HAW HAMBURG

BACHELORTHESIS
Tobias Dittmann

Multi-Tenancy Microservice Architektur für einen datenschutzkonformen Austausch von Dateien

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Tobias Dittmann

Multi-Tenancy Microservice Architektur für einen datenschutzkonformen Austausch von Dateien

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

Betreuender Prüfer: Prof. Dr. Jens von Pilgrim
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 31. Dezember 2021

Tobias Dittmann

Thema der Arbeit

Multi-Tenancy Microservice Architektur für einen datenschutzkonformen Austausch von Dateien

Stichworte

Multi-Tenancy, Microservices, SaaS, DSGVO

Kurzzusammenfassung

Vorgestellt wird eine mandantenfähige und datenschutzkonforme Microservice Architektur zum Versenden und Empfangen von Dateien. Das Ziel der Arbeit ist ein lauffähiger Prototyp. Die Arbeit zeigt den Verlauf der Umsetzung von der Anforderungsermittlung über die Konzeption der Anwendung bis hin zur Realisierung und den dabei aufgetretenen Hürden. Folgend wurde der fertiggestellte Prototyp evaluiert und Optimierungen herausgearbeitet. Betrachtet wurde die Erfüllung der Anforderungen an das System.

Tobias Dittmann

Title of Thesis

Multi-Tenancy microservice architecture for privacy related file transfers

Keywords

Multi-Tenancy, Microservices, SaaS, GDPR

Abstract

This thesis will introduce a multi-tenant and privacy related microservices architecture for sending and receiving files. The main goal is a working prototype. The thesis will guide through the process of requirements engineering, various concepts and implementation details. After this the prototype has been evaluated and optmizations were presented. The implementation of the requirements was used to evaluate the system.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

ACL Access Control List.

AWS Amazon Web Services.

CD Continuous Deployment.

CI Continuous Integration.

CISPE Cloud Infrastructure Services Providers in Europe.

CRI Container Runtime Interface.

CSI Container Storage Interface.

DI Dependency Injection.

DSGVO Datenschutz Grundverordnung.

FIQL Feed Item Query Language.

HATEOAS Hypertext As The Engine Of Application State.

HOTP HMAC-based One Time Password.

IoC Inversion of Control.

JPA Java Persistance API.

JVM Java Virtual Machine.

MJML Mailjet Markup Language.

MVC Model-View-Controller.

OCI Open Container Initiative.

OIDC OpenID Connect.

ORM Object-Relational Mapping.

OTP One Time Password.

RBAC Role Based Access Control.

REST Representational State Transfer.

SaaS Software as a Service.

SEO Search Engine Optimization.

SPA Single Page Application.

SSR Server Side Rendering.

TDD Test Driven Development.

TOTP Time-based One Time Password.

VM Virtuelle Maschine.

1 Einleitung

1.1 Datenschutz

Jeder Mensch ist am Schutz seiner Daten interessiert. Kommerzielle Anbieter erstellen Profile über das Verhalten und die Interessen der Benutzer. Geschaltete Werbung basiert auf diesen Informationen, um den Nutzer stärker anzusprechen mit dem Ziel, gesteigerte Umsätze und Klickquoten zu erzielen. Ethisch stellt sich auch die Frage, inwiefern die Weitergabe und Verarbeitung von personenbezogenen Daten vertretbar ist. Lange Zeit lag dies im Ermessen der Unternehmen, die selbst bestimmen konnten, wie sie mit den erfassten Informationen umgehen [MM17].

Nicht nur der Schutz vor Werbung liegt im Interesse der Benutzer. Die Internetkriminalität, gerade der Identitätsdiebstahl, hat in den letzten Jahren immer weiter zugenommen. Das ist jedoch nur indirekt ein Problem des Datenschutzes. Mit genügend Informationen lässt sich zwar das Verhalten einer Person studieren und imitieren, jedoch müssen für den tatsächlichen Identitätsdiebstahl weitere Dokumente wie Ausweiskopien vorliegen. Dementsprechend sollte es immer im Interesse von Datenverarbeitern sein, die Daten ihrer Kunden und Nutzer so gut es geht zu schützen und nicht weiterzugeben [Bra15].

Um die Weitergabe und das angesprochene uneingewilligte Sammeln von Daten zu verhindern, verabschiedete die Europäische Union eine Verordnung, die 2016, mit einem Anpassungszeitraum von zwei Jahren, in Kraft getreten ist. Der Name dieser Verordnung lautet Datenschutz Grundverordnung (DSGVO). Als Ziele der Verordnung werden, wie bereits erwähnt, die Einschränkung der Verarbeitung personenbezogener Daten, das Recht auf Schutz personenbezogener Daten als Grundrecht und Kontrolle des uneingewilligten Datenverkehrs angestrebt. Die DSGVO betrifft alle Menschen, die in der Europäischen Union leben. Dementsprechend müssen sich auch Unternehmen, die Kunden in der EU haben, an diese Verordnung halten [Com16].

1.2 Problemstellung

Im Zuge der Modernisierung und Digitalisierung von Geschäftsprozessen ist der Austausch von Dateien ein wichtiger Faktor. Viele Prozesse, sowohl zwischen Unternehmen als auch Privatpersonen, beinhalten das Senden und Empfangen von Dateien. Fotografen senden die bearbeiteten Bilder an den Kunden, Unternehmen stellen Dokumente für Kunden bereit oder aber, der aktuellen Corona-Pandemie geschuldet, empfangen Lehrkräfte Arbeitsergebnisse von Schülern oder Studenten. Betrachtet man bestehende Lösungen und achtet dabei auf die in Abschnitt 1.1 vorgestellte DSGVO, so sind rechtlich gesehen viele dieser Prozesse in Deutschland für Unternehmen ungeeignet. Die meisten Produkte kommen aus den USA oder werden dort betrieben. Da dort jedoch personenbezogene Daten gespeichert werden und der Staat aufgrund des Patriot Acts[1] berechtigt ist, diese Daten einzusehen, gelangen gegebenenfalls Kundendaten oder personenbezogene Metadaten wie IP-Adressen ohne das Einverständnis der Kunden in die Hände Dritter [Foi21].

1.3 Motivation

Aufgrund eines meiner Hobbies, der Fotografie, muss ich oft Dateien an Dritte versenden. Da ich dafür jedoch keine Cloud-Anbieter, wie Dropbox oder Google-Drive, aufgrund der in Abschnitt 1.2 angesprochenen Probleme in Bezug auf den Datenschutz verwenden will, und der Versand über E-Mail meist aufgrund von Dateigrößen problematisch ist, wäre die angedachte Applikation eine Erleichterung für mich und auch andere. So kann ich sichergehen, dass keine personenbezogenen Daten von mir, aber auch von meinen Kontakten gesammelt und weitergegeben werden.

1.4 Zielsetzung

Das Ergebnis dieser Arbeit soll ein lauffähiger Prototyp sein, der einen datenschutzkonformen Austausch von Dateien ermöglicht. In den folgenden Kapiteln wird der Schwerpunkt auf die Konzeption und Umsetzung dieses Prototypen gelegt.

[1] https://www.fincen.gov/resources/statutes-regulations/usa-patriot-act

1.5 Aufbau der Arbeit

Die Thesis ist in sechs Kapitel gegliedert. Nach einer kurzen Einleitung in die Problemstellung und Motivation im ersten Kapitel werden in Kapitel 2 die Anforderungen analysiert und dokumentiert. Dazu werden mittels Requirements Engeneering funktionale Anforderungen, aber auch beispielsweise Randbedingungen an das System herausgearbeitet und festgehalten.

Im darauf folgenden Kapitel 3 wird die aus den Anforderungen abgeleitete Architektur konzipiert und aus verschiedenen Perspektiven beleuchtet. Die Perspektiven beziehen sich dabei auf die vier Sichten des 4+1 Modells nach Kruchten [Kru95, S. 42-50]. Dazu werden Konzepte wie die gewählte Architektur diskutiert und vorgestellt sowie Vorkehrungen für ein datenschutzkonformes System getroffen.

Das Kapitel 4 beschäftigt sich mit der Realisierung der vorgestellten Konzepte und deren Betrieb. So werden hier verwendete Methodiken, aber auch Frameworks, Libraries oder die Bereitstellung vorgestellt.

Darauf folgt in Kapitel 5 eine Evaluation des entwickelten Prototypen in Bezug auf die gestellten Anforderungen.

Das letzte Kapitel 6 soll die Arbeit abrunden, indem es ein kurzes Fazit sowie einen Ausblick auf mögliche Verbesserungen oder Erweiterungen des Systems bietet.

2 Anforderungsanalyse

Zur Entwicklung einer qualifizierten Architektur müssen Anforderungen bekannt und dokumentiert sein. Dazu gehören die Anforderungen an das eigene System, die benutzten Komponenten von Drittanbietern, aber auch der Einfluss der Stakeholder. Im Folgenden wird mittels des Requirements Engineerings genau diese Dokumentation geschaffen [PR15, S. 3-5].

2.1 Funktionsweise

Die zu entwickelnde Webanwendung soll es Kunden ermöglichen, Dateien mit Kontakten zu teilen oder anzufordern. Der Kunde kann über eine Weboberfläche einen sogenannten Pool erstellen. Ein Pool ist eine Sammlung von Dateien und kann in zwei Typen unterschieden werden. Ein *DOWNLOAD*-Pool stellt Dateien für Kontakte des Kunden bereit, wohingegen ein *UPLOAD*-Pool den Dateiupload von Kontakten an den Kunden erlaubt. Dabei ist ein Kontakt nur berechtigt, die von ihm hochgeladenen Dateien zu sehen. Unabhängig von der Art des Pools erfolgt der Zugriff auf diesen mittels eines personalisierten Links in einer E-Mail Einladung. Der Text dieser Einladung wird vom Kunden pro Pool angegeben. Ein Pool hat standardmäßig eine begrenzte Lebenszeit und wird nach Ablauf eines vom Kunden definierten Zeitfensters gesperrt. Ab diesem Zeitpunkt ist keine Interaktion mehr zwischen den Kontakten und dem Pool möglich. Bei Bedarf kann die zeitliche Limitierung entfernt werden. Um gescheiterten Uploads, beispielsweise durch eine instabile Internetverbindung, entgegenzuwirken, sollen Benutzer nach Wiederaufnahme der Verbindung Uploads fortführen können. Gleichzeitig sollen alle hochgeladenen Dateien auf Viren überprüft werden.

2.2 Stakeholder

Zuerst müssen die Stakeholder identifiziert werden, da diese als Betroffene des Projekts einen direkten oder indirekten Einfluss auf das System haben. Sie sind in aller Regel die wichtigste Quelle für die Ermittlung von Anforderungen [PR15, S. 3-4].

Ein wichtiger Stakeholder ist das **Management**. Da es sich um ein Cloud-Produkt handelt, welches nicht direkt auf einen Kunden zugeschnitten, sondern der breiten Masse zur Verfügung gestellt wird, entscheidet dieses in den Anfangsphasen wesentlich über das Featureset. Nicht wegzudenken ist außerdem der **Datenschutzbeauftragte**. Die Anwendung soll datenschutzkonform nach DSGVO konzipiert werden. Der Datenschutzbeauftragte dient hier als Kontrollinstanz um sicherzustellen, dass so wenig personenbezogene Daten wie möglich gespeichert oder einsehbar sind. Das ist auch im Interesse der **Kunden**. Diese sind die Hauptnutzer der Plattform und werden im Laufe der Zeit maßgeblich zu Featurewünschen oder Änderungen beitragen. Hier muss wieder das Management entscheiden, welche Featurerequests sinnvoll für die Gesamtanwendung sind und welche nicht. Um Features umsetzen zu können, bedarf es zudem der **Entwickler**. In deren Interesse steht klar das Verringern von Komplexität und die Erhaltung der Stabilität. Sie sorgen für die Produktentwicklung, aber auch den Betrieb der Software. Das liegt daran, dass keine dedizierte Abteilung für den Betrieb der Software existiert, sondern das DevOps Konzept seine Anwendung findet [EGHS16]. Zuletzt wären da noch die **Nutzer** der Anwendung. Diese beinhalten die Kunden, aber auch deren Kontakte. Der umgesetzte Quellcode soll sicher und konsistent sein. Verwaiste Abschnitte oder unbehandelte Fehler sind zu vermeiden. Zur Einhaltung dieser qualitativen Aspekte kontrolliert die *Qualitätssicherung* regelmäßig Änderungen am Code und dem System.

Folglich ergeben sich die folgenden Stakeholder:

- Management
- Entwickler
- Kunden
- Nutzer
- Datenschutzbeauftragter
- Qualitätssicherung

2.3 Systemkontext

Im Zuge der Dokumentation des Umfangs des zu entwickelnden Systems gibt der Systemkontext Klarheit über Kontextgrenzen. Das bedeutet im Detail, dass man sich vor Augen führt, was genau zum eigenen System gehört und was von außerhalb kommt. Der Systemkontext zeigt auf, welche Komponenten relevant für Interaktionen mit dem zu planenden System sind. Die Abhängigkeit von fremden Systemen kann dabei in beide Richtungen verlaufen, sodass das System abhängig von Fremdkomponenten sein kann, aber auch eine Abhängigkeit eines fremden Systems werden kann [PR15, S. 13-20].

Im Mittelpunkt des Systemkontexts steht immer das eigene System. Um dieses herum baut sich der relevante Kontext auf. Das ist in der folgenden Abbildung 2.1 zu erkennen.

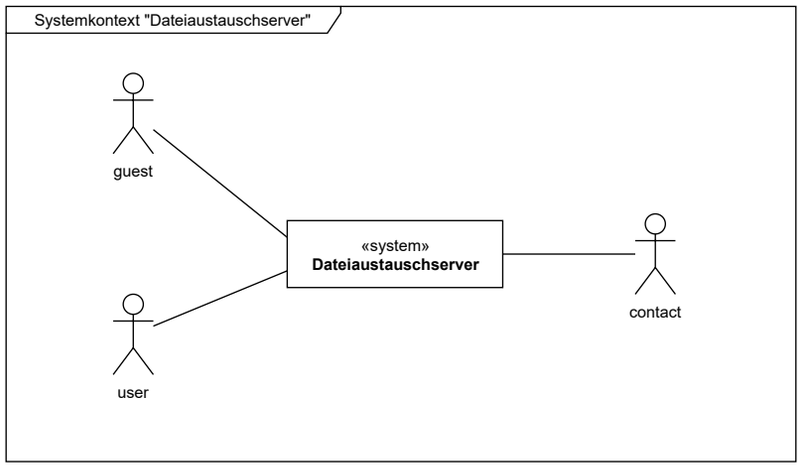


Abbildung 2.1: UML Systemkontextdiagramm des Dateiaustauschservers

Wie in Abbildung 2.1 gezeigt kristallisieren sich drei Akteure heraus. Das sind der *guest*, der *user* und der *contact*. Der *guest* (im Folgenden Gast) ist ein neuer Besucher der Anwendung, welcher noch kein eigenes Kundenkonto besitzt. Der *user* (im Folgenden Benutzer) besitzt bereits ein Konto und kann auf die Daten der Mandaten zugreifen, bei denen der Benutzer Mitglied ist. Der *contact* (im Folgenden Kontakt) ist ein Kontakt eines Mandanten, mit dem der Austausch von Dateien stattfindet. Dabei besitzt der Kontakt kein eigenes Konto und kann sich auch nicht anmelden. Er kann lediglich via ihm zugesandten Links aus Einladungen mit dem System interagieren.

2.4 Anforderungen

In Absprache mit den Stakeholdern, hauptsächlich dem Management, und unter Beachtung der in Abschnitt 2.1 beschriebenen Funktionsweise wurden Anforderungen an das System definiert. Diese wurden im Prozess des Requirements Engineering in funktionale sowie nicht-funktionale Anforderungen unterschieden [PR15, S. 8-9].

2.4.1 Nicht-funktionale Anforderungen

Bei der Konzeption eines Systems gilt es, nicht nur der Funktionalität, sondern auch den umliegenden Faktoren Aufmerksamkeit zu widmen. So haben gewisse, nicht-funktionale Anforderungen starke Auswirkungen auf die Implementierung, da diese bestimmte Bedingungen voraussetzen, welche bei der Umsetzung berücksichtigt werden müssen. Dabei kann man grundsätzlich in zwei Kategorien unterscheiden [PR15, S. 8], welche im Folgenden vorgestellt werden.

Die **Qualitätsanforderungen** beziehen sich auf die Qualität des zu entwickelnden Systems. Hierzu zählen beispielsweise Anforderungen an die Performanz, Austauschbarkeit oder Skalierbarkeit. **Randbedingungen** hingegen sind einschränkende Bedingungen in Bezug auf die Umsetzung der funktionalen und Qualitätsanforderungen. Beispiele hierfür wären Bedingungen für einen bestimmten Prozess der Anwendungslogik oder Vorgaben zur Separierung von mandatenspezifischen Daten. [PR15, S. 8-9]. Im Folgenden sind die festgelegten Qualitätsanforderungen und Randbedingungen festgehalten.

Qualitätsanforderungen

QA/01 Das System muss kostengünstig und schnell horizontal skalierbar sein.

QA/02 Der Dateispeicher muss flexibel sein und ohne große Umgestaltung ausgetauscht werden können.

QA/03 Prozesse, die keine direkte Antwort benötigen, müssen asynchron verarbeitet werden, um die Antwortzeit der API zu verringern.

QA/04 Umgebungen wie Testsysteme müssen nach der Veröffentlichung einer neuen Version automatisch aktualisiert werden.

Randbedingungen

RB/01 Die Anwendung soll von Grund auf datenschutzkonform konzipiert werden. Somit sollen Server und Storage im Einflussbereich der DSGVO, folglich innerhalb der Europäischen Union, betrieben werden, vorzugsweise in Deutschland. Gleichzeitig werden von den Nutzern des Systems nur die technisch notwendigen, personenbezogenen Daten gespeichert und auch nur so lange, wie rechtlich vorgeschrieben. Ganz kann nicht auf diese Daten verzichtet werden, da beispielsweise im Falle der unberechtigten Verbreitung von rechtlich geschütztem Material der Verantwortliche ausfindig gemacht werden muss.

RB/02 Die Daten der einzelnen Mandanten sind voneinander zu trennen. Dazu gehört auch, die Möglichkeit bieten zu können, dass Mandanten ein eigenes Schema in der Datenbank haben, um mandantenspezifische Erweiterungen zu ermöglichen. Das Gleiche gilt auch für hochgeladene Dateien. Diese müssen zudem verschlüsselt abgelegt werden.

RB/03 Auf Social-Logins soll verzichtet werden. Ein Benutzer soll sich lediglich mit einer Kombination aus Nutzername und Passwort authentifizieren können. Der Nutzername stellt dabei die E-Mail Adresse des Benutzers dar. Zudem muss die Möglichkeit geboten werden, sich mit mindestens zwei Faktoren authentifizieren zu können.

RB/04 Ein Benutzer kann gleichzeitig mehreren Mandanten angehören. Ein neuer Mandant muss jederzeit vom Gast selbst erstellt werden können. Das darf keinen administrativen Aufwand mit sich ziehen.

RB/05 Jeder Mandant hat eine eindeutige Subdomain und kann optional eine eigene Domain aufschalten.

RB/06 Ein Dateiupload, der ungewollt unterbrochen worden ist, kann binnen 5 Tagen ohne weiteres Zutun vom Benutzer weitergeführt werden. Eine Limitierung auf die Nutzung des selben Gerätes ist dabei gewollt.

RB/07 Es ist eine dem Representational State Transfer (REST) entsprechende API für die Client-Server Kommunikation bereitzustellen.

RB/08 Auflistungen von Daten müssen möglichst flexibel gefiltert werden können.

RB/09 Mehrere Dateien sind als gebündeltes ZIP-Archiv herunterladbar.

RB/10 Hochgeladene Dateien müssen auf Schadsoftware geprüft werden.

RB/11 Zur Bedienung der API ist eine Weboberfläche als Frontend umzusetzen.

RB/12 Die Verbindung zwischen der Weboberfläche und dem Server muss durch SSL geschützt sein.

2.4.2 Funktionale Anforderungen

Funktionale Anforderungen sind jene, die die Funktionalität und das Verhalten eines Systems beschreiben. Dokumentieren kann man diese als Use-Cases. Ein Use-Case fasst eine Menge von Aktionen zusammen und beschreibt einen Anwendungsfall für die Interaktion zwischen Akteuren und dem System [PR15, S. 69].

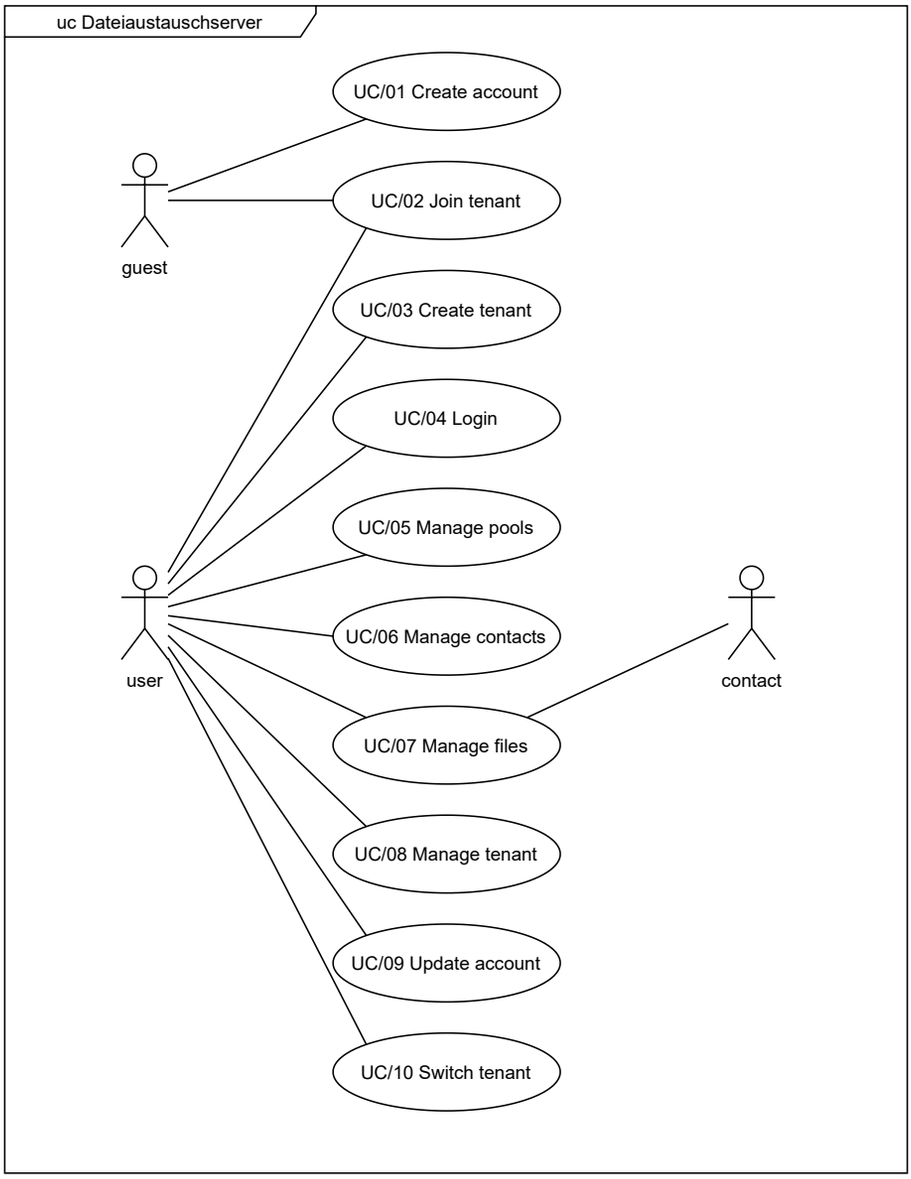


Abbildung 2.2: UML Use-Case-Diagramm

Das in Abbildung 2.2 zu sehende Use-Case-Diagramm gibt einen Überblick über die Funktionalität des Systems. Es bringt die ermittelten Use-Cases in den Systemkontext. Auf der nächsten Abstraktionsebene werden nun die informationslosen Use-Cases mittels Use-Case-Beschreibungen spezifiziert. Eine solche Beschreibung ist eine textuelle Darstellung des Ablaufs der Anwendungsfälle. Zusätzlich zum Hauptszenario werden auch alternative Abläufe betrachtet sowie Vor- und Nachbedingungen definiert [PR15, S. 72-74]. Diese Beschreibungen sind in Anhang A.1 tabellarisch dokumentiert.

Als Vorgehensmodell für die Entwicklung des Prototypen kommt Scrum zum Einsatz. Scrum ist ein Vorgehensmodell für die agile Entwicklung von Software. Aus den ausgeschriebenen Use-Case-Beschreibungen kann man User-Stories ableiten. User-Stories sind keine Anforderungen. Sie bieten eine Diskussionsgrundlage für die tatsächlichen Aufgaben und beschreiben kurz und knapp eine gewünschte Funktionalität. Der Backlog ist die Menge aller User-Stories. Mehrere Stories können in einem *Epic* zusammengefasst werden [Glo11, S. 130-131]. In der folgenden Auflistung der User-Stories ist die Definition eines Epics daran zu erkennen, dass ein Eintrag der Liste eine Unterliste führt.

US/01 As a guest, I want to create an account

US/02 As a user, I want to confirm my email address

US/03 As a user, I want to reset my password if forgotten

US/04 As a user, I want to update my account information

US/05 As a user, I want to upload a profile picture

US/06 As a user, I want to delete my account

US/07 As a user, I want to enable or disable MFA

US/08 As a user, I want to login to my account

US/09 As a user, I want to create a new tenant

US/10 As a user, I want to join a tenant

US/11 As a guest, I want to join a tenant

US/12 As a user, I want to delete a tenant

US/13 As a user, I want to invite members to my tenant

US/14 As a user, I want to switch my tenant

US/15 As a user, I want to remove members from my tenant

US/16 As a user, I want to manage the pools of my tenant

a) As a user, I want to create a pool

b) As a user, I want to edit a pool

c) As a user, I want to delete a pool and its data

US/17 As a user, I want to manage the contacts of my tenant

a) As a user, I want to create a contact

b) As a user, I want to edit a contact

c) As a user, I want to delete a contact

US/18 As a contact, I want to upload files to a pool

US/19 As a user, I want to upload files to a pool

US/20 As a contact, I want to download files from a pool

US/21 As a user, I want to download files from a pool

2.5 Anwendungsdomäne

Die Domäne beschreibt die Problemwelt der Anforderungen aus der Strukturperspektive. In ihr finden sich Entitäten und deren Beziehungen wieder. Zur Dokumentation und Veranschaulichung eignet sich das Domänenklassendiagramm, welches im Grunde ein vereinfachtes UML-Klassendiagramm ist [PR15, S. 75-84]. Zur Vereinfachung und der besseren Übersicht halber wurde das Domänenklassendiagramm in zwei Teildiagramme aufgeteilt. Dabei zeigt das Diagramm in Abbildung 2.3 die Benutzer- und das in Abbildung 2.4 die Poolverwaltung.

