

Hochschule Darmstadt

– Fachbereich Informatik –

Einfluss von externer Zugriffskontrolle auf die Performance in OAuth2 Systemen

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

vorgelegt von

Matthias Adrian

Matrikelnummer: 752237

Referent : Prof. Dr. Peter Altenbernd
Korreferent : Prof. Dr. Arnim Malcherek

ERKLÄRUNG

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, 31. August 2021

Matthias Adrian

ABSTRACT

Open Policy Agent (OPA) decouples access control from a server. When a client sends a request to a server, this server does not evaluate an access decision itself, but OPA does as an external program. This creates additional communication between the server and OPA. In OAuth2 systems, the evaluation of access decisions is based on tokens that the client sends to the server. In past performance tests, the latency of the communication between server and OPA was not taken into account, and no access decisions were made on the basis of tokens [5].

In order to investigate to what extent decoupling the access control with OPA affects the performance, two test servers were implemented. In one server the access control is decoupled with OPA, in the other the server evaluates access decisions itself. Apache JMeter was used to examine the performance. This tool generates requests with valid tokens to the servers and measures the response times. With ten simultaneous users, response times on the server with OPA turned out to be three times as high compared to the server without OPA. In addition, the server with OPA scales worse, because with increasing load up to 100 users, the response times increased linearly, while on the server without OPA these remained more consistent. A significant performance disadvantage could be demonstrated. The attempt, to improve the performance of the system with OPA by deploying it in a Kubernetes Cluster, has failed.

ZUSAMMENFASSUNG

Open Policy Agent (OPA) entkoppelt die Zugriffskontrolle (access control) von einem Server. Wenn ein Client eine Anfrage an einen Server sendet, evaluiert dieser Server nicht selbst eine Zugriffsentscheidung, sondern OPA als externes Programm. Hier entsteht zusätzliche Kommunikation zwischen Server und OPA. In OAuth2 Systemen geschieht die Evaluierung von Zugriffsentscheidungen anhand von Token, die der Client an den Server sendet. In bisherigen Performancetests wurde die Latenz bei der Kommunikation zwischen Server und OPA nicht in Betrachtung gezogen, sowie keine Zugriffsentscheidungen (access decisions) anhand von Token getroffen [5].

Um zu untersuchen, inwiefern sich eine Entkopplung der Zugriffskontrolle mit OPA auf die Performance auswirkt, wurden zwei Testserver implementiert. In dem einen Server wird die Zugriffskontrolle mit OPA entkoppelt (decoupled), in dem anderen evaluiert der Server Zugriffsentscheidungen selbst. Um die Performance zu untersuchen, wurde Apache JMeter verwendet. Dieses Tool generiert Anfragen mit validem Token an die Server und misst die Response Time der Antworten. Bei zehn gleichzeitigen Nutzern stellte es sich heraus, dass die Response Times bei dem Server mit OPA durchschnittlich dreimal so hoch ausfielen im Vergleich zu dem Server ohne OPA. Zudem skaliert der Server mit OPA schlechter, denn bei ansteigender Last auf bis zu 100 Nutzer stiegen die Response Times linear, während bei dem Server ohne OPA diese weitaus konstanter blieben. Es konnte also ein signifikanter Performancenachteil nachgewiesen werden. Der Versuch, die Performance des Systems mit OPA durch ein Deployment in einem Kubernetes Cluster zu verbessern, ist gescheitert.

INHALTSVERZEICHNIS

I THESIS

1	EINLEITUNG	2
1.1	Motivation	3
1.2	Vorgehensweise und Ergebnisse	4
1.3	Gliederung	5
2	TECHNISCHE GRUNDLAGEN	6
2.1	Grundlegende Begriffe der IT-Sicherheit	6
2.2	OAuth2	7
2.2.1	Rollen in OAuth2	7
2.2.2	Erhalt von Token	7
2.3	JSON Web Token	8
2.3.1	JSON Web Token Signatur	9
2.3.2	JSON Web Key	9
2.3.3	Base64 Kodierung	10
2.4	OpenID Connect	10
2.4.1	ID Token	10
2.4.2	Hybrid Flow	11
2.5	OAuth2 Endpunkte des Autorisationsservers	12
2.5.1	Authorization Endpunkt	12
2.5.2	Token Endpunkt	13
2.5.3	JSON Web Key Set Endpunkt	13
2.6	Role Based Access Control	13
2.7	Open Policy Agent	13
2.8	Apache Tomcat	15
2.9	Spring	15
2.9.1	Spring Boot	15
2.9.2	Spring Security	15
2.10	Keycloak	16
2.11	Apache JMeter und Metriken	16
2.12	Application Performance Index	17
2.13	Postman	18
2.14	Docker-Container	18
2.15	Kubernetes	18
2.16	Zusammenfassung der technischen Grundlagen	19
3	ZUGRIFFSKONTROLLE UND PERFORMANCE IN OAUTH2	21
3.1	Keycloak als Authorization Server und Identity Provider	21
3.2	Erhalt eines Tokens mit Postman	23
3.2.1	Authorization Code Grant mit Postman	23
3.2.2	Darstellung des erhaltenen Tokens	24
3.3	Ressource Server	27
3.3.1	Schnittstelle	27

3.3.2	Spring Security OAuth2	29
3.3.3	Ressource Server mit interner Zugriffskontrolle	30
3.3.4	Ressource Server mit externer Zugriffskontrolle	32
3.3.4.1	Zugriffskontrolle mit Open Policy Agent	32
3.3.4.2	AccessDecisionManager	34
3.4	Kubernetes Deployment	35
3.5	Performancetests mit Apache JMeter	37
3.5.1	Lasttest	38
3.5.2	Skalierbarkeitstest	41
3.5.3	Stresstest	42
3.5.4	Messung und Protokollierung von Messdaten	43
3.6	Systemhardware und Testumfeld	44
4	AUSWERTUNG DER PERFORMANCETESTS	46
4.1	Lasttest	46
4.2	Skalierbarkeitstest	50
4.3	Stresstest	53
4.4	Kubernetes Cluster	55
4.5	Fazit	57
5	STAND DER TECHNIK	59
6	ZUSAMMENFASSUNG	61
 II APPENDIX		
A	APPENDIX	64
 LITERATUR		68

ABBILDUNGSVERZEICHNIS

Abbildung 2.1	Authorization Code Grant	8
Abbildung 2.2	Kubernetes Cluster	19
Abbildung 3.1	Keycloak Sample User	22
Abbildung 3.2	Keycloak Sample User Role	22
Abbildung 3.3	Authorization Code Grant mit Postman	23
Abbildung 3.4	Projekstruktur Spring Applikation	28
Abbildung 3.5	Bearer Token Authentication	29
Abbildung 3.6	JWT Authentication Provider	31
Abbildung 3.7	Open Policy Agent und OAuth2	33
Abbildung 3.8	AccessDecisionManager in Spring Security	34
Abbildung 3.9	Thread Group-Last	39
Abbildung 3.10	Test Plan-Last	39
Abbildung 3.11	HTTP Header Manager	40
Abbildung 3.12	HTTP Request	41
Abbildung 3.13	Thread Group-Skalierung	42
Abbildung 3.14	Thread Group-Stress	43
Abbildung 4.1	Response Time Graph von Lasttest von Ressource Server ohne OPA	47
Abbildung 4.2	Latency Time Graph von Lasttest von Ressource Server ohne OPA	47
Abbildung 4.3	Response Time Graph von Lasttest von Ressource Server mit OPA	48
Abbildung 4.4	Statistiken von Lasttest von Ressource Server ohne OPA	48
Abbildung 4.5	Statistiken von Lasttest von Ressource Server mit OPA	49
Abbildung 4.6	Perfmon von Lasttest von Ressource Server ohne OPA	49
Abbildung 4.7	Perfmon von Lasttest von Ressource Server mit OPA	49
Abbildung 4.8	Aktive Threads über Zeitraum des Tests	50
Abbildung 4.9	Response Time Graph von Skalierbarkeitstest von Ressource Server ohne OPA	51
Abbildung 4.10	Response Time Graph von Skalierbarkeitstest von Ressource Server mit OPA	51
Abbildung 4.11	Perfmon von Skalierbarkeitstest von Ressource Server ohne OPA	52
Abbildung 4.12	Perfmon von Skalierbarkeitstest von Ressource Server mit OPA	52
Abbildung 4.13	Statistiken von Skalierbarkeitstest von Ressource Server mit OPA	52
Abbildung 4.14	Response Time Graph von Stresstest von Ressource Server ohne OPA	53
Abbildung 4.15	Response Time Graph von Stresstest von Ressource Server mit OPA	54

Abbildung 4.16	Request Summary von Stresstest von Ressource Server mit OPA	54
Abbildung 4.17	APDEX (Application Performance Index) von Stresstest von Ressource Server mit OPA	55
Abbildung 4.18	APDEX (Application Performance Index) von Stresstest von Ressource Server ohne OPA	55
Abbildung 4.19	Response Time Graph von Lasttest von Ressource Server mit OPA in einem Kubernetes Cluster	56
Abbildung 4.20	Statistiken von Lasttest von Ressource Server mit OPA in Kubernetes Cluster	56
Abbildung 4.21	APDEX (Application Performance Index) von Skalierbarkeitstest von Ressource Server mit OPA und Kubernetes	57
Abbildung 4.22	APDEX (Application Performance Index) von Skalierbarkeitstest von Ressource Server mit OPA ohne Kubernetes	57
Abbildung A.1	Statistiken von Skalierbarkeitstest von Ressource Server mit OPA in einem Kubernetes Cluster	66
Abbildung A.2	Response Time Graph von Skalierbarkeitstest von Ressource Server mit OPA in einem Kubernetes Cluster	67
Abbildung A.3	Statistiken von Stresstest von Ressource Server mit OPA in einem Kubernetes Cluster	67

TABELLENVERZEICHNIS

Tabelle 2.1	ID Token	11
Tabelle 2.2	Standard Claims	11
Tabelle 2.3	Beispiel von Grenzwerten eines Application Performance Index	17
Tabelle 3.1	Application Performance Index	44
Tabelle 3.2	System des Clients	45
Tabelle 3.3	System des Servers	45

LISTINGS

Listing 2.1	Beispiel JSON Web Token	9
Listing 2.2	Zugriffsrichtlinie in Rego	14
Listing 2.3	Input-Daten für Open Policy Agent	14
Listing 2.4	Zugriffsentscheidung in Rego	14
Listing 3.1	Authorization Request	24
Listing 3.2	Base64-kodierter JSON Web Token	24
Listing 3.3	Dekodierter JSON Web Token	25
Listing 3.4	HTTP-Schnittstelle der Ressource Server	27
Listing 3.5	JwtAuthenticationConverter	31
Listing 3.6	HttpSecurity	32
Listing 3.7	Rollenbasierte Zugriffsrichtlinie in Rego	33
Listing 3.8	Deployment von Ressource Server und OPA	35
Listing 3.9	Service von Ressource Server und OPA	37
Listing 4.1	Fehlermeldung Spring-Boot	53
Listing A.1	Deployment und Service von Keycloak	64
Listing A.2	Deployment und Service von PostgreSQL als Daten- bank für Keycloak	65

ABKÜRZUNGSVERZEICHNIS

OPA	Open Policy Agent
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
CPU	Central processing unit
RBAC	Role Based Access Control
WSL 2	Windows-Subsystem für Linux 2
RSA	Rivest-Shamir-Adleman
URI	Uniform Resource Identifier
RAM	Random-Access Memory
TCP	Transmission Control Protocol

Teil I

THESIS

EINLEITUNG

OAuth2 als Spezifikation kann unter anderem dafür genutzt werden Hypertext Transfer Protocol ([HTTP](#))-Schnittstellen zu sichern. Es ist heutzutage praktisch Standard und hat eine Vielzahl von Einsatzzwecken, wie neben der Sicherung von Schnittstellen auch die Authentifizierung. Viele Konzerne setzen Implementierungen dieser Spezifikation ein, so auch beispielsweise Microsoft [33]. Ein typischer Ablauf in OAuth2 ist folgendermaßen zu beschreiben:

Ein Nutzer authentifiziert sich am Client-Computer und daraufhin erhält der OAuth2-Client von einem Autorisationsserver einen Token, mit dem er auf gesicherte Schnittstellen eines Ressource Servers zugreifen kann. Hierbei ist zu erwähnen, dass nutzerspezifische Attribute in den Token gemappt werden. Dies kann beispielsweise der Vor-und-Nachname sein, die E-Mail-Adresse, die Abteilung, in der der Nutzer arbeitet, sowie etwaige Rollen und Gruppenzugehörigkeiten. In dem Ressource Server ist es oftmals notwendig nach diesen Nutzerattributen zu autorisieren. Beispielsweise kann die Anforderung bestehen, dass ein Nutzer mit der Rolle *ROLE_ADMIN* oder ein Nutzer der in der Abteilung *Human Ressources* arbeitet, Admin-Privilegien erhalten soll, das heißt auf Schnittstellen zugreifen kann, die voraussetzen das diese Attribute in dem Token vorhanden sind.

Grundsätzlich lassen sich diese Zugriffsrichtlinien in dem Ressource Server selbst implementieren. Allerdings haben wir es in der Praxis in Firmen oftmals mit einer Vielzahl von Applikationen, das heißt Ressource Servern, zu tun die verschiedene Zugriffsrichtlinien haben und zudem mit verschiedenen Programmiersprachen und Frameworks geschrieben sind. Dies kann vor allem bei komplexen und sich häufig ändernden Zugriffsrichtlinien zum einen zu einem hohen Wartungsaufwand führen und zum anderen auch zu Sicherheitsrisiken führen. Unter anderem aus diesem Grund wurde Open Policy Agent ([OPA](#)) entwickelt. Es entkoppelt die Zugriffskontrolle von dem Ressource Server als externes Programm in einer leicht verständlichen Programmiersprache, die ausschließlich für die Zugriffskontrolle zuständig ist. Wenn ein Ressource Server von einem Client eine HTTP-Anfrage mit einem validen Token erhält, sendet er diese Anfrage mit dem Token an den OPA-Service. In dem OPA-Service sind Richtlinien zur Zugriffskontrolle in der Programmiersprache Rego hinterlegt. Beispielsweise kann in dem OPA-Service eine Zugriffskontrolle hinterlegt sein, die besagt, dass auf die HTTP-GET-Schnittstelle mit dem Pfad `/secretData` nur Zugriff gewährleistet werden darf, wenn in dem Token das Schlüssel-Wert-Paar *roles: ROLE_ADMIN* hinterlegt ist. Falls dem so ist, würde der OPA-Service eine Zugriffserlaubnis an den Ressource Server zurücksenden, der wiederum dem Client entsprechend die Daten von der GET-Schnittstelle `/secretData` zusendet, da ja

OPA evaluiert hat, dass dieser Token die nötigen Befugnisse verfügt und damit der Nutzer ordnungsgemäß autorisiert ist diese Schnittstelle aufzurufen. Ein weiterer Vorteil der Entkopplung von Zugriffskontrolle ist aus Sicht des Software-Engineering die Trennung der Zuständigkeiten (Separation of Concerns).

1.1 MOTIVATION

Ein Nachteil dieser Entkopplung speziell in OAuth2 Systemen ist der, dass der Ressource Server jedes Mal bei einkommenden HTTP-Anfragen dem OPA-Service die HTTP-Anfrage inklusive des Tokens zusenden muss, dieser den Token dekodieren, parsen und schlussendlich anhand dessen eine Zugriffsentscheidung evaluieren und dem Ressource Server zusenden muss. Das heißt es entsteht ein zusätzlicher Kommunikationsverkehr, was zu Performanceproblemen führen kann. Bei einer hohen Last auf Schnittstellen, werden diese in der Regel horizontal skaliert. Das bedeutet, falls eine Schnittstelle aufgrund zu hoher Last nicht mehr in akzeptabler Zeit HTTP-Anfragen beantworten kann, wird eine neue Instanz dieser Applikation auf einem zweiten Host erstellt und ein Loadbalancer sorgt dafür, dass die Last gleichmäßig verteilt wird und dadurch die Antwortzeiten möglichst niedrig gehalten werden [28]. Diese Metrik wird auch die Response Time genannt, also die Zeit vor dem Senden einer HTTP-Anfrage an den Server bis zum Eintreffen des letzten Bytes der Antwort des Servers [18].

Die Entwickler von Open Policy Agent geben an, dass anhand von durchgeführten Benchmarks Zugriffsentscheidungen in der Regel nur Rechenzeit im Bereich von unter einer Millisekunde benötigen [5]. In diesen Benchmarks wurden allerdings weder zu dekodierenden Tokens verwendet noch wurde auf die umso wichtigere Response Time eingegangen: Nämlich die Zeit, die gebraucht wird, wenn ein Client eine HTTP-Anfrage an einen Ressource Server sendet, dieser die Anfrage an den OPA-Service zur Evaluierung einer Zugriffsentscheidung sendet und schlussendlich der Ressource Server basierend auf der Zugriffsentscheidung des OPA-Services dem Client antwortet. Zudem ist ungewiss, wie sich externe Zugriffskontrolle mit OPA unter Last im Vergleich zur Zugriffskontrolle, die in dem Ressource Server selbst implementiert ist, verhält. Potenziell könnten nicht vertretbar hohe Latenzen entstehen, die entweder durch horizontale Skalierung gelöst werden müssen oder das Nutzererlebnis leidet unter hohen Antwortzeiten.

In der Arbeit *Verteilte Policy-basierte Autorisierung mit OAuth 2.0 und OpenID Connect* wurde behauptet, dass keine nennenswerte Latenz zwischen Ressource Server und OPA entsteht, wenn diese in einem Kubernetes Cluster ausgeführt werden und sich OPA als Sidecar in einem Pod mit dem Ressource Server befindet [42]. Hierauf wurde sich auf einen Artikel von Microsoft berufen [36]. Diese Behauptung wurde allerdings nicht bewiesen, Performancetests hinsichtlich der Response Time beziehungsweise Latenz wurden nicht durchgeführt.

1.2 VORGEHENSWEISE UND ERGEBNISSE

Um den Einfluss von externer Zugriffskontrolle auf die Performance in OAuth2 Systemen zu untersuchen, wurden zwei Ressource Server implementiert. In dem einen System wird die Zugriffskontrolle in dem Ressource Server selbst gehandhabt und in dem anderen wird sie entkoppelt durch den Open Policy Agent. Als Tool zur Generierung von Last und dem Messen und Protokollieren der Response Time wurde Apache JMeter verwendet. Neben dem Last-Test wurden zwei weitere Testpläne in JMeter erstellt. Ein Test zur Skalierbarkeit und ein Stress-Test. Bei dem Skalierbarkeitstest wird nicht eine gleichbleibende Last erzeugt, sondern es wird periodisch Last durch hinzukommende Threads erhöht, um zu sehen, inwiefern sich die Response Time bei hinzukommender Last erhöht. Diese Art des Tests ist aus wirtschaftlicher Sicht sinnvoll, denn anhand dessen lässt sich herleiten, wie sehr sich Antwortzeiten bei hinzukommender Anzahl von Nutzern erhöhen. Dadurch kann vorausgesehen werden, wann Ressource Server skaliert werden müssen, damit die Anfragen von Nutzern akzeptabel niedrige Antwortzeiten haben.

Bei dem Stresstest wird eine hohe Last auf den Ressource Servern generiert mit dem Ziel herauszufinden, wie viele Nutzer der Ressource Server gleichzeitig bedienen kann bis entweder der Ressource Server vollkommen unerreichbar ist oder die Antwortzeiten unakzeptabel hoch werden. Diese drei Testpläne, Last-, Skalierbarkeit- und Stresstest wurden auf dem Ressource Server mit und ohne OPA-Service durchgeführt und die Resultate verglichen.

Zudem wurden die jeweiligen Performancetests für das System mit OPA, einmal mit Kubernetes durchgeführt, wobei sich hier Ressource Server und OPA in einem Pod befinden, als auch ohne Kubernetes durchgeführt. Wenn sich zwei Container in einem Pod befinden, bezeichnet man dies als Sidecar-Muster. Hierbei teilen sich die Container Speicher- und Netzwerkressourcen [36].

Bei zehn gleichzeitigen Threads (Lasttest) stellte es sich heraus, dass die Response Times bei dem System mit OPA durchschnittlich mehr als dreimal so hoch ausfielen. Außerdem fiel das System mit Open Policy Agent hier durch eine doppelt so hohe Central processing unit (CPU)-Auslastung negativ auf im Vergleich zu dem System ohne externe Zugriffskontrolle. Zudem skaliert das System mit OPA deutlich schlechter, denn bei ansteigender Last auf bis zu 100 Nutzer stiegen die Response Times deutlich stärker als bei dem System ohne OPA. Unter starker Last war Open Policy Agent zeitweise nicht erreichbar für den Ressource Server. Es konnte also ein signifikanter Performancenachteil nachgewiesen werden, sowie eine Unerreichbarkeit von Open Policy Agent unter starker Last festgestellt werden. Die Hypothese, dass ein negativer Einfluss auf die Performance durch den zusätzlichen Kommunikationsaufwand der zwischen Server und OPA auftritt, entsteht, wurde bestätigt. Die Nutzung von Kubernetes hatte keine Verbesserung der Performance des Systems mit OPA zur Folge, sondern im Gegenteil, die

Performance hat sich sogar verschlechtert. Die Behauptung, dass keine nennenswerte Latenz entsteht, wenn sich Ressource Server und OPA in einem Pod in einem Kubernetes Cluster befinden, wurde widerlegt.

1.3 GLIEDERUNG

Zunächst werden in [Kapitel 2](#) die benötigten technischen Grundlagen zusammengetragen. Darauf folgt [Kapitel 3](#), in dem die jeweiligen Ressource Server und die Performancetests, die mit Apache JMeter ausgeführt werden, beschrieben werden. In [Kapitel 4](#) werden die Ergebnisse der Performance-tests ausgewertet. In dem Kapitel Stand der Technik, [Kapitel 5](#), werden verwandte Arbeiten diskutiert. Zum Schluss folgt eine Zusammenfassung und Ausblick, dies ist [Kapitel 6](#).

TECHNISCHE GRUNDLAGEN

In diesem Kapitel werden alle nötigen technischen Grundlagen zusammengetragen. Da es sich um ein OAuth2 System handelt, wird diese Spezifikation erläutert. Zudem wird auf die Tokens eingegangen, die die Ressource Server jeweils validieren müssen. Schließlich werden die Metriken zum Messen der Performance, das Programm zum Messen dieser Werte sowie das Framework zur Implementierung der Ressource Server erläutert.

2.1 GRUNDLEGENDE BEGRIFFE DER IT-SICHERHEIT

Im Bereich der IT-Sicherheit gibt es eine Reihe relevanter Begriffe, die regelmäßig verwendet werden.

AUTHENTIFIZIERUNG: Bei einer Authentifizierung beweist ein Nutzer seine Identität, indem er dem gegenüberliegenden System Informationen übermittelt, über die nur der Nutzer verfügen kann wie beispielsweise ein Nutzernamen und Passwort oder ein Token [11].

AUTORISIERUNG: Eine Autorisierung findet grundsätzlich nach der Authentifizierung statt. Hier überprüft das System nach der Authentifizierung des Nutzers, ob der Nutzer über die notwendigen Rechte verfügt eine Aktion auszuführen [44].

INTEGRITÄT: Bei der Integrität wird verifiziert, dass Daten seit ihrer Erstellung nicht verändert wurden. Dies kann beispielsweise durch eine Signatur, die nur der Ersteller der Daten erzeugen kann, gewährleistet werden [46].

