Wissenschaftliche Ausarbeitung

des Projektes

VerseVault

für das Modul

„Cloud Engineering“

des Masterstudiengangs Angewandte Informatik

von

Pascal Friedrichsen	670188
Dominik Heckner	670005
Fabian Petersen	670111

Inhaltsverzeichnis

1. Einleitung

VerseVault ist eine Website, um online Schriftstücke zu publizieren und zu lesen. Was für ein Werk ist dem Autor selbst überlassen. Sowohl fiktive als auch nicht-fiktive Werke können hochgeladen werden – so können zum Beispiel Kurzgeschichten und Romane, aber auch Essays, hochgeladen werden. Angebunden daran herrscht ein Monetarisierung-Modell, welches es ermöglicht, dass angehende Schriftsteller mit ihren Werken Geld verdienen. Dafür wird die interne Währung VerseVault-Coins, kurz *VV-Coins*, verwendet. Diese Münzen lassen sich mit echtem Geld erwerben (1 Euro → 100 VV-Coins). Beim Kauf eines Werkes bekommt der Autor die bezahlten VV-Coins gutgeschrieben und kann diese entweder selbst auf der Plattform ausgeben, oder zu einem verminderten Wert in Echtgeld umwandeln (100 VV-Coins → 80 Cent). Um Werke hochzuladen, erstellt der Autor ein „Buch“ oder wählt eines der vorherigen eigenerstellten Bücher aus. Ein Buch besteht aus einem Namen, einer Beschreibung und „Kapiteln“. Ein Kapitel besteht aus einem Namen, den textlichen Inhalt und einen Preis. So kann ein Buch mehrere zusammenhängende Kapitel haben, oder der Autor nutzt das Buch nur als „Sammelstelle“ für ähnliche Texte, so zum Beispiel als Kurzgeschichten-Sammlung. Erstellte Kapitel sind am Anfang noch nicht öffentlich gestellt und können auf der Plattform geschrieben werden. Sobald das Kapitel bereit ist, kann es publiziert werden und wird dann jedem anderen Nutzer auf der Plattform angezeigt. Falls kleine Fehler im publizierten Werk sind, können diese im Nachhinein auch herauseditiert werden.

Eine Plattform wie VerseVault lebt von Nutzercontent und Nutzerinteraktion, weshalb für ein marktfähiges Produkt Social-Media-Features wie Nutzerprofile, Nutzerprofilen folgen, Werke bewerten und Werke kommentieren, wichtige Features sind. Auch Werken Tags geben zu können macht es einfacher, dass die richtigen Personen das Werk finden.

Dazu gibt es auch viele Quality-Of-Life Features, welche das Nutzen von VerseVault angenehmer machen kann. Das beinhaltet eine Suchfunktion, eine Trendpage mit den neusten und beliebtesten Werken und eine Download-Funktion, die es ermöglicht, Werke herunterzuladen. Auch die Funktion direkt VV-Coins an Autoren spenden zu können, macht es einfacher für Autoren Geld zu verdienen, ohne ihre Zielgruppe mit einer Paywall auszusperren.

Aber auch für die Seitenbetreiber gibt es einige Funktionen, die wichtig sind. So ist ein Plagiats-Erkenner essenziell, um die Nutzer der Plattform, aber auch sich selbst rechtlich zu schützen. Auch Admin-Tools zum Moderieren der Seite sind wichtig, um so schädliche, illegale, oder einfach ungewollten Inhalt von der Seite zu entfernen. Ein Daten-Dashboard - welches einem z.B. die momentanen VV-Coins im Umlauf und anderen Traffic anzeigt - können helfen, bessere Prognosen über die Nutzung der Seite zu erhalten.

VerseVault wird jedoch nur im Rahmen des 6cp ein-Semester Moduls „Cloud Engineering“ des Master-Studienganges Angewandte Informatik an der Hochschule Flensburg entwickelt. Somit fehlen sowohl

die Zeit und die Mittel, um VerseVault auf dieser Stufe zu entwickeln. Es wird darauf abgezielt, die grundlegenden Funktionen zu implementieren, also die Möglichkeit Texte zu schreiben, hochzuladen und anderen bereitzustellen. Falls die Arbeitslast es zulässt, werden auch weitere Funktionen hinzugefügt, wie das Kaufsystem. Alle weiteren Funktionen gehören zu einer vollwertig-Marktfähigen Version, die in diesen Rahmen nicht entwickelt werden kann. Somit stellt diese Arbeit lediglich VerseVault als Prototypen dar, wohlwissend, dass das volle Potenzial des Konzeptes nicht implementiert sein wird.

2. Theoretischer Teil

In diesem Kapitel geht es um die einzelnen Technologien, die benötigt werden, um diese Arbeit ohne großes Vorwissen zu verstehen. So werden Mikroservices erklärt, was gRPC ist, was Reverse Proxies und Load Balancer sind und wie Kubernetes damit zusammenhängt. Dazu wird auch erklärt, was Continuous Integration und Continuous Delivery ist, wie Load Testing funktioniert, was Request-Coalescing ist und was Monitoring und Alarming ist.

2.1. Microservices

Die Mikroservice-Architektur bestimmt, dass die Anwendung aus vielen kleinen Services bestehen soll, die voneinander unabhängig sind und nur gering miteinander gekoppelt werden. Jeder der Services kann so seine eigenen Datenmanagementmodelle und Datenbankmodelle haben. Durch die geringe Kopplung wird nicht direkt, sondern über andere Wege, wie REST-APIs oder gRPC, untereinander kommuniziert (vgl. IBM o. D.). Das steht im Gegensatz zur Monolith-Architektur, bei denen die Dienste nicht in Services aufgeteilt werden, sondern als eine stark-verkoppelte Anwendung integriert sind (vgl. ebd).

2.2. gRPC

Das Akronym gRPC steht für „Google Remote Procedure Call“ und ist ein open source umgebungsunabhängiges Framework. Es wird benutzt, um effizient Services miteinander zu verbinden, wie zum Beispiel bei der Mikroservice-Architektur. Das bietet integrierte Unterstützung für beispielsweise Load Balancing, Tracing, Gesundheitschecks und Authentifizierung (vgl. gRPC Authors o. D.).

2.3. Load Balancer

Ein Load-Balancer ist dafür zuständig, die eingehenden Anfragen auf mehrere Server oder Netzwerkressourcen zu verteilen, damit eine optimale Ressourcenauslastung gewährleistet wird, um die Leistung zu verbessern und die Verfügbarkeit von Anwendungen oder Diensten zu erhöhen (vgl. Cloudflare o. D. a).

2.4. Reverse Proxy

Der Reverse-Proxy sorgt dafür, dass der Datenverkehr zwischen Clients und Servern in Computernetzwerken vermittelt wird. Reverse-Proxies spielen eine wichtige Rolle bei der Verbesserung der Sicherheit, Leistung und Skalierbarkeit von Netzwerken (vgl. Cloudflare o. D. b).

2.5. Kubernetes

Kubernetes, auch als K8s abgekürzt, stellt ein Open-Source-System dar, das darauf ausgerichtet ist, die Automatisierung von Bereitstellung, Skalierung und Verwaltung von containerisierten Anwendungen zu ermöglichen. Es organisiert die Container - die eine Anwendung bilden - in sinnvolle Einheiten, um eine effiziente Verwaltung und Identifikation zu erleichtern (vgl. The Kubernetes Authors o. D. a).

Es orchestriert die Infrastruktur für Computer, Netzwerk und Speicher im Auftrag der Benutzer-Workloads. Dabei kombiniert es die Benutzerfreundlichkeit von *Platform as a Service* (PaaS) mit der Anpassungsfähigkeit von *Infrastructure as a Service* (IaaS) und ermöglicht so die problemlose Übertragbarkeit zwischen verschiedenen Infrastrukturanbietern (vgl. The Kubernetes Authors o. D. b).

2.6. Continuous Integration

Continuous Integration (CI) ermöglicht die nahtlose Integration von Codeänderungen in ein gemeinsames Repository durch automatisierte Prozesse. Entwickler integrieren ihren Code mehrmals am Tag, wodurch Konflikte minimiert und die Codequalität kontinuierlich verbessert wird. Schlüsselmerkmale umfassen häufige Integration, automatisierte Builds nach jeder Codeänderung und automatisierte Tests, darunter Unit-Tests und Integrationstests. Diese Vorgehensweise gewährleistet eine stabile Codebasis, frühzeitige Fehlererkennung und eine schnelle Feedbackschleife für Entwickler. Eine standardisierte Entwicklungsumgebung fördert zusätzlich die konsistente Zusammenarbeit im Team (vgl. Amazon o. D.).

2.7. Continuous Delivery

Continuous Delivery (CD) automatisiert laut GitLab (o. D.) den Software-Bereitstellungsprozess, indem Änderungen automatisch getestet und veröffentlichungsfertig gemacht werden. Nach dem Einspielen von Änderungen erfolgt eine Überprüfung durch Continuous Integration, gefolgt von der Erstellung von Docker-Images bei erfolgreichen Tests. Die Hauptmerkmale umfassen automatisierte Bereitstellung, schnelle Softwareänderungen, Testautomatisierung für Funktionalität und eine konstante Umgebung. Frequente Releases ermöglichen kontinuierliche Veröffentlichungen, um schnell auf Kundenfeedback zu reagieren, und eine kollaborative Kultur fördert den Wissensaustausch für die Verbesserung von Softwarequalität und Bereitstellungsprozessen.

