HTWK

Hochschule für Technik,
Wirtschaft und Kultur Leipzig

Technologievergleich zwischen API-Datenabfragen an einen Service mittels REST/GraphQL/gRPC im E-Commerce Kontext

Abschlussarbeit zur Erlangung des akademischen Grades
Bachelor of Science

Elias Anton

Erstprüfer: Herr Prof. Dr.-Ing. Thomas Kudraß
Zweitprüfer: Herr Dr. rer. nat. Florian Scheffler

Leipzig, den 18. Juli 2023

Kurzfassung

Diese Bachelorarbeit vergleicht die Kommunikationstechnologien REST, GraphQL und gRPC im Kontext der bestehenden E-Commerce-Architektur der Relaxdays GmbH. Ziel ist es, die geeignetste Technologie für den Datenaustausch zwischen Microservices mit eigenen Datenbanken zu ermitteln. Dabei werden verschiedene Testfälle entwickelt, um reale Unternehmensprozesse mit unterschiedlichen Datenmengen zu simulieren. Die Technologien werden hinsichtlich ihrer Übertragungsleistung und Umsetzbarkeit im Unternehmenskontext verglichen.

Letztendlich hängt die Entscheidung über die beste Technologie an einer Abwägung zwischen Flexibilität, Performanz, Funktionalität und Implementierungsaufwand gegenüber der aktuellen Implementierung.

Die Auswertung ergibt, dass gRPC bei üblichen Datenmengen ähnliche Geschwindigkeiten wie REST aufweist. Nur bei großen Datenpaketen pro Anfrage war gRPC schneller. Da REST jedoch bereits im Unternehmen implementiert ist und gRPC eine neue Implementierung erfordern würde, wurden die Vorteile von gRPC als nicht ausreichend betrachtet, um einen Wechsel zu rechtfertigen.

Als solide Technologie stellt sich REST heraus, welche eine gute Rundlaufzeit für typische Datenmengen und eine sehr gute Integration in die weitere Infrastruktur des Unternehmens aufweist. Allerdings überzeugt REST nicht mit Flexibilität.

GraphQL zeigte sich als geeignete Alternative, insbesondere wenn Flexibilität bei Datenabfragen erforderlich ist. Es bietet eine bessere Leistung als REST und gRPC, wenn entweder nur reduzierte Datensätze benötigt werden oder REST und gRPC für die selben Daten mehrere Anfragen durchführen müssen.

REST und gRPC sind wiederum performanter als GraphQL, wenn gesonderte Endpunkte zur Abfrage diese spezifischen Datensätze implementiert werden um die Anzahl der Anfragen zu reduzieren. Eine Implementierung neuer Endpunkte für jede spezifische Datenabfrage wird jedoch als unpraktisch und schlecht skalierend bewertet, wenn viele solcher Endpunkte benötigt werden würden.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und nicht anderweitig zu Prüfungszwecken vorgelegt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht.

Ort, Datum: _____ Unterschrift: _____

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

HTTPS	Hypertext Transfer Protocol - secure
CRUD	Create, Read, Update, Delete
REST	Representational State Transfer
gRPC	Google Remote Procedure Call
GraphQL	Graph Query Language
XML	Extensible Markup Language
JSON	JavaScript Object Notation
Protobuf	Protocol Buffer
LINQ	Language-Integrated Query
ORM	object-relational mapping
API	Application Programming Interface
SKU	Shop Keeping Unit

1. Einleitung

In diesem Teil wird erklärt, mit welcher Absicht die Arbeit verfasst wird.

1.1. Motivation und Fragestellung

Die Relaxdays GmbH welche diese Arbeit unterstützt, ist ein schnell wachsendes, flexibles E-Commerce Unternehmen. Es hat sich entschieden, alle benötigte Software selbst zu implementieren anstatt fertige E-Commerce Lösungen einzukaufen. Damit die produzierten Programme auch flexibel und wartbar bleiben, ist eine Microservice-Architektur in Zukunft wünschenswert, die bisher nur in Ansätzen existiert. Sie bietet viele Vorteile. Zum Beispiel, dass einzelne Komponenten mit sehr wenig Aufwand ausgetauscht und skaliert werden können oder dass das Gesamtsystem robuster wird. Der erste Schritt in Richtung Microservices ist eine Auftrennung der bislang monolithischen Datenbankstruktur in einzelne, spezifischere Datenbanken. Vor allem vor dem Hintergrund der Wachstums- und Effizienzsteigerungsambitionen des Unternehmens ist eine hohe Relevanz festzustellen, da bei einem Wachstum der zu verarbeitenden Datenmengen eine gut strukturierte Datenhaltung und effiziente Datenflüsse essenziell sind.

Viele voneinander unabhängige Programmteile des Partnerunternehmens greifen momentan auf dieselbe monolithische Datenbank zu. Innerhalb des Datenbankschemas gibt es bereits natürliche Trennlinien zwischen Datenclustern[1] die untereinander stark, aber zu anderen Clustern nur schwach oder gar nicht verknüpft sind. Es existieren bereits festgelegte Repositories, welche im Code adressiert werden können und welche als Eintrittspunkte in die Datenbankstruktur dienen (z.B.: Artikel oder Bestellungen).

