



Westsächsische Hochschule Zwickau

University of Applied Sciences

HOCHSCHULE FÜR MOBILITÄT | UNIVERSITY FOR MOBILITY

Masterarbeit

Entwicklung einer RealWorld-App mit Axon Mikroservices und GRPC

Eingereicht an der

Fakultät Physikalische Technik/Informatik der Westsächsischen Hochschule
Zwickau – University of Applied Sciences for the degree of

Master of Science (M. Sc.)

Eingereicht von: **Nurseiit Tursunkulov** Geboren : 31.03.1994

Studienprogramm: Informatik

Ausgestellt von: Prof. Dr.Wolfgang Golubski

Erster Supervisor: Prof. Dr.Wolfgang Golubski

Zweite Supervisor: Prof. Dr. Frank Grimm

Abstrakt

In dieser Arbeit werden die Vorteile von GRPC im Vergleich zu REST in Client Server Kommunikationsarchitekturmodellen untersucht. Um dies zu ermitteln, wurde eine Blogging Anwendung mit einer domänengesteuerten Designarchitektur implementiert, die beide Kommunikationsschnittstellen für die Client Kommunikation bieten. Eine Reihe von Last Tests wurde für beide Schnittstellen durchgeführt und die Ergebnisse ausgewertet. Das Ergebnis zeigt, dass GRPC im Durchschnitt eine um 11% schnellere Antwortzeit als REST hat und GRPCs Kapazität ist um 18% größer als REST.

Erklärung zur selbständigen Erstellung der Arbeit

Ich versichere, dass ich diese Arbeit selbstständig angefertigt habe und bisher keine Hilfe oder Unterstützung erhalten habe, die gegen die Richtlinien zur akademischen Integrität meiner Institution verstoßen würde. Ich versichere außerdem, dass alle in dieser Arbeit verwendeten Quellen gemäß den Richtlinien meiner Einrichtung ordnungsgemäß zitiert und referenziert worden sind. Ich habe kein Material plagiiert oder nicht autorisierte Quellen verwendet. Außerdem erkläre ich, dass ich keine Version dieser Arbeit, weder ganz noch teilweise, zur Bewertung in einem anderen Kurs oder Studiengang eingereicht habe. Alle in dieser Arbeit präsentierten Arbeiten sind Originale und wurden noch nie zu akademischen Zwecken eingereicht.

Zwickau, 13.03.2023

A handwritten signature in black ink, enclosed in a rectangular box with blue corner markers. The signature is stylized and appears to be 'H. J. J. J.' followed by a large, sweeping flourish.

Gliederung

Abbildungsverzeichnis	VI
Tabellenverzeichnis	VIII
Kapitel 1 Einführung	1
1.1 Motivation	1
1.2 Problemstellung und Ziele der Arbeit	2
1.3 Aufbau der Arbeit	2
Kapitel 2 Grundlagen	3
2.1 HTTP1 und HTTP2	3
2.2 GRPC	4
2.2.1 GRPC Vorteile	5
2.2.2 GRPC Nachteile	6
2.2.3 GRPC vs REST	6
2.3 REST	7
2.3.1 Verwendung von REST-APIs	7
2.3.2 Ressourcen	8
2.3.3 REST-Prinzipen	8
2.3.4 Nachteile von REST	11
2.4 DDD	13
2.4.1 DDD Konzepte	13
2.4.2 Domain Model	14
2.5 Load Testing	14
2.5.1 Bewertungskriterien	14
2.5.2 Gatling	15
2.6 Axon	16
2.6.1 Domain Model Komponenten	17
2.6.2 Dispatch Model Komponenten	17
Kapitel 3 Anwendungsmodellierung und Entwicklung	18

3.1 Anwendung	18
3.2 Anwendungsarchitektur	18
3.3 REST-Schnittstelle	21
3.4 GRPC-Schnittstelle	26
Kapitel 4 Planung der Experimente	29
4.1 Test Szenarien	30
4.2 Test-Umgebung	38
Kapitel 5 Darstellung der Messergebnisse und Auswertung	38
5.1 Nutzer Erstellung Scenario REST vs GRPC	38
5.2 Artikel Erstellung Scenario REST vs GRPC	41
5.3 Comment Erstellung Scenario REST vs GRPC	43
5.4 POST Request Test REST vs GRPC	47
5.5 GET Request Test REST vs GRPC	49
5.6 PUT Request Test REST vs GRPC	51
Kapitel 6 Zusammenfassung der Ergebnisse	52
6.1 Vergleich zu anderen Forschungen	54
6.1.1 Forschung von Moscow Technical University of Communications and Informatics MTUCI	54
6.1.2 Forschung Rzeszov University of Technology (RUT)	54
6.1.3 Forschung von Masaryk University	58
Literatur	59

Abkürzungsverzeichnis

API	Application Programming Interfaces
CQRS	Command and Query Responsibility Segregation
DDD	Domain Driven Design
DSL	Domain Specific Language
GRPC / gRPC	Google Remote Procedure Call
HTML	Hypertext Markup Language
HTTP	Hyper Text Transfer Protocol
JSON	Java Script Object Notation
MTUCI	Moscow Technical University of Communications and Informatics
REST	Representational State Transfer
RPC	Remote Procedure Call
RUT	Rzeszov University of Technology
SSL	Secure Sockets Layer
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language

Abbildungsverzeichnis

Abbildung 1 Protobuf Beispiel	4
Abbildung 2 Client Server Architektur	9
Abbildung 3 Caching mechanism	10
Abbildung 4 Einheitliche Schnittstelle.....	11
Abbildung 5 Mangelnde Dokumentation	12
Abbildung 6 Gatlings Ergebniss	16
Abbildung 7 Kommunikationsschnittstellen	19
Abbildung 8 Microservices Communication Scheme	20
Abbildung 9 Test Planung.....	30
Abbildung 10 Gatling-Szenario zur Benutzererstellung mit REST	32
Abbildung 11 Gatling-Szenario zur Benutzererstellung mit GRPC.....	33
Abbildung 12 Gatling-Szenario zur ArtikelErstellung mit GRPC.....	34
Abbildung 13 Gatling-Szenario zur ArtikelErstellung mit REST	35
Abbildung 14 Gatling-Szenario zur CommentlErstellung mit GRPC	36
Abbildung 15 Gatling-Szenario zur CommentlErstellung mit REST	37
Abbildung 16 Nutzer Erstellung Szenario REST Ergebnis	39
Abbildung 17 Nutzer Erstellung Szenario GRPC Ergebnis	39
Abbildung 18 Nutzer Erstellung Szenario GRPC Ergebnis mit 220 Nutzern	40
Abbildung 19 Artikel Erstellung Szenario REST Ergebnis mit 220 Nutzern	41
Abbildung 20 Artikel Erstellung Szenario GRPC Ergebnis mit 270 Nutzern	41
Abbildung 21 Artikel Erstellung Szenario GRPC Ergebnis mit 220 Nutzern	42
Abbildung 22 Comment Erstellung Szenario REST Ergebnis mit 800 Nutzern	44
Abbildung 23 Comment Erstellung Szenario GRPC Ergebnis mit 900 Nutzern	45
Abbildung 24 Comment Erstellung Szenario GRPC Ergebnis mit 800 Nutzern	46
Abbildung 25 : User erstellung POST Anfrage REST Ergebnis mit 275 Nutzern	47
Abbildung 26 : User erstellung POST Anfrage GRPC Ergebnis mit 275 Nutzern.....	48
Abbildung 27 User erstellung POST Anfrage GRPC Ergebnis mit 350 Nutzern.....	49
Abbildung 28 Artikel Erstellung GET Anfrage REST Ergebnis mit 275 Nutzern	50
Abbildung 29 Artikel erstellung POST Anfrage GRPC Ergebnis mit 275 Nutzern	50
Abbildung 30 PUT Anfrage REST Ergebnis mit 220 Nutzern.....	51
Abbildung 31 PUT Anfrage GRPC Ergebnis mit 220 Nutzern	52
Abbildung 32 GRPC und REST Vergleich	53
Abbildung 33 GRPC und REST Vergleich "Hello World" (source Forschung [1] chapter 3 experimental result)	55
Abbildung 34 GRPC und REST Vergleich "Lore ipsum" 615 Charakters (source Forschung [1] chapter 3 experimental result)	55
Abbildung 35 REST HelloWorld	56
Abbildung 36 GRPC HelloWorld.....	56
Abbildung 37 REST Ergebnis "Hello World" und "Lore ipsum..."	57
Abbildung 38 GRPC Ergebnis "Hello World" und "Lore ipsum..."	58

Tabellenverzeichnis

Tabelle 1 GRPC und REST Vergleich [9] [9, P.53].....	7
Tabelle 2 Article Service Endpunkte	23
Tabelle 3 Comment Service Endpunkte.....	24
Tabelle 4 User Service Endpunkte.....	26
Tabelle 5 Article Service Endpunkte	27
Tabelle 6 Comment Service Endpunkte.....	28
Tabelle 7 User Service Endpunkte.....	29
Tabelle 8 Test Umgebung	38
Tabelle 9 Vergleich von GRPC und REST Nutzer Erstellung.....	40
Tabelle 10 Vergleich von GRPC und REST mit 220 Nutzern Erstellung	41
Tabelle 11 Vergleich von GRPC (270) und REST (220) Artikel Erstellung	42
Tabelle 12 Vergleich von GRPC (220) und REST (220) Artikel Erstellung	43
Tabelle 13 Vergleich von GRPC (900) und REST (800) Comment Erstellung	46
Tabelle 14 Vergleich von GRPC (800) und REST (800) Comment Erstellung	47
Tabelle 15 User erstellung POST Request GRPC und REST Vergleich Ergebnis mit 275 Nutzern.....	48
Tabelle 16 User erstellung POST Request GRPC und REST Vergleich Ergebnis mit 275 Nutzern.....	48
Tabelle 17 User erstellung GRPC und REST Vergleich Ergebnis mit 275 Nutzern	51
Tabelle 18 PUT Anfrage GRPC und REST Vergleich mit 220 Nutzern	52
Tabelle 19 GRPC und REST Vergleich.....	53

Kapitel 1 Einführung

Unser Leben ist wie ein Fahrradmarathon: um die Siegeslinie zu erreichen, muss man in die Pedalen treten, d.h. sich ständig weiterentwickeln und verbessern. Wenn man aufhört, in die Pedalen zu treten, wird es allmählich zu einem Sturz kommen. Sollte man es unterlassen, sich zu verbessern und zu forschen, wird das zum Niedergang des Unternehmens führen. Ein eindrucksvolles Beispiel ist das Unternehmen Nokia. Im Zeitraum 2000-2012 war es das führende Unternehmen beim Verkauf von Handys, aber einer der Gründe für seinen Niedergang ist, dass es keine Ressourcen in die technologische Entwicklung investiert hat [1]. Wir können daraus die Lehre ziehen, dass wir mit der Zeit gehen müssen, um die Wettbewerbsfähigkeit des Unternehmens zu erhalten. Es müssen die neuen Technologien erforscht und adaptiert werden, falls sie nützlich sind. In dieser Masterarbeit werden zwei Kommunikationsprotokolle verglichen. Eines davon wird von den meisten Unternehmen verwendet und ist erprobt. Das andere ist neu, verspricht aber, die Produktivität zu steigern.

1.1 Motivation

Laut Forschung von Rzeszow University of Technology die erste Wahl für Mikroservice-Kommunikationstechnologien ist REST, aber es wurde festgestellt, dass das Interesse an GRPC wächst [2]. Laut einer anderen Forschung ist GRPC 2-mal schneller als REST. [3]

Jeremy H in blog.dreamfactory.com hat GRPC mit REST verglichen. Laut seinem Ergebnis ist REST die beste Wahl. [4]

In einer anderen Forschung von R.Fernando wurde festgestellt, dass GRPC ist 7 mal schneller als REST ist. [5] [6, S. 69]

Michail Stefanic von der Masaryc-Universität kam in seiner Masterarbeit zu dem Schluss, dass GRPC eine Technologie ist, die nicht für jede Architektur und Umgebung geeignet ist und das es keinen Grund gibt, REST achtlos zu verwerfen und alles auf GRPC umzustellen. Die REST-Architektur ist und wird auch in den kommenden Jahren für viele die Lösung Nummer eins für die Service-Kommunikation sein. Der Hauptgrund dafür ist ihre Beliebtheit, Unterstützung und Einfachheit. Daher ist es wichtig, vor der Migration einen Entscheidungsprozess zu durchlaufen und alle Vor- und Nachteile der beiden Technologien abzuwägen. [6, S. 73]

Moderne Anwendungen umfassen komplexe Architekturen, die effiziente und leistungsstarke Werkzeuge zur Interaktion mit dem Backend erfordern. Die Wahl zwischen verschiedenen Technologien und Protokollen wie GRPC und REST kann jedoch eine schwierige Forschungsfrage sein. Die Motivation für diese Masterarbeit ist zu recherchieren, welche von

beiden die bessere Leistung und Effizienz bietet, die auf der Grundlage des Axon-Frameworks entwickelt wurde.

1.2 Problemstellung und Ziele der Arbeit

Es gibt verschiedene Forschungen welche die Vorteile von GRPC aufzeigen, [3] [2] [4] [5] aber jede Forschung ist in unterschiedlichen Kontext aufgebaut und jeder gibt verschiedene Forschungsergebnisse ab.

Es wurde beschlossen, eine Blogging Anwendung wie RealWorld zu entwickeln. Denn RealWorld ist ein Projekt, das als Benchmark für die Demonstration und den Vergleich verschiedener Frameworks, Technologien und Ansätze in der Softwareentwicklung dient. Um die Arbeit realitätsnäher zu gestalten, wurde die Anwendung auf der Grundlage moderner Architekturlösungen wie Microservice-Architektur unter Berücksichtigung der Prinzipien des Domain-Driven Design (DDD), Command and Query Responsibility Segregation (CQRS) mit Frameworks Axon und Spring aufgebaut. Das Ziel der Arbeit ist, die Unterschiede zwischen der Verwendung von GRPC und REST auf der Ebene des Anwendungseingangspunkts zu bestimmen.

Um oben genannte Ziele zu erreichen wird folgendes geplant:

1. Aufbau der Blogging Anwendung.
2. Load Test Durchführung auf der App.
3. Ergebnis und Bewertung.

1.3 Aufbau der Arbeit

Die Struktur dieser Masterarbeit ist wie folgt aufgebaut:

Kapitel 2: "Grundlagen" führt in die grundlegenden technologischen Konzepte ein, die in dieser Arbeit verwendet werden, einschließlich des Axon-Frameworks, des GRPC-Protokolls, der Prinzipien des Domain-Driven Design (DDD) und des Load Testings.

Kapitel 3: "Design und Implementierung" behandelt die Blogging Anwendung und die Beschreibung von Testszenarien mit deren Implementierung

Kapitel 4: Umfasst die Planung der Experimente für GRPC und REST.

Kapitel 5: "Analyse der Ergebnisse" vergleicht die Leistung der beiden Einstiegspunkte und identifiziert die Schlüsselfaktoren, die die Ergebnisse beeinflussen.

In Kapitel 6 "Zusammenfassung der Ergebnisse" werden die Erkenntnisse zusammengefasst und praktische Empfehlungen auf der Grundlage der Ergebnisse gegeben. In den folgenden Abschnitten wird jeder der oben genannten Aspekte der Studie im Detail erläutert.

Kapitel 2 Grundlagen

In diesem Kapitel werden die grundlegenden Konzepte und Technologien vorgestellt, auf deren Basis die RealWorld-App entwickelt wurde. Außerdem wird die Technologie vorgestellt, mit der die Load Testing für den GRPC- und REST-Einstiegspunkt durchgeführt wurde.

2.1 HTTP1 und HTTP2

“Das Hyper Text Transfer Protocol (HTTP) ist ein Protokoll auf Anwendungsebene Protokoll für verteilte, kollaborative, hypermediale Informationssysteme Informationssysteme. Es ist ein generisches, zustandsloses Protokoll, das für viele Aufgaben verwendet werden kann Hypertext verwendet werden kann, z. B. als Namensserver und verteilte Objektverwaltungssysteme, durch Erweiterung seiner Anforderungsmethoden, Fehlercodes und Header. Ein Merkmal von HTTP ist die Typisierung und Aushandlung der Datendarstellung, wodurch Systeme unabhängig von den zu übertragenden Daten aufgebaut werden können.” [7]

HTTP1. HTTP2 wurde im Mai 2015 als RFC 7540 veröffentlicht und unterscheidet sich wesentlich von HTTP/1.1, da es ein binäres Protokoll ist. Diese binäre Codierung erleichtert das Framing und ermöglicht eine einfachere Identifizierung von Frames, im Gegensatz zu HTTP1, das auf Text basiert und weniger klare Anfangs- und Endpunkte für Frames aufweist [6, S.12].

HTTP2 bietet auch den Vorteil einer reduzierten Anzahl von Netzwerk-Roundtrips, was die Kommunikation beschleunigt. Es ermöglicht zudem die Verwendung vieler paralleler Streams und priorisiert den Datenfluss, um wichtige Daten zuerst zu übertragen [6, S.12].

Insgesamt führen diese Verbesserungen zu schnelleren Ladezeiten von Webseiten und erhöhter Datensicherheit.

Das ist wichtig, weil GRPC verwendet das HTTP2 Protokoll.

2.2 GRPC

Remote Procedure Call (RPC) ist ein Konzept, das ermöglicht, Funktionen über ein Netzwerk aufzurufen. Bei RPC sendet ein Client eine Anfrage, die den Namen und die Parameter der gewünschten Funktion enthält, an einen Server. Während des Prozesses wartet der Client auf eine Antwort. Nachdem der Server die Funktion bearbeitet hat, schickt er die Antwort zurück, woraufhin der Client seine Aktivitäten fortsetzt [8, S.17].

GRPC ist ein offenes RPC-Framework, das von Google-Entwicklern erstellt wurde und auf verschiedenen Plattformen eingesetzt werden kann.

RPC verwendet das HTTP/2-Protokoll, das effizienter und leistungsfähiger ist als frühere Versionen von HTTP. Dieses Protokoll unterstützt Multiplexing, Datenkompression, Streaming und andere Funktionen, die eine schnelle und zuverlässige Datenübertragung zwischen Clients und Servern ermöglichen [6, S.21].

Protocol buffers

Protocol buffer Abbildung 1 ist ein von Google entwickeltes, flexibles, effizientes und automatisiertes System für Serialisierung strukturierter Daten, ähnlich wie XML oder JSON. Protobuf wurde entwickelt, um schneller als XML oder JSON zu sein und bietet dabei auch eine bessere Typsicherheit und Fehlerprüfung [6, S.22].

```
message CreateUserProfileRequest {  
    string userName = 1;  
    string name = 2;  
    string email = 3;  
}  
service User {  
    rpc createUserProfile(CreateUserProfileRequest) returns  
        (StringMessageParam) {}  
}
```

Abbildung 1 Protobuf Beispiel

GRPC Kommunikation

Der GRPC-Protokollpuffer-Compiler generiert den gesamten erforderlichen Code auf Server und Client-Seite. Der GRPC-Server interpretiert und bearbeitet eingehende Remote-Anfragen, entsprechende Proto-Datei und sendet kodierte Antworten zurück an den Client [6, S.23].