2.8. Lasttest

In einem Lasttest wird in einem System, welches getestet werden soll, eine Belastung erzeugt. Das Hauptziel besteht darin zu überprüfen, ob das System diese Belastung bewältigen kann. Dies könnte beispielsweise bedeuten zu prüfen, ob eine Webseite in akzeptabler Zeit Antworten an mehrere Besucher gleichzeitig senden kann oder ob ein Textverarbeitungsprogramm in der Lage ist, eine umfangreiche Datei zügig zu öffnen. Daher ist das Durchführen von Lasttests vor der Implementierung solcher Systeme von großer Bedeutung, da ein Ausfall dieser Systeme teilweise erheblichen Schaden oder finanzielle Verluste verursachen kann (vgl. Campero 2022).

2.9. Monitoring & Alarming (Prometheus + Grafana)

Um bei möglichen Problemen schnellstmöglich handeln zu können ist es ebenfalls wichtig das System jederzeit zu überwachen, und bei auftretenden Problemen eine Fehlermeldung zu erhalten. Für solche Aufgaben gibt es verschiedene Tools, wie zum Beispiel Grafana mithilfe von Prometheus und Node-Exporter, die eine manuelle Überwachung redundant machen, und frühzeitig Warnungen ausgeben, um so mögliche größere Probleme verhindern (vgl. Grafana Labs o. D. & Prometheus o. D.).

2.10. Request-Coalescing

Beim Request-Coalescing werden mehrere gleichzeitige Anfragen zu einer Anfrage zusammengefasst. So werden mehrere Anfragen von verschiedenen Verbindungen an denselben Pfad zusammengefasst, verarbeitet und dann an alle Verbindungen gleichzeitig zurückgeschickt (vgl. Bunny.net Support Hub o. D.). Dadurch wird eine Optimierung der Ressourcennutzung und Verbesserung der Reaktionszeiten erreicht. So lässt sich das „Thundering Herd Problem“, bei dem eine große Anzahl von Prozessen oder Threads, welche auf ein Ereignis warten, und bei Eintreten dieses Ereignisses um die verfügbaren Ressourcen konkurrieren, und dadurch das System einfrieren können, vermeiden (vgl. Anand 2023).

3. Architektur

Unsere Software benutzt viele der Technologien, die im theoretischen Teil vorgestellt wurden. In diesem Kapitel wird über die Implementation dieser Technologien berichtet. So wird zuerst die allgemeine Architektur dieser Anwendung dargestellt mit einem Fokus auf die verschiedenen Services im späteren Verlauf. Des Weiteren wird über das Testen und Monitoring der Architektur berichtet.

3.1. Software-Architektur

Die VerseVault Software wurde als Mikro-Service Architektur entwickelt. Jeder dieser Services stellt externe REST-Endpunkte bereit, die von Usern unserer Applikation genutzt werden. Wie in Abbildung 1 zu sehen, sind die Services voneinander getrennt, doch einige von ihnen besitzen interne Schnittstellen, die entweder über REST oder gRPC den anderen Services zur Verfügung stehen.

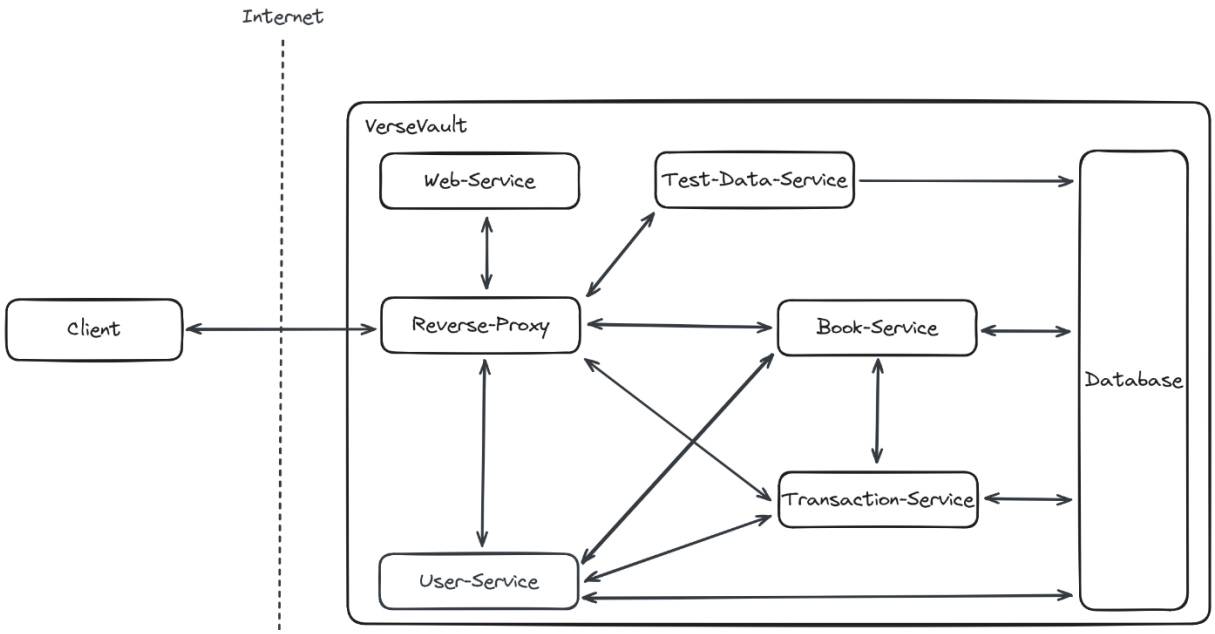


Abbildung 1: Mikroservice-Architektur der Anwendung

Der interne Aufbau der meisten Services ist in der Abbildung 2 dargestellt. Hierbei wird mit bis zu 3 Layern gearbeitet. Einer der Layer ist der Controller-Layer, welcher für die Verarbeitung der eingehenden Requests und das Aufbereiten der Informationen zuständig ist. Dazu kommt der Service-Layer, welcher unsere Business-Logik implementiert, jedoch in vielen Bereichen bereits im Controller-Layer implementiert wurde. Als letztes gibt es den Repository-Layer, welcher für die Kommunikation mit externen Systemen und Aufbereitung der Daten von und für diese zuständig ist.

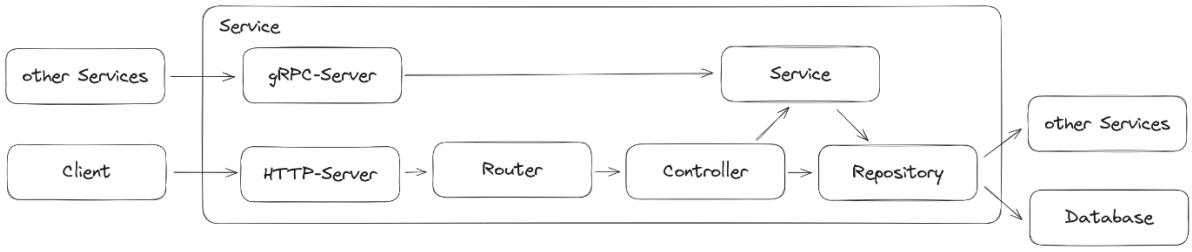


Abbildung 2: Aufbau eines generischen Services

Die Applikation kann nur von authentifizierten Benutzern genutzt werden. Hierbei wurde der folgende Auth-Flow entwickelt. Ein User kann sich einen neuen Benutzer über den Endpunkt *‚/api/v1/register*‘ erstellen. Anschließend kann er sich über den Endpunkt *‚/api/v1/login*‘ gegenüber der Software authentifizieren und erhält einen JWT-Access-Token im Body der Antwort, sowie einen JWT-Refresh-Token als HTTP-only Cookie. Der Access-Token muss bei jeder Anfrage an die Applikation im Authorization-Header mitgeschickt werden, um sich gegenüber den Services zu autorisieren. Ist der Access-Token nach einer vorher-konfigurierten Zeit abgelaufen, kann mit Hilfe des Refresh-Tokens im Cookie über den Endpunkt *‚/api/v1/refresh-token*‘ ein neuer Access-Token, sowie ein neuer Refresh-Token, erhalten werden. Über den Endpunkt *‚/api/v1/logout*‘ wird das Cookie mit dem Refresh-Token

entfernt. Zusätzlich können auch alle Tokens des Benutzers ungültig gemacht werden, indem der Query-Parameter ‚*all=true*‘ gesetzt wird.

Die Nutzer-Passwörter werden mit Bcrypt gehasht und dann in der Datenbank abgelegt. Die Access- und Refresh-Tokens werden durch ein RSA-Keypair signiert und verifiziert. Im JWT-Token ist eine Token-Version enthalten, die bei jeder Übertragung mit der in der Datenbank enthaltenen Token-Version überprüft wird, Stimmen diese nicht überein, wird der Zugang verwahrt. Die Version wird immer dann erhöht werden der Endpunkt ‚*/api/v1/logout*‘ mit dem Query-Parameter ‚*all=true*‘ aufgerufen wird.