AUTHENTIZITÄT: Das Ziel der Authentizität ist die Verifizierung des Erstellers der Daten [46]. Das bedeutet, es soll möglich sein zu überprüfen, dass die Daten, die zugesendet werden, auch tatsächlich von der Person oder dem System kommen, von der angenommen wird, dass sie der Urheber ist.

VALIDIERUNG: Es lässt sich von einer Validierung sprechen, wenn die Authentizität und Integrität überprüft werden [8].

ZUGRIFFSKONTROLLE: Zugriffskontrolle ist eine Sicherheitsfunktion, die gemeinsame Ressourcen gegen unautorisierten Zugriff schützt. Der Unterschied zwischen autorisierten und unautorisierten Zugriff wird anhand einer Zugriffskontrollrichtlinie gemacht [10].

2.2 OAUTH2

OAuth2 ist eine Spezifikation, die ursprünglich entwickelt wurde, um Dritten Zugriff auf die Daten eines sogenannten Ressource Owners zu ermöglichen, ohne dass dieser Ressource Owner dem Dritten, in der Spezifikation Client genannt, sein Nutzernamen und Passwort übermitteln muss [13]. Dies wird dadurch ermöglicht, dass dem Client von einem Autorisationsserver ein Token übermittelt wird und mit diesem Token kann der Client Daten des Ressource Owners abfragen oder Aktionen im Namen des Ressource Owners durchführen. Diese Daten des Ressource Owners werden auf dem sogenannten Ressource Server gehostet. Im Laufe der Zeit wurde diese Spezifikation erweitert, um weitere Anwendungsfälle abzudecken, wie die Spezifikation OpenID Connect, welche Single Sign-On ermöglicht [38].

2.2.1 Rollen in OAuth2

Es gibt vier verschiedene Rollen, die in OAuth2 spezifiziert sind [13].

RESOURCE OWNER: Der Ressource Owner ist dazu in der Lage, Dritten Zugriff auf seine Daten zu geben oder Aktionen in seinem Namen durchzuführen. Falls der Ressource Owner eine Person ist, wird dieser als End-Nutzer bezeichnet.

RESOURCE SERVER: Der Ressource Server hostet die geschützten Ressourcen des Ressource Owners und ist in der Lage Anfragen zu diesen Ressourcen zu beantworten, falls ein valider Token, ein sogenannter Access Token, in der Anfrage zur Ressource vorliegt.

CLIENT: Der Client ist diejenige Komponente, die die Ressource des Ressource Owner's von dem Ressource Servers erlangen möchte und dafür einen Access Token benötigt.

AUTHORIZATION SERVER: Der Authorization Server, zu Deutsch Autorisationsserver, übermittelt Access Token falls sich der Ressource Owner erfolgreich authentifiziert und gegebenenfalls den Client autorisiert hat, Aktionen durchzuführen.

2.2.2 Erhalt von Token

Gemäß der Spezifikation gibt es fünf verschiedene Wege, wie ein Client von einem Authorization Server einen Token erhalten kann [13]. Im Folgenden wird nur der sogenannte Authorization Code Grant beschrieben, da die anderen vier (Client Credential Grant, User Password Credential Grant, Implicit Grant, Refresh Token) für diese Arbeit keine Relevanz haben.

Bei dem Authorization Code Grant wird der Ressource Owner, der einen Webbrowser verwendet, von dem Client auf den Authorization Server weitergeleitet. Hier wird der Ressource Owner gebeten, sich bei dem Authoriza-

tion Server zu authentifizieren und dann den Client zu autorisieren auf eine geschützte Ressource des Ressource Servers zuzugreifen [13].

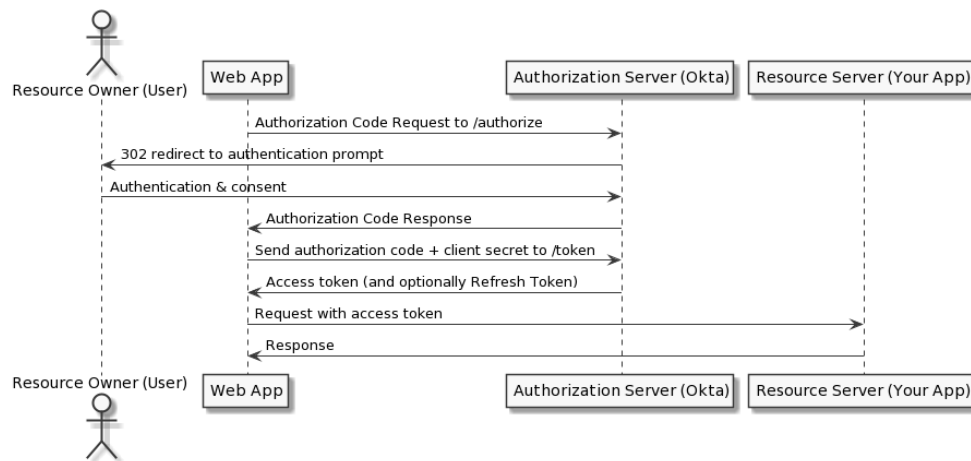


Abbildung 2.1: Authorization Code Grant [50]

In [Abbildung 2.1](#) ist der Authorization Code Grant Flow als Sequenzdiagramm abgebildet. Es ist zu sehen, dass es Beziehungen zwischen allen vier Rollen gibt, nämlich dem Ressource Owner, dem Client, in diesem Fall als „Web App“ gekennzeichnet, dem Authorization Server, Okta, und dem Resource Server. In der Praxis ist der Ablauf folgendermaßen zu beschreiben:

Ein Nutzer bedient eine Webapplikation (Client) und diese Webapplikation möchte, dass der Nutzer sie autorisiert, in ihrem Namen beispielsweise seine Adressdaten von dem Google-Konto abzufragen. In diesem Fall ist Google der Authorization und Resource Server. Die Webapplikation, der Client, leitet den Nutzer auf den Google-Server weiter, wo er dann aufgefordert wird sich zu authentifizieren und der Webapplikation Erlaubnis zu erteilen, seine Adressdaten abzufragen. Daraufhin wird der Webbrowser des End-Nutzers zu dem Client zurückgeleitet und der Client erhält einen Token von dem Authorization Server, womit er die Ressource des End-Nutzers von dem Resource Server abfragen kann.

2.3 JSON WEB TOKEN

Wenn ein Client einen Access Token von einem Authorization Server erhält, um Zugriff auf HTTP-Schnittstellen des Resource Servers zu erhalten, sind diese Token bei den heutigen Implementierungen von Autorisationsservern sogenannte JavaScript Object Notation ([JSON](#)) Web Token. Dies ist auch bei dem Autorisationsserver von Microsoft [39], Azure Active Directory, sowie Keycloak [23] der Fall. [JSON](#) ist ein leichtgewichtiges Datenaustauschformat [12].

JSON Web Token ist eine Spezifikation für Token, die beschreibt wie die Tokens aufgebaut sind und welche Inhalte sie haben [32]. Signierte JSON Web Token sind in drei Teile unterteilt: Header, Payload und die Signatur. Die Gesamtstruktur ist in JSON dargestellt, das heißt es gibt jeweils

Schlüssel-Wert-Paare in JSON. Zur Übertragung werden die JSON Web Token base64-kodiert und hierbei werden jeweils die drei Teile, also Header, Payload und Signatur, durch einen Punkt (.,) voneinander getrennt [32]. In dem Header und Payload stehen sogenannte Claims. Ein Claim ist ein Schlüssel-Wert-Paar. Diese liefern Informationen über den Token wie zum Beispiel der Claim „exp“, kurz für „Expiration Time“, der angibt, bis wann der Token gültig ist. Einen Token, der schon abgelaufen ist, sollte der Resource Server nicht akzeptieren. Die Namen der Schlüssel der Claims sind standardisiert.

Listing 2.1: Beispiel JSON Web Token

```
{
  alg: "RS256",
  typ: "JWT",
  kid: "KaD7XcL-5_olEdjzt18fCqrR2R-uCf1BtCyngSfl7mg"
}.
{
  exp: 1625764809,
  iat: 1625763009,
  jti: "7df07f70-5c50-41fd-b757-2e3faa8c50e4",
  iss: "HTTP://localhost:9080/auth/realms/sample",
  aud: "account",
  sub: "de16edd9-a654-4769-b71c-3faf34583890",
  typ: "Bearer",
  azp: "test-app",
  session_state: "a19e1f5f-4080-488f-b193-a943925d62fc",
  acr: "1",
}.
[signature]
```

Obenstehend ist ein rudimentärer dekodierter JSON Web Token dargestellt. In dem [Unterabschnitt 2.4.1](#) wird erläutert, welche Bedeutung die einzelnen Claims haben.

2.3.1 JSON Web Token Signatur

JSON Web Tokens sind in der Regel mit dem asymmetrischen Verfahren Rivest-Shamir-Adleman ([RSA](#)) signiert [32]. Das bedeutet, dass mittels eines privaten Schlüssels, über den nur der Ersteller des Tokens verfügt, also der Autorisationsserver, die Signatur erzeugt wird. Daneben gibt es einen öffentlichen Schlüssel, mit dem die Signatur überprüft werden kann. Das heißt es ist nicht möglich, dass der Client den Token verändern kann, da dann die Signatur nicht mehr übereinstimmt.

2.3.2 JSON Web Key

Im Falle eines durch RSA signierten JSON Web Token, ist der JSON Web Key der in JSON dargestellte öffentliche Schlüssel. Mithilfe dieses öffentlichen

Schlüssels kann die Signatur aus dem Header und Payload des JSON Web Tokens berechnet werden. Entspricht die berechnete Signatur, der Signatur die in dem JSON Web Token vorhanden ist, ist der Token erfolgreich validiert und Integrität sowie Authentizität sind sichergestellt [20].

2.3.3 Base64 Kodierung

JSON Web Token werden, wenn sie übertragen werden, in Base64 kodiert [32]. Base64 ist keine Verschlüsselung und auch keine Datenkomprimierung, sondern dient vor allem dem Zweck, dass Daten in ausschließlich ANSI-Zeichen kodiert übertragen werden. Irreguläre Zeichen wie Umlaute werden nicht übertragen [21].

2.4 OPENID CONNECT

OpenID Connect ist eine Authentifizierungsschicht, die auf der OAuth2 Spezifikation aufbaut. OpenID Connect baut auf OAuth2 eine Authentifizierung auf, indem es in dem JSON Web Token Claims steckt, dass den Client, der den Token erhalten hat, nachdem sich ein End-Nutzer bei dem Authorization Server authentifiziert hat, die Möglichkeit gibt die Identität dieses End-Nutzers zu verifizieren. Diese Art von JSON Web Token, die der Client erhält, nennt sich ID Token [38].

2.4.1 ID Token

ID Token sind JSON Web Token, die gewisse Claims in ihrem Payload gesetzt haben [38].

CLAIM	BESCHREIBUNG
iss	Die Issuer-URL. Dies ist die URL des Authorization Server, der den Token erstellt hat („Herausgeber“).
sub	Das Subject. Dies ist ein eindeutiger Identifier des Nutzers, der sich authentifiziert hat.
aud	Die Audience. Das ist der Empfänger des Tokens, in der Regel die Client-ID.
exp	Expiration Time. Das ist der Zeitpunkt, zu dem der Token abgelaufen ist.
iat	Issued at. Das ist der Zeitpunkt, zu dem der Token herausgegeben wurde.

Tabelle 2.1: ID Token.

In [Tabelle 2.1](#) sind die Claims zu sehen, die in dem Payload gesetzt sein müssen, damit ein JSON Web Token als ID-Token bezeichnet werden kann.

MEMBER	TYPE	BESCHREIBUNG
sub	String	Die eindeutige ID des End-Nutzers.
name	String	Vor-und-Nachname.
given_name	String	Vorname.
family_name	String	Nachname.
nickname	String	Der Nickname. Das kann beispielsweise der Vorname sein.
email	String	Die E-Mail-Adresse.

Tabelle 2.2: Standard Claims.

Zudem sind sogenannte Standard Claims spezifiziert, die Informationen über den authentifizierten End-Nutzer liefern und in den Payload eingetragen werden können [38]. Einige dieser Standard Claims sind in [Tabelle 2.2](#) definiert.

2.4.2 Hybrid Flow

Der Hybrid Flow ist der „Flow“ der beschreibt, wie der Client in OpenID Connect einen Token erhalten kann. Es ist identisch mit dem Authorization Code Grant in OAuth2, nur dass hier neben einem Access Token der Client

auch einen ID-Token von dem Authorization Server erhält. Der Hybrid Flow teilt sich in die folgenden acht Schritte auf [38].

1. Der Client bereitet eine Authorization Request vor. In dieser Request müssen folgende Daten hinterlegt sein:
`RESPONSE_TYPE`: Im Falle des Authorization Code Grants muss hier `code` stehen.
`STATE`: Ein zufälliger String.
`CLIENT_ID`: Die ID des Clients.
`SCOPE`: Ein Wert, der den Geltungsbereich (scope) des Tokens angibt.
`REDIRECT_URI`: Das ist die URI des Clients. An diese URI sendet der Autorisationsserver den Authorization Code.
2. Der Client sendet die Authorization Request zu dem Authorization Server.
3. Der Authorization Server authentifiziert den End-Nutzer.
4. Der Authorization Server holt sich die Autorisation/Einwilligung des End-Nutzers ein.
5. Der Authorization Server leitet den End-Nutzer zurück zu dem Client mit einem Authorization Code.
6. Der Client sendet eine Request mit dem Authorization Code an den Token-Endpunkt des Authorization Server, um Token zu erhalten.
7. Der Client erhält einen ID-Token und Access Token von dem Token-Endpunkt des Authorization Server.
8. Der Client validiert den ID-Token und erhält den Subject Identifier des End-Nutzers.

2.5 OAUTH2 ENDPUNKTE DES AUTORISATIONSSERVERS

Der Autorisationsserver ist über eine Reihe von Endpunkten für den Client und Ressource Server ansprechbar. Über den Token Endpunkt kann der Autorisationsserver beispielsweise Token an den anfragenden Client übermitteln. Im Folgenden werden die relevanten Endpunkte erläutert.

2.5.1 *Authorization Endpunkt*

Dies ist der Endpunkt, an den der Client den End-Nutzer weiterleitet. Hier muss sich der End-Nutzer authentifizieren und gegebenenfalls den Client autorisieren, damit der Client einen Authorization Code erhält [13].

2.5.2 *Token Endpunkt*

Dies ist der Endpunkt des Autorisationsservers an den ein Client, falls er die notwendigen Informationen wie beispielsweise den Authorization Code besitzt, einen Token erhalten kann [13].

2.5.3 *JSON Web Key Set Endpunkt*

Dies ist der Endpunkt, in dem alle öffentlichen Schlüssel für die [RSA](#)-Signatur des JSON Web Token abgefragt werden können. Der Ressource Server nutzt diesen Endpunkt, um sich den passenden öffentlichen Schlüssel der Signatur des Tokens zu holen und damit die Signatur zu validieren und die Authentizität und Integrität des Tokens sicherzustellen [20].

2.6 ROLE BASED ACCESS CONTROL

Role Based Access Control ([RBAC](#)), zu Deutsch rollenbasierte Zugriffskontrolle, ist eine ANSI-NORM und sie beschreibt die Verknüpfung von Rollen, die Nutzern zugewiesen werden, mit Zugriffsprivilegien [9]. Sie ist eine Form der Zugriffskontrolle. Gemäß der Norm gibt es mehrere Definitionen:

COMPONENT: Es gibt mehrere Komponenten in der ANSI-NORM, die beschreiben wie [RBAC](#) umgesetzt werden kann.

OBJECTS: Das ist das Objekt, auf das der Zugriff beschränkt werden soll. Das kann beispielsweise ein Datenbankeintrag, ein Drucker oder eine HTTP-Schnittstelle sein.

OPERATIONS: Eine Operation ist ein ausführbares Programm, was dem Nutzer Funktionen bereitstellt.

PERMISSIONS: Das ist die Zustimmung, eine Aktion auf eine durch [RBAC](#) geschützte Ressource durchzuführen.

ROLE: Eine Rolle ist eine Jobfunktion innerhalb einer Organisation, die Autorität und Verantwortlichkeiten eines Nutzers und der zugewiesenen Rolle assoziiert.

USER: Ein Nutzer ist definiert als eine Person, allerdings kann prinzipiell ein Nutzer auch ein Objekt wie beispielsweise eine Maschine sein.

2.7 OPEN POLICY AGENT

Open Policy Agent (OPA) ist eine quelloffene Policy Engine, die Zugriffskontrolle von Applikationen entkoppelt [5]. Es entkoppelt das Treffen von Zugriffsentscheidungen von der Komponente, die Richtlinien durchsetzen muss. Als Programmiersprache um diese Zugriffskontrolle zu definieren,

wird Rego verwendet. Die Dateiendung von in Rego programmierten Zugriffsrichtlinien ist `.rego`.

Im Wesentlichen werden dem OPA-Service Input-Daten in JSON zugesendet anhand dessen eine programmierte Zugriffskontrolle in Rego evaluiert, ob Zugriff gewährleistet werden darf und entsprechend sendet der OPA-Service an das anfragende Programm eine Zugriffsentscheidung. Dadurch das OPA als eigenständiger Service in einem Docker-Container ausgeführt wird und um die Zugriffskontrolle von Systemen mit Open Policy Agent zu entkoppeln, diese lediglich eine Unterstützung für HTTP und JSON mitbringen müssen, ist Open Policy Agent praktisch system und plattformunabhängig.

Listing 2.2: Zugriffsrichtlinie in Rego

```
package application.authz

# Only owner can update the pet's information
# Ownership information is provided as part of OPA's input
default allow = false
allow {
    input.method == "PUT"
    some petid
    input.path = ["pets", petid]
    input.user == input.owner
}
```

In [Listing 2.2](#) [6] ist eine solche Zugriffskontrolle in Rego programmiert. Diese Zugriffsrichtlinie ist wie folgt zu beschreiben:

Falls eine HTTP-PUT-Anfrage auf den Pfad `/pets/petid` eintrifft, muss der User dem Owner entsprechen. Anfragen, die nicht dem Pfad `/pets/petid` entsprechen, werden abgelehnt.

Listing 2.3: Input-Daten für Open Policy Agent

```
{
  "method": "PUT",
  "owner": "bob@hooli.com",
  "path": [
    "pets",
    "pet113-987"
  ],
  "user": "alice@hooli.com"
}
```

In [Listing 2.3](#) sind Input-Daten dargestellt. Wenn Open Policy Agent diese Input-Daten erhält, würde es eine negative Zugriffsentscheidung evaluieren. Die Zugriffsentscheidung ist in [Listing 2.4](#) zu sehen.

Listing 2.4: Zugriffsentscheidung in Rego

```
{
```

```
"allow": false
}
```

Der Zugriff auf den Pfad `/pets/pet113-987` kann nicht gegeben werden, da in [Listing 2.3](#) der owner nicht dem user entspricht. `bob@hooli.com` ist ungleich zu `alice@hooli.com`.

2.8 APACHE TOMCAT

Apache Tomcat ist ein Webserver, der Spezifikationen der Java Enterprise Edition (EE) Plattform implementiert [\[17\]](#). Dieser Webserver kann HTTP-Anfragen abhören und diese entgegennehmen und beantworten. Die für diese Arbeit relevanten Spezifikationen, die Tomcat implementiert, sind Jakarta Servlet, Servlet-Container und Filter.

SERVLET: Servlets interagieren mit Webclients über ein Anfrage/Antwort-Paradigma, das vom Servlet-Container implementiert wird [\[43\]](#).

SERVLET-CONTAINER: Der Servlet-Container ist ein Teil eines Webserver oder Anwendungsservers, der die Netzwerkdienste bereitstellt über die Anfragen und Antworten gesendet werden. Ein Servlet-Container enthält und verwaltet auch Servlets während ihres Lebenszyklus [\[43\]](#).

FILTER: Ein Filter ist ein wiederverwendbarer Code, der den Inhalt von HTTP-Anfragen, Antworten und Header-Informationen umwandeln kann. Filter erzeugen im Allgemeinen keine Antwort oder antworten auf eine Anfrage, wie es Servlets tun, sondern modifizieren oder passen die Anfragen für eine Ressource an und modifizieren oder passen Antworten von einer Ressource an [\[43\]](#).

2.9 SPRING

Spring ist ein quelloffenes Java-Framework, das die Implementierung von Anwendungen vereinfacht [\[52\]](#). Es teilt sich in eine Reihe von Teilprojekte auf.

2.9.1 *Spring Boot*

Spring-Boot erleichtert die Implementierung mit dem Spring-Framework. In Spring-Boot Applikationen wird ein Apache Tomcat Webserver eingebettet [\[51\]](#). Tomcat ermöglicht es dann, HTTP-Anfragen anzunehmen und diese durch Mappings in der Applikation zu beantworten.

2.9.2 *Spring Security*

Spring Security ist ein Teilprojekt des Spring-Frameworks. Spring Security bietet Unterstützung für OAuth2 und unter anderem auch für OAuth2

Ressource Server [53]. Mittels Spring Security ist es möglich einen OAuth2 Ressource Server zu implementieren, der Zugriff auf HTTP-Schnittstellen nur ermöglicht, wenn ein valider JSON Web Token in dem Authorization Header der HTTP-Anfrage gesendet wird.

2.10 KEYCLOAK

Keycloak ist eine Implementierung des Autorisationsserver und OpenID Provider der OAuth2 und OpenID Connect-Spezifikation. Das bedeutet, dass Clients von Keycloak Access Token sowie ID Token erhalten können.

Daneben ist Keycloak auch ein Identity Provider, das heißt er ermöglicht es hinterlegten Nutzern sich zu authentifizieren. Damit ist der Authorization Code Grant und Hybrid Flow der OpenID Connect und OAuth2 Spezifikation möglich, denn es kann sich ein Nutzer authentifizieren, sodass dem Client ein Access Token und ID Token übermittelt werden kann [23].

2.11 APACHE JMETER UND METRIKEN

Apache JMeter ist ein Programm, mit dem es unter anderem möglich ist die Schnittstellen eines HTTP-Servers auf Performance zu testen. Es kann Last auf den Server erzeugen und die Latenzen der Antworten des Servers sowie andere Metriken messen und protokollieren und diese Werte in Graphen visualisieren [16]. Da zum Testen der Performance verschiedene Metriken betrachtet werden, werden diese Metriken nachfolgend definiert.

ELAPSED TIME / RESPONSE TIME: Gemäß der Definition von Apache JMeter ist die Elapsed Time, auch Response Time genannt, die Zeit von bevor dem Senden der HTTP-Anfrage an den Server bis nach dem Eintreffen des in der Regel letzten Bytes der Antwort des Servers [18].

LATENZ: Gemäß der Definition von Apache JMeter ist die Latenz die Zeit von bevor dem Senden der HTTP-Anfrage an den Server bis nach dem Eintreffen des in der Regel ersten Bytes der Antwort des Servers [18].

DATENDURCHSATZ: Der Datendurchsatz (Throughput) wird berechnet aus dem Quotienten von Anzahl der Anfragen und Zeiteinheit [18]. Es wird in dem Testplan selbst aus der ersten Anfrage bis zur letzten Anfrage berechnet mit der folgenden Formel:

$$\text{Datendurchsatz} = \frac{\text{Anzahl der Anfragen}}{\text{Gesamtzeit}} \quad (2.1)$$

CONNECT TIME: Die Connect Time ist die Zeit, die in Anspruch genommen wird, eine Verbindung mit dem Server aufzunehmen [18]. Im Falle von der Nutzung von Transmission Control Protocol (TCP) ist das der Three-Way-Handshake. TCP ist ein verbindungsorientiertes Protokoll [40].