RPC-Typen in GRPC

Unary RPC: Das ist der einfachste RPC-Typ, bei dem der Client eine einzelne Anfrage sendet und auf eine einzelne Antwort des Servers wartet [6, S.23].

Server Streaming RPC: Server schickt Stream von Antworten. Nachdem alle Antworten verarbeitet wurden, sendet der Server den Statuscode und die abschließende Nachricht an den Client. Das beendet die Verarbeitung der RPC-Anfrage serverseitig. Der Client betrachtet die Anfrage als abgeschlossen, sobald er alle Nachrichten vom Server erhalten hat [6, S.23].

Client Streaming RPC: Der Client erstellt einen Stream und sendet mehrere Nachrichten an den Server, der auf alle Nachrichten wartet und dann eine einzelne Antwort sendet [6, S.23].

Bidirektionaler Streaming RPC: Das ist die Kombination von Client und Server Streaming. Der Client startet das Streaming und der Server nimmt die Nachrichten an. Die Reihenfolge von Anfragen und Antworten hängt von der Anwendung ab. Der Server kann entweder auf alle Nachrichten des Clients warten oder mit dem Client schrittweise Nachrichten verarbeiten und antworten [6, S.23].

2.2.1 GRPC Vorteile

- **Starke API-Verträge:**

GRPC verwendet Protocol Buffers für Datenübertragung und Serialisierung. Dadurch werden feste Datentypen für API-Nachrichten festgelegt. Dies gewährleistet gleiche Daten zwischen Client und Server. Mit den Protocolbuffers können komplexe Nachrichtenstrukturen erstellen werden [6, S.30].

- **Bessere Leistung:**

GRPC verwendet HTTP2 was schneller als HTTP1 ist, Datenkompression und Protocol Buffers liefert die höhere Leistung. Laut der Forschung von MTUCI [9] [9, S.54] ist die GRPC Leistung 5 mal größer als REST. Beim Streaming 7 mal größer [6, S.30].

- **Datenkompression:**

GRPC kann die Anfrage- und Antwortdaten komprimieren. Die Datenkompression ist freiwillig [6, S.30].

- **Client and server streaming:**

Wie oben beschrieben, bietet GRPC bidirektionale Streaming. Solche Funktion hat REST nicht [6, S.30].

2.2.2 GRPC Nachteile

- **Neue Technologie:** GRPC ist eine relativ neue Technologie, die trotz ihres Potenzials mit etablierten Technologien wie REST konkurriert. Die Entscheidung für eine bekannte und geprüfte Technologie fühlt sich oft sicherer an [6, S.34].
- **Weniger Browserunterstützung:** GRPC hat keine direkte Browserunterstützung, da HTTP/1.1 in Browsern weit verbreitet ist. Es gibt zwar Lösungen wie GRPC-Web, aber sie erfordern zusätzliche Proxys, um HTTP-Versionen und Nachrichtenformate zu überbrücken. Daher ist GRPC keine direkte Alternative zu REST für browserbasierte Anwendungen [6, S.34].
- **Test Schwierigkeiten:** Im Gegensatz zu REST, die URLs und Werkzeuge wie Curl oder Postman für Tests verwenden, basiert GRPC auf direkten Methodenaufrufen. Das macht den Test komplexer, weil Protobuf-Dateien nicht so leicht lesbar sind und spezielle Tools erfordern [6, S.34].

2.2.3 GRPC vs REST

MTUCI Universität haben Forschungen durchgeführt, wo GRPC mit REST verglichen wurde und die Ergebnisse in der Tabelle 1 dargestellt. Sie sind zum Schluss gekommen das GRPC 5 mal schneller ist, wenn massive Daten von mehr als 437kB Payload benutzt werden. Aber wenn die Daten kleiner sind, dann ist der Gewinn nicht so groß [9, S.53].

criteria	GRPC	REST
Data serialization (basic)	Protobuf	JSON/XML
Data transmission protocol	HTTP/2, asynchronous frame transmission	HTTP/1.1
Safety tools	Supports SSL/TLS for connection security, various authentication and authorization mechanisms. In addition, security tools developed by Google are supported	Supports SSL/TLS for connection security, various authentication and authorization mechanisms. A wide range of options for establishing a secure connection link.
Transmitted data compression	HPACK, supports header compression	GZIP, header compression is not available

Performance	Faster than the REST when transferring massive data. If compared to the REST, it has insignificant performance advantage when low pay-load is conveyed.	Slower than the gRPC when transferring massive data. If compared to the gRPC, it has insignificant performance advantage when low pay-load is conveyed.
Code generation tools	Has built-in code generation tools. In most cases, system implementation is more complex and time-consuming.	Built-in code generation tools are not available. Swagger, Mashape, Apiary and more can be used. Many objectives can be achieved much more easily and quickly than using the gRPC.

Tabelle 1 GRPC und REST Vergleich [9] [9, P.53]

2.3 REST

Representational State Transfer (REST) ist ein Architekturstil für die Entwicklung von Webanwendungen. REST ist bekannt für ein Application Programming Interface (API), der auf einem klassischen Anfrage-Antwort Ansatz basiert. Der REST-API-Architekturstil wird für die Entwicklung von Webdiensten und die Kommunikation im Internet verwendet. REST verwendet das HTTP-Protokoll, um mit HTTP-Methoden wie, POST, GET, PUT oder PATCH und DELETE auf Ressourcen zu ändern oder auf Ressourcen zuzugreifen [6, S.13].

2.3.1 Verwendung von REST-APIs

Viele große Technologieunternehmen wie Google, Facebook oder Amazon verwenden REST-APIs. REST hat sich zu einer sehr beliebten Lösung für das Cloud-Computing entwickelt, die eine Möglichkeit bietet, Services zu veröffentlichen und mit Clienten zu kommunizieren. Bei der Entwicklung von REST APIs oder Webdiensten haben Entwickler die Freiheit, jede Sprache zu

nutzen, wie Go, Python, JavaScript und mehr. Diese Flexibilität erlaubt es Programmierern, Technologien auszuwählen, die sie bevorzugen und die am besten zu ihren Bedürfnissen passen [6, S.13].

2.3.2 Ressourcen

Im Zusammenhang mit REST stehen Ressourcen für alle Objekte oder Komponenten, mit denen Entwicklern interagieren möchten. Diese Ressourcen können Datenobjekte sein, wie Bilder, Textdokumente, Videos oder Datenbankeinträge. Jede Ressource in einer REST-API ist durch eine eindeutige URI identifizierbar [8, S.11]. REST ermöglicht den Zugriff und die Manipulation dieser Ressourcen über standardisierte HTTP-Methoden. Zum Beispiel verwendet eine GET-Anfrage die URI der Ressource zum Abrufen der Daten, während eine POST-Anfrage Daten an die Server-Ressource sendet. PUT und PATCH erlauben das Aktualisieren von Ressourcen, und DELETE erlaubt das Löschen von Ressourcen [6, S.13].

2.3.3 REST-Prinzipen

Zustandslose Kommunikation

REST APIs speichern keinen Zustand. Der aktuelle Zustand einer Ressource wird durch ihre Darstellung angezeigt. Wenn ein Vorgang läuft, kann kein anderer gleichzeitig die Ressource ändern. Alle Aktionen des Servers sind zustandsfrei, daher muss der Client den Zustand selbst verwalten. “Jede Anfrage muss unabhängig von allen anderen gemacht werden” [8, S.11].

Einmal an den Client gesendet, kümmert sich der Server nicht mehr um den Zustand dieser Ressource [20]. Solche zustandsfreien Dienste sind üblich im Internet. Zum Beispiel, wenn Nutzer die Facebook-App Nachrichten öffnet, holt die App die Nachrichten vom Server. Sobald sie auf dem Handy sind, gibt es keine direkte Verbindung zum Server mehr [6, S13].

Client-Server

In einer Client-Server Architektur es gibt Client und Server. Ein Client ist in der Regel ein Endbenutzer-Gerät wie ein Computer, Smartphone oder Tablet. Der Client stellt Anfragen an den Server und bekommt Daten als Antwort. Der Prozess kann wie in der Abbildung 2 dargestellt

werden. Ein Server ist ein Gerät oder eine Maschine die Dienste oder Daten für Clients bereitstellt.

Funktions Beispiel:

Der Client schickt eine Anfrage an den Server um eine Webseite zu laden, Daten aus einer Datenbank abzurufen oder eine Datei herunterzuladen. Der Server bekommt die Anfrage, verarbeitet und sendet eine entsprechende Antwort zurück an den Client. Der Client bekommt die Antwort und stellt sie dem Endbenutzer zur Verfügung, z.B. durch Anzeigen einer Webseite im Browser oder eine Tabelle mit Daten [8, S.13].

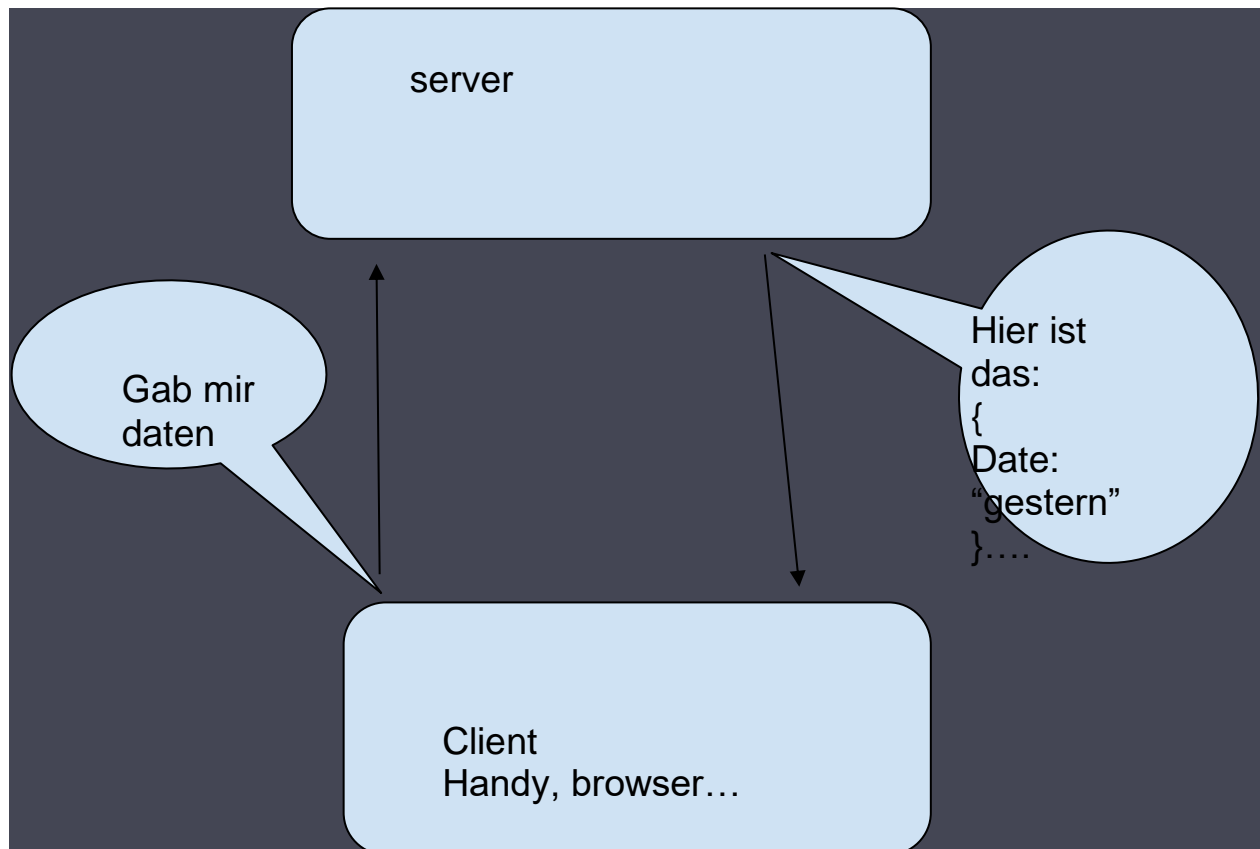


Abbildung 2 Client Server Architektur

Caching

Durch das Verwenden eines Caches kann eine zusätzliche Schicht zwischen Client und Server hinzugefügt werden. Dieser Cache speichert Antworten auf frühere Anfragen. Wenn ein Client eine Anfrage stellt, die bereits im Cache gespeichert ist, kann der Cache die Antwort direkt liefern, ohne dass der Server eingreifen muss. Die Abbildung 3 beschreibt den Caching Vorgang. Nur wenn die Antwort nicht im Cache vorhanden ist, wird die Anfrage an den Server weitergegeben. Das funktioniert, weil jede Anfrage für sich alleinsteht und nicht von früheren Anfragen abhängt. Ebenso enthält jede Serverantwort alle erforderlichen Informationen [8, S.13].

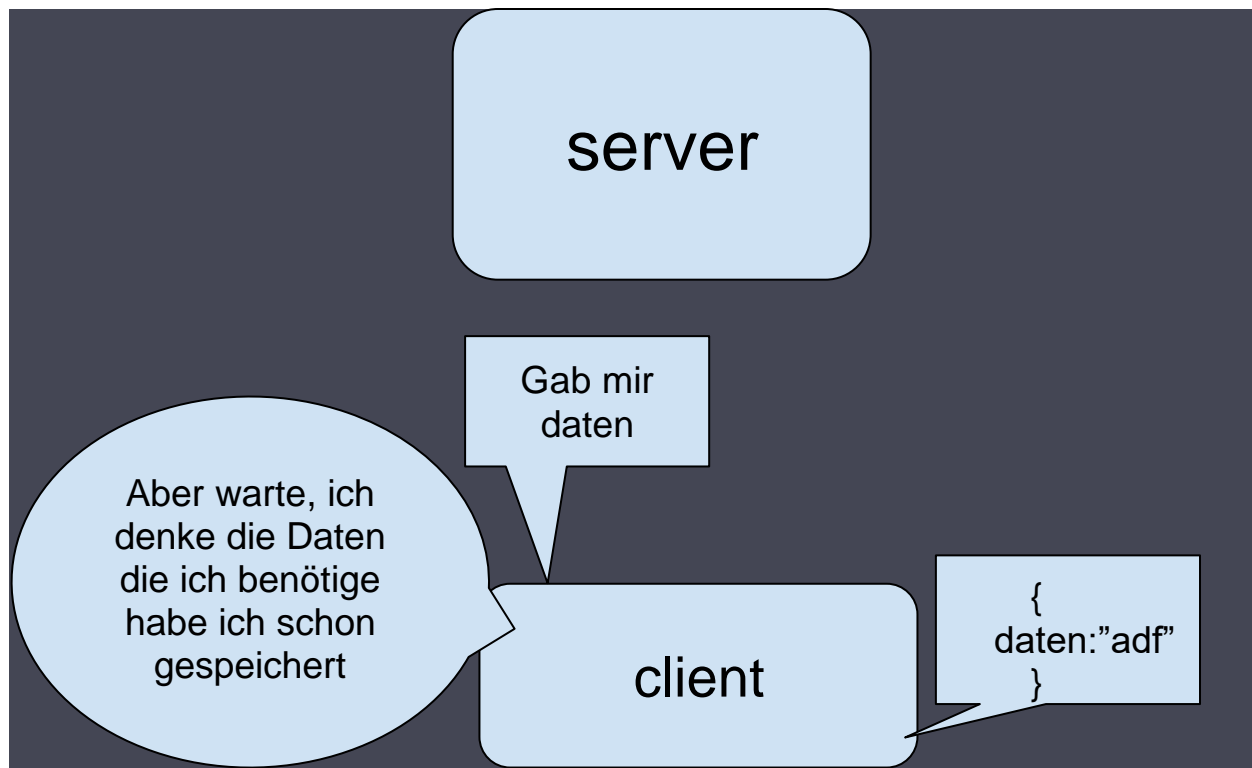


Abbildung 3 Caching Vorgang

Einheitliche Schnittstelle

Laut dem Prinzip sprechen Client und Server in einer gemeinsamen, standardisierten Sprache, zum Beispiel HTTP. Die Abbildung 4 beschreibt die Einheitliche Schnittstelle.

Die Nutzung einer einheitlichen Schnittstelle müssen Ressourcen in REST-Systemen durch eine spezifische URL erkennbar sein. Die Manipulation dieser Ressourcen sollte ausschließlich durch die grundlegenden Methoden des Netzwerkprotokolls, wie DELETE, PUT und POST im Rahmen von HTTP erfolgen [8, S.14].

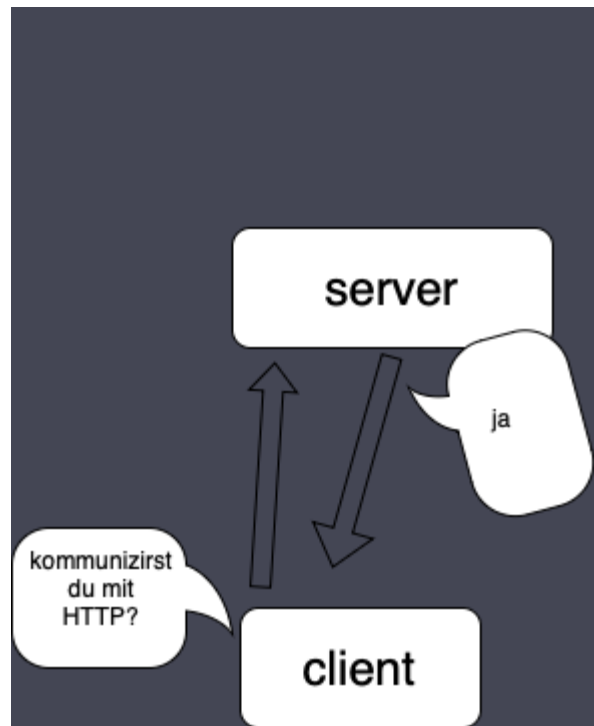


Abbildung 4 Einheitliche Schnittstelle

2.3.4 Nachteile von REST

Falsche Verwendung der Methoden GET und POST.

Ein häufiger Fehler ist die falsche Verwendung von GET und POST. Obwohl GET eine Payload body im Request zulässt, sollte es nicht genutzt werden. GET sollte idempotent sein und nur Ressourcen abrufen, nicht neue erstellen oder vorhandene ändern. Ein weiterer Fehler ist die Verwendung von POST, um Ressourcen abzurufen, was eher für SOAP typisch ist. Dies kann zu Problemen führen, wenn ein neuer POST-Endpunkt benötigt, aber bereits für GET-Zwecke verwendet wird [6, S.17].