Die Applikation beinhaltet einen selbst implementiertes Router-Package. Es werden Endpunkte - sowie die dazugehörigen Handler - registriert, welche dann, sobald die URL mit einem Endpunkt übereinstimmt, den zugehörigen Handler ausführen. Zusätzlich wurde das Konzept der Middleware implementiert. Hierbei lässt sich eine Funktion definieren, die für die konfigurierte und darauffolgenden Routen vor dem eigentlichen Handler ausgeführt wird. Hierbei erhält die Middleware-Funktion den Übergabe-Parameter *Next*. Wird diese Funktion nicht ausgeführt, werden auch alle folgenden Middlewares, sowie der kommende Handler, nicht ausgeführt. Somit lässt sich die Überprüfung des Access-Tokens aus dem Auth-Flow einfach in einer Middleware überprüfen, statt in jedem Handler selbst. In der Applikation werden diese Middlewares verwendet, um den Access-Token zu überprüfen und um bspw. Das Buch mit der ID zu laden, wenn die Ressource `*/api/v1/books/1*` angefragt wird, unabhängig davon, um welche Methode es sich handelt.

Es wurde ein Load-Balancer entwickelt, welcher zum aktuellen Zeitpunkt nicht mehr verwendet wird. Dieser diente dazu ein generelles Verständnis für die Funktionsweise eines Load-Balancers aufzubauen. Dafür wurden drei unterschiedliche Algorithmen für das Load-Balancing implementiert. Diese sind Round Robin, wobei die Anfragen in einer zirkulären Reihenfolge an jeden Server weitergeleitet werden. Least Connections, wobei neue Anfragen an den Server mit den wenigsten aktiven Verbindungen gesendet werden. Und IP-Hash, wobei eine Hash-Funktion basierend auf der IP-Adresse des Clients verwendet wird, um zu bestimmen, welcher Server die Anfrage bearbeiten wird.

Es wurde ein Reverse-Proxy entwickelt, welcher anhand einer bereitgestellten Konfigurationsdatei, verschiedene Anfragen an unterschiedliche Services weiterreicht. Dies wird benötigt, um unter anderen CORS-Problemen vorzubeugen. Weiterhin kann sich so der Service im Hintergrund ändern, ohne, dass der Client etwas mitbekommt. Dieser wird für die lokale Entwicklung weiterverwendet, aber in der Produktion durch den Nginx-Ingress-Controller in Kubernetes ersetzt.

Kubernetes ist ein Container-Orchestrator, mit denen anhand von einfachen Manifest-Dateien ganze Applikationen bereitgestellt werden können. Die eigens-implementierten Load-Balancer, Orchestrator, sowie Reverse-Proxy sind bereits in Kubernetes enthalten und müssen somit in diesem Kontext nicht mehr verwendet werden. Folgende Konzepte aus Kubernetes werden in unserer Applikation durch die

mitgelieferten Manifest-Dateien verwendet. Ein Pod ist die kleinste Einheit von Kubernetes und besteht aus einem oder mehreren Containern. Ein Deployment überwacht den Lifecycle aller definierten Pods und skaliert diese anhand der angegebenen Replikas automatisch. Stirbt ein Container, wird ein neuer gestartet, ohne das von außen etwas beigetragen werden muss. In Secrets und ConfigMaps werden die Konfigurationen der Applikation gespeichert, die beim Start eines Containers als Datei oder ins Environment geladen werden. Volumes und VolumeClaims definieren eine Ressource, in der die Pods Daten ablegen können, die selbst nach einem Neustart oder Start eines neuen Containers beibehalten werden. Kubernetes-Services sind ungefähr gleichzusetzen mit dem LoadBalancer, eine Sammlung von Pods mit dem gleichen Label können über die IP-Adresse des Services angesprochen werden und werden automatisch lastenverteilt. Beim Ingress handelt es sich wie zuvor erwähnt um den Reverse-Proxy. Hier werden Pfade und Domains angegeben, anhand denen eingehende Requests an die unterliegenden Services weitergeleitet werden.

In Bezug auf Continuous Integration und Continuous Delivery (CI/CD) wurde eine GitHub Workflow-Integration implementiert. Diese Automatisierung sieht vor, dass bei erfolgreichen Testläufen die Docker-Images automatisch auf GitHub hochgeladen werden. Diese Vorgehensweise ermöglicht eine effiziente und zuverlässige Bereitstellung von Softwareupdates, indem sie sicherstellt, dass nur getestete und funktionsfähige Versionen der Anwendung in den bereitgestellten Docker-Images enthalten sind.

3.2. Tests der Projektstruktur

Die auth-middleware hat auch *default_controller* und *http_repository* Mock-Tests, als auch integration-tests für das *grpc-repository*. Das Passwort-Hashing mit Bcrypt, der Health-Check und der Router werden auch getestet, jedoch als normale Unit-Tests ohne Mocks. Die Worker der Benchmark, der Reverse Proxy und der Load Balancer werden auch mit Mocks getestet, während die einzelnen Strategien des Load Balancers mit Unit-Tests ohne überprüft werden. Auf die Unit- und Integration Tests der Services wird im Kapitel 3.4 noch einmal etwas genauer eingegangen.

3.3. Services

Es wird die Mikroservice Architektur verwendet, weshalb der Programmcode in viele einzelne Services aufgeteilt ist. Nicht jeder der konzipierten Services hat es in die derzeitige Version geschafft. So sind zur Zeit dieser Arbeit fünf Services implementiert. Die nicht implementieren Funktionen, welche in der Einleitung erwähnt wurden, werden in Kapitel fünf behandelt.

3.3.1. Grundlegende Service-Struktur

Die meisten der implementierten Services sind grundlegend gleich strukturiert. Das Verzeichnis ‚*_mocks‘* beinhaltet die Mock-Dateien, welche von diversen Tests des Services verwendet werden. Unter ‚api/router‘ befinden sich die vom Service bereitgestellten REST-Endpunkte. Jeder Router stellt den Endpunkt ‚GET(„/health“)‘ bereit zur Überprüfung des Gesundheitszustandes der Service-Instanz.

Die Anfragen, welche an den Router gehen, werden zum Controller gesendet, welcher die Anfragen an das Repository weiterleitet und eine *http.response* aus der Antwort des Repositories fertigt. Für die Antwort macht das Repository eine Interaktion mit der Datenbank, so beispielsweise das Hinzufügen eines neuen Nutzers.

Das Verzeichnis ‚*grpc*' hat die Implementation des gRPC-Servers und ist ausschließlich für die interne Kommunikation zwischen den Services. Andere Services, die mit diesem gRPC-Server kommunizieren wollen, haben für den jeweiligen Service ein ‚*[serviceName]-service-client*' Verzeichnis. So hat z.B. der Book-Service ein Verzeichnis namens ‚*transaction-service-client*', welcher benutzt wird, um die Transaktionsdaten zu bekommen, um zu verifizieren, dass der Nutzer wirklich das zu ladende Kapitel besitzt.

Das Verzeichnis ‚*model*' beinhalten structs für die Daten in der Datenbank, meist mit verschiedenen Varianten für verschiedene Anfragen. Die Model-structs werden im Controller und im Repository benutzt. So sind structs zum Beispiel ein DbUser mit allen Daten und eine UserDTO struct, welches diese Informationen bloß ohne Passwort verwendet. Dazugehörig ist das Verzeichnis ‚*service*' die Ebene zwischen Controller und Repository, damit die REST- und gRPC-Anfragen, also externe und interne Anfragen, die gleiche business-Logik verwenden können.

Dazu besitzen die Services eine Dockerfile für das Docker-Image, eine go.mod mit go.sum für die Abhängigkeiten, eine main.go zum Starten der Service-Instanz und eine readme.md Datei mit einer Kurzbeschreibung und weitere Informationen über den Service.

3.3.2. User-Service

Der User-Service ist in dem momentan implementieren Zustand gleichzusetzen mit einem Authentication-Service, da er sich um das allgemeine Nutzermanagement kümmert, welches hauptsächlich die Authentifizierung des Nutzers bedeutet. Der Service besteht aus vielen verschiedenen Verzeichnissen, wovon das meiste die grundlegende Service-Struktur ist. Zusätzlich dazu ist das Verzeichnis ‚*auth*' und kümmert sich um die Konfiguration und dem Umgang mit JWT-Tokens, welche für die User-Autorisierung bei gesicherten externen REST-Endpunkten der Services benötigt werden.

Tabelle 1: Datenbankaufbau der "users"-Tabelle mit blau-markierten Primärschlüssel

Feldname	**Datentyp**	**Besondere Eigenschaften**	
id	serial	primary key	
email	varchar(100)	not null	unique
password	byte	not null	
profile_name	varchar(100)	not null	
balance	int	not null	default 0
token_version	bigint	not null	default 0

Den Aufbau der Datenbanktabelle ist der Tabelle 1 zu entnehmen. Die E-Mail-Adresse und das Passwort sind die Daten, welche zum Einloggen benötigt werden. Beim Registrieren wird außerdem der Profilname angegeben. Das Feld ‚*balance*' beschreibt die aktuelle Menge an VV-Coins, die der Nutzer

besitzt und die Token-Version wird im Auth-Flow verwendet, um einen Benutzer an allen Geräten auszuloggen, indem die Token-Version erhöht wird und die Tokens somit ungültig gemacht werden. Die Nutzerprofil-Eigenschaften sind nicht implementiert, nur der Profilname bleibt als nach-außenstehende Information über den Nutzer implementiert.