2.12 APPLICATION PERFORMANCE INDEX

Der Application Performance Index (Apdex) ist ein offener Standard, der die Nutzerzufriedenheit hinsichtlich der Response Time in Webapplikationen berechnet [15]. Es wird ein Wert T festgelegt, der die Response Time angibt, bis zu der der Nutzer zufrieden ist. Dann gibt es einen zweiten Wert, der ein Vielfaches von T ist und angibt, bis zu welcher Länge der Response Time der Nutzer Antwortzeiten toleriert. Bei allen Werten darüber ist der Nutzer frustriert. Aus diesen Werten errechnet sich ein Ergebnis für den Application Performance Index, wobei hier 1,0 der beste Wert ist, das heißt der Nutzer ist vollständig zufrieden.

LEVEL	MULTIPLIER	TIME T
Zufrieden	$\leq T$	$\leq 1,2$ Sekunden
Toleriert	$> T, \leq 4T$	$]1,2 \text{ Sekunden}, 4,8 \text{ Sekunden}]$
Frustriert	$> 4T$	$> 4,8 \text{ Sekunden}$

Tabelle 2.3: Beispiel von Grenzwerten eines Application Performance Index.

In [Tabelle 2.3](#) sind beispielhaft Grenzwerte für einen Application Performance Index dargestellt. Wenn die durchschnittliche Response Time $\leq 1,2$ Sekunden beträgt, ist der Nutzer vollständig zufrieden. Response Times zwischen 1,2 und 4,8 Sekunden toleriert der Nutzer und bei Response Times größer als 4,8 Sekunden ist der Nutzer frustriert. Am Beispiel Webanwendungen wäre hier die Response Time die Zeit von bevor dem Senden einer Anfrage für eine HTTP-Schnittstelle, beispielsweise die GET-Anfrage, um eine Webseite darzustellen, bis zum Erreichen des letzten Bytes der Antwort des Servers, d.h. bis die Webseite bei dem Nutzer vollständig dargestellt ist.

Aus den durchschnittlichen Response Times und den festgelegten Werten T , errechnet sich der Application Performance Index.

$$Apdex_T = \frac{\text{Satisfied count} + \frac{\text{Tolerating count}}{2}}{\text{Total samples}} \quad (2.2)$$

In der obenstehenden Gleichung ist die Formel zur Berechnung des Application Performance Index dargestellt [15]. Satisfied count ist hierbei die Anzahl der Anfragen deren Beantwortung $\leq T$ dauerten und der Tolerating count diejenigen Anfragen die $\leq 4T$ dauerten, wobei diese nur eine halbe Gewichtung haben. Die Summe aus beiden Werten wird durch die Anzahl aller Anfragen geteilt. Wenn die Response Times aller Anfragen $\leq T$ betragen, errechnet sich ein perfekter Index, der 1,0 beträgt.

2.13 POSTMAN

Postman ist ein API-Client, mit dem unter anderem HTTP-Anfragen gesendet und Antworten des Servers erhalten werden können. Außerdem bietet Postman Unterstützung für OAuth2 an, das heißt es ist mittels Postman möglich, den Authorization Code Grant und Hybrid Flow abzuhandeln, um dadurch Token von einem Authorization Server zu erhalten. Hierbei ist Postman der Client der OAuth2 Spezifikation [1].

2.14 DOCKER-CONTAINER

Ein Docker-Container ist ein leichtes, eigenständiges, ausführbares Softwarepaket, das alles enthält, was zum Ausführen einer Anwendung erforderlich ist: Code, Laufzeit, Systemtools, Systembibliotheken und Einstellungen. [14].

2.15 KUBERNETES

Kubernetes ist eine Plattform zur Verwaltung von Container-Anwendungen. Es ermöglicht die Konfiguration von Container-Anwendungen sowie die Automatisierung der Bereitstellung von Container-Anwendungen. Außerdem bietet Kubernetes Funktionen zur Bereitstellung, Skalierung, Lastausgleich, Protokollierung und Überwachung der Container-Anwendungen [28]. Mit Container-Anwendungen sind in diesem Fall Docker-Container gemeint, in denen Anwendungen ausgeführt werden. Kubernetes wurde von Google 2014 als Open-Source Projekt zur Verfügung gestellt. Es gibt eine Reihe von Komponenten in Kubernetes, die nachfolgend definiert werden.

NODE Ein Node ist ein physischer Computer oder virtuelle Maschine, der als Arbeitsmaschine in einem Kubernetes Cluster dient. Auf einem Node werden Container-Anwendungen ausgeführt [24].

CLUSTER In einem Cluster können sich mehrere Nodes befinden. Kubernetes koordiniert in einem Cluster Nodes, die miteinander verbunden sind, um als eine Einheit zu arbeiten [24].

POD Ein Pod ist die kleinste Einheit, die in Kubernetes erstellt und verwaltet werden kann. In einem Pod laufen ein oder mehrere Docker-Container, die sich Speicher- und Netzwerkressourcen teilen [30].

DEPLOYMENT Ein Deployment weist an, wie Instanzen einer Anwendung erstellt und aktualisiert werden. Hierzu wird eine Deployment Konfiguration geschrieben und auf ein Kubernetes Cluster angewandt, woraufhin Kubernetes ein Pod erstellt. Darüber hinaus bietet Kubernetes für Deployments Selbstheilungsmechanismen. Wenn beispielsweise ein Node, der eine Instanz einer Applikation hostet, ausfällt, ersetzt der sogenannte Deployment Controller diese Instanz durch eine Instanz auf einem anderen Node im Cluster [25].

SERVICE Ein Service definiert Richtlinien wie auf Pods zugegriffen werden können. Es regelt also, wie ein Nutzer auf einen Pod in einem Cluster von außerhalb des Clusters zugreifen kann, als auch wie Pods miteinander innerhalb des Clusters kommunizieren können. Hierfür muss ebenfalls eine Konfiguration geschrieben werden [31].

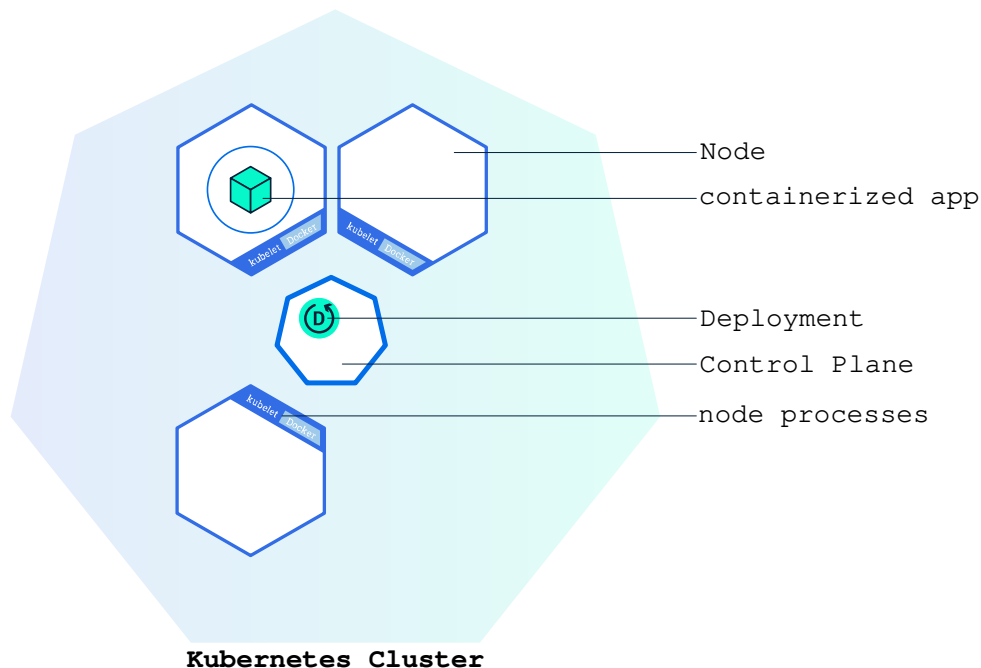


Abbildung 2.2: Kubernetes Cluster [25]

In [Abbildung 2.2](#) ist ein Kubernetes Cluster mit drei Nodes beispielhaft dargestellt. In einem Node ist zu sehen, dass dort eine kontainerisierte Anwendung läuft, die ein Docker-Container ist. Auf jedem Node im Cluster wird kubelet ausgeführt, der ein Agent ist und sicherstellt, dass Container in einem Pod ausgeführt werden. Um mit dem Cluster zu kommunizieren, um beispielsweise Pods zu starten, gibt es die Kubernetes-API und kubectl als Kommandozeilenprogramm [27].

2.16 ZUSAMMENFASSUNG DER TECHNISCHEN GRUNDLAGEN

Es wurden die nötigen technischen Grundlagen zusammengetragen. Was im nachfolgenden Kapitel folgt ist die Systemarchitektur. Beide Systeme sind OAuth2 Ressource Server, die Spring-Boot Webapplikationen sind und mit Spring-Security ihre HTTP-Schnittstellen sichern. Das bedeutet, dass diese Schnittstellen erwarten, dass ein valider JSON Web Token in dem Authorization Header der HTTP-Anfrage mitgesendet wird. Dieser JSON Web Token

wird durch Keycloak ausgestellt, in OAuth2-Vokabular ist dies der Authorization Server. In OpenID Connect Vokabular würde bei Keycloak von einem OpenID Provider gesprochen werden. In dem ID Token, der von dem Authorization Server übermittelt wird, werden Attribute des Nutzers in den Payload gemappt, unter anderem auch die Rollenzugehörigkeit. Die Spring-Boot Applikationen validieren den Token jeweils, in dem sie die Signatur des Tokens überprüfen. Dies geschieht, indem der öffentliche Schlüssel über den JSON Web Key Set Endpunkt des Authorization Server geholt und damit aus dem Header und Payload die Signatur berechnet und auf Gleichheit überprüft wird. Bei Erfolg entspricht dies einer Authentifizierung des Nutzers.

Beide Ressource Server haben als Zugriffskontrolle [RBAC](#) implementiert, das heißt in dem Token muss die Rolle `ROLE_USER` vorhanden sein, damit Zugriff auf die HTTP-Schnittstelle erfolgen kann. Allerdings wird dies in beiden Servern auf unterschiedliche Weise realisiert. In dem einen Server wurde die Zugriffskontrolle in dem Server selbst implementiert, in dem anderen Server wird diese entkoppelt durch den Open Policy Agent. Dieser wird als eigenständiges Programm in einem Docker-Container ausgeführt und die Kommunikation zwischen Ressource Server und OPA-Service geschieht auf localhost-zu-localhost Basis. Um die Performance zu testen, wird Apache JMeter verwendet. JMeter erzeugt Last auf die beiden Servern, das heißt es sendet HTTP-Anfragen mit validem Token an die Ressource Server und misst die Latenzen der Antworten. Neben der Latenz wird auch die Response Time und Connect Time gemessen und der Datendurchsatz berechnet. Außerdem werden während aller Tests die CPU-Auslastung und Arbeitsspeicherbelegung betrachtet.

Zudem wird das System mit entkoppelter Zugriffskontrolle als drittes Testsystem auch in ein Kubernetes Cluster deployed und verglichen inwiefern sich eine Nutzung von Kubernetes auf die Performance auswirkt.

ZUGRIFFSKONTROLLE UND PERFORMANCE IN OAUTH₂

Zunächst werden alle Komponenten des Systems beschrieben. Da es sich um ein OAuth₂-System handelt, gibt es alle Rollen, die auch im [Unterabschnitt 2.2.1](#) beschrieben wurden. Also Authorization Server, Ressource Server, Client und End-Nutzer.

3.1 KEYCLOAK ALS AUTHORIZATION SERVER UND IDENTITY PROVIDER

Keycloak ist eine Implementierung des Authorization Server der OAuth₂ und OpenID Connect Spezifikation. Das bedeutet, dass Keycloak unter anderem dafür zuständig ist, den Clients Access und ID Token durch den Hybrid Flow zuzusenden. Keycloak wurde in der Version 12.0.4 genutzt, und in einem Docker-Container ausgeführt. Die Tokens, die Keycloak herausgibt, sind JSON Web Token und mit RSA asymmetrisch signiert. Konkret sind sie mit RS256 signiert, das heißt die Schlüssel sind 256 Bit groß.

Neben einem Authorization Server ist Keycloak auch ein Identity Provider. Das bedeutet, dass in Keycloak Nutzer angelegt werden können und diese Nutzer auch verwaltet werden können. Damit sich Nutzer bei Keycloak authentifizieren können, musste zunächst ein Nutzer in Keycloak angelegt werden. Dies ist wichtig, damit dem Client überhaupt ein Token ausgestellt werden kann durch den Hybrid Flow. Die Charakterisierung des Nutzers ist an dieser Stelle wichtig, da die vorhandenen Attribute, die den Nutzern zugewiesen werden, auch in den Token gemappt werden und die Größe des Tokens Einfluss auf die Performance hat.

Abbildung 3.1: Keycloak Sample User

In [Abbildung 3.1](#) ist der erstellte Nutzer in Keycloak zu sehen. Neben den Standardattributen wird dem Nutzer auch eine Rolle zugewiesen, und zwar die Rolle `ROLE_USER`. Dies ist in Keycloak möglich und damit lässt sich in den Applikationen eine rollenbasierte Zugriffskontrolle realisieren. Zudem wurde dem Nutzer auch ein Nutzernamen und Passwort zugewiesen.

Abbildung 3.2: Keycloak Sample User Role

In [Abbildung 3.2](#) sind die zugewiesenen Rollen zu sehen. Die Rolle `default-roles-sample` ist eine durch Keycloak automatisch zugewiesene Rolle und hat hier keine weitere Relevanz. Außerdem wurde diesem Nutzer noch eine Gruppenzugehörigkeit und ein weiteres Attribut hinzugewiesen. Gruppenzugehörigkeiten sind in Firmen oftmals die Bürostandorte, Abteilungen und dergleichen. Nach Gruppenzugehörigkeiten wird in der Regel nicht autorisiert, denn dafür sind Rollen zuständig. Die Gruppenzugehörigkeit und das weitere Sample-Attribut hat hier nur den Sinn, dass es den Token umfangreicher macht und dies dient dazu, realistische Testbedingungen zu schaffen. Der Nutzer ist also in der Gruppe `Users` und hat als zusätzliches Attribut das statische Schlüssel-Wert-Paar `Attribute1: UserAttribute1`.

3.2 ERHALT EINES TOKENS MIT POSTMAN

Um Token durch den Hybrid Flow zu erhalten, wird ein Client und ein Nutzer benötigt, der sich für den Client authentifiziert und der Authorization Server, der den Token ausstellt. Der erstellte Nutzer wurde im vorhergehenden Kapitel gezeigt. Der End-Nutzer muss einen Browser bedienen, damit der Client über den Hybrid Flow Token erhalten kann.

3.2.1 Authorization Code Grant mit Postman

Als Client wurde der API-Client Postman verwendet, der eingebauten Support für OAuth2 besitzt und mittels eines Browsers den Hybrid Flow ausführen und damit Postman, also der OAuth2 Client, Token erhalten kann.

The screenshot shows the Postman interface with the 'Authorization' tab selected. The 'Type' is set to 'OAuth 2.0'. A note states: 'The authorization data will be automatically generated when you send the request. [Learn more about authorization](#)'. Below this, 'Add authorization data to' is set to 'Request Headers'. The main configuration area includes the following fields:

- Token Name:** Enter a token name...
- Grant Type:** Authorization Code
- Callback URL:** <https://oauth.pstmn.io/v1/callback>
- Auth URL:** <http://localhost:9080/auth/realms/sample/f...>
- Access Token URL:** <http://localhost:9080/auth/realms/sample/f...>
- Client ID:** sample-app
- Client Secret:** cb0adde7-bf8d-4705-87c4-d617c5b3ce5f
- Scope:** openid
- State:** State
- Client Authentication:** Send as Basic Auth header

The checkbox 'Authorize using browser' is checked.

Abbildung 3.3: Authorization Code Grant mit Postman

In [Abbildung 3.3](#) ist das Menü in Postman zu sehen. Um einen Token in Postman über den Hybrid Flow zu erhalten, müssen einige Daten eingegeben werden. Wie in [Unterabschnitt 2.2.2](#) beschrieben, gibt es mehrere Wege wie ein Client einen Token erhalten kann.

Hier musste als Grant Type Authorization Code angegeben werden. Dann musste die Auth URL (Authorization Endpunkt) angegeben werden. Hier wird der Browser des End-Nutzers hingeleitet, wo sich der End-Nutzer authentifizieren muss. Die Access Token URL (Token Endpunkt) ist diejenige URL, von der Postman in Austausch von einem Authorization Code einen Token erhalten kann. Den Authorization Code erhält Postman, nachdem sich der End-Nutzer bei Keycloak authentifiziert hat. Diese Endpunkte sind in Keycloak in der Admin-Konsole vorzufinden. Als Client ID und Client Secret wurden die Werte eingetragen, die bei der Registrierung des Clients in Keycloak erhalten wurden. Als Scope wurde openid angegeben, damit die notwendigen Nutzerattribute von dem End-Nutzer, der sich bei Keycloak


```
mZmxpbmVfYWVjZXNzIiwidW1hX2F1dGhvcml6YXRpb24iXX0sInJlc291cmNlX2FjY
2VzcyI6eyJhY2NvdW50Ijp7InJvbGVzIjpbIm1hbmFnZS1hY2NvdW50
IiwibWFuY2NvdW50IjpbZ291bnQtbnQtaWV3LXByb2ZpbGUuXX19LCJzY29
wZSI6Im9wZW5pZCBlbWVpYCBwcm9maWx1IiwidW1
haWxfdmVyaWZpZWQ1OmZhbHNlLCJyb2x1cyI6WyJkZWZhdWx0
LXJvbGVzLXNhbnBzZSI6IlJPTeVfVFNfUiIsIm9mZmxpbmVfYWVjZXNzIiwidW1hX2
F1dGhvcml6YXRpb24iXSwibWVtZSI6IkZpcnN0TmFtZU9mVXNlciBMVXN0TmFtZU9
mVXNlciIsImF0dHJpYnV0ZTEiOiJVC2VyQXR0cmVidXRlMSIsImdyb3VwcyI6
WyIvVXNlcnMiXSwicHJlZmVycmVkaXZvZXJ1Y2N1IjoidXNlciIsImdpdmVudX25
hbWU0iOiJGaXZdE5hbWVpZlVzZXIiLCJmYW1pbHlfbmFtZSI6Ikxhc3R0YW1lT2ZVc
2VyIiwidW1haWwiOiJlc2VyQG1haWwuY29tIn0.
```

```
b9BWNcf6ZaG-3sgX86YLu58vyk-HiZkDbLEdp5PgHEuZ6Q9omqjRsZCv4
poQZtiqWrHSbprjiwfladfpwz8sk9U0CrDs2KGd9Ev1nZrnI8JbhS3
yfixkKTDy00Cja6KGAoYzfAnc4UVwfgP7xevBrWAnbpWGfVSSStAPWBSlk0ICCHw8
Kwkrly95tggsZLpuKFi7Mr0wrKlbB9-KjKYSOqhVZ6pQCFFJ83SjZ-sj_v7tkHXX79
wf-exIgy2k64JBXViT66JKg9t33wnFEHtnnLG-nvoqGirPeRTZT0T_skmHNTg-p9
zIxN3uYPMfKJ0o1yRUjvDD4LG1WYWKUahu82Q
```

In [Listing 3.2](#) ist der Base64 kodierte Token zu sehen. Er ist in drei Teile unterteilt: Dem Header, dem Payload und der Signatur. Um ihn zu dekodieren und den Inhalt in JSON betrachten zu können, gibt es Dienste, die das Übernehmen. Auf <https://jwt.io/> ist es möglich, Base64 kodierte JSON Web Token zu dekodieren.

Listing 3.3: Dekodierter JSON Web Token

```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "ht0BHD8h1Q_CPu0LMj2TjiBJd6ukIR0uxFmimbECbsk"
}.
{
  "exp": 1626691594,
  "iat": 1626689794,
  "auth_time": 1626689528,
  "jti": "dacddb77-6de4-4464-b623-b766a8624cda",
  "iss": "http://localhost:9080/auth/realms/sample",
  "aud": "account",
  "sub": "8cf89140-e8d6-4850-b416-3cceda982ec2",
  "typ": "Bearer",
  "azp": "sample-app",
  "session_state": "c8e12b53-1203-4d8e-a27c-ea54f3f86d60",
  "acr": "0",
  "realm_access": {
    "roles": [
      "default-roles-sample",
      "ROLE_USER",
      "offline_access",
      "uma_authorization"
    ]
  }
},
```

```

"resource_access": {
  "account": {
    "roles": [
      "manage-account",
      "manage-account-links",
      "view-profile"
    ]
  }
},
"scope": "openid email profile",
"email_verified": false,
"roles": [
  "default-roles-sample",
  "ROLE_USER",
  "offline_access",
  "uma_authorization"
],
"name": "FirstNameOfUser LastNameOfUser",
"attribute1": "UserAttribute1",
"groups": [
  "/Users"
],
"preferred_username": "user",
"given_name": "FirstNameOfUser",
"family_name": "LastNameOfUser",
"email": "user@mail.com"
}.
[Signature]

```

In [Listing 3.3](#) ist der dekodierte JSON Web Token dargestellt, den Keycloak erstellt und an Postman gesendet hat. Wie zu sehen ist, sind die Nutzerattribute, das heißt die Standard Claims, in diesen Token gemappt. Also in diesem Fall der Vor-und-Nachname, die E-Mail-Adresse, der Nickname sowie das Claim sub. Zudem ist die Gruppenzugehörigkeit zu sehen, denn der Nutzer ist Mitglied in der Gruppe Users. Außerdem ist unter dem Schlüssel roles auch die Rolle ROLE_USER vorzufinden. Diese Rolle wurde dem End-Nutzer in Keycloak zugewiesen und nach dieser Rolle wird bei der Schnittstelle im Resource Server die Zugriffe kontrolliert. Das heißt in dem Token muss diese Rolle gemappt sein, ansonsten sollte kein Zugriff erlaubt werden. Die restlichen Rollen, die in diesem Token zu sehen sind wie beispielsweise default-roles-sample, wurden automatisch von Keycloak erstellt und werden hier nicht erläutert.

Außerdem sind die Claims vorzufinden, die in einem ID Token Pflicht sind wie beispielsweise exp, der angibt, wann der Token abläuft. Erklärungen zu diesen Claims sind in [Unterabschnitt 2.4.1](#) zu finden. Im Header ist unter anderem das Claim alg: RS256 vorzufinden, das angibt, dass der Token durch RSA mit 256 Bit großen Schlüssel asymmetrisch signiert ist. Der Resource Server kann sich den passenden öffentlichen Schlüssel von Keycloak über den JSON Web Key Set Endpunkt holen und damit die Authentizität

und Integrität des Tokens validieren. Nun wurde also durch den Authorization Code Grant beziehungsweise Hybrid Flow ein Token von dem Authorization Server Keycloak erhalten mit dem es möglich ist, auf die HTTP-Schnittstellen eines Ressource Servers zuzugreifen. Diese Ressource Server werden im nachfolgenden Kapitel beschrieben.

3.3 RESSOURCE SERVER

Beide Ressource Server sind in Java mithilfe des Spring-Frameworks implementiert. Die Ressource Server sind Spring-Boot Anwendungen, das bedeutet, dass ein Apache Tomcat Webserver in der Applikation eingebettet ist und automatisch gestartet wird. Die Ressource Server können somit HTTP-Anfragen von Clients entgegennehmen und diese beantworten.