Fehlerbehandlung

Entwicklern nutzen die 200 Statuscode für fast alles, unabhängig vom tatsächlichen Erfolg oder möglichen Fehlern. Es ist empfehlenswert, die HTTP-Statuscodes korrekt zu verwenden. Die meisten Entwickler kennen diese Codes und wissen, wie sie damit umgehen sollen. Bei einem

500 Statuscode werden technische Probleme serverseitig erwartet. Wenn jedoch eine Fehlermeldung mit einem 200 Statuscode zurückgegeben wird, müssen Entwickler die Antwort analysieren, um das Problem zu identifizieren. Es ist ratsam, spezifische Fehlermeldungen für unsere API einzuführen [6, S.17]

Mangelnde Dokumentation

Probleme können durch fehlende Dokumentation, fehlerhafte Validierung oder Änderungen im Format der API-Antworten entstehen, wenn es unklar ist, was in einer API-Antwort zu erwarten ist. Solche Situationen führen zu Fehlern. Deswegen müssen Entwickler Dokumentation schreiben [6, S.17].

Die Abbildung 5 beschreibt die Mangelnde Dokumentation.

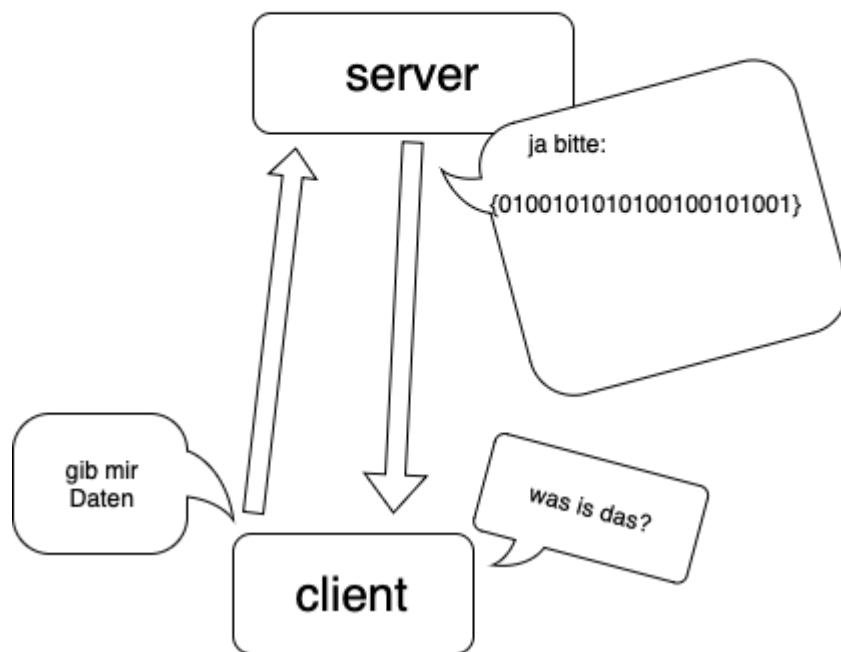


Abbildung 5 Mangelnde Dokumentation

Werkzeuge wie Swagger können dabei helfen. API-Antworten sollten so knapp wie möglich gehalten werden. Übermäßig lange Antworten erfordern viel Rechenleistung, was die API verlangsamen kann [6, S.17].

API Caching

Viele Entwickler ignorieren die Caching-Funktion der API, obwohl sie die Antwortzeiten erheblich verkürzen kann. Der Entwickler erhält nur eine Flagge, die besagt, dass die Ressource durch die zwischengespeicherte Antwort nicht geändert wurde [6, S.17].

2.4 DDD

Domain-Driven Design (DDD) ist eine Softwareentwicklungsmethodik, die den Schwerpunkt auf Geschäftslogik und Domänenmodellierung legt. Domain Driven Design bietet einen soliden, systematischen und umfassenden Ansatz für Softwaredesign und Entwicklung. Es bietet eine Reihe von Werkzeugen und Techniken, die dabei helfen, die Komplexität des Unternehmens zu reduzieren, während das Kerngeschäftsmodell im Mittelpunkt des Ansatzes steht.

DDD ist seit langem ein bevorzugter Ansatz für traditionelle (d. h. monolithische) Projekte, und mit dem Aufkommen der Microservices-Architektur werden DDD-Konzepte zunehmend auch auf dieses neue Architekturparadigma angewendet [12, S.1].

2.4.1 DDD Konzepte

Business Domain

Der Business Domain ist ein grundlegendes Konzept im Domain-Driven Design (DDD). Er dient als Ausgangspunkt für DDD und umfasst die Identifizierung des Hauptgeschäftsproblems, das DDD in einer bestimmten Branche oder Organisation lösen möchte.

Der Zweck des Business Domain besteht darin, den Kontext für die Anwendung der DDD-Prinzipien festzulegen, indem das zentrale Geschäftsproblem, das gelöst werden muss, definiert wird. Diese Identifizierung hilft dabei, die Softwareentwicklung an den realen geschäftlichen Herausforderungen auszurichten. Das Verstehen und Definieren des Problem Space oder der Business Domain ist die Grundlage des DDD-Prozesses. Die Identifizierung des Problem Space sorgt für Klarheit in Bezug auf die Ziele der DDD. Sie hilft den Stakeholdern und den Entwicklungsteams, ein gemeinsames Verständnis für das Hauptgeschäftsproblem zu entwickeln, das gelöst werden muss. [11].

Sub-Domain und Bounded Contexts

Sub-Domains sind wie die Unterteilung eines komplexen Unternehmens in einfach zu handhabende Teile, von denen jeder seine eigene spezifische Rolle und Funktion hat. Diese Organisation hilft beim Aufbau effektiver Lösungen für verschiedene Aspekte des Unternehmens. Subdomains machen es einfacher, verschiedene Aspekte eines Unternehmens zu verstehen und separat zu verwalten. Es ist zum Beispiel wie die Aufgabe, ein Fahrrad

herzustellen. eine Aufgabe, die weitere Unteraufgaben hat. Wie ein Rad, ein Lenkrad und Bremsen Herstellung.

Durch die Identifizierung und Definition von Sub-Domains kann man Lösungen und Systeme entwickeln, die für jeden Bereich des Unternehmens spezifisch sind. So kann man klare Grenzen ziehen und sicherstellen, dass jeder Teil gut funktioniert.

Bounded Contexts sind Designlösungen für die identifizierte Business Domains und Sub-Domains [11].

2.4.2 Domain Model

Das Domain Model ist die Implementierung der Kerngeschäftslogik innerhalb eines bestimmten Bounded Contexts. Es definiert, wie die Geschäftskomponenten wie Entitäten, Regeln, Abläufe, Operationen und Events in Software als Aggregat, Sagas, Commands, Events dargestellt und implementiert werden. [11].

Aggregates

Das Aggregat ist das zentrale Geschäftsobjekt in Bounded Context und definiert den Umfang der Konsistenz innerhalb dieses Bounded Contexts. [11].

Domain Rules sind Unternehmensregeln. [11].

Commands and Queries stellen alle Arten von Operationen innerhalb des Bounded Context dar, die entweder den Zustand des Aggregats ändern oder den Zustand des Aggregats abfragen. [11].

Events zeigen jede Art von Zustandsänderung entweder bei einem Aggregat oder einer Entität innerhalb des Bounded Context. [11].

Sagas reagieren auf mehrere Geschäftsereignisse in verschiedenen Bounded Contexts und "orchestrieren den Geschäftsprozess", indem sie die Interaktionen zwischen diesen Bounded Contexts koordinieren. [11].

2.5 Load Testing

Load Testing ist die Bewertung einer Anwendung unter Last. So kann man feststellen, wie sich das System unter echter Last verhält und mögliche Schwachpunkte identifizieren.. Für ein erfolgreiches Testen ist es notwendig, Bewertungskriterien zu definieren und spezielle Werkzeuge zu verwenden.

2.5.1 Bewertungskriterien

Zu den Kriterien für die Leistungsbewertung gehören Parameter wie Responsezeit und Durchsatz. Diese helfen bei der Bewertung der Leistung des Systems unter Last.

2.5.2 Gatling

Gatling ist ein Framework für die Durchführung von Leistungstests von Webanwendungen. Es wurde entwickelt, um Last- und Stresstests zu erstellen und die Systemleistung unter verschiedenen Belastungen zu analysieren.

Die wichtigsten Aspekte von Gatling sind:

1. eine DSL in Scala: Gatling bietet eine DSL (Domain-Specific Language) in der Programmiersprache Scala zur Erstellung von Testszenarien. Diese DSL ermöglicht es Ihnen, Benutzeraktionen und Testszenarien in einem sauberen und deklarativen Stil zu beschreiben.
2. asynchrone und Lasttests: Gatling wurde für die Arbeit mit asynchronen Systemen entwickelt und kann Webanwendungen einer hohen Last aussetzen, um ihre Leistung und Stabilität unter verschiedenen Bedingungen zu testen.
3. Berichterstellung: Gatling erstellt automatisch detaillierte Testberichte mit Grafiken und Statistiken zu Antwortzeiten, Durchsatz, Fehlern und anderen Leistungskennzahlen. Dies hilft Entwicklern und Ingenieuren, Probleme schnell zu erkennen und Anwendungen zu optimieren.
4. Unterstützung verschiedenen Protokolle: Gatling unterstützt verschiedene Protokolle wie HTTP, WebSocket und GRPC, so dass Sie eine Vielzahl von Webanwendungen und -diensten testen können.
5. Offene und freie Software: Gatling ist eine offene und freie Software, die einem breiten Publikum von Entwicklern und Ingenieuren zugänglich ist.
- 6) Plugin-Ökosystem: Es gibt ein reichhaltiges Ökosystem von Plugins für Gatling, mit denen Sie die Funktionalität erweitern und in verschiedene Überwachungssysteme und Analysetools integrieren können.

Gatling ist ein Tool für die Durchführung von Last- und Leistungstests von Webanwendungen und Webdiensten. Es liefert zuverlässige Ergebnisse, so dass Probleme mit der Anwendungsleistung erkannt und behoben werden können, bevor sie in der Produktion eingesetzt werden [12].

Gatling liefert solche Tabelle wie in Abbildung 6.

Requests ^	Executions					Response Time (ms)							
	Total ↕	OK ↕	KO ↕	% KO ↕	Cnt/s ↕	Min ↕	50th pct ↕	75th pct ↕	95th pct ↕	99th pct ↕	Max ↕	Mean ↕	Std Dev ↕
All Requests	660	660	0	0%	94.286	2	18	175	308	362	396	83	108
CreateUserReq	220	220	0	0%	31.429	61	214	284	353	376	396	222	78
Verify Created User	220	220	0	0%	31.429	5	16	23	33	36	36	17	9
Check User status is verified	220	220	0	0%	31.429	2	6	8	76	78	79	11	17

Was die Abkürzungen bedeuten:

Total: Gesamtzahl der Ausführungen des Ergebnis.

OK: Erfolgreiche Events.

KO: Fehlgeschlagene Events.

% KO: Prozentsatz der Fehlgeschlagene Events.

Cnt/s: Anzahl des Events pro Sekunde.

Response Time

Min: Die schnellste Antwortzeit.

Percentiles: Wie die Antwortzeit im Vergleich zu anderen ist.

Max: Die langsamste Antwortzeit.

Mean: Die durchschnittliche Antwortzeit.

StdDev: Zeigt wie weit weichen die Antwortzeit vom Durchschnitt ab.

Im folgenden Beitrag wird untersucht, wie diese Kernkomponenten miteinander interagieren und wie sie im Zusammenhang mit der Untersuchung der Leistung von GRPC- und REST-Einstiegspunkten in Anwendung eingesetzt werden.

2.6 Axon

Das Axon Framework ist ein leistungsfähiges Tool zur Implementierung von Domain-Driven Design (DDD), Command Query Responsibility Segregation (CQRS) und Event Sourcing (ES) Patterns. Es vereinfacht die Implementierung komplexer Muster und ermöglicht es Entwicklern, sich auf die Geschäftslogik zu konzentrieren [11].

Axon bietet die folgenden Komponenten:

- **Domain Model.** Ein Kernframework, das bei der Erstellung eines Domain-Modells unterstützt, das auf DDD, Event Sourcing und CQRS Mustern basiert.
- **Dispatch Model.** Logische Infrastruktur zur Unterstützung Domain Modells, die Weiterleitung und Koordinierung von Befehlen und Abfragen, die den Zustand des Domain Modells betreffen.
- **Axon Server.** Physische Infrastruktur zur Unterstützung Domain Dispatch-Modells [11].

2.6.1 Domain Model Komponenten

Aggregate: Dies sind Kernkomponenten, die Geschäftseinheiten innerhalb eines Bounded Context darstellen und kapseln. Aggregate enthalten einen Zustand und Methoden zur Änderung dieses Zustands. Sie werden mit `@Aggregate` annotiert.

Commands und Command Handlers: Command stellen Aktionen zur Änderung des Zustands von Aggregaten dar. Command Handlers, die mit `@CommandHandler` annotiert sind, definieren, wie diese Command zu behandeln sind und Zustandsänderungen einleiten.

Events und Event Handlers: Events werden als Ergebnis der Verarbeitung von Befehlen erzeugt. Event-Handler, die mit `@EventHandler` annotiert sind, abonnieren Ereignisse und aktualisieren den Zustand des Aggregats.

Query Handler: Queries werden verwendet, um den Zustand von Aggregaten abzufragen. Query Handler, die mit `@QueryHandler` annotiert sind, definieren, wie Abfragen zu behandeln sind und geben Daten aus dem Read Model zurück [11]

2.6.2 Dispatch Model Komponenten

Command Bus: Der Command Bus sendet Command an die entsprechenden Command Handler innerhalb eines Bounded Contexts. Er hilft bei der Weiterleitung von Befehlen an ihre entsprechenden Handler.

Query Bus: Der Query Bus leitet Abfragen an die entsprechenden Query Handler weiter und ermöglicht so den Abruf von Daten aus Aggregaten.

Event Bus: Der Event Bus bekommt Events, die von Command Handlers generiert werden und leitet sie an Event Handlers in verschiedenen Bounded Contexts weiter. Er ermöglicht die Kommunikation zwischen verschiedenen Teilen des Systems.

Sagas: Sagas stellen langlaufende Prozesse dar, die als Reaktion auf Event Aktionen über Bounded Contexts hinweg koordinieren.

Axon Server vereinfacht die Implementierung von CQRS- und ES-Mustern und erleichtert Entwicklern die Erstellung ereignisgesteuerter Microservices. Er kombiniert DDD, CQRS und ES zu einer umfassenden Lösung für die Erstellung robuster Anwendungen.

[11]

Kapitel 3 Anwendungsmodellierung und Entwicklung

Dieses Kapitel gibt einen Überblick über das Design der implementierten Anwendung, die einem Microservices-Ansatz folgt. Die Anwendung besteht aus drei Bounded Contexts, die nach den Prinzipien des Domain-Driven Design (DDD) definiert sind, und nutzt vier Microservices. Drei Microservices sind für jeden Bounded Context zuständig, wobei ein Microservice als Vermittler für Transaktionen zwischen den Bounded Contexts fungiert.

Jeder Mikroservice bietet die beiden oben beschriebenen Kommunikationsschnittstellen REST und GRPC.

Intern kommunizieren die Microservices untereinander über GRPC, da das Axon-Framework so aufgebaut ist.

3.1 Anwendung

Die entwickelte Anwendung ist ein Nachbau einer Blogging-Anwendung, die Funktionalitäten für einen bestimmten Benutzer bietet, einen Artikel zu erstellen, der veröffentlicht werden kann und anderen Benutzern die Möglichkeit bietet, Kommentare zu hinterlassen.

Die Idee für die Anwendung wurde von einem RealWorld [13] Anwendungsbeispiel übernommen.

RealWorld ist ein Open-Source-Projekt, das produktionsreife Anwendungsdemos für verschiedene Frameworks und Tech-Stacks bereitstellt. RealWorld bietet ein Github-Repository [14] mit Anwendungsbeispielen, die reale Anwendungsanforderungen darstellen.

Die Anwendung wurde nach den Prinzipien des Domain Driven Design mit Spring Boot und Axon Framework neu entwickelt.

3.2 Anwendungsarchitektur

Das Anwendungsarchitektur ist ein entscheidender Aspekt der Entwicklung, insbesondere bei komplexen Systemen. Die Anwendung der Grundsätze des bereichsorientierten Designs (DDD) ermöglicht die Strukturierung der Anwendung um den jeweiligen Bereich herum und betont die wichtigsten Aggregate, Entitäten und wichtigen Events. Dies macht die Anwendung leichter verständlich und anpassungsfähiger an Änderungen der Geschäftslogik.

Dieses Kapitel gibt einen Überblick über das Design der implementierten Anwendung, die einem Microservices-Ansatz folgt. Die Anwendung besteht aus drei Bounded Contexts, die nach den

Prinzipien des Domain-Driven Design (DDD) und CQRS definiert sind, und nutzt vier Microservices. Drei Microservices sind für jeden Bounded Context zuständig, wobei ein Microservice als Vermittler für Transaktionen zwischen den Bounded Contexts agiert.

Jeder Microservice bietet die beiden oben beschriebenen Kommunikationsschnittstellen REST und GRPC. Die Abbildung 7 beschreibt die Kommunikationsschnittstellen

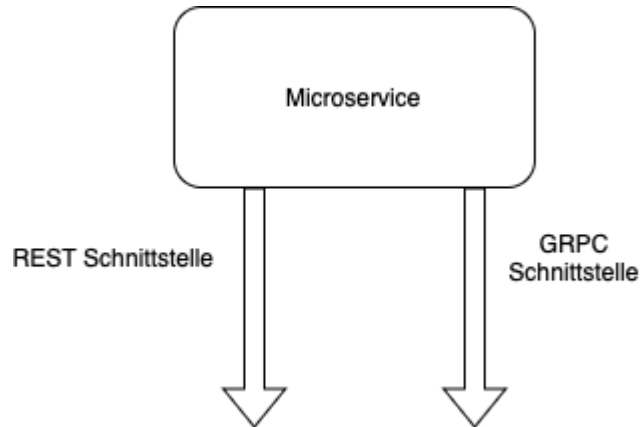


Abbildung 7 Kommunikationsschnittstellen

Intern kommunizieren die Microservices untereinander über GRPC, da das Axon-Framework so aufgebaut ist.