Die Benutzerverwaltung (Abbildung 2.3) zeigt die Benutzer (*User*), Mandanten (*Tenant*) und deren Beziehungen im Kontext der Zugriffsberechtigung. Ein Benutzer kann gleichzeitig mehreren Mandaten angehören und ist jedem dieser Mandanten über eine Rolle (*Role*) zugeordnet. Diese Rolle ist als Gruppe (*Group*) zu sehen, wobei eine Gruppe eine Menge von Berechtigungen (*Permission*) ist. Dies entspricht dem flachen Role Based Access Control (RBAC)-Modell [Tak17, S. 358-359] mit der Ausnahme, dass eine Gruppe keine Rollen bündelt, sondern eine Rolle ist. Demnach kann ein Benutzer nur eine Rolle besitzen. Ein Benutzer kann durch eine Einladung (*Invitation*) berechtigt werden, einem Mandanten beizutreten. Gleichzeitig besitzt ein Benutzer mehrere *RefreshToken*, einen pro aktiver Session in der Weboberfläche. Jeder Mandant hat einen *Plan*, der für spätere Quota und andere Limitierungen benutzt werden soll, welcher aber nicht Teil

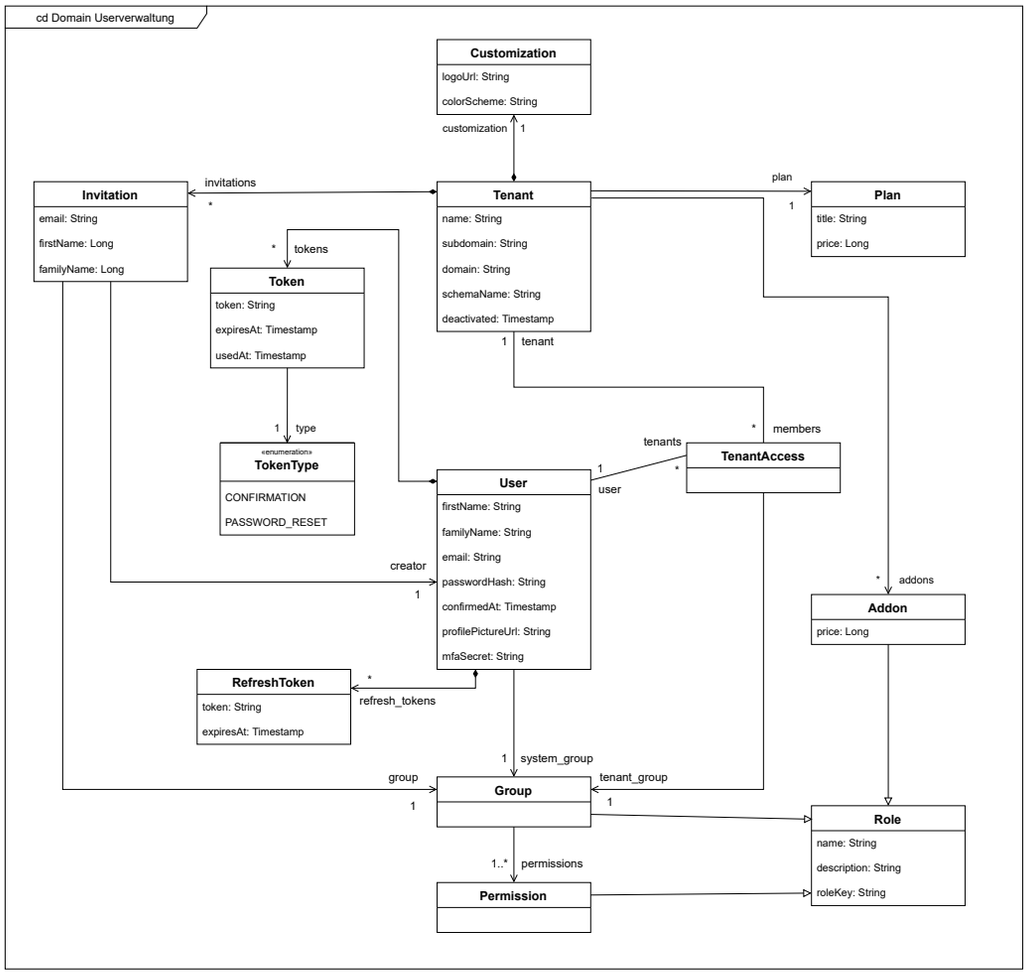


Abbildung 2.3: Domänenklassendiagramm für die Benutzerverwaltung

des Prototypen ist. Zusätzlich kann ein Mandant Erweiterungen (*Addon*) dazubuchen. Auch diese sind für die spätere Verwendung bereits als Tabelle angelegt. Eine Abrechnungskomponente oder das eigene Hinzubuchen von Features aus der Anwendung heraus ist nicht Teil des Prototypen. Zuletzt ist jedem Mandanten eine *Customization* zugeordnet, die Informationen für die Umgebung und das Erscheinungsbild des Mandanten, wie beispielsweise eine URL zu einem Logo, enthält.

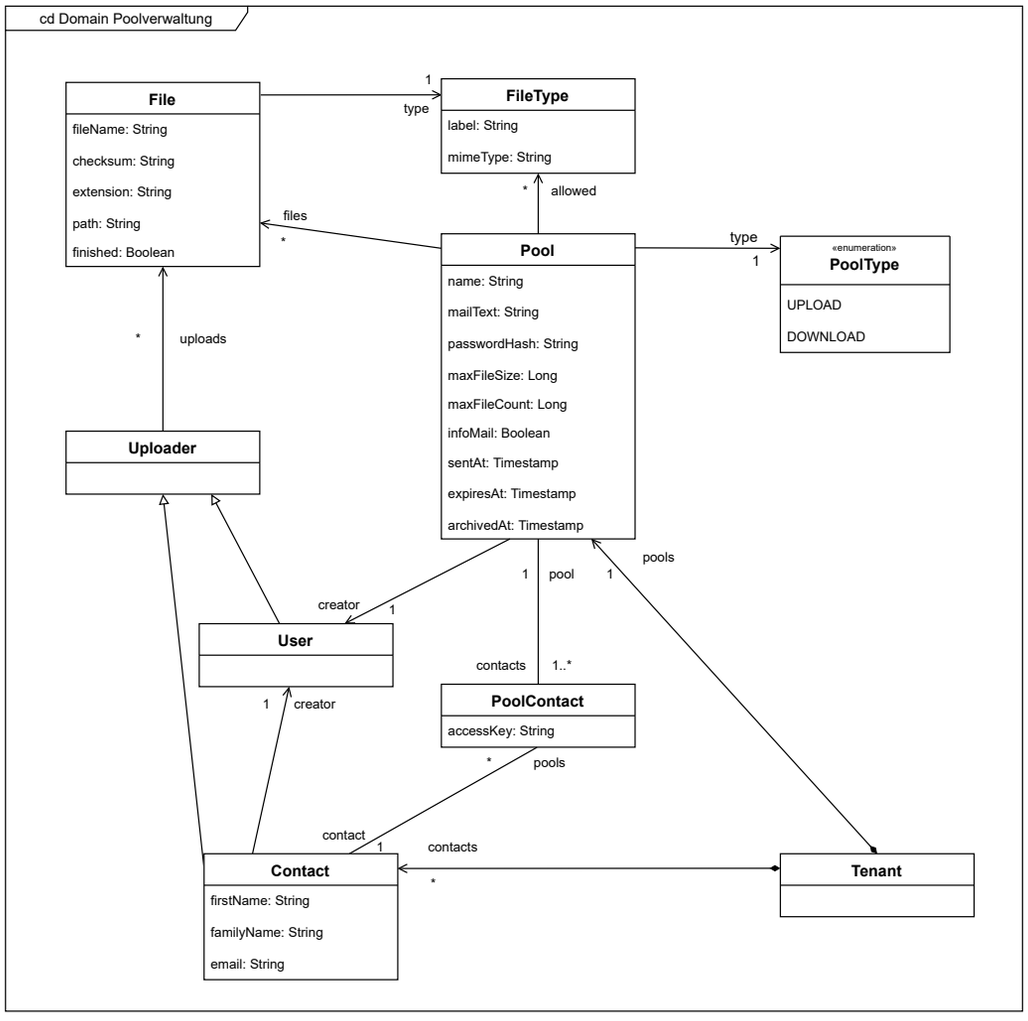


Abbildung 2.4: Domänenklassendiagramm für die Poolverwaltung

Die Poolverwaltung zeigt die Zusammenhänge der eigentlichen Businesslogik. Im Zentrum der Poolverwaltung steht der *Pool*. Er enthält Dateien (*File*) und wird von einem Benutzer (*User*) erstellt und gepflegt. Die Attribute von *User* und *Tenant* wurden weggelassen, da diese in Abbildung 2.3 bereits abgebildet werden. Der Pool hat, wie in der

Vorstellung der Funktionsweise bereits erwähnt, einen Typen. Je nach Typ unterscheiden sich Limitierungen, dennoch sind die Basiseigenschaften identisch. Der Typ wird durch die Assoziation zwischen *PoolType* und *Pool* festgelegt. Damit externe Personen, in diesem Fall die Kontakte (*Contact*) des Mandanten, auf den Pool und damit verbunden die Dateien zugreifen können, werden diese über die *PoolContact*-Klasse mit dem Pool verbunden. Diese Klasse enthält einen Schlüssel, mithilfe dessen Nutzung der Kontakt für den Zugriff auf den Pool autorisiert wird. Des Weiteren ist eine Komposition zwischen Mandant und Pool, sowie Mandant und Kontakt dokumentiert, da es ohne einen Mandanten keine mandantenspezifischen Daten, wie es diese beiden Entitäten sind, geben kann.

3 Konzeption

Nach der Dokumentation der Anforderungen kann mithilfe dieser das System konzipiert werden. Dabei werden in diesem Kapitel Konzepte vorgestellt und erläutert, mit denen sich die Anforderungen umsetzten lassen. Im Wesentlichen werden die Architektur vorgestellt und Design-Entscheidungen begründet.

3.1 Infrastruktur

Die Infrastruktur bietet die Grundlage für den Betrieb der Software. Durch die Wahl der richtigen Infrastruktur kann einigen Problemen, die bei der Entwicklung auftreten könnten, bereits im Vorwege vorgebeugt werden. Daher wird in diesem Abschnitt die gewählte Infrastruktur, auf der die Konzeption der restlichen Architektur beruht, vorgestellt und die Wahl begründet. Dies beginnt mit einer Einführung in die zugrundeliegenden Konzepte und deren Vorteile in Bezug auf die Entwicklung des Prototypen.

3.1.1 Docker

Für die Isolation von Prozessen ist eine Virtuelle Maschine (VM) eine bekannte Lösung. Docker[1] bietet eine Plattform zum Betrieb von Software über Containerisierung. Dabei schafft sie die gleiche Isolation wie die einer VM, benötigt aber kein eigenes Betriebssystem. Es handelt sich bei Docker-Containern um isolierte Prozesse, die alle den Kernel des unterliegenden Betriebssystems nutzen. Das bietet die gleichen Freiheiten wie die einer VM. Dazu zählen zum Beispiel der mögliche Verzicht der Installation von Software auf dem Host-Betriebssystem oder eine einfache Versionsverwaltung der genutzten Komponenten. So arbeiten alle Entwickler mit den gleichen Versionen, ohne die zusätzliche Last eines ganzen Betriebssystems mit sich tragen zu müssen [Ind18, S. 219 f.].

[1] https://www.docker.com/

Um einen Docker-Container starten zu können, wird ein *Image* benötigt. Ein Docker-*Image* ist ein Paket aus der ausgelieferten Software und deren Abhängigkeiten. Definiert wird ein Image über eine *Dockerfile*, welche Instruktionen zum Paketieren des Projekts enthält. Dazu gehören unter anderem Instruktionen zum Kompilieren des Quellcodes, welche Dateien im *Image* präsent sein müssen und Anweisungen zum Start der Anwendung. Eine *Dockerfile* kann in mehrere *Stages* unterteilt werden. Diese helfen bei der Organisation der Instruktionen, führen aber auch zu Optimierungen in der Verarbeitungszeit, um das *Image* zu bauen. Das liegt in der Natur, wie Docker die *Stages* ausführt. *Stages* und einzelne Anweisungsschritte werden lokal in einem Cache festgehalten und nur dann erneut ausgeführt, wenn sich die Instruktion oder eine referenzierte Datei geändert hat [Ind18, S. 219 f.].

Docker wurde zur Vereinfachung der lokalen Entwicklung verwendet. Es eignet sich gerade durch seine intuitive Benutzung für die lokale Entwicklung und empfiehlt sich gegenüber Alternativen wie buildah[2]. So kann schnell eine Umgebung aufgesetzt werden, die dem Echtsystem strukturell entspricht. Hier kommt ein weiteres Tool zum Einsatz: Docker Compose[3]. Es bietet die Möglichkeit, über eine Datei deskriptiv eine Umgebung zu definieren und diese einfach über jeweils ein Kommando zu starten oder wieder zu beenden. Dabei ist zu beachten, dass Docker-Container keine Persistenz bieten. Nach einem Neustart sind die Daten der vorherigen Instanz nicht mehr vorhanden. Das ist für die lokale Entwicklung kein Problem. Sollte es aber gewünscht sein, die Daten über die Lebensdauer des Containers zu halten, bietet Docker mit *Volumes* eine Möglichkeit, Daten zu persistieren [Ind18, S. 230 f.].

3.1.2 Kubernetes

Dank der Flexibilität von Docker können Applikationen in isolierten Prozessen ausgeführt werden. Dennoch fehlt die Verwaltung des Lebenszyklus eines solchen Prozesses, welche die Containerisierung nicht bietet. Hier kommt die Orchestrierung ins Spiel. Durch automatische Lastverteilung, dynamisches Starten neuer Instanzen oder Neustarten eines abgestürzten Containers kann das System hochverfügbar und skalierbar bereitgestellt werden [Ind18, S. 235].

[2] https://github.com/containers/buildah
[3] https://docs.docker.com/compose/

Zum Einsatz kommt das bewährte Framework Kubernetes[4]. Entwickelt von Google wurde der Quellcode 2014 mit dem Ziel veröffentlicht, die eigene, stetig wachsende Infrastruktur skalierbar und verfügbar zu betreiben. Im Kontext von Kubernetes existieren zwei Unterscheidungen von Servertypen: *Master* und *Worker*. Jeder Server ist eine *Node*, welche zusammen als Einheit das *Cluster* bilden. Ein *Cluster* besteht demnach aus mehreren *Nodes*, in denen es mindestens eine *Worker-* und eine *Master-Node* geben muss. Die *Master-Node* agiert als das Gehirn des Systems und bildet die *control pane*, bestehend aus vier Komponenten: dem *API server*, *scheduler*, *controller manager* und *etcd*. Der *API server* stellt Schnittstellen für die *Worker-Nodes* und die anderen Komponenten der *control pane* bereit. Um die Ressourcen des *Clusters* möglichst effizient zu nutzen, verteilt der *scheduler* die Arbeitslast intelligent über die einzelnen *Worker-Nodes*. Zur Verwaltung der *Nodes*, aber auch aller anderen Komponenten des *Clusters* dient der *controller manager*. Er sorgt unter anderem dafür, dass Komponenten skaliert, repliziert oder nach einem Fehler neugestartet werden. Dafür müssen jedoch Metainformationen, wie beispielsweise die gewünschte Anzahl an Replikationen pro Container, gespeichert werden. Dies geschieht im *etcd*, einem hochverfügbaren und konsistenten Datenspeicher. In ihm werden die Konfigurationen des *Clusters* abgelegt und nachvollzogen. Die *Worker-Nodes*, die die eigentlichen Prozesse ausführen, bestehen wiederum aus drei Komponenten: dem *kubelet*, *kube-proxy* und der *container runtime*. Die *container runtime* dient dazu, die Container auszuführen. Dazu wird ein Open Container Initiative (OCI) basiertes Image benötigt, zu dem auch die Images von Docker gehören. Diese sind kompatibel mit dem von der *container runtime* bereitgestellen Container Runtime Interface (CRI), welches die tatsächlichen Prozesse ausführt. Der *kubelet* Agent dient zur Kommunikation zwischen den *Workern* und der *control pane*, genauer gesagt dem *API server*. Er verwaltet die *Node* und gibt die Anweisungen des *Masters* weiter. Zu guter Letzt fehlt noch der *kube-proxy*. Dieser leitet Anfragen an die jeweiligen unterliegenden Komponenten weiter [Say20, S. 5 f.].

In Kubernetes existieren verschiedene Typen von Ressourcen. Diese Ressourcen bilden zusammen das Kernkonzept von Kubernetes und dessen Architektur. Um Ressourcen auf dem Cluster zu erstellen, werden diese textuell beschrieben und die resultierenden Beschreibungen auf das Cluster angewendet [Say20, S. 5]. Folgend werden die für diese Arbeit wichtigsten Ressourcen im Kontext der Architektur vorgestellt.

[4] https://kubernetes.io/

Pod

Ein Pod bezeichnet eine zusammengehörige Gruppe von Containern. Der *scheduler* sorgt dafür, dass alle Container auf der gleichen physischen Node laufen. In den allermeisten Fällen umfasst ein Pod jedoch nur einen Container. Der Pod ist die kleinste und auf der untersten Ebene befindliche Einheit in Kubernetes. Die Container in einem Pod teilen sich das selbe Netzwerkinterface und den Speicher. Jeder Pod erhält eine eigene IP. Somit können alle Containter innerhalb eines Pods über localhost und den entsprechenden Port kommunizieren [Say20, S. 6].

Deployment

Das Deployment dokumentiert den gewünschten Zustand der Pods. Intern arbeitet es mit einem *ReplicaSet*. Das Deployment bestimmt beispielsweise die gewünschte Anzahl an Replikationen der Software. Zur Einhaltung dieser Vorgabe überwacht das ReplicaSet die Pods und startet neue, sollten Pods unerwartet beendet werden. Deployments bieten einen einfachen, abstrakten Zugriff auf die Pods, um zum Beispiel Softwareaktualisierungen einzuspielen. Bei einer neuen Version kann das Deployment benachrichtigt werden. Dieses kümmert sich dann in Verbindung mit dem ReplicaSet darum, dass nacheinander Pods mit der neuen Version die veralteten Pods ersetzen [Say20, S. 10].

Service

Zur Kommunikation werden Services verwendet. Ein Service erhält wie ein Pod eine IP, jedoch bleibt diese IP erhalten. Dabei bündelt ein Service, ähnlich wie der Pod Container, mehrere Pods. Ein Service agiert als Load Balancer vor den Pods und abstrahiert so die Notwendigkeit von Clients, mit Pods direkt sprechen zu müssen. Zudem kann so Skalierbarkeit und Verfügbarkeit erreicht werden, da Pods wie bereits beschrieben innerhalb eines Deployments zu jeder Zeit hoch- und heruntergefahren werden können. Der Service sorgt dafür, dass die Anfragen immer zu einem verfügbaren Pod durchgestellt werden [Say20, S. 9]. Es gibt drei Arten von Services: *ClusterIP*, *NodePort* und *LoadBalancer*. *LoadBalancer* sind dabei meist von externen Anbietern, es gibt aber auch Open Source Implementationen. Sie haben eine eigene, feste, öffentliche IP und bieten so eine Schnittstelle von außen in das Cluster zu den gewünschten Pods. Der *NodePort*-Typ öffnet einen zufälligen Port auf der Node, auf der der Pod läuft. Über diesen Port kann ebenfalls von

außen auf den Service zugegriffen werden. Dabei kann zusätzlich konfiguriert werden, ob nur auf der Node, auf der der Service und die Pods liegen, der Port geöffnet wird oder ob alle Nodes diesen Port öffnen und Anfragen nach Erhalt innerhalb des Clusters an die entsprechende Node weiterleiten. Möchte man keine Möglichkeit bieten, von außerhalb auf das Cluster zuzugreifen, so bietet sich *ClusterIP* als Typ an. Dieser Typ sorgt lediglich für den Erhalt einer IP im internen Netz und bietet keine Schnittstelle nach außen an. Die Kommunikation mit dem Service ist somit nur innerhalb des Clusters möglich [Say20, S. 317 f.].

Ingress

Ein Ingress ist wie ein Service vom Typ LoadBalancer eine zentrale Eintrittsmöglichkeit zum Zugriff auf ein Cluster. Dabei sendet dieser eingehenden Datenverkehr an einen unterliegenden Service. Der Ingress hat im Normalfall selbst einen vorgeschalteten LoadBalancer und ersetzt einen eigenen Reverse Proxy vor Anwendungen. Er bietet so Möglichkeiten zur Lastverteilung und auf Pfaden basierendes, virtuelles Routing. Das geht mit der Eigenschaft einher, dass dieser per URL angesteuert wird und so auch SSL-Zertifikate verwenden kann. Es existieren diverse Ingress controller, am weitesten verbreitet ist jedoch der NGINX Ingress Controller[5] [Say20, S. 149].

Volume

Die Daten innerhalb der Pods und deren Containern sind flüchtig und werden bei einem Neustart nicht persistiert. Für zustandslose Anwendungen ist dies kein Problem. Möchte man jedoch Datenbanken betreiben oder benötigt einen persistenten Speicher zum Ablegen von Dateien, so benötigt man Volumes. Ein Volume kann je nach Konfiguration von mehreren Pods, aber auch mehreren Nodes genutzt werden. Es existieren mehrere Typen von Volumes, so zum Beispiel lokale Volumes, die auf der Node verankert sind, aber auch externe Speicherquellen. Dazu ist Kubernetes über das Container Storage Interface (CSI) erweiterbar. So können beispielsweise Cloud Speicher von externen Anbietern als Volume angehängt und benutzt werden [Say20, S. 10 f.].

Kubernetes passt durch die angesprochene Skalierbarkeit sehr gut zu den Anforderungen (siehe **QA/01**). Es bietet eine solide Grundlage und eine Reihe von Funktionen wie

[5] https://kubernetes.github.io/ingress-nginx/

das automatisierte Lebenszyklusmanagement und die dynamische Skalierung, die den reibungslosen und hochverfügbaren Betrieb der Architektur gewährleisten können.

3.2 Architektur

Im Folgenden wird die gewählte Architektur für das System beschrieben und erläutert. Begonnen wird dies mit einer Einführung in das Konzept der umgesetzten Architektur, gefolgt von der Präsentation der aus den Anforderungen abgeleiteten Komponenten.

3.2.1 Microservices & Monolithen

Als Architekturkonzept wurde das der Microservices gewählt. Die Microservice Architektur gewinnt immer mehr an Bedeutung bei Startups und bereits bestehenden Unternehmen. Laut einer Umfrage aus dem Jahr 2021 mit über 4000 Teilnehmern haben knapp 71% der Unternehmen Microservices vollständig oder zum Teil in Benutzung [Sta21b]. Der Sinn hinter diesem Ansatz ist es, die Architektur in kleine und unabhängige Services aufzuteilen.

Konträr dazu wurde gerade in der Vergangenheit oft eine monolitischen Architektur für Anwendungen gewählt. Bei dieser wird eine Anwendung als eine Deploy-Unit ausgeliefert. Dementsprechend finden alle Transaktionen und Prozesse in nur einer Anwendung statt. Dies trägt dazu bei, dass insbesondere die Fehlerbehandlung und Einhaltung der ACID Eigenschaften vereinfacht wird. Im Gegensatz dazu handelt es sich bei Microservices um ein verteiltes System, bei dem Transaktionen über Systemgrenzen hinausgehen und so die Integrität der Daten anders sichergestellt werden muss. Des Weiteren sind Monolithen auch einfach horizontal skalierbar. Die Skalierbarkeit beschränkt sich dennoch auf das gesamte System. Das ist insofern problematisch, als dass einzelne Teile unterschiedlich viele Ressourcen verbrauchen oder benötigen. Für kleinere Anwendungen minimiert dieser Ansatz die Komplexität immens. Wird die Anwendung jedoch komplexer, so kann ein Monolith schnell kompliziert und dadurch schwer zu verstehen oder erweitern werden. Die Einarbeitung neuer Entwickler erschwert sich, und gleichzeitig muss für die Einhaltung von Entwicklungsrichtlinien gesorgt werden, dass interne Systemgrenzen nicht umgangen werden (zum Beispiel die Nutzung eines Repositories direkt in einem Controller statt einer Service-Zwischenschicht). Zudem fehlt der Architektur die Flexibilität. Für neue Features muss die ganze Anwendung neu bereitgestellt werden. Dies erschwert das

simultane Arbeiten an verschiedenen Features und die dadurch entstehende, zwingend notwendige, ständige Koordination innerhalb der Teams. Diese Kommunikation muss bei den kleineren Teams in der Entwicklung von Microservice-Architekturen auch stattfinden, doch durch das unabhängige Deployment können schneller Fehler an spezifischen Stellen beseitigt oder Features, die nur einen Service betreffen, ausgeliefert werden [Ns14]. Da nach den Anforderungen die Software flexibel skaliert werden können soll (siehe **QA/01**), eignet sich das Microservice-Konzept sehr gut. Es können einzelne Services automatisch ohne weiteres Zutun hoch- und runterskaliert werden.

Aus den Anforderungen lassen sich die Ressourcen und die damit verbundenen Services extrahieren. Jeder Service steht dabei für sich und hat die Hoheit über eine Untermenge der Daten des Systems. Dies beschränkt sich meist auf wenige Entitäten der Domäne. So wird ein Single-Point-Of-Truth erschaffen, über den die Konsistenz der Daten sichergestellt werden kann. Viele kleinere Services haben durchaus Vorteile, da kleinere Teams an diesen arbeiten und Verantwortlichkeiten leichter geklärt werden können. Jedoch geht dies auf Kosten der Komplexität. Je mehr Microservices es gibt, umso mehr müssen diese untereinander kommunizieren und zusammenarbeiten, um Prozesse zu verarbeiten. Doch je mehr Services pro Aktion kommunizieren müssen, desto komplexer wird der Erhalt einer stabilen und konsistenten Datenhaltung, gerade in Verbindung mit Multi-Tenancy. Aufgrund dessen wurden die initial ausgearbeiteten Microservices in größere Services zusammengefasst. Das betrifft vor allem den User- und Pool-Service. Der Pool-Service kombiniert die Entitäten File und Pool. Der User-Service ersetzt einen Auth- und Tenant-Service, indem er die Entitäten User und Tenant sowie das Rechtemanagement bündelt [Ns14]. So wurden die sechs initial ausgearbeiteten Services in vier finale Services zusammengefasst.

3.2.2 Sichten

Sichten sind ein weit verbreitetes Mittel, um Softwarearchitekturen zu beschreiben. Eine Architektursicht zeigt dabei eine bestimmte Perspektive des Systems. Im Folgenden werden die Sichten nach iSAQB, welche auf dem 4+1 Modell von Kruchten [Kru95, S. 42-50] basieren, vorgestellt und mit ihrer Hilfe die Architektur des System dokumentiert [Gha18, S. 84].

Kontextabgrenzung

Die Kontextabgrenzung wurde bereits in Abschnitt 2.3 in der Abbildung 2.1 dargestellt und wird hier nicht weiter beleuchtet.

Bausteinsicht

Die Bausteinsicht zeigt die Bausteine, Schnittstellen und Beziehungen einer Architektur auf. Als Baustein bezeichnet man eine Komponente der Architektur. In diesem Fall ist der Begriff Baustein eine Generalisierung für die einfachere Kommunikation. Das hilft für Klarheit in der Semantik, da so spezifische Begriffe aus beispielsweise Programmiersprachen vermieden werden [Gha18].

Die folgenden Diagramme wurden aus der Blackbox-Sicht entworfen, um die Übersichtlichkeit zu verbessern. Dabei zeigt das Diagramm in Abbildung 3.1 die Bausteine des Systems mit den Beziehungen untereinander sowie zu den Fremdsystemen. Gleichzeitig lassen sich die Artefakte der selbstentwickelten Komponenten nachvollziehen. In diesem Fall repräsentieren die Artefakte Images im Kontext von Docker.

Die Services kommunizieren untereinander synchron über die bereitgestellten REST APIs (siehe **RB/07**) und asynchron über pub/sub[6] [Sta20, S. 121 f.]. Mehr dazu folgt in Abschnitt 3.5.