Inhalt dieser Arbeit soll es nun sein, im Hinblick auf eine zukünftige Entwicklung hin zu einer Microservice-Architektur, die Datenbankstruktur sinnvoll aufzutrennen und die beste Technologie für die Kommunikation mit der aufgetrennten Datenbank zu finden. In Abschnitt 2.2 wird eine Web-Service-Architektur dargestellt, die mit mehreren Datenbanken umgehen kann. Dabei existieren mehrere Web-Services mit je einer Datenbank-Instanz.

Um diese Services zuverlässig und schnell mit den anderen Services zu verbinden, von denen sie auch Daten benötigen, muss eine passende Kommunikationstechnologie benutzt werden. Betrachtet werden sollen hier die bereits im Partnerunternehmen genutzte REST-Technologie,

[1] Begriff "Datencluster" im Sinne von Daten, die untereinander enger zueinander gehören als andere Bereiche der Datenbank

aber auch die beiden Alternativen GraphQL und gRPC. Auf genau diesem Vergleich liegt der Fokus dieser Arbeit. Beim Vergleichen der Technologien soll auf Leistung, Funktionalitäten und auch Umsetzbarkeit für verschiedene Anwendungsfälle in dem Partnerunternehmen Bezug genommen werden.

Von dieser Motivation abgeleitet stellt sich die folgende Forschungsfrage: *Welche der drei Technologien REST, GraphQL und gRPC eignet sich in welchen Anwendungsfällen am besten für eine Kommunikation zwischen verschiedenen Services mit eigenen Datenbank-Instanzen in einer Web-basierten Microservice-Architektur in Bezug auf Leistung und Umsetzbarkeit?*

Im Optimalfall kann diese Bachelorarbeit als Entscheidungsgrundlage für tiefer greifende Änderungen in der Softwarearchitektur des Partnerunternehmens oder auch als Grundlage für weitere Arbeiten in dieser Richtung dienen. Zusätzlich werden einige Ergebnisse dieser Arbeit auch über den Unternehmenskontext hinaus einen allgemeinen Mehrwert bieten, da grundlegende Vor- und Nachteile der Technologien gRPC, REST und GraphQL erforscht werden.

1.2. Struktur

Zuerst wird in dieser Arbeit der aktuelle Forschungsstand von Datenhaltung in Microservice-Architekturen betrachtet und verschiedene Architekturkonzepte dargestellt (Abschnitt 2.2). Daraufhin werden die drei Kommunikationstechnologien REST, gRPC und GraphQL mit ihren Funktionalitäten, Vorteilen und Nachteilen ausführlich beschrieben (Abschnitt 2.3). Außerdem werden die anderen genutzten Technologien eingeführt.

In dem Kapitel 3 wird dann mit der Beantwortung der Forschungsfrage begonnen. Dazu wird zuerst der Versuchsaufbau beschrieben (Abschnitt 3.1) und dann werden die Metriken zum Vergleichen eingeführt (Abschnitt 3.2). Hier wird auch auf die verschiedenen Testfälle eingegangen, die für diese Arbeit konzipiert wurden. Sie reichen von einfachen bis hin zu komplexeren Datenabfragen.

Danach werden die Beobachtungen über die Ergebnisse festgehalten, anschließend ausgewertet und diskutiert (Kapitel 4). In diesem Zuge wird dann die Forschungsfrage differenziert beantwortet.

Abschließend werden die Ergebnisse noch ein mal übersichtlich zusammen gefasst, ein Fazit gezogen und ein Ausblick über weitere Forschungsthemen gegeben (Kapitel 5).

2. Grundlagen

In diesem Kapitel wird auf die Grundlagen eingegangen, die für dieses Thema relevant sind. Zuerst wird das Architekturprinzip der Microservices beschrieben und im Weiteren das Problem der Datenhaltung in einer Microservice-Struktur dargestellt.

2.1. Microservices

Der Begriff Microservices beschreibt nicht etwa ein festes Framework, sondern ein Architekturprinzip in der modernen Softwareentwicklung. Das Grundprinzip von Microservices ist eine Aufteilung des Programmcodes in einzelne, logisch getrennte und eigenständig lauffähige Programmteile (in der Web-Entwicklung auch Services genannt). Die Services sind untereinander über Kommunikationstechnologien verbunden. Dazu stellen die Services eine Schnittstelle mit Datentyps-Definitionen für die anderen Services zur Verfügung.

Durch die Nutzung dieser Architekturgrundlage ergeben sich einige Vorteile. Zum Einen können die Programmteile einzeln gewartet oder aktualisiert werden, ohne das ganze Programm neu starten zu müssen. Das wiederum bedeutet mehr Flexibilität für kleinere Aktualisierungen und auch eine erhöhte Ausfallsicherheit. Sollte ein Fehler einen Programmteil zum Absturz bringen, ist so nur der eine Programmteil betroffen und andere Prozesse können ungestört weiter laufen.

Auch in der Handhabbarkeit ergeben sich Vorteile gegenüber einer monolithischen Architektur. Zum einen können neue Mitarbeiter den bestehenden Code viel einfacher verstehen, wenn dieser sinnvoll in Services strukturiert ist. Ein weiterer Vorteil ist, dass verschiedene Entwicklerteams unabhängig voneinander in unterschiedlichen Programmiersprachen entwickeln können, da am Ende ja über einheitliche Schnittstellen zwischen den Services kommuniziert wird[1].

Auf der anderen Seite ergeben sich allerdings auch Nachteile durch diese Strukturierung. So erzeugt die verteilte Struktur auch eine erhöhte Komplexität in der Verwaltung. Es ist deutlich aufwändiger eine funktionierende Kommunikationsstruktur zwischen den einzelnen Komponenten aufzubauen. Dazu kommen Herausforderungen in der Netzwerkverwaltung wie Lastverteilung oder Netzwerklatenzen. Außerdem ergeben sich Abhängigkeiten zwischen den einzelnen Komponenten, welche unter Anderem das Testen schwieriger machen.

Es ist noch zu erwähnen, dass echte Microservices eigentlich wirklich nur sehr kleine, elementare Aufgaben erfüllen sollen[2]. In der Praxis gibt es allerdings häufig mehrere Services mittlerer Größe, da eine bestehende, monolithische Codebasis häufig in mehreren Iterationen zu kleineren Services umgebaut wird[3].

2.2. Datenhaltung in Microservice-Architekturen

Die Datenhaltung in Microservice-Architekturen stellt ein vielschichtiges Problem dar, allerdings gibt es einige Lösungsansätze in der Literatur. Ein paar Ansätze werden hier dargestellt, nachdem auf den aktuellen Zustand bei der Relaxdays GmbH eingegangen wurde.

2.2.1. Aktueller Zustand in der Relaxdays GmbH

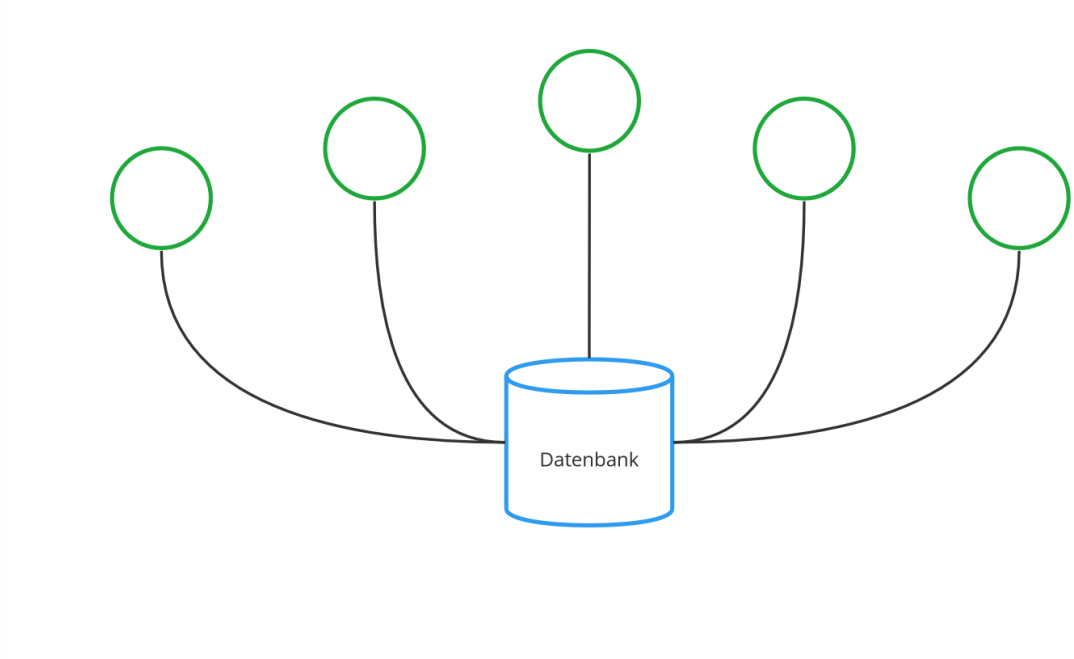

Abb. 2.1.: Eine Datenbank für alle Services

In Abbildung 2.1 kann man sehen, dass alle Services momentan auf eine zentrale Datenbank zugreifen. Das ist ein intuitiver und einfacher Aufbau, allerdings hat er auch einige Nachteile.

Ist die Datenbank auf diese Art aufgebaut, stellt sie alleine eine sehr große Abhängigkeit dar (Single-Point-Of-Failure). Sollte sie ausfallen, können große Teile des Programms nicht funktionieren. Zusätzlich stellt so eine Datenbank auch einen Kommunikationsengpass dar, denn alle Services versuchen parallel verschiedene Daten zu lesen und zu verändern. Das ist

mit Hinblick auf wachsende Datenmengen nicht gut skalierbar. Auch nicht zu vergessen ist der Aspekt der IT-Sicherheit. Liegen alle Daten in einer Datenbank, kann ein etwaiger Angreifer mit nur einem Datenbankzugriff alle Daten erhalten. Um die Problematik der Ausfallsicherheit zu Lösen, wäre auch eine Replikation der Datenbank ausreichend. Allerdings bringt ein Aufteilen der Datenbank noch weitere wichtige Vorteile mit sich.

Ziel ist, die Datenbank in mehrere kleinere Datenbanken aufzuteilen. Sie sollten so aufgeteilt werden, dass sich möglichst alle Tabellen, die semantisch eng zusammen gehören, in der selben Datenbank befinden. Dies würde alle oben genannten Nachteile vermeiden. Fällt eine Datenbank aus, wäre nur der konkrete Teil des Programms betroffen, der darauf zugreift. Datenbankzugriffe, die unabhängige Datensätze betreffen, können parallel laufen und ein potentieller Angreifer könnte nur einen Teil der Daten auf einmal einsehen. Dazu kommt noch, dass das Datenbankschema dann sehr viel flexibler angepasst werden kann, ohne dass alle Programmteile eine Aktualisierung benötigen. Schema-Änderungen betreffen dann nur noch die Services, die diese Daten tatsächlich benötigen.

Leider ist es nicht trivial, eine Datenbankstruktur aufzuspalten und den einzelnen Services zuzuordnen denn es gibt immer Anwendungsfälle, bei denen ein Service Daten aus dem Hoheitsgebiet eines andern Services benötigt[4]. Beispielsweise würde ein Service, der Bestellungen behandelt, auch Daten zu den bestellten Artikeln benötigen, die aber wiederum dem Artikel-Service zugeordnet wären. Trotzdem soll aber die Unabhängigkeit der Services untereinander möglichst beibehalten werden.

Zu diesem Problem gibt es bereits verschiedene Lösungsansätze[5], von denen im Folgenden eine Auswahl dargestellt wird.

2.2.2. Lösungsansätze

In den Quellen finden sich verschiedene Ansätze, um eine verteilte Datenbankstruktur in einer Microservice-Architektur umzusetzen. Im folgenden werden mehrere Ansätze grundsätzlich vorgestellt.

Klonen der Datenbanken

Eine auf den ersten Blick einfache Lösung wäre die Datenbank für jeden Microservice zu klonen (vgl. Abbildung 2.2). Damit wäre eine gewisse Unabhängigkeit und Ausfallsicherheit gegeben. Diese Lösung ist allerdings ziemlich ineffizient, denn alle Services hätten in ihren Datenbanken dann sehr viele Daten die sie nie benötigen. Dazu kommt noch der immense

technische Mehraufwand, um die geklonten Datenbanken synchron zu halten und die schlechte Skalierbarkeit, da jeder neue Service eine komplett neue Datenbank benötigen würde.

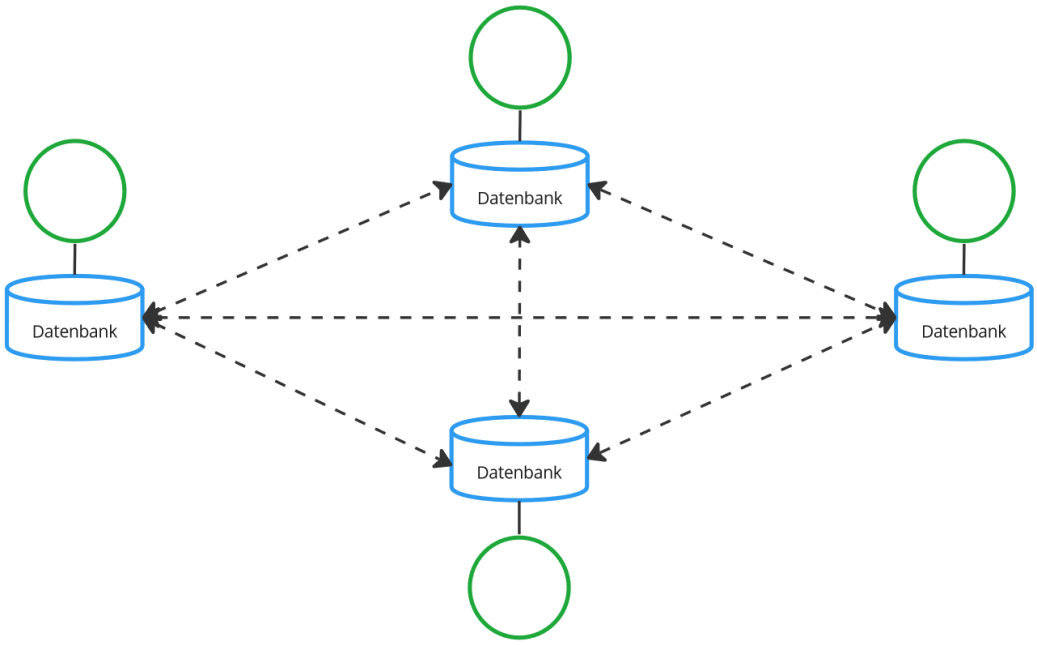

Abb. 2.2.: Geklonte Datenbanken pro Service

Ein zentraler Datenverwaltungsknoten

Ein anderer Ansatz wäre, einen zentralen Routing-Service zu implementieren, der mehrere Datenbanken abstrahiert (vgl. Abbildung 2.3). Die Datenbanken wären dann alle voneinander getrennt, man würde davon aber beim Benutzen der Datenbanken nichts merken. Alle Datenanfragen würden immer an den selben Service gehen und der Service würde die Queries an die entsprechenden Datenbanken weiterleiten um dann alle benötigten Daten zurück zu geben. GraphQL bietet Möglichkeiten, so einen Aufbau zu realisieren[6]. Das klingt relativ einfach, allerdings würde eine solche Struktur einen Flaschenhals darstellen denn alle Datenflüsse müssten durch diesen einen Datenservice hindurch. Zudem stellt eine solche Struktur auch ein großes Risiko dar, denn sollte dieser eine Knoten ausfallen, könnten alle anderen Services nicht mehr richtig funktionieren. Dies widerspricht dem Ziel der Datenbankaufspaltung eine höhere Ausfallsicherheit zu ermöglichen.

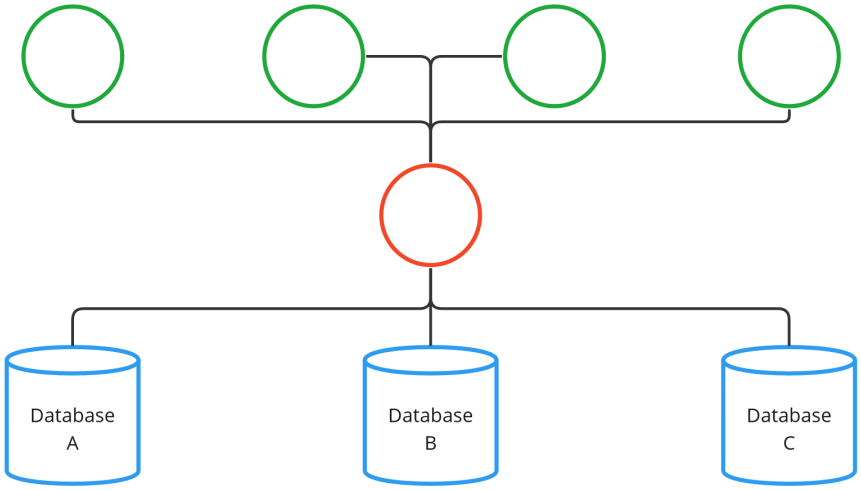

Abb. 2.3.: Ein zentraler Verwaltungsknoten

Datenbank pro Service

Eine weitere Lösung wäre, jedem Microservice eine dedizierte Datenbank zuzuweisen, die nur die benötigten Tabellen aus der aufzulösenden Datenbank enthält, die der Service am meisten benötigt. So wird eine doppelte Speicherung von Daten in verschiedenen Datenbanken vermieden. Sollten andere Services Daten benötigen, die nicht in ihren eigenen Datenbanken existieren, müssen sie den Service anfragen, der diese Daten verwaltet (vgl. Abbildung 2.4).

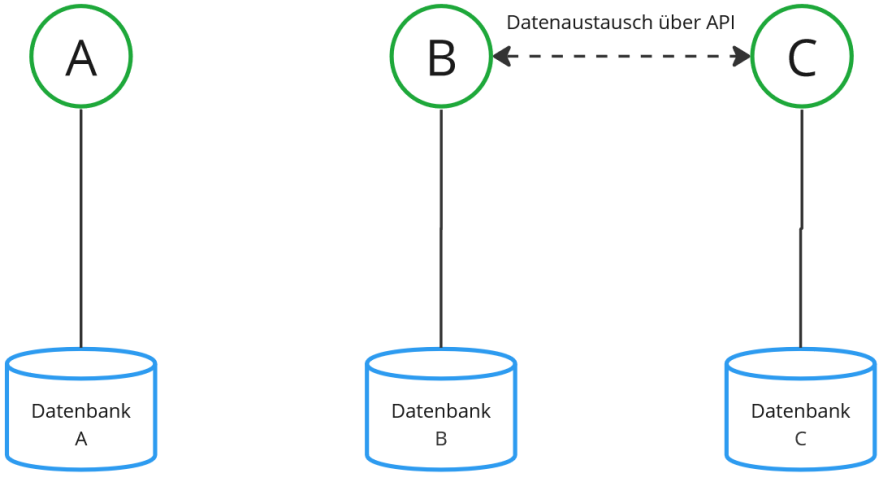

Abb. 2.4.: Individuelle, logisch getrennte Datenbanken

Demzufolge benötigen die Services zusätzlich zu ihren Hauptfunktionalitäten noch einen Teil, der die Datenverwaltung für die anderen Services übernimmt. Um die Datenintegrität innerhalb eines Services zu gewährleisten kann festgelegt werden, dass andere Services nur lesenden Zugriff auf die Daten bekommen. Schreiben kann dann nur der Service, in dessen Hoheitsgebiet sich der Datensatz befindet. Diese zusätzliche Datenverwaltung bedeutet einen technischen Mehraufwand in jedem Service um relevante Daten für andere Services zur Verfügung zu stellen. Außerdem ist es für den Service, der die Daten von anderen Services benötigt schwieriger, die Daten aus verschiedenen Quellen zu kompletten Modellen zusammenzuführen.

Für den konkreten Anwendungsfall im Partnerunternehmen gibt es aber nicht viele Anwendungsfälle bei denen Daten von verschiedenen Datenbanken benötigt werden. Auf Grund dessen überwiegen die Vorteile dieses Lösungsansatzes um die Datenbank aufzuteilen. Im Weiteren wird dieses Modell benutzt werden, um verschiedene Technologien für diesen Datenaustausch zwischen den Services zu vergleichen. Hier bieten sich vor allem drei verschiedene Kommunikationstechnologien an, auf welche im Folgenden genauer eingegangen wird.

```
{
    "Person": {
        "id": 1,
        "name": "John Smith",
        "age" : 25
    }
}
```

Listing 1: JSON Beispiel

2.3. Technologien

In diesem Abschnitt werden die in dem Testaufbau verwendeten Technologien näher beschrieben.

2.3.1. Kommunikationstechnologien

Für die Kommunikation zwischen den verschiedenen Microservices können verschiedene Technologien eingesetzt werden. Das weitverbreitetste Kommunikationsprotokoll bei Web-basierten Anwendungen ist das "Hyper Text Transfer Protocol - secure" (HTTPS). HTTPS definiert die grundlegende Request-Methoden `Create`, `Read`, `Update` und `Delete` (CRUD) und ist für die Verschlüsselung des Übertragungsweges verantwortlich. Im Folgenden werden drei Technologien vorgestellt, die alle für diesen Anwendungsfall in Frage kommen aber unterschiedlich funktionieren und jeweils ihre Vor- und Nachteile haben.

REST

Die "Representational State Transfer"-Technologie (REST) stellt eigentlich ein Architekturparadigma für Webservices dar. Wenn im Weiteren aber von REST geschrieben wird, ist eine nach REST konzipierte Schnittstelle gemeint, welche auf den CRUD-Operationen von HTTPS basiert. In einer auf REST basierenden Schnittstelle werden für diese Operationen die Schlüsselwörter `Put`, `Get`, `Post` und `Delete` genutzt. Daten werden im REST-Kontext auch "Resourcen" genannt und ihnen werden eigene Adressen zugeordnet, auf die dann mit den CRUD-Operationen zugegriffen werden kann.

Das Datenformat, welches in REST-Schnittstellen meistens zur Übertragung zum Einsatz kommt, ist das "JavaScript Object Notation"-Format (JSON). Ein beispielhafter Aufbau einer .json-Datei ist in Listing 1 zu finden. JSON ist sehr weit verbreitet und wird für viele Anwendungsfälle genutzt. Ein großer Vorteil von JSON ist die einfache Handhabbarkeit da es als menschenlesbare Zeichenkette übertragen wird. Das zieht allerdings nach sich, dass sich die

```
syntax = "proto3";

//innerhalb einer Message sind die einzelnen Felder nummeriert
message Person {
    int32 id = 1;
    optional string name = 2;
    optional int32 age = 3;

    //dies ist die Darstellung einer Array-Definition
    repeated Friend friends = 4;
}

message Friend {
    Person person = 1;
    bool isBestFriend = 2;
}
```

Listing 2: .proto Beispiel

Serialisierungs- und Deserialisierungsgeschwindigkeit verringert. Durch die weite Verbreitung hat es hier aber viel Optimierungsarbeit gegeben, weswegen dieser Punkt nicht sonderlich stark ins Gewicht fällt.

REST unterstützt auch andere Dateiformate wie das “Extensible Markup Language” (XML) - Format, welche in bestimmten Anwendungen von Vorteil sein können.

gRPC

“Google Remote Procedure Call” (gRPC) basiert im Gegensatz zu REST nicht strikt auf den CRUD-Operationen sondern versucht eine möglichst einfache Möglichkeit zu bieten, externe Funktionen, auch über verschiedene Programmiersprachen hinweg aufzurufen. Trotzdem werden Clients und Server mit Schnittstellen benötigt, um die Technologie zu nutzen[7].

Das besondere Alleinstellungsmerkmal von gRPC gegenüber anderen Technologien ist das Datenformat für die Datenübertragung. Das Format in welchem die zu übertragenden Objekte (sogenannte “Messages”) definiert werden nennt sich “Protocol Buffer”(Protobuf) und wird in .proto-Dateien gespeichert (vgl. Listing 2). Eine Protobuf-Message ist in ihrer Definition auch menschenlesbar und hat Ähnlichkeiten zu XML oder JSON, wird aber auf dem Übertragungsweg in ein Byte-Array übersetzt. Dadurch ist die Größe der Datenpakete gegnüber JSON deutlich reduziert. Diese Optimierung sorgt für eine effizientere und damit schnellere Datenübertragung, Serialisierung und Deserialisierung verglichen mit JSON[8]. Dieser Effizi-

enzvorteil zeigt sich besonders bei großen Datenmengen.

Einige große Unternehmen setzen bereits auf diese Technologie. Zum einen natürlich Google, durch welches die Technologie entwickelt wird, aber auch beispielsweise Netflix oder Square[7].

GraphQL

Die dritte Technologie die im Rahmen dieser Arbeit verglichen werden soll ist "Graph Query Language" (GraphQL). Die ursprünglich von Facebook entwickelte Technologie ist besonders gut für spezifische Datenanfragen geeignet, da sie immer nur genau die Daten zurück liefert, die auch vom Client benötigt werden. Damit kann vor allem bei komplexeren Datenabfragen ein beträchtlicher Performancegewinn erzielt werden, da keine unnötigen Daten vom Server geliefert werden und immer nur eine einzige Abfrage nötig ist. Man spricht auch davon, dass Over- und Underserving[1] von Daten vermieden wird. Overserving ist ein typisches Problem von Technologien wie REST und gRPC mit statischen Endpunkten. GraphQl wird unter anderem bereits von Facebook, GitHub, Pinterest oder Shopify verwendet[9].

Um diese Funktionalität zu bieten, muss in der Anfrage genau definiert werden, welche Objekte mit welchen Attributen erwartet werden. Auf der Server-Seite wird dann durch sogenannte Resolver die Antwort mit genau diesen Spezifikationen zusammen gebaut und zurück gesendet. Durch diese Umwandlung der Anfrage in eine Graph-Struktur und die anschließende Auflösung der Attribute durch die Resolver entsteht hier ein nicht zu vernachlässigbarer, Server-seitiger Mehraufwand. Durch diese spezielle Auflösung der einzelnen Attribute, kann GraphQL nicht automatisch von Optimierungen der genutzten Datenabfragesprache "Language-Integrated Query" (LINQ) in Zusammenarbeit mit der "object-relational mapping"[2] (ORM)-Schicht profitieren. GraphQL benötigt dazu zusätzlich einen sogenannten `Data-Loader`, welcher ähnliche, kleinere Datenbankabfragen einer größeren GraphQL-Anfrage in einer einzigen größeren Datenbankanfrage bündelt, damit die Datenbanklast verringert und so die GraphQL-Anfrage insgesamt teils deutlich beschleunigen kann.

Für die Datenübertragung vom Server zum Client wird das JSON-Format verwendet. Die Anfragen erfolgen aber über Queries, die im .graphql-Dateiformat definiert werden. Die Queries die vom Client gesendet werden, orientieren sich im Aufbau auch an JSON (vgl. Listing 3). Queries stellen dabei immer nur Lese-Zugriffe dar. Schreibzugriffe sind über sogenannte Mutations möglich.

[1] Zu viele oder zu wenige Daten für den Anwendungsfall liefern
[2] deutsch: Objektrelationale Abbildung

```
type Person {
    id: Int!
    name: String
    age: Int
}

type Query{
    getPersonById(id: Int!) : Person
}

#Akzeptierte Query für die obige Definition
#Gibt nur die angegebenen Attribute zurück
query {
    getPersonById(id: 1){
        id
        name
    }
}
```

Listing 3: GraphQL Beispiel: Objekt Person, eine Query-Definition und eine beispielhafte Anfrage

2.3.2. Framework: ASP.NET Core

Als Framework für den Testaufbau wird ASP.NET Core von Microsoft verwendet. Microsoft selbst beschreibt ASP.NET Core als "plattformübergreifendes, leistungsstarkes Open-Source-Framework zum Erstellen von modernen, cloudfähigen und mit dem Internet verbundenen Apps"[10]. Es bietet eine Vielzahl von leistungsstarken Funktionen und Werkzeugen. Ein zentraler Vorteil von ASP.NET Core liegt in seiner Unterstützung für Microservices-Architekturen, was eine flexible Entwicklung und Skalierung einzelner Komponenten ermöglicht. Zudem zeichnet sich das Framework durch seine hohe Leistungsfähigkeit aus, die durch optimierte HTTP-Verarbeitung und skalierbare Anfragen- und Antwortmechanismen erreicht wird. Sicherheit spielt ebenfalls eine wichtige Rolle, da ASP.NET Core eingebaute Schutzmechanismen gegen gängige Angriffe bietet. Darüber hinaus unterstützt das Framework die sichere Kommunikation über HTTPS und die nahtlose Integration mit Cloud-Diensten. Das Framework wird bereits überall im Partnerunternehmen genutzt. Aus diesem Grund kommt es auch im Rahmen dieser Arbeit zum Einsatz. ASP.NET Core ist in der Programmiersprache C# [3] geschrieben und die Programmierung im Rahmen dieser Bachelorarbeit basiert aus diesem Grund auch auf C#.

2.3.3. Objektrelationale Abbildung (ORM): Entity Framework Core

Zum Verknüpfen des Datenmodells in der Datenbank mit den im Code verwendeten Datenobjekten wurde Enity Framework Core (EF Core) genutzt. Der objektrelationale Mapper vereinfacht den Umgang mit Objekten und der Datenbank deutlich. Der Datenzugriff erfolgt über ein Kontext-Objekt (`DBContext`), welches eine Sitzung mit der Datenbank verwaltet. Datenobjekte lassen sich so im Code aus der Datenbank lesen, verändern und abspeichern während EF Core im Hintergrund alle Daten in einzelnen Datenbanktabellen verwaltet. So fällt ein großer Teil des Datenzugriffscodes weg. Außerdem bietet EF Core mit sogenannten Migrations eine einfache Möglichkeit das Datenmodell weiter zu entwickeln[11].

2.3.4. Virtualisierung: Docker

Docker ist eine Open-Source-Container-Technologie, die es ermöglicht Software in virtuellen Containern auszuführen. Diese Container basieren in der Regel auf Linux und sind äußerst leichtgewichtig. Sie bieten eine hohe Kompatibilität, da Entwickler ihre Anwendungen und Abhängigkeiten in einer isolierten und konsistenten Umgebung kapseln können. Dadurch werden Probleme vermieden, die durch Unterschiede in Betriebssystemen oder installierten Softwareversionen in verschiedenen Entwicklungsumgebungen auftreten können[12].

[3] gesprochen: (englisch) C-Sharp

Zudem ist es mit Docker relativ einfach, diese Container mithilfe von Anwendungen wie Kubernetes in großen Mengen zu verwalten und zu überwachen, um beispielsweise eine Microservice-Architektur zu ermöglichen. Diese zentrale Verwaltung erleichtert auch das effiziente Skalieren, indem bei hoher Systemlast zusätzliche Container gestartet werden können.

Die Entscheidung, die Datenbank in einem Docker-Container zu halten, bringt dem Versuchsaufbau mehrere Vorteile. Erstens ermöglicht sie eine Entkopplung der Datenbank von der restlichen Infrastruktur. Dadurch kann die Datenbank unabhängig von anderen Komponenten gestartet und aktualisiert werden, ohne Auswirkungen auf den übrigen Versuchsaufbau zu haben. Zweitens ermöglicht Docker eine konsistente und reproduzierbare Bereitstellung der Datenbank in beliebigen Umgebungen. Dadurch läuft die Datenbank unter denselben Bedingungen, unabhängig von der zugrunde liegenden Infrastruktur. Docker bietet somit eine effiziente Möglichkeit, die Datenbank in einer isolierten und kontrollierten Umgebung zu betreiben, was die Zuverlässigkeit und Wiederverwendbarkeit des Versuchsaufbaus verbessert.

2.3.5. Datenhaltung: MariaDB

In dem vorliegenden Versuchsaufbau wird eine relationale MariaDB-Datenbank eingesetzt, die auf dem bewährten MySQL-System aufbaut. MariaDB ist eine international weit verbreitete Open-Source-Datenbanktechnologie, die von namhaften Unternehmen und Plattformen wie Wikipedia, Wordpress.com und Google genutzt wird[13]. Diese Tatsache unterstreicht die Zuverlässigkeit und Beliebtheit dieser Datenbanklösung in der Praxis. Die Entscheidung, auf eine etablierte und weit verbreitete Datenbanktechnologie wie MariaDB zu setzen, gewährleistet eine solide Basis für den Versuchsaufbau. Außerdem ist diese Datenbanktechnologie für den Kontext der Relaxdays GmbH relevant, da diese Technologie dort bereits genutzt wird.

3. Methodik

Nachdem die verschiedenen Technologien und ihre jeweiligen Stärken und Schwächen dargestellt sind, wird im Folgenden auf die Methodik eingegangen.

3.1. Versuchsaufbau

Es wurde ein Versuchsaufbau konzipiert, welcher grundlegend in Abbildung 3.1 dargestellt ist. Man erkennt zwei klar getrennte Services mit den darunterliegenden zugehörigen Datenbanken. Der Bestellungen-Service fungiert in diesem Aufbau als eine Art Client und fragt Daten des Artikel-Services über dessen "Application Programming Interface" (API) ab. Es existiert auch ein geteiltes Datenschema, welches in der Implementierung durch gemeinsam genutzte C#-Klassen gebildet wird. Diese Klassen stellen eine Objekt-orientierte Sicht auf das relationale Datenmodell der Datenbank dar. Ein solches Schema ist nötig um Typ-Sicherheit zwischen den Anfragen und den Endpunkten zu ermöglichen.