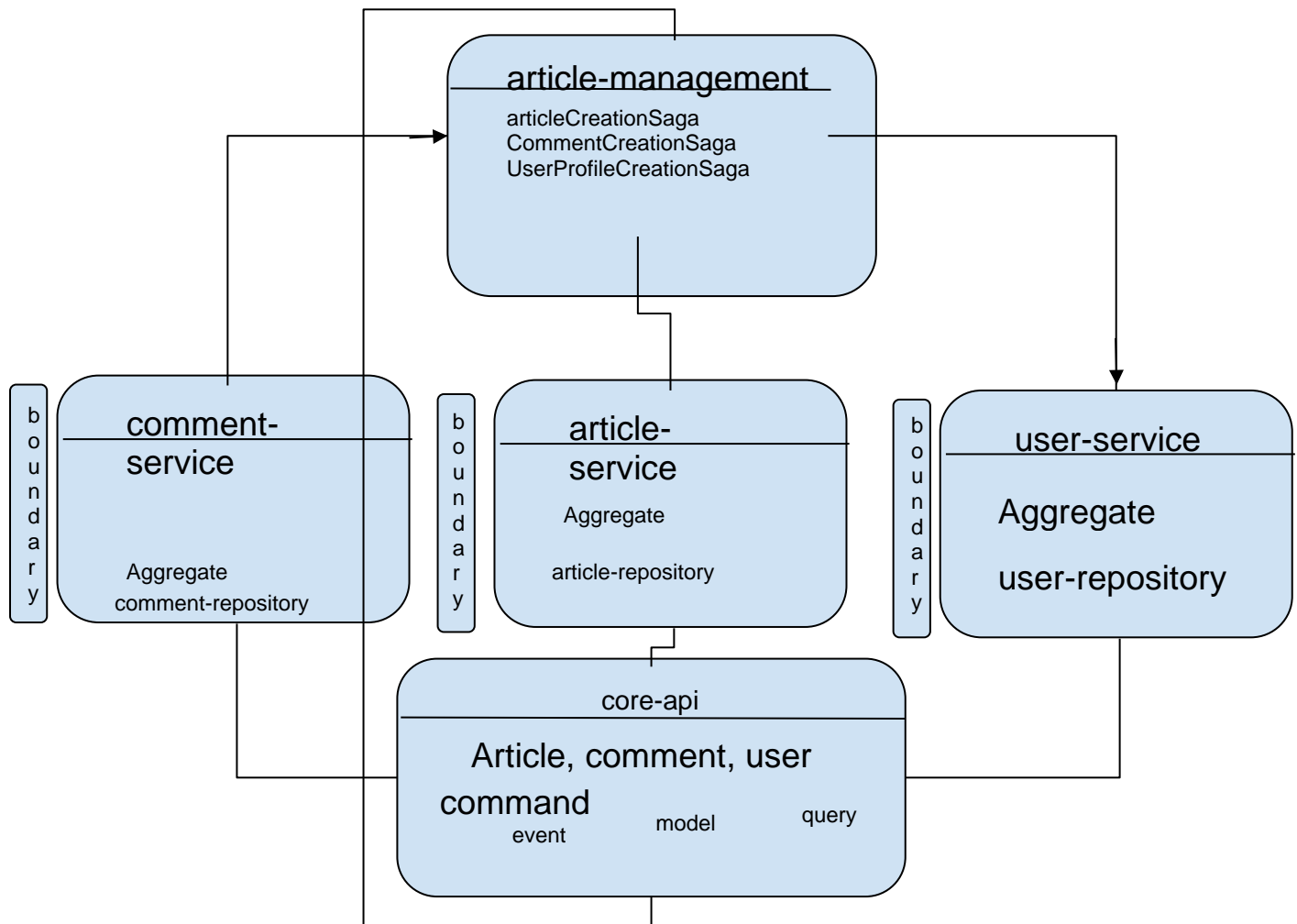


Abbildung 8 Microservices Communication Scheme

Die Abbildung 8 beschreibt die Anwendungsarchitektur.

Artikel-management

Dieser Microservice fungiert als Vermittler für Transaktionen zwischen anderen Microservices. Er bietet drei Sagas, um die ordnungsgemäße Erstellung verschiedener Komponenten sicherzustellen: `ArticleCreationSaga`, die die Erstellung eines Artikels verifiziert; `CommentCreationSaga`, die sicherstellt, dass die Erstellung eines Kommentars korrekt erfolgt; und `UserProfileCreationSaga`, die die Erstellung eines Benutzerprofils validiert.

Article-Service

Dieser Microservice bietet domänenspezifische Funktionalitäten im Zusammenhang mit Artikeln, wie z. B. die Erstellung, die Veröffentlichung und die Aufhebung der Veröffentlichung von Artikeln.

Comment-Service

Dieser Microservice bietet domänenspezifische Funktionalitäten im Zusammenhang mit Comment, wie z. B. die Erstellung und den Empfang beim Id, beim Autor, beim Artikel.

User-Service

Dieser Microservice bietet domänenspezifische Funktionalitäten im Zusammenhang mit Nutzern, wie z. B. die Erstellung, die Verifizierung und die Empfang beim Id, Status.

Aggregate: Diese Komponente stellt die Domänenmodell-Aggregate dar, die für die Verarbeitung der Geschäftslogik und die Speicherung des Zustands verantwortlich sind.

Boundary: Boundary dient als Einstiegspunkt für externe Anfragen und verwaltet den Datenfluss zwischen Clients und dem Microservice.

Handler: Handler führen spezifische Operationen auf der Grundlage von Eingabebefehlen und Ereignissen aus, die mit der Geschäftslogik des Microservice zusammenhängen.

Model: Das Model enthält Objekte und Entitäten, die die im Microservice verwendeten Geschäftsobjekte und Daten darstellen.

Repository: Repositories sind für den Zugriff auf Datenbanken und die Speicherung von Daten zuständig.

Im nächsten Abschnitt werden die bereitgestellten Client-Kommunikationsschnittstellen vorgestellt.

3.3 REST-Schnittstelle

Die REST-Schnittstelle dient als einer der beiden in Kapitel 2 besprochenen Kommunikation Zugangspunkte zu der implementierten Anwendung und erleichtert die Kommunikation zwischen Clients und dem Backend über eine RESTful API.

Die REST-Schnittstellen wurden mit dem Spring Boot Rest Controller implementiert, der JSON-Anfragen und Antworten unterstützt.

Jeder Bounded Context Micro-Service implementierte einen Rest Controller, welcher Endpunkte für die Kommunikation anbietet.

Article-Service

Bietet die in der Tabelle 2 Article Service Endpunkte beschriebenen Endpunkte

Beschreibung	HTTP-Methode	URL	JSON Inhalt	Beispiele für Antworten	Rückmeldung Status
Einen neuen Artikel anlegen	POST	api/articles	{ "authorID": "", "articleDetails": { "title": "", "description": "", "body": "" } }	4b871ac7-c276-433d	200 OK
Artikel anpassen	PUT	api/articles/{articleId:..+}/update	{ "articleDetails": { "title": "", "description": "", "body": "" } }	Article is being process for update	200 OK
Artikel schließen	POST	api/articles/{articleId:..+}/close		Article is being process for update	200 OK
Einen bestimmten Artikel anhand der angegebenen Id abrufen	GET	api/articles/{articleId:..+}		{ "id": "", "title": "", "description": "", "body": "", "authorID": "" , "status": "", "createdOn": " ", "updatedOn": "" }"	200 OK

Bestimmte Artikel nach dem angegebenen Status abrufen	GET	api/articles/status/{status:.+}		[{ "id": "", "title": "blog1", ", "description": "blog1", "body": "", "authorID": "", , "status": "", "createdOn": "" "}, "updatedAt": "" }]	200 OK
---	-----	---------------------------------	--	--	--------

Tabelle 2 Article Service Endpunkte

Comment-Service

Bietet die in der Tabelle 3 Comment Service Endpunkte beschriebenen Endpunkte

Beschreibung	HTTP-Methode	URL	JSON Inhalt	Beispiele für Antworten	Rückmeldung Status
Einen neuen Kommentar anlegen	POST	api/comments	{ "message": "", "articleId": "", "authorID": "" }	28d0f086-a789-4bc0	200 OK
Einen bestimmten Kommentar anhand der angegebenen Id abrufen	GET	api/comments/{commentId:.+}		{ "id": "", "message": "", "articleID": "", , "authorID": "", , "status": "", "createdOn": ""	200 OK

				<pre> ", "updatedAtOn": "" } </pre>	
Kommentare für einen bestimmten Artikel abrufen	GET	api/comments/article/{articleId:.+}		<pre> [{ "id": "", "message": "", "articleID": "", , "authorID": "", , "status": "", "createdOn": "", "updatedAtOn": "" }] </pre>	200 OK
Kommentare der angegebenen Autoren-ID abrufen	GET	api/comments/author/{authorId:.+}		<pre> [{ "id": "", "message": "", "articleID": "", , "authorID": "", , "status": "", "createdOn": "", "updatedAtOn": "" }] </pre>	200 OK

Tabelle 3 Comment Service Endpunkte

User-Service

Bietet die in der
Tabelle 4 User Service Endpunkte beschriebenen Endpunkte

Beschreibung	HTTP-Methode	URL	JSON Inhalt	Beispiele für Antworten	Rückmeldung Status
Neues Benutzerprofil erstellen	POST	api/profiles	{"userName": "", "name": "", "email": ""}	28d0f086-a789-4bc0	200 OK
Benutzerprofil verifizieren	POST	api/profiles/verify/{profileId:.+}		User is being process for verification.	200 OK
Ein bestimmtes Benutzerprofil mit der angegebenen Id abrufen	GET	api/profiles/{profileId:.+}		{ "id": "", "name": "", "username": "", "email": "", "status": "", "registeredOn": ""}	200 OK
Alle registrierten Benutzerprofile abrufen	GET	api/profiles		[{ "id": "", "name": "", "username": "", "email": "", "status": "", "registeredOn": "" }]	200 OK
Bestimmte Benutzerprofile mit dem angegebenen Status abrufen	GET	api/profiles/status/{status:.+}		[{ "id": "", "name": "", "username": "", "email": "", "status": "", "registeredOn": "" }]	200 OK

Tabelle 4 User Service Endpunkte

3.4 GRPC-Schnittstelle

Die GRPC-Schnittstelle bietet einen alternativen, aber ähnlichen Zugangspunkt (Wie in Abschnitt 3.3 beschrieben) zur implementierten Anwendung und ermöglicht die Interaktion zwischen Clients und dem Backend über einen so genannten Protocol Buffers [15], die als Vertragsbindung zwischen Client und Backend dient.

Wie bei der zuvor beschriebenen REST-Schnittstelle bietet jeder Bounded Context Micro-Service auch eine proto-Datei zur Definition von Nachrichten und Diensten für die Kommunikation zwischen dem Client und dem Server.

Article-Service

Bietet die in der Tabelle 5 beschriebenen RPC-Dienste an.

Beschreibung	Name der Methode	Anfrage Nachricht	Antwort-Nachricht	Rückmeldung Status
Einen neuen Artikel anlegen	rpc createArticle	CreateArticleRequest{ string authorID CreateArticleDetailsarticleDetails;}	StringMessageParam{ string value;}	0 OK
Artikel anpassen	rpc updateArticle	UpdateArticleRequest{ string articleID ; CreateArticleDetailsarticleDetails ; ;}	StringMessageParam{ string value;}	0 OK
Artikel schließen	rpc closeArticle	StringMessageParam{ string value;}	StringMessageParam{ string value;}	0 OK
Einen bestimmten	rpc getArticleById	StringMessageParam{	ArticleResponse {	0 OK

Artikel anhand der angegebenen Id abrufen		string value;}	string id; string title; string description; string body; string authorID; ArtStatus status; google.protobuf.Timestamp createdOn; google.protobuf.Timestamp updatedOn; }	
Bestimmte Artikel nach dem angegebenen Status abrufen	rpc getAllArticleByStatus	GetArticlesByStatusRequest{ ArtStatus status; }	GetAllArticlesResponse{ repeated ArticleResponse articles; }	0 OK

Tabelle 5 Article Service Endpunkte

Comment-Service

Bietet die in der Tabelle 6 beschriebenen RPC-Dienste an

Beschreibung	Name der Methode	Anfrage Nachricht	Antwort-Nachricht	Rückmeldung Status
Einen neuen Kommentar anlegen	rpc createComment	CreateCommentRequest{ string message; string articleId; string authorID; }	StringMessageParam{ string value;}	0 OK
Einen bestimmten Kommentar anhand der angegebenen Id abrufen	rpc getCommentsById	StringMessageParam{ string value;}	CommentResponse{ string id; string articleID; string authorID; stringmessage;C	0 OK

			MStatusstatus; google.protobuf. Timestamp createdOn; google.pr otobuf.Timestam p updatedOn; }	
Kommentare für einen bestimmten Artikel abrufen	rpc getCommentsBy Article	StringMessagePa ram{ string value;}	MultipleComme ntResonse{ repeated CommentRespon se comments; }	0 OK
Kommentare der angegebenen Autoren-ID abrufen	rpc getCommentsBy Author	StringMessagePa ram{ string value;}	MultipleComme ntResonse{ repeated CommentRespon se comments; }	0 OK

Tabelle 6 Comment Service Endpunkte

User-Service

Bietet die in der Tabelle 7 beschriebenen RPC-Dienste an

Beschreibung	Name der Methode	Anfrage Nachricht	Antwort-Nachricht	Rückmeldung Status
Neues Benutzerprofil erstellen	rpc createUserProfile	CreateUserProfileRequest { string userName; string name; string email; }	StringMessagePa ram{ string value;}	0 OK
Benutzerprofil verifizieren	rpc verifyProfile	StringMessagePa ram{ string value;}	StringMessagePa ram{ string value;}	0 OK
Ein bestimmtes Benutzerprofil	rpc getUserProfileB	StringMessagePa ram{	GetUserProfileResponse{	0 OK

mit der angegebenen Id abrufen	yId	string value;}	string id; string name; string username; string email; UPStatus status; google.protobuf. Timestamp registeredOn; }	
Alle registrierten Benutzerprofile abrufen	rpc getUserProfiles	EmptyMessageP aram {}	GetAllUserProfil esResponse{ repeated GetUserProfileR esponse userProfiles; }	0 OK
Bestimmte Benutzerprofile mit dem angegebenen Status abrufen	rpc getUserProfilesB yStatus	GetUserProfiles ByStatusRequest { UPStatus status; }	GetAllUserProfil esResponse{ repeated GetUserProfileR esponse userProfiles; }	0 OK

Tabelle 7 User Service Endpunkte

Kapitel 4 Planung der Experimente

Dieses Kapitel befasst sich mit der Umsetzung von Ziel 2 (siehe Kapitel 1, Abschnitt 1.2), nämlich die Durchführung von Tests zur Leistungsfähigkeit der beiden in Kapitel 3 beschriebenen Kommunikationsschnittstellen. Auf dem Boundet Context für jede Schnittstelle werden Anfragen gleichzeitig aufgerufen wie in Abbildung 9.

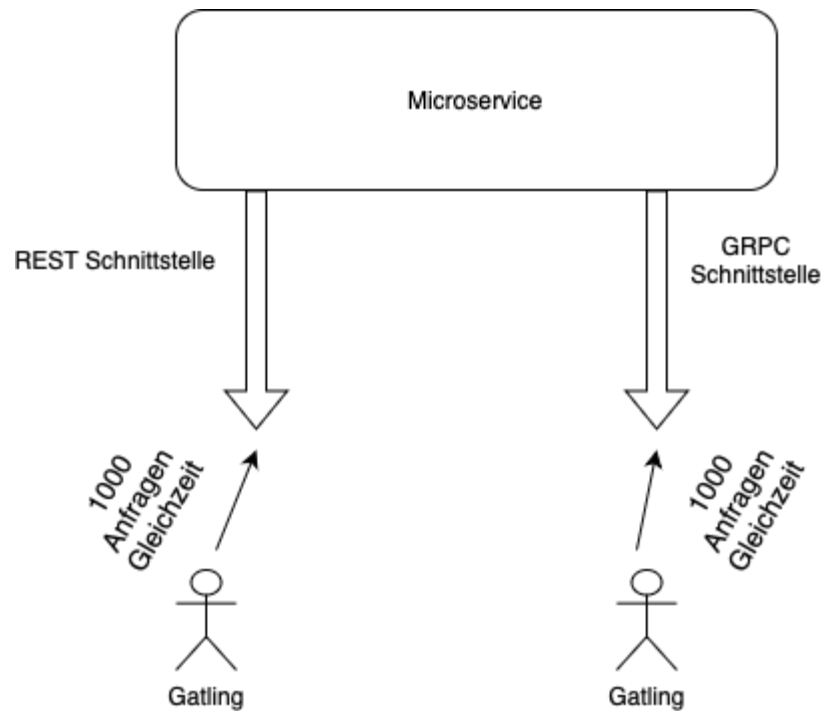


Abbildung 9 Test Planung

Dieses Kapitel ist wie folgt gegliedert:

- Abschnitt 4.1 beschreibt die für die Leistungstests definierten Testszenarien.
- Abschnitt 4.2 beschreibt das System, das für die Testumgebung eingerichtet wurde.

4.1 Test Szenarien

Es wurden drei Use-Case-Szenarien definiert:

1. Benutzer Erstellung Testszenario
2. Artikel Erstellung Testszenario
3. Kommentar Erstellung Testszenario
4. Einzelne POST Anfrage
5. Einzelne GET Anfrage
6. Einzelne PUT Anfrage

Benutzer Erstellung Testszenario

Dieses Testszenario wird in drei Schritten durchgeführt: Zuerst wird ein neuer Benutzer erstellt, dann wird der erstellte Benutzer verifiziert und schließlich wird das Benutzerprofil überprüft, ob der Verifizierungsprozess erfolgreich war.