[6] Publish / Subscribe Pattern

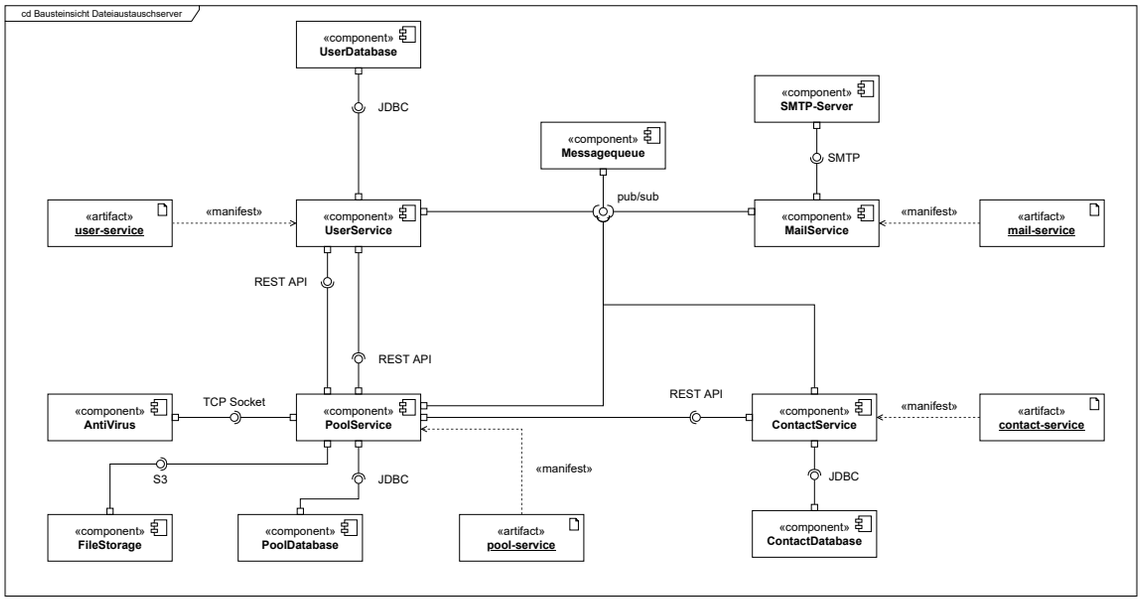


Abbildung 3.1: UML Komponentendiagramm des Systems aus Blackbox-Sicht

In der Abbildung 3.1 wird nur die interne Kommunikation zwischen den Services dargestellt. Damit nicht jeder Service frei zugänglich im Internet verfügbar ist, wird ein API-Gateway vorgeschaltet. Das Gateway ist somit der einzige Zugriffspunkt auf den Server vom Client aus (vgl. Client-Server Architektur [Sta20, S. 117]) und bietet Schutz sowie Flexibilität, indem es zentralisiert die Anfragen an die jeweiligen Services weiterleitet und so die einzelnen Services im internen Netz bleiben können. Gleichzeitig kann hier global ein Rate-Limiting oder die Autorisierung vorgenommen werden [Ind18, S. 54]. Die Einbindung des Gateways lässt sich in Abbildung 3.2 nachvollziehen.

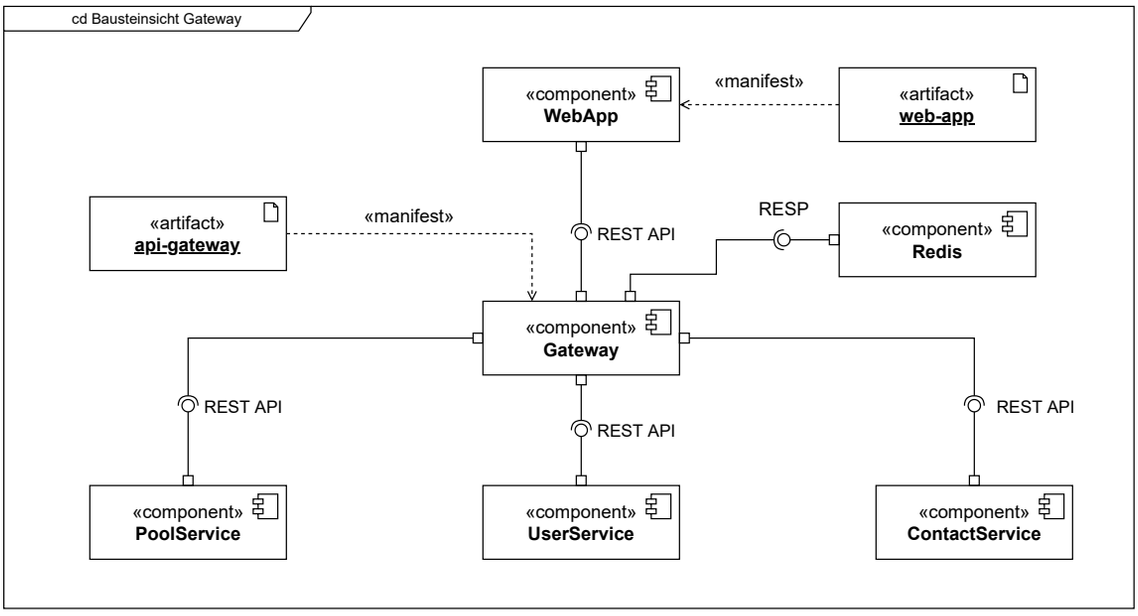


Abbildung 3.2: UML Komponentendiagramm mit Gateway aus Blackbox-Sicht

Das Gateway stellt selbst eine REST Schnittstelle bereit und delegiert die Anfragen an die unterliegenden Services. Zudem wurde an dieser Stelle ein zentralisiertes Rate-Limiting umgesetzt (siehe Unterabschnitt 3.4.3).

Laufzeitsicht

Die Laufzeitsicht zeigt genauer, wie sich das System zur Laufzeit verhält. Hier werden Abläufe und Prozeduren dargestellt, beispielsweise Aktionen zum Systemstart oder die Verbildlichung von Use-Cases der Businesslogik anhand von Szenarien [Zö15, S. 123 f.]. Folgend finden sich zur Veranschaulichung Sequenzdiagramme, die wichtige Szenarien für die Entwicklung verdeutlichen.

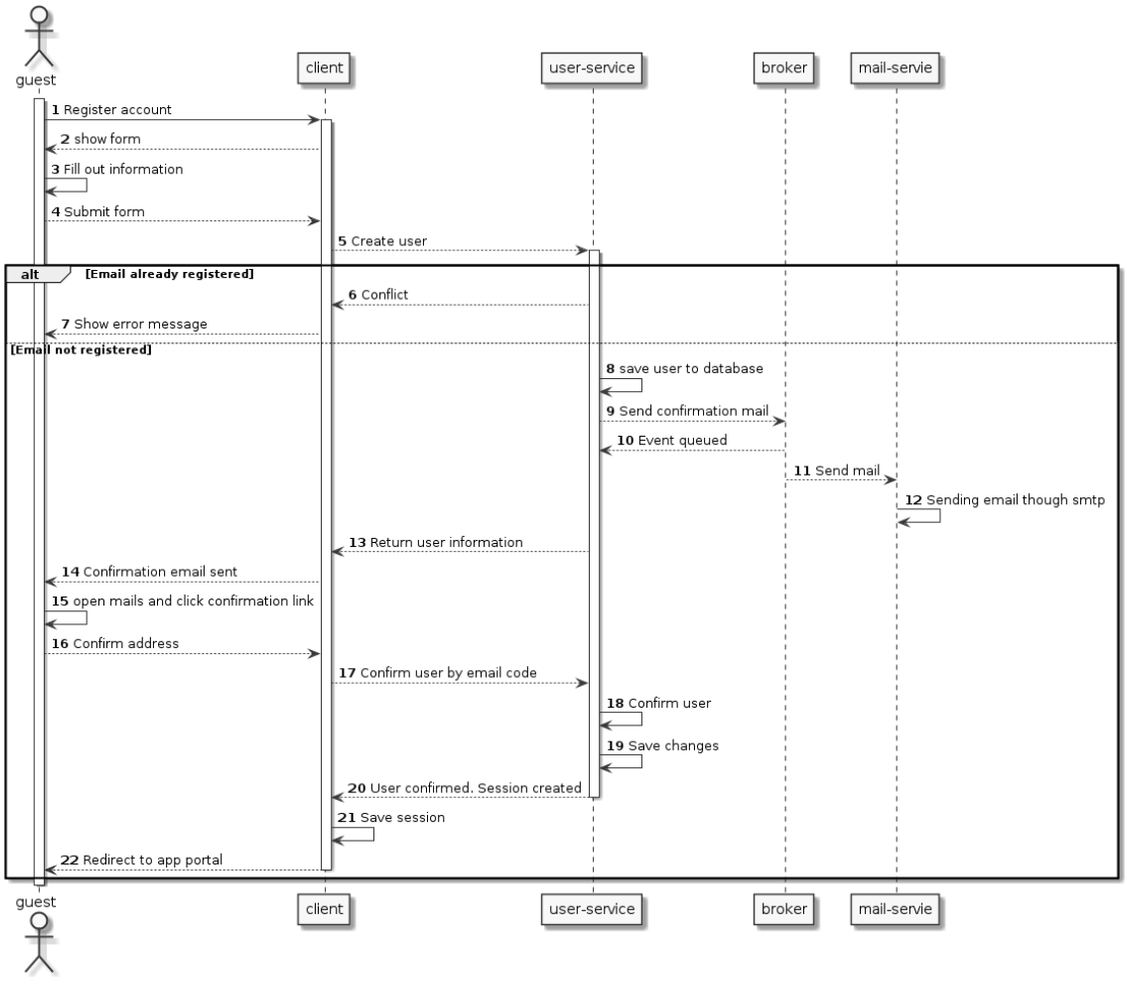


Abbildung 3.3: UML Sequenzdiagramm der Registrierung (**US/01**)

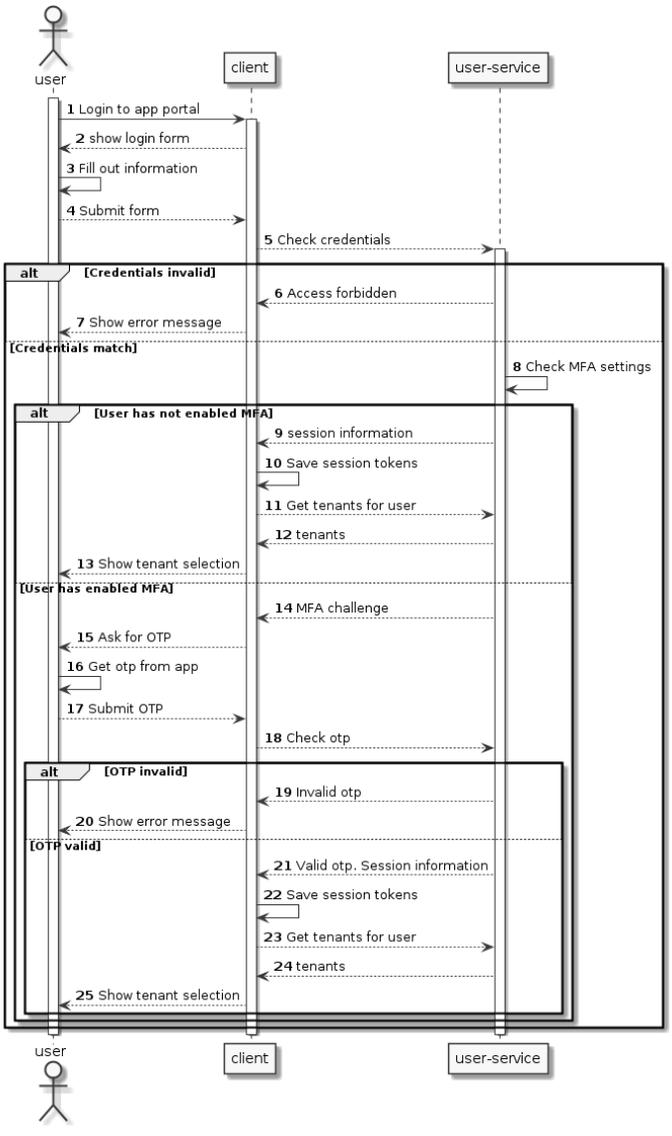


Abbildung 3.4: UML Sequenzdiagramm des Logins (**US/08**)

Die Sequenzdiagramme in den Abbildungen 3.3 und 3.3 zeigen den Prozess des Einloggens und der Registrierung. Das Ergebnis beider Szenarien ist eine neue Sitzung ohne Mandantenbezug. Der Benutzer wurde lediglich authentifiziert. Mit dieser Sitzung kann der Benutzer nun in die Umgebung eines Mandanten wechseln, in welchem der Nutzer Mitglied ist, oder einen neuen Mandanten erstellen. Die Erstellung eines neuen Mandanten ist als Beispielablauf in Abbildung 3.5 abgebildet.

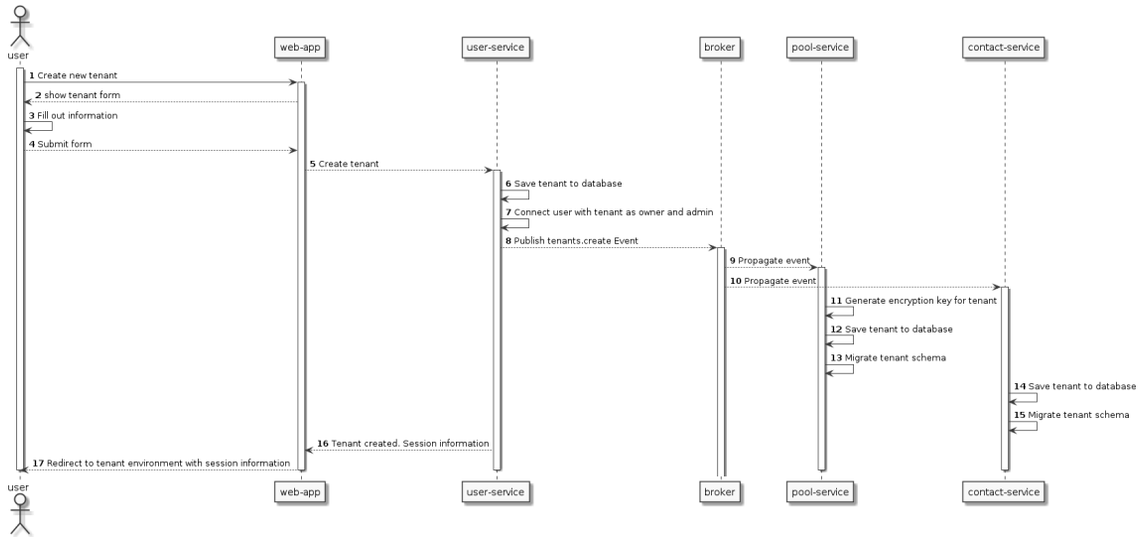


Abbildung 3.5: UML Sequenzdiagramm der Erstellung eines neuen Mandanten (**US/09**)

Sobald sich der Benutzer im Kontext eines Mandanten befindet, kann dieser mit der tatsächlichen Anwendung kommunizieren, um zum Beispiel Kontakte zu pflegen oder Dateien zu versenden. Letzteres ist als weiteres Szenario ausgearbeitet worden und in Abbildung 3.6 dokumentiert. Der Zweck dieses Szenarios ist die beispielhafte Darstellung der Erstellung eines Pools bis zum Versand. Dazu gehört die Verknüpfung von Kontakten und das Hochladen von Dateien. Die Interaktion der Kontakte mit dem Pool ist dabei nicht mit inbegriffen.

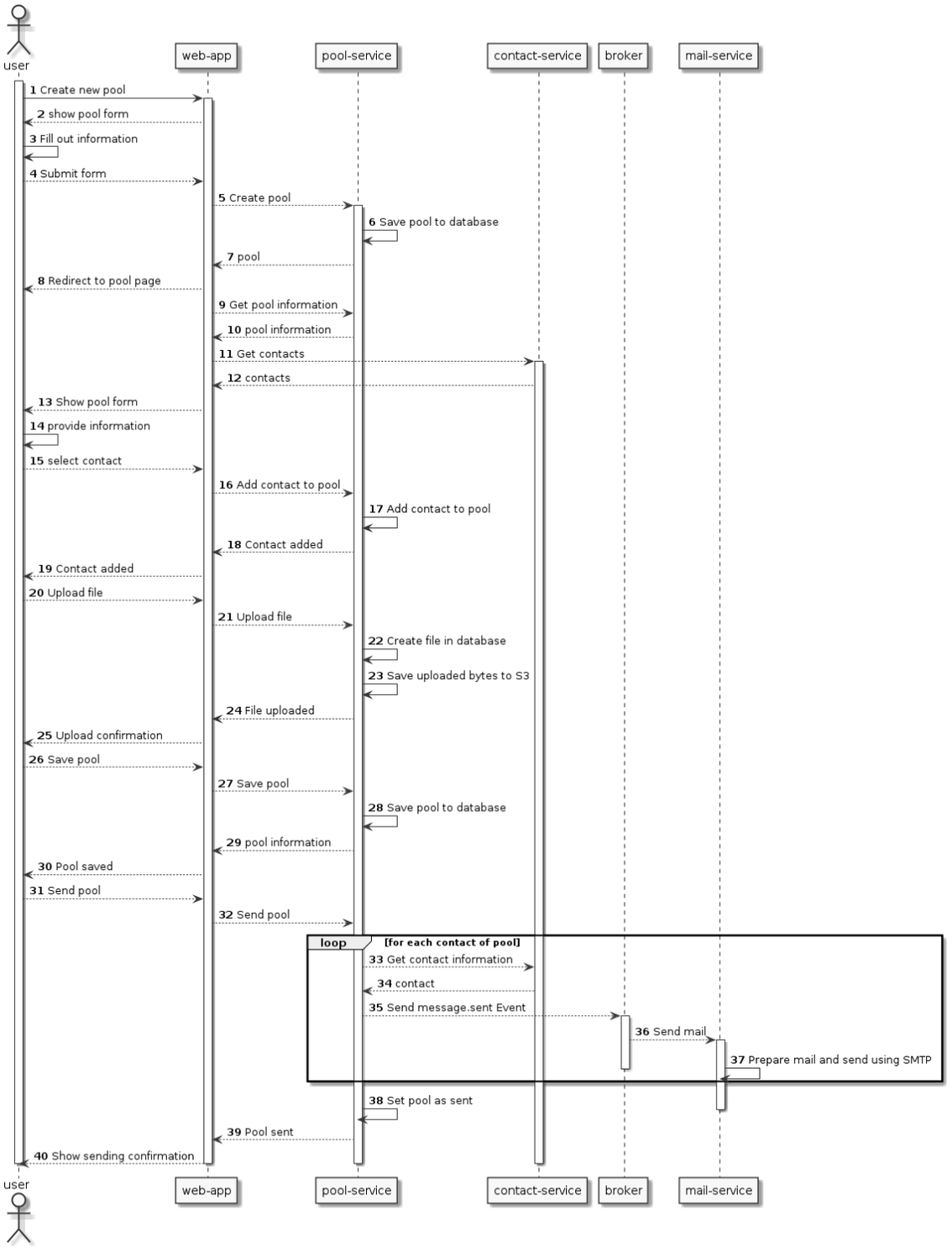


Abbildung 3.6: UML Sequenzdiagramm für das Senden eines Pools (**US/09**, **US/19**)

Verteilungssicht

Die Verteilungssicht beschreibt die Infrastruktur, genauer gesagt den Betrieb des Systems. Sie verbindet die Bausteinsicht mit der Zielumgebung und stellt auch Lösungen in Bezug auf die Anforderungen dar (siehe **QA/01** & **QA/02**) [Zö15]. Zur Darstellung wird ein UML Deploymentdiagramm genutzt.

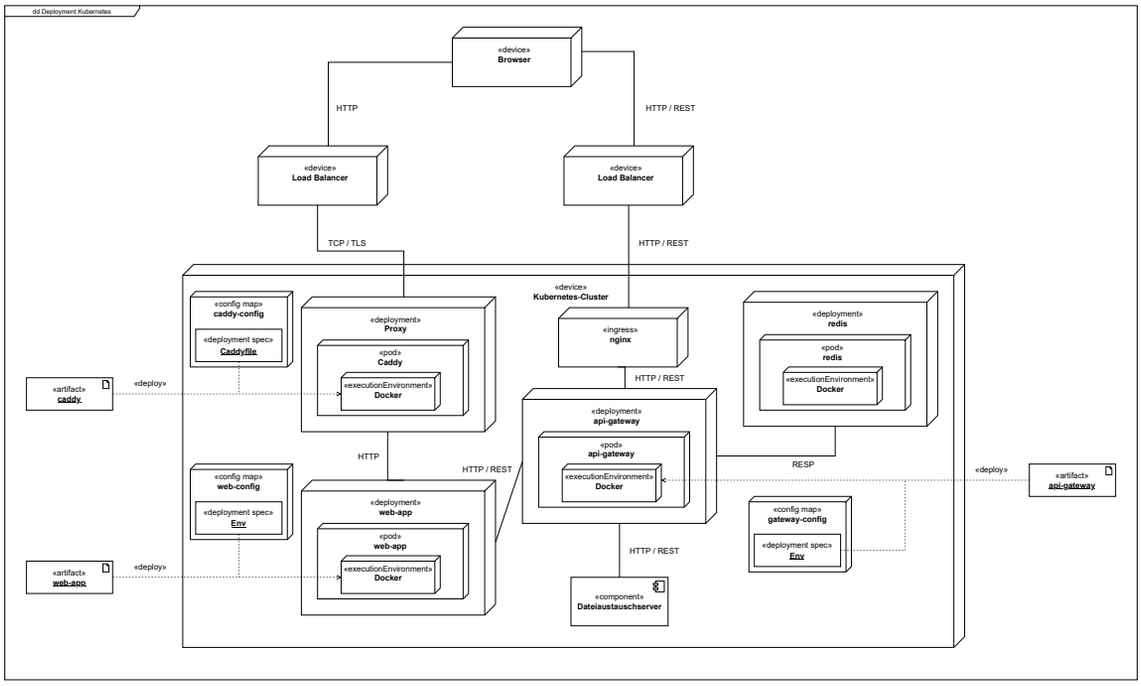


Abbildung 3.7: UML Deploymentdiagramm der grundlegenden Kubernetes Struktur

Auch dieses Diagramm wurde in mehrere Teildiagramme getrennt. Aufgrund der Wahl der Architektur wird die Anwendung nicht mehr als eine Deploy Unit, sondern in mehreren kleineren Einheiten bereitgestellt.

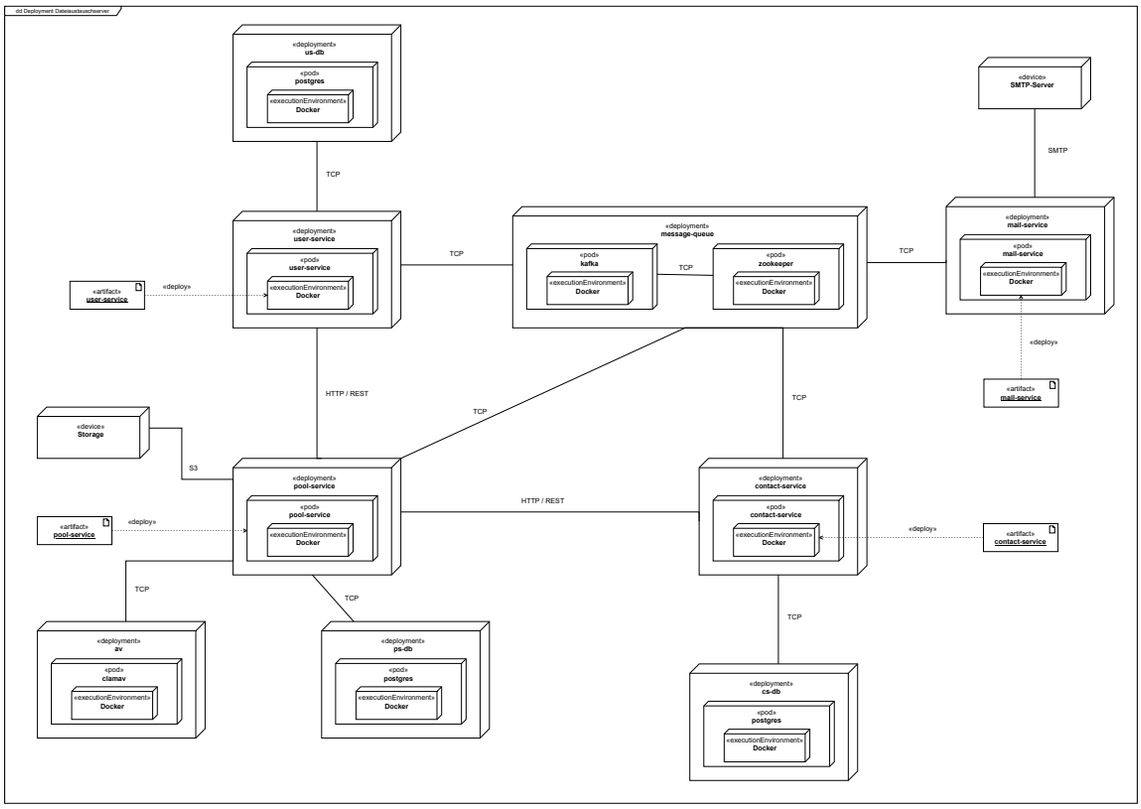


Abbildung 3.8: UML Deploymentdiagramm der Komponenten des Dateiaustauschservers

Die Artefakte der Deployments sind gleichzeitig auch die Spezifikationen für die Datenbanken. Beim Start der Applikation wird das aktuelle Datenbankschema mit dem der Applikation verglichen. Weicht dieses ab, wird der aktuelle Stand migriert. Die Konfiguration der Services erfolgt über ConfigMaps und ist in Abbildung 3.9 zu finden.

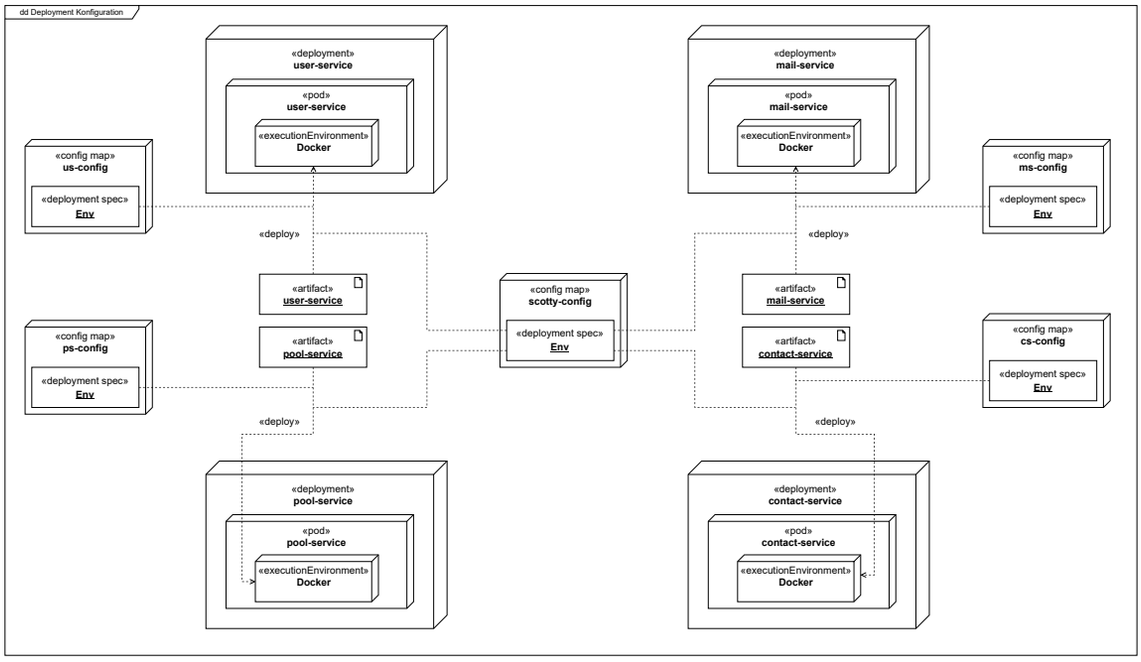


Abbildung 3.9: UML Deploymentdiagramm der Komponentenkonfiguration

3.3 Datenspeicherung

Wie bereits im vorherigen Unterabschnitt 3.2.2 angedeutet sollen die Dateien der Kunden mittels S3 API gespeichert werden. S3 wurde von Amazon für deren Simple Storage Service[7] entwickelt. Die S3 API hat sich zum Standard für Object Storage entwickelt und wird von vielen Dienstleistern unterstützt. S3 bietet ein flexibles Zugangsmanagement mittels Access Control Lists (ACLs). Die Hauptkomponenten des Object Storage sind *Buckets* und *Objects*. Ein Bucket ist dabei ein isolierter Datencontainer. Dieser enthält Objects, welche durch einen Key eindeutig identifizierbar sind. Dieser Key bietet gleichzeitig auch Möglichkeiten zur Organisation. Benutzt man als Key einen relativen Pfad, so wird eine virtuelle Ordnerstruktur innerhalb des Buckets erzeugt [SBH19].