Die verwendete Spring-Boot Version ist 2.4.8 mit Apache Tomcat 9.0.48, welcher über den Port 8080 erreichbar ist. Alle Versionen der verwendeten Technologien wie die Datenbank und der Object-Relational Mapper Hibernate können aus der Spring-Boot Version hergeleitet werden und werden deshalb im weiteren Verlauf nicht genannt. Als Java Runtime Environment (JRE) wurde Version 11 verwendet.

Gemäß der OAuth2 Spezifikation bezeichnet man diese Server Ressource Server, weil sie valide Tokens erwarten, die von einem Authorization Server ausgestellt werden, damit sie den Clients Zugriff auf ihre Schnittstellen geben können. Im weiteren Verlauf werden beide Ressource Server auf ihre Performance getestet. Der einzige Unterschied zwischen beiden Servern besteht darin, wie sie Zugriffsentscheidungen evaluieren, das heißt wie sie ermitteln, ob der Client berechtigt ist auf die Schnittstelle des Servers zuzugreifen. Dies wird als Zugriffskontrolle bezeichnet. Der eine Ressource Server implementiert diese Zugriffskontrolle in der Spring-Boot Applikation selbst, während der andere diese entkoppelt mit Open Policy Agent, das ein externes Programm ist. Es ist per HTTP für den Server erreichbar ist, um eine Zugriffsentscheidung zu erhalten.

Zu erwähnen ist, dass eine Implementierung von Ressource Servern grundsätzlich in jeder Programmiersprache erfolgen kann, die HTTP-, JSON-, Base64-Kodierung sowie RSA unterstützt.

3.3.1 Schnittstelle

Beide Ressource Server haben eine HTTP-GET-Schnittstelle implementiert, welche Daten aus einer SQL-Datenbank holt und sie dem anfragenden Client sendet. Dies ist genau die Schnittstelle, die nicht für alle Nutzer frei zugänglich ist, sondern nur Nutzern zur Verfügung steht, die einen validen JSON Web Token haben und der Rolle `ROLE_USER` zugehörig sind.

Listing 3.4: HTTP-Schnittstelle der Ressource Server

```
/**
```

```

* {@code GET /documents} : get all the documents.
*
* @return the {@link ResponseEntity} with status {@code 200 (OK)}
* and the list of documents in body.
*/
@GetMapping("/documents")
public List<Document> getAllDocuments() {
    log.debug("REST request to get all Documents");
    return documentService.findAll();
}

```

In [Listing 3.4](#) ist diese Schnittstelle dargestellt. Sie wird auf „/documents“ gemappt, das bedeutet, dass falls der Ressource Server auf dem localhost gestartet wird, und der Client sich ebenso auf dem localhost befindet, er diese Schnittstelle über den Pfad „localhost:8080/documents“ ansprechen kann. 8080 ist hierbei der Port, auf dem Apache Tomcat HTTP-Anfragen abhört. Diese Schnittstelle wurde in eine für Spring-Anwendungen typischen objektorientierten Architektur implementiert.

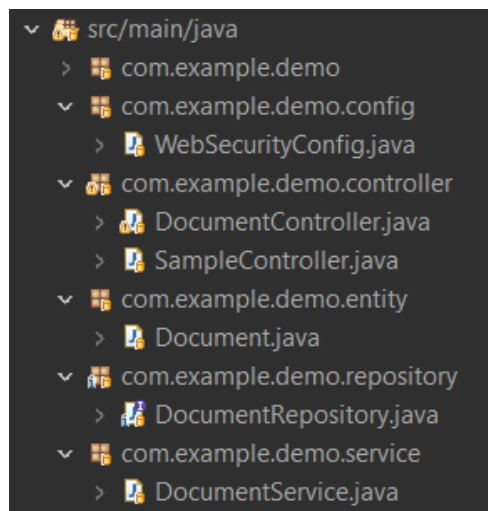


Abbildung 3.4: Projektstruktur Spring Applikation

In [Abbildung 3.4](#) ist die Projektstruktur dargestellt und die typische in Spring-Anwendungen vorzufindende geschichtete Architektur. Es gibt die Packages *.config, *.controller, *.entity, *.repository und *.service.

In dem *.controller-Package befindet sich die DocumentController-Klasse, die in [Listing 3.4](#) zu sehen ist. Diese ruft eine Funktion der DocumentService-Klasse aus dem *.service-Package auf, welche alle Daten aus dem DocumentRepository in dem Package *.repository abrufen. Dieses Repository ist ein JpaRepository. JPA steht für Jakarta Persistence API, welche eine Spezifikation für die Schnittstelle zwischen Java-Anwendungen und SQL Datenbanken ist. Als Implementierung dieser Spezifikation wird in dem Projekt der object-relational Mapper (ORM) Hibernate verwendet. Das JpaRepository implementiert unter anderem die Funktion .findAll(), welche

alle Daten aus der Document-Table aus der verbundenen SQL-Datenbank holt. Diese Funktion wird in der DocumentService-Klasse aufgerufen. In dem Package *.entity befindet sich die Klasse Document, welche die durch Hibernate annotierte Entität darstellt. Diese repräsentiert die Table Document in der SQL-Datenbank. Als Datenbank wird H2 Database Engine verwendet. Sample-Daten werden in diese Datenbank bei dem Start der Anwendung automatisch durch ein SQL-insert-Skript geladen.

Diese HTTP-GET-Schnittstelle /documents sendet autorisierten anfragenden Clients alle Document-Daten in JSON aus der SQL-Datenbank zu. In der SQL-Datenbank befinden sich lediglich zwölf Document-Daten. Jedes Document besitzt zwei Attribute, die ID und einen Namen. Das eine SQL-Datenbank mit der Applikation verbunden wurde, dient der Simulation von realistischen Testbedingungen, da in der Regel HTTP-GET-Schnittstellen Daten aus einer Datenbank holen und sie den anfragenden Clients zusenden.

3.3.2 Spring Security OAuth2

Um die Server und die Schnittstelle als OAuth2 Ressource Server zu konfigurieren, wurde Spring Security verwendet, welches ein Teilprojekt des Spring Frameworks ist. Um Zugriffe auf Schnittstellen nur Anfragen, die einen validen JSON Web Token haben zu erlauben, implementiert Spring Security eine SecurityFilterChain. Diese ist eine Implementierung der Servlet Filter Spezifikation.

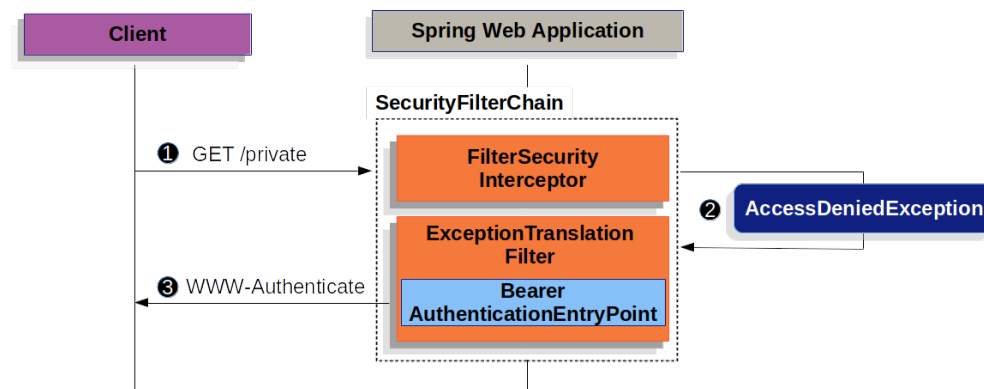


Abbildung 3.5: Bearer Token Authentication [8]

In [Abbildung 3.5](#) ist die Funktionsweise der SecurityFilterChain abgebildet. Der Client sendet eine Anfrage auf den durch OAuth2 geschützten Pfad /private der Spring Web Application. Falls die Anfrage nicht beantwortet werden darf, weil der Token des Clients invalide oder überhaupt kein Token vorhanden ist, wird eine AccessDeniedException geworfen und der Client wird aufgefordert sich zu authentifizieren. Ein valider Token, den der Client an den Server sendet, entspricht konzeptionell einer Authentifikation.

Damit die Ressource Server die Tokens validieren können, musste der JSON Web Key Set Endpunkt des Authorization Server in der Spring-Boot-

Applikation angegeben werden. Dadurch kann die Applikation sich die öffentlichen Schlüssel der RSA-Signatur von diesem Endpunkt holen und damit die RSA-Signatur validieren, indem aus dem durch Base64 kodierten Header und Payload des JSON Web Tokens die Signatur berechnet und auf Gleichheit überprüft wird. Da der Ressource Server auf derselben Host-Maschine wie Keycloak, der Authorization Server, läuft, ist der Endpunkt in diesem Fall:

```
http://localhost:9080/auth/realms/sample/protocol/openid-connect/certs
```

Zudem werden die folgenden Claims des JSON Web Tokens validiert:

exp: Der exp-Claim (Expiration Time) ist derjenige Claim der angibt, bis wann der Token gültig ist.

nbf: Der nbf-Claim (Not Before) gibt den Zeitpunkt an, ab wann der Token akzeptiert werden darf.

iss: Der iss-Claim (Issuer) gibt die Uniform Resource Identifier ([URI](#)) des Authorization Server an. Sie sollte dem Authorization Server entsprechen, von dem auch die öffentlichen Schlüssel für die RSA-Signatur geholt werden.

In beiden Ressource Servern wird eine erfolgreiche Authentifizierung, ein valider JSON Web Token, der von Keycloak erstellt wurde, erwartet. Die Autorisierung aber, das heißt die Prüfung, ob der authentifizierte Nutzer die benötigten Rechte verfügt, um Zugriff auf die Schnittstelle zu erhalten, wird in den zwei Ressource Servern auf zwei unterschiedliche Weisen realisiert und in den nachfolgenden zwei Kapiteln beschrieben.

3.3.3 Ressource Server mit interner Zugriffskontrolle

Um eine rollenbasierter Zugriffskontrolle in dem Ressource Server selbst zu realisieren, wurde JwtAuthenticationConverter verwendet, das Teil von Spring Security ist. Mit interner Zugriffskontrolle ist gemeint, dass der Server, also die Spring-Boot-Applikation, die Zugriffskontrolle selbst implementiert. Ein externes Programm wie Open Policy Agent gibt es hier nicht.

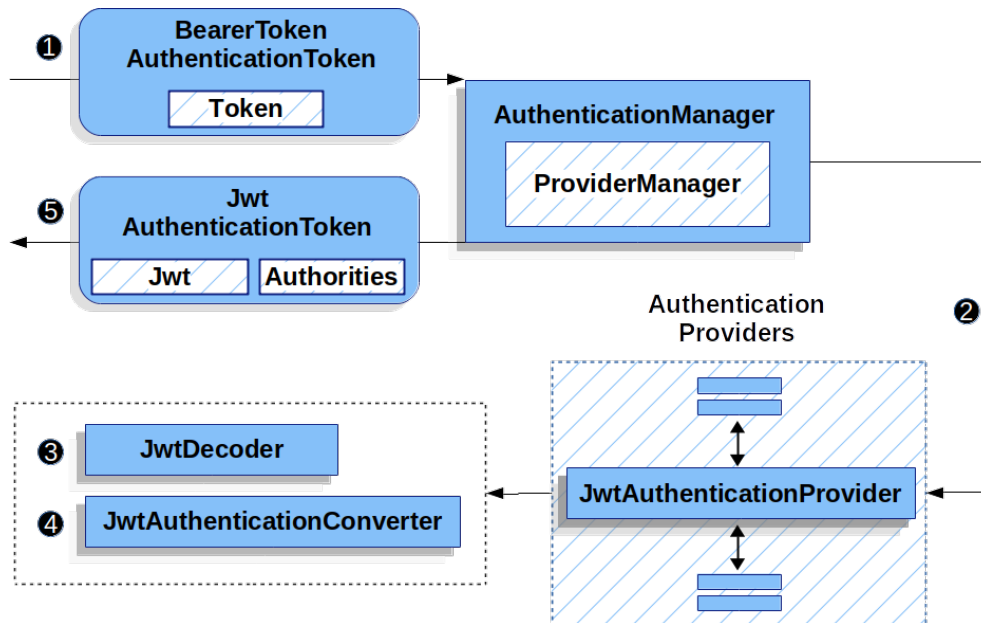


Abbildung 3.6: JWT Authentication Provider [8]

In [Abbildung 3.6](#) ist der Ablauf dargestellt, um eine rollenbasierte Zugriffskontrolle in dem Server mit dem `JwtAuthenticationConverter` zu realisieren. Im Wesentlichen wird hier der JSON Web Token durch den `JwtAuthenticationProvider` von einem `JwtDecoder` dekodiert und validiert und Autoritäten aus dem Token in eine Collection gemappt. Als `JwtDecoder` wird die Nimbus Bibliothek verwendet [8].

Der `JwtAuthenticationConverter` wurde so konfiguriert, dass er Rollen aus dem JSON Web Token in sogenannte `GrantedAuthorities` mappt. Diese sind Objekte, die Privilegien von Nutzern widerspiegeln wie beispielsweise Rollen [8].

Listing 3.5: `JwtAuthenticationConverter`

```

@Bean
public JwtAuthenticationConverter jwtAuthenticationConverter() {
    JwtGrantedAuthoritiesConverter grantedAuthoritiesConverter
        = new JwtGrantedAuthoritiesConverter();
    grantedAuthoritiesConverter.setAuthoritiesClaimName("roles");
    grantedAuthoritiesConverter.setAuthorityPrefix("");

    JwtAuthenticationConverter jwtAuthenticationConverter
        = new JwtAuthenticationConverter();
    jwtAuthenticationConverter
        .setJwtGrantedAuthoritiesConverter(grantedAuthoritiesConverter);
    return jwtAuthenticationConverter;
}
  
```

In [Listing 3.5](#) ist die verwendete Konfiguration dargestellt. Es wird in dem JSON Web Token, der von dem Client gesendet wurde, nach dem Claim `roles`

in dem Payload des Tokens gesucht und die Werte, die darin enthalten sind in GrantedAuthorities konvertiert. In dem Claim roles steht unter anderem auch die Rolle des Nutzers, der sich bei Keycloak authentifiziert hat, und zwar ROLE_USER. Um die Schnittstelle aus [Unterabschnitt 3.3.1](#) nur Nutzern zugänglich zu machen, die die Autorität ROLE_USER besitzen, wurde HttpSecurity verwendet, das Teil von Spring Security ist.

Listing 3.6: HttpSecurity

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    HTTP
        .authorizeRequests(authorize -> authorize
            .mvcMatchers("/documents").hasAuthority("ROLE_USER")
            .anyRequest().denyAll()
        )
        .oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt);
}
```

In [Listing 3.6](#) ist die verwendete Konfiguration zu sehen. Hierbei wurde konfiguriert, dass auf den Pfad /documents nur zugegriffen werden darf, wenn dem anfragenden Client die GrantedAuthority ROLE_USER zugewiesen wurde. Anfragen auf andere Schnittstellen werden grundsätzlich abgelehnt.

3.3.4 Ressource Server mit externer Zugriffskontrolle

Um die Zugriffskontrolle von dem zweiten Ressource Server zu entkoppeln, wurde der AccessDecisionManager verwendet, der Teil von Spring Security ist und es erlaubt Zugriffsentscheidungen externen Programmen zu überlassen. Dies wird auch als externe Zugriffskontrolle bezeichnet, denn ein externes Programm neben dem Server übernimmt die Zugriffskontrolle des Servers. Als externes Programm um diese Zugriffsentscheidungen zu fällen, wurde Open Policy Agent in der Version 0.30.2 verwendet, das in einem Docker-Container ausgeführt wird und auf dem gleichen Host wie der Ressource Server läuft. Der Ressource Server selbst wird als kompiliertes Java-Programm ausgeführt.

3.3.4.1 Zugriffskontrolle mit Open Policy Agent

Zunächst wird die Systemarchitektur dargestellt, die das Zusammenspiel zwischen dem Authorization Server, Keycloak, dem Ressource Server, der Spring-Boot Applikation und dem Open Policy Agent erklärt.

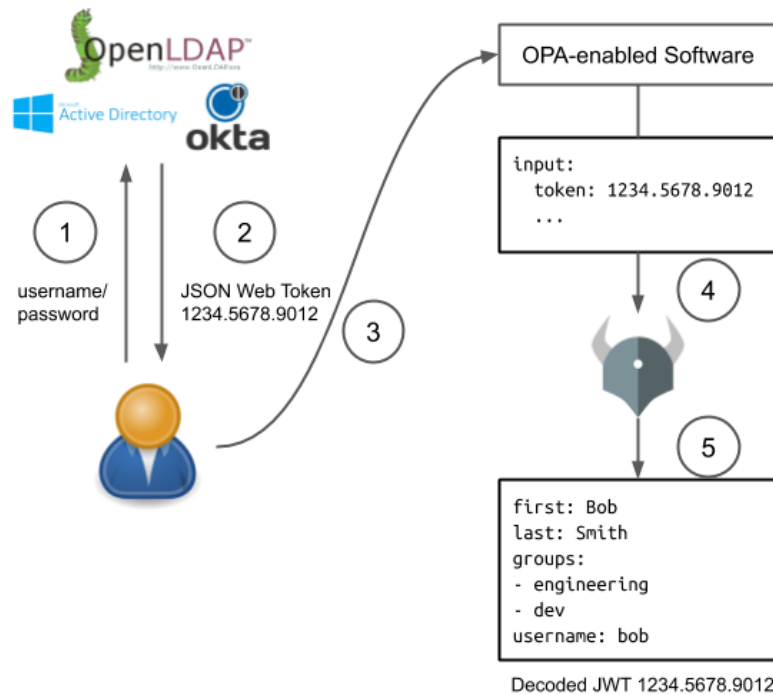


Abbildung 3.7: Open Policy Agent und OAuth2 [3]

In [Abbildung 3.7](#) ist die Systemarchitektur dargestellt. Zunächst authentifiziert sich der End-Nutzer bei dem Authorization Server. Hier ist als Beispiel Okta dargestellt, in diesem System wird allerdings Keycloak verwendet. Nachdem sich der End-Nutzer authentifiziert hat, erhält der Client einen JSON Web Token. Der Client ist Postman. Mit diesem JSON Web Token sendet der Client eine Anfrage an die OPA-enabled Software. Dies ist der Ressource Server, also die Spring-Boot Applikation. Um eine Zugriffsentscheidung zu evaluieren, muss die OPA-enabled Software nun die angefragte Schnittstelle inklusive Input-Daten an Open Policy Agent senden. Diese Input-Daten sind die JSON Web Token, in welchem unter anderem auch Nutzerattribute vorhanden sind, anhand dessen Open Police Agent mithilfe programmierter Zugriffsrichtlinien in Rego, eine Zugriffsentscheidung trifft. Diese Zugriffsentscheidung wird der OPA-enabled Software mitgeteilt, woraufhin der Client eine Antwort auf seine Anfrage erhält basierend auf der Zugriffsentscheidung von Open Policy Agent.

Listing 3.7: Rollenbasierte Zugriffsrichtlinie in Rego

```

package HTTP.authz

default allow = false

allow {
  input.method == "GET"
  input.path == ["documents"]
  token.payload.roles[_] == "ROLE_USER"
}
  
```

```

# Helper to get the token payload.
token = {"payload": payload} {
  [header, payload, signature] := io.jwt.decode(input.auth.token.
    tokenValue)
}

```

In [Listing 3.7](#) ist die verwendete programmierte rollenbasierte Zugriffskontrolle in Rego, der Programmiersprache von Open Policy Agent, um Zugriffsrichtlinien zu definieren, dargestellt. Es wird zunächst der JSON Web Token, den Open Policy Agent von dem Ressource Server erhält, dekodiert, da JSON Web Token in Base64 kodiert sind. Neben dem Token wird Open Policy Agent auch die angefragte HTTP-Schnittstelle mitgeteilt. In der ersten `allow` Methode, wird definiert, dass auf eine GET-Anfrage auf den Pfad `documents`, nur Zugriff erhalten werden darf, falls in dem Payload des JSON Web Token der Claim `roles` vorhanden ist und in diesem Claim der Wert `ROLE_USER` vorhanden ist. Falls dem so ist, wird dem Ressource Server mitgeteilt, dass der Client autorisiert ist, eine Antwort auf seine Anfrage zu erhalten. Anfragen, die nicht der Schnittstelle `documents` entsprechen, werden grundsätzlich abgelehnt.

3.3.4.2 AccessDecisionManager

Um Open Policy Agent die Zugriffsentscheidungen zu überlassen, wurde der `AccessDecisionManager` verwendet, der Teil von Spring Security ist.

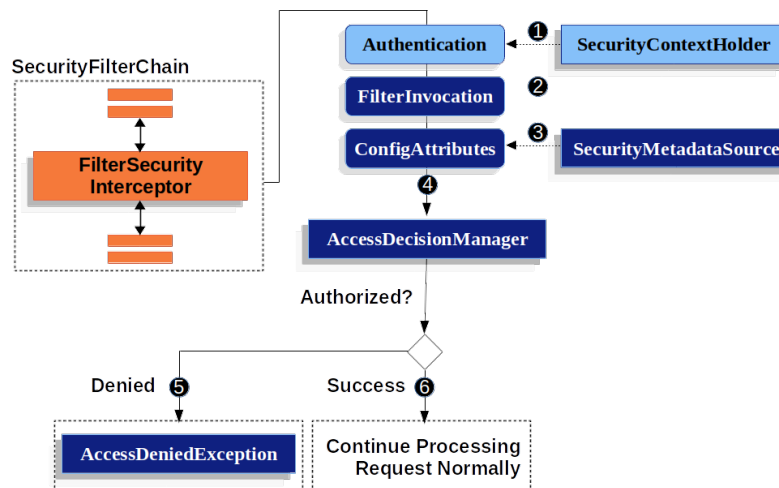


Abbildung 3.8: AccessDecisionManager in Spring Security [8]

In [Abbildung 3.8](#) ist die Funktionsweise des `AccessDecisionManagers` dargestellt. Zunächst durchläuft die ankommende Anfrage des Clients die `SecurityFilterChain`. Im Fall einer erfolgreichen Authentifikation, was im Falle von OAuth2 einer erfolgreichen Validierung des JSON Web Tokens entspricht, trifft der `AccessDecisionManager` eine Zugriffsentscheidung. In dem `AccessDecisionManager`-Objekt wurde die Funktion `vote()` implementiert,

die eine HTTP-Verbindung mit Open Policy Agent aufbaut, der auf der gleichen Host-Maschine wie die Spring-Boot Applikation läuft, um eine möglichst geringe Latenz zwischen beiden Programmen zu haben. Dies würde in den allermeisten Anwendungsszenarien auch im Produktiveinsatz so gemacht werden, da Open Policy Agent äußerst leichtgewichtig ist in der Belegung des Arbeitsspeichers. Die `vote()`-Funktion, sendet Open Policy Agent alle nötigen Informationen, damit Open Policy Agent anhand dieser Informationen mit der programmierten Zugriffsrichtlinie eine Zugriffsentscheidung treffen kann, also:

- Der Pfad auf den der Client zugreifen möchte
- Die HTTP-Methode, mit der der Client auf diesen Pfad zugreifen möchte (GET, PUT, POST, etc.)
- Die Input-Daten, also der in Base64-kodierte JSON Web Token, welcher in dem `SecurityContextHolder`-Objekt gespeichert wird

Daraufhin wird dem Ressource Server von Open Policy Agent, eine Zugriffsentscheidung per HTTP mitgeteilt. Anhand dieser Entscheidung erlaubt oder verweigert der Ressource Server dem Client den Zugriff auf die Schnittstelle.

Es wurde nun beide Ressource Server mit ihrer jeweiligen Zugriffskontrolle beschrieben.

3.4 KUBERNETES DEPLOYMENT

Da auch ein weiterer Performancetest von dem System mit OPA in einem Kubernetes Cluster durchgeführt werden soll, werden in diesem Kapitel die jeweiligen Konfigurationen von Kubernetes-Deployment und Service beschrieben. Um ein Single-Node Cluster auf einem Computer zu betreiben, wurde Minikube verwendet [48]. Single-Node Cluster bedeutet, dass es lediglich ein Node in dem gesamten Cluster gibt. Das heißt, die jeweiligen Pods laufen auf einer Maschine. Als Kubernetes Client Version wurde `v1.21.3` und als Server Version `v1.21.2` verwendet.

Listing 3.8: Deployment von Ressource Server und OPA

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ressource-server-opa
  labels:
    app: ressource-server-opa
spec:
  replicas: 2
  selector:
    matchLabels:
      app: ressource-server-opa
```

```

template:
  metadata:
    labels:
      app: ressource-server-opa
  spec:
    containers:
      - name: oauth2-opa
        image: oauth2-opa:latest
        imagePullPolicy: IfNotPresent
        ports:
          - containerPort: 8080

      - name: opa
        image: openpolicyagent/opa:0.30.2
        ports:
          - name: http
            containerPort: 8181
        args:
          - "run"
          - "--ignore=.*" # exclude hidden dirs created by Kubernetes
          - "--server"
          - "/policies"
        volumeMounts:
          - readOnly: true
            mountPath: /policies
            name: rbac-policy
    volumes:
      - name: rbac-policy
        configMap:
          name: rbac-policy

```

In [Listing 3.8](#) ist die Deployment-Konfiguration zu sehen. Es wurde konfiguriert, dass zwei Container in einem Pod gestartet werden. Der eine Container ist der Ressource Server und der andere ist OPA. Der Ressource Server wurde mithilfe des Jib-Plugins in der Version 3.1.3 containerisiert. Das bedeutet, es wurde ein Docker-Image des Ressource Servers erstellt mit dem Namen *oauth2-opa*, welches in der Deployment-Konfiguration referenziert wird unter dem Schlüssel *image*. Dieses akzeptiert Verbindungen auf den Port 8080. Damit Minikube dieses lokale Image ordnungsgemäß pullt, musste als *imagePullPolicy IfNotPresent* konfiguriert werden, da ansonsten Minikube versucht, das Image aus dem öffentlichen Docker Hub zu pullen. Um Minikube das Image aus dem lokalen Docker Registry zugänglich zu machen, wurde ein Terminal mit dem Docker Daemon von Minikube verknüpft und in diesem Terminal das Image mittels Jib erstellt. Dies ist gemäß der Dokumentation von Minikube ein performantes Vorgehen [49].