Die Codeausschnitte Abbildung 10 und Abbildung 11 zeigen die Gatling-Implementierung dieses Szenarios.

```
public class UserCreationRestSimulation extends Simulation {
    Faker faker = new Faker();
    FakeValuesService fakeValuesService = new FakeValuesService(new Locale("en-GB"), new
RandomService());
    GsonBuilder builder = new GsonBuilder();
    Gson gson = builder.create();

    ChainBuilder createUserReq = exec(
        http("CreateUserReq")
            .post("http://localhost:9093/api/profiles/")
            .header("accept", "application/json")
            .header("content-type", "application/json")
            .body(...)
            .check(status().is(200))
            .check(bodyString().saveAs("userID")))
        .pause(Duration.ofMillis(3000))
        .exec(
            http("Verify Created User")
                .post("http://localhost:9093/api/profiles/verify/#{userID}")
                .header("accept", "application/json")
                .header("content-type", "application/json")
                .check(status().is(200)))
        .pause(Duration.ofMillis(3000))
        .exec(http("Check User status is verified")
            .get("http://localhost:9093/api/profiles/#{userID}")
            .header("accept", "application/json")
            .header("content-type", "application/json")
            .check(status().is(200))
            .check(jmesPath("status").ofString().is("VERIFIED")))
    );
}
```

```

ScenarioBuilder scn = scenario("Create 1000 Users").exec(createUserReq);
{
    setUp(scn.injectOpen(atOnceUsers(220)));
}
}

```

Abbildung 10 Gatling-Szenario zur Benutzererstellung mit REST

```

public class UserCreationGRPCSimulation extends Simulation {

    Faker faker = new Faker();
    FakeValuesService fakeValuesService = new FakeValuesService(new Locale("en-GB"), new
RandomService());

    StaticGrpcProtocol userServiceProtocol =
grpc(ManagedChannelBuilder.forAddress("localhost", 7073).usePlaintext());

    GrpcCallActionBuilder<CreateUserProfileRequest, StringMessageParam> createUserReq =
grpc("Create User")
        .rpc(UserGrpc.getCreateUserProfileMethod())

        .payload(...)
        .check(statusCode().is(Status.Code.OK))
        .check(
            extract((StringMessageParam m) -> m.getValue())
                .saveAs("userID"));

    GrpcCallActionBuilder<StringMessageParam, StringMessageParam> verifyUserReq =
grpc("Verify User")
        .rpc(UserGrpc.getVerifyProfileMethod())
        .payload(...)
        .check(statusCode().is(Status.Code.OK));

    GrpcCallActionBuilder<StringMessageParam, GetUserProfileResponse> checkUserisVerified
= grpc(
        "Check user Verification")
        .rpc(UserGrpc.getGetUserProfileByIdMethod())
        .payload(session ->
StringMessageParam.newBuilder().setValue(session.getString("userID"))
            .build())
        .check(statusCode().is(Status.Code.OK))
        .check(extract(GetUserProfileResponse::getStatus).is(UPStatus.UP_VERIFIED));

    ChainBuilder userCreationSteps = exec(createUserReq)

        .pause(Duration.ofMillis(3000))
        .exec(verifyUserReq)
        .pause(Duration.ofMillis(3000))
        .exec(checkUserisVerified)
        .exitHereIfFailed();

    ScenarioBuilder scn = scenario("Create 1000 Users").exec(userCreationSteps);
}

```



```

GrpcCallActionBuilder<EmptyMessageParam, GetAllUserProfilesResponse> getUsers = grpc(
    "Get All Users")
    .rpc(UserGrpc.getGetUserProfilesMethod())
    .payload(session -> EmptyMessageParam.newBuilder().build())
    .check(statusCode().is(Status.Code.OK));

ScenarioBuilder scn2 = scenario("Get Users").exec(createUserReq);

{
    setUp(scn.injectOpen(atOnceUsers(220)).protocols(userServiceProtocol);
}

}

```

Abbildung 11 Gatling-Szenario zur Benutzererstellung mit GRPC

Artikel Erstellung Testszenario

Ähnlich wie bei der Erstellung eines Benutzers umfasst auch dieses Testszenario drei Schritte, nämlich die Erstellung eines neuen Artikels mit einer bestimmten Benutzer-ID, dann die Prüfung, ob der Artikel veröffentlicht wurde, und schließlich die Aktualisierung des Artikels.

Die Codeausschnitte Abbildung 12 und Abbildung 13 zeigen die Gatling-Implementierung dieses Szenarios.

```

public class ArticleUseCaseGrpcTest extends Simulation {
    StaticGrpcProtocol articleService = grpc(ManagedChannelBuilder.forAddress("localhost",
7071).usePlaintext());

    GrpcCallActionBuilder<CreateArticleRequest, StringMessageParam> createArticle = grpc(
        "Create new article")
        .rpc(ArticleGrpc.createArticleMethod())
        .payload(...)
        .check(statusCode().is(Status.Code.OK))
        .check(extract((
            StringMessageParam m) -> m.getValue()).saveAs("articleID"));
    GrpcCallActionBuilder<StringMessageParam, ArticleResponse> getArticleById = grpc(
        "Get article by ID and check if status is published")
        .rpc(ArticleGrpc.getGetArticleByIdMethod())
        .payload(session -> StringMessageParam.newBuilder()
            .setValue(session.getString("articleID"))
            .build())
        .check(statusCode().is(Status.Code.OK))
        .check(extract(ArticleResponse::getStatus).is(ArtStatus.Art_PUBLISHED));

    GrpcCallActionBuilder<UpdateArticleRequest, StringMessageParam> updateArticle = grpc(
        "Update the article")
        .rpc(ArticleGrpc.updateArticleMethod())
        .payload(...)

```

```

        .check(statusCode().is(Status.Code.OK));

ChainBuilder articleCreationBuilder = exec(createArticle)
    .pause(Duration.ofSeconds(12))
    .exec(getArticleByID)
    .exec(updateArticle);

ScenarioBuilder articleCreationScn = scenario("Create
article").exec(articleCreationBuilder);

{
    setUp(articleCreationScn.injectOpen(atOnceUsers(220)).protocols(articleService);
}

}

```

Abbildung 12 Gatling-Szenario zur Artikel Erstellung mit GRPC

```

public class ArticleRestTest extends Simulation {
    Faker faker = new Faker();
    FakeValuesService fakeValuesService = new FakeValuesService(new Locale("en-GB"), new
RandomService());

    GsonBuilder builder = new GsonBuilder();
    Gson gson = builder.create();

    ChainBuilder createArticleReq = exec(
        http("Create new article")
            .post("http://localhost:9091/api/articles/")
            .header("accept", "application/json")
            .header("content-type", "application/json")
            .body(StringBody(session ->
gson.toJson(ArticleCreationRequestDTO.builder()
            .articleDetails(new
ArticleDetails(faker.name().fullName(), faker.name().fullName(), faker.name().fullName()))
            .authorID("01c34530-7936-4d61-88f0-452228ae9c9b")
            .build()))))
            .check(status().is(200))
            .check(bodyString().saveAs("articleID")))
        .pause(Duration.ofMillis(12000))
        .exec(
            http("get created article")
                .get("http://localhost:9091/api/articles/#{articleID}")
                .header("accept", "application/json")
                .header("content-type", "application/json")
                .check(status().is(200))
                .check(jmesPath("status").ofString().is("PUBLISHED"))
            )
    )

    //
        .pause(Duration.ofMillis(1_000))
        .exec(http("update article")
            .put("http://localhost:9091/api/articles/#{articleID}/update")

```

```

        .header("accept", "*/*")
        .header("content-type", "application/json")
        .body(StringBody(session -> gson.toJson(UpdateArticleRequestDTO.builder()
            .articleDetails(new
ArticleDetails(faker.name().lastName(), faker.name().fullName(), faker.name().fullName())
                .build()))
            .build()))
        .check(status().is(200))
    )
    ;

    ScenarioBuilder scn = scenario("Create articles scenario").exec(createArticleReq);
    {
        setUp(scn.injectOpen(atOnceUsers(220)));
    }
}

```

Abbildung 13 Gatling-Szenario zur Artikel Erstellung mit REST

Kommentar Erstellung Testszenario

Dieses Testszenario umfasst zwei Schritte: Zunächst wird ein Kommentar mit einer bestimmten Benutzer-ID und Artikel-ID erstellt, dann wird geprüft, ob der erstellte Kommentar veröffentlicht wurde.

Die Codeausschnitte Abbildung 14 und Abbildung 15 zeigen die Gatling-Implementierung dieses Szenarios.

```

public class CommentGrpcTest extends Simulation {

    Faker faker = new Faker();

    StaticGrpcProtocol commentService = grpc(ManagedChannelBuilder.forAddress("localhost",
7072).usePlaintext());

    GrpcCallActionBuilder<CreateCommentRequest, StringMessageParam> createCommentReq =
grpc("Create Comment")
        .rpc(CommentGrpc.getCreateCommentMethod())

        .payload(session -> CreateCommentRequest.newBuilder()
            .setMessage(faker.name().fullName())
            .setArticleId("d00a8e3c-0636-4b3d-95fa-2ae16f524522")
            .setAuthorID("6faf9dab-18c2-4661-85b2-a021cb3911a9")
            .build())
        .check(statusCode().is(Status.Code.OK))
        .check(
            extract((StringMessageParam m) -> m.getValue())
                .saveAs("commentId"));

    GrpcCallActionBuilder<StringMessageParam, CommentResponse> getCommentsByIdReq = grpc("Get
created article")
        .rpc(CommentGrpc.getGetCommentsByIdMethod())
        .payload(session ->
StringMessageParam.newBuilder().setValue(session.getString("commentId"))
            .build())

```

```

        .check(statusCode().is(Status.Code.OK));

    GrpcCallActionBuilder<StringMessageParam, MultipleCommentResonse> getCommentsByArticleReq
= grpc(
    " comments by article")
    .rpc(CommentGrpc.getGetCommentsByArticleMethod())
    .payload(session ->
StringMessageParam.newBuilder().setValue(session.getString("commentId"))
    .build())
    .check(statusCode().is(Status.Code.OK));

    ChainBuilder userCreationSteps = exec(createCommentReq)

        .pause(Duration.ofMillis(12000))
        .exec(getCommentsByIdReq)
        .exec(getCommentsByArticleReq)
        .exitHereIfFailed();

    ScenarioBuilder scn = scenario("Create 1000 Users").exec(userCreationSteps);

    GrpcCallActionBuilder<EmptyMessageParam, GetAllUserProfilesResponse> getUsers = grpc(
    "Get All Users")
    .rpc(UserGrpc.getGetUserProfilesMethod())
    .payload(session -> EmptyMessageParam.newBuilder().build())
    .check(statusCode().is(Status.Code.OK));

    {
        setUp(scn.injectOpen(atOnceUsers(900)).protocols(commentService);
    }
}

```

Abbildung 14 Gatling-Szenario zur Comment Erstellung mit GRPC

```

public class CommentRestTest extends Simulation {
    Faker faker = new Faker();
    GsonBuilder builder = new GsonBuilder();
    Gson gson = builder.create();

    ChainBuilder createArticleReq = exec(
        http("Create new comment")
            .post("http://localhost:9092/api/comments/")
            .header("accept", "application/json")
            .header("content-type", "application/json")
            .body(StringBody(session ->
gson.toJson(CreateCommentRequestDTO.builder()
                .message(faker.name().fullName())
                .authorID("6faf9dab-18c2-4661-85b2-a021cb3911a9")
                .articleId("d00a8e3c-0636-4b3d-95fa-
2ae16f524522")

```

```

        .build()))
        .check(status().is(200))
        .check(bodyString().saveAs("commentId")))
    .pause(Duration.ofMillis(12000))
    .exec(
        http("get created comment")
            .get("http://localhost:9092/api/comments/#{commentId}")
            .header("accept", "application/json")
            .header("content-type", "application/json")
            .check(status().is(200))
        )
    .pause(Duration.ofMillis(1000))
    .exec(
        http("get comments by article ")
            .get("http://localhost:9092/api/comments/article/#{commentId}")
            .header("accept", "application/json")
            .header("content-type", "application/json")
            .check(status().is(200))
        )
    );

    ScenarioBuilder scn = scenario("Create articles
scenario").exec(createArticleReq);
    {
        setUp(scn.injectOpen(atOnceUsers(800)));
    }
}

```

Abbildung 15 Gatling-Szenario zur Comment Erstellung mit REST

Da die Kapazität für die gleichzeitige Bearbeitung von Anfragen unbekannt ist, werden die Tests mit einer schrittweisen Erhöhung der Anzahl der Benutzer durchgeführt. Ist die Anzahl der Benutzer stabilisiert, wird der Test 10-mal mit dieser stabilisierten Anzahl von Benutzern wiederholt. Das arithmetische Mittel der Test Durchlaufzeit und der maximalen Anzahl der Benutzer wird dann als entscheidender Faktor für den Vergleich der beiden Protokolle verwendet

Um einen einheitlichen Ausgangszustand für jeden Testlauf zu erhalten, gibt es bestimmte obligatorische manuelle Schritte, die in jedem Test enthalten sind. Die Methode zur Durchführung des Tests sieht wie folgt aus:

- Neustart des Docker-Containers, auf dem das zu testende System ausgeführt wird.
- Löschen aller Datenbank- und Event-Store-Daten
- Konfigurieren von Gatling zum Laden des entsprechenden Testfalls
- Ausführen der Tests

4.2 Test-Umgebung

Die Anwendung wurde in einem Docker-Container ausgeführt, während die Gatling-Tests direkt auf dem Host-Computer stattfanden. Tabelle 8 Test Umgebung enthält die Spezifikationen des Computersystems

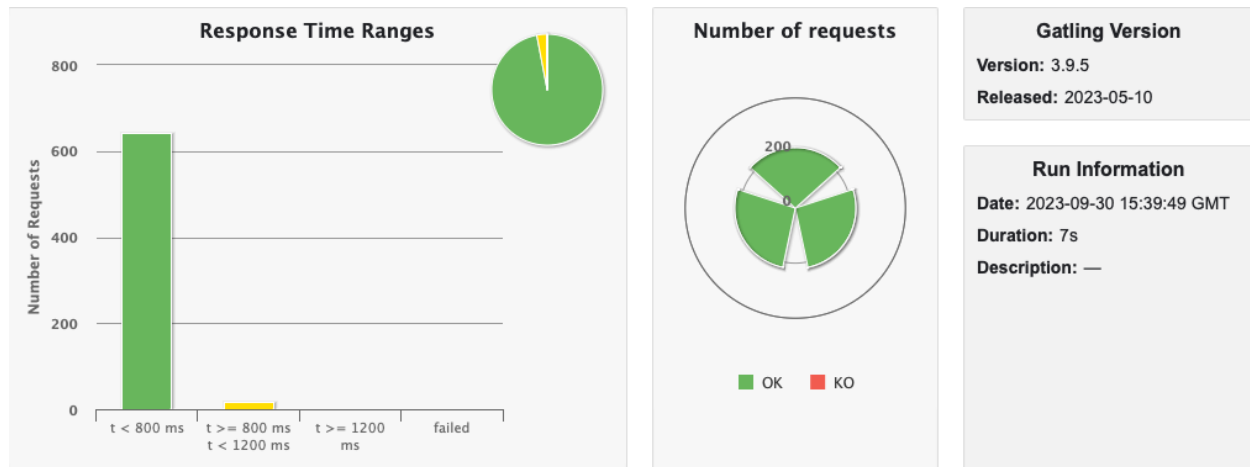
CPU	3.2 GHz 10-core processor. Arm 64
CPU-Geschwindigkeit	3.2 GHz
RAM Speicher	32GB
Betriebssystem	Mac OS Ventura 13.5

Tabelle 8 Test Umgebung

Kapitel 5 Darstellung der Messergebnisse und Auswertung

5.1 Nutzer Erstellung Scenario REST vs GRPC

Die Abbildung 16 beschreibt das REST Test Ergebnis.



<div>Full Screen</div> <div>Expand all groups</div> <div>Collapse all groups</div>													
Requests ^	Executions					Response Time (ms)							
	Total ↕	OK ↕	KO ↕	% KO ↕	Cnt/s ↕	Min ↕	50th pct ↕	75th pct ↕	95th pct ↕	99th pct ↕	Max ↕	Mean ↕	Std Dev ↕
All Requests	660	660	0	0%	82.5	2	10	440	631	1088	1097	188	275
CreateUserReq	220	220	0	0%	27.5	289	502	608	1076	1094	1097	546	183
Verify Created User	220	220	0	0%	27.5	4	9	12	20	24	25	10	5
Check User status is verified	220	220	0	0%	27.5	2	5	7	12	72	74	8	12

Abbildung 16 Nutzer Erstellung Scenario REST Ergebnis

Die Abbildung 17 beschreibt das GRPC Test Ergebnis.

<div>Full Screen</div> <div>Expand all groups</div> <div>Collapse all groups</div>													
Requests ^	Executions					Response Time (ms)							
	Total ↕	OK ↕	KO ↕	% KO ↕	Cnt/s ↕	Min ↕	50th pct ↕	75th pct ↕	95th pct ↕	99th pct ↕	Max ↕	Mean ↕	Std Dev ↕
All Requests	825	825	0	0%	117.857	2	10	191	431	501	549	102	148
Create User	275	275	0	0%	39.286	76	286	373	491	511	549	289	115
Verify User	275	275	0	0%	39.286	4	8	13	26	39	39	11	7
Check user Verification	275	275	0	0%	39.286	2	5	7	22	26	30	7	6

Abbildung 17 Nutzer Erstellung Scenario GRPC Ergebnis

Von dem Load Test Ergebnis Vergleich Tabelle 9 kann man sehen, dass die Kapazität für die gleichzeitige Bearbeitung von GRPC 275 ist und von REST 220. Somit ist dies 25% höher. Die GRPC-Antwortzeit von 75th pct ist 191ms, was 56% schneller als von REST. Und die GRPC Antwortzeit ist im Durchschnitt (Mean) 45% schneller.

Requests	Executions Compare				
	Total	OK	KO	% KO	Cnt/s
All Requests	25.00%	25.00%	0	0%	233.33%
Create User	25.00%	25.00%	0	0%	233.33%

Verify User	25.00%	25.00%	0	0%	233.33%
Check user Verification	25.00%	25.00%	0	0%	233.33%

Requests	Response Time (ms)							
	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
All Requests	0,00%	0,00%	-56,59%	-31,70%	-53,95%	-49,95%	-45,74%	-46,18%
Create User	-73,70%	-43,03%	-38,65%	-54,37%	-53,29%	-49,95%	-47,07%	-37,16%
Verify User	0,00%	-11,11%	8,33%	30,00%	62,50%	-49,95%	10,00%	40,00%
Check user Verification	0	0,00%	0,00%	83,33%	-63,89%	-59,46%	-12,50%	-50,00%

Tabelle 9 Vergleich von GRPC und REST Nutzer Erstellung

GRPC Vergleich mit gleichen Nutzern Anzahl Tabelle 10 zeigt das Antwortzeit von 75th pct von allen Request ist 38% schneller als REST, im Durchschnitt (Mean) 37% schneller.

Die Abbildung 18 beschreibt das GRPC Test Ergebnis mit 220 Nutzern.

<div>Full Screen</div> <div>Expand all groups</div> <div>Collapse all groups</div>													
Requests ^	Executions					Response Time (ms)							
	Total ↕	OK ↕	KO ↕	% KO ↕	Cnt/s ↕	Min ↕	50th pct ↕	75th pct ↕	95th pct ↕	99th pct ↕	Max ↕	Mean ↕	Std Dev ↕
All Requests	660	660	0	0%	220	2	15	188	322	350	371	90	112
Create User	220	220	0	0%	73.333	87	231	288	348	359	371	235	69
Verify User	220	220	0	0%	73.333	5	11	31	128	134	142	28	35
Check user Verification	220	220	0	0%	73.333	2	5	7	15	26	28	7	4

Abbildung 18 Nutzer Erstellung Scenario GRPC Ergebnis mit 220 Nutzern

Die Tabelle 10 beschreibt GRPC und REST Vergleich mit 220 Nutzern.