Durch die breite Adaption und die definierte API bietet S3 eine gute Austauschbarkeit (siehe **QA/02**). So kann schnell zwischen Cloudanbietern, aber auch zu Open Source Implementationen wie Minio[8] gewechselt werden, ohne im Code Änderungen vornehmen

[7] https://docs.aws.amazon.com/s3/index.html
[8] https://min.io

zu müssen. Vorteilhaft ist auch die implizit gewonnene zentralisierte Datenhaltung. Da es sich um ein verteiltes System handelt, bei dem mehrere Instanzen auf die gespeicherten Dateien zugreifen müssen, ist dies ohnehin eine implizite Anforderung.

Das Konzept der Buckets eignet sich zudem hervorragend zur Datentrennung (siehe **RB/02**). Jeder Mandant besitzt und benutzt einen Bucket, in dem alle hochgeladenen Dateien zu finden sind. Um die Sicherheit der Daten noch weiter zu erhöhen, werden diese vor dem Transport verschlüsselt. Die Verschlüsselung erfolgt über einen symmetrischen, 256-Bit AES Schlüssel, welcher nur dem Pool-Service bekannt ist und die Anwendung zu keinem Zeitpunkt verlässt. Um die Sicherheit zu erhöhen, wird jedes Object mittels eines eigenen symmetrischen Schlüssels verschlüsselt. Dieser sogennante Data-Key wird von der SDK von Amazon generiert. Nach der Verschlüsselung der Daten wird dieser Key mit dem Schlüssel des Mandanten verschlüsselt und mit den verschlüsselten Daten an den S3 Server gesendet. Beim Download von Dateien wird eben jener Data-Key mit übertragen und mittels des Schlüssels des Mandanten entschlüsselt, um dann die eigentlichen Daten entschlüsseln zu können [Inc06, S. 363-364].

Als Backend für die Daten wird die Nutzung des Amazon Web Services (AWS) Simple Storage Service angestrebt. Dadurch, dass die S3 API von AWS entwickelt wird, stehen auch neue Funktionen als erstes bereit. Gleiches gilt für Problembehebungen oder Optimierungen. Das liegt unter anderem daran, dass das S3 Protokoll nicht fest definiert ist[9]. Es gibt also bei Fremdanbietern keine Garantie, dass die Implementation des Protokolls der aktuellen Version entspricht oder vollumfänglich zur Verfügung steht. Gleichzeitig bietet AWS auch einen guten Datenschutz[10] und ist konform mit dem CISPE-Verhaltenscodex[11]. Cloud Infrastructure Services Providers in Europe (CISPE) ist ein Zusammenschluss von Cloud-Anbietern, die ihre Dienste in Europa anbieten. Der Codex beinhaltet, beschränkt sich aber nicht auf die DSGVO. Trotz dieser Verpflichtung verhindert die Einhaltung eines Codex nicht die Möglichkeit, auf die Kundendaten zugreifen zu können, da dies auch durch die Regierung vorgegeben ist (vgl. Abschnitt 1.2). Da die Daten jedoch verschlüsselt abgelegt werden und der geheime Schlüssel den Servern von AWS nicht bekannt ist, ist die Nutzung des Cloud-Speichers von AWS aus der Sicht des Datenschutzes vertretbar. Zudem werden keine personenbezogenen Metdadaten des Benutzers an AWS übertragen und die Daten nur im deutschen Rechenzentrum in Frankfurt abgelegt.

[9] https://www.object-storage.info/the-s3-protocoll/
[10] https://aws.amazon.com/de/compliance/gdpr-center/
[11] https://aws.amazon.com/de/compliance/cispe/

Als weitere Maßnahme zur Umsetzung der Austauschbarkeit (siehe **QA/02**) wird das Adapter-Pattern eingesetzt. Dafür wird ein Interface produziert, welches alle Funktion zum Verwalten von Dateien bietet. Dieses Interface wird für den Prototypen für die S3 API und das lokale Dateisystem umgesetzt. So muss für die Einführung eines neuen Speichers nur ein neuer Adapter geschrieben werden, statt die bestehende Logik im Code zu ersetzen [Mus21].

3.4 Sicherheit

In Bezug auf die Sicherheit werden in diesem Abschnitt Konzepte und Vorkehrungen erläutert, die zum Schutz des Systems, aber auch für die Nutzer der Anwendung getroffen worden sind.

3.4.1 Antivirus

Da es sich um einen Dateiaustauschserver handelt, können Nutzer beliebige Dateien hochladen. Um Missbrauch oder der Verbreitung von Schadsoftware entgegenzuwirken, werden Dateien nach dem Hochladen einer Prüfung auf Schadsoftware unterzogen (siehe **RB/10**).

Die Suche nach Schadsoftware erfolgt über Signaturen. Wird eine Datei zur Prüfung ausgeschrieben, so erstellt die Antivirus-Software eine digitale Signatur für diese Datei. Anschließend wird diese Signatur mit aus mehreren Datenbanken bekannten Signaturen von Viren, Trojanern, Würmern und weitere Gefahren für den Computer und das Netzwerk verglichen. So lassen sich Fußspuren von Gefahren erkennen, zum Beispiel an bestimmten Abfolgen von Bytes. Da sich Schadsoftware jederzeit weiterentwickelt, werden besagte Datenbanken regelmäßig um neue Signaturen erweitert. Diese Art der Gefahrenerkennung ist schon lange im Einsatz und dementsprechend mit den gesammelten Erfahrungen der letzten Jahren gereift. Letztendlich ist keine Software zur Malware-Erkennung perfekt, jedoch hilft sie dabei, einen Großteil von Gefahren abzuwehren [Sop20].

3.4.2 Authentifizierung & Autorisierung

Ein System muss nicht nur vor externen Angriffen geschützt, sondern auch in sich sicher entworfen werden. Dazu zählt die Authentifizierung und Autorisierung. Während die

Authentifizierung vorgibt, wer mit dem System interagieren darf und wer nicht, kümmert sich die Autorisierung darum, welche Interaktionen vorgenommen werden können.

Authentifiziert werden Benutzer über eine Kombination aus Benutzername und Passwort (siehe **RB/03**). Das ist jedoch nur ein Faktor zur Bestätigung der Identität, was nicht ausreichend ist. Daher sollte man auf weitere Faktoren zur Identifikation setzen. Wenn mehrere Faktoren bei der Authorisierung benötigt werden, spricht man von einer Mehrfaktorauthentifizierung (MFA). Für diese Applikation wurde sich für eine Lösung über 2FA, also zwei Faktoren zur Identifizierung, entschieden (siehe **RB/03**). Der Wissensfaktor stellt die oben bereits genannte Kombination aus E-Mail Adresse und Passwort dar. Als weiterer Faktor kommt ein Time-based One Time Password (TOTP) ins Spiel, der an die E-Mail Adresse des Akteurs gesendet wird. Alternativ wäre noch der Versand über SMS möglich. Da dieser Transportweg jedoch wesentlich unwirtschaftlicher ist, wird darauf verzichtet, die Möglichkeit, diesen später trotzdem einzubauen, aber nicht genommen [Tak17, S. 334 f.]. One Time Passwords (OTPs) sind Einmal-Passwörter und werden zum Beispiel benutzt, um die Identität einer Person zu Überprüfen. Dabei existieren zwei grundlegende Techniken, zu deren Generierung: HMAC-based One Time Password (HOTP) und TOTP. Der HOTP generiert seine Tokens mittels eines geheimen Schlüssels und einem Counter. Bei jeder Erstellung eines Passworts wird dieser Counter inkrementiert. Aus der Kombination aus Counter und Schlüssel wird ein Hashwert mittels HMAC gezogen und dieser auf eine vorgegebene Länge gekürzt. Im Regelfall sind dies sechs Zeichen. Im Gegensatz zu HOTP wird beim TOTP kein Counter, sondern ein Zeitstempel eingesetzt. Dadurch können regelmäßig rotierende Passwörter erzeugt werden [MMPR11].

TOTP können auch ohne Versand via E-Mail oder SMS erzeugt werden. Die Generierung erfolgt dabei beispielsweise auf dem Smartphone des Benutzers, statt auf dem Server. Ein Beispiel hierfür ist die App Authy[12]. Die Funktionsweise der Passworterzeugung ist dabei identisch.

Für dieses System wird eine rollenbasierte Autorisierung angestrebt. Das bedeutet, dass dem Benutzer Rollen zugewiesen werden. Dies geschieht über die Zuweisung einer Gruppe, welche selbst auch eine Rolle ist, die mehrere Berechtigungen enthält. Daraus ergeben sich zwei Entitäten, die die Eigenschaften einer Rolle übernehmen (Gruppe und Berechtigung). Jede Rolle wird dabei über einen sprechenden Key identifiziert.

[12] https://authy.com/

Nach dem Login erhält der Benutzer zwei Token, einen AccessToken und einen RefreshToken. Der AccessToken ist kurzlebig, während der RefreshToken länger gültig ist. Nach Ablauf der Gültigkeit des AccessTokens kann der Benutzer sich mithilfe des RefreshTokens einen neuen AccessToken generieren lassen. Bei den Token handelt es sich um JWTs. Im JWT sind Informationen über den Benutzer, seine Session und die Umgebung enthalten. Dazu zählen unter anderem: aktiver Mandant, Rollen und identifizierende Eigenschaften wie die E-Mail. Ein JWT ist im Grunde nichts anderes als ein signiertes, Base64 kodiertes JSON. Die Signatur kann dabei mittels symmetrischer oder asymmetrischer Verfahren erstellt werden. In einem verteilten System eignen sich asymmetrische Verfahren, wie beispielsweise RSA [MKJR16] besser, da so nur der Authentifizierungsdienst den geheimen Schlüssel besitzt und den anderen Services nur der öffentliche Schlüssel bekannt sein muss, um die Signatur zu validieren.

Der AccessToken kann, muss aber keinen Mandantenschlüssel enthalten. Zur Übersicht der Mandanten eines Benutzers existiert ein Portal. Wenn der Benutzer sich über dieses Portal authentifiziert, so erhält er einen AccessToken ohne Mandantenbezug. Dieser beinhaltet lediglich die nötigen Rollen für das Portal. Dazu zählen das Ansehen der Mandanten des Benutzers, das Erstellen eines neuen Mandanten und der Zugriff auf die eigenen Kontodaten.

3.4.3 Rate-Limiting

Um das System verfügbar zu halten und Überlastungen vorzubeugen, wurde eine Limitierung des Anfragendurchsatzes vorgenommen. Dabei werden nach der Überschreitung einer gewissen Anzahl an Anfragen in einem vorgegeben Zeitraums keine Anfragen mehr verarbeitet. Stattdessen wird statisch mit dem dafür vorgesehenen Status-Code 429 von HTTP geantwortet. Dabei wird dem Client bei jeder Anfrage über die Antwortkopfzeilen mitgeteilt, wie viele Anfragen noch getätigt werden können und wann wieder das volle Budget bereitsteht [Pol19].

Vorgenommen wird das Rate-Limiting im Gateway. Dazu werden in Redis[13], einer In-Memory Key-Value Datenbank, die IP und das Limit gespeichert. Bei jeder Anfrage wird der Zähler dekrementiert; Steht dieser auf 0, so wird wie oben beschrieben die Anfrage abgelehnt, ist das Zeitintervall abgelaufen, wird dieser wieder auf das Limit gesetzt. Redis

[13] https://redis.io/

wird benötigt, da auch das Gateway horizontal skaliert werden kann. Ohne externen Datenspeicher müsste es den Zustand im Arbeitsspeicher halten, was Inkonsistenz zwischen den Instanzen zur Folge hätte.

3.5 Kommunikation

Bei der Kommunikation muss zwischen zwei Arten der inter-service Kommunikation unterschieden werden. Auf der einen Seite wäre die asynchrone, auf der anderen die synchrone Kommunikation. In diesem Abschnitt werden die Anwendungsfälle für beide Kommunikationsarten vorgstellt.

3.5.1 Synchrone Kommunikation

Jeder Service stellt, wie in Unterabschnitt 3.2.2 gezeigt, eine API bereit. Diese ist nach dem REST, welches auf dem HTTP Protokoll basiert, modelliert. Im Kontext von REST ist jeder Entität eine URI zuzuweisen. So ergeben sich ressourcenbasierte Schnittstellen. Die Kommunikation läuft nach dem Request-Response Konzept. Dabei sendet der Client eine Anfrage, welche der Service bearbeitet und beantwortet. Der Client wartet dabei immer die Antwort ab, bevor dieser mit weiterer Logik fortfahren kann. Für die Anfragen genutzt werden die HTTP-Verben (*GET*, *POST*, *PUT*, *DELETE*, *HEAD*, *OPTIONS*), während der Server mit einem HTTP Statuscode und optional auch mit einem Payload antworten kann. Ein wichtiges Merkmal der HTTP-Methoden ist das der Idempotenz. So führt das mehrfache Absenden derselben Anfrage zu lediglich einer Änderung der Daten. Davon ausgenommen ist die Methode *POST*. Für diese müssen, wenn gewünscht, eigene Mechanismen in der Geschäftslogik implementiert werden, um die Operation idempotent zu machen [Tak17, S. 48-49].

Dieser Aufbau der Schnittellen entspricht dem zweiten Level des Richardson Maturity Models[14]. Gegen das höhere und letzte Level 3, besser bekannt unter Hypertext As The Engine Of Application State (HATEOAS), wurde sich bewusst entschieden. Als Erweiterung zu Level 2 beschreiben Ressourcen die Verbindung zu anderen Ressourcen oder sich selbst. Dies geschieht über direkte Verlinkungen zu anderen Ressourcen in den Antworten des Servers. So ist die API ohne weitere Kenntnisse navigierbar. Da für dieses System keine externen Clients geplant sind und selbst entwickelte Anwendungen

[14] https://www.crummy.com/writing/speaking/2008-QCon/act3.html

die Struktur der API kennen, ist der Verwaltungsaufwand für die Referenzen eingespart worden.

Die synchrone Kommunikation wird im System für die Client-Server Kommunikation und inter-service Kommunikation genutzt. Die synchrone inter-service Kommunikation dient zum Erhalt wesentlicher Daten aus anderen Services, ohne die die aktuelle Operation nicht weiter ausgeführt werden kann.

3.5.2 Asynchrone Kommunikation

Im Gegensatz zur synchronen Kommunikation wartet der Client bei der asynchronen Kommunikation nicht auf eine Antwort des Servers. Das ist gerade für Abläufe interessant, die keine direkte Antwort benötigen oder sich über mehrere Services erstrecken. In diesem Fall publiziert ein Service ein Event, welches alle betroffenen Services im Anschluss verarbeiten können. Durch die Natur der Asynchronität und Dezentralisierung der Daten geht hier die Transaktionssicherheit von ACID verloren. Stattdessen ist die Datenkonsistenz möglicherweise zeitlich verzögert. Diese sogenannte eventual consistency ist aber in einem Großteil der Fälle ausreichend und sollte auch in diesem System keine Nachteile mit sich bringen [Gha18, S. 347-348].

Benutzt wird die asynchrone Kommunikation im System, um Events wie das Erstellen oder Löschen eines Mandanten oder das Versenden von E-Mails zu propagieren (siehe **QA/03**).

In der asynchronen Kommunikation existieren zwei primäre Stile, wie Nachrichten versendet werden: *single receiver* und *multiple receivers*. Der Unterschied leitet sich bereits aus dem Namen ab. Im Vergleich zum single-receiver Stil, bei dem eine publizierte Nachricht für genau einen Empfänger bestimmt ist, können beim multiple receivers Stil mehrere Empfänger unabhängig voneinander die gleiche Nachricht erhalten. Für den hier angedachten Einsatzzweck der Events eignet sich der single-receiver Stil wenig, da die ausgesendeten Events in den meisten Fällen mehrere Services interessieren. Die Verteilung der Nachrichten erfolgt über sogenannte *Topics*. Services, die ein Event auslösen, publizieren eine Nachricht an ein bestimmtes Topic, welches dann von interessierten Services abonniert werden kann. Ein Topic kann optional in mehrere *Partitionen* aufgeteilt werden. Partitionen dienen der Parallelisierung des Nachrichtenkonsums, auch über mehrere Nodes hinweg. Das Weiterreichen der Nachrichten an die betroffenen Services erfolgt über den *Broker*. Ein Broker ist eine Komponente eines verteilten Systems ohne Businesslogik.

Ihr einziger Zweck ist die Verbreitung von Nachrichten an die betroffenen Empfänger. Als Broker wird Kafka[15] eingesetzt, einer der am meisten benutzten Broker im Bereich der Microservices. Kafka ist ein verteiltes publish/subscribe (pub-sub) System. Durch das grundlegende Design des Systems ist dieses sehr gut skalierbar, wenn es als Cluster betrieben wird. Im Cluster werden die Daten zwischen den einzelnen Knoten repliziert. Zur Koordination verwendet Kafka ZooKeeper[16] [Ind18, S. 76 f.].

3.5.3 Service Discovery

Im Kontext Microservices fällt schnell das Stichwort Discovery. Damit die Services miteinander kommunizieren können, müssen diese erst wissen, wo sich das jeweilige Ziel befindet. Da sich IP-Adressen oder Ähnliches regelmäßig ändern können, ist es wenig ratsam diese statisch im Code oder per dynamischer Konfiguration zur Laufzeit einzubinden. Jede Änderung der IP eines Services hätte demnach Refactorings am Code oder Änderungen an der Konfiguration zur Folge. Hier kommt die Discovery ins Spiel, bei der sich jeder am System beteiligte Service zum Start anmeldet. Dabei gibt er seinen Namen und aktuelle IP Adresse mit. Zugleich muss er auch einen Endpunkt für einen Healthcheck angeben. Dieser Endpunkt wird von der Discovery regelmäßig aufgerufen um zu determinieren, ob der Service verfügbar ist. Antwortet dieser gar nicht oder nicht innerhalb einer festgelegten Zeitspanne, so wird die IP des Services gesperrt. Sollte es mehrere Services mit demselben Namen geben, so wird bei der Abfrage einer IP zufällig eine der verfügbaren Adressen gewählt und zurückgegeben [Gha18, S 346].

Da für diesen Anwendungsfall die Vorteile einer Discovery, wie beispielsweise das regionsbasierte Load-Balancing, nicht genutzt werden, das System aus nur vier Services besteht und auf einem Kubernetes-Cluster betrieben werden soll, wurde in diesem Szenario auf eine eigenständige Discovery verzichtet. Kubernetes bietet intern einen eigenen DNS an, mithilfe dessen die Services einfach kontaktiert werden können. Gleichzeitig ändern sich die Adressen der Services auch zwischen Deployments nicht, sofern sich der Name des Service nicht verändern sollte. Jeder Service im Cluster ist unter Benutzung des folgenden Patterns erreichbar:

$$http://<service-name>.<namespace>.svc.cluster.local:<port>$$

[15] https://kafka.apache.org
[16] https://zookeeper.apache.org/

Dies spart eine weitere Komponente im System und minimiert damit die Komplexität [Kub21].

3.6 Multi-Tenancy

Multi-Tenancy ist ein Konzept, das gerade in modernen Software as a Service (SaaS)-Systemen Anwendung findet. SaaS zeichnet sich dadurch aus, dass Kunden keine Lizenzen für Software kaufen, sondern diese mit regelmäßigen Raten mieten. So müssen sich Kunden nicht mehr selbst um die Verwaltung und Einrichtung kümmern. Dafür entsteht jedoch eine direkte Bindung an das System des Betreibers (vendor-lock-in). Die Kunden benutzen alle die gleiche Version der Software, so können Fehlerbehebungen schneller verbreitet werden. Zudem kann so Geld gespart werden, da sich die Kunden die zur Verfügung stehenden Ressourcen des Servers teilen. Die Teilung liefert Effizienz, da ein einzelner Mandant seine Ressourcen selten ausschöpft. [Red20].

Im Bereich der Multi-Tenancy gibt es mehrere Strategien zur Trennung von mandantenspezifischen Daten. Dabei weist jede Strategie eine andere Stärke der Trennung auf. In der folgenden Abbildung 3.10 lassen sich diese erkennen.

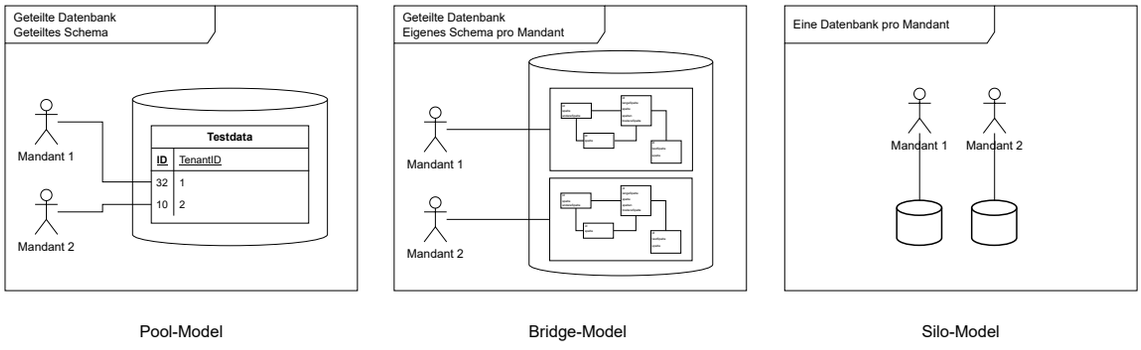


Abbildung 3.10: Strategien zur Trennung von mandantenbezogenen Daten

Die schwächste Trennung besitzt das Pool-Model. Hier werden die Daten der Mandanten auf Spaltenebene getrennt. Dafür erhalten die Einträge in der Datenbank einen Fremdschlüssel, der sich auf den dazugehörigen Mandanten bezieht. So benutzen auch alle Mandanten die gleiche Datenbank und das gleiche Schema. Die stärkste Trennung hingegen weist das Silo-Model auf. Bei diesem besitzt jeder Mandant eine eigene Datenbankinstanz mit eigenem Schema. So können die Daten auch physisch getrennt werden. Jedoch bringt dies auch Nachteile, da für jeden neuen Mandanten eine neue Datenbank automatisch

provisioniert werden muss. Zudem müssen die Zugangsdaten zu dieser Datenbank global gespeichert werden und die betroffenen Services dynamisch zur Laufzeit bestimmen, zu welcher Datenbank eine Verbindung hergestellt werden muss [TG16].

Die Entscheidung fiel letztendlich auf das Bridge-Model. Dabei benutzen alle Mandanten die gleiche Datenbank, jedoch hat jeder Mandant sein eigenes Schema. Hier ist das Schema im Kontext PostgreSQL[17] gemeint. Eine Datenbank kann in mehrere Schemata aufgeteilt werden. Die Tabellenstruktur bleibt dabei für jeden Mandanten identisch. Es muss nur eine Datenbankverbindung konfiguriert werden, wobei das aktuelle Schema dynamisch aufgelöst werden kann. Zudem bietet es die größte Flexibilität und gleichzeitig auch eine geringere Komplexität. Sollte später auf das Silo-Modell umgestellt werden, so ist dies nur noch eine Frage der Konfiguration, das Schema bleibt bestehen.

Um das richtige Schema zur Laufzeit auflösen zu können, muss der Mandant bekannt sein. Da die Verbindung zwischen den Services und der Weboberfläche jedoch zustandslos ist, muss bei jeder Anfrage eine identifizierbare Eigenschaft mitgegeben werden, mithilfe derer der aktive Mandant ermittelt werden kann. Im AccessToken, der ohnehin bei den meisten Anfragen übertragen wird, ist die ID des Mandanten hinterlegt. Daher eignet sich dieser gut, um einen Großteil aller Anfragen abdecken zu können. Gerade jene, die mit mandantenspezifischen Daten zu tun haben, sind so gleichzeitig auch abgesichert, da mit dem AccessToken eine Authentifizierung und Autorisierung einhergeht. In den anderen Fällen, in denen ein Mandant definiert werden muss, geschieht dies über einen Header, welcher die ID des Mandanten beinhaltet.

[17] https://www.postgresql.org/

4 Realisierung

Nach der Konzeption des Systems wird in diesem Kapitel die Realisierung erläutert. Hierfür werden die wichtigsten Aspekte der Entwicklung beleuchtet und auf Hindernisse oder Probleme während dieser eingegangen. Entscheidende Codeausschnitte sind im Anhang zu finden und werden im Text referenziert.

4.1 Frameworks

Einleitend in die Realisierung werden folgend die verwendeten Frameworks vorgestellt. Dabei wird auch darauf eingegangen, wieso diese gewählt wurden und welche Auswirkungen dies auf die Entwicklung hatte.

4.1.1 Frontend

Das Frontend ist eine grafische Oberfläche, mittels derer die Benutzer mit dem Backend interagieren können. Es ist zur Zeit der aktuell einzige Client des Systems (siehe **RB/11**).

Zum Einsatz kommt hier ein JavaScript Framework namens NextJS[1], welches auf React[2] basiert. React gehört zu den beliebtesten Web Frameworks in 2021 [Sta21a]. Das Framework wurde von Facebook im Jahr 2013 veröffentlicht. Gearbeitet wird mit Komponenten. Diese werden wiederum von anderen Komponenten benutzt. Dabei hält jede Komponente einen eigenen Zustand. In untergeordnete Komponenten können von der Elternkomponente Zustandsvariablen übergeben werden. Diese können zum Beispiel benutzt werden, um Dinge anzuzeigen, zu berechnen oder den Zustand der übergeordneten Komponente zu verändern. React arbeitet auf einem virtuellen DOM, um Seiteninhalte

[1] https://nextjs.org/
[2] https://reactjs.org/

reaktiv aktualisieren zu können. So werden bei einem Update des Zustands der Anwendung betroffene Komponenten, und auch nur diese, neu gezeichnet. Dazu werden die Änderung im virtuellen DOM analysiert und auf den tatsächlichen DOM der Applikation angewendet. Entwickelt wird mittels JSX, einer Transformationsschicht, die lesbares, HTML-ähnliches XML in den von React aufgebauten, virtuellen DOM umwandelt. Eine mit React entwickelte Anwendung wird Single Page Application (SPA) genannt. Der Name leitet sich aus der Natur der Funktionsweise ab. Im Vergleich zu anderen Konzepten, wie Model-View-Controller (MVC), wird die Seite erst im Browser des Besuchers aufgebaut. Die Antwort des Servers enthält dabei minimales HTML, welches dynamisch per JavaScript um die gewünschten Inhalte ergänzt wird [Gac15].

NextJS basiert, wie bereits erwähnt, auf React. Es bietet eine übersichtliche Ordnerstruktur, Routing und Server Side Rendering (SSR). SSR löst eines der Probleme von SPAs, Search Engine Optimization (SEO) und das Holen von Daten von externen Diensten, zum Beispiel einer API. Ohne SSR muss, bis angeforderte Daten von einem anderen Server empfangen worden sind, ein Ladezustand implementiert werden. Stattdessen wird unter Benutzung von SSR die Seite auf einem NodeJS[3] Server vorgeladen und als vollständiges HTML zurückgegeben. Gleichzeitig wird der Zustand der Komponenten vom Server beibehalten und kann im Browser weiterverwendet und verändert werden. Ähnliches gilt für die SEO. Zur Bestimmung eines Ratings und der Inhalte, wie Keywords, wird nur die Antwort vom Server analysiert und nicht das ausgeführte JavaScript Bundle einer SPA [ILKL20].