Die interne Kommunikation mittels ‚*grpc*' wird zur Validierung des Tokens und zum Verändern des VV-Coin-Kontostandes verwendet. Der User-Service bietet auch verschiedenste /api Endpunkte, so die POST-Endpunkte zum Einloggen, Ausloggen und Registrieren und auch fürs Auffrischen des Tokens. Die Authentication-Middleware wird für die auslog-Anfrage verwendet, da es möglich ist bei der Anfrage anzugeben, sich auf allen Geräten gleichzeitig auszuloggen. Für alle folgenden Anfragen wird auch die Authentication-Middleware verwendet, da Nutzerspezifische Daten gehandhabt werden. So lassen sich eingeloggt auch mit GET alle Nutzer, seine eigenen Nutzerdaten und die Nutzerdaten einer bestimmten id anfragen. Der eigene Nutzer kann dazu auch PATCH und DELETE, also aktualisiert oder gelöscht, werden.

3.3.3. Book-Service

Der Book-Service dient zur Bereitstellung von Büchern *(eng. Books)* und deren Kapitel *(eng. Chapter)*. Das macht den Service zum zentralen Service der Anwendung.

Tabelle 2: Datenbankaufbau der "books"-Tabelle mit blau-markierten Primärschlüssel

Feldname	**Datentyp**	**Besondere Eigenschaften**	
id	serial	primary key	
name	varchar(100)	not null	
authorid	int	not null	foreign key users (id)
description	text	not null	

Die Datenbankstruktur von Büchern ist wie in Tabelle 2 zu sehen mit vier Feldern klein gehalten. Ein Buch benutzt den Primärschlüssel id. Das Feld ‚name' speichert den Buchtitel, während das ‚description'-Feld die Beschreibung des Buches speichert. Zusätzlich wird der Autor im Feld ‚authorid' gespeichert. Die Kapitel eines Buches werden in einer separaten Tabelle gespeichert und besitzen eine Referenz zum Primärschlüssel, um sie zuordbar zu machen.

Geplant war es den Umgang mit den Kapiteln in einen eigenen Service zu legen, doch im Verlauf der Entwicklung hat der Book-Service den Chapter-Service vollständig assimiliert, da eine zu große Abhängigkeit herrscht. Chapter sind vollständig abhängig von Books und können ohne nicht existieren.

Tabelle 3: Datenbankaufbau der "chapters"-Tabelle mit blau-markierten Primärschlüssel

Feldname	**Datentyp**	**Besondere Eigenschaften**		
id	int	not null		primary key
bookId	int	not null	foreign key book (id)	
name	varchar(100)	not null		
price	int	not null		
content	text	not null		
status	int	not null	default 0	

Das spiegelt sich auch in der Kapitel-Tabelle, welche in Tabelle 3 zu sehen ist wider, bei dem der Primärschlüssel eines Kapitels aus der id des Kapitels und der id des Buches (bookId) besteht. Das ist so durch die Änderung, dass die Kapitel ids nur innerhalb eines Buches hochzählen. Ohne diese Änderung, hätte z.B. das fünfte Buch als erstes Kapitel die id 53, also */books/5/chapters/53*. Mit der Änderung startet jedoch jedes Buch mit */1*, was es einfacher macht, die Kapitel durchzugehen und unnötig große Zahlen zu vermeiden. Eine andere Lösung wäre zum Beispiel das Differenzieren von unique chapterId und der öffentlichen buch-intern-inkrementierenden chapterId, so z.B. chapterId = 53, bookChapterId = 1. Das führt aber zu mehr Speicherverbrauch und unklaren Referenzierungen, weshalb nur eine id für Kapitel benutzt wird, welche mit der bookId als Primärschlüssel dient und somit auch unique ist.

Der Book-Service hat wie der User-Service die grundlegende Service-Struktur. So existieren hier auch die Verzeichnisse ‚*_mocks*', ‚*api/router*', ‚*grpc*' und ‚*service*' und die Dateien Dockerfile, go.mod, go.sum, main.go und README.md. Die Verzeichnisse für Controller, Model und Repository sind für die Bücher und Kapitel getrennt und liegen in den Verzeichnissen ‚*books*' und ‚*chapters*', spezialisiert auf die jeweiligen Daten.

Zusätzlich existiert noch ein ‚*transaction-service-client*'-Verzeichnis, welches für die Überprüfung an den Transaction-Service zuständig ist, ob das Kapitel auch vom Nutzer gekauft wurde.

Es können im Frontend keine Bücher oder Kapitel gelöscht werden, da diese gekauft werden können und somit nicht von gekauften Nutzern genommen werden sollen. Auch Probleme mit der chapterId könnten auftreten, weshalb auch Drafts nicht gelöscht werden können, obwohl diese noch nicht gekauft sind.

Die interne Kommunikation mittels ‚*grpc*' wird zur Validierung der Kapitel-Id genutzt. Der Book-Service bietet wie die anderen Services auch */api* Endpunkte. Alle Anfragen werden zuerst mit der Authentication-Middleware geprüft, da nur eingeloggte Nutzer auf diese Endpunkte zugreifen dürfen. So lässt sich ‚*/api/v1/books*' verwenden, um über GET alle Bücher ausgeben zu lassen. Über POST kann es hingegen verwendet werden, um ein Buch hochzuladen. Bei Anfragen, die ein spezifisches Buch beinhalten, wird die LoadBook-Middleware verwendet, die das Buch zur Anfrage zulädt. Das geschieht zum Beispiel bei Anfragen an ‚*/api/v1/books/:bookid*', welche vom Typ GET, PATCH und DELETE sein können. Fügt man daran noch ‚*/chapters*' hinzu, kann man sich entweder dessen Kapitel mit GET holen, oder ein neues Chapter mit POST hochladen. Falls zum spezifischen Buch noch ein spezifisches

Kapitel geladen werden soll, wird die LoadChapter-Middleware verwendet, um das für diese Anfragen zu machen. Das trifft zu für die Anfragen an ‚*/api/v1/books/:bookid/chapters/:chapterid*', welche so wie bei den Büchern von Typ GET, PATCH und DELETE sein können.

3.3.4. Transaction-Service

Der Transaction-Service kümmert sich um das Handhaben der Kapitel-Transaktionen und speichert, wer welche Kapitel besitzt. Wie auch bei User- und Book-Service gibt es die gleichen Verzeichnisse und grundlegende Struktur. Wie auch beim Book-Service werden Service-Clients verwendet, ein book-service-client und ein user-service-client.

Tabelle 4: Datenbankaufbau der "transactions"-Tabelle mit blau-markierten Primärschlüssel

Feldname	**Datentyp**	**Besondere Eigenschaften**	
id	serial	primary key	
bookid	int	not null	foreign key chapters (id, bookId)
chapterid	int	not null	
receivinguserid	int	not null	foreign key users (id)
payinguserid	int	not null	foreign key users (id)
amount	int	not null	default 0

Bei einer Transaktion wird ein Eintrag in der Transaction-Tabelle erstellt, welche in Tabelle 4 dargestellt wird. Es wird gespeichert, welches Buch und welches Kapitel davon gekauft wurde. Dazu wird gespeichert, welcher Nutzer das Kapitel gekauft hat und welcher Nutzer das Geld für das Kapitel bekommt, also der Autor des Kapitels. Da der Preis des Kapitels geändert werden kann, wird auch die Summe an bezahlten VV-Coins mitgespeichert. Anhand des Eintrages kann festgestellt werden, dass der bezahlende Nutzer das Kapitel besitzt.

An den Transaction-Service kann mittels gRPC überprüft werden, ob das Kapitel vom Nutzer schon gekauft wurde. Bei den API-Endpunkten wird mit der Authentication-Middleware, so wie auch bei den anderen nutzerbezogenen Daten, überprüft, ob der Nutzer eingeloggt ist. So kann der Transaction-Service nur die nutzerbezogenen Transaktionen durch GET ausgeben, oder eine neue Transaktion über eine POST-Anfrage erstellen.

3.3.5. Web-Service

Der Web-Service stellt das Frontend zur Verfügung, und anders als die vorherigen Services werden dort keine Daten verwaltet. Das Frontend wurde mittels React entwickelt, und nutzt Komponenten von ‚*shdcn UI*'. Außerdem wird das CSS-Framework Tailwind genutzt, um den Entwicklungsprozess zu beschleunigen und den Code übersichtlicher zu halten.

In dem Verzeichnis ‚*src*' befinden sich im Unterverzeichnis ‚*components*' neben den genutzten Komponenten von ‚*shdcn UI*' noch zwei weitere, welche dazu dienen die Datenbank für Testzwecke zurückzusetzen und um zwischen dem hellen und dem dunklen Design zu wechseln. Im Unterverzeichnis ‚*layouts*' sind zum einen die Datei *‚main-layout.tsx'* zu finden, welche die

Navigationsleiste der Website beinhaltet und die Datei ‚*root-layout.tsx'*, welches das allgemeine Layout der Seite beschreibt. Das ‚*lib*'-Verzeichnis beinhaltet zum einen den ‚*api-client.ts*', welcher zum Erstellen eines Axios-API-Clients dient, der eine Token-basierte Authentifizierung unterstützt, und ‚*utils.ts*' welche zum Zusammensetzen von ‚*ClassValues*' verwendet wird. Unter ‚*provider*' sind verschiedene Dienste zu finden, welche als Unterstützung dienen, und eine React-Kontext-Provider-Komponente implementieren. Im Verzeichnis ‚*repository*' werden die Funktionen zur Übermittlung an das Backend bereitgestellt. Im Unterverzeichnis ‚*routes*' sind die tsx-Dateien, welche den Code der React-Komponenten definieren, die in der Webanwendung dargestellt werden. Und im ‚*types*'-Verzeichnis sind die verschiedene Data Transfer Objects (DTO) enthalten. In der Datei ‚*main.tsx*' sind die Routen zu den einzelnen Seiten definiert und sie ist der Startpunkt der Website.