Der zweite Container ist OPA. Dieser akzeptiert Anfragen innerhalb des Clusters auf den Port 8181. Damit dieser die rollenbasierte Zugriffskontrolle aus [Listing 3.7](#) zur Verfügung gestellt bekommt, wurde eine ConfigMap für diese Zugriffskontrolle erstellt mit dem Namen *rbac-policy*. ConfigMaps sind

Objekte, in denen Schlüssel-Wert Paare gespeichert werden können, die von Pods konsumiert werden können [29]. Als Vorlage wurde die Konfiguration aus der Dokumentation von OPA verwendet [2].

Als Wert für Replicas wurde zwei gewählt. Das bedeutet, dass zwei Instanzen dieses Pods auf dem Node erstellt werden.

Listing 3.9: Service von Ressource Server und OPA

```
kind: Service
apiVersion: v1
metadata:
  name: ressource-server-opa
  labels:
    app: ressource-server-opa
spec:
  type: LoadBalancer
  selector:
    app: ressource-server-opa
  ports:
    - name: http
      protocol: TCP
      port: 8080
      targetPort: 8080
```

Die Service-Konfiguration ist in Listing 3.9 dargestellt. Es wurde konfiguriert, dass der Pod *ressource-server-opa* von außen über den Port 8080 durch das Protokoll TCP erreichbar ist. Eine Öffnung des Ports 8181 ist hier nicht notwendig und wäre sicherheitstechnisch fatal, da Nutzer nicht ohne Weiteres Zugriffsrichtlinien in den OPA-Service pushen dürfen. Es ist also lediglich der Ressource Server für Nutzer außerhalb des Clusters zugänglich.

Als Typ für den Service wurde LoadBalancer gewählt. Dies bedeutet, dass Anfragen durch den LoadBalancer an den korrekten Port des Pods weitergeleitet werden und Anfragen gleichmäßig auf alle verfügbaren Instanzen des Pods verteilt werden [26].

Die Deployment und Service-Konfiguration, um Keycloak ebenfalls innerhalb des Clusters auszuliefern, ist in Anhang A vorzufinden. Hierbei wurden Konfigurationen aus der Dokumentation von Keycloak [22] sowie PostgreSQL-Konfigurationen von Shanika Wickramasinghe [45] als Vorlage benutzt und angepasst. Die PostgreSQL Datenbank ist für Keycloak notwendig, damit Keycloak Einstellungen und Nutzer persistieren kann.

Um schließlich die Services des Clusters von außerhalb des Clusters zugänglich zu machen, wurde minikube tunnel verwendet.

3.5 PERFORMANCETESTS MIT APACHE JMETER

Um Performancetests auf beiden Ressource Server durchzuführen und die Ergebnisse vergleichend auszuwerten, um den Einfluss von externer Zu-

griffskontrolle mit Open Policy Agent auf OAuth2 Ressource Server zu untersuchen, wurde Apache JMeter in der Version 5.4.1 verwendet. Mit Apache JMeter ist es möglich durch die Verwendung von Threads, Nutzer zu simulieren, die HTTP-Anfragen an einen Server senden. In diesen HTTP-Anfragen wird jeweils ein valider JSON Web Token an die Ressource Server gesendet und diese evaluieren jeweils eine Zugriffsentscheidung. Es wird gemessen, wie lange die Ressource Server jeweils für eine Evaluierung einer Zugriffsentscheidung und das darauffolgende Senden der Antwort benötigen.

Wenn ein Single Sign-On mittels OpenID Connect implementiert wird, senden eingeloggte Nutzer bei jeder Anfrage den JSON Web Token, den sie nach der Authentifikation von dem Authorization Server erhalten haben, an den Ressource Server, um Zugriff auf dessen Schnittstellen zu erhalten. Bei den Performancetests werden also eingeloggte Nutzer simuliert, die auf gesicherte Schnittstellen eines Servers zugreifen möchten. Jeder Thread entspricht einem eingeloggten Nutzer.

Es wurden drei Testpläne erstellt, die verschiedene Aspekte der Server testen, wobei alle drei unter dem Überpunkt der Performance gehören. Die drei Testpläne werden als Lasttest, Skalierbarkeitstest und Stresstest bezeichnet.

LASTTEST: Bei dem Lasttest wird der Server mit üblicher Last getestet und die Latenz gemessen.

SKALIERBARKEITSTEST: Bei dem Skalierbarkeitstest wird ansteigende Last auf dem Server generiert, wobei hierbei untersucht werden soll, inwiefern sich Antwortzeiten des Servers bei hinzukommender Last ändern.

STRESSTEST: Bei dem Stresstest wird eine unüblich hohe Last auf den Server generiert, um die Grenzen des Servers feststellen zu können.

Bei jedem Test wird auch die CPU-Auslastung und Arbeitsspeicherbelegung betrachtet. Alle Tests wurden verteilt gestartet, das bedeutet, dass Client (Apache JMeter) und Server auf jeweils einem Rechner laufen. Dies dient dem Zweck, dass die Testergebnisse nicht verfälscht werden, da bei einer hohen Anzahl von Threads sowohl das Senden von HTTP-Anfragen CPU-Last erzeugt als auch das Abarbeiten der Anfragen durch den Server und beide Prozesse sich gegenseitig nicht ausbremsen sollten. Die verwendete Systemhardware von Client und Server, ist in [Abschnitt 3.6](#) zu sehen.

3.5.1 Lasttest

In dem Lasttest wurde eine übliche Last auf den Server generiert und mittels sogenannten Listener die Ergebnisse gemessen und protokolliert.

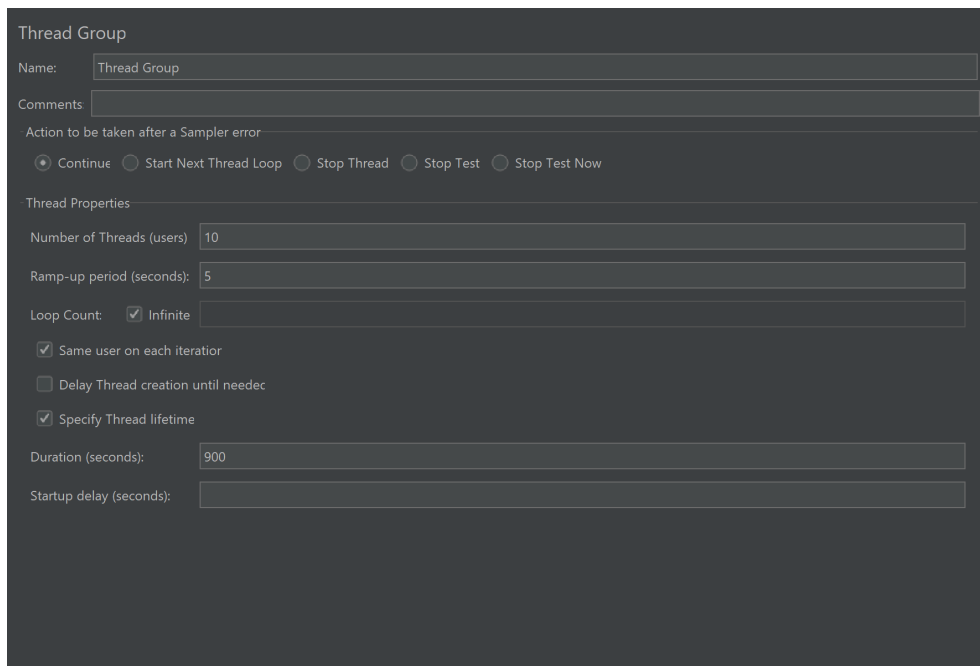


Abbildung 3.9: Thread Group-Last

In [Abbildung 3.9](#) ist die verwendete Konfiguration dargestellt. Es werden zehn Threads gestartet, die jeweils gleichzeitig HTTP-Anfragen an den Server senden. Als Ramp-up period wurde fünf Sekunden gesetzt. Das bedeutet, es dauert fünf Sekunden, bis alle zehn Threads gestartet sind. Als Duration ist 900 Sekunden eingestellt worden, das bedeutet, dass der Test 15 Minuten dauert.

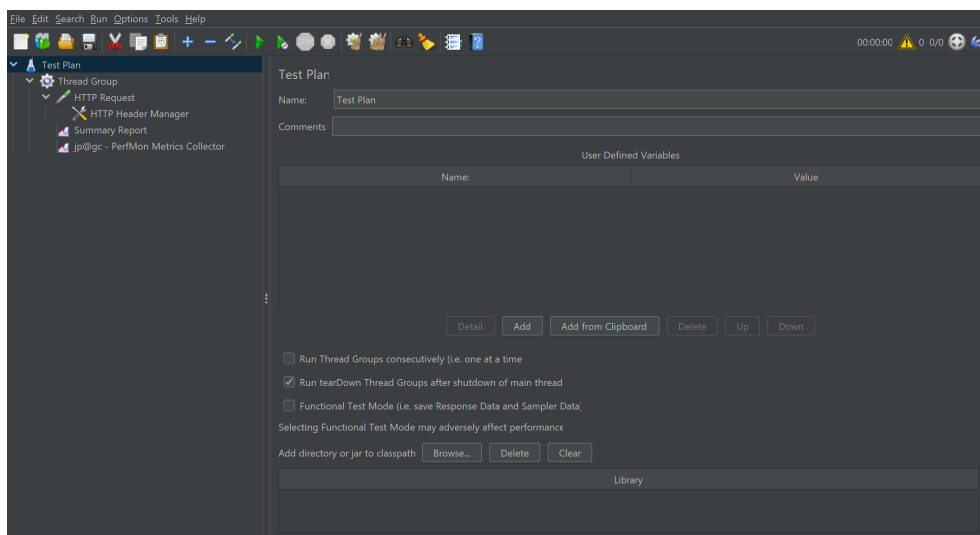


Abbildung 3.10: Test Plan-Last

In [Abbildung 3.10](#) ist eine Gesamtübersicht des Testplans dargestellt. In Thread Group wurde die Konfiguration vorgenommen, die in [Abbildung 3.9](#) dargestellt ist. Es wurden zwei Listener konfiguriert, nämlich *Summary Re-*

port und *jp@gc – Performance Metrics Collector. Summary Report* ist für das Protokollieren der Messdaten zuständig. Hier werden unter anderem die Latenz, Response Time und Connect Time protokolliert. Der Listener *jp@gc – Performance Metrics Collector* ist für das Messen von CPU-Auslastung und Random-Access Memory (RAM)-Belegung zuständig. Dieser erhält per TCP kontinuierlich Informationen über die momentane CPU-Auslastung sowie RAM-Belegung von dem Server und stellt sie graphisch dar [37].

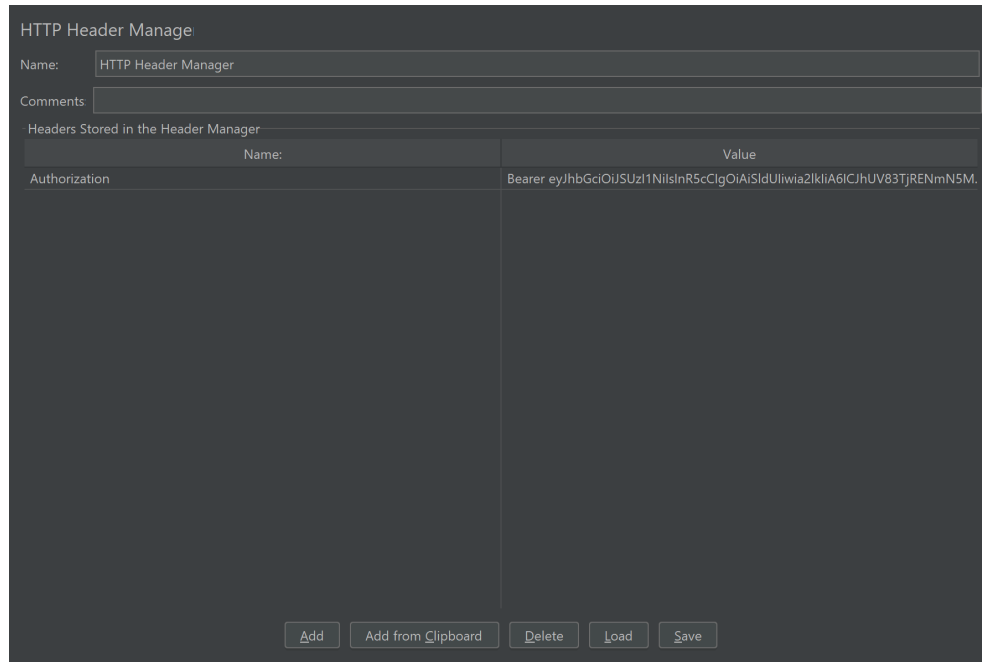


Abbildung 3.11: HTTP Header Manager

In [Abbildung 3.11](#) ist der verwendete HTTP Header Manager zu sehen. Hier wurde lediglich der JSON Web Token in dem Schlüssel Authorization des HTTP-Headers hinterlegt. Als Prefix wird Bearer angegeben, um dem empfangenden System zu signalisieren, dass in dem Authorization-Feld des HTTP-Headers ein Bearer Token steht, das heißt ein durch OAuth2 erlangter Token [13].

HTTP Request

Name: HTTP Request

Comments:

Basic Advanced

Web Server

Protocol [http]: Server Name or IP: 192.168.0.104 Port Number: 8080

HTTP Request

Method: GET Path: documents Content encoding:

☐ Redirect Automatically ☒ Follow Redirects ☒ Use KeepAlive ☐ Use multipart/form-data ☐ Browser-compatible headers

Parameters Body Data Files Upload

Send Parameters With the Request

Name:	Value	URL Encode?	Content-Type	Include Equals?
-------	-------	-------------	--------------	-----------------

Detail Add Add from Clipboard Delete Up Down

Abbildung 3.12: HTTP Request

In [Abbildung 3.12](#) ist die HTTP-Anfrage zu sehen, die jeweils an den Server gesendet wird. Hier wurde die Anfragemethode auf GET eingestellt und der Pfad auf documents gesetzt, denn die Schnittstelle des Ressourcen-Server ist eine HTTP-GET-Schnittstelle. In dem Feld *Server Name or IP* wurde die IP-Adresse des Servers angegeben.

3.5.2 Skalierbarkeitstest

Bei dem Skalierbarkeitstest wird eine ansteigende Last auf den Server generiert, um zu untersuchen, inwiefern sich die Latenzen verändern. Je mehr Last erzeugt wird, desto höher werden tendenziell die Latenzen.

The screenshot shows the 'Thread Group' configuration window in Apache JMeter. The 'Name' field is set to 'Thread Group'. The 'Comments' field is empty. Under 'Action to be taken after a Sampler error', the 'Continue' radio button is selected. The 'Thread Properties' section includes: 'Number of Threads (users)' set to 100, 'Ramp-up period (seconds)' set to 300, 'Loop Count' set to 'Infinite' with the checkbox checked, 'Same user on each iteration' checked, 'Delay Thread creation until needed' unchecked, and 'Specify Thread lifetime' checked. The 'Duration (seconds)' is set to 300, and the 'Startup delay (seconds)' field is empty.

Abbildung 3.13: Thread Group-Skalierung

In [Abbildung 3.13](#) ist die verwendete Konfiguration zu sehen. Es werden 100 Threads gestartet, wobei hier die Ramp-up period 300 beträgt, das bedeutet das erst nach 300 Sekunden alle 100 Threads gestartet wurden. Es werden pro Sekunde $300/100 = 3$ Threads gestartet. Die Gesamtdauer des Tests beträgt 300 Sekunden, das heißt 5 Minuten. Was dadurch ermöglicht wird, ist das bis zum Ende des Tests eine ansteigende Last auf den Server entsteht. Dadurch lässt sich betrachten, inwiefern sich die Latenzen beziehungsweise Response Times der Anfragen bei ansteigender Last verhalten.

3.5.3 *Stresstest*

Das Ziel des Stresstests ist es, eine möglichst hohe Last auf den Server zu erzeugen, um die Grenzen des Systems auszuloten. Es ist wichtig zu wissen, wie viele Nutzer ein System gleichzeitig bedienen kann, bis das System entweder unerreichbar wird oder die Latenzen der Antworten unakzeptabel hoch sind, damit dem zum geeigneten Zeitpunkt durch Skalierung des Servers entgegengewirkt werden kann. Zudem können durch diesen Test, Schwachstellen der jeweiligen Implementierungen wie beispielsweise Memory Leaks, erkannt werden.

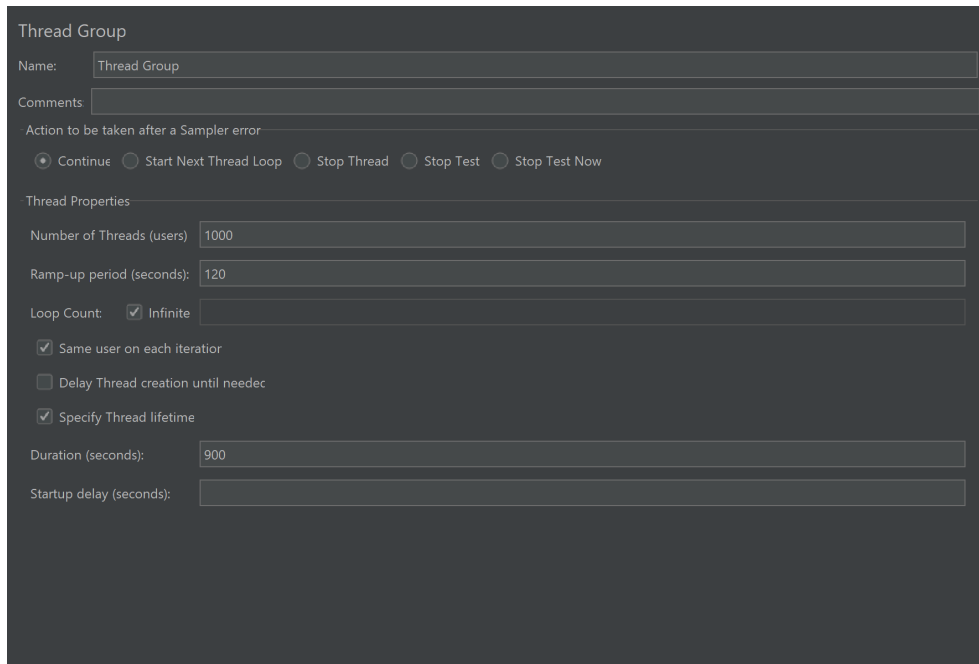
The image shows the 'Thread Group' configuration window in Apache JMeter. It has a dark grey background with white text and input fields. The 'Name' field is set to 'Thread Group'. Below it is a 'Comments' field. The 'Action to be taken after a Sampler error' section has five radio buttons: 'Continue' (selected), 'Start Next Thread Loop', 'Stop Thread', 'Stop Test', and 'Stop Test Now'. The 'Thread Properties' section contains several settings: 'Number of Threads (users)' is 1000, 'Ramp-up period (seconds)' is 120, 'Loop Count' is 'Infinite' (checked), 'Same user on each iteration' is checked, 'Delay Thread creation until needed' is unchecked, 'Specify Thread lifetime' is checked, 'Duration (seconds)' is 900, and 'Startup delay (seconds)' is empty.

Abbildung 3.14: Thread Group-Stress

In [Abbildung 3.14](#) ist die verwendete Konfiguration zu sehen. Es werden insgesamt 1000 Threads gestartet wobei alle 1000 Threads erst nach 120 Sekunden gestartet sind. Der Test dauert insgesamt 900 Sekunden, also 15 Minuten. Während dieser Zeit sendet jeder Thread kontinuierlich Anfragen an den Server. Dieser Test ist deutlich ressourcenintensiver als der Last- und Skalierbarkeitstest. Verlässlich können pro Maschine in Apache JMeter ca. 1000-2000 Threads gestartet werden, was allerdings abhängig ist von der jeweiligen Hardware des Systems, auf dem Apache JMeter ausgeführt wird [19]. Dieser Test wird ohne die GUI von Apache JMeter ausgeführt, da diese speziell bei ressourcenintensiven Stresstests die Testergebnisse verfälschen kann.