Um Daten im Frontend zu persistieren, können vom Browser bereitgestellte Methodiken benutzt werden. Dazu zählen primär Cookies und der *localStorage*[4]. Cookies sind Textdokumente, die mit einer bestimmten Gültigkeitsdauer auf dem Gerät des Besuchers gespeichert sind. Der *localStorage* bietet ebenfalls eine API, um Daten im Browser zu speichern. In dem zum Prototypen gehörenden Frontend werden die meisten Daten in Cookies gespeichert. Dazu gehören zum Beispiel der AccessToken und RefreshToken.

Wie oben bereits erwähnt, können Variablen nur von oben nach unten propagiert werden. Sollten Daten einer Komponente in tief verschachtelten Komponenten benutzt werden müssen, so spricht man von *prop drilling*, also das ständige Weiterreichen einer Variable an untergeordnete Komponenten über mehrere Ebenen hinweg. Dabei handelt es sich jedoch um ein Anti-Pattern, was es zu vermeiden gilt. React bietet für solche Fälle

[3] https://nodejs.org/
[4] https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage

eine Context-API[5]. Dabei wird ein Zustand erstellt. Dieser Zustand stellt eine *Provider*-Komponente bereit, innerhalb dieser andere Komponenten auf die Daten zugreifen können. Kontexte sollten jedoch nur dann genutzt werden, wenn diese wirklich gebraucht werden, da bei einer Veränderung des Zustands alle Komponenten in den Providern neu gezeichnet werden, was durchaus zu Seiteneffekten führen kann [Dod18]. Im Prototypen werden Kontexte benutzt für Authentifizierungsdaten, Mandanteneinstellungen und Benachrichtigungen (zum Beispiel Erfolgsmeldungen beim Speichern).

Der primäre Grund für die Entscheidung zur Nutzung von NextJS ist es, die Kombination der Vorteile von React mit denen des SSR zu verbinden. Das Zustandshandling von React vereinfacht dynamische Formulare und bietet die Möglichkeit schnell und simpel eine gute, flüssige und benutzerfreundliche Anwendung zu entwickeln. Gleichzeitig können über den Server Daten von der API vorgeladen werden um Sprünge im Layout zu vermeiden.

4.1.2 Services

Ein Vorteil der Microservice-Architektur ist die Unabhängigkeit von Programmiersprachen und Frameworks. Es steht also ein breites Spektrum an potentiellen Technologien bereit, die für die Entwicklung benutzt werden können.

Die Services sowie das Gateway wurden alle in Java mithilfe des Frameworks Spring Boot[6] implementiert. Das hat den Hintergrund einer vereinfachten Wartbarkeit und der bereits existierenden Vorerfahrung mit den verwendeten Frameworks und Sprachen. Zudem bietet Spring Boot ein breites Ökosystem von Projekten[7], die gerade für Cloud-Infrastrukturen die Implementation vereinfachen.

Der Aufbau von Spring Boot ist modular was eine flexible Entwicklung ermöglicht. [Gol19, S. 11]. Die Verwaltung der Module übernimmt der Spring Container. Er sorgt für die korrekte Abwicklung von Prozessen und instanziert und pflegt die Beans. Wolfgang Golubski beschreibt ein Bean wie folgt:

"Beans sind (Java-)Objekte, die vom Spring Container verwaltet werden, inklusive ihrer Instanziierung, Überwachung und Löschung. Die Beans werden durch Meta-Informationen

[5]https://reactjs.org/docs/context.html#gatsby-focus-wrapper
[6]https://spring.io/projects/spring-boot
[7]https://spring.io/projects

im Programm ausgewiesen (per Annotation oder XML-Deklaration). Sie durchlaufen einen Lebenszyklus, angefangen von der Instanziierung bis zu ihrem Ende." [Gol19, S. 22]

Die Definition eines Beans geschieht über Annotationen, welche Klassen oder Methoden markieren und so für die Aufnahme in den Container von Spring Boot sorgen. Diese Funktionalität basiert auf dem Konzept Inversion of Control (IoC), welches mittels Dependency Injection (DI) realisiert ist. Der Kern von IoC ist die Abgabe Verwaltung und Instanzierung von Objekten an eine Dritte Partei, in diesem Fall das Framework. Mittels DI können die markierten Komponenten in anderen Komponenten automatisiert injiziert werden (zum Beispiel über den Konstruktor), was eine lose Kopplung der Komponenten ermöglicht [Gol19, S. 64 f.].

Die wichtigsten Typen von Beans sind Controller und Service. Controller definieren Endpunkte und erschaffen so die benötigten Zugriffspunkte zum Ausführen der Geschäftslogik [Gol19, S.47], welche im Service umgesetzt ist. Im Service findet beispielsweise die Kommunikation mit der Datenbank statt. Es können auch mehrere Services zusammenarbeiten [Gol19, S.4].

Die einzige Ausnahme, die nicht mit Spring Boot entwickelt worden ist, ist der Mail-Service, welcher in NodeJS umgesetzt wurde. Das liegt an der einfachen Nutzung von asynchronen Methoden. Die Verarbeitung von Events und das Versenden der E-Mails erfolgt asynchron, sodass viele Events trotzdem performant behandelt werden können. Zudem wird für das Gestalten der E-Mails Mailjet Markup Language (MJML)[8] genutzt. Das Versenden von HTML-Emails, die in allen Clients so aussehen, wie sie aussehen sollen ist leider aufwendig, da es keinen Standard gibt. So nutzen unterschiedliche Clients unterschiedliche Render-Engines zur Darstellung des Inhalts. MJML versucht dieses Problem zu lösen, indem es eine einfache, XML-ähnliche Syntax bietet, die in sehr einfaches HTML umgewandelt werden kann. Diese Umwandlung funktioniert entweder über eine kostenlose API oder über ein offizielles JavaScript-Modul, bereitgestellt von den Entwicklern von MJML. Um weitere Fremdsysteme zu vermeiden, wurde daher die offizielle Library genutzt, was die Nutzung von JavaScript voraussetzt.

[8] https://mjml.io/

4.2 Konfiguration

In modernen Web-Applikationen reicht eine statische Konfiguration meist nicht aus. Zum einen muss bei der Änderung der Konfiguration eine neue Version des Systems bereitgestellt werden, (Beispiel: bei der Änderung der Datenbankverbindung), zum anderen existieren für Systeme meist mehrere Umgebungen. Die Applikation im Systembetrieb greift beispielsweise auf eine andere Datenbank zu als die Testumgebung. Um Software dynamisch zur Laufzeit konfigurieren zu können, gibt es verschiedene Methoden. Entschieden wurde sich für Umgebungsvariablen (*Environment Variables*). Umgebungsvariablen werden durch einen Schlüssel identifiziert und halten einen bestimmten Wert. Beim Start der Applikation werden besagte Variablen über ihren Schlüssel eingelesen. Je nachdem, wie die Umgebungsvariablen in dem Framework oder der Programmiersprache behandelt werden, können Änderung an der Umgebung direkt wiedergespiegelt werden. In jedem Fall reicht ein Neustart, um die neue Konfiguration zu übernehmen [Put18].

4.3 Multi-Tenancy

In SaaS-Applikationen ist die Mandantenfähigkeit wichtig, um Kosten zu sparen. So ist auch in diesem System eine Anforderung, dass mehrere Mandanten auf dem gleichen System arbeiten können (siehe **RB/04**). Das Konzept ist in Abschnitt 3.6 dokumentiert. Um regelmäßige Abfragen beim User-Service zu vermeiden, werden die Mandantendaten, die für die einzelnen Services relevant sind, dezentral gehalten.

4.3.1 Migrationen

Beim Start einer jeden Anwendung, die eine Datenbankverbindung verwendet, wird zunächst das globale und gemeinsame Schema migriert und auf den aktuellen Stand gebracht. Danach werden die mandantenbezogenen Tabellen pro Mandant in einem eigenen Schema nacheinander migriert. So werden mögliche Datenbankänderungen mit einer neuen Version der Software direkt mit ausgeliefert. Die SQL-Dateien, die die Änderungen am Schema definieren, sind im Programm als Ressource enthalten.

Zur Verwaltung kommt Flyway[9] zum Einsatz. Es arbeitet, wie die meisten Libraries zum Migrieren von Datenbanken auch, mit einer eigenen Tabelle zum Zweck der Versionsver-

[9] https://flywaydb.org/

waltung. In dieser wird hinterlegt, wann welche Version des Datenbankschemas migriert wurde und ob der Versuch erfolgreich gewesen ist. Über eine Konfiguration wird der Library mitgeteilt, welcher Ordner die Migrationen beinhaltet und wie diese auf welche Datenquelle ausgeführt werden soll. Zudem wird das Schema angegeben, für welches die Skripte ausgeführt werden. Zusätzlich zum globalen Schema wird in einer weiteren Methode über alle Mandanten in der Datenbank iteriert und für jeden die Migration des mandantenspezifischen Schemata durchgeführt (siehe Anhang A.10).

4.3.2 Zuordnung

Die Zuordnung zu einem Mandanten findet über den AccessToken oder *Tenant*-Header statt. Übertragen wird die ID des Mandanten. Um hier regelmäßige Abfragen beim User-Service zu vermeiden, werden die Mandantendaten, die für die einzelnen Services relevant sind, dezentral gehalten. Dazu wird bei der Erstellung des Mandanten ein Event ausgelöst. Dieses beinhaltet die Informationen des angelegten Mandanten. Für die Services ist dabei nur die ID und der Name des Schemas von Relevanz. Andere Werte werden von den Services intern initialisiert und gepflegt. Wird die ID des Mandanten bei einer Anfrage übertragen, wird diese in der Tabelle der Mandanten gesucht. Dies geschieht in demselben Filter, in dem auch die Authentifizierung geregelt wird, da so das JWT nur einmal destrukturiert werden muss. Wird ein AccessToken übergeben, wird versucht, die ID des Mandanten aus diesem auszulesen. Andernfalls wird geprüft, ob ein Header mit der ID gesetzt wurde (siehe Anhang A.16). Ist eine ID in einer Anfrage vorhanden, so wird eine, nur im Thread des Requests verfügbare, Variable auf den Namen des Schemas gesetzt. Diese ThreadLocal-Variable dient als Kontext für alle künftigen Aktionen in diesem Request (siehe Anhang A.13). Das hat den Hintergrund, dass Spring Boot und der unterliegende Tomcat-Server pro Anfrage einen Thread verwendet[10]. Nach der Initialisierung kann zu jedem Zeitpunkt, während der Request noch nicht beendet wurde, der Name des Schemas des Mandanten nachvollzogen werden. Verwendet wird dies beim Aufbau der Datenbankverbindung. Hierzu wird die Konfiguration von Hibernate[11] angepasst. Hibernate ist das von Spring Data genutzte Object-Relational Mapping (ORM). Ein ORM bietet eine Abstraktionsschicht zwischen der objektorientierten Programmierung und der Datenbank. Dadurch müssen für die Interaktion mit der Datenbank keine SQL-Abfragen mehr geschrieben werden. Diese werden durch die Nutzung bestimmter

[10] https://www.baeldung.com/java-threadlocal
[11] https://hibernate.org/orm/

Interfaces und Klassen automatisch generiert. Natürlich bietet ein ORM auch die Möglichkeit, eigene Abfragen zu definieren und auszuführen [Mag20]. Vieles der Komplexität in Bezug auf die mandantengetrennte Datenhaltung wird von Hibernate abstrahiert. So bietet Hibernate Funktionalitäten, um ohne viel Extraaufwand mandantenspezifische Datenbankverbindungen aufzubauen. Dafür müssen in der Konfiguration von Hibernate (siehe Anhang A.15) die Implementation von zwei Interfaces bereitgestellt werden. Dazu zählen CurrentTenantIdentifierResolver und MultiTenantConnectionProvider. Der CurrentTenantIdentifierResolver leitet aus dem bereits beschrieben Kontext das Schema des Mandanten ab. Ist kein Schema definiert, so wird das *default* Schema der Datenbank gewählt, in welchem globale Tabellen der Instanz liegen. Das ausgewertete Schema wird dann im MultiTenantConnectionProvider benutzt, um eine Datenbankverbindung herzustellen. Genutzt wird hier die gleiche Verbindung, welche auch automatisch von Hibernate zum Systemstart aufgebaut wird. Einzig das aktive Schema wird verändert [bae21]. Die Umsetzung der beschriebenen Funktionalität ist in Anhang A.11 & A.12 zu finden.

4.4 Filterung

Benutzer sollen Auflistungen filtern können (siehe **RB/08**). Dazu zählen zum Beispiel die Übersichten der Pools oder Kontakte. Um an dieser Stelle möglichst flexibel zu bleiben, wurde versucht, auf strikt festgelegte *GET*-Parameter zu verzichten. Die Anforderung an den Filtermechanismus ist es also, möglichst wenig Abweichung vom eigentlichen Datenmodell der Ressource, Flexibilität in den Operationen und dem Aufbauen von komplexen Abfragen sowie eine möglichst wartungsfreundliche Umsetzung im Programmcode zu bieten.

Zum Einsatz kommt daher RSQL, eine auf Feed Item Query Language (FIQL) basierende Abfragesprache für REST-APIs. Übertragen wird der Filter über einen einzigen *GET*-Parameter. Die Abfrage wird zuerst in einen Syntax-Tree und dann in SQL umgewandelt. Durch diese Umwandlung werden zum Beispiel SQL-Injection-Angriffe verhindert und die Performanz der Abfragegenerierung erhöht [Not07]. Für Java und Java Persistance API (JPA) ist die Einbindung in Spring Boot sehr einfach. Hierfür wird ein eigenes Repository implementiert, welches das Standardrepository von JPA beerbt. Auf diesem wird eine neue Methode eingeführt, welche den erhaltenen Filter entgegennimmt, umwandelt und

gegen die Datenbank ausführt. Die Umwandlung übernimmt die Bibliothek *rsql-parser*[12], welche gut getestet ist. Die Umsetzung ist in Anhang A.7 & A.8 zu finden. Wichtig ist hier die @NoRepositoryBean Annotation auf dem Repository. Diese signalisiert Spring Boot, dass es sich lediglich um eine Erweiterung eines bestehenden Repositories handelt und keine Instanz als Bean notwendig ist [Gol19, S. 121]. Die Implementation muss dann lediglich noch von Spring Boot registriert werden, und damit ist die Einbindung abgeschlossen (siehe Anhang A.9).

4.5 Dateioperationen

Im Folgenden wird die Umsetzung des in Abschnitt 3.3 vorgestellten Speicherkonzeptes beschrieben, die Besonderheiten von Downloads und Uploads beleuchtet und Probleme sowie deren Lösungen erläutert.

4.5.1 Download

Mit S3 lassen sich problemlos Dateien hoch- und herunterladen. Jedoch handelt es sich um nicht frei zugängliche Dateien. Ein Direktzugriff via Link zum Download darf nicht möglich sein. So fungiert der Pool-Service als Proxy, der eine Anfrage zum Download entgegennimmt und prüft, ob der Anfragende berechtigt ist, die Datei herunterzuladen. Diese Prüfung geschieht über den als GET-Parameter übergebenen AccessToken. Wird der Zugriff gewährt, so wird die Datei als Stream von S3 geladen und auch als Stream für den Client bereitgestellt. Um die Speicherauslastung möglichst gering zu halten, wird der Stream in Segmenten ausgelesen und übertragen. Der Code ist in Anhang A.2 dokumentiert.

Eine weitere Anforderung ist die Möglichkeit des Downloads eines ZIP-Archivs aller Dateien eines Pools, auf die zugegriffen werden darf (siehe **RB/09**). Die Logik dafür ist die gleiche, als würde nur eine Datei heruntergeladen werden. Einziger Unterschied ist das sequentielle Schreiben der einzelnen Streams in einen ZIP-Stream.

[12] https://github.com/jirutka/rsql-parser

4.5.2 Upload

Nach den Anforderungen sollen gescheiterte Uploads wiederaufgenommen werden (siehe **RB/06**). So können auch Benutzer mit schlechterer oder instabiler Internetverbindung größere Dateien hochladen. Normalerweise wird eine Datei bei einem Dateiupload als ein Ganzes an den Server übertragen und dort verarbeitet. In diesem Fall bietet sich das nicht an, da beispielsweise der Browser über JavaScript als Client über keine Mechanismen verfügt, um diesen Upload nach Verbindungsabbruch fortführen zu können. Als Lösung hierfür wird spezifiziert, dass der Client Dateien in Chunks an den Poolservice überträgt. Ein Chunk ist eine Teilmenge der Daten einer Datei. Eine Datei mit der Größe von 50MB kann beispielsweise in zehn Chunks à 5MB oder aber auch 200 Chunks à 250KB unterteilt werden. Der Upload kann zudem vom Benutzer jederzeit pausiert oder abgebrochen werden. Die folgende Abbildung 4.1 zeigt die möglichen Zustände eines Uploads und deren Übergänge.

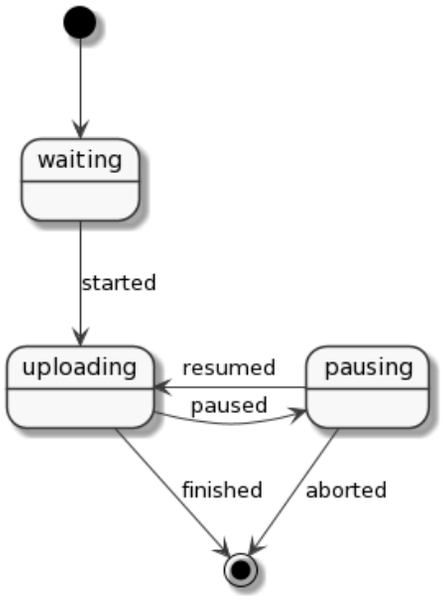


Abbildung 4.1: Zustandsdiagramm: Zustandsübergänge des Upload Prozesses

Zur Umsetzung wurde das Protokoll tus[13] implementiert. Es basiert auf HTTP und ist erweiterbar. Zum eigentlichen Protokoll gehören lediglich drei Endpunkte. Dabei verhält sich das Protokoll wie bereits beschrieben. Es lädt die Datei in einzelnen Chunks sequentiell hoch. Zu Beginn eines Uploads wird am Server der Stand der Datei abgefragt.

[13] https://tus.io/

Wenn die Datei bereits existiert und noch nicht fertig hochgeladen wurde, werden die bereits hochgeladenen Bytes in einem Header zurückgegeben, woraufhin der Client mit dem Upload ab diesem Offset fortfährt. Wie bereits erwähnt, ist das Protokoll erweiterbar. Implementiert wurden die Erweiterungen *Creation*, *Termination* und *Expiration*. Mittels *Creation* wird auf dem Server eine Ressource erstellt. Die dadurch erhaltene ID wird für zukünftige Anfragen genutzt, beispielsweise den eigentlichen Upload oder die Abfrage des Fortschritts. *Termination* bietet die Möglichkeit, einen Upload abzubrechen oder eine bereits hochgeladene Datei zu löschen. Der Ablauf ist nochmals in Abbildung 4.2 dokumentiert. Um Speicherplatz zu sparen, wurde auch die *Expiration*-Erweiterung umgesetzt. Diese bewirkt, dass ein begonnener Upload nach Ablauf einer gewissen Zeitspanne nicht mehr fortgeführt werden kann und die bereits hochgeladenen Teile gelöscht werden müssen [GZK+16]. Diese Zeitspanne ist auf 5 Tage gesetzt. Dabei läuft ein Job jeden Tag um Mitternacht und räumt den temporären Ordner auf (siehe Anhang A.4).

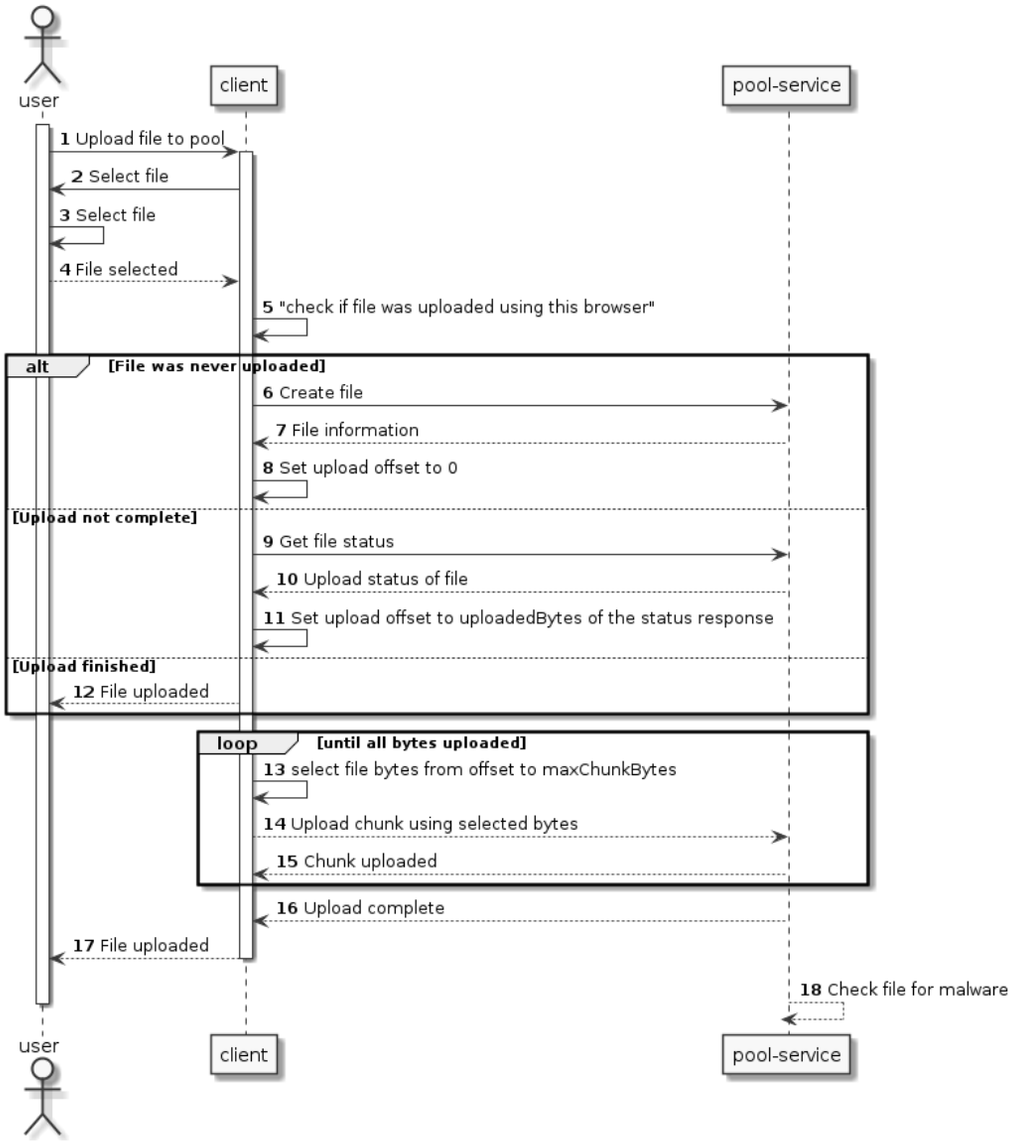


Abbildung 4.2: Sequenzdiagramm: Ablauf eines Uploads durch das tus Protokoll

In der Theorie könnte hier sehr passend das Multipart-Upload Feature der S3 API verwendet werden. Dabei werden nur Teile einer Datei hochgeladen und diese dann nach Upload des letzten Teils in S3 zu einem *Object* zusammengesetzt. Es wäre somit keine Mehrarbeit nötig. Eingehende Daten würden direkt an S3 weitergereicht und die Logik des temporären Speicherns und der Adaption würde delegiert. Da aber die Daten **vor** der Übertragung an externe Dienstleister verschlüsselt werden sollen und die aktuelle Version der Java SDK clientseitige Verschlüsslung von Dateien bei Multipart-Uploads

nicht unterstützt, wurde ein Caching der Chunks im Pool-Service implementiert. Diese werden temporär auf einem zentralen, von allen Instanzen genutztem Speicherort hinterlegt. Nachdem alle Bytes übertragen worden sind, wird diese Datei verschlüsselt und an S3 übertragen. Dadurch werden die Kundendaten zu jedem Zeitpunkt vor der Einsicht von Dritten geschützt.

Nach dem Hochladen erfolgt dann die in Unterabschnitt 3.4.1 angesprochene Prüfung auf Gefahren. Hierzu wird die Datei über TCP an ClamAV gesendet, dort verarbeitet und das Resultat analysiert. Werden potentiell schadhafte Inhalte erkannt, so evaluiert die Funktion mit *false* und die Datei wird aus der Datenbank und dem Dateispeicher gelöscht (siehe Anhang A.5). Zusätzlich dazu wird eine E-Mail an die Person versendet, die die Datei hochgeladen hat. Diese dient lediglich als Information, da die Löschung zeitversetzt eintreten kann und so keine Dateien ohne Kommunikation aus dem System verschwinden. Die Prüfung erfolgt periodisch mit einem Intervall von 30 Sekunden. Dabei werden alle noch nicht geprüften Dateien sequentiell überprüft. Der Job, der die Prüfung und den Mailversand bei positivem Befund verwaltet, ist in Anhang A.6 dokumentiert. Der Versand an den Hochladenden fehlt, wenn es sich um einen User und nicht um einen Kontakt handelt, da hier die Kommunikation zwischen User- und Pool-Service noch nicht definiert wurde. Getestet werden kann das ganze über die bekannte Eicar Testdatei[14]. Dabei handelt es sich um ein harmloses DOS Programm, welches in ASCII Zeichen versteckt ist.

4.6 Silent Token Refresh

Da der vom User-Service ausgestellte AccessToken kurzlebig ist, muss dieser über den RefreshToken regelmäßig erneuert werden. Dazu wurde ein Silent Token Refresh implementiert. Im Grunde ist dieser sehr simpel. Der Client agiert so, als wäre der AccessToken unlimitiert gültig. Wenn der Token abgelaufen ist, antwortet die API mit einem bestimmten HTTP Status Code: *401 Unauthorized*. Dieser bestimmte Status Code wird nur zurückgegeben, wenn etwas mit dem übertragenen AccessToken nicht stimmt, zum Beispiel da dieser abgelaufen ist. Die im Client benutzte Library Axios[15] bietet die Möglichkeit, mithilfe eines *Interceptors* diesen Status Code abzufangen. Dabei wird der Request, der die Antwort ausgelöst hat, wieder zur Verfügung gestellt. An dieser Stelle versucht der

[14] https://www.eicar.org/?page$_id = 3950$
[15] https://axios-http.com/

Client nun, mithilfe des RefreshTokens einen neuen AccessToken vom Server zu holen. Vorher wird der Interceptor wieder entfernt, um Endlosschleifen durch wiederholte 401 Status Codes zu verhindern. Ist die Abfrage erfolgreich, wird der *Authorization*-Header des originalen Requests überschrieben und dieser neu abgesendet [bez21]. Die Umsetzung des Interceptors ist in Anhang A.14 zu finden.

4.7 OpenAPI & Swagger

OpenAPI ist eine Spezifikation zur Dokumentation von REST-APIs. Im Grunde ist die Spezifikation die Repräsentation der API Endpunkte und DTOs als JSON- oder YAML-Schema. Dieses kann dann in Verbindung mit Swagger UI genutzt werden, um eine grafische Oberfläche für eine Entwicklerdokumentation zu generieren. Des Weiteren kann die Spezifikation zur Codegenerierung benutzt werden. Dadurch, dass alle Endpunkte mit den möglichen Response Codes und Payloads definiert sind, kann eine ganze SDK aus dem Dokument generiert werden, welcher clientseitig benutzt werden kann, um mit der API zu kommunizieren. Für den Server könnten DTOs oder Stubs generiert werden, die die Implementierung vereinfachen [Ind18, S. 300]. An dieser Stelle existiert ein Henne-Ei-Problem: Code-First oder API-First? Code-First bedeutet, dass die API implementiert wird und aus der Implementierung eine API Dokumentation generiert wird. Bei API-First ist dies genau anders herum: Zuerst wird die API definiert, dann können aus dieser Definition wie oben beschrieben Stubs oder Klassen generiert werden. Beide Herangehensweisen haben Vor- wie Nachteile, doch ein Punkt unterscheidet die beiden Ansätze deutlich. Bei API-First muss, wenn an der API etwas verändert werden muss, erst die Dokumentation geändert werden und im Anschluss müssen mithilfe der neuen Spezifikation die Stubs neu generiert werden. Das schafft Mehrarbeit, statt einfach aus dem Code, in diesem Fall in Java per Annotationen, die Dokumentation generieren zu lassen. So werden auch Änderungen direkt abgebildet.

