276                 <name>{name}.qcow2</name>
277                 <allocation>0</allocation>
278                 <capacity>0</capacity>
279                 <target>
280                     <path>/var/lib/virt/images/{name}.qcow2</path>
281                     <format type='qcow2'/>
282                     <permissions>
283                         <owner>107</owner>
284                         <group>107</group>
285                         <mode>0744</mode>
286                         <label>vir_image_t</label>
287                     </permissions>
288                 </target>
289             </volume>"""
290         try:
291             pool = conn.storagePoolLookupByName("default")
292             stgvol = pool.createXML(xmlconfig, 0)
293             return "Storage volume sucessfully created "
294         except libvirtError as e:
295             raise APIError(str(e))
296 
297     def libvirt_connect(self):
298         try:
299             conn = libvirt.open('qemu:///system')
300             return conn
301         except libvirt.libvirtError:
302             raise ConnectionFailed()
303 
304     def get_vm(self, id):
305         conn = self.libvirt_connect()
306         try:
307             return conn.lookupByUUIDString(id)
308         except libvirtError:
309             raise ResourceNotFound()
310 
311     def get_vm_snapshots(self, id):
312         conn = self.libvirt_connect()
313         dom = self.get_vm(id)
314         try:
315             return dom.listAllSnapshots()
316         except libvirtError as e:
317             raise APIError(e.args[0])
```

```
319     def get_vm_status(self, dom):
320         state, maxmem, mem, cpus, cput = dom.info()
321         str = ""
322         if state == 1:
323             str = "Running"
324         elif state == 3:
325             str = "Stopped(Paused)"
326         elif state == 5:
327             str = "Not Running"
328         return {"state": state,
329                 "desc": str}
330
```

6.4. Security.py

```python
from contextlib import asynccontextmanager
from datetime import datetime, timedelta
from typing import Union,Annotated
import time
from fastapi import Depends, HTTPException, Security , status
from fastapi.security import (
    OAuth2PasswordBearer,
    OAuth2PasswordRequestForm,
    SecurityScopes
)
from jose import JWTError, jwt
from passlib.context import CryptContext
from pydantic import BaseModel, ValidationError

SECRET_KEY = "b18cddaef06d377b97f01a3de062a2e1ec2cca8cf9a37b543786b9227155ae64"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30

fake_users_db = {
    "johndoe": {
        "username": "johndoe",
        "full_name": "John Doe",
        "email": "johndoe@example.com",
        "hashed_password": "$2b$12$EixZaYVK1fsbw1ZfbX3OXePaWxn96p36WQoeG6Lruj3vjPGga31lW",
        "disabled": False,
    },
    "alice": {
        "username": "alice",
        "full_name": "Alice Chains",
        "email": "alicechains@example.com",
        "hashed_password": "$2b$12$gSvqqUPvlXP2tfVFaWK1Be7DlH.PKZbv5H8KnzzVgXXbVxpva.pFm",
        "disabled": False,
    },
}

class Token(BaseModel):
    access_token: str
    token_type: str

class TokenData(BaseModel):
    username: str | None = None
    scopes: list[str] = []

class User(BaseModel):
    username: str
    email: str | None = None
    full_name: str | None = None
    disabled: bool | None = None

class UserInDB(User):
    hashed_password: str

oauth2_scheme = OAuth2PasswordBearer(
    tokenUrl="token",
    scopes={"basic": "Basic ead permission on resources",
            "advanced": "Write and run permission on resources"
    }
)

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
```

```python
def verify_password(plain_password, hashed_password):
    return pwd_context.verify(plain_password, hashed_password)

def get_password_hash(password):
    return pwd_context.hash(password)

def get_user(db, username: str):
    if username in db:
        user_dict = db[username]
        return UserInDB(**user_dict)

def authenticate_user(fake_db, username: str, password: str):
    user = get_user(fake_db, username)
    if not user:
        return False
    if not verify_password(password, user.hashed_password):
        return False
    return user

def create_access_token(data: dict, expires_delta: timedelta | None = None):
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() + timedelta(minutes=15)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt

async def get_current_active_user(
        security_scopes: SecurityScopes, token: Annotated[str, Depends(oauth2_scheme)]
):
    if security_scopes.scopes:
        authenticate_value = f'Bearer scope="{security_scopes.scope_str}"'
    else:
        authenticate_value = "Bearer"
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": authenticate_value},
    )
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        if username is None:
            raise credentials_exception
        token_scopes = payload.get("scopes", [])
        token_data = TokenData(scopes=token_scopes, username=username)
    except (JWTError, ValidationError):
        raise credentials_exception
    user = get_user(fake_users_db, username=token_data.username)
    if user is None:
        raise credentials_exception

    for scope in security_scopes.scopes:
        if scope not in token_data.scopes:
            raise HTTPException(
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail="Not enough permissions",
                headers={"WWW-Authenticate": authenticate_value},
            )
    if user.disabled:
        raise HTTPException(status_code=400, detail="Inactive user")
    return user
```

6.5. Exceptions.py