Das Frontend ist so aufgebaut, dass beim Aufrufen der Seite die Homepage geladen wird, falls der Nutzer jedoch nicht eingeloggt ist, wird andernfalls die Login-Seite angezeigt, auf der neben dem Formular zum Anmelden auch ein Knopf ist, um zu der Registrierungsseite zu wechseln. Wenn der Nutzer dann angemeldet ist, sieht er die Homepage, auf der eine Liste der zuletzt editierten Bücher angezeigt wird. In der Navigationsleiste gibt es Knöpfe um zu den verschiedenen Seiten zu navigieren. Im rechten Teil der Navigationsleiste wird der Name des angemeldeten Nutzers und seine aktuelle Anzahl an ‚*VV-Coins*' angezeigt und es besteht die Möglichkeit zwischen dem hellen und dem dunklen Design zu wechseln. Wird hier auf die Benutzer, oder auf die ‚*VV-Coins*' gedrückt wird ein Dropdown-Menü angezeigt, in dem die ‚*VV-Coins*' verwaltet werden können, wobei diese entweder auf- oder abgeladen werden können. Außerdem kann sich der Nutzer hier abmelden. Der Nagivationspunkt ‚*All Books*' zeigt die Startseite, mit der bereits erwähnten Liste an zuletzt editierten Büchern. Unter ‚*My Books*' sind die geschriebenen Bücher des angemeldeten Nutzers zu finden, und es besteht die Möglichkeit ein neues Buch zu erstellen. Wenn bereits ein Buch erstellt wurde, kann dieses angeklickt werden, und es werden die existierenden Kapitel angezeigt. Hier bestehen die Möglichkeiten die Informationen des Buches zu editieren, ein neues Kapitel zu erstellen, oder bei möglichen vorhandenen Kapiteln Änderungen vorzunehmen, oder diese zu lesen. Wenn ein neues Kapitel erstellt wurde, befindet sich dieses erst im ‚*Draft*'-Zustand. In diesem Zustand ist es für andere Nutzer nicht zu sehen und kann vom Autor weiterbearbeitet werden, bis es fertig ist und dann veröffentlicht wird. Auf der Seite ‚*My bought Books*' werden die bereits gekauften Bücher des Nutzers angezeigt. Wenn eines ausgewählt wird, werden die Buchinformationen dargestellt, und die Kapitel des Buches angezeigt. Hier können dann entweder die bereits gekauften Kapitel gelesen werden, oder neue Kapitel gekauft werden. Wenn ein Kapitel geöffnet wird, besteht über dem Text die Auswahl zum Buch zurückzukehren oder zum nächsten oder vorherigen Kapitel zu gehen. Wenn das nächste bzw. vorherige Kapitel noch nicht gekauft wurde, kann es so auch gekauft werden. Unter ‚*My Transactions*' ist eine Auflistung der getätigten Käufe unter dem Tab ‚*Bought Chapters*' zu finden, oder unter ‚*Earnings*' die Anzeige, welche selbst geschriebenen Kapitel von anderen Nutzern gekauft wurden. Der Knopf ‚*Load/Reset Test Data*' ist nur zu diesem Stand

des Projektes vorhanden, um das Testen der Anwendung zu erleichtern und wird mit dem dazugehörigen Service niemals für den Endnutzer zugänglich sein.

3.3.6. Test-Data-Service

Der Test-Data-Service ist ausschließlich dazu da, die Datenbank auf schnellem Wege bei dem Testen mit dem Frontend zurückzusetzen. Der Test-Data-Service benutzt die grundlegende Service-Struktur, jedoch ohne ‚service' und ‚grpc' da keine Kommunikation mit anderen Services herrscht.

3.4. Servicetests

Im folgenden Abschnitt werden die verschiedenen Tests, welche in der Anwendung implementiert sind, erläutert. Die meisten der implementierten Tests verwenden die Testbibliothek "github.com/stretchr/testify/assert" zum Überprüfen der Testergebnisse und den Mock-Controller von "github.com/golang/mock/gomock" zur Bereitstellung der Testumgebung. Das übergeordnete Ziel besteht darin, sicherzustellen, dass die Services angemessen auf diverse Szenarien reagieren und korrekte Ergebnisse liefern.

Viele Tests sind Unit-Tests, welche mit Mocks umgesetzt wurden. So wird bei den Services der *router*, der *default_controller*, das *psql_repository* und der *default_service* mit Mocks getestet. Auch die Komponenten des *Auth*-Verzeichnisses des User-Service oder die *http_repository*-Tests der gRPC-Service-Clients werden mit Mocks Unit-getestet. Die *psql_repository* der einzelnen Services werden auch integrationsgetestet.

3.5. Monitoring

Das System wird standartmäßig auf Fehler wie ‚*High CPU*', ‚*High Memory*' und außerdem wird der Netzwerkverkehr überwacht, damit es nicht zu größeren Problemen kommt. Dazu werden Prometheus und Grafana auf dem Kubernetes-Cluster aufgespielt. In Prometheus werden die Metriken gespeichert, welche von Node-Exporter bereitgestellt werden. Diese Daten werden dann über Grafana visuell angezeigt.

4. Testaufbau und -ergebnisse

In diesem Kapitel geht es um das Testen verschiedener Hypothesen und deren Ergebnisse. So werden die folgenden Hypothesen beleuchtet:

H_1: Request Coalescing verringert in jeder Konfiguration des Systems die Ressourcenauslastung im Vergleich zu ohne.
H_0: Request Coalescing verringert in keiner Konfiguration des Systems die Ressourcenauslastung im Vergleich zu ohne.

H_2: Mehr Replikas des Web-Service sorgt für eine schnellere Abarbeitung der Anfragen.
H_0: Mehr Replikas des Web-Service sorgen für keine schnellere Abarbeitung der Anfragen.

H_3: gRPC eignet sich besser als REST für die Kommunikation zwischen den Services.
H_0: gRPC eignet sich nicht besser als REST für die Kommunikation zwischen den Services.

4.1. Systemaufbau

Um das System testen zu können, wurde ein Kubernetes-Cluster auf einem Proxmox-Host aufgebaut, bestehend aus insgesamt fünf virtuellen Maschinen. Das Cluster setzt sich zusammen aus einer Control Plane und vier Worker-Nodes. Die Control Plane ist mit 4 GB RAM und 4 CPU-Kernen ausgestattet, während jeder Worker mit 2 GB RAM und 2 CPU-Kernen konfiguriert ist. Zur kontinuierlichen Überwachung der Systemressourcen wie CPU-Auslastung und RAM-Verfügbarkeit wurde der Node-Exporter in Verbindung mit Prometheus implementiert. Die gesammelten Daten werden durch ein Grafana-Dashboard visualisiert, was einen leicht-verständlichen Überblick über die Cluster-Performance bietet. Dieser visuelle Ansatz erleichtert die Identifizierung von Engpässen oder ungewöhnlichen Verhaltensweisen, und ermöglicht eine effektive Fehlerbehebung und Ressourcenoptimierung. Es ist zu beachten, dass die Datenbank innerhalb des Clusters derzeit kein explizites Ressourcenlimit definiert hat. Dies eröffnet die Möglichkeit, die Auslastung der Datenbankressourcen zu überwachen und gegebenenfalls Anpassungen vorzunehmen, um eine optimale Leistung sicherzustellen.

4.2. Coalescing und Replika Tests

Dieser Abschnitt widmet sich der Analyse von Coalescing und Replika-Tests, um festzustellen, wieviel Replikas für dieses System am geeignetsten sind, und ob die Benutzung von Request Coalescing gegenüber dem nicht-Benutzen vorteilhaft ist. Vorrangehend wird H_1 angenommen, dass Coalescing in jeder Konfiguration des Systems besser mit den verfügbaren Ressourcen funktioniert, als wenn drauf verzichtet werden würde.

4.2.1. Testaufbau

Bei den Coalescing- und Replika-Tests ist die Datenbank auf Worker-Node 2, während die Book-Services auf Worker-Node 1, 3 und 4 verteilt wurden. Diese Verteilung ermöglichte die Untersuchung von Interaktionen zwischen den Servicekomponenten und der Datenbank über das Netzwerk. Außerdem

wurde die Authentifizierung für diese Tests deaktiviert. Dies ermöglichte eine dedizierte Analyse der Auswirkungen von Request-Coalescing in Verbindung mit der Skalierung von Replikas auf die Leistung, insbesondere bei einer beträchtlichen Datenmenge. Zusätzlich zu den vorhandenen Testdaten von drei Büchern wurden in der Datenbank 100 weitere Bücher mit je 50 Worten aus "*Lorem Ipsum*" in der Beschreibung und dem Namen "*The Name of the book(number)*" erstellt. Dies diente dazu, realistischere und umfangreichere Szenarien zu simulieren. Der Fokus des Experiments lag auf dem Coalescing von Anfragen zwischen den Services und der Datenbank. Dieser Prozess wurde analysiert, um die Effizienz und die Auswirkungen auf die Datenübertragung zu bewerten. Bei einer einzelnen Anfrage wurden insgesamt 103 Bücher zurückgegeben, was zu einer Übertragung von etwa 37 kB an Daten pro Anfrage führte. Dies lieferte wichtige Einblicke in die Übertragungseffizienz und den Ressourcenverbrauch. Um präzise Metriken zu erhalten, wurden umfangreiche Tests mithilfe von Apache Benchmark durchgeführt.