3.5.4 Messung und Protokollierung von Messdaten

Um die Messdaten der Tests aus den vorangegangenen Kapiteln zu protokollieren und in geeigneter Weise zu visualisieren, wurden Apache JMeter Listener verwendet, die Messdaten in Comma-separated values (CSV)-Dateien schreiben. Um diese Daten visuell darzustellen, wurde der HTML Report von Apache JMeter verwendet. Als Listener wurde jeweils Summary Report verwendet. Dieser misst und protokolliert unter anderem die folgenden Daten:

- Response Time
- HTTP-Status Code der Antwort des Servers
- Latenz

- Connect Time
- Gesendete Bytes
- Erhaltene Bytes

Für die Bestimmung des Application Performance Index wurden folgende Werte gewählt:

LEVEL	MULTIPLIER	TIME T
Zufrieden	$\leq T$	≤ 500 Millisekunden
Toleriert	$> T, \leq 3T$]0,5 Sekunden, 1,5 Sekunden]
Frustriert	$> 3T$	$> 1,5$ Sekunden

Tabelle 3.1: Application Performance Index.

Das bedeutet, dass Nutzer bis zu einer Response Time von 500 Millisekunden zufrieden sind. Response Times bis 1500 Millisekunden tolerieren Nutzer und bei Response Times ab 1500 Millisekunden sind Nutzer frustriert. Der Multiplier ist hier $3T$. Aus diesen Werten wird der Performance Index berechnet.

3.6 SYSTEMHARDWARE UND TESTUMFELD

Es wird die verwendete Systemhardware für Client und Server, die in allen drei Tests verwendet wurde, aufgelistet. Dies ist erwähnenswert, da die Messwerte von der Leistungsfähigkeit der Hardware abhängen. Da allerdings ein Vergleich zwischen zwei Systemen gezogen wird, können auch allgemeingültige Erkenntnisse aus den Ergebnissen der Tests gezogen werden. Ein weiterer wichtige Punkt ist der, dass beide Rechner, Client und Server, sich in demselben Netzwerk befinden und über denselben Router über Ethernet mit dem Internet verbunden sind.

Open Policy Agent wird in einem Docker-Container ausgeführt und für die Simulation der virtuellen Maschine des Docker-Containers wird Windows-Subsystem für Linux 2 ([WSL 2](#)) verwendet. Für die Ressourcenbereitstellung des Docker-Containers, wurden die Standardeinstellungen verwendet. Diese sind in der Dokumentation von Microsoft vorzufinden [34]. Das bedeutet, der Docker-Container kann alle Prozessorkerne verwenden und ist bei der Arbeitsspeicherbelegung auf 8 Gigabyte begrenzt.

KOMPONENTE	SPEZIFIKATION
CPU	Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz
Arbeitsspeicher	16 Gigabyte
Betriebssystem	Microsoft Windows 10 Pro, 64 Bit Version 10.0.19042 Build 19042

Tabelle 3.2: System des Clients.

In [Tabelle 3.2](#) ist das System des Clients dargestellt. Auf diesem System wird Apache JMeter ausgeführt, es sendet jeweils die HTTP-Anfragen an den Server.

KOMPONENTE	SPEZIFIKATION
CPU	Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
Arbeitsspeicher	16 Gigabyte
Betriebssystem	Microsoft Windows 10 Education, 64 Bit Version 10.0.19042 Build 19042

Tabelle 3.3: System des Servers.

In [Tabelle 3.3](#) ist das System des Servers dargestellt. Hier laufen die jeweiligen Ressource Server, die die Anfragen von Apache JMeter beantworten.

AUSWERTUNG DER PERFORMANCETESTS

In diesem Kapitel werden die Performancetests, die mit Apache JMeter mit den jeweiligen Testplänen durchgeführt wurden, ausgewertet. Es wurden drei Testpläne durchgeführt, Lasttest, Skalierbarkeitstest sowie der Stress-test und zu jedem Test folgt eine Auswertung. Diese Auswertung wird vergleichend betrachtet, da es zwei Ressource Server gibt, auf denen die Tests jeweils angewandt wurden, nämlich einmal der Ressource Server, der die Zugriffskontrolle mit Open Policy Agent entkoppelt und der zweite Resource Server, der die Zugriffskontrolle in dem Server selbst implementiert. Der HTML-Report, der von Apache JMeter aus den Messwerten in der csv-Datei erstellt wird, gibt eine Reihe von Grafiken und Statistiken aus anhand dessen die Auswertung geschieht.

Es ist wichtig zu erwähnen, dass die Messergebnisse in einem Testumfeld zustande gekommen sind, in dem sich Client und Server in demselben Netzwerk befinden. Dadurch entsteht zwischen beiden Systemen durch die Entfernung beider Systeme zueinander keine nennenswerte Latenz. Typischerweise befinden sich im Praxiseinsatz Client und Server nicht in demselben Netzwerk, sondern sie befinden sich in unterschiedlichen Netzwerken und sind geographisch über eine gewisse Entfernung voneinander entfernt. Dadurch entsteht eine entfernungsabhängige Latenz bei der Kommunikation zwischen Systemen, diese ist hier bei den Messergebnissen nicht mitinbegriffen. Das Anpingen von handelsüblichen deutschen Webseiten, ergibt eine Latenz von ca. 20 Millisekunden, wobei auch dies stark von der Internetverbindung abhängt. Eine spezifische Latenz kann auf die Messergebnisse in den nachfolgenden Kapiteln hinzugerechnet werden.

4.1 LASTTEST

In diesem Test wurden zehn Threads, die jeweils kontinuierlich HTTP Anfragen mit validen JSON Web Token an den Server senden, über einen Zeitraum von 15 Minuten gestartet. Es stellte es sich heraus, dass der Server, der die Zugriffskontrolle mit Open Policy Agent entkoppelt, eine durchschnittlich 3,36-mal höhere Response Time aufweist im Vergleich zu dem Server, der die Zugriffskontrolle nicht entkoppelt.

Zunächst werden die Messergebnisse von der Spring-Boot Applikation dargestellt, die eine rollenbasierte Zugriffskontrolle mittels dem JwtAuthenticationConverter realisiert hat. Dies ist der Server, der die Zugriffskontrolle in dem Server selbst implementiert.

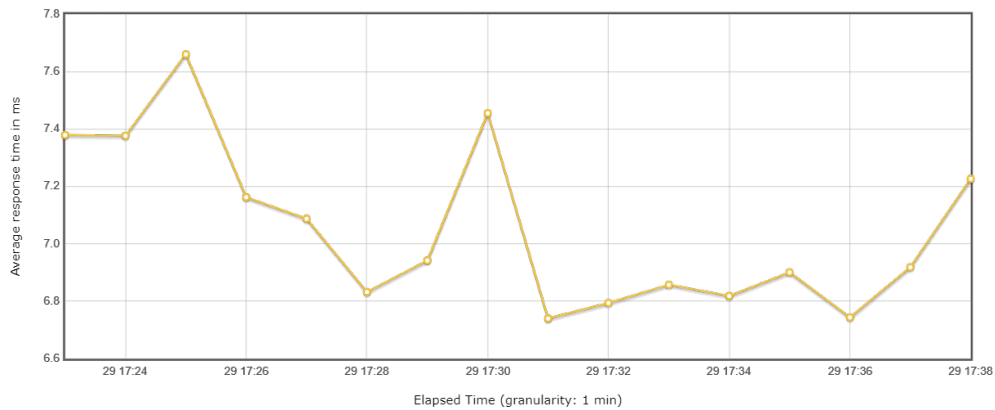


Abbildung 4.1: Response Time Graph von Lasttest von Ressource Server ohne OPA

In [Abbildung 4.1](#) ist ein Diagramm dargestellt, das die durchschnittliche Response Time von dem Ressource Server ohne Open Policy Agent über den gesamten Zeitraum des Lasttests darstellt. Auf der x-Achse ist die Zeit und auf der y-Achse die durchschnittliche Response Time zu dem jeweiligen Zeitpunkt. Es ist zu sehen, dass die Response Time anfangs höher ausfällt und sie kontinuierlich sinkt, bis sie auf ungefähr dem gleichen Niveau verbleibt. Das könnte damit erklärt werden, dass jeder Thread zunächst eine Verbindung durch den 3-Way-Handshake mit dem Server aufbauen muss, welcher zusätzliche Zeit in Anspruch nimmt. Sobald die Verbindung aufgebaut ist, fällt die durchschnittliche Response Time niedriger aus. Über den gesamten Zeitraum bewegen sich aber die Response Times im Bereich zwischen 7,8 Millisekunden und 6,6 Millisekunden.

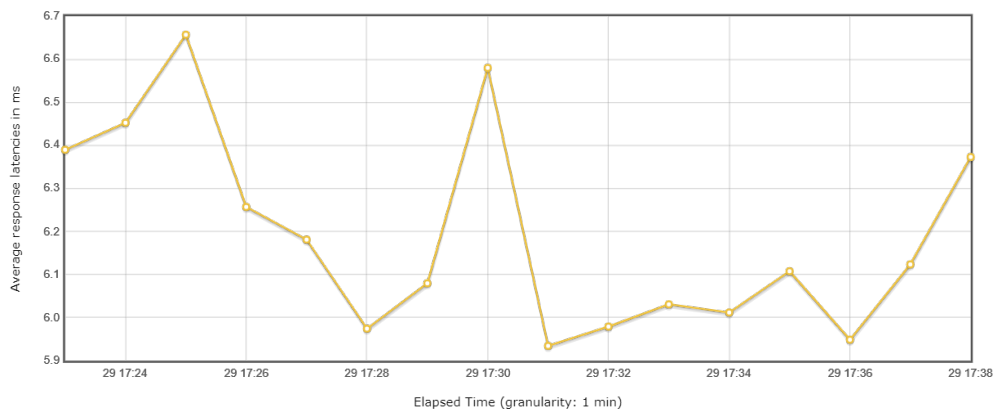


Abbildung 4.2: Latency Time Graph von Lasttest von Ressource Server ohne OPA

In [Abbildung 4.2](#) ist prinzipiell der gleiche Graph dargestellt, nur das hier anstatt der Response Time die Latenz dargestellt ist. Der Unterschied ist hier, wie in [Abschnitt 2.11](#) erläutert, dass die Latenz die Zeit von bevor dem Senden der Anfrage bis zum Eintreffen des ersten Bytes der Antwort darstellt, während die Response Time die Zeit bis zum letzten Byte der Antwort misst. Die Latenz ist also immer kürzer als die Response Time. Hier ist zu sehen das zu dem Zeitpunkt 17:38 die Latenz 6,4 Millisekunden beträgt, während

zum selben Zeitpunkt die Response Time 7,2 Millisekunden beträgt. In den folgenden Betrachtungen wird allerdings nur noch auf die Response Time eingegangen, da sie die wichtigere Metrik aus Nutzersicht ist, da ein Nutzer eine vollständige Antwort von dem Server haben möchte. Es ist keine zweckmäßige Beurteilung von Antwortzeiten möglich, wenn das erste Byte der Antwort des Servers zeitnah eintrifft, bis zum Eintreffen des letzten Bytes allerdings zwei Stunden vergehen.

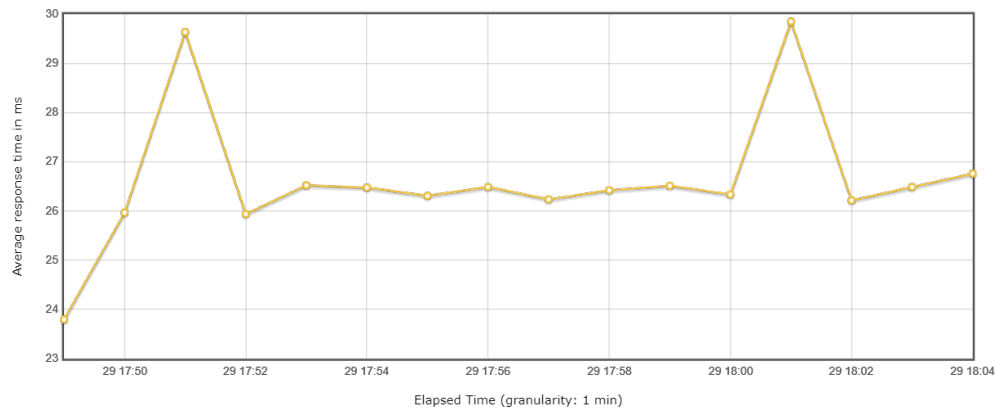


Abbildung 4.3: Response Time Graph von Lasttest von Ressource Server mit OPA

In [Abbildung 4.3](#) ist der Response Time Graph von dem Server dargestellt, der die Zugriffskontrolle mittels Open Policy Agent entkoppelt hat. Auch hier ist die Response Time anfangs höher, bis sie sich auf ein niedrigeres Level einpendelt. Aber das weitaus wichtigere ist, dass die Response Times hier durchschnittlich um ein Vielfaches höher ausfallen als bei dem Server mit interner Zugriffskontrolle. Die meiste Zeit über bewegen sich die Response Times hier zwischen 26 und 30 Millisekunden, während sie bei dem Server mit interner Zugriffskontrolle sich zwischen 6,4 und 7,2 Millisekunden bewegen.

Dies ist insofern überraschend, da die Kommunikation, die zwischen Resource Server und Open Policy Agent auftritt, auf localhost zu localhost Basis verläuft. Das heißt der Resource Server und Open Policy Agent laufen auf derselben Maschine und es war nicht vorhersehbar, dass durch die HTTP-Kommunikation zwischen diesen beide Komponenten eine derart hohe zusätzliche Latenz entsteht.

Requests	Executions			Response Times (ms)							Throughput		Network (KB/sec)	
	Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total		1255959	0	0.00%	7.03	3	2184	7.00	10.00	11.00	15.00	1395.50	952.20	2329.02
HTTP Request		1255959	0	0.00%	7.03	3	2184	7.00	10.00	11.00	15.00	1395.50	952.20	2329.02

Abbildung 4.4: Statistiken von Lasttest von Ressource Server ohne OPA

In [Abbildung 4.4](#) ist abschließend noch eine statistische Auswertung des Lasttests für den Resource Server ohne Open Policy Agent zu sehen. Die durchschnittliche Response Time beträgt 7.03 Millisekunden und innerhalb von 15 Minuten wurden 1255959 Anfragen bearbeitet.

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label ^	#Samples ^	FAIL ^	Error % ^	Average ^	Min ^	Max ^	Median ^	90th pct ^	95th pct ^	99th pct ^	Transactions/s ^	Received ^	Sent ^
Total	334421	0	0.00%	26.69	10	1204	26.00	33.00	36.00	42.99	371.58	253.54	620.14
HTTP Request	334421	0	0.00%	26.69	10	1204	26.00	33.00	36.00	42.99	371.58	253.54	620.14

Abbildung 4.5: Statistiken von Lasttest von Ressource Server mit OPA

In [Abbildung 4.5](#) ist die statistische Auswertung des Lasttests für den Ressource Server mit entkoppelter Zugriffskontrolle dargestellt. Hier beträgt die durchschnittliche Response Time 26.69 Millisekunden. Erwähnenswert ist, dass in diesem Test der Server lediglich 334421 Anfragen innerhalb 15 Minuten beantworten konnte. Die durchschnittliche Response Time ist um den Faktor drei höher und entsprechend können auch um etwa um den Faktor drei weniger Anfragen beantwortet werden.

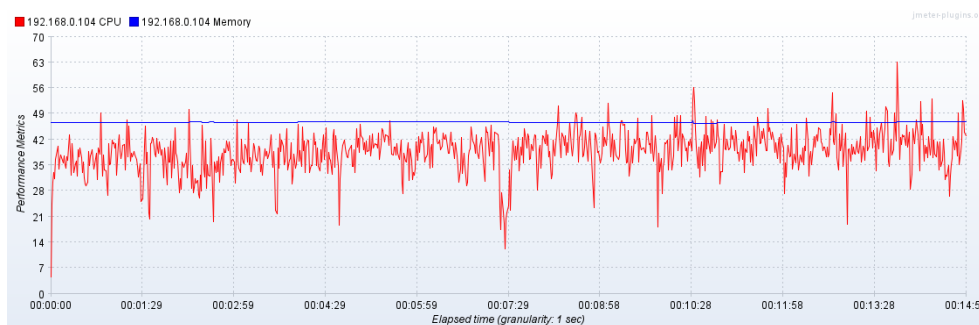


Abbildung 4.6: Perfmon von Lasttest von Ressource Server ohne OPA

In [Abbildung 4.6](#) ist die CPU-Auslastung während des Lasttests für den Ressource Server ohne entkoppelte Zugriffskontrolle zu sehen. Dies ist die rote Linie. Die CPU-Auslastung bewegt sich im Bereich von 42%. Die RAM-Belegung bleibt konstant, dies ist durch die blaue Linie gekennzeichnet.



Abbildung 4.7: Perfmon von Lasttest von Ressource Server mit OPA

In [Abbildung 4.7](#) ist die CPU-Auslastung sowie RAM-Belegung während dem Lasttest des Ressource Servers mit entkoppelter Zugriffskontrolle zu sehen. Hier ist zu sehen, dass die CPU-Auslastung deutlich höher ausfällt. Die Zeitachse auf beiden Abbildungen haben jeweils die gleiche Skalierung. Die CPU-Auslastung beträgt hierbei größtenteils ca. 91%. Dies ist damit zu erklären, dass Open Policy Agent die JSON Web Token, die durch eine

HTTP-Verbindung zwischen Server und Open Policy Agent erhalten werden, dekodieren und parsen muss, um eine Zugriffsentscheidung zu treffen. Dieser Vorgang ist anscheinend ressourcenintensiv. Die regelmäßigen Einbrüche der CPU-Auslastung auf unter 50% sind schwer zu erklären. Die RAM-Belegung entspricht derselben wie in [Abbildung 4.6](#). Sie bleibt konstant, Memory Leaks und dergleichen treten nicht auf.

In beiden Servern wurden alle Anfragen erfolgreich bearbeitet und es kamen keine Fehlermeldungen wie Unerreichbarkeit des Servers vor. Eine erfolgreiche Bearbeitung entspricht hierbei, dass die Server den Token validiert haben und entsprechend erkannt wurde, dass der Nutzer autorisiert ist, eine Antwort von dem Server zu erhalten. Dies entspricht in diesem Fall dem HTTP-Code 200 (OK).

4.2 SKALIERBARKEITSTEST

In diesem Test wurde eine ansteigende Last auf bis zu 100 Threads auf die jeweiligen Server erzeugt.

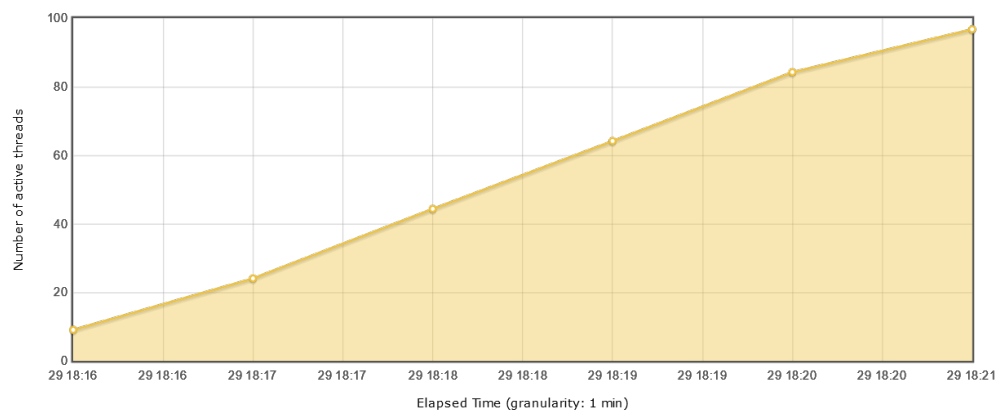


Abbildung 4.8: Aktive Threads über Zeitraum des Tests

In [Abbildung 4.8](#) ist die Anzahl der aktiven Threads über den Zeitraum des Tests dargestellt. Es werden bis zu 100 Threads gestartet und die Anzahl der Threads nimmt kontinuierlich zu. Es wird also eine zunehmende Last auf den Server erzeugt.

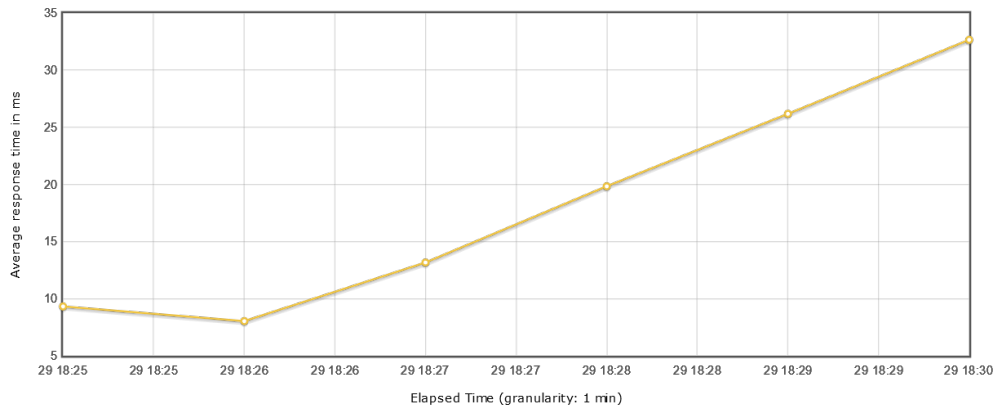


Abbildung 4.9: Response Time Graph von Skalierbarkeitstest von Ressource Server ohne OPA

In [Abbildung 4.9](#) ist ersichtlich, dass bei dem Ressource Server mit interner Zugriffskontrolle die Response Times ab einen gewissen Zeitpunkt kontinuierlich zunehmen. Während sie anfangs lediglich ca. 10 Millisekunden betragen, steigen sie auf bis zu 35 Millisekunden an. Das ist ein relativer Anstieg von 350% und ein absoluter Anstieg von 25 Millisekunden.

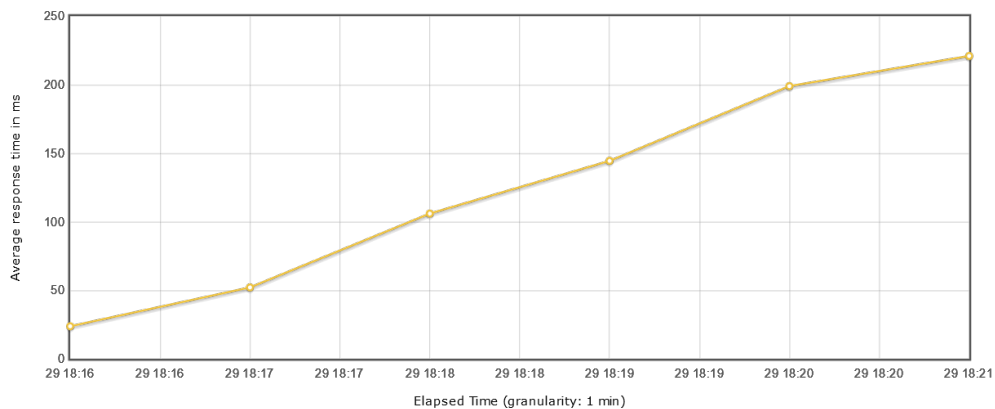


Abbildung 4.10: Response Time Graph von Skalierbarkeitstest von Ressource Server mit OPA

In [Abbildung 4.10](#) sind die Response Times für den Ressource Server mit entkoppelter Zugriffskontrolle zu sehen. Hier ist eine ähnliche Situation vorzufinden, nämlich die Response Times steigen kontinuierlich mit steigender Last. Allerdings steigen sie hier von ca. 25 Millisekunden auf bis zu ca. 250 Millisekunden. Das ist ein Anstieg um den Faktor 10 beziehungsweise 1000% und ein absoluter Anstieg um 225 Millisekunden. Es ist offensichtlich, dass der Server mit entkoppelter Zugriffskontrolle deutlich schlechter skaliert.