Requests	Executions				
	Total	OK	KO	% KO	Cnt/s
All Requests	0.00%	0.00%	0	0%	14.29%
Create User	0.00%	0.00%	0	0%	14.29%
Verify User	0.00%	0.00%	0	0%	14.29%
Check user Verification	0.00%	0.00%	0	0%	14.29%

Requests	Response Time (ms)							
	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
All Requests	0.00%	40.00%	-37.95%	-32.81%	-53.40%	-50.50%	-37.77%	-42.18%
Create User	-58.48%	-35.06%	-35.03%	-55.11%	-52.29%	-50.50%	-39.74%	-50.27%
Verify User	0.00%	44.44%	50.00%	75.00%	83.33%	-50.50%	50.00%	80.00%
Check user Verification	0	20.00%	28.57%	75.00%	-62.50%	-54.05%	0.00%	-50.00%

Tabelle 10 Vergleich von GRPC und REST mit 220 Nutzern Erstellung

5.2 Artikel Erstellung Scenario REST vs GRPC

Die Abbildung 19 beschreibt REST Test Ergebnis mit 220 Nutzern.

<div>Full Screen</div> <div>Expand all groups</div> <div>Collapse all groups</div>													
Requests ^	Executions					Response Time (ms)							
	Total ↕	OK ↕	KO ↕	% KO ↕	Cnt/s ↕	Min ↕	50th pct ↕	75th pct ↕	95th pct ↕	99th pct ↕	Max ↕	Mean ↕	Std Dev ↕
All Requests	660	660	0	0%	60	3	35	392	585	627	671	172	229
Create new article	220	220	0	0%	20	104	540	575	619	649	671	471	151
get created article	220	220	0	0%	20	3	14	19	28	32	32	15	8
update article	220	220	0	0%	20	5	35	41	48	52	53	32	13

Abbildung 19 Artikel Erstellung Scenario REST Ergebnis mit 220 Nutzern

Load Test Ergebnis Tabelle 11 zeigt das die Kapazität für die gleichzeitige Bearbeitung von GRPC 270 und von REST 220 ist. Was 22% größer ist. Die GRPC Antwortzeit von 75th pct ist 470ms was 19% größer als von REST. Die GRPC-Antwortzeit ist im Durchschnitt (Mean) 33% schneller. DieAbbildung 20 beschreibt das GRPC Test Ergebnis mit 270 Nutzern.

<div>Full Screen</div> <div>Expand all groups</div> <div>Collapse all groups</div>													
Requests ^	Executions					Response Time (ms)							
	Total ↕	OK ↕	KO ↕	% KO ↕	Cnt/s ↕	Min ↕	50th pct ↕	75th pct ↕	95th pct ↕	99th pct ↕	Max ↕	Mean ↕	Std Dev ↕
All Requests	810	810	0	0%	62.308	3	49	470	768	825	843	230	289
Create new article	270	270	0	0%	20.769	213	664	733	819	834	843	619	153
Get article by ID and check if st...	270	270	0	0%	20.769	3	21	30	44	53	58	22	12
Update the article	270	270	0	0%	20.769	6	42	79	97	101	103	50	29

Abbildung 20 Artikel Erstellung Scenario GRPC Ergebnis mit 270 Nutzern

Requests	Executions				
	Total	OK	KO	% KO	Cnt/s
All Requests	22,73%	22,73%	0	0%	3,85%
Create new article	22,73%	22,73%	0	0.00%	3,84%
Get article by ID and check	22,73%	22,73%	0	0.00%	3,84%
Update the article	22,73%	22,73%	0	0.00%	3,84%

Requests	Response Time (ms)							
	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
All Requests	0,00%	40,00%	19,90%	31,28%	31,58%	25,63%	33,72%	26,20%
Create new article	104,81%	22,96%	27,48%	32,31%	28,51%	25,63%	31,42%	1,32%
Get article by ID and check	0,00%	50,00%	57,89%	57,14%	65,63%	25,63%	46,67%	50,00%
Update the article	0,2	20,00%	92,68%	102,08%	94,23%	94,34%	56,25%	123,08%

Tabelle 11 Vergleich von GRPC (270) und REST (220) Artikel Erstellung

Die Abbildung 21 beschreibt das GRPC Test Ergebnis mit 220 Nutzern.

<div>Full Screen</div> <div>Expand all groups</div> <div>Collapse all groups</div>													
Requests ^	Executions					Response Time (ms)							
	Total ↕	OK ↕	KO ↕	% KO ↕	Cnt/s ↕	Min ↕	50th pct ↕	75th pct ↕	95th pct ↕	99th pct ↕	Max ↕	Mean ↕	Std Dev ↕
All Requests	660	660	0	0%	50.769	5	40	340	543	596	610	162	201
Create new article	220	220	0	0%	16.923	107	450	527	576	600	610	431	111
Get article by ID and check if st...	220	220	0	0%	16.923	5	16	25	43	64	71	20	12
Update the article	220	220	0	0%	16.923	8	34	52	64	73	79	37	16

Abbildung 21 Artikel Erstellung Scenario GRPC Ergebnis mit 220 Nutzern

GRPC Vergleich mit gleichen Nutzern Anzahl Tabelle 12 Zeigt das Antwortzeit von 75th pct von allen Requests ist um 13% schneller als REST. Die GRPC Antwortzeit ist im Durchschnitt (Mean) 5% schneller.

Requests	Executions				
	Total	OK	KO	% KO	Cnt/s
All Requests	0.00%	0.00%	0	0%	-15.39%

Create new article	0.00%	0.00%	0	0%	-15.39%
Get article by ID and check if status is published	0.00%	0.00%	0	0%	-15.39%
Update the article	0.00%	0.00%	0	0%	-15.39%

Requests	Response Time (ms)							
	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
All Requests	66.67%	14.29%	-13.27%	-7.18%	-4.94%	-9.09%	-5.81%	-12.23%
Create new article	2.88%	-16.67%	-8.35%	-6.95%	-7.55%	-9.09%	-8.49%	-26.49%
Get article by ID and check if status is published	66.67%	14.29%	31.58%	53.57%	100.00%	-9.09%	33.33%	50.00%
Update the article	0.6	-2.86%	26.83%	33.33%	40.38%	49.06%	15.63%	23.08%

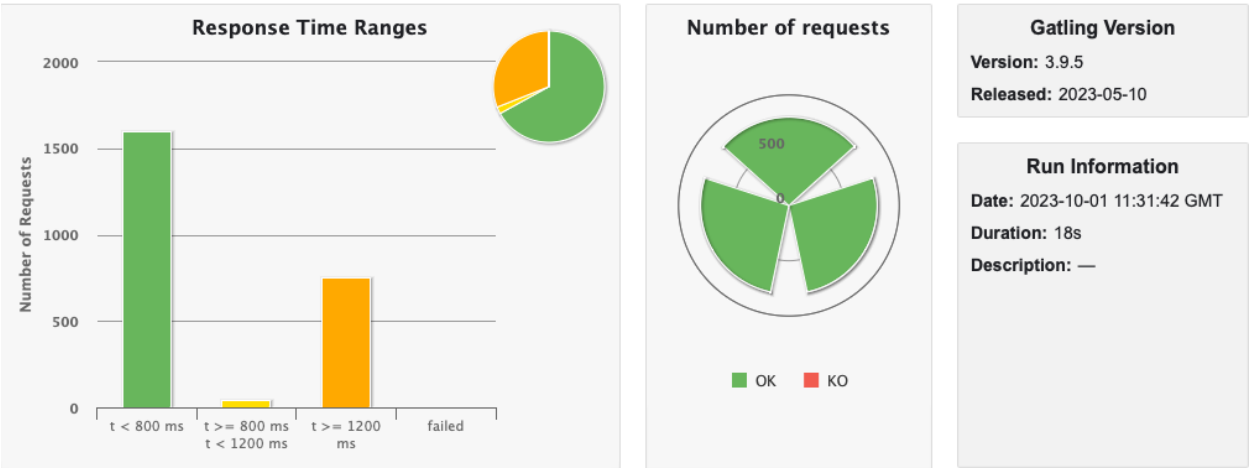
Tabelle 12 Vergleich von GRPC (220) und REST (220) Artikel Erstellung

5.3 Comment Erstellung Scenario REST vs GRPC

Load Test Ergebnis Abbildung 23 zeigt das die Kapazität für die gleichzeitige Bearbeitung von GRPC 900 und Abbildung 22 zeigt das Kapazität von REST 800 ist. Was 12% größer ist. Tabelle 13 zeigt, dass die GRPC Antwortzeit von 75th pct ist 1090ms, was 51% weniger als von REST dessen Antwortzeit 2228 ist. Und die GRPC Antwortzeit im Durchschnitt (Mean) 57% schneller ist.

Die Abbildung 22 beschreibt das REST Test Ergebnis mit 800 Nutzern.

CommentRestTest



Full Screen

Expand all groups

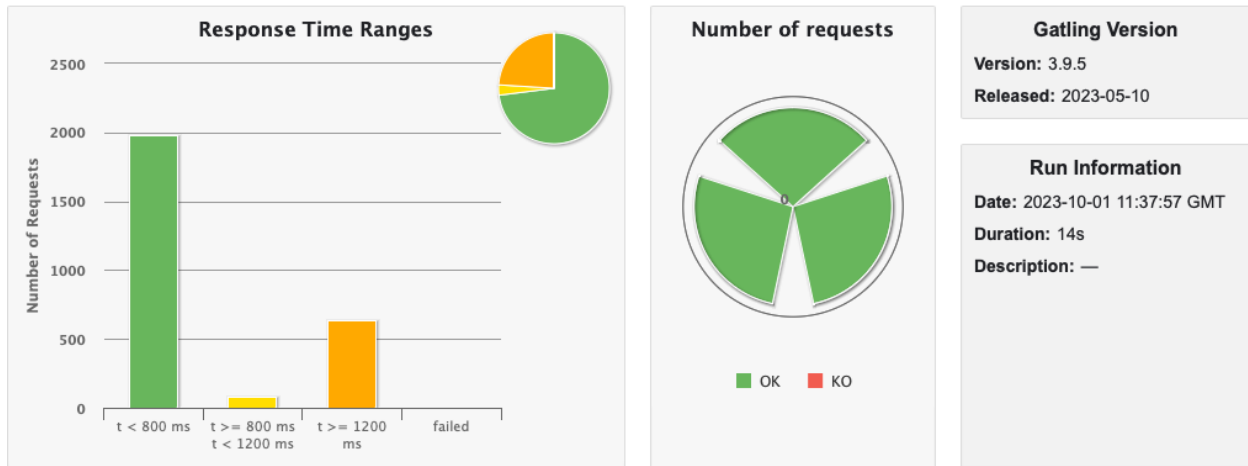
Collapse all groups

Requests ^	Executions					Response Time (ms)							
	Total ↕	OK ↕	KO ↕	% KO ↕	Cnt/s ↕	Min ↕	50th pct ↕	75th pct ↕	95th pct ↕	99th pct ↕	Max ↕	Mean ↕	Std Dev ↕
All Requests	2400	2400	0	0%	126.316	5	48	2228	4837	5361	5495	1125	1723
Create new comment	800	800	0	0%	42.105	815	3334	4462	5285	5422	5495	3305	1334
get created comment	800	800	0	0%	42.105	7	32	48	68	89	99	35	18
get comments by article	800	800	0	0%	42.105	5	32	47	71	106	169	35	21

Abbildung 22 Comment Erstellung Szenario REST Ergebnis mit 800 Nutzern

Abbildung 23 beschreibt das GRPC Test Ergebnis mit 900 Nutzern.

CommentGrpcTest



<div>Full Screen</div> <div>Expand all groups</div> <div>Collapse all groups</div>													
Requests ^	Executions					Response Time (ms)							
	Total ↕	OK ↕	KO ↕	% KO ↕	Cnt/s ↕	Min ↕	50th pct ↕	75th pct ↕	95th pct ↕	99th pct ↕	Max ↕	Mean ↕	Std Dev ↕
All Requests	2700	2700	0	0%	180	3	41	1090	1948	2139	2306	482	720
Create Comment	900	900	0	0%	60	69	1537	1817	2096	2231	2306	1387	572
Get created article	900	900	0	0%	60	3	21	42	87	104	119	30	24
comments by article	900	900	0	0%	60	3	23	40	70	96	119	29	21

Abbildung 23 Comment Erstellung Szenario GRPC Ergebnis mit 900 Nutzern

Requests	Executions				
	Total	OK	KO	% KO	Cnt/s
All Requests	12.50%	12.50%	0	0%	42.50%
Create Comment	12.50%	12.50%	0	0%	42.50%
Get created article	12.50%	12.50%	0	0%	42.50%
comments by article	12.50%	12.50%	0	0%	42.50%

Requests	Response Time (ms)							
	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
All Requests	-40%	-15%	-51%	-60%	-60%	-58%	-57%	-58%
Create Comment	1280%	4703%	3766%	2852%	2005%	1264%	3863%	2624%
Get created article	-57%	-34%	-13%	28%	17%	1264%	-14%	33%

comments by article	-100%	-99%	-99%	-99%	-98%	-98%	-99%	-98%
---------------------	-------	------	------	------	------	------	------	------

Tabelle 13 Vergleich von GRPC (900) und REST (800) Comment Erstellung

GRPC Vergleich mit gleichen Nutzern Anzahl Tabelle 14 Zeigt das die Antwortzeit von 75th pct von allen Request 2.4% langsamer als REST ist und die GRPC Antwortzeit im Durchschnitt (Mean) 15 % schneller ist.

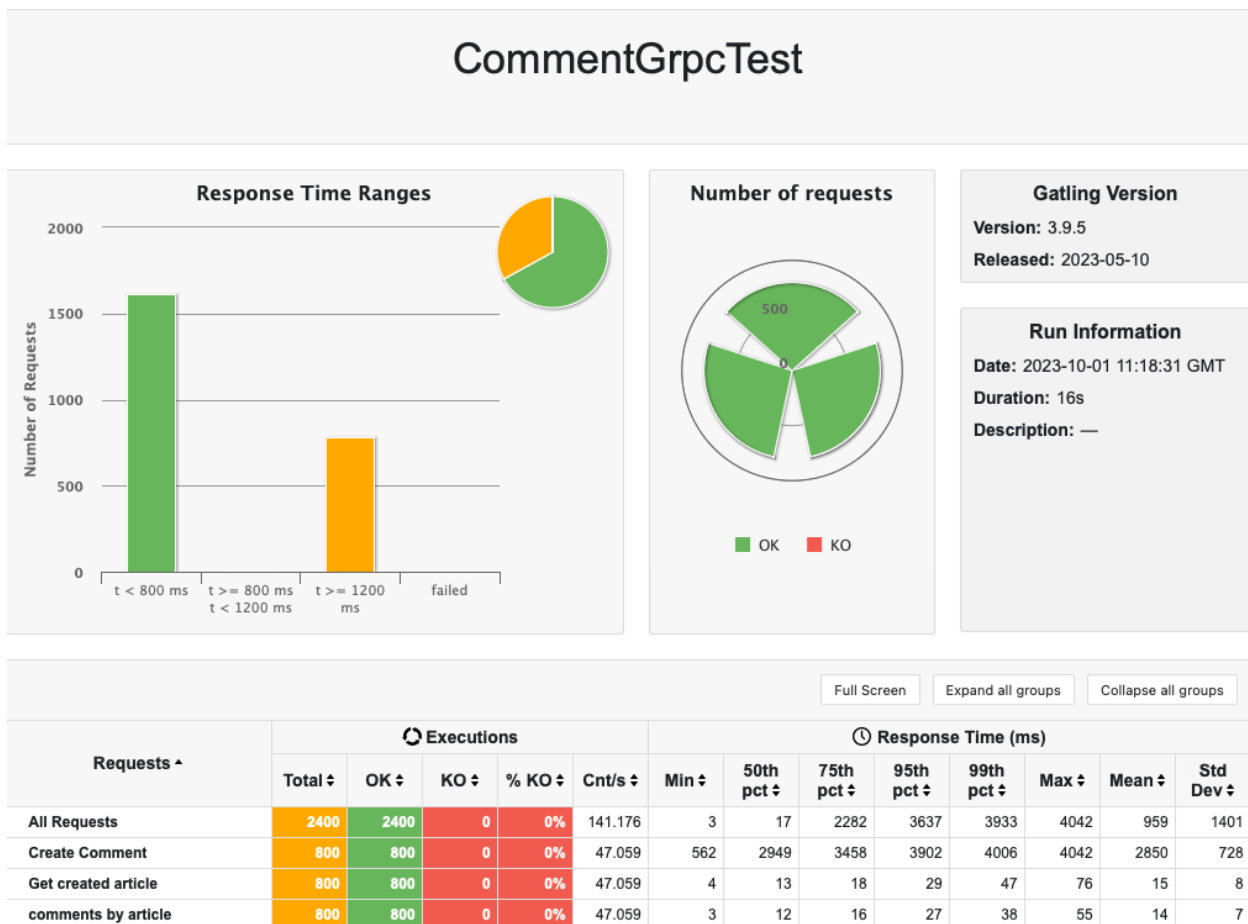


Abbildung 24 Comment Erstellung Scenario GRPC Ergebnis mit 800 Nutzern

Requests	Executions				
	Total	OK	KO	% KO	Cnt/s
All Requests	0,00%	0,00%	0	0%	11,76%

Create Comment	0,00%	0,00%	0	0%	11,77%
Get created article	0,00%	0,00%	0	0%	11,77%
comments by article	0,00%	0,00%	0	0%	11,77%

Requests	Response Time (ms)							
	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
All Requests	-40,00%	-64,58%	2,42%	-24,81%	-26,64%	-26,44%	-14,76%	-18,69%
Create Comment	11140,00%	9115,63%	7257,45%	5395,77%	3679,25%	2291,72%	8042,86%	3366,67%
Get created article	-42,86%	-59,38%	-62,50%	-57,35%	-47,19%	2291,72%	-57,14%	-55,56%
comments by article	-99,63%	-99,64%	-99,64%	-99,49%	-99,30%	-99,00%	-99,58%	-99,48%

Tabelle 14 Vergleich von GRPC (800) und REST (800) Comment Erstellung

5.4 POST Request Test REST vs GRPC

Die Abbildung 25 beschreibt die POST Anfrage REST Test Ergebnis von Nutzer Erstellungen.

<div>Full Screen</div> <div>Expand all groups</div> <div>Collapse all groups</div>													
Requests ^	Executions					Response Time (ms)							
	Total ↕	OK ↕	KO ↕	% KO ↕	Cnt/s ↕	Min ↕	50th pct ↕	75th pct ↕	95th pct ↕	99th pct ↕	Max ↕	Mean ↕	Std Dev ↕
All Requests	1375	1375	0	0%	343.75	71	554	650	896	1052	1160	578	163
CreateUserReq	1375	1375	0	0%	343.75	71	554	650	896	1052	1160	578	163

Abbildung 25 : User Erstellung POST Anfrage REST Ergebnis mit 275 Nutzern

Die Abbildung 26 beschreibt die POST Anfrage Alternative GRPC Test Ergebnis von Nutzer Erstellung.