4.2.2. Testdurchführung

Der Befehl "*ab -n 50000 -c 1000 http://vv.hsfl.de:32131/api/v1/books*" wurde verwendet, um 50.000 Anfragen mit einer Parallelität von 1000 gleichzeitigen Anfragen durchzuführen. Die Ergebnisse dieser Testdurchführung sind auch im Anhang als grafische Darstellung in Grafana zu finden.

Tabelle 5: Metriken der Request-Coalescing Tests

	1 Replica without Coalescing	1 Replica with Coalescing	2 Replicas without Coalescing	2 Replicas with Coalescing	3 Replicas without Coalescing	3 Replicas with Coalescing	10 Replicas with Coalescing
Time taken for tests	58,77s	30,33s	39,42s	26,58s	39,51s	30,75s	39,55s
Connection Time total median	410ms	535ms	207ms	433ms	685ms	448ms	538ms
95% Requests within time	4.166ms	1.061ms	3.326ms	1.010ms	1.597ms	1.429ms	1.478ms
Requests per Second	850,76/s	1.648,34/s	1.268,42/s	1.880,95/s	1.265,67/s	1.626,12/s	1.264,29/s
Time per Request	1.175,42ms	606,00ms	788,38ms	531,65ms	790,09ms	614,96ms	790,96ms
DB Worker Node CPU-Util	25,10%	8,32%	39,80%	10,00%	46,70%	13,30%	39,5%

Wie der Tabelle 5 zu entnehmen, ist die CPU-Auslastung der Datenbank-Worker-Node mit einer Replika ohne Request Coalescing bei 25,10%, während es mit bei 8,32% liegt. Durch Request Coalescing sinkt die Auslastung auf circa. ein-drittel der Auslastung ohne Request Coalescing. Auch bei den 95% der schnellsten Anfragen, ist es mit Request Coalescing um das knapp vierfache schneller. Dies deutet bereits darauf hin, dass die Benutzung von Request Coalescing zu einer erheblichen Reduzierung der CPU-Belastung führt. Im direkten Vergleich, insbesondere bei der Verwendung von bis zu drei Replikas, zeigt sich, dass Coalescing stets überlegen ist, insbesondere in Bezug auf die CPU-Auslastung der Datenbank.

Eine interessante Beobachtung ist, dass bei deaktiviertem Coalescing die Nutzung von mehr Replikas sinnvoll ist, um die Anzahl der Anfragen pro Sekunde zu erhöhen. Allerdings zeigt sich mit Coalescing zwischen zwei und drei Replikas ein Rückgang dieser Leistung. Diese Beobachtung lässt sich damit erklären, dass bei aktiviertem Coalescing weniger Anfragen pro Server fusioniert werden können, da die Anfragen auf alle drei Dienste aufgeteilt sind. Somit wird die Datenbank stärker belastet.

Zusammenfassend unterstützen die Ergebnisse die Hypothese, dass die Benutzung von Request Coalescing gegenüber dem nicht-Benutzen vorteilhaft ist. Darüber hinaus widerlegen die Ergebnisse die Hypothese, dass mehr Replikas immer zu einer höheren Anzahl von Anfragen pro Sekunde führen. Es wurde festgestellt, dass es bestimmte Schwellenwerte gibt, bei denen eine Zunahme der Replikas zu einem Rückgang der Effektivität führt, da die Lösung durch mehr Replikas weniger effizient wird. Dies wurde weiter durch einen zusätzlichen Test mit 10 Replikas bestätigt, der ebenfalls eine schlechtere Zeitersparnis im Vergleich zu zwei Replikas zeigte.

4.3. Web-Service Replika Test

In diesem Test wird die Hypothese H_2 überprüft, dass mehr Replikas dabei helfen, die Abfragen schneller zu bearbeiten. Dazu wurde die Replika-Menge bei gleicher Anfragenmenge erhöht.

Bei der Testdurchführung hat sich des Web-Service Replika Tests gezeigt, dass unabhängig von der gewählten Anzahl an Replikas keine signifikanten Veränderungen an den Requests-Per-Second von etwa 4000 festgestellt wurden. Deshalb wurde entschieden, zwei Replikas zu erstellen, um sicher zu stellen, dass auch wenn einer ausfällt, die Anwendung weiter funktioniert. Dieser Ansatz bietet eine ausgewogene Lösung zwischen Ressourcennutzung und Gewährleistung der Anwendungsverfügbarkeit im Falle eines Ausfalls.

4.4. gRPC vs. REST für zwischen-Service Kommunikation

In diesem Abschnitt wird getestet, ob sich entweder gRPC oder REST besser für die Kommunikation der Services untereinander eignet. Dabei gehen wir von unserer Hypothese aus, dass gRPC sich besser eignet als REST für die Kommunikation zwischen den Services.

4.4.1. Testaufbau

Bei diesem Test ist die Datenbank wieder auf der Worker-Node 2 und der Ingress-Controller auf Node 1. Dazu laufen User-Services auf den Worker-Nodes 3 und 4 und Book-Services auf Node 1 und 3. Bei diesem Testaufbau ist zusätzlich die Authentifizierung aktiv. In diesem Test wird der Endpunkt ‚*/api/v1/books/a*‘ getestet, wobei "a" als ungültige Buch-ID betrachtet wird. Infolgedessen reagiert der "Book-Service" mit einem Statuscode 400, da die ID nicht valide ist. In diesem Szenario finden keinerlei Datenbankzugriffe durch den “Book-Service” stattfinden. Stattdessen wird lediglich die Authentifizierungsmiddleware aktiviert, und im Authentifizierungsprozess überträgt der "Book-

Service" über gRPC oder REST den Access-Token an den "User-Service". Hierbei wartet die Authentifizierungsmiddleware auf das Ergebnis der Token-Validierung.

4.4.2 Testdurchführung

Der Befehl ‚*ab -n 50000 -c 1000 -H "Authorization:Bearer $TOKEN" http://vv.hsfl.de:32131/api/v1/books/a*‘ wurde verwendet, um 50.000 Anfragen mit einer Parallelität von 1000 gleichzeitigen Anfragen durchzuführen. Es werden mehrere Durchläufe durchgeführt, wo zwischen Replika-Menge, Token-Zustand und Protokoll gewechselt wurde. Bei den Tests, wo der Token invalide ist, wurden die doppelte Menge an Anfragen, 100.000, gesendet.

Tabelle 6: Vergleich von gRPC und REST Anhand von Book-Service und User-Service mit einem validen oder invaliden Token

Book-Service Replicas	1	1	2	2	1	1	2	2
User-Service Replicas	1	1	2	2	1	1	2	2
Token-State	valid	invalid	valid	invalid	valid	invalid	valid	invalid
Protocol	gRPC	gRPC	gRPC	gRPC	REST	REST	REST	REST
Time taken for tests	42,63s	30,626s for double req	42,349s	31,883s for double req	52,028s	**OOM**	33,868s	**OOM**
Connection Time total median	360ms	292ms	331ms	297ms	466ms	**OOM**	222ms	**OOM**
95% Requests within time	3.024ms	635ms	2.975ms	665ms	3.584ms	**OOM**	2.616ms	**OOM**
Requests per Second	1.172,99/s	3.265,23/s	1.180,67/s	3.136,48/s	961,02/s	**OOM**	1.476,31/s	**OOM**
Time per Request	852,526ms	306,257ms	846,978ms	318,829ms	1.040,558ms	**OOM**	677,363ms	**OOM**

Bei der Analyse wurde festgestellt, dass die Setzung eines ungültigen Tokens im Authorization-Header zu erheblichen Problemen führt. In einem Szenario, in dem der Token ungültig ist, führt dies zu einem signifikanten Anstieg des Speicherverbrauchs bei REST-Anfragen. Diese unkontrollierte Erhöhung der Speichernutzung führt dazu, dass Dienste aufgrund von zu hohem Arbeitsspeicher-Verbruach durch Kubernetes gewaltsam beendet werden. Dieser Vorfall zeigt, dass die Handhabung ungültiger Tokens entscheidend für die Stabilität und Leistung der verteilten Dienste ist. Eine weitere Erkenntnis dieser Untersuchung ist, dass das Fehlen von Limits für die Anzahl der ungültigen Token-Anfragen zu schwerwiegenden Problemen führt. In einem betrachteten Fall führte das Fehlen eines Limits dazu, dass der Worker-Node selbst aufgrund unkontrollierter Ressourcennutzung ausfiel. Dies verdeutlicht die Notwendigkeit einer effektiven Überwachung und Begrenzung von Token-Anfragen, um die Stabilität des Gesamtsystems zu gewährleisten. Die Untersuchung ergab, dass bei Verwendung gültiger Tokens die Leistung bei REST-Anfragen mit zwei Replikationen verbessert wird. Interessanterweise zeigte sich bei gRPC-Anfragen keine signifikante Veränderung in der Leistung, unabhängig von der Anzahl der Replikationen. Dies deutet darauf hin, dass die Auswirkungen von Token-Gültigkeit und Replikation auf verschiedene Kommunikationsprotokolle unterschiedlich sind.