Die Vorteile der Dokumentation machen sich im Frontend deutlich. Mithilfe des OpenAPI Generators[16] kann eine vollständige SDK generiert werden. Diese kann im Frontend benutzt werden, um Daten von der API abzufragen. Die Generierung macht die Entwicklung sehr flexibel. Änderungen an der API können direkt durch die Codegenerierung übernommen werden. So müssen auch keine geänderten Routen im Code ausfinding gemacht und angepasst werden. Zudem bietet die SDK eine einfache Möglichkeit zur Kon-

[16] https://openapi-generator.tech/

figuration. Hier kann beispielsweise je nach Umgebung die richtige Adresse des Gateways eingetragen werden. Doch stellt sich ein Problem: mehrere Definitionen. Jeder Service stellt eine eigene Definition zur Verfügung. Die Library kann jedoch nur eine Spezifikation zur Zeit verarbeiten. Versucht man mehrere Generierungen hintereinander, so schafft man Probleme, da der zuvor generierte Code wieder verändert wird. Als Lösung kommt eine weitere Library zum Einsatz. Durch openapi-merge-cli[17] können mehrere OpenAPI Definitionen in eine JSON-Datei zusammengefasst werden. Diese kann dann zur Codegenerierung verwendet werden.

4.8 Monitoring

Das Monitoring ist ein essentieller Part von modernen Systemen. Durch die Überwachung der einzelnen Komponenten können Fehlerzustände schnell gefunden und behoben werden. Dazu stellen alle Services bestimmte Schnittstellen und Metriken bereit. Diese Endpunkte nennt man *Actuator*. Über diese Schnittstellen können dann Healthchecks, Informationen oder Metriken wie der aktuelle Speicherverbrauch bezogen werden. Eingesammelt werden die Metriken von Prometheus[18], einem kostenfreien System zur Überwachung von Anwendungen. Dieser ruft in einem vorgegebenen Intervall eingerichtete Endpunkte ab. In Prometheus können dann Alarme festgelegt werden, zum Beispiel für den Fall, dass die Auslastung des Arbeitsspeichers einen bestimmten Schwellenwert übersteigt [Ind18, S. 390 f.].

4.9 Betrieb

Für den reibungslosen Betrieb einer Software muss auf Vieles geachtet werden. Dazu zählt die grundlegende Infrastruktur, aber auch die Laufzeit. Im Folgenden wird der Proxy für die Web-Applikation erläutert und der Betrieb der Services und die dahinterstehende Infrastruktur vorgestellt und erklärt.

[17] https://www.npmjs.com/package/openapi-merge-cli
[18] https://prometheus.io/

4.9.1 Caddy

Laut den Anforderungen soll jeder Mandant über eine eigene Subdomain verfügen und optional eine eigene Domain aufschalten können (siehe **RB/05**). Das scheint auf den ersten Blick keine große Herausforderung zu sein, jedoch muss man einige Faktoren betrachten. So zum Beispiel ist wichtig, dass auch nach den Anforderungen jederzeit und ohne die Arbeit von Entwicklern oder des Anbieters ein neuer Mandant erstellt werden kann (siehe **RB/04**). Gleichzeitig gilt auch, dass die Verbindung zwischen Client und Server über SSL verschlüsselt sein muss (siehe **RB/12**). Die Bereitstellung von Subdomains ist dabei kein Problem. Die Lösung hierfür ist ein standardmäßiges Wildcard-Zertifikat über eine vertraute CA, wie die von Let's Encrypt[19]. So muss bei Erstellung eines neuen Mandanten kein Zertifikat erstellt oder hinterlegt werden. Sollte aber ein Mandant sich dazu entscheiden, eine eigene Domain aufschalten zu wollen, so muss für genau diese Domain ein gültiges Zertifikat generiert werden.

Hier kommt Caddy[20] ins Spiel. Caddy ist ein Webserver, vergleichbar mit dem bekannten Nginx[21], der als Reverse Proxy vor der eigentlichen Webanwendung sitzt, statt diese direkt in das öffentliche Netz zu stellen. Dafür hat Caddy ein außerordentlich hilfreiches Feature: On-Demand-TLS[22]. Mithilfe dieses Features können zur Laufzeit on-the-fly Zertifikate erstellt werden. Dabei ist der initiale Aufruf der Seite selbstverständlich langsam, da hier erst das Zertifikat generiert werden muss, bevor der TLS Handshake stattfinden kann. Der Kunde muss lediglich einen DNS-Eintrag für seine gewünschte Domain oder Subdomain vornehmen und nach der Propagierung der Änderungen über die Nameserver ist die neue Domain eingerichtet. Um das System vor betrügerischen Aktionen oder unnötig erstellten Zertifikaten zu schützen, fragt Caddy beim Aufruf einer Domain, für die es noch kein, oder nur ein fast abgelaufenes Zertifikat existiert, einen konfigurierten Endpunkt ab [Cad21]. Hier wird geprüft, ob es einen aktiven Mandanten gibt, der die angefragte Domain hinterlegt hat und ob dieser Mandant das Addon gebucht hat. Ist dies der Fall, gibt der Endpunkt einen HTTP-Status Code *200* zurück, welcher Caddy signalisiert, dass ein Zertifikat generiert werden soll. Die Konfiguration des Caddyservers ist in Anhang A.20 zu finden.

[19] https://letsencrypt.org/
[20] https://caddyserver.com/v2
[21] https://nginx.org/en/
[22] https://caddyserver.com/docs/automatic-httpson-demand-tls

Zusätzlich dazu kann man Caddy auch in einem Cluster verwalten, welches die Zertifikate automatisch synchronisiert und keine unnötigen Anfragen an die CA von Let's Encrypt schickt [Cad21].

Gleichzeitig erweist sich Caddy auch sehr nützlich für die lokale Entwicklung. So können hier über eine eigene, von einer im lokalen System vertrauten CA, selbstsignierte SSL Zertifikate automatisiert ausgestellt werden, um auch lokal die Funktionalität der Subdomain und Domain Zugriffe zu simulieren. Dafür ist lediglich noch ein Eintrag in die Hosts-Datei des Computers durchzuführen, sodass die DNS Auflösung auf den eigenen Computer zeigt und keine Nameserver kontaktiert werden.

4.9.2 Kubernetes Cluster

Das Cluster, auf welchem die Software betrieben wird, basiert auf der Cloud von Hetzner[23]. Hetzner ist ein deutscher Anbieter mit vergleichsweise günstigen Preisen. Gewählt wurde dieser Anbieter aufgrund der Lage der Datenzentren in Deutschland und Europa (siehe **RB/01**) und des guten Preis-Leistungs-Verhältnisses. Dazu kommt noch die eigene persönliche und positive Erfahrung mit Hetzner aus der Vergangenheit.

Da Hetzner Kubernetes nicht als verwalteten Dienst anbietet, muss das Cluster selbst eingerichtet und verwaltet werden. Das ist für den Prototypen an sich kein Problem, der dadurch entstehende Mehraufwand an Arbeitsstunden für Wartung und Betrieb muss jedoch vor dem Start eines Produktivsystems den höheren initialen Kosten eines verwalteten Kubernetes Dienstes, zum Beispiel von OVH[24], gegengerechnet werden.

Zur Einrichtung und Verwaltung eines Kubernetes Clusters auf der Infrastruktur von Hetzner wurde ein sehr hilfreiches Tool namens hetzner-k3s[25] verwendet. Es bietet die Möglichkeit, mit Hilfe einer einfachen Konfiguration in YAML (siehe Anhang A.19) ein Cluster mit nur einem Kommando zu initialisieren. Dabei ist der Befehl idempotent. Bei erneuter Ausführung werden bereits erstellte Ressourcen nicht erneut erstellt, neue dafür schon. Server, die aus der Konfiguration gelöscht worden sind, müssen jedoch mit der aktuellen Version der Software manuell gelöscht werden. Das Cluster beruht dabei auf k3s[26], einer schlanken und damit ressourcensparenden Kubernetes Distribution. Sie verzichtet

[23] https://www.hetzner.com/cloud
[24] https://www.ovhcloud.com/de/public-cloud/kubernetes/
[25] https://github.com/vitobotta/hetzner-k3s
[26] https://k3s.io/

auf einige Dinge, die im Alltag nicht benötigt werden, aber in k8s[27], der Standarddistribution, enthalten sind. Zudem benutzt es keinen *etcd* zum Speichern der Konfiguration, sondern eine SQLite[28] Datenbank. Dadurch ergibt sich eine einzige Binary, die kleiner als 40 Megabyte ist und nur 512 Megabyte Speicher benötigt.

4.9.3 DevOps

Unter DevOps versteht man einen nicht fest definierten, digitalen Prozess, der die Entwicklung und den Betrieb von Software vereinen soll. Durch diesen Prozess können Aufgaben wie das Paketieren oder Ausliefern von Applikationen schnell und flexibel gemeistert werden (siehe **QA/04**). Automatisierung spielt eine große Rolle, um dies zu erreichen. Gleichzeitig bietet DevOps auch eine weitere Möglichkeit der automatisierten Qualitätskontrolle [EGHS16].

DevOps integriert sich sehr gut mit dem Architekturprinzip der Microservices und deren Ziel, schneller individuell Software auszuliefern. Die zur Verfügung stehenden Tools zur Automatisierung kann man dabei grob in zwei Kategorien aufteilen [Ind18, S. 259 f.]. Diese werden im Folgenden vorgestellt und die Verwendung für die Prototypen erläutert. Alle angesprochenen Pipelines sind mittels der in GitLab[29] integrierten Pipelines umgesetzt. Als Beispiel ist in Anhang A.18 die Konfiguration der Pipelines der Java basierten Services zu finden.

Continuous Integration

Continuous Integration (CI) Werkzeuge helfen bei der schnellen Integration von neuem Programmcode. Wenn benötigt, wird das Projekt kompiliert oder gebaut, um so die Änderungen automatisiert zu testen oder die Codequalität bewerten zu können. So werden nicht nur die geänderten Dateien betrachtet, sondern deren Auswirkungen auf das Gesamtsystem, da alle vorher erfolgreichen Tests auch nach den Änderungen erfolgreich ausfallen müssen. Gleichzeitig können auch so plattformabhängige Fehler gefunden und behoben werden, da die CI auf jeder erdenklichen Umgebung laufen kann [EGHS16].

[27] https://kubernetes.io/
[28] https://sqlite.org/index.html
[29] https://about.gitlab.com/

Im Zuge der CI wurde eine Pipeline zur Analyse der Codequalität eingeführt. Diese wird mittels Qodana[30] durchgeführt und dient als Kontrolle für übersehene oder vergessene Codesegmente. Ein Beispiel-Screenshot einer Analayse des Contact-Service ist in Anhang A.17 zu finden. Dabei besitzt die Pipeline kein Abbruchkriterium, wodurch diese immer erfolgreich durchläuft und rein informativer Natur ist. Dieses Verhalten ist frei konfigurierbar und sollte von der Qualitätssicherung vorgegeben werden. Im nächsten und letzten Schritt werden die Anwendungen paketiert. Dieser Schritt dient auch als reine Kontrolle, ob die übermittelten Änderungen einen startbaren Stand der Anwendung hinterlassen haben. Es werden keine Artefakte gespeichert.

Continuous Deployment

Während sich die CI mit der Integration von Code beschäftigt, stellt das Continuous Deployment (CD) Tools zur automatisierten Bereitstellung von Code zur Verfügung. Dabei ist CD ein Aufsatz auf *Continuous Delivery*. Im Unterschied zu Continuous Delivery werden bei CD nicht nur publizierbare und betriebsbereite Pakete erstellt, sondern auch deren automatisierte Bereitstellung in ausgewählte Umgebungen ermöglicht. Dadurch stehen Codeänderungen direkt für andere zur Verfügung, zum Beispiel in einem Testsystem [EGHS16].

In dieser Phase werden Docker-Images gebaut, gegebenenfalls mit einem Docker Tag versehen und in einer Registry veröffentlicht. Abhängig ist dies von dem Branch, auf dem der Code aktualisiert wurde. Bei...

- ...einem push auf die Branches *main*, *develop* oder der Erstellung eines Tags wird ein Docker Image gebaut, welches mit der kurzen Commit-SHA als Docker Tag versehen wird.
- ...einem push auf den *main* Branch wird das Image mit einem *latest* Docker Tag versehen. Dieser repräsentiert immer den aktuellen Produktionsstand und damit die aktuelle, stabile Version der Software.
- ...der Erstellung eines Tags wird ein Docker Image veröffentlicht, das mit der gleichen Versionsnummer versehen wird, die auch der Tag erhalten hat.

[30] https://www.jetbrains.com/help/qodana/welcome.html

Während der Entwicklung wurde lediglich mit dem *main* Branch gearbeitet. Die Pipelines wurden dennoch umgesetzt, da in naher Zukunft das trunk-based branching Modell[31] eingesetzt werden soll. Dabei arbeiten die Entwickler auf kurzlebigen *feature*-Branches, welche dann in den *develop* Branch integriert werden. Von dieser sogenannten *mainline* werden dann neue Versionen der Software abgeleitet.

Für die Umsetzung einer automatisierten Bereitstellung der Software wurde eine projektübergreifende Pipeline eingerichtet. Dabei liegen die Kubernetes Spezifikationen der Deploy-Units in einem eigenen Repository in git. Werden Änderungen an das Repository eines Services publiziert, so wird nach der Paketierung eine Pipeline auf diesem zentralen Repository angestoßen. Diese Pipeline erhält als Parameter den Namen des Deployments in Kubernetes sowie das Image, welches ausgeliefert werden soll, und führt diese Änderung auf dem Cluster aus.

[31] https://trunkbaseddevelopment.com/

5 Evaluation

Nach der Realisierung des Prototypen folgt nun eine Beurteilung relevanter Aspekte. Dazu wird in diesem Kapitel eine Bewertung in Bezug auf die Erfüllung der Anforderungen aus Kapitel 2 und der daraus entstandenen Architektur vorgenommen.

Zur Evaluation der Anforderungen werden zunächst die funktionalen Anforderungen betrachtet. Diese wurden final als User-Stories in Unterabschnitt 2.4.2 dokumentiert. Die folgende Tabelle 5.1 bietet einen Überblick über diese Anforderungen und inwiefern sie im Prototypen zur Verfügung stehen. Dabei wird zwischen der rein technischen Implementation der Geschäftslogik, und ob diese im Frontend benutzt werden kann.

User Story	Implementiert	Angebunden an Frontend
As a guest, I want to create an account	Ja	Ja
As a user, I want to confirm my email address	Ja	Ja
As a user, I want to reset my password if forgotten	Nein	Nein
As a user, I want to update my account information	Nein	Nein
As a user, I want to upload a profile picture	Ja	Nein
As a user, I want to delete my account	Nein	Nein
As a user, I want to enable or disable MFA	Ja	Ja
As a user, I want to login to my account	Ja	Ja
As a user, I want to create a new tenant	Ja	Ja
As a user, I want to join a tenant	Ja	Ja
As a guest, I want to join a tenant	Ja	Ja
As a user, I want to delete a tenant	Ja	Nein
As a user, I want to invite members to my tenant	Ja	Ja
As a user, I want to switch my tenant	Ja	Ja
As a user, I want to remove members from my tenant	Ja	Nein
As a user, I want to create a pool	Ja	Ja
As a user, I want to edit a pool	Ja	Ja
As a user, I want to delete a pool and its data	Ja	Ja
As a user, I want to create a contact	Ja	Ja
As a user, I want to edit a contact	Ja	Nein
As a user, I want to delete a contact	Ja	Ja
As a contact, I want to upload files to a pool	Ja	Ja
As a user, I want to upload files to a pool	Ja	Ja
As a contact, I want to download files from a pool	Ja	Ja
As a user, I want to download files from a pool	Ja	Ja

Tabelle 5.1: Überblick des Entwicklungsstandes der User Stories

Wie man sehen kann, wurden überwiegend alle Stories umgesetzt und stehen im Frontend des Prototypen zur Verfügung. Lediglich die Passwort-Zurücksetzen-Funktionalität, das Aktualisieren der Kontoinformationen des Benutzers und das Löschen des eigenen Kontos stehen noch nicht zur Verfügung. Die Passwort-Zurücksetzen-Funktionalität ist jedoch essentiell für jedes System, indem sich Benutzer mit einer Kombination aus Benutzername und Passwort authentifizieren müssen. Verlorene oder vergessene Passwörter würden ohne diese self-service Implementation viele Support-Anfragen generieren, die vermeidbar sind. Diese sind zudem zeitaufwendig, da sichergestellt werden muss, dass es sich auch wirklich um den Inhaber des Kontos handelt. Gleiches gilt für die fehlende Möglichkeit, die Kontoinformationen eines Benutzers zu aktualisieren. Die Möglichkeit zum Löschen eines Benutzerkontos bringt viel Verwaltungsaufwand mit sich, für den ein eigener Use-Case geschrieben werden müsste. So muss definiert werden, was mit den Mandaten geschieht, die dem Benutzer gehören. Hier muss die Eigenschaft des Eigentümers an einen anderen Benutzer übertragen werden oder zumindest die Bestätigung des

Benutzers eingeholt werden, ob auch alle Mandanten und damit verbunden nicht nur die eigenen Daten, sondern auch die von anderen Benutzern gelöscht werden sollen.

In Bezug auf die Anforderung der Skalierbarkeit des Systems ist die Wahl von Kubernetes die richtige gewesen. Mit Hilfe der Deployments und ReplicaSets können einzelne Teile des Systems dynamisch je nach Last skaliert werden. Dennoch birgt hier die Wahl der Programmiersprache Java für die Umsetzung der Microservices eine Schwäche. Auf einem klassischen Server teilen sich alle Java-Applikationen eine Java Virtual Machine (JVM)[1]. Die JVM führt den kompilierten Bytecode eines Java-Programms aus. Es interpretiert diesen und übersetzt ihn in tatsächliche Systembefehle auf dem unterliegenden Betriebssystem. Da jedoch durch die Nutzung von Containern jede Applikation in einem isolierten Prozess gestartet wird, muss auch jeder Container seine eigene JVM starten. Diese hat in Verbindung mit der eigentlichen Applikation im Vergleich zu anderen Sprachen einen erhöhten Speicherverbrauch[2]. So benötigt ein Service, ohne Anfragen zu erhalten, bereits im Schnitt über 300 Megabyte Arbeitsspeicher. Das schränkt die Skalierbarkeit zwar nicht ein, senkt aber das Potential, Kosten zu sparen, da generell mehr Ressourcen benötigt werden. Allgemein kann man jedoch festhalten, dass sich Sprachen wie Golang[3] durch den geringen Speicherverbrauch besser für die Realisierung der Services eignen.

Ein Kernaspekt des Prototypen ist der Datenschutz. Dieser äußert sich nicht nur in der Speicherung oder Verarbeitung von personenbezogenen Daten, sondern auch in deren Anzeige an Benutzer des Systems. Hierfür müssen Benutzer berechtigt werden. Durch das implementierte flache RBAC-Modell wird große Flexibilität in Bezug auf den Datenschutz gewonnen. So können Attribute von Kontakten oder Zugriffe in der Theorie sehr fein gesteuert werden. Im jetzigen Prototypen werden die Berechtigungen des Benutzers aus dem signierten AccessToken gezogen. Das ist sicherheits- und datenschutztechnisch ein Problem, nicht zwingend in Bezug auf die Kompromittierung über gefälschte AccessTokens, sondern viel mehr über die Aktualität der erteilten Berechtigungen. Ein solcher Token hat immer eine gewisse Lebenszeit. Ändern sich in dieser Lebenszeit die Berechtigungen eines Benutzers, so kann er bis zur Beendigung dieses Zeitraumes auch Aktionen ausführen, für die er gegebenenfalls nicht mehr berechtigt ist [JLE18].

Das gewählte Architekturkonzept der Microservices eignet sich sehr gut für den Betrieb in Kubernetes. Dabei bietet das Cluster die benötigten Funktionalitäten für die automatische Skalierung der Services. Insgesamt bietet das Konzept gute Wartbarkeit und

[1] https://www.w3schools.in/java-tutorial/java-virtual-machine/
[2] https://www.javaadvent.com/2018/12/docker-and-the-jvm.html
[3] https://go.dev/

das unabhängige Entwickeln von neuen Funktionen. Dennoch gab es bei der Entwicklung wesentliche Hürden, die überwunden werden mussten. Dadurch, dass die meisten Prozesse des Systems mandantenbezogen sind, erschwert sich die inter-service Kommunikation, wenn kein Benutzerbezug vorhanden ist. Wenn ein Benutzer Daten eines Services abfragt, woraufhin der Service einen anderen Service kontaktieren muss, stellt dieser Service die Anfrage im Namen des anfragenden Benutzers über den erhaltenen JWT. Ohne Benutzerbezug fehlt im Umkehrschluss auch besagter JWT, in welchem die Zuordnung zum Mandanten festgehalten ist. Das ist zum Beispiel bei der aktuellen Implementation der Virenerkennung der Fall. Diese erfolgt als regelmäßige Hintergrundaufgabe im File-Service. Wurde ein Virus erkannt, wird eine E-Mail an den Benutzer geschickt, der diese Datei hochgeladen hat. Dies kann entweder ein Kontakt oder ein Benutzer sein. Egal, um welchen Akteur es sich handelt, es müssen Daten aus einem anderen Service abgefragt werden, in diesem Fall die E-Mail Adresse. Dies stellt im Falle eines Benutzers kein Problem dar, da kein Mandantenbezug vorhanden sein muss. Die Benutzer sind im User-Service global gespeichert. Ist jedoch ein Kontakt betroffen, muss der aktuelle Mandant anders an den Service weitergeben werden, da das Schema des Mandanten aufgelöst werden muss, in diesem Fall über einen Header. Zusätzlich muss ein zusätzlicher Endpunkt, der nur von internen Services benutzt werden kann, vom Contact-Service bereitgestellt werden, da kein AccessToken verfügbar ist. Hier zeigt sich ein Nachteil der Entscheidung, die Authentifizierung und Autorisierung selbst zu implementieren. Dadurch, dass sich an keinen erprobten Standard gehalten wird, müssen die Services umfangreich erweitert werden, um Anfragen ohne Benutzerbezug verarbeiten zu können. Zudem kann so kein zero-trust Modell benutzt und nicht sichergestellt werden, dass die Anfrage von einem eigenen Service gesendet worden ist. Erhält ein Angreifer Zugang zum Netzwerk von Kubernetes, so kann dieser ohne Gegenwehr die internen Schnittstellen nutzen, um Daten zu erhalten oder manipulieren.

Durch den Einsatz von asynchronen fire-and-forget Nachrichten zur Propagierung von Events können Komponenten einfach und problemlos getauscht werden. Gleichzeitig erhöhen die genutzten Queues die Performanz. So können bei erhöhter Last beispielsweise der E-Mail Versand kontrolliert werden, wenn der Server mit dem Versenden von zu vielen E-Mails überlastet wird. Dazu ist auch anzumerken, dass das Event in jedem Fall an den Service ausgeliefert wird, auch wenn dieser zum Zeitpunkt der versuchten Zustellung nicht zur Verfügung stehen sollte. Hierzu sind im Broker Retry-Mechanismen implementiert die zwischengespeicherte Nachrichten erneut ausliefern, wenn der Service wieder erreichbar ist [Ind18, S. 76-85]. Die dadurch aufgezwungene eventual consisten-

cy ist kritisch zu betrachten, gerade für den Datenschutz. Dadurch, dass die Konsistenz zeitlich verzögert eintreten kann, ergibt sich das gleiche Problem wie bereits im Rahmen der Authentifizierung und Autorisierung angesprochen. Es ist im Bereich des Möglichen, dass Benutzer des Systems über einen kurzen Zeitraum Dinge sehen oder vornehmen können, für die sie eigentlich keine Berechtigungen mehr haben. Auslöser hierfür könnte beispielsweise der Ausfall des Brokers sein. Das ist gerade in Anbetracht auf die mögliche Einsicht in personenbezogenen Daten, wie hier im System gespeicherte Kontaktdaten oder Dateien, ein Risikofaktor. Dementsprechend ist in dem Fall der hier angestrebten Steigerung der Performanz abzusehen, und es sollte sich auf eine klare Konsistenz der Daten fokussiert werden.

Zum Datenschutz gehört auch die Speicherung von Dateien. Diese liegen, trotz der in Kapitel 1 vorgestellten Bedenken, bei dem amerikanischen Dienstleister AWS. Das ist in diesem besonderen Fall jedoch vertretbar. Zum einen werden keine personenbezogenen Daten der Benutzer an AWS übertragen. Die Interaktion mit den Servern von S3 findet ausschließlich über die Services des Backends statt. Zum anderen werden die Dateien vor der Übertragung verschlüsselt, wobei der Schlüssel AWS nicht bekannt ist. So können weder AWS noch Dritte auf die hochgeladenen Dateien zugreifen. Die einzige Schwachstelle ist das Caching auf dem Pool-Service selbst. Hier liegen die Dateien in einem lesbaren Format, bis der Upload finalisiert wird. Die vollständige Datei existiert zwar nur kurz, dennoch könnten aus den bereits hochgeladenen Teilen sensible Inhalte ausgelesen werden, wenn ein Angreifer Zugang zum Cache erhalten sollte.

Durch die Isolation der mandantenbezogenen Daten in eigene Schemata in der Datenbank wird zusätzlich eine weitere Hürde erschaffen, die es verhindert, dass durch etwaige Fehler in der Programmierung auf mandantenbezogene Daten zugegriffen werden kann. Gleichzeitig bietet sie die Möglichkeit, im Verlauf der Weiterentwicklung des Produkts spezifische Änderungen an einem einzelnen Datenbankschema zu ermöglichen. Das kann von Vorteil sein, wenn ein bestimmter Prozess eines Mandanten bestimmte Voraussetzungen mit sich bringt, die aber zu spezifisch für ein allgemeines Feature der Gesamtanwendung sind.

Ein weiteres Sicherheitsrisiko liegt in der Nutzung von externen Libraries. Zwar ist es meist zeitsparender, bereits umgesetzte Funktionalität zu nutzen. Zudem sind viele Libraries sehr gut getestet und von Experten entwickelt. Dazu kommen noch die Vorteile von Open-Source, dass Probleme direkt von einer breiten Community gefunden und behoben werden können. Dennoch ist der Einsatz von nicht selbst verwaltetem Code ein

Risiko. Als Beispiel dafür dient die just aufgedeckte Sicherheitslücke im bekannten und weit verbreitetem Java Logging-Framework Log4J, durch welche Code auf fremden Servern ausgeführt werden konnte (Remote Code Execution) [4]. Das Problem liegt hier in der Reproduzierbarkeit. Ist eine Schwachstelle gefunden, kann diese auf jedem System ausgenutzt werden, das dieselbe Version der Abhängigkeit verwendet. Ein weiteres Beispiel sind veraltete Libraries wie die im System zur Filterung genutzte *rsql-parser*-Library[5], welche gut getestet ist, aber nicht aktiv gewartet wird. So wurden die letzten Änderungen 2016 in das Projekt eingepflegt. Das kann daran liegen, dass keine Funktionalität mehr ergänzt werden muss und der bestehende Code gut getestet ist. Andererseits benutzt die Library auch andere Libraries, die wiederum neue Versionen bereitgestellt haben. Da die eigene Implementierung in den meisten Fällen jedoch nicht in demselben Umfang getestet werden kann, wie bereitgestellte Libraries, welche von Experten des Fachs entwickelt worden sind, empfiehlt es sich nicht, das Rad neu zu erfinden, sondern zu bestehenden Lösungen zu greifen.