<div>Full Screen</div> <div>Expand all groups</div> <div>Collapse all groups</div>													
Requests ^	Executions					Response Time (ms)							
	Total ↕	OK ↕	KO ↕	% KO ↕	Cnt/s ↕	Min ↕	50th pct ↕	75th pct ↕	95th pct ↕	99th pct ↕	Max ↕	Mean ↕	Std Dev ↕
All Requests	1375	1375	0	0%	343.75	171	549	663	1083	1250	1402	608	201
Create User	1375	1375	0	0%	343.75	171	549	663	1083	1250	1402	608	201

Abbildung 26 : User Erstellung POST Anfrage GRPC Ergebnis mit 275 Nutzern

Das Testergebnis in Tabelle 15 zeigt, dass die durchschnittliche Antwortzeit für POST-Anfragen bei REST im Durchschnitt um 5% schneller ist als bei GRPC und beim 75th pct um 2%. Insgesamt kann festgestellt werden, dass das Endergebnis gleich ist.

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
REST	1375	1375	0	0%	343.75	71	554	650	896	1052	1160	578	163
GRPC	1375	1375	0	0%	343.75	171	549	663	1083	1250	1402	608	201
AVG			0	0%	.	-58%	1%	-2%	-17%	-16%	-17%	-5%	-19%

Tabelle 15 User Erstellung POST Request GRPC und REST Vergleich Ergebnis mit 275 Nutzern

Von dem Load Test Ergebnis Vergleich Tabelle 16 kann man sehen, dass die Kapazität für die gleichzeitige Bearbeitung von GRPC ist 350 und von REST ist 275. Was 27% größer ist. Aber die GRPC Antwortzeit ist im Durchschnitt (Mean) 1227% langsamer.

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
All Requests	27%	27%	0%	0%	-88%	-62%	198%	248%	3649%	3169%	2894%	1227%	7309%

Tabelle 16 User Erstellung POST Request GRPC und REST Vergleich Ergebnis mit 275 Nutzern

Die Abbildung 27 beschreibt POST Anfrage Alternative GRPC Test Ergebnis von Nutzer Erstellung mit 375 Nutzern.

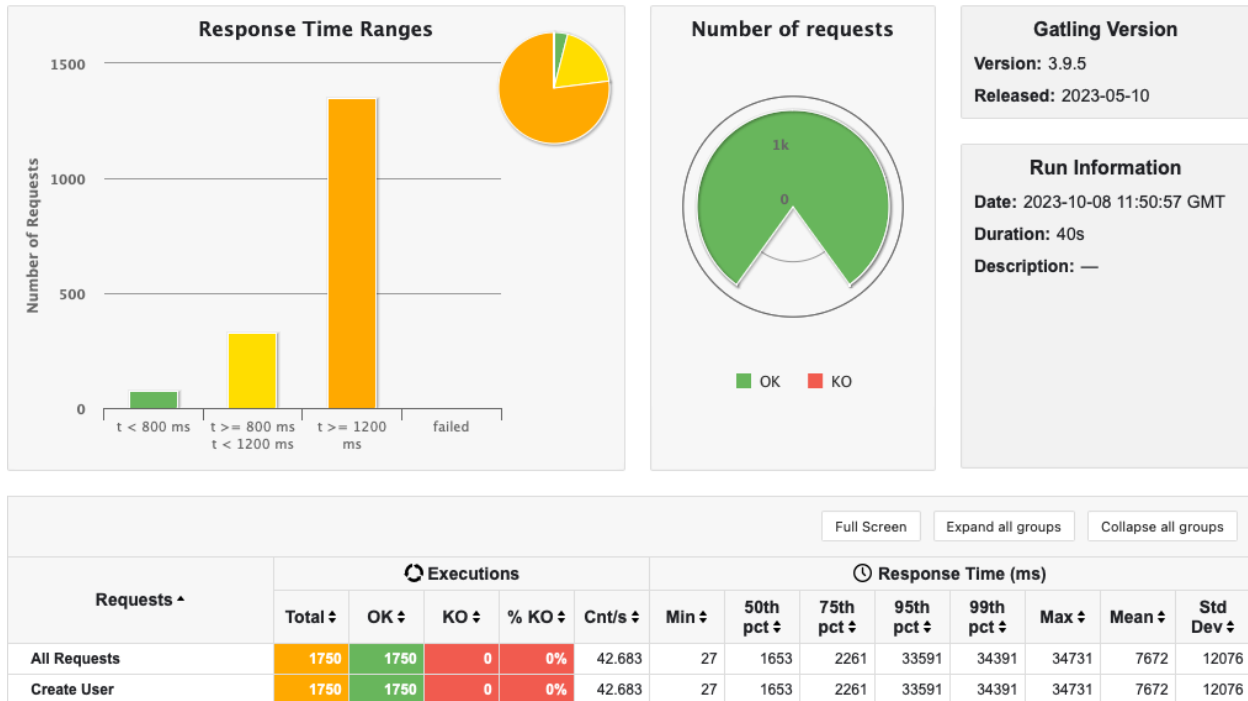
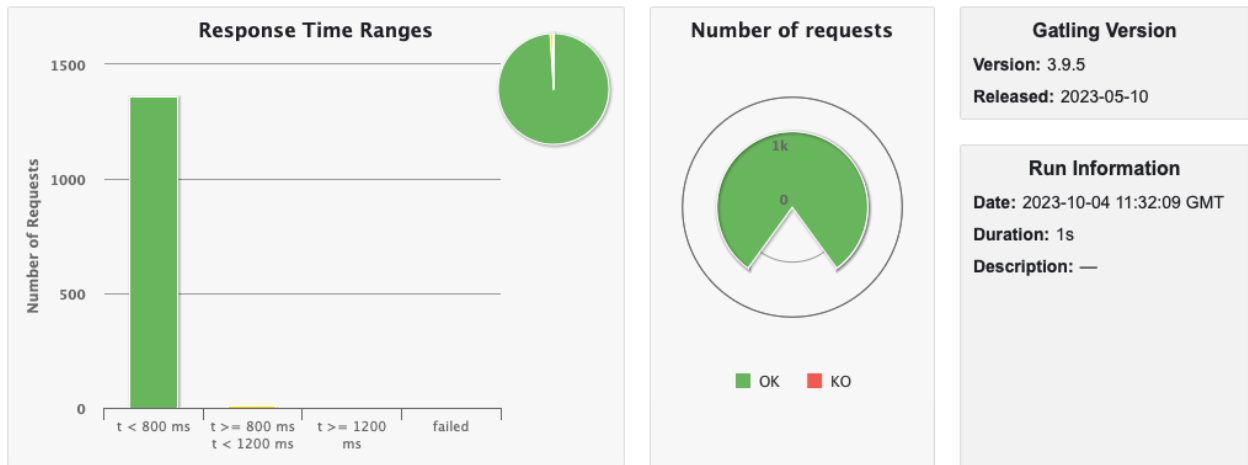


Abbildung 27 User Erstellung POST Anfrage GRPC Ergebnis mit 350 Nutzern

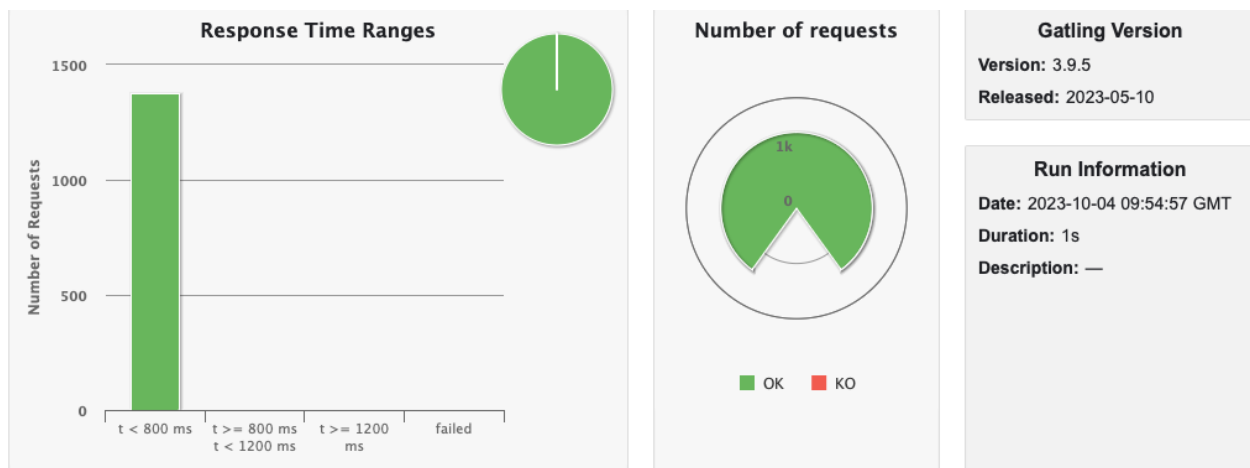
5.5 GET Request Test REST vs GRPC

Die Abbildung 28 beschreibt die GET Anfrage REST Test Ergebnis von Nutzer Erstellung.



Requests ^	Executions					Response Time (ms)							
	Total ↕	OK ↕	KO ↕	% KO ↕	Cnt/s ↕	Min ↕	50th pct ↕	75th pct ↕	95th pct ↕	99th pct ↕	Max ↕	Mean ↕	Std Dev ↕
All Requests	1375	1375	0	0%	687.5	5	143	185	550	600	1135	209	174
get created article	1375	1375	0	0%	687.5	5	143	185	550	600	1135	209	174

Abbildung 28 Artikel Erstellung GET Anfrage REST Ergebnis mit 275 Nutzern



Requests ^	Executions					Response Time (ms)							
	Total ↕	OK ↕	KO ↕	% KO ↕	Cnt/s ↕	Min ↕	50th pct ↕	75th pct ↕	95th pct ↕	99th pct ↕	Max ↕	Mean ↕	Std Dev ↕
All Requests	1375	1375	0	0%	687.5	23	130	226	636	681	729	208	184
Get article by ID and check if st...	1375	1375	0	0%	687.5	23	130	226	636	681	729	208	184

Abbildung 29 Artikel Erstellung POST Anfrage GRPC Ergebnis mit 275 Nutzern

Von Testergebnis Tabelle 17 GET Anfrage Antwortzeit Unterschied im Durchschnitt ist gleich.

Reques ts	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
REST	1375	1375	0	0%	687.5	23	130	226	636	681	729	208	184
GRPC	1375	1375	0	0%	687.5	5	143	185	550	600	1135	209	174
AVG	1375	1375	0	0%	687.5	360%	-9%	22%	16%	14%	-36%	0%	6%

Tabelle 17 User Erstellung GRPC und REST Vergleich Ergebnis mit 275 Nutzern

5.6 PUT Request Test REST vs GRPC

Die
Abbildung 30 beschreibt PUT Anfrage REST Test Ergebnis von Nutzer Erstellung.



Abbildung 30 PUT Anfrage REST Ergebnis mit 220 Nutzern

Die Abbildung 31 beschreibt PUT Anfrage GRPC Test Ergebnis von Nutzer Erstellung.

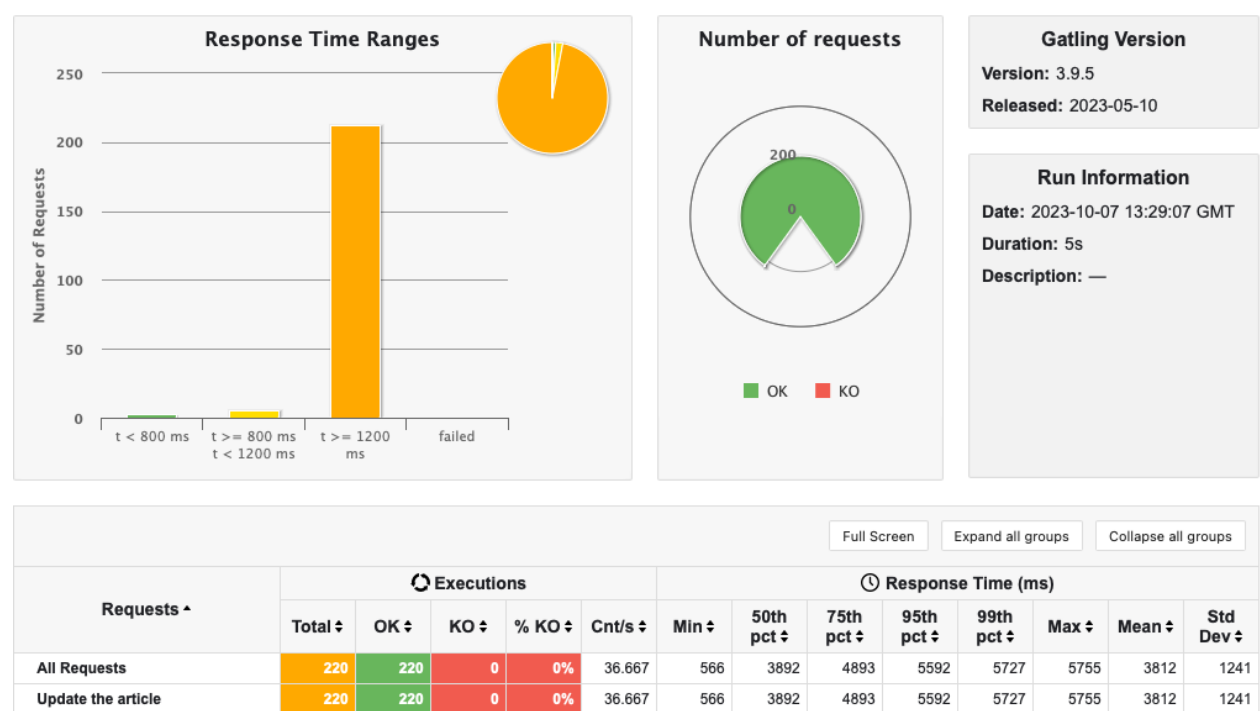


Abbildung 31 PUT Anfrage GRPC Ergebnis mit 220 Nutzern

Die Testergebnisse in Tabelle 18 zeigen, dass im Durchschnitt die Antwortzeit für PUT-Anfragen bei gRPC um 29% schneller ist. Beim 75. Perzentil ist gRPC 31% schneller.

Die Tabelle 18 beschreibt PUT Anfrage GRPC und REST Vergleich von Nutzer Erstellung.

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
GRPC	220	220	0	0%	36.667	566	3892	4893	5592	5727	5755	3812	1241
REST	220	220	0	0%	24.444	974	5353	7081	8231	8438	8506	5348	1912
AVG	0%	0%	0%	0%	50%	-42%	-27%	-31%	-32%	-32%	-32%	-29%	-35%

Tabelle 18 PUT Anfrage GRPC und REST Vergleich mit 220 Nutzern

Kapitel 6 Zusammenfassung der Ergebnisse

Die Abbildung 32 beschreibt GRPC und REST Vergleich Ergebnis in grafischer Form.

GRPC und REST Vergleich

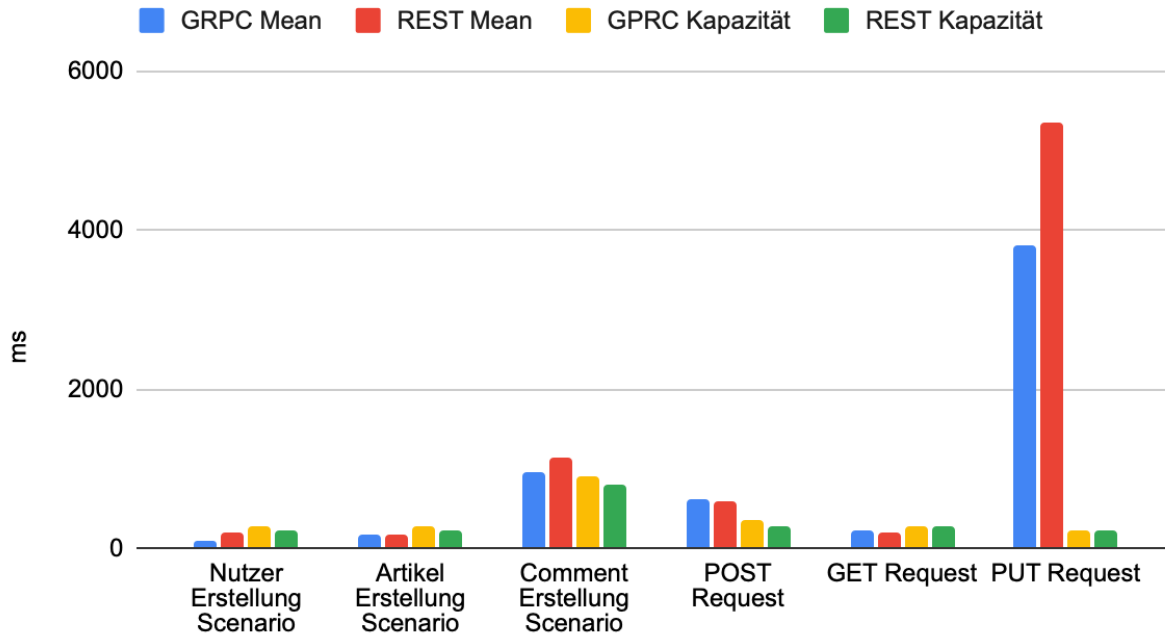


Abbildung 32 GRPC und REST Vergleich

Aus Tabelle 19 kann beobachtet werden dass GRPC im Durchschnitt eine 11% schnellere Antwortzeit als REST hat. GRPC Kapazität ist 18% größer als REST. Aus Abbildung 32 kann auch beobachtet werden dass der Unterschied nicht so groß ist.

Die Tabelle 19 beschreibt GRPC und REST Vergleich Ergebnis.

	GRPC Mean	REST Mean	GRPC Kapazität	REST Kapazität	GRPC REST Mean Vergleich	GRPC REST Kapazität Vergleich
Nutzer Erstellung Scenario	90	188	275	220	52.13%	20.00%
Artikel Erstellung Scenario	162	172	270	220	5.81%	18.52%
Comment Erstellung Scenario	959	1125	900	800	14.76%	11.11%
POST Request	608	578	350	275	-5.19%	21.43%
GET Request	209	208	275	275	-0.48%	0.00%
PUT Request	3812	5348	220	220	28.72%	0.00%

Tabelle 19 GRPC und REST Vergleich

Die Verwendung von GRPC in unserem Projekt hat nur eine geringe Steigerung der Leistungsfähigkeit bewirkt. Es ist jedoch bemerkenswert, dass diese Optimierung, obwohl sie geringfügig ist, in bestimmten Szenarien einen entscheidenden Vorteil bieten kann.