```python
class ResourceNotFound(Exception):
    def __init__(self, message="Requested Resource not found -> Check identifier"):
        self.message = message
        super().__init__(self.message)

class ArgumentNotFound(Exception):
    def __init__(self, message):
        self.message = message+" -> Search the documentation for the right argument"
        super().__init__(self.message)

class ImageNotFound(Exception):
    def __init__(self, message="Image not found -> Search for spellling mistakes"):
        self.message = message
        super().__init__(self.message)

class APIError(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)

class ConnectionFailed(Exception):
    def __init__(self, message="Failed to establish a connection to daemon -> Contact your system administrator"):
        self.message = message
        super().__init__(self.message)

class ResourceNotRunning(Exception):
    def __init__(self, message="Requested resource is not running"):
        self.message = message
        super().__init__(self.message)

class ResourceAlreadyRunning(Exception):
    def __init__(self, message="Requested resource is already running"):
        self.message = message
        super().__init__(self.message)

class ResourceRunning(Exception):
    def __init__(self, message="Requested Resource is still running, please shutoff/stop resource before continue"):
        self.message = message
        super().__init__(self.message)
```

Literaturverzeichnis

[1] C. Baun, *Projektbeschreibung DESIGN DaaS_ohne_Grenzen,* 2022.

[2] M. Chae, H. Lee und K. Lee, „A performance comparison of linux containers and virtual machines using Docker and KVM," *Cluster and Computer, 22 (Suppl 2),* pp. 1765 - 1775, 2019.

[3] VMware, „Virtualization Overview," 2006. [Online]. Available: http://www.vmware.com/pdf/virtualization. pdf .. [Zugriff am April 2023].

[4] R. P. Goldberg, „Architectural principals for virtual computer systems," *Havard University, Cambridge MA,* pp. 22-27, 1973.

[5] F. Sierra-Arriaga, R. Branco und B. Lee, „Security Issues and Challenges for virtualization," *ACM Computing Surveys,* 2020.

[6] RedHat, „What is KVM (online)," 2022. [Online]. Available: https://www.redhat.com/de/topics/virtualization/what-is-KVM. [Zugriff am April 2023].

[7] R. Morabito, J. Kjällman und M. Komu, „Hypervisors vs. lightweight virtualization: a performance comparison," *IEEE International Conference on cloud engineering,* pp. 386 - 393, 2015.

[8] R. Pike, D. Presotto, K. Thompson, H. Trickey und W. P., „The use of name spaces in plan 9," pp. 72-76, 1993.

[9] E. Casalicchio und S. Iannucci, „The state-of-the-art in container technologies: Application, orchestration and security," *Concurrency and Computation: Practice and Experience, 32,* 2020.

[10] Docker, „Docker Documentation," 2023. [Online]. Available: https://docs.docker.com/get-started/. [Zugriff am April 2023].

[11] A. M. Potdar, D. G. Narayan, S. Kengond und M. M. Mulla, „Performance evaluation of docker container and virtual machine," *Procedia Computer Science , 171,* 1419 - 1428 2020.

[12] REST, „What is REST?," 2023. [Online]. Available: https://restfulapi.net. [Zugriff am Mai 2023].

[13] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte und D. Vrgoc, „Foundations of JSON schema," *Proceedings ot the 25th international conference on World Wide Web,* April 2016.

[14] B. Pollard, HTTP/2 in Action, Simon and Schuster, 2019.

[15] Docker Engine API, „Documentation,“ 2023. [Online]. Available: https://docs.docker.com/engine/api/v1.42/.

[16] docker-py, „Docker SDK for Python,“ 2023. [Online]. Available: https://docker-py.readthedocs.io/en/stable/ . [Zugriff am April 2023].

[17] Portainer, „Portainer.io,“ Juni 2023. [Online]. Available: https://www.portainer.io.

[18] Libvirt, „API Documentation,“ 2023. [Online]. Available: https://libvirt.org/docs.html.

[19] libvirt-python, „SDK Documentation,“ 2023. [Online]. Available: https://libvirt-python.readthedocs.io.

[20] Proxmox, „Proxmox VE API,“ 2023. [Online]. Available: https://pve.proxmox.com/wiki/Proxmox_VE_API#Python. [Zugriff am Mai 2023].

[21] Merixstudio, „9 Best Practices to implement in REST API development,“ Mai 2023. [Online]. Available: https://www.merixstudio.com/blog/best-practices-rest-api-development/?utm_source=youtube&utm_medium=video&utm_campaign=rest-api-vlog.

[22] Fast API, „Documentation,“ 2023. [Online]. Available: https://fastapi.tiangolo.com/tutorial/.

[23] B. Chen, P. Zavarsky, R. Ruhl und D. Lindskog, „A Study of the Effectiveness of CSRF Guard,“ *IEEE Third International Conference on Privacy, Security, Risk and Trust and IEEE Third International Conference on Social Computing,* 2011.

[24] Python, 2023. [Online]. Available: https://www.python.org.