Zuletzt ist noch anzumerken, dass ein wichtiges Sicherheitsmerkmal nicht berücksichtigt worden ist: Validierung. Übertragene Daten werden aktuell nicht validiert. So können Fehlermeldungen aus der Perspektive des Anwenders nicht wirklich differenziert und, wenn überhaupt, erst beim Speichern in die Datenbank bemerkt werden. Das bietet keine gute Benutzererfahrung und kann zu Inkonsistenzen in den Daten führen.

[4] https://www.cisa.gov/uscert/apache-log4j-vulnerability-guidance
[5] https://github.com/jirutka/rsql-parser

6 Schluss

Zur Abrundung der Arbeit wird zur Realisierung und zum fertigen Prototypen ein persönliches Fazit gezogen. Des Weiteren wird ein Ausblick für Optimierungen und Erweiterungen des konzipierten Systems gegeben.

6.1 Fazit

Das Ergebnis dieser Arbeit stellt einen Prototypen für eine mandantenfähige Architektur zum datenschutzkonformen Austausch von Dateien dar. Zusätzlich zur Konzeption dieses Prototypen wurden bereits Vorkehrungen zum Betrieb geliefert.

Bei der Realisierung ist aufgefallen, wie wichtig die Planung als Grundlage ist. Zudem hat sich verdeutlicht, dass diese Planung zwar durchaus hilft, aber dennoch nicht in Stein gemeißelt ist. So haben sich während der Umsetzung der Anforderungen neue Hürden aufgetan, die wiederum Einfluss auf die Änderung der bisher geplanten Architektur hatten. Es haben sich gerade die Sequenzdiagramme als durchaus nützlich herausgestellt. Sie boten eine gute Möglichkeit, Abläufe zu definieren, und konnten gut mit dem Code verglichen werden. Außerdem halfen sie bei der Erstellung von Testfällen, da solche Sequenzdiagramme bereits ein Szenario beschreiben.

Der Umfang der Realisierung einer Microservice Architektur durch nur eine Person wurde deutlich unterschätzt. Die Vorteile, die diese Architektur bietet, sind weitestgehend nur zu ernten, wenn mehrere kleine Teams am Projekt arbeiten. Es konnte nicht parallel an mehreren Services geschrieben werden, und Änderungen mussten sequentiell in allen Services umgesetzt werden. Das hat die Projektplanung teils durcheinander geworfen und eine neue Priorisierung der Aufgaben erzwungen.

Der Verzicht auf Test Driven Development (TDD) mit dem Ziel der Einsparung von Zeit hat sich ins Gegenteil verkehrt. Vorab definierte Testfälle hätten Probleme viel schneller

und präziser aufgedeckt als das stattgefundene Debugging auf der Suche von Fehlern nach der blinden Umsetzung. Die dadurch verlorene Zeit übersteigt nach eigener Einschätzung deutlich die zum Schreiben und Definieren von Tests.

6.2 Ausblick

Der Prototyp ist zwar soweit funktional, aber dennoch nicht reif für den Produktivbetrieb. Deshalb werden zum Abschluss dieser Arbeit die Probleme aus der Evaluation aufgegriffen und Möglichkeiten vorgestellt, diese zu beheben. Zudem werden kurz ein paar Ideen zur Erweiterung der Funktionalität des Systems vorgestellt. Das betrifft die Optimierung bestehender Prozesse und die Einbindung neuer Funktionen.

6.2.1 Sicherheit

Zur Ersetzung der Authentifizierung und Autorisierung sollte diese aus dem User-Service extrahiert werden und das System um einen Auth-Service erweitert werden. Dieser dient als Autorisierungsdienst im Konzept von OAuth. OAuth ist ein weit verbreiteter Standard und aktuell eine der besten Möglichkeiten zur sicheren Einbindung von Autorisierung. Im besten Fall sollte dieser auch OpenID Connect (OIDC) unterstützen. OIDC basiert auf OAuth, bringt aber zusätzlich noch die Möglichkeit, Benutzer zu authentifizieren. Der User-Service würde dann zu einem Tenant-Service migriert werden, der sich lediglich um die Mandanten und die RBAC Berechtigungen eines Benutzers im Kontext eines Mandanten kümmert. Die Services können dann nach Erhalt den AccessToken beim Auth-Service prüfen und je nach berechtigten *scopes* und Rollen entscheiden, wie fortgefahren werden soll. Dabei wird der Claim zum Identifizieren des Mandanten nicht mehr im AccessToken vorhanden sein. Dazu muss zusätzlich nach der Prüfung beim Auth-Service noch der Tenant-Service kontaktiert werden, welcher die Zuordnung zum Mandanten prüft.

Zwischen den Services könnte so auch eine geschützte Kommunikation stattfinden. Dazu würde für jeden Service ein Client in OAuth angelegt, welcher sich über den Client Credentials Grant Type einen AccessToken holen kann. Mithilfe dieses Tokens können die Services auch untereinander prüfen, ob der anfragende Service berechtigt ist, die gewünschte Aktion auszuführen. Das würde das Einschleusen eines kompromittierten Service in das System verhinden.

Die Nutzung von OAuth würde auch implizit die Problematik der nicht ohne weiteres möglichen Invalidierung von AccessTokens beheben. Die Spezifikation von OAuth sieht dabei einen Endpunkt vor, mithilfe dessen ein AccessTokens widerrufen werden kann. Folglich kann dieser direkt nach der Ausführung nicht mehr verwendet werden.

Neben der Authentifizerung und Autorisierung sollte außerdem eine Validierung von eingehenden Daten implementiert werden. Dazu zählt zum einen die strukturelle Prüfung der Daten und zum anderen eine logische Validierung. Durch strukturelle Prüfungen könnten viele Fehler, wie nicht übertragene Felder oder Formatierungsfehler, entdeckt und entsprechend kommuniziert werden, bevor die Daten in die Datenbank geschrieben werden. Die logische Validierung geht noch einen Schritt weiter und hälfe dabei, weitere mögliche Inkonsistenzen zu vermeiden. Dabei gilt diese Art der Prüfung vor allem Statusübergängen und dem damit verbundenen Zustand, da Entitäten je nach Status unterschiedliche Pflichtfelder haben können.

6.2.2 Automatisierungen

Das System bietet bereits ausgiebige Automatisierungen, so zum Beispiel die automatische Einrichtung einer Mandantenumgebung nach der Erstellung durch den Nutzer. Dennoch ergeben sich weitere Möglichkeiten, im aktuellen Stand noch manuelle Prozesse zu automatisieren. Dazu zählt zum Beispiel die Einrichtung einer eigenen Domain. Das ist derzeitig nur manuell über die Datenbank möglich, sollte aber vom User zu jeder Zeit durchgeführt werden können. Dazu sollte dieser eine kurze Anleitung erhalten, wie er seine Wunschdomain per CNAME-Eintrag auf die Domain des Dateiaustauschservers zeigen lassen kann. Auf der gleichen Seite könnte dann eine Möglichkeit zur Überprüfung der DNS-Einträge und der Eingabe der Wunschdomain eingebunden werden.

Da die Nutzung einer eigenen Domain Kosten nach sich zieht, würde sich die Umsetzung eines Abrechnungsmodells anbieten. Die Abrechnung könnte in einem eigenen Service stattfinden. In heutigen SaaS-Anwendungen wird meist mit nutzungsbasierten Plänen zur Abrechnung gearbeitet. Eben jene würde dieser Service definieren und sich zudem um die fristgerechte Abrechnung der genutzten Funktionalität kümmern. So würden optionale Erweiterungen wie die angesprochene Aufschaltung einer eigenen Domain, aber auch sich steigernde Quota-Limitierungen pro Plan, ermöglicht werden. Diese Limitierungen könnten sich in Upload- oder Downloadgeschwindigkeit, maximalen Dateigrößen oder begrenztem Speicherplatz äußern und würden dann an den jeweiligen Stellen in der

Geschäftslogik Anwendung finden. Grundlegende Vorkehrungen zur Umsetzung dieser Funktionalität sind bereits vorhanden, wie bereits in Abschnitt 2.5 vorweggenommen.

Literaturverzeichnis

[bae21] BAELDUNG: *A Guide to Multitenancy in Hibernate 5*. https://www.baeldung.com/hibernate-5-multitenancy. Version: 2021

[bez21] BEZKODER: *React Refresh Token with JWT and Axios Interceptors*. https://www.bezkoder.com/react-refresh-token/. Version: 2021

[Bra15] BRADBURY, Danny: Fighting ID theft [Security Personal Information]. DOI 10.1049/et.2016.1207. In: *Engineering Technology* 10 (2015), Nr. 12, S. 60–63

[Cad21] CADDY: *Caddy Documentation*. https://caddyserver.com/docs. Version: 2021

[Com16] COMMISSION, European: *VERORDNUNG (EU) 2016/679 DES EUROPÄISCHEN PARLAMENTS UND DES RATES*. https://eur-lex.europa.eu/legal-content/DE/TXT/PDF/?uri=CELEX:32016R0679. Version: 2016

[Dod18] DODDS, Kent C.: *Prop Drilling*. https://kentcdodds.com/blog/prop-drilling. Version: 2018

[EGHS16] EBERT, Christof; GALLARDO, Gorka; HERNANTES, Josune ; SERRANO, Nicolas: DevOps. DOI 10.1109/MS.2016.68. In: *IEEE Software* 33 (2016), Nr. 3, S. 94–100

[Foi21] FOITZICK, Klaus: *Datenschutz-Probleme beim Einsatz von US-Dienstleistern*. https://www.activemind.de/magazin/us-dienstleister/. Version: 2021

[Gac15] GACKENHEIMER, Cory: *Introduction to React*. Apress http://dx.doi.org/10.1007/978-1-4842-1245-5

[Gha18] Gharbi, Mahbouba; Koschel, Arne 1. (Hrsg.): *Basiswissen für Softwarearchitekten Aus- und Weiterbildung nach iSAQB-Standard zum Certified Professional for Software Architecture - Foundation Level*. 3., überarbeitete und aktualisierte Auflage. dpunkt.verlag, 2018

[Glo11] Gloger, Boris: *Scrum Produkte zuverlässig und schnell entwickeln*. 3., aktualisierte Auflage. Hanser Verlag http://dx.doi.org/10.3139/9783446426528

[Gol19] Golubski, Wolfgang: *Entwicklung verteilter Anwendungen Mit Spring Boot Co*. 1st ed. 2019. Springer Fachmedien Wiesbaden (erfolgreich studieren: Springer eBook Collection). https://doi.org/10.1007/978-3-658-26814-5

[GZK+16] Geisendörfer, Felix; Zonneveld, Kevin van; Koschützki, Tim; Venkataraman, Naren ; Kleidl, Marius: tus Protocol. – Forschungsbericht. – Online–Ressource. https://tus.io/protocols/resumable-upload.html

[ILKL20] Iskandar, Taufan F.; Lubis, Muharman; Kusumasari, Tien F. ; Lubis, Arif R.: Comparison between client-side and server-side rendering in the web development. DOI 10.1088/1757-899x/801/1/012136. In: *IOP Conference Series: Materials Science and Engineering* 801 (2020), may, Nr. 1, S. 012136

[Inc06] Inc, Amazon Web S.: *Amazon Simple Storage Service User Guide*. https://docs.aws.amazon.com/AmazonS3/latest/userguide/s3-userguide.pdf. Version: 2006

[Ind18] Indrasiri, Kasun; Siriwardena, Prabath (Hrsg.): *Microservices for the Enterprise Designing, Developing, and Deploying*. Apress L. P.. – 434 S. https://ebookcentral.proquest.com/lib/kxp/detail.action?docID=5598950

[JLE18] Jánoky, László; Levendovszky, J. ; Ekler, Péter: An analysis on the revoking mechanisms for JSON Web Tokens. DOI 10.1177/1550147718801535. In: *International Journal of Distributed Sensor Networks* 14 (2018), 09, S. 155014771880153

[Kru95] Kruchten, P.B.: The 4+1 View Model of architecture. DOI 10.1109/52.469759. In: *IEEE Software* 12 (1995), Nr. 6, S. 42–50. ISSN 0740–7459

[Kub21] Kubernetes: *DNS for Services and Pods*. https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/. Version: 2021

[Mag20] Maggio, Alessandro: *What is ORM? Object-Relational Mapping Explained*. https://www.ictshore.com/software-design/what-is-orm/. Version: 2020

[MKJR16] Moriarty, K.; Kaliski, B.; Jonsson, J. ; Rusch, A.: PKCS #1: RSA Cryptography Specifications Version 2.2 / RFC Editor. RFC Editor, November 2016 (8017). – RFC. – ISSN 2070–1721

[MM17] Martin, Kelly D.; Murphy, Patrick E.: The role of data privacy in marketing. In: *Journal of the Academy of Marketing Science* 45 (2017), Nr. 2, S. 135–155

[MMPR11] M'Raihi, D.; Machani, S.; Pei, M. ; Rydell, J.: TOTP: Time-Based One-Time Password Algorithm / RFC Editor. RFC Editor (6238). – RFC. – Online–Ressource. http://www.rfc-editor.org/rfc/rfc6238.txt. http://www.rfc-editor.org/rfc/rfc6238.txt. – ISSN 2070–1721

[Mus21] Musch, Olaf: *Design Patterns mit Java*. Springer Vieweg. – 669–682 S. DOI 10.1007/978-3-658-35492-3

[Not07] Nottingham, Mark: FIQL: The Feed Item Query Language / Internet Engineering Task Force. Internet Engineering Task Force (draft-nottingham-atompub-fiql-00). – Internet-Draft. – Online–Ressource. Work in Progress. https://datatracker.ietf.org/doc/html/draft-nottingham-atompub-fiql-00

[Ns14] Namiot, Dmitry; sneppe, Manfred sneps: On Micro-services Architecture. In: *International Journal of Open Information Technologies* 2 (2014), 09, S. 24–27

[Pol19] Polli, Roberto: RateLimit Header Fields for HTTP / IETF Secretariat (draft-polli-ratelimit-headers-00). – Internet-Draft. – Online–Ressource.

http://www.ietf.org/internet-drafts/draft-polli-ratelimit-headers-00.txt. http://www.ietf.org/internet-drafts/draft-polli-ratelimit-headers-00.txt

[PR15] Pohl, Klaus; Rupp, Chris: *Basiswissen Requirements Engineering : Aus- und Weiterbildung nach IREB-Standard zum Certified Professional for Requirements Engineering Foundation Level*. dpunkt.verlag, 2015

[Put18] In: Putrady, Ecky: *Configuration*. Apress. – ISBN 978–1–4842–3739–7, S. 209–217. DOI 10.1007/978-1-4842-3739-7_10

[Red20] Redhat: *What is multitenancy?* https://www.redhat.com/en/topics/cloud-computing/what-is-multitenancy. Version: April 2020

[Say20] Sayfan, Gigi: *Mastering Kubernetes : Level up Your Container Orchestration Skills with Kubernetes to Build, Run, Secure, and Observe Large-Scale Distributed Apps, 3rd Edition*. Packt Publishing, Limited https://ebookcentral.proquest.com/lib/hawhamburg-ebooks/detail.action?docID=6245753#

[SBH19] Saeed, Iman; Baras, Sarah ; Hajjdiab, Hassan: Security and Privacy of AWS S3 and Azure Blob Storage Services. DOI 10.1109/C-COMS.2019.8821735. In: *2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS)*, 2019 S. 388–394

[Sop20] Sophos: *What are Signatures and How Does Signature-Based Detection Work?* https://home.sophos.com/en-us/security-news/2020/what-is-a-signature. Version: 2020

[Sta20] Starke, Gernot 1.; Verlag, Carl H. (Hrsg.): *Effektive Softwarearchitekturen ein praktischer Leitfaden*. 9., überarbeitete Auflage. Hanser (Hanser eLibrary). http://dx.doi.org/10.3139/9783446465893

[Sta21a] Stack Overflow: *Stack Overflow 2021 Developer Survey*. https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-web-frameworks. Version: 2021

[Sta21b] Statista: *Organizations' adoption level of microservices worldwide in 2021*. https://www.statista.com/statistics/1233937/microservices-adoption-level-organization/. Version: 2021

[Tak17] Takai, Daniel 1.: *Architektur für Websysteme serviceorientierte Architektur, Microservices, domänengetriebener Entwurf.* Hanser (Hanser eLibrary). http://dx.doi.org/10.3139/9783446452480

[TG16] T. Golding, Z. Christopherson B. W.: SaaS Storage Strategies / AWS. – White Paper. – Online–Ressource. https://d0.awsstatic.com/whitepapers/Multi_Tenant_SaaS_Storage_Strategies.pdf

[Zö15] Zörner, Stefan: *Softwarearchitekturen dokumentieren und kommunizieren Entwürfe, Entscheidungen und Lösungen nachvollziehbar und wirkungsvoll festhalten.* 2., überarbeitete und erweiterte Auflage. Hanser (Hanser eLibrary). http://dx.doi.org/10.3139/9783446444423

A Anhang

A.1 Use-Case Beschreibungen

Use-Case Beschreibungen

UC/01 Create account

Use case	UC/01 Create account
Actors	• guest
Preconditions	• None
1. Enter user information 2. Submit form 3. Show confirmation 4. Click confirmation link in email	
Alternative flow 1	The guests mail address is already in use in step 1
1. Show a message to the guest, that the mail address is already in use 2. Continue at 1	
Postconditions	• A new user was saved to the database and was redirected to the dashboard

UC/02 Join tenant

Use case	UC/02 Join tenant
Actors	• user • guest
Preconditions	• Valid invitation link • User is not logged in
1. Click invitation link in email 2. Enter account information 3. Submit form	
Alternative flow 1	The actor already has an account
1. Click on “login” link 2. Proceed with UC/04	
Postconditions	• A new user was saved to the database

UC/03 Create tenant

Use case	UC/03 Create tenant
Actors	• user
Preconditions	• User is logged in
1. Click invitation link in email 2. Enter account information 3. Submit form	
Alternative flow 1	The actor already has an account
1. Click on “login” link 2. Proceed with UC/04	
Postconditions	• A new user was saved to the database

UC/04 Login

Use case	UC/04 Login

Actors	• user
Preconditions	• User is confirmed • User is not logged in
1. Enter login credentials 2. Submit form	
Alternative flow 1	The credentials provided in step 1 are invalid
1. Show a message of invalid credentials 2. Continue with step 1	
Alternative flow 2	The user has MFA enabled
1. Enter generated OTP 2. Confirm log in	
Alternative flow 3	The user forgot his password
1. Click “Forgot password” link 2. Enter email address 3. Click confirmation link in email 4. submit new password 5. Redirect to login page 6. Continue with step 1	
Postconditions	• The user is logged in with a new session and was redirected to the dashboard

UC/05 Manage pools

Use case	UC/05 Manage pools
Actors	• user
Preconditions	• User is logged in
1. Go to pools page 2. View list of pools	
Alternative flow 1	The user wants to create a pool
1. Click on “Create pool” button 2. Enter pool information 3. Submit form 4. Success message is shown 5. Redirect to pool	
Alternative flow 2	The user wants to edit a pool
1. Click on a pool in the list (or the edit button in context menu) 2. Update pool details 3. Submit form 4. Success message is shown	
Alternative flow 3	The user wants to delete a pool
1. Click on delete button in context menu of the button 2. Success message is shown	
Exceptional flow 1	When saving the pool, the user provided invalid data
1. Show error message to the user 2. The user may try again after changing values	
Postconditions	• When created or updated, the pool is saved to the database. • When deleted the pool and all its data are deleted.

UC/06 Manage contacts

Use case	UC/06 Manage contacts
Actors	• user
Preconditions	• User is logged in
1. Go to contacts page 2. View list of contacts	
Alternative flow 1	The user wants to create a contact
1. Click on “Create contact” button 2. Enter contact information 3. Submit form 4. Success message is shown 5. Add contact to list	
Alternative flow 2	The user wants to edit a contact
1. Click on a contact in the list (or the edit button in context menu) 2. Update contact details 3. Submit form 4. Success message is shown	
Alternative flow 3	The user wants to delete a contact
1. Click on delete button in context menu of the button 2. Success message is shown	
Exceptional flow 1	When saving the contact, the user provided invalid data
1. Show error message to the user 2. The user may try again after changing values	
Postconditions	• When created or updated, the contact is saved to the database. • When deleted the contact and all its data are deleted.

UC/07 Manage files

Use case	UC/07 Manage files
Actors	• user • contact
Preconditions	• Active session
1. Show a list of all uploaded files 2. Select file(s) by drag & drop or click on “Upload” 3. Success message is shown 4. Add file to list	
Alternative flow 1	The file belongs to an incomplete upload
1. Continue the upload 2. Continue with step 3	
Alternative flow 2	The type of the selected file is not allowed
1. Error message is shown 2. Continue with step 1	
Alternative flow 3	The maximum amount of files is reached
1. Error message is shown 2. Continue with step 1	
Alternative flow 4	An uploaded file should be downloaded
1. Select a file of all uploaded files 2. Click on “Download” button 3. Save file to computer	

Postconditions	• The file was uploaded and saved to the database or downloaded

UC/08 Manage tenant

Use case	UC/08 Manage tenant
Actors	• user
Preconditions	• User is logged in
1. Go to workspace settings 2. Update information 3. Submit form 4. Success message is shown	
Alternative flow 1	The user wants to delete the tenant
1. Confirm deletion 2. Redirect the user to the app portal 3. Success message is shown	
Alternative flow 2	The user wants to invite a member to the tenant
1. Go to members tab 2. Enter information 3. Send invitation 4. Success message is shown	
Alternative flow 3	The user wants to remove a user from the tenant
1. Go to members tab 2. Select user to delete 3. Click “Delete” button in the context menu 4. Success message is shown	
Exceptional flow 1	When saving the data, the user provided invalid data
1. Show error message to the user 2. The user may try again after changing values	
Postconditions	• When the information were updated, the data is persisted in the database • When a new member was invited, an invitation has been sent to the new member • If a user was removed, the user and its data are deleted from the database

UC/09 Update account

Use case	UC/09 Update account
Actors	• user
Preconditions	• User is logged in
1. Go to account settings 2. Update information	
Alternative flow 1	The user wants to enable MFA
1. Toggle MFA setting on 2. Scan QR-Code 3. Enter generated OTP and password 4. Confirm activation	
Alternative flow 2	The user wants to disable MFA
1. Toggle MFA setting off 2. Enter password 3. Confirm deactivation	

Alternative flow 3	The user wants to delete his account
1. Click on “Delete account” button 2. Confirm deletion	
Alternative flow 4	The user wants to upload a new profile picture
1. Select profile picture from files 2. Confirm selection 3. Success message is shown and profile picture is updated	
Exceptional flow 1	When saving the data, the user provided invalid data
1. Show error message to the user 2. The user may try again after changing values	
Postconditions	• When the information were updated, the data is persisted in the database • When MFA settings were updated, the new settings are saved to the database and active • When the user deleted his account, all data and tenants were deleted from the database

UC/10 Switch tenant

Use case	UC/10 Switch tenant
Actors	• user
Preconditions	• User is logged in
1. Open tenant selection in header 2. Click on tenant 3. Redirect with silent authentication	
Exceptional flow 1	The user is not authorized to log in to the selected tenant
1. Error message is shown 2. User may select another tenant 3. PC: Redirect to portal tenant selection	
Postconditions	• The user is logged in in a new tab for the selected tenant

A.2 Download Endpunkt im Pool-Service

```
@PreAuthorize("hasAnyAuthority('pools:interactor',
↪  'pools:manage')")
@GetMapping("/{fileId}")
public ResponseEntity<StreamingResponseBody>
↪  downloadSingleFile(
        JwtAuthentication authentication,
        @PathVariable UUID fileId,
        @PathVariable UUID poolId,
        final HttpServletResponse response
) {
    Pool pool = poolService.getPoolById(poolId);
    File file = poolService.getFileById(fileId);

    if(authentication.getAuthorities().stream().noneMatch(a ->
    ↪  a.getAuthority().equals("pools:manage"))
            && file.getUploaderId() !=
            ↪  authentication.getUserPrincipal().getId()) {
        continue;
    }

    storageAdapter.setBucketName(
        authenticati-
        ↪  on.getUserPrincipal().getTenant().getSchemaName()
    );

    Resource resource = storageAdapter.get(poolId.toString() +
    ↪  "/" + fileId.toString());

    log.info("Loading file {}", file.getFileName());

    if(resource == null) {
        return ResponseEntity.noContent().build();
    }
```

```
    response.setContentType(file.getMimeType());
    response.setHeader(
            "Content-Disposition",
            "attachment;filename=" + file.getFileName()
    );

    StreamingResponseBody stream = outputStream -> {
        byte[] buff = new byte[4096];
        int readLength;
        InputStream inputStream = resource.getInputStream();
        while ((readLength = inputStream.read(buff)) != -1) {
            outputStream.write(buff, 0, readLength);
        }
        inputStream.close();
        log.info("Wrote file {}", file.getFileName());
    };
    return ResponseEntity.ok(stream);
}
```

A.3 ZIP-Download Endpunkt im Pool-Service

```
@PreAuthorize("hasAnyAuthority('pools:interactor',
 ↪  'pools:manage')")
@GetMapping
public ResponseEntity<StreamingResponseBody> downloadZip(
        JwtAuthentication authentication,
        @PathVariable UUID poolId,
        @RequestParam(name = "f", required = false) String
         ↪  filter,
        final HttpServletResponse response
) {
    Pool pool = poolService.getPoolById(poolId);
    List<File> files;
    if(filter != null && !filter.trim().isEmpty()) {
        files = fileRepository.findAll(filter);
    } else {
        files = pool.getFiles();
    }

    log.info("Downloading zip");

    storageAdapter.setBucketName(
        authenticati-
         ↪  on.getUserPrincipal().getTenant().getSchemaName()
    );

    response.setContentType("application/zip");
    response.setHeader(
            "Content-Disposition",
            "attachment;filename=" + pool.getName() + ".zip"
    );

    StreamingResponseBody stream = outputStream -> {
        byte[] buff = new byte[4096];
        int readLength;
```

```
ZipParameters zipParameters = new ZipParameters();
try {
    ZipOutputStream zipOutputStream = new
     ↪  ZipOutputStream(response.getOutputStream());

    for (File file : files) {
        if(file.getPool().getId() != poolId) {
            response.setHeader("Content-Disposition",
             ↪  null);
            throw new ServiceException("Not allowed",
             ↪  HttpStatus.FORBIDDEN);
        }


         ↪  if(authentication.getAuthorities().stream().noneMatch(a
         ↪  -> a.getAuthority().equals("pools:manage"))
                && file.getUploaderId() != authentica-
                 ↪  tion.getUserPrincipal().getId())
                 ↪  {
            continue;
        }

        log.info("Loading file {}",
         ↪  file.getFileName());
        Resource data =
         ↪  storageAdapter.get(poolId.toString() + "/"
         ↪  + file.getId().toString());
        if(data == null) {
            continue;
        }
        zipParame-
         ↪  ters.setFileNameInZip(file.getFileName());
        zipOutputStream.putNextEntry(zipParameters);
        InputStream inputStream =
         ↪  data.getInputStream();
```

```
                while ((readLength = inputStream.read(buff)) !=
                ↪  -1) {
                    zipOutputStream.write(buff, 0, readLength);
                }
                inputStream.close();
                log.info("Wrote file {}", file.getFileName());
                zipOutputStream.closeEntry();
            }
            zipOutputStream.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    };

    return ResponseEntity.ok(stream);
}
```