6.1 Vergleich zu anderen Forschungen

6.1.1 Forschung von Moscow Technical University of Communications and Informatics MTUCI

Die Ergebnisse dieser Arbeit bestätigen die Schlussfolgerungen und Forschung von MTUCI: *“Für Ziele, die eine hochperformante Übertragung von Massendaten erfordern, empfiehlt sich die Verwendung einer der GRPC-Übertragungsarten. Für die Implementierung von kleinen Datenübertragungssystemen oder für die Implementierung von komplexen Übertragungssystemen lohnt es sich, die voll entwickelte REST API”* [9, S.53].

6.1.2 Forschung Rzeszov University of Technology (RUT)

Die Ergebnisse dieser Arbeit stehen im Gegensatz zu RUTs Forschung [2, Kapitel 3, Experimental Result]. Wo REST mit kleinen Daten doppelt so schnell als GRPC war.

Sie haben 2 Microservices erstellt die miteinander durch REST und GRPC kommunizieren. Microservice A schickt Anfrage zum Microservice B und Microservice B antwortet mit Datei Abbildung 33.

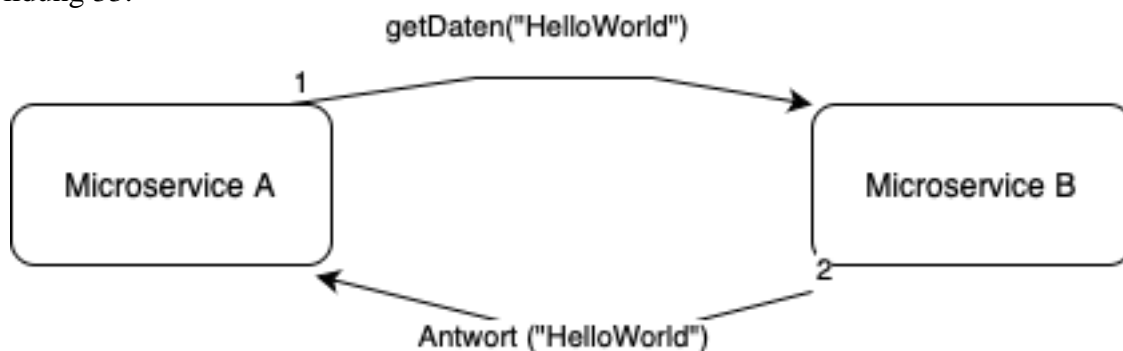


Abbildung 33 RUTs Test Scheme

Der Text “Hello World”, welcher als Funktionsparameter und als Antwort verwendet wurde, betrug die Antwortzeit für REST 4 Sekunden und für GRPC 8 Sekunden. Bei Verwendung einer Textvorlage "Lorem ipsum..." Abbildung 35 mit der Größe von 615 Zeichen zeigte REST Ergebnis von 57 Sekunden und GRPC 68 Sekunden [2, Kapitel 3, Experimental Result] .

Abbildung 34 beschreibt das GRPC und REST “Hello World” Test Ergebnis von RUT [2, Kapitel 3, Experimental Result]

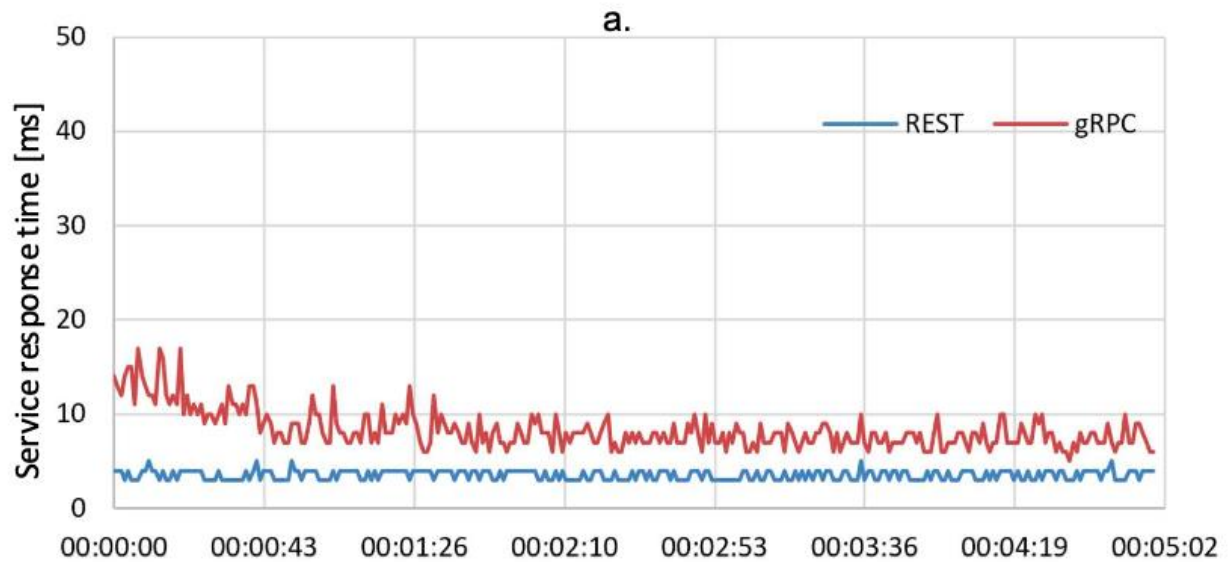


Abbildung 34 GRPC und REST Vergleich “Hello World” (source [2, Kapitel 3, Experimental Result])

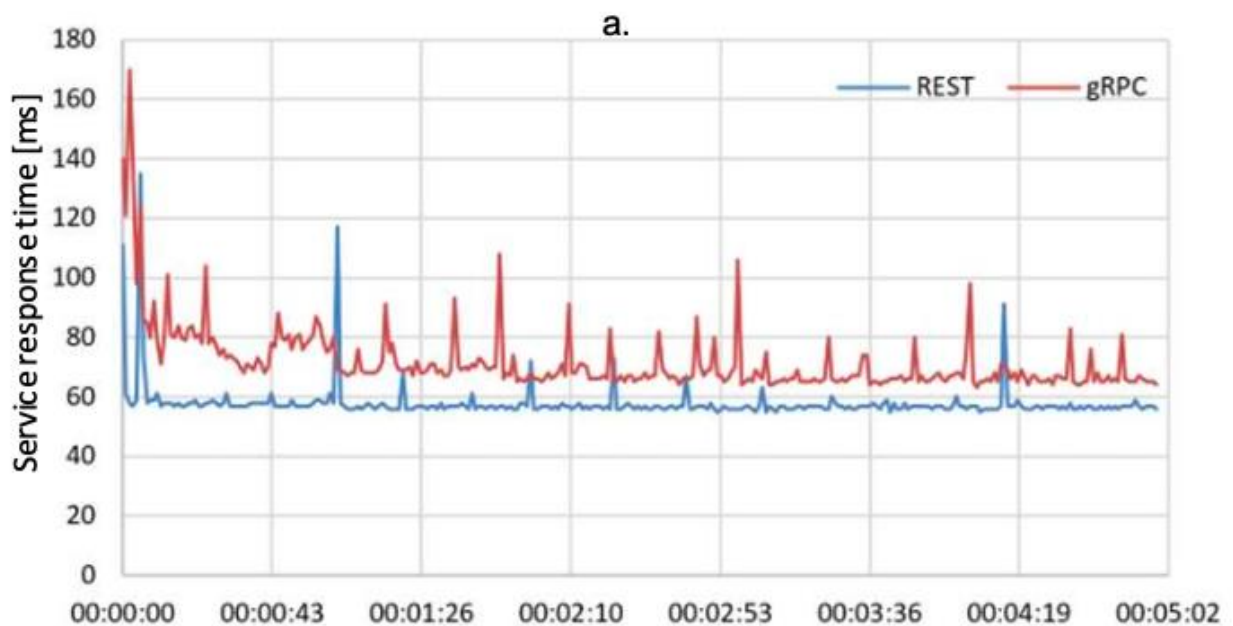


Abbildung 35 GRPC und REST Vergleich “Lorem ipsum” 615 Charakters (source [2, Kapitel 3, Experimental Result])

Um herauszufinden, warum die Ergebnisse voneinander abweichen, wurde ein identischer Test durchgeführt wie in Abbildung 36, und Abbildung 37.

Die Ergebnisse Abbildung 38 und Abbildung 39 zeigen das die Antwortzeit zwischen REST und GRPC gleich ist. Die Gründe für die unterschiedlichen Ergebnisse dieser Arbeit und von RUTs kann verschiedene Bibliothek Versionen und des Computers Technische Eigenschaften sein.

```
@GetMapping("/helloWorld/{status:.+}")
@Operation(summary = "helloWorld", description = "helloWorlds")
public String getHelloWorld(
    @PathVariable @NotEmpty final String status) throws
    InterruptedException, ExecutionException {
    return status;
}
```

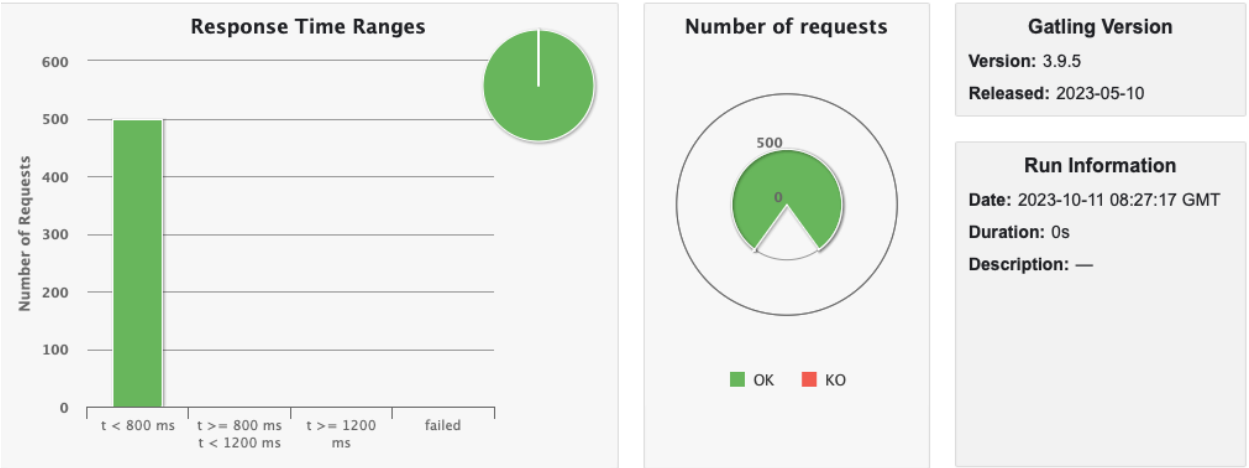
Abbildung 36 REST HelloWorld

```
@Override
public void helloWorld(StringMessageParam request,
    StreamObserver<StringMessageParam> responseObserver) {
    responseObserver.onNext(
        StringMessageParam.newBuilder().setValue(request.getValue()).build()
    );
    responseObserver.onCompleted();
}
```

Abbildung 37 GRPC HelloWorld

Abbildung 38 beschreibt das REST “Hello World” Ergebnis unserer Arbeit.

HelloWorldTest



Full Screen Expand all groups Collapse all groups

Requests ^	Executions					Response Time (ms)							
	Total ↕	OK ↕	KO ↕	% KO ↕	Cnt/s ↕	Min ↕	50th pct ↕	75th pct ↕	95th pct ↕	99th pct ↕	Max ↕	Mean ↕	Std Dev ↕
All Requests	500	500	0	0%	500	0	1	1	1	3	17	1	1
get created article	500	500	0	0%	500	0	1	1	1	3	17	1	1

Abbildung 38 REST Ergebnis “Hello World” und “Lore ipsum...”

Abbildung 39 beschreibt das GRPC “Hello World” Test Ergebnis unserer Arbeit.

HelloWorldGrpcTest

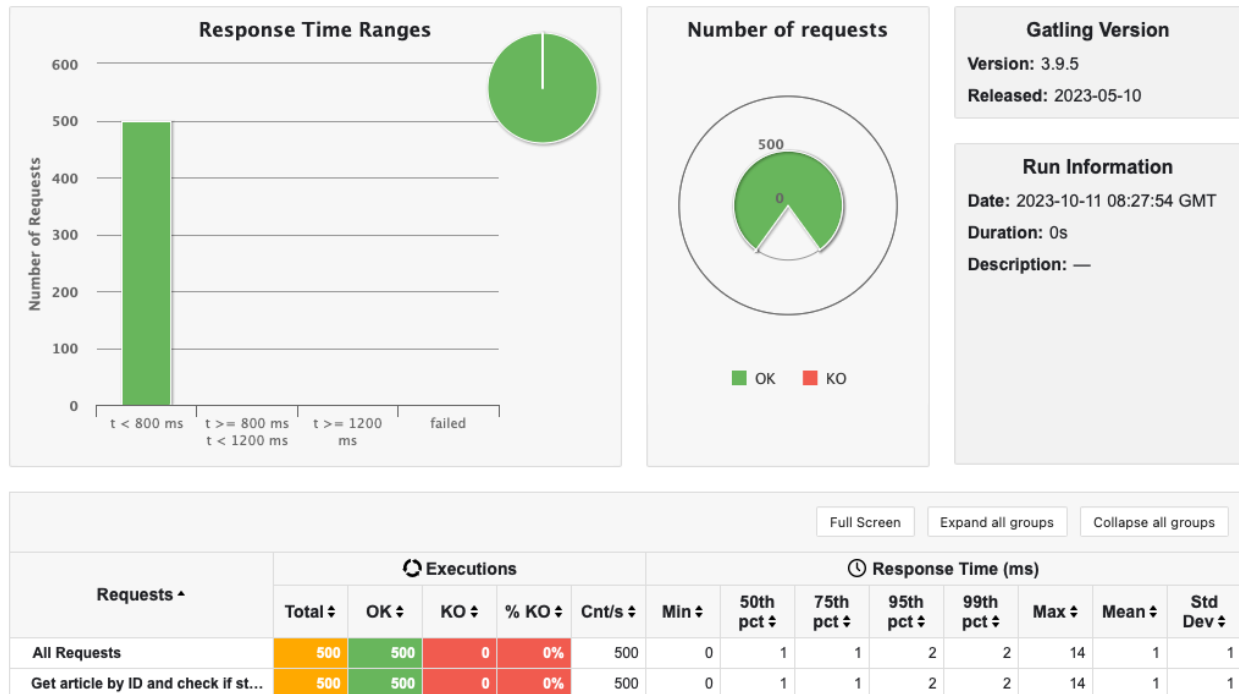


Abbildung 39 GRPC Ergebnis “Hello World” und “Lore ipsum...”

6.1.3 Forschung von Masaryk University

Ergebnisse unserer Arbeit ergänzen die Arbeit von Masaryk University. [6]

Die Forscher von Masaryk University verwendeten eine reale Fallstudie im Unternehmen Kiwi.com, in dem sie zweifache Produktivitätssteigerung festgestellt haben. *“Während in der Literatur beschrieben wird, dass GRPC etwa 7-10 mal schneller ist als REST, haben wir keine so signifikante Verbesserung erreicht. Beim Senden großer Nutzdaten ist GRPC schneller als REST, aber bei kleineren Nutzdaten ist der Unterschied nicht so groß. In einem der am häufigsten aufgerufenen*

Endpunkten des migrierten Dienstes ist die GRPC-Ausführungszeit doppelt so schnell wie die REST-Ausführungszeit (hauptsächlich aufgrund von HTTP/2 und schnellerem Marshalling/Entmarshalling der Nutzdaten). Auch wenn sie nicht siebenmal schneller ist, sind wir mit dieser Verbesserung zufrieden.

“ [6, S.69].

Obwohl in unserer Arbeit nicht die doppelte Performance Verbesserung beobachtet werden konnte, wurde eine Performance Verbesserung von 20% festgestellt. Der Grund dafür ist die kleine Datenmenge die in unserer Arbeit verwendet wurde.

Literatur

- [1] „The Rise and Fall of Nokia,“ Eaton Business School, [Online]. Available: <https://ebsedu.org/blog/the-rise-and-fall-of-nokia-lessons-learned-so-far>. [Zugriff am 14 10 2023].
- [2] Marek BOLANOWSKI, „Efficiency of REST and gRPC realizing communication tasks in microservice-based ecosystems,“ August 2022.
- [3] V. Selvaraj, „<https://www.vinsguru.com/grpc-vs-rest-performance-comparison/>,“ 31 08 2020. [Online]. [Zugriff am 14 10 2023].
- [4] J. H, „[blog.dreamfactory.com](https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis/),“ [Online]. Available: <https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis/>. [Zugriff am 14 10 2023].
- [5] R. Fernando, „[medium.com](https://medium.com/@EmperorRXF/evaluating-performance-of-rest-vs-grpc-1b8bdf0b22da#:~:text=gRPC%20is%20roughly%207%20times,of%20HTTP%2F2%20by%20gRPC..),“ 03 04 2019. [Online]. Available: <https://medium.com/@EmperorRXF/evaluating-performance-of-rest-vs-grpc-1b8bdf0b22da#:~:text=gRPC%20is%20roughly%207%20times,of%20HTTP%2F2%20by%20gRPC..> [Zugriff am 14 10 2023].
- [6] M. Štefanič, „Developing the guidelines for migration from RESTful microservices to gRPC,“ 2021.
- [7] R. Fielding, June 1999. [Online]. Available: <https://www.rfc-editor.org/rfc/rfc2616?data1=dwnsb4B&data2=abmurltv2b>. [Zugriff am 14 10 2023].
- [8] F. Gerlinghoff, „Vergleich von Introspected REST mit alternativen Ansätzen für die Entwicklung von Web-APIs hinsichtlich Performance, Evolvierbarkeit und Komplexität,“ 18 September 2020.
- [9] I. G. Buzhin, „COMPARATIVE ANALYSIS OF THE REST AND GRPC USED IN THE MONITORING SYSTEM OF COMMUNICATION NETWORK VIRTUALIZED INFRASTRUCTURE,“ 14 04 2023.
- [10] V. Nair, „Practical Domain-Driven Design in Enterprise Java Using Jakarta EE, Eclipse MicroProfile, Spring Boot, and the Axon Framework,“ Mountain View, CA, 2019.
- [11] „DDD,“ [Online]. Available: <https://www.dddcommunity.org>. [Zugriff am 14 10 2023].
- [12] „Gatling,“ [Online]. Available: <https://gatling.io/open-source/>. [Zugriff am 14 10 2023].
- [13] „RealWorld,“ [Online]. Available: <https://www.realworld.how>. [Zugriff am 14 10 2023].
- [14] "Repository," [Online]. Available: <https://codebase.show/projects/realworld>. [Accessed 14 10 2023].
- [15] [Online]. Available: <https://protobuf.dev>. [Zugriff am 14 10 2023].
- [16] „Technical Highlights,“ [Online]. Available: <https://developer.axoniq.io/axon-framework/technical-highlights>. [Zugriff am 14 10 2023].
- [17] „Axoniq.io,“ [Online]. Available: <https://developer.axoniq.io/axon-framework/overview>. [Zugriff am 14 10 2023].