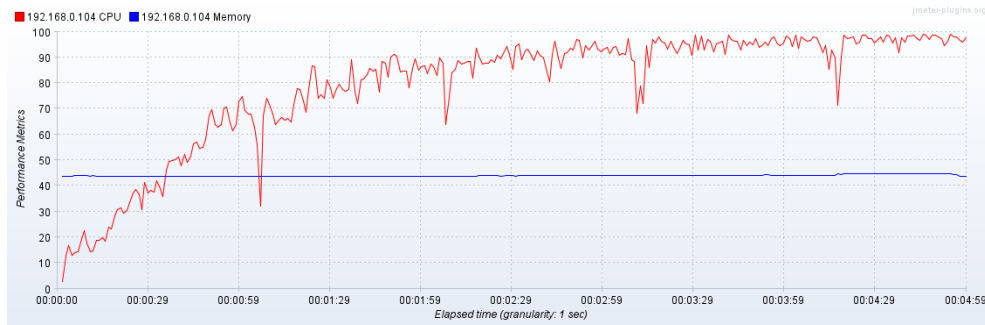


Abbildung 4.11: Perfmon von Skalierbarkeitstest von Ressource Server ohne OPA

In [Abbildung 4.11](#) ist die CPU-Auslastung (rote Linie) im Laufe des Tests bei dem Server ohne Open Policy Agent dargestellt. Es ist ein langsamer Anstieg der CPU-Auslastung festzustellen, die sich auf etwa 95% einpendelt.

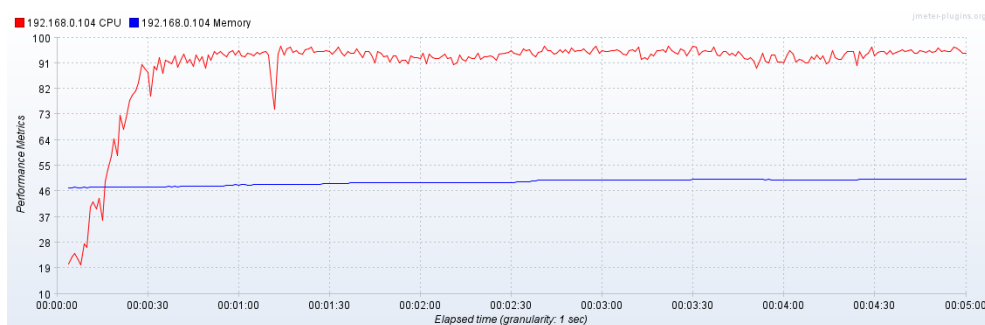


Abbildung 4.12: Perfmon von Skalierbarkeitstest von Ressource Server mit OPA

In [Abbildung 4.12](#) ist die CPU-Auslastung bei dem Server mit Open Policy Agent zu sehen. Hier ist zu sehen, dass die CPU-Auslastung deutlich schneller und steiler ansteigt. Auch daraus kann auf eine schlechtere Skalierung geschlossen werden. Die RAM-Belegung bleibt in beiden Systemen jedoch konstant. Dies ist nicht verwunderlich, da Systeme mit JSON Web Token als statuslos gelten, das heißt, dass der Server keine Informationen des Clients speichern muss, da alle Informationen in dem JSON Web Token vorhanden sind. Ein weiterer äußerst erwähnenswerter Punkt ist der, dass während des Skalierbarkeitstests, zeitweise Open Policy Agent un erreichbar war.

Requests	Executions				Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent	
Total	125306	26456	21.11%	120.74	11	1158	183.00	413.00	466.00	601.00	417.54	263.46	696.86	
HTTP Request	125306	26456	21.11%	120.74	11	1158	183.00	413.00	466.00	601.00	417.54	263.46	696.86	

Abbildung 4.13: Statistiken von Skalierbarkeitstest von Ressource Server mit OPA

In dem Server, der die Zugriffskontrolle mit Open Policy Agent entkoppelt hat, wurden 21,11% aller Anfragen nicht erfolgreich beantwortet, stattdessen wurden sie mit dem HTTP-Code 500 beantwortet. Der Fehler-Code 500 steht für Internal Server Error [\[35\]](#). Die Ursache davon ist, dass der Ressource

Server, also die Spring-Boot-Applikation, Open Policy Agent nicht erreichen konnte.

Listing 4.1: Fehlermeldung Spring-Boot

```
org.springframework.web.client.ResourceAccessException: I/O error on
  POST request for "HTTP://localhost:8181/v1/data/HTTP/authz/allow":
    Address already in use: connect; nested exception is java.net.
    BindException: Address already in use: connect
```

Die Meldung, die die Spring-Boot Applikation, ausgibt ist in [Listing 4.1](#) zu sehen. Der Ressource Server sendet jeweils bei jeder einkommenden Anfrage von Clients eine HTTP-POST Anfrage an Open Policy Agent, um eine Zugriffsentscheidung für diese Anfrage des Clients zu erhalten. Dies war in 21.11% aller Anfragen nicht möglich, weil Open Policy Agent blockiert. Da dieses Verhalten bei dem Lasttest nicht aufgetreten ist und der Lasttest mit lediglich zehn Threads eine weitaus geringere Belastung generiert, ist davon auszugehen, dass ab einer gewissen Belastung Open Policy Agent sperrt.

Open Policy Agent selbst nutzt als Webserver kein Apache Tomcat, sondern einen eigens in Go programmierten Webserver [4]. Etwaige Performanceeinstellungen lassen sich nicht vornehmen, ohne den Source-Code zu modifizieren und Open Policy Agent neu zu kompilieren und ein Docker-Container daraus wieder zu erstellen.

4.3 STRESSTEST

Bei dem Stresstest wurde eine abnormal hohe Belastung auf beide Server erzeugt, um die Grenzen des Systems auszuloten. Hier wurden 1000 Threads über einen Zeitraum von 15 Minuten gestartet.

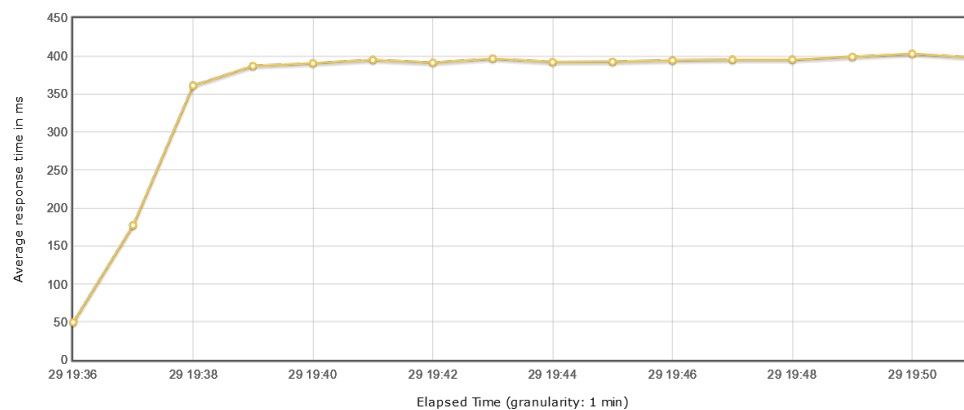


Abbildung 4.14: Response Time Graph von Stresstest von Ressource Server ohne OPA

In [Abbildung 4.14](#) sind die Response Times für den Server ohne OPA über den Zeitraum des gesamten Stresstests dargestellt. Die Response Times

stiegen kontinuierlich an, bis sie sich auf etwa durchschnittlich 400 Millisekunden einpendeln.

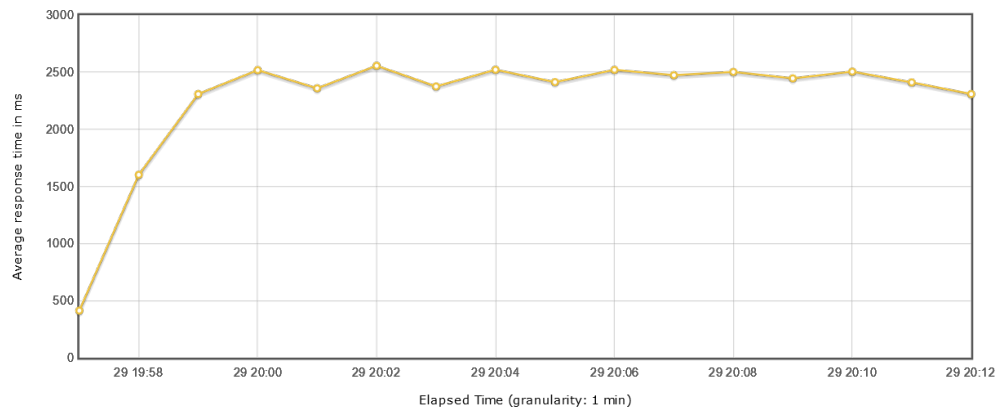


Abbildung 4.15: Response Time Graph von Stresstest von Ressource Server mit OPA

In [Abbildung 4.15](#) sind die Response Times für den Server mit entkoppelter Zugriffskontrolle mit Open Policy Agent zu sehen. Das Verhalten ist insgesamt sehr ähnlich, das heißt die Response Times steigen bis zu einem gewissen Zeitpunkt und bleiben dann konstant. Hier pendeln sich die Response Times allerdings auf 2500 Millisekunden ein. Das bedeutet im Vergleich zu dem Server ohne entkoppelter Zugriffskontrolle, sind diese Response Times ca. 6,25-mal höher.

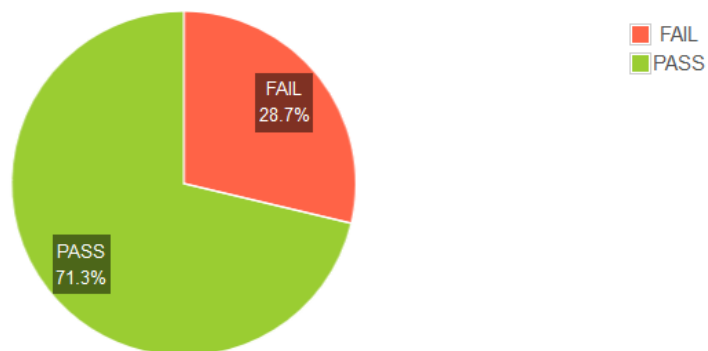


Abbildung 4.16: Request Summary von Stresstest von Ressource Server mit OPA

Allerdings wurden auch bei dem Stresstest ähnlich wie bei dem Skalierbarkeitstest eine hohe Anzahl von Anfragen bei dem Server mit entkoppelter Zugriffskontrolle nicht erfolgreich beantwortet. Dies ist in [Abbildung 4.16](#) erkenntlich. 28,7% aller Anfragen wurden nicht erfolgreich beantwortet. Auch hier war Open Policy Agent schlichtweg nicht erreichbar für den Server, sodass der Server dem Client mit dem HTTP-Code 500 antwortete. Aufgrund der hohen Response Times und den nicht erfolgreich beantworteten Anfragen, ist der Application Performance Index bei dem Server mit Open Policy Agent deutlich schlechter ausgefallen.

Apdex ▲	T (Toleration threshold) ◆	F (Frustration threshold) ◆	Label ◆
0.064	500 ms	1 sec 500 ms	Total
0.064	500 ms	1 sec 500 ms	HTTP Request

Abbildung 4.17: APDEX (Application Performance Index) von Stresstest von Resource Server mit OPA

Der Application Performance Index für das System mit Open Policy Agent ist in [Abbildung 4.17](#) zu sehen. Lediglich weniger als 6.4% aller Anfragen wurden bei diesem Server mit einer Response Time von weniger als 500 Millisekunden beantwortet. In anderen Worten: Der Großteil aller Antworten der Anfragen dauern aus Nutzersicht zu lange, sodass diese Nutzer frustriert sind.

Apdex ▲	T (Toleration threshold) ◆	F (Frustration threshold) ◆	Label ◆
0.938	500 ms	1 sec 500 ms	Total
0.938	500 ms	1 sec 500 ms	HTTP Request

Abbildung 4.18: APDEX (Application Performance Index) von Stresstest von Resource Server ohne OPA

Bei dem Server ohne entkoppelter Zugriffskontrolle hingegen, fällt der Score des Application Performance Index mit 0.938 wesentlich besser aus, siehe [Abbildung 4.18](#). Fast alle Anfragen konnten hier innerhalb von 500 Millisekunden erfolgreich beantwortet werden.

4.4 KUBERNETES CLUSTER

In diesem Kapitel wird gesondert betrachtet, inwiefern sich ein Deployment von Ressource Server und OPA in einem Kubernetes Cluster auf die Performancewerte auswirkt. Hierbei befinden sich die Docker-Container von Ressource Server und OPA in demselben Pod. OPA wird also als Sidecar verwendet [36].

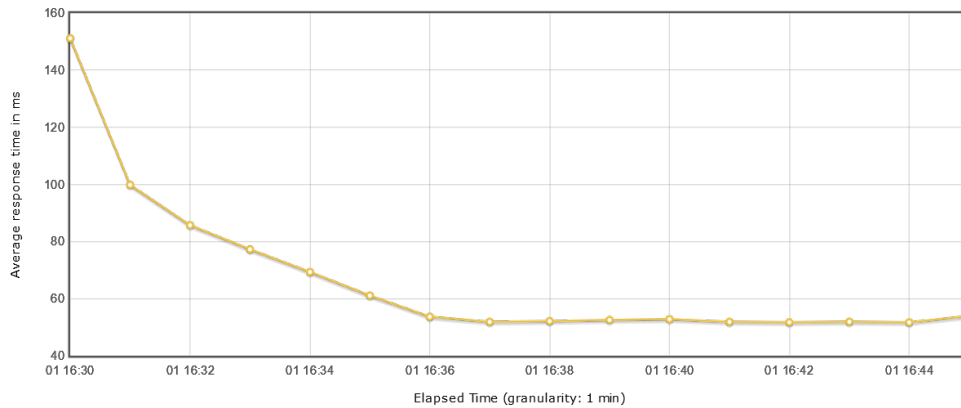


Abbildung 4.19: Response Time Graph von Lasttest von Ressource Server mit OPA in einem Kubernetes Cluster

In [Abbildung 4.19](#) ist der Response Time Graph von dem Lasttest abgebildet. Es ist zu sehen, dass hier die Response Times deutlich höher ausfallen im Vergleich zu dem System mit entkoppelter Zugriffskontrolle, das ohne Kubernetes ausgeführt wird, siehe [Abbildung 4.5](#). Auffallend ist außerdem, dass die Response Times kontinuierlich abnehmen, bis sie sich auf ca. 55 Millisekunden einpendeln.

Requests	Executions				Response Times (ms)							Throughput	Network (KB/sec)	
	Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total		147094	0	0.00%	60.86	9	994	73.00	87.00	94.00	99.00	163.44	111.52	262.56
HTTP Request		147094	0	0.00%	60.86	9	994	73.00	87.00	94.00	99.00	163.44	111.52	262.56

Abbildung 4.20: Statistiken von Lasttest von Ressource Server mit OPA in Kubernetes Cluster

In [Abbildung 4.20](#) sind die Gesamtstatistiken des Lasttests zu sehen. Die durchschnittliche Response Time beträgt hier ca. 60 Millisekunden während sie bei dem System ohne Kubernetes lediglich durchschnittlich 27 Millisekunden beträgt, siehe [Abbildung 4.5](#).

Bei dem Skalierbarkeitstest fielen die Response Times ebenfalls weitaus höher aus, diese werden hier allerdings nicht weiter diskutiert. Die Statistiken und Graphen können dem [Anhang A](#) entnommen werden. Was allerdings erwähnenswert ist, ist das während des Skalierbarkeitstest OPA immer verfügbar war, was bei demselben Test in dem System ohne Kubernetes nicht der Fall gewesen ist. Dies ist durch die Nutzung der Replica zu begründen, denn hier wurde ein LoadBalancer verwendet, der gleichmäßig die Anfragen auf die zwei Instanzen des Pods verteilt.

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
0.939	500 ms	1 sec 500 ms	Total
0.939	500 ms	1 sec 500 ms	HTTP Request

Abbildung 4.21: APDEX (Application Performance Index) von Skalierbarkeitstest von Ressource Server mit OPA und Kubernetes

Dadurch hat sich auch der Application Performance Index verbessert, wie es in [Abbildung 4.21](#) zu sehen ist. Dieser beträgt 0.939.

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
0.787	500 ms	1 sec 500 ms	Total
0.787	500 ms	1 sec 500 ms	HTTP Request

Abbildung 4.22: APDEX (Application Performance Index) von Skalierbarkeitstest von Ressource Server mit OPA ohne Kubernetes

In [Abbildung 4.22](#) ist der Application Performance Index des Skalierbarkeitstest des Systems mit OPA aber ohne Kubernetes zu sehen. Hier fiel der Index mit 0.787 deutlich schlechter aus aufgrund der Nichtverfügbarkeit von OPA.

Der Stresstest fiel hinsichtlich der Response Time schlechter aus, dieser wird hier allerdings nicht näher erläutert. Die entsprechenden Messergebnisse können aber dem [Anhang A](#) entnommen werden.

4.5 FAZIT

Es konnte gezeigt werden, dass eine entkoppelte Zugriffskontrolle mittels Open Policy Agent in einem OAuth2 System einen signifikanten negativen Einfluss auf die Performance hat. Bei leichter Last mit lediglich zehn gleichzeitigen Threads, fallen die Response Time durchschnittlich um ein dreifaches höher aus als bei dem Ressource Server, der die Zugriffskontrolle in der Spring-Boot Applikation implementiert hat.

Monolithen sind zwar grundsätzlich schneller als Microservice Architekturen durch die hinzukommende HTTP-Kommunikation zwischen den Microservices, aber ein derartig großer Performanceunterschied wurde nicht erwartet unter dem Hintergrund, dass bei dem Ressource Server mit entkoppelter Zugriffskontrolle, der Server und Open Policy Agent auf demselben Host laufen. Zu erwähnen ist hierbei allerdings, dass wenn jeweils auf die Response Time eine entfernungsabhängige Latenz zwischen Client und Server hinzugerechnet wird, der prozentuale Unterschied in der Performance zwischen Server mit interner vs. externer Zugriffskontrolle, geringer ist. Zusätzlich muss beachtet werden, dass die hier programmierte http-GET-

Schnittstelle in den jeweiligen Servern äußerst rudimentär ist. Das bedeutet, es wird keine lange Rechenzeit benötigt, die die Response Time maßgeblich beeinflusst. Auch dies kann in anderen Szenarien anders sein und den relativen Unterschied in der Performance zwischen beiden Servern verringern.

Neben der Response Time fiel zudem die äußerst hohe CPU-Auslastung im System mit entkoppelter Zugriffskontrolle auf. Bei dem Lasttest ist diese doppelt so hoch im Vergleich zu dem Server ohne Open Policy Agent. Selbst bei nur zehn gleichzeitig Threads bewegt sich das System mit Open Policy Agent bei annähernd 100-prozentiger CPU-Auslastung beinahe am Anschlag.

Zudem sperrt der Server von Open Policy Agent ab einer bestimmten Belastung, sodass der Ressource Server viele Client-Anfragen mit einer HTTP-Fehlermeldung 500 beantworten musste. Es scheint, dass mehrere Instanzen von Open Policy Agent für einen einzigen Ressource Server erstellt werden müssen, damit das Gesamtsystem unter starker Last regulär funktioniert. Da allerdings schon bei zehn Threads, das System mit Open Policy Agent am Anschlag arbeitet, scheint es nicht sinnvoll, mehrere Instanzen von Open Policy Agent auf der gleichen Host-Maschine zu starten. Wenn allerdings mehrere Open Policy Agent Instanzen auf mehreren Hosts laufen, würde die Latenz zwischen Ressource Server und Open Policy Agent zunehmen, da diese ja wiederum nicht mehr auf dem gleichen Host laufen. Es scheint also, dass bei der Nutzung von Open Policy Agent unweigerlich ein Kompromiss eingegangen werden muss.

Eine Verbesserung der Performance durch ein Deployment von Ressource Server und OPA in einem Kubernetes Cluster konnte nicht erreicht werden. Zwar konnte die Verfügbarkeit zumindest in dem Skalierbarkeitstests verbessert werden, allerdings fielen die Response Times in allen Fällen deutlich höher aus als in dem System, das kein Kubernetes verwendet. Dies ist auf den zusätzlichen Kommunikationsaufwand zurückzuführen, denn die Services in einem Kubernetes Cluster sind nicht direkt erreichbar, sondern der LoadBalancer leitet Anfragen entsprechend an die Ports der Pods weiter. Dadurch ist zwar eine horizontale Skalierung möglich, allerdings ist dies logischerweise in einem Cluster, das lediglich ein Node besitzt, wenig effektiv.

Eric Speidel hat in seiner Bachelorarbeit ein Single Sign-On-System entwickelt und hierbei unter anderem auch OpenID Connect verwendet und Performancetests mit Apache JMeter auf geschützte Schnittstellen des Servers durchgeführt [41]. Als Metrik bei diesen Performancetests wurde aber nur die CPU-Auslastung betrachtet und für die Zugriffskontrolle des Servers wurde Open Policy Agent nicht verwendet. Anders als in dieser Arbeit, hat Eric Speidel zur Implementierung des Ressource Servers .NET Core von Microsoft verwendet, was vergleichbar mit dem Java-Framework Spring ist.

Es wurde mittels Apache JMeter Last auf den Server erzeugt und die CPU-Auslastung des Servers mittels des Visual Studio 2017 Leistungsprofilers analysiert. Es ist davon auszugehen, dass Apache JMeter Last auf den Server erzeugt hat, indem es HTTP-Anfragen in einem Performancetest mit validen und in einem anderen Performancetest mit invalidem Token zu dem Server sendet. Bei den Performancetests mit validem Token wurden 100 Threads und 1000 Threads über einen Zeitraum von jeweils 30 Sekunden gestartet. Eric Speidel hat für einen kurzen Zeitraum eine maximale CPU-Auslastung von 10% bei 100 Threads und 15% bei 1000 Threads messen können, woraus er schließt, dass bei ansteigender Last, die CPU-Auslastung nicht linear steigt. Bei einem Performancetest mit invalidem Token und 1000 Threads konnte eine maximale CPU-Auslastung von 20% festgestellt werden. Der nur kurzfristige Anstieg der CPU-Auslastung und das darauffolgende Absinken der CPU-Auslastung auf jeweils unter 10% in allen drei Tests, erklärt sich Eric Speidel mit der ressourcenintensive Tokenvalidierung. Das nur eine relativ geringe CPU-Auslastung gemessen wurde, könnte damit begründet werden, dass die jeweiligen Performancetests nur für lediglich 30 Sekunden gestartet wurden. Eine Betrachtung der Latenz als Metrik in den Performancetests wurde nicht vorgenommen.

In der Bachelorarbeit „Verteilte Policy-basierte Autorisierung mit OAuth 2.0 und OpenID Connect“ von Johannas Steinleitner, wurde ein System entwickelt, indem die Authentifizierung von Nutzern mit OpenID Connect und Keycloak und die Autorisierung der Nutzer durch Open Policy Agent geschieht [42]. Ein entscheidender Unterschied ist der, dass in dem System von Johannas Steinleitner der Ressource Server nicht die JSON Web Token validiert, sondern diese Aufgabe Open Policy Agent übernimmt. Open Policy Agent bietet zwar Unterstützung für die Validierung von JSON Web Token, aber ein Nachteil hierbei ist, dass bei Anfragen auf den Ressource Server mit invalidem Token diese Anfragen dennoch Open Policy Agent zugesendet werden, der wiederum den Zugriff ablehnt, nachdem es den Token validiert hat. Wenn allerdings der Ressource Server die Validierung der Token übernimmt, würde bei dem Eintreffen von invalidem Token gar

keine Kommunikation zwischen Server und Open Policy Agent auftreten, was wiederum der Performance zugutekommt. Etwaige Performancetests wurden nicht durchgeführt. Die Behauptung, dass keine merklich hohe Latenz zwischen Komponenten entstehen, wenn diese sich in dem gleichen Kubernetes Pod befinden, wurde widerlegt.