A.4 CleanupUploadsJob.java

```
@Component
@Slf4j
public class CleanupUploadsJob {

    private final FileRepository fileRepository;

    private final TenantRepository tenantRepository;

    private final StorageProvider localStorageProvider;

    public CleanupUploadsJob(FileRepository fileRepository,
     ↪  TenantRepository tenantRepository, StorageProvider
     ↪  localStorageProvider) {
        this.fileRepository = fileRepository;
        this.tenantRepository = tenantRepository;
        this.localStorageProvider = localStorageProvider;
    }

    @Scheduled(cron = "@midnight")
    public void runJob() {
        log.info("Cleaning up old uploads");
        List<Tenant> tenants = tenantRepository.findAll();

        for (Tenant tenant : tenants) {
            TenantCon-
             ↪  text.setTenantSchema(tenant.getSchemaName());
            List<File> files = fileReposito-
             ↪  ry.findAllExpired(LocalDateTime.now().minusDays(5));
            for (File file : files) {
                localStorageProvider.delete(tenant.getId() +
                 ↪  "/" + file.getId() + ".part");
                fileRepository.delete(file);
            }
        }
```

```
            log.info("Cleaned up not resumed uploads");
        }

}
```

A.5 VirusDetectionService.java

```
@Service
@Slf4j
public class VirusDetectionService {

    @Value("${av.hostname}")
    private String hostname;

    @Value("${av.port}")
    private int port;

    private ClamavClient client;

    @PostConstruct
    private void init() throws IOException {
        this.client = new ClamavClient(hostname, port);
        log.info("Connecting to ClamAV server {}:{} ...",
        ↪  hostname, port);
    }

    public boolean scanFile(UUID fileId, InputStream
    ↪  inputStream) {
        if(client.getServer().isUnresolved()) {
            log.warn("ClamAV Server not responding");
            throw new RuntimeException("ClamAV Server not
            ↪  responding");
        }
        ScanResult result = client.scan(inputStream);
        if(result instanceof ScanResult.VirusFound) {
            log.error("Virus check was positive for file {}",
            ↪  fileId.toString());
            Map<String, Collection<String>> viruses =
            ↪  ((ScanResult.VirusFound)
            ↪  result).getFoundViruses();
            log.error("Found {} viruses", viruses.size());
```

```
            return false;
        } else {
            log.info("No viruses found for file {}",
             ↪  fileId.toString());
            return true;
        }
    }

}
```

A.6 CheckUploadsJob.java

```
@Component
@Slf4j
public class CheckUploadsJob {

    private final FileRepository fileRepository;

    private final TenantRepository tenantRepository;

    private final VirusDetectionService virusDetectionService;

    private final S3StorageAdapter storageProvider;

    private final PoolService poolService;

    private final MailService mailService;

    public CheckUploadsJob(FileRepository fileRepository,
     ↪  TenantRepository tenantRepository,
     ↪  VirusDetectionService virusDetectionService,
     ↪  S3StorageAdapter storageProvider, PoolService
     ↪  poolService, MailService mailService) {
        this.fileRepository = fileRepository;
        this.tenantRepository = tenantRepository;
        this.virusDetectionService = virusDetectionService;
        this.storageProvider = storageProvider;
        this.poolService = poolService;
        this.mailService = mailService;
    }

    @Scheduled(cron = "0/30 * * * * *")
    public void init() throws IOException {
        log.info("Checking uploads for viruses");

        tenantRepository.findAll().forEach(this::run);
```

```
        log.info("Finished job");
    }

    public void run(Tenant tenant) {
        TenantContext.setTenantSchema(tenant.getSchemaName());
        storageProvider.initialize(tenant);

        List<File> unchecked = fileRepository.findUnchecked();

        for (File file : unchecked) {
            Resource resource = storageProvi-
             ↪  der.get(file.getPool().getId().toString() + "/"
             ↪  + file.getId().toString());
            try {
                if(virusDetectionService.scanFile(file.getId(),
                 ↪  resource.getInputStream())) {
                    file.setChecked(true);
                    fileRepository.save(file);
                } else {
                    storageProvi-
                     ↪  der.delete(file.getPool().getId().toString()
                     ↪  + "/" + file.getId().toString());
                    fileRepository.delete(file);

                    try {
                        ContactDto contactDto = poolSer-
                         ↪  vice.getContactById(file.getUploaderId());

                        HashMap<String, Object> data = new
                         ↪  HashMap<>();
                        data.put("ts", null);
                        data.put("file_name",
                         ↪  file.getFileName());
```

```
                        data.put("uploader_name",
                        ↪  (contactDto.getFirstName() + " " +
                        ↪  contactDto.getFamilyName()));
                        data.put("uploader_email",
                        ↪  contactDto.getEmail());

                        RecipientDto uploader = new
                        ↪  RecipientDto(contactDto.getEmail(),
                        ↪  (contactDto.getFirstName() + " " +
                        ↪  contactDto.getFamilyName()), data);

                        SendMailEvent mail = MailBuil-
                        ↪  der.template("virusDetection.mjml")
                                .subject("[scotty] Schädliche
                                ↪  Datei entfernt")
                                .from("Scotty Security")
                                .addRecipient(uploader)
                                .build();

                        mailService.sendMail(mail);
                    } catch (Exception ex) {
                        // Ignore - Contact not found or user
                        ↪  uploaded the file
                    }

                    log.info("Deleted file {}", file.getId());
                }
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }

}
```

A.7 JpaFilterRepository.java

A.8 FilterRepository.java

```
public class FilterRepository<T, K extends Serializable>
 ↪  extends SimpleJpaRepository<T, K> implements
 ↪  JpaFilterRepository<T, K> {

    private EntityManager entityManager;
    private Class<T> clazz;

    public FilterRepository(JpaEntityInformation<T, ?>
     ↪  entityInformation, EntityManager entityManager) {
        super(entityInformation, entityManager);
        this.clazz = entityInformation.getJavaType();
        this.entityManager = entityManager;
    }

    public FilterRepository(Class<T> domainClass, EntityManager
     ↪  em) {
        super(domainClass, em);
    }

    @Transactional
    @Override
    public List<T> findAll(String query) {
        JpaCriteriaQueryVisitor<T> visitor = new
         ↪  JpaCriteriaQueryVisitor<>();
        visitor.setEntityClass(clazz);

        Node rootNode = new RSQLParser().parse(query);

        CriteriaQuery<T> criteriaQuery =
         ↪  rootNode.accept(visitor, entityManager);

        return entityMana-
         ↪  ger.createQuery(criteriaQuery).getResultList();
    }
```

```
}
```

A.9 JpaConfig.java

```
@Configuration
@EnableJpaRepositories(
        basePackages = {"de.jkdv.scotty.poolservice.data.*"},
        repositoryBaseClass = FilterRepository.class
)
public class JpaConfig {

}
```

A.10 FlywayConfig.java

```
@Configuration
@Slf4j
public class FlywayConfig {

    public static final String DEFAULT_SCHEMA = "public";

    @Bean
    public Flyway flyway(DataSource dataSource) {
        log.info("Migrating global schema");
        Flyway flyway =
        ↪  Flyway.configure().locations("db/migration/global")
                .dataSource(dataSource)
                .schemas(DEFAULT_SCHEMA)
                .validateOnMigrate(false)
                .load();
        flyway.migrate();
        return flyway;
    }

    @Bean
    public boolean tenantsFlyway(TenantRepository
    ↪  tenantRepository, DataSource dataSource) {
        tenantRepository.findAll().forEach(tenant -> {
            log.info("Migrating tenant schema: {}",
            ↪  tenant.getSchemaName());
            String schema = tenant.getSchemaName();
            Flyway flyway = Fly-
            ↪  way.configure().locations("db/migration/tenants")
                    .dataSource(dataSource)
                    .schemas(schema)
                    .validateOnMigrate(false)
                    .load();
            flyway.migrate();
        });
```

```
            return true;
        }

}
```

A.11 TenantSchemaResolver.java

```
@Component
public class TenantSchemaResolver implements
↪  CurrentTenantIdentifierResolver {

    @Override
    public String resolveCurrentTenantIdentifier() {
        String t =  TenantContext.getTenantSchema();
        return Objects.requireNonNullElse(t,
        ↪  FlywayConfig.DEFAULT_SCHEMA);
    }

    @Override
    public boolean validateExistingCurrentSessions() {
        return true;
    }

}
```

A.12 TenantConnectionProvider.java

```
@Component
@Slf4j
public class TenantConnectionProvider implements
 ↪  MultiTenantConnectionProvider {

    private final DataSource datasource;

    public TenantConnectionProvider(DataSource dataSource) {
        this.datasource = dataSource;
    }

    @Override
    public Connection getAnyConnection() throws SQLException {
        return datasource.getConnection();
    }

    @Override
    public void releaseAnyConnection(Connection connection)
     ↪  throws SQLException {
        connection.close();
    }

    @Override
    public Connection getConnection(String tenantIdentifier)
     ↪  throws SQLException {
        log.info("Get connection for schema {}",
         ↪  tenantIdentifier);
        final Connection connection = getAnyConnection();
        connection.setSchema(tenantIdentifier);
        return connection;
    }

    @Override
```

```
    public void releaseConnection(String tenantIdentifier,
    ↪  Connection connection) throws SQLException {
        log.info("Release connection for schema {}",
        ↪  tenantIdentifier);
        connection.setSchema(FlywayConfig.DEFAULT_SCHEMA);
        releaseAnyConnection(connection);
    }

    @Override
    public boolean supportsAggressiveRelease() {
        return false;
    }

    @Override
    public boolean isUnwrappableAs(Class unwrapType) {
        return false;
    }

    @Override
    public <T> T unwrap(Class<T> unwrapType) {
        return null;
    }

}
```

A.13 TenantContext.java

```
@Slf4j
public final class TenantContext {

    private static final ThreadLocal<String> context = new
     ↪  ThreadLocal<>();

    public static void setTenantSchema(String schema) {
        log.info("Set tenant schema to {}", schema);
        context.set(schema);
    }

    public static String getTenantSchema() {
        return context.get();
    }

    public static void clear() {
        context.set(null);
    }

}
```

A.14 Axios Silent Token Refresh Interceptor

```
let interceptor;

export function createAxiosResponseInterceptor(tokens:
↪  AuthTokens, update) {
    const { authenticationApi } = getApiClient();

    if (interceptor) {
        axios.interceptors.response.eject(interceptor);
    }

    interceptor = axios.interceptors.response.use(
        (response) => response,
        (error) => {
            // Reject promise if usual error
            if (!error.response || error.response?.status !==
            ↪  401) {
                return Promise.reject(error.response ?
                ↪  error.response : error);
            }

            /*
             * When response code is 401, try to refresh the
↪  token.
             * Eject the interceptor so it doesn't loop in case
             * token refresh causes the 401 response
             */
            axios.interceptors.response.eject(interceptor);

            if (typeof window !== "undefined") {
                console.log("Refreshing access token client
                ↪  side");
            } else {
                console.log("Refreshing access token ssr");
            }
```

```
if (!tokens?.refresh_token) {
    return Promise.reject(error);
}

const { tenant_id } = jwtDecode<{ tenant_id?:
↪  string }>(
    tokens?.access_token
);

return authenticationApi
    .refresh(
        {
            refresh_token: tokens.refresh_token,
        },
        tenant_id
    )
    .then((response) => {
        if (typeof window !== "undefined") {
            tokens = response.data;
            if (tokens && update) {
                update(response.data);
            }
        }
        er-
        ↪  ror.response.config.headers["Authorization"]
        ↪  =
            "Bearer " + response.data.access_token;
        return axios(error.response.config);
    })
    .catch((error) => {
        //destroyToken();
        tokens = {
            access_token: null,
            refresh_token: null,
        };
```

```
                    if (typeof window !== "undefined" &&
                    ↪  update) {
                        update(null);
                    }
                    return Promise.reject(error);
                })
                .finally(() =>
                ↪  createAxiosResponseInterceptor(tokens,
                ↪  update));
        }
    );
}
```

A.15 HibernateConfig.java

```
@Configuration
public class HibernateConfig {

    private final JpaProperties jpaProperties;

    public HibernateConfig(JpaProperties jpaProperties) {
        this.jpaProperties = jpaProperties;
    }

    @Bean
    JpaVendorAdapter jpaVendorAdapter() {
        return new HibernateJpaVendorAdapter();
    }

    @Bean
    LocalContainerEntityManagerFactoryBean
     ↪  entityManagerFactory(
            DataSource dataSource,
            MultiTenantConnectionProvider
             ↪  multiTenantConnectionProvider,
            CurrentTenantIdentifierResolver
             ↪  tenantIdentifierResolver
    ) {

        Map<String, Object> jpaPropertiesMap = new
         ↪  HashMap<>(jpaProperties.getProperties());
        jpaPropertiesMap.put(Environment.MULTI_TENANT,
         ↪  MultiTenancyStrategy.SCHEMA);
        jpaProperties-
         ↪  Map.put(Environment.MULTI_TENANT_CONNECTION_PROVIDER,
         ↪  multiTenantConnectionProvider);
        jpaProperties-
         ↪  Map.put(Environment.MULTI_TENANT_IDENTIFIER_RESOLVER,
         ↪  tenantIdentifierResolver);
```

```
        LocalContainerEntityManagerFactoryBean em = new
        ↪  LocalContainerEntityManagerFactoryBean();
        em.setDataSource(dataSource);
        em.setPackagesToScan("de.jkdv*");
        em.setJpaVendorAdapter(this.jpaVendorAdapter());
        em.setJpaPropertyMap(jpaPropertiesMap);
        return em;
    }


}
```

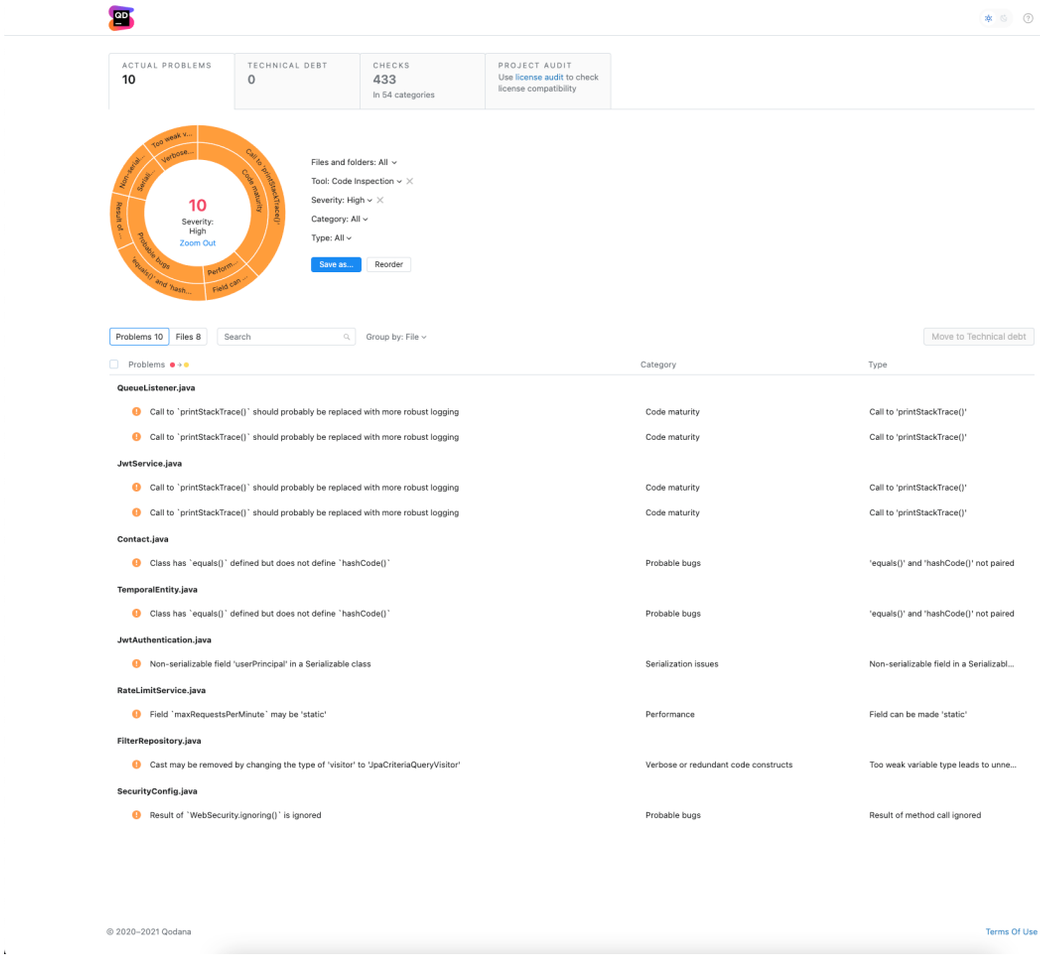
A.16 AuthFilter.java

```
@Slf4j
public class AuthFilter extends OncePerRequestFilter {

    private final JwtService jwtService;
    private final TenantService tenantService;

    public AuthFilter(JwtService tokenservice, TenantService
    ↪ tenantService) {
        this.jwtService = tokenservice;
        this.tenantService = tenantService;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest
    ↪ httpServletRequest, HttpServletResponse
    ↪ httpServletResponse, FilterChain filterChain) throws
    ↪ ServletException, IOException {
        String token =
        ↪ jwtService.resolveToken(httpServletRequest);
        log.info("Token: {}", token);
        if (token != null && jwtService.validateToken(token)) {
            String uuid = jwtService.getUserId(token);
            log.info("User with id: {} authenticated", uuid);

            Claims claims = jwtService.getClaims(token);
            String tenantId = claims.get("tenant_id",
            ↪ String.class);

            Tenant tenant = null;
            if(tenantId != null) {
                tenant = tenantSer-
                ↪ vice.getTenantById(UUID.fromString(tenantId));
                TenantCon-
                ↪ text.setTenantSchema(tenant.getSchemaName());
```

```
            }

            List<SimpleGrantedAuthority> authorities = new
             ↪  ArrayList<>();
            List<String> jwtAuthorities =
             ↪  claims.get("authorities", List.class);

            String firstName = claims.get("first_name",
             ↪  String.class);
            String familyName = claims.get("family_name",
             ↪  String.class);
            String email = claims.get("email", String.class);

            for(String authority : jwtAuthorities) {
                authorities.add(new
                 ↪  SimpleGrantedAuthority(authority));
            }
            Authentication authentication = new
             ↪  JwtAuthentication(new UserPrincipal(uuid,
             ↪  UUID.fromString(uuid), tenant, token, email,
             ↪  firstName, familyName), authorities);
            SecurityContextHol-
             ↪  der.getContext().setAuthentication(authentication);
        } else {
            String tenantId =
             ↪  httpServletRequest.getHeader("tenant");
            if(tenantId != null) {
                Tenant tenant = tenantSer-
                 ↪  vice.getTenantById(UUID.fromString(tenantId));
                TenantCon-
                 ↪  text.setTenantSchema(tenant.getSchemaName());
            }
        }

        filterChain.doFilter(httpServletRequest,
         ↪  httpServletResponse);
```

```
        }
}
```

A.17 Qodana Code Quality Linter - Ergebnisansicht



A.18 CI Pipeline der Java-Services

```
image: alpine:latest

stages:
  - qa
  - build
  - test
  - dockerize
  - tag
  - deploy

variables:
  SPRING_PROFILES_ACTIVE: test
  MAVEN_OPTS: "-Dmaven.repo.local=.m2/repository"
  DOCKER_REGISTRY: git.haw-hamburg.de:5005
  DOCKER_REPOSITORY_URL:
   ↪  $DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$CI_PROJECT_NAME
  DOCKER_IMAGE_TAGGED:
   ↪  $DOCKER_REPOSITORY_URL:$CI_COMMIT_SHORT_SHA
  DOCKER_IMAGE_LATEST: $DOCKER_REPOSITORY_URL:latest
  DOCKER_IMAGE_CURRENT_VERSION_TAG:
   ↪  $DOCKER_REPOSITORY_URL:$CI_COMMIT_REF_NAME

cache:
  paths:
    - .m2/repository

build:
  stage: build
  tags:
    - docker
  image: maven:3.6.3-jdk-11-slim
  script:
    - mvn package -DskipTests
```

```
qodana:
  stage: qa
  tags:
    - docker
  image:
    name: jetbrains/qodana-jvm
    entrypoint: ['']
  script:
    - /opt/idea/bin/entrypoint
     ↪  --results-dir=$CI_PROJECT_DIR/qodana --save-report
     ↪  --report-dir=$CI_PROJECT_DIR/qodana/report
  artifacts:
    paths:
      - qodana

#testing:
#  stage: test
#  tags:
#    - docker
#  image: maven:3.6.3-jdk-11-slim
#  services:
#    - postgres:11.4
#  script:
#    - mvn test

build_docker_image:
  stage: dockerize
  image: docker:latest
  services:
    - docker:dind
  tags:
    - dind
    - docker
  before_script:
    - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN
     ↪  $DOCKER_REGISTRY
```

```
  script:
    - docker build -t $DOCKER_IMAGE_TAGGED .
    - docker push $DOCKER_IMAGE_TAGGED
  only:
    - main
    - tags
    - develop

tag_latest_docker_image:
  stage: tag
  image: docker:latest
  services:
    - docker:dind
  tags:
    - dind
    - docker
  before_script:
    - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN
    ↪  $DOCKER_REGISTRY
  script:
    - docker pull $DOCKER_IMAGE_TAGGED
    - docker tag $DOCKER_IMAGE_TAGGED $DOCKER_IMAGE_LATEST
    - docker push $DOCKER_IMAGE_LATEST
  only:
    - main

tag_release_docker_image:
  stage: tag
  image: docker:latest
  services:
    - docker:dind
  tags:
    - dind
    - docker
  before_script:
```

```
    - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN
      ↪  $DOCKER_REGISTRY
  script:
    - docker pull $DOCKER_IMAGE_TAGGED
    - docker tag $DOCKER_IMAGE_TAGGED
      ↪  $DOCKER_IMAGE_CURRENT_VERSION_TAG
    - docker push $DOCKER_IMAGE_CURRENT_VERSION_TAG
  only:
    - tags


# Deployment
deploy:
  stage: deploy
  only:
    - main
    - tags
  trigger:
    project: scotty/deployment
    branch: main
    strategy: depend
  variables:
    DEPLOY_IMAGE: $DO-
      ↪  CKER_REGISTRY/$CI_PROJECT_NAMESPACE/$CI_PROJECT_NAME:$CI_COMMIT_SHORT_
    SERVICE_NAME: $CI_PROJECT_NAME
```

A.19 Kubernetes Cluster Konfiguration

```
---
hetzner_token: # Hetzner API token
cluster_name: ba-dev-cluster
kubeconfig_path: "./kubeconfig"
k3s_version: v1.21.3+k3s1
public_ssh_key_path: "~/.ssh/id_rsa.pub"
private_ssh_key_path: "~/.ssh/id_rsa"
ssh_allowed_networks:
  - 0.0.0.0/0
verify_host_key: false
location: nbg1
schedule_workloads_on_masters: true
masters:
  instance_type: cx21
  instance_count: 1
worker_node_pools:
- name: worker-pool-1
  instance_type: cx21
  instance_count: 3
```

A.20 Caddyserver Konfiguration - Caddyfile

```
{
    email td@jkdv.de
    on_demand_tls {
        ask https://api.scotty.cloud/tenants/check
        interval 2m
        burst 5
    }
}

*.scotty.cloud {
    tls {
        dns digitalocean <DigitalOcean API Token>
    }
    reverse_proxy web-app-peer.scotty.svc.cluster.local
}

https:// {
    tls {
        on_demand
    }
    reverse_proxy web-app-peer.scotty.svc.cluster.local
}
```

Glossar

AccessToken Ein AccessToken dient zur Identifikation und Authentifizierung eines Benutzers der Anwendung. Er muss, wenn existent, bei jeder Anfrage im Header übertragen werden.

ACID ACID steht für Atomic, Consistent, Isolated Durable und definiert angestrebte Eigenschaften für Transaktionen in datenbankorientierten Anwendungen.

API Abkürzung für Application Programming Interface. Eine API bietet Schnittstellen zur Ausfühung von Aktionen.

Backlog Die Summe aller User-Stories ergeben den Backlog.

Base64 Kodierung für Text.

Branch Ein Branch ist eine Abzweigung des Hauptzweiges in der Programmierung. Es gibt dabei immer einen Hauptbranch. Auf diesen müssen alle Zweigbranches zu einem Zeitpunkt wieder geführt werden.

Chunk Ein Chunk ist eine Teilmenge einer Datenmenge.

Claim Eigenschaft eines JWT.

Cluster Ein Cluster ist ein Zusammenschluss mehrerer Nodes, die als eine Einheit agieren. Dabei unterscheidet man meist in Worker und Master Nodes.

Containerisierung Die Containerisierung ist ein beliebtes Konzept, bei dem keine virtuellen Maschinen mehr gebraucht werden um lokal unabhängige Umgebungen vom Host-System zu errichten. Es bietet Flexibilität und Einfachheit.

DNS DNS steht für Domain Name Service. Sie bieten Nameserver an, mit Hilfe derer Adressen in IP-Adressen übersetzt werden können.

Docker Docker bietet eine Plattform zur einfachen Ausführung isolierter Prozesse ohne eigenem Betriebssystem.

Docker Tag Mit Hilfe eines Tags werden Versionen eines Images markiert. So ist dieses eindeutig identifizierbar.

git Versionskontrollsystem für Programmcode.

Header Ein Header im HTTP-Protokoll überträgt Metainformation zu einer Anfrage.

Healthcheck Ein Healthcheck wird beispielsweise für die Lastverteilung und das Monitoring benutzt. Hierfür wird ein Endpunkt bereitgestellt, über den der Zustand der Anwendung abgefragt werden kann.

HTTP Protokoll, welches die Grundlage der Datenkommunikation im Web bildet.

Image Ein Docker-Image ist ein Archiv, dass den Programmcode und seine Abhängigkeiten enthält. Es beinhaltet zudem Instruktionen zum Start der Anwendung und kann von Docker interpretiert werden.

JavaScript Scriptsprache für Webanwendungen.

Job Ein Job ist eine Aufgabe, die normalerweise im Hintergrund der eigentlichen Anwendung ausgeführt wird. Jobs können einmalig oder periodisch ausgeführt werden.

JSON JSON ist ein Dateiformat, dass oft für API Antworten verwendet wird.

JWT Ein JSON Web Token ist ein signiertes JSON Objekt, welches als AccessToken dienen kann. Es dient zur Authentifizierung und liefert gleichzeitig Metainformationen über den Benutzer an den Client.

Kernel Der Kernel ist das Herz eines Betriebssystems und steuert beispielsweise das Speichermanagement.

Kubernetes Kubernetes ist eine Orchestrierungs-Plattform für den verfügbaren Betrieb von Containern.

Multi-Tenancy Mehrere Mandanten benutzen die gleiche Instanz eines Systems.

Objekt Objekt im OOP (object oriented programming), eine Instanz einer Klasse aus Java beispielsweise.

Orchestrierung Die Orchestrierung basiert auf der Containerisierung und bietet ein Lebenszyklus-Management für die Container. Dazu gehören automatisierte Neustarts von abgestürtzten Containern oder die dynamische Skalierung der Containeranzahl.

Pool Ein Pool ist im Kontext der Architektur eine Sammlung von Dateien..

RefreshToken Der RefreshToken kann benutzt werden, wenn der kurzlebige AccessToken abgelaufen ist. Dafür wird er gegen ein neues Tokenpaar gestauscht.

Registry Eine Registry ist ein Speicher für Docker Images. Hier können Versionen von gebauten Images abgelegt und -gerufen werden.

REST REST, ausgeschrieben Representational State Transfer, ist ein Modell basierend auf HTTP zur Erstellung von APIs.

Reverse Proxy Ein Reverse Proxy wird vor eine interne Anwendung geschaltet und ist aus dem öffentlichen Netz aufrufbar. Er dient unter anderem zur Lastverteilung.

Schema Ein Schema kann in Datenbanken benutzt werden um Kontexte zu trennen, aber nur eine Datenbank zu verwenden. Schemas sind voneinander unabhängig und können ohne Konfiguration miteinander kommunizieren.

Tag Ein Tag in git stellt einen archivierten Zweig der Programmierung dar. Er setzt den aktuellen Programmcode fest und markiert so Versionen einer Anwendung.

Erklärung zur selbstständigen Bearbeitung

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

Ort Datum Unterschrift im Original