4.5. Evaluation der Ergebnisse

Aus den Testergebnissen geht hervor, dass bei diesem System zwei Replikas mit aktiviertem Coalescing für die Services Book, User, Transaction und Web am geeignetsten sind. Die betroffenen Endpunkte sind hierbei ‚*GetUser*', ‚*GetBooks*', ‚*GetChapterPreview*' und ‚*GetChapter*'. Die Hypothese H_1 kann in dieser Form nicht angenommen werden, da das Skalieren auf mehr Replikas zu schlechteren Ergebnissen führt. Aufgrund der Ergebnisse kann jedoch eine Abwandlung der Hypothese angenommen werden. Diese lautet wie folgt:

H_{1_1}: *Request Coalescing kann je nach Menge der Replikas die Ressourcenauslastung im Vergleich zu ohne verringern.*

Die veränderte Form der Hypothese H_1 - die Hypothese H_{1_1} - kann mittels der Ergebnisse angenommen werden. Ferner besagen die Resultate, dass eine höhere Anzahl von Replikas zu einer Verschlechterung der bearbeiteten Anfragen pro Sekunde führen kann.

Aus dieser Information ist H_2 entstanden, die besagt, dass mehr Replikas des Web-Service zu einer Verbesserung der bearbeiteten Anfragen pro Sekunde führt. Die Ergebnisse besagen jedoch, dass mehr Replikas weder zu einer Verbesserung noch zu einer Verschlechterung führen. H_2 wird somit abgelehnt.

Die Tests zeigen auch, dass entgegen der Hypothese H_3 bei Verwendung von zwei Replikas mit REST gegenüber gRPC eine schnellere Leistung aufweist. Dies könnte darauf zurückzuführen sein, dass möglicherweise beide Dienste jeweils mit demselben User-Service verbunden sind und die Verbindung offengehalten wird, ohne bei jeder Anfrage neu initialisiert zu werden. Die Analyse weist darauf hin, dass die Verwendung von REST bei der Coalescing-Strategie zwar zu schnelleren Ergebnissen führt, jedoch bei ungültigen Anfragen zu ernsthaften Out-of-Memory (OOM)-Problemen der Dienste führen kann. Dies stellt eine erhebliche Einschränkung dar, da ineffiziente Anfragen die Stabilität der Dienste gefährden. Deshalb wird gRPC für die interne Kommunikation zwischen den Diensten genutzt. Dies ermöglicht eine effiziente und zuverlässige Datenübertragung zwischen den Services, wodurch potenzielle OOM-Probleme vermieden werden können, die bei REST auftreten könnten. Dadurch zeigt sich, die aufgestellte Hypothese H_3, dass gRPC sich besser eignet als REST für die Kommunikation zwischen den Services, nur bedingt stimmt, da bei zwei Replikas und der Verwendung gültiger Tokens eine bessere Leistung von REST-Anfragen zu erkennen war, und bei gRPC-Anfragen unabhängig von der Anzahl der Replikationen keine signifikante Veränderung in der Leistung zu erkennen war und dadurch flexibler zu benutzen ist. Somit lässt sich die Hypothese H_3 in dieser Form nicht annehmen, da gRPC zwar in den meisten Fällen bessere Ergebnisse liefert - in Hinsicht auf Schnelligkeit und Zuverlässigkeit - jedoch auch Konfigurationen existieren, bei denen REST schneller antwortet.

4.6. Aufgetretene Probleme

Ein bedeutendes Problem, das während der Analyse auftrat, war das Auftreten von Out-of-Memory (OOM)-Ereignissen in den Services. Diese Ereignisse deuten darauf hin, dass die verfügbaren

Ressourcen für einen bestimmten Dienst erschöpft waren, was zu einem unerwarteten Abbruch des Dienstes führte, was wie bereits weiter oben beschrieben durch die Verwendung von gRPC gelöst wurde. Ein weiteres signifikantes Problem war das unerwartete Beenden von Worker-Nodes. Dieses Verhalten kann zu einer Beeinträchtigung der Skalierbarkeit und Leistungsfähigkeit des Gesamtsystems führen, was sich aber leicht durch eine Begrenzung der Container Ressourcen beheben lies. Ein weiterer Engpass, der identifiziert wurde, betrifft die Konfiguration der maximalen Verbindungen zur Datenbank. Insbesondere im Zusammenhang mit dem PostgreSQL-Datenbankmanagementsystem wurde festgestellt, dass die Anzahl der gleichzeitigen Verbindungen die vom System akzeptierte Obergrenze überschritten hat. Dies führte zu Beeinträchtigungen bei der Skalierbarkeit und Leistung der Datenbankzugriffe. Das Problem konnte durch die richtigen Konfigurationen ebenfalls schnell behoben werden.

5. Fazit und Ausblick

Das Ziel dieses Projektes war es, ein System zu entwickeln, welches eine Plattform bietet, auf der Autoren ihre Schriftwerke, für einen von ihnen gewählten Preis, anderen zur Verfügung stellen können. Das Verwenden der Microservice-Architektur hat sich als positive Entscheidung erwiesen, da durch diese Architektur das Projekt gut erweiterbar und wartbar ist.

Die Hypothese H_1 konnte in dieser Form nicht angenommen werden und wurde zur Hypothese H_{1_1} umgewandelt. Mit dem Ablehnen der Hypothese H_1 war zu rechnen, da allgemein-formulierte Hypothesen leicht auf Grenzfälle stoßen können, bei denen die Hypothesen nicht zutreffen. Dass die spezifischere Hypothese H_{1_1} angenommen wurde, ist verständlich, da Request Coalescing ein beliebter und Effizienter Weg ist, um Anfragen zu optimieren. Das Ablehnen der zweiten Hypothese H_2 ist entgegen unseren Erwartungen, aber nicht problematisch, da der Service keine Berechnungen und einzig zur Dateibereitstellung dient. Deshalb werden hier nicht viele Replikas benötigt. Das Ablehnen der dritten Hypothese H_3 ist wie H_2 unerwartet, da es vereinzelte Fälle gibt, bei denen das nicht zutrifft. Dennoch besteht die Aussage, dass im Durchschnitt sich GRPC besser eignet als REST, da es im Gegensatz zu REST binäre Daten überträgt und somit keine Konvertierung von und zu JSON macht und somit im Schnitt schneller sein sollte. Dazu kam es bei REST zu Out-of-Memory Problemen, welche bei gleichen Einstellungen mit gRPC nicht vorkamen, was für die Zuverlässigkeit von gRPC spricht.

Wie in der Einleitung aufgezählt wurden verschiedene Funktionen und somit auch dessen Services festgelegt, welche für eine Marktfähige Version von VerseVault benötigt werden. Aus den anfänglich 13 verschiedenen Services, wurden die vier wichtigsten implementiert. Diese umfassen den User-Service, welcher zum jetzigen Zeitpunkt nur einen Authentication-Service darstellt, den Book-Service, in dem die Verwaltung der einzelnen Bücher und Kapitel stattfindet, den Transaction-Service, welcher die einzelnen Zahlungen verwaltet, und den Web-Service, der für die visuelle Darstellung im Webbrowser zuständig ist. Zusätzlich zu den ursprünglich geplanten Services kam der Service Test-

Data-Service hinzu, welcher das Zurücksetzen der Testdaten während des manuellen Testens mit der Frontendoberfläche vereinfacht.

Zukünftige Änderungen an dem System könnten beinhalten, dass der User-Service weitere Funktionen erhält, wie vollwertige Nutzerprofile mit Namen, Profilbildern und Beschreibungen. Auch das Ändern dieser Daten, als auch das Passwort, gehören zum User-Service dazu. Dem würde dann vorhergehen, dass der User-Service wie er zum jetzigen Zeitpunkt ist, zum Authentifizierung-Service benannt wird, da die momentanen- und die geplanten Funktionen jeweils eigenständig agieren und nicht im selben Service enthalten sein müssen bzw. sollten.

Angeschlossen am Nutzerservice wäre ein Notifikations-Service eine interessante Erweiterung. Dieser Service ermöglicht einem Nutzer oder Bücher zu folgen, um Benachrichtigungen bei Updates zu erhalten.

Außerdem wäre es wichtig, die aktuelle Startseite besser zu gestalten. Das lässt sich mit Nutzerzugeschnittenem Inhalt leicht machen. So werden verschiedene Ranglisten an Büchern angezeigt, aus denen Nutzer leicht gute Bücher finden können. So zum Beispiel die allzeit-beliebten, die momentan beliebten und auch Werke, die ähnlich sind zu den zuvor gelesenen Kapiteln. Um die Beliebtheit festzustellen könnte man den Traffic analysieren, oder man lässt Nutzer mithilfe von Bewertungen und Kommentaren auf Kapiteln entscheiden. Auch Tags auf Büchern und Kapiteln können helfen, bessere Empfehlungen zu geben. Die zuvor erwähnte Suchfunktion, Kapitel-Download und Spendenfunktion bieten sich gleichermaßen für zukünftige Versionen an, um das Erlebnis zu verbessern.

Zudem ist das Zahlungssystem vollständig zu implementieren, bevor das System für die Öffentlichkeit zugänglich gemacht werden kann, da dieses aktuell nur als Platzhalter fungiert, um die einzelnen Funktionen zu testen. Nützliche Zahlungsmethoden sind Kredit- Debit- und Visakarten, Zahlung per Bankeinzug und PayPal. Zahlung per Gutscheinkarte wäre auch sinnvoll, benötigt jedoch einen weiteren Service.

Der implementiere Web-Service ist stark verbesserungsfähig. So sollte VerseVaults UI-Entwicklung den Ansatz „mobile-first“ folgen. Das Frontend ist nur rudimentär implementiert und folgt nicht den mobile-first Ansatz, da bei der Entwicklung keine Priorität auf die Frontend-Entwicklung gelegt werden sollte. Das Web-Interface dient lediglich als eine schnelle Lösung zur Darstellung und Testen der Anwendung.

Das Editieren der Kapitel ist eine nützliche implementiere Funktion, kann aber missbraucht werden. So könnte jemand, da Kapitel und Bücher nicht löschbar sind, den Inhalt von jedem Kapitel rauslöschen. Das würde den Nutzern, die das gekauft haben, die Möglichkeit nehmen, darauf zuzugreifen. Deshalb wäre es vorteilhaft eine Historie für Kapitel hinzuzufügen, um im schlimmsten Fall das Kapitel wiederherzustellen. So etwas sollte aber vor Veröffentlichung von VerseVault rechtlich abgeklärt werden und als Bruch der Nutzungsrichtlinien (Terms-of-Service) gelten. Auch ein Plagiatserkenner

und Filter für schädlichen und illegalen Inhalt sind essenziell, damit VerseVault rechtlich geschützt ist und auch einen besseres Nutzererlebnis bietet. Der rechtliche Aspekt des Projektes ist monumental und sollte für zukünftige Versionen gründlich professionell geprüft werden.

Weiterhin gibt es bereits Funktionen im Backend, die im Frontend noch nicht dargestellt werden. Dazu gehören unter anderem das Ausloggen eines Benutzers an allen Geräten, das Löschen eines bisher noch im Status ‚*Draft*' befindlichen Kapitels - wobei hier weitere Anpassung im Hinblick auf den Umgang der Chapter-Ids nötig sind - und das Ändern des eigenen Profilnamen und Passwort.

Eine weitere Änderung im Auth-Flow im Backend wäre es, den PublicKey zur Valididerung der JWT-Access-Token für die anderen Services zur Verfügung zu stellen, sodass diese den Token selbst validieren können und nicht zum "User-Service" schicken und auf die Antwort zu warten.

In der Softwarearchitektur sollte der Service-Layer weiter ausgebaut werden, und die Business-Logik aus dem Controller Layer komplett entfernt werden. Somit könnte man auch für außenstehende andere Protokolle zur Kommunikation mit der API hinzufügen, so wie gRPC oder htmx. Weiterhin wäre zu überprüfen, wie sich unsere Applikation verhält, wenn wir getrennte Datenbanken für jeden einzelnen Service haben. Erste Tests zeigen, dass wenn die Foreign-Keys zu den Tabellen der anderen Services gelöscht werden, unsere Applikation weiterhin funktioniert. Nur der Test-Data-Service kann damit zum aktuellen Zeitpunkt nicht umgehen.

Zusammenfassend ist VerseVault in dieser Version ein gelungenes Cloud Engineering Projekt, an welches wir unser gelerntes Wissen des Moduls angewandt haben. Es weist viele interessante Erweiterungsmöglichkeiten auf, welche in der Zukunft umgesetzt werden können.

6. Literaturverzeichnis

IBM (o. D.): Was sind Microservices?, IBM, [online] https://www.ibm.com/de-de/topics/microservices [abgerufen am 13.01.2024].

gRPC Authors (o. D.): gRPC: A high performance, open source universal RPC framework, gRPC Authors, [online] https://grpc.io/ [abgerufen am 13.01.2024].

Cloudflare (o. D. a): Was ist Load Balancing? | Wie Load Balancer funktionieren, Cloudflare, [online] https://www.cloudflare.com/de-de/learning/performance/what-is-load-balancing/ [abgerufen am 13.01.2024].

Cloudflare (o. D. b): What is a reverse proxy | Proxy servers explained, Cloudflare, [online] https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/ [abgerufen am 13.01.2024].

The Kubernetes Authors (o. D. a): Kubernetes, The Kubernetes Authors, [online] https://kubernetes.io/de/ [abgerufen am 14.01.2024].

The Kubernetes Authors (o. D. b): Kubernetes, The Kubernetes Authors, [online] https://kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/ [abgerufen am 14.01.2024].

Amazon (o. D.): Was ist Continuous Integration?, Amazon, [online] https://aws.amazon.com/de/devops/continuous-integration/ [abgerufen am 13.01.2024].

GitLab (o. D.): What is CI/CD?, GitLab [online] https://about.gitlab.com/topics/ci-cd/ [abgerufen am 13.01.2024].

Campero, W. (2022): Lasttest und Performancetest (Load Testing) - Konzept und Tools, Qytera, [online] https://www.qytera.de/blog/lasttest-konzept-tools [abgerufen am 14.01.2024].

Grafana Labs (o. D.): Alerting, Grafana Labs, [online] https://grafana.com/docs/grafana/latest/alerting/ [abgerufen am 14.01.2024].

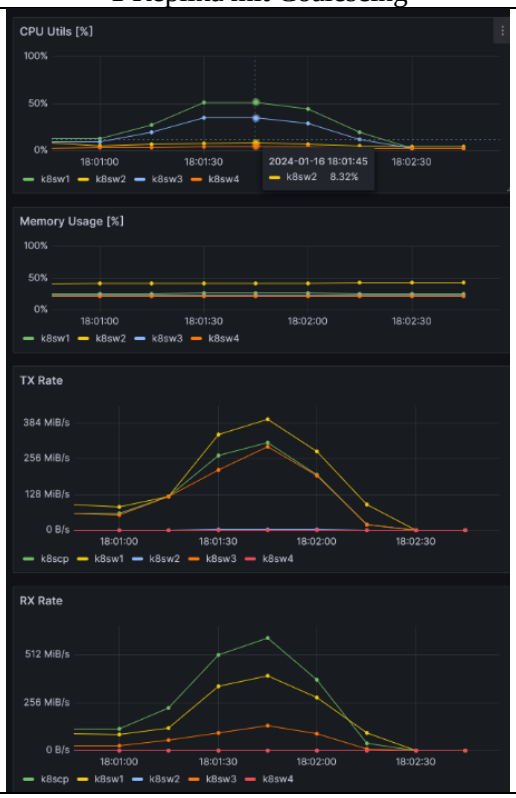
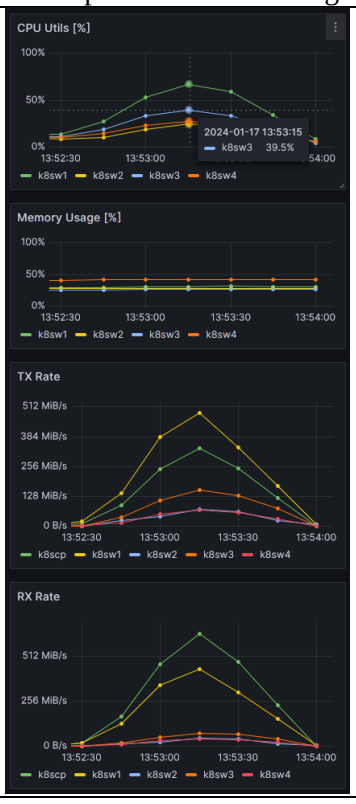
Prometheus (o. D.): Alertmanager, Prometheus, [online] https://prometheus.io/docs/alerting/latest/alertmanager/ [abgerufen am 14.01.2024].

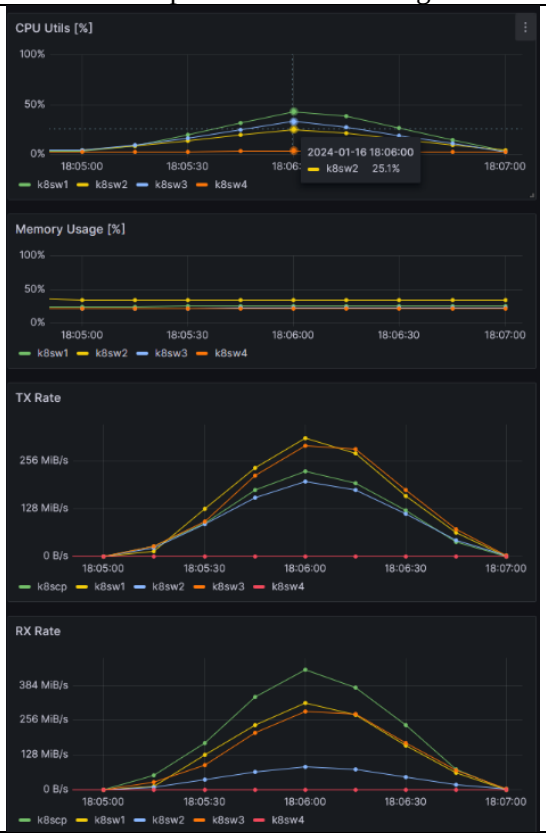
Bunny.net Support Hub (o . D.): Understanding Request Coalescing, bunny.net Support Hub, [online] https://support.bunny.net/hc/en-us/articles/6762047083922-Understanding-Request-Coalescing [abgerufen am 14.01.2024].

Anand, A. (2023): Request Coalescing: A shield against traffic spikes | Implementation in Go, Medium, [online] https://medium.com/@atarax/request-coalescing-a-shield-against-traffic-spikes-implementation-in-go-8d6cb3258630 [abgerufen am 14.01.2024].

Anhang

Grafana-Darstellung der Tabelle 5. Es ist zu beachten, dass auf k8sw2 die Datenbank ist, und auf den restlichen Workern die Book-Service Replikas sind.

1 Replika mit Coalescing	2 Replikas mit Coalescing
	
3 Replikas mit Coalescing	**10 Replikas mit Coalescing**
	

1 Replika ohne Coalescing	2 Replikas ohne Coalescing
	

3 Replikas ohne Coalescing