Die Entwickler von Open Policy Agent haben selbst Performancetests durchgeführt, wobei sie zu dem Ergebnis gekommen sind, dass Open Policy Agent Zugriffsentscheidungen in der Regel innerhalb von einer Millisekunde evaluiert [5]. Um dies zu zeigen, wurden Benchmarks durchgeführt, in denen Zugriffsentscheidungen evaluiert werden und die Rechenzeit für die Evaluation anhand einer programmierten Zugriffsrichtlinie gemessen wird.

In diesen Tests wurde die Zeit gemessen, um eine Zugriffsentscheidung für eine rollenbasierte Zugriffskontrolle (engl. Role Based Access Control – RBAC) zu evaluieren, wobei hier die Input-Daten schon in der Rego-Datei in der die Zugriffskontrolle programmiert ist, vorhanden sind [7]. Hierbei wurde eine positive Zugriffsentscheidung in 605.076µs und eine negative Zugriffsentscheidung in 318.047µs getroffen, das heißt jeweils unter einer Millisekunde und damit aus Performancesicht in den aller meisten Anwendungsszenarien kein Problem.

Allerdings sind hier die Input-Daten schon in der Rego-Datei hinterlegt, während im Produktionseinsatz diese dem OPA-Service durch den Server per HTTP zugesendet werden. Dies wurde in den Tests von Open Policy Agent nicht berücksichtigt. Hier geht es allein um die reine Rechenzeit, die benötigt wird, um eine Zugriffsentscheidung innerhalb des OPA-Service zu evaluieren. Jeglicher Zeitaufwand, der durch die HTTP-Kommunikation zwischen Server und OPA-Service zustande kommt, wird nicht berücksichtigt.

Zudem sind die Input-Daten für den OPA-Service in OAuth2-Systemen kodierte JSON Web Token, was in den von Open Policy Agent durchgeführten Tests nicht der Fall ist. JSON Web Token können umfangreich sein, da neben dem Header, der Signatur und Pflicht-Claims, auch alle Nutzerinformationen in den Payload gemappt werden wie beispielsweise im Falle eines Mitarbeiters in einer Firma der Vor-und-Nachname, E-Mail-Adresse, Abteilung, Position, Büro, Standort sowie Gruppen-und-Rollenzugehörigkeiten. Zudem müssen JSON Web Token dekodiert werden, damit Open Policy Agent ihren Inhalt in JSON interpretieren kann, was zusätzliche Rechenzeit verursacht. In den Performance Tests von Open Policy Agent wurde also weder die Kommunikation zwischen Server und OPA-Service betrachtet noch Zugriffsentscheidungen anhand von zu dekodierenden JSON Web Tokens getroffen. Zudem wurde nicht getestet, wie sich ein Server, der die Zugriffskontrolle entkoppelt mit OPA, unter Last verhält und es wurde kein Vergleich gezogen, inwiefern sich die Performance im Vergleich zu einem Server ohne OPA verhält, da die Zugriffskontrolle natürlich auch in dem Server selbst implementiert werden kann.

ZUSAMMENFASSUNG

Es wurde anhand von Performancetests mit Apache JMeter gezeigt, dass eine externe Zugriffskontrolle mit Open Policy Agent eine negative Auswirkung auf die Response Times und Verfügbarkeit des Servers sowie auf die CPU-Auslastung des Systems hat. Hierbei wurden mittels Apache JMeter Anfragen mit validem JSON Web Token auf die Server generiert. Diese Server sind nach der OAuth2 Spezifikation Resource Server, da sie Zugriff auf die Schnittstellen nur zulassen, wenn ein valider Token vorliegt. Dieser JSON Web Token wird durch eine Authentifizierung eines Nutzers an einen Authorization Server erhalten. Die Resource Server prüfen einerseits, ob der Token valide ist und andererseits ob in dem Token die notwendige Berechtigung enthalten ist, um einen Zugriff auf die Schnittstelle des Servers zuzulassen. Hier wurde eine rollenbasierte Zugriffskontrolle implementiert, das heißt der authentifizierte Nutzer muss einer bestimmten Rolle angehörig sein, um Zugriff auf die Schnittstelle zu erhalten. Diese Zugriffskontrolle wurde in beide Server auf unterschiedliche Weise realisiert. Denn um den Einfluss von externer Zugriffskontrolle mit Open Policy Agent auf die Performance zu untersuchen, musste ein Zweitserver als Vergleichsserver implementiert werden. Dieser setzt die Zugriffskontrolle in der Serverapplikation selbst um. Die anfängliche Hypothese, dass durch die zusätzlich entstehende Kommunikation zwischen Server und Open Policy Agent ein Performance-nachteil entsteht, konnte bestätigt werden.

Da dieser Performancenachteil signifikant ist und unter starker Last das System mit entkoppelter Zugriffskontrolle nicht stabil arbeitet, da der Server von Open Policy Agent sperrt, kann als weiterer Ausblick untersucht werden, inwiefern sich die Performance von Open Policy Agent verbessern lässt. Hier sollte zum einen die zusätzliche Latenz, die bei der HTTP-Kommunikation zwischen Server und Open Policy Agent entsteht, verbessert werden und die CPU-Auslastung, verringert werden. Denn selbst bei nur zehn gleichzeitigen Threads arbeitete das System mit Open Policy Agent am Anschlag. Außerdem könnte Open Policy Agent mit weiteren Lösungen zur externen Zugriffskontrolle verglichen werden. Hier ist als weitere Lösung zum Beispiel die Keycloak Client Adapter zu nennen [47]. Hier werden Zugriffsrichtlinien nicht programmiert, sondern können in dem graphischen Admin-Menü von Keycloak festgelegt werden. Ein Nachteil dieser Lösung hingegen ist, dass sich hier zum einen an Keycloak als Authorization Server gebunden wird und zum anderen sind Keycloak Client Adapter nicht für alle Programmiersprachen verfügbar im Gegensatz zu Open Policy Agent, das praktisch system-, und plattformunabhängig ist. Die Lösung von Keycloak könnte unter den Kriterien der Performance, Funktionalität und Kompatibilität mit Open Policy Agent verglichen werden. Außerdem könnte untersucht wer-

den, inwiefern sich die Performance mittels Kubernetes verbessern lässt. Es könnte versucht werden, mehr als ein Node in einem Cluster zu verwenden. Hierdurch kann die Last auf mehreren physikalischen Maschinen verteilt werden, was potenziell die Performance speziell in einem System mit entkoppelter Zugriffskontrolle merklich verbessern könnte, da sich Open Policy Agent als äußerst CPU-intensive Komponente herausgestellt hat.

Teil II

APPENDIX

APPENDIX

Listing A.1: Deployment und Service von Keycloak

```
apiVersion: v1
kind: Service
metadata:
  name: keycloak
  labels:
    app: keycloak
spec:
  ports:
    - name: http
      port: 9080
      targetPort: 9080
  selector:
    app: keycloak
  type: LoadBalancer
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: keycloak
  namespace: default
  labels:
    app: keycloak
spec:
  replicas: 1
  selector:
    matchLabels:
      app: keycloak
  template:
    metadata:
      labels:
        app: keycloak
    spec:
      containers:
        - name: keycloak
          image: quay.io/keycloak/keycloak:12.0.4
          env:
            - name: PROXY_ADDRESS_FORWARDING
              value: "true"
            - name: DB_VENDOR
              value: "postgres"
            # Der PostgreSQL Nutzer
            - name: DB_USER
              value: "postgres"
```

```

# Das Passwort des PostgreSQL Nutzers
- name: DB_PASSWORD
  value: "postgrespw"
# Die Adresse der PostgreSQL-Datenbank in der Notation: "<
  postgres-service-name>". "<namespace>"
- name: DB_ADDR
  value: "postgres-db-lb.default"
# Name der PostgreSQL Datenbank, in der Keycloak-
  Einstellungen gespeichert werden
- name: DB_DATABASE
  value: "postgres"
# Name des Keycloak Nutzers
- name: KEYCLOAK_USER
  value: "admin"
# Passwort des Keycloak Nutzers
- name: KEYCLOAK_PASSWORD
  value: "admin"
ports:
- name: http
  containerPort: 9080
- name: https
  containerPort: 9443
args:
- "-Djboss.socket.binding.port-offset=1000"
readinessProbe:
  httpGet:
    path: /auth/realms/master
    port: 9080

```

Listing A.2: Deployment und Service von PostgreSQL als Datenbank für Keycloak

```

# PostgreSQL StatefulSet
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgresql-db
spec:
  serviceName: postgresql-db-service
  selector:
    matchLabels:
      app: postgresql-db
  replicas: 1
  template:
    metadata:
      labels:
        app: postgresql-db
    spec:
      containers:
        - name: postgresql-db
          image: postgres:13.4
          volumeMounts:
            - name: postgresql-db-disk

```

```

        mountPath: /data
    env:
      - name: POSTGRES_DATABASE
        value: "postgres"
      - name: POSTGRES_PASSWORD
        value: "postgrespw"
      - name: PGDATA
        value: /data/pgdata
# Volume Claim
volumeClaimTemplates:
  - metadata:
      name: postgresql-db-disk
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
---
# PostgreSQL StatefulSet Service
apiVersion: v1
kind: Service
metadata:
  name: postgres-db-lb
spec:
  selector:
    app: postgresql-db
  type: LoadBalancer
  ports:
    - port: 5432
      targetPort: 5432

```

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	56695	0	0.00%	267.46	7	2432	384.00	723.00	884.95	1200.99	188.76	128.81	303.24
HTTP Request	56695	0	0.00%	267.46	7	2432	384.00	723.00	884.95	1200.99	188.76	128.81	303.24

Abbildung A.1: Statistiken von Skalierbarkeitstest von Ressource Server mit OPA in einem Kubernetes Cluster

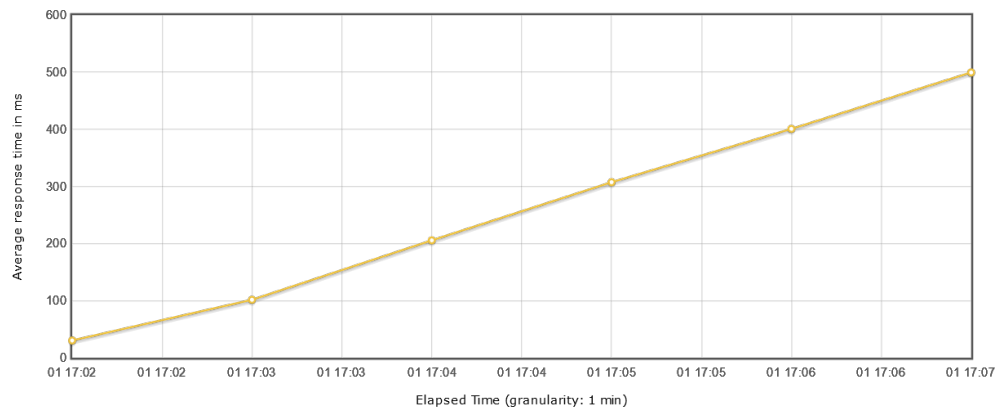


Abbildung A.2: Response Time Graph von Skalierbarkeitstest von Ressource Server mit OPA in einem Kubernetes Cluster

Requests	Executions				Response Times (ms)							Throughput	Network (KB/sec)	
Label ^	#Samples ^	FAIL ^	Error % ^	Average ^	Min ^	Max ^	Median ^	90th pct ^	95th pct ^	99th pct ^		Transactions/s ^	Received ^	Sent ^
Total	185324	33421	18.03%	4554.05	7	25372	4706.50	7236.00	7979.00	9885.00		204.37	202.23	269.10
HTTP Request	185324	33421	18.03%	4554.05	7	25372	4706.50	7236.00	7979.00	9885.00		204.37	202.23	269.10

Abbildung A.3: Statistiken von Stresstest von Ressource Server mit OPA in einem Kubernetes Cluster

LITERATUR

- [1] The Collaboration Platform for API Development. *Postman*. [Online; accessed 17-July-2021]. 2021. URL: <https://www.postman.com/>.
- [2] Open Policy Agent. *Deployment*. [Online; accessed 28-August-2021]. 2021. URL: <https://www.openpolicyagent.org/docs/latest/deployments/>.
- [3] Open Policy Agent. *External Data*. [Online; accessed 23-July-2021]. 2021. URL: <https://www.openpolicyagent.org/docs/latest/external-data/>.
- [4] Open Policy Agent. *Open Policy Agent*. [Online; accessed 30-July-2021]. 2021. URL: <https://github.com/open-policy-agent/opa>.
- [5] Open Policy Agent. *Policy Performance*. [Online; accessed 06-July-2021]. 2021. URL: <https://www.openpolicyagent.org/docs/latest/policy-performance/>.
- [6] Open Policy Agent. *The Rego Playground*. [Online; accessed 05-August-2021]. 2021. URL: <https://play.openpolicyagent.org/p/qUkvgJRpIU>.
- [7] Open Policy Agent. *The Rego Playground*. [Online; accessed 14-July-2021]. 2021. URL: <https://play.openpolicyagent.org/p/Ru5RQ5jkrd>.
- [8] Ben Alex, Luke Taylor, Rob Winch, Gunnar Hillert, Joe Grandja, Jay Bryant, Eddú Meléndez, Josh Cummings, Dave Syer und Eleftheria Stein. *Spring Security Reference*. [Online; accessed 21-July-2021]. 2021. URL: <https://docs.spring.io/spring-security/site/docs/current/reference/html5/#oauth2resourceserver>.
- [9] Inc. American National Standards Institute. *Role Based Access Control*. [Online; accessed 18-July-2021]. 2004. URL: <https://profsandhu.com/journals/tissec/ANSI+INCITS+359-2004.pdf>.
- [10] Gerald Brose. "Access Control". In: *Encyclopedia of Cryptography and Security*. Hrsg. von Henk C. A. van Tilborg und Sushil Jajodia. Boston, MA: Springer US, 2011, S. 2–7. ISBN: 978-1-4419-5906-5. DOI: [10.1007/978-1-4419-5906-5_179](https://doi.org/10.1007/978-1-4419-5906-5_179). URL: https://doi.org/10.1007/978-1-4419-5906-5_179.
- [11] EbruCelikel Cankaya. "Authentication". In: *Encyclopedia of Cryptography and Security*. Hrsg. von Henk C. A. van Tilborg und Sushil Jajodia. Boston, MA: Springer US, 2011, S. 61–62. ISBN: 978-1-4419-5906-5. DOI: [10.1007/978-1-4419-5906-5_772](https://doi.org/10.1007/978-1-4419-5906-5_772). URL: https://doi.org/10.1007/978-1-4419-5906-5_772.
- [12] D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON)*. [Online; accessed 04-August-2021]. 2006. URL: <https://datatracker.ietf.org/doc/html/rfc4627>.

- [13] Ed. D. Hardt. *The OAuth 2.0 Authorization Framework*. [Online; accessed 19-July-2021]. 2012. URL: <https://datatracker.ietf.org/doc/html/rfc6749>.
- [14] Docker. *Use containers to Build, Share and Run your applications*. [Online; accessed 16-July-2021]. 2021. URL: <https://www.docker.com/resources/what-container>.
- [15] Kerry Field, Michael Kennedy, Peter Christy, Mike Pennacchi und Jeffrey Sult. *Application Performance Index – Apdex Technical Specification*. [Online; accessed 05-August-2021]. 2007. URL: https://www.apdex.org/wp-content/uploads/2020/09/ApdexTechnicalSpecificationV11_000.pdf.
- [16] The Apache Software Foundation. *Apache JMeter*. [Online; accessed 05-August-2021]. 2021. URL: <https://jmeter.apache.org/>.
- [17] The Apache Software Foundation. *Apache Tomcat*. [Online; accessed 20-July-2021]. 2021. URL: <https://tomcat.apache.org/>.
- [18] The Apache Software Foundation. *Glossary*. [Online; accessed 12-July-2021]. 2021. URL: <https://jmeter.apache.org/usermanual/glossary.html>.
- [19] Anastasia Golovkova. *What's the Max Number of Users You Can Test on JMeter?* [Online; accessed 26-July-2021]. 2017. URL: <https://dzone.com/articles/whats-the-max-number-of-users-you-can-test-on-jmet>.
- [20] M. Jones. *JSON Web Key (JWK)*. [Online; accessed 20-July-2021]. 2015. URL: <https://datatracker.ietf.org/doc/html/rfc7517>.
- [21] S. Josefsson. *The Base16, Base32, and Base64 Data Encodings*. [Online; accessed 20-July-2021]. 2006. URL: <https://datatracker.ietf.org/doc/html/rfc4648>.
- [22] Keycloak. *Keycloak on Kubernetes*. [Online; accessed 28-August-2021]. 2021. URL: <https://www.keycloak.org/getting-started/getting-started-kube>.
- [23] Keycloak. *Securing Applications and Services Guide*. [Online; accessed 08-July-2021]. 2021. URL: https://www.keycloak.org/docs/latest/securing_apps/.
- [24] Kubernetes. *Minikube zum Erstellen eines Clusters verwenden*. [Online; accessed 28-August-2021]. 2019. URL: <https://kubernetes.io/de/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/>.
- [25] Kubernetes. *Verwenden von kubectl zum Erstellen eines Deployments*. [Online; accessed 28-August-2021]. 2019. URL: <https://kubernetes.io/de/docs/tutorials/kubernetes-basics/deploy-app/deploy-intro/>.
- [26] Kubernetes. *Create an External Load Balancer*. [Online; accessed 28-August-2021]. 2020. URL: <https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/>.

- [27] Kubernetes. *Kubernetes Komponenten*. [Online; accessed 28-August-2021]. 2020. URL: <https://kubernetes.io/de/docs/concepts/overview/components/>.
- [28] Kubernetes. *Was ist Kubernetes?* [Online; accessed 28-August-2021]. 2020. URL: <https://kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/>.
- [29] Kubernetes. *ConfigMaps*. [Online; accessed 28-August-2021]. 2021. URL: <https://kubernetes.io/docs/concepts/configuration/configmap/>.
- [30] Kubernetes. *Pods*. [Online; accessed 28-August-2021]. 2021. URL: <https://kubernetes.io/de/docs/concepts/workloads/pods/>.
- [31] Kubernetes. *Service*. [Online; accessed 28-August-2021]. 2021. URL: <https://kubernetes.io/docs/concepts/services-networking/service/>.
- [32] N. Sakimura M. Jones J. Bradley. *JSON Web Token (JWT)*. [Online; accessed 20-July-2021]. 2015. URL: <https://datatracker.ietf.org/doc/html/rfc7519>.
- [33] David Maguire und olprod. *Aktivieren oder Deaktivieren der modernen Authentifizierung für Outlook in Exchange Online*. [Online; accessed 20-July-2021]. 2021. URL: <https://docs.microsoft.com/de-de/exchange/clients-and-mobile-in-exchange-online/enable-or-disable-modern-authentication-in-exchange-online>.
- [34] Microsoft. *WSL commands and launch configurations*. [Online; accessed 30-July-2021]. 2021. URL: <https://docs.microsoft.com/en-us/windows/wsl/wsl-config>.
- [35] Mozilla. *500 Internal Server Error*. [Online; accessed 10-August-2021]. 2021. URL: <https://developer.mozilla.org/de/docs/Web/HTTP/Status/500>.
- [36] Masashi Narumoto und olprod. *Sidecar-Muster*. [Online; accessed 05-August-2021]. 2021. URL: <https://docs.microsoft.com/de-de/azure/architecture/patterns/sidecar>.
- [37] Andrey Pokhilko. *Servers Performance Monitoring*. [Online; accessed 10-August-2021]. 2021. URL: <https://jmeter-plugins.org/wiki/PerfMon/>.
- [38] Nat Sakimura, John Bradley, Michael B. Jones, Breno de Medeiros und Chuck Mortimore. *OpenID Connect Core 1.0 incorporating errata set 1*. [Online; accessed 20-July-2021]. 2014. URL: https://openid.net/specs/openid-connect-core-1_0.html.
- [39] Hirsch Singhal und olprod. *Microsoft Identity Platform-Zugriffstoken*. [Online; accessed 20-July-2021]. 2021. URL: <https://docs.microsoft.com/de-de/azure/active-directory/develop/access-tokens>.
- [40] Information Sciences Institute University of Southern California. *TRANSMISSION CONTROL PROTOCOL*. [Online; accessed 28-August-2021]. 1981. URL: <https://datatracker.ietf.org/doc/html/rfc793>.

- [41] Eric Speidel. *Entwicklung eines Single Sign-On-Systems mit verschiedenen tokenbasierten Authentifizierungsmethoden*. [Online; accessed 20-July-2021]. 2017. URL: https://www.ericSpeidel.de/bachelor-thesis-informatik_eric-speidel.pdf.
- [42] Johannes Steinleitner. *Verteilte Policy-basierte Autorisierung mit OAuth 2.0 und OpenID Connect*. [Online; accessed 20-July-2021]. 2020. URL: <https://opus4.kobv.de/opus4-haw-landshut/frontdoor/index/index/docId/229>.
- [43] Jakarta Servlet Team. *Jakarta Servlet Specification*. [Online; accessed 09-August-2021]. 2020. URL: <https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.pdf>.
- [44] "Authorization". In: *Encyclopedia of Cryptography and Security*. Hrsg. von Henk C. A. van Tilborg und Sushil Jajodia. Boston, MA: Springer US, 2011, S. 65–65. ISBN: 978-1-4419-5906-5. DOI: [10.1007/978-1-4419-5906-5_1025](https://doi.org/10.1007/978-1-4419-5906-5_1025). URL: https://doi.org/10.1007/978-1-4419-5906-5_1025.
- [45] Shanika Wickramasinghe. *Deploying PostgreSQL as a StatefulSet in Kubernetes*. [Online; accessed 28-August-2021]. 2021. URL: <https://www.bmc.com/blogs/kubernetes-postgresql/>.
- [46] connect2id. *How to select a JOSE / JWT cryptographic algorithm for your application*. [Online; accessed 06-July-2021]. 2021. URL: <https://connect2id.com/products/nimbus-jose-jwt/algorithm-selection-guide>.
- [47] keycloak. *Securing Applications and Services Guide*. [Online; accessed 28-August-2021]. 2021. URL: https://www.keycloak.org/docs/latest/securing_apps/.
- [48] minikube. *minikube start*. [Online; accessed 28-August-2021]. 2020. URL: <https://minikube.sigs.k8s.io/docs/start/>.
- [49] minikube. *Pushing images*. [Online; accessed 28-August-2021]. 2021. URL: <https://minikube.sigs.k8s.io/docs/handbook/pushing/>.
- [50] okta. *OAuth 2.0 and OpenID Connect Overview*. [Online; accessed 14-July-2021]. 2021. URL: <https://developer.okta.com/docs/concepts/oauth-openid/>.
- [51] spring. *Spring Boot*. [Online; accessed 05-August-2021]. 2021. URL: <https://spring.io/projects/spring-boot>.
- [52] spring. *Spring Framework*. [Online; accessed 05-August-2021]. 2021. URL: <https://spring.io/projects/spring-framework>.
- [53] spring. *Spring Security*. [Online; accessed 05-August-2021]. 2021. URL: <https://spring.io/projects/spring-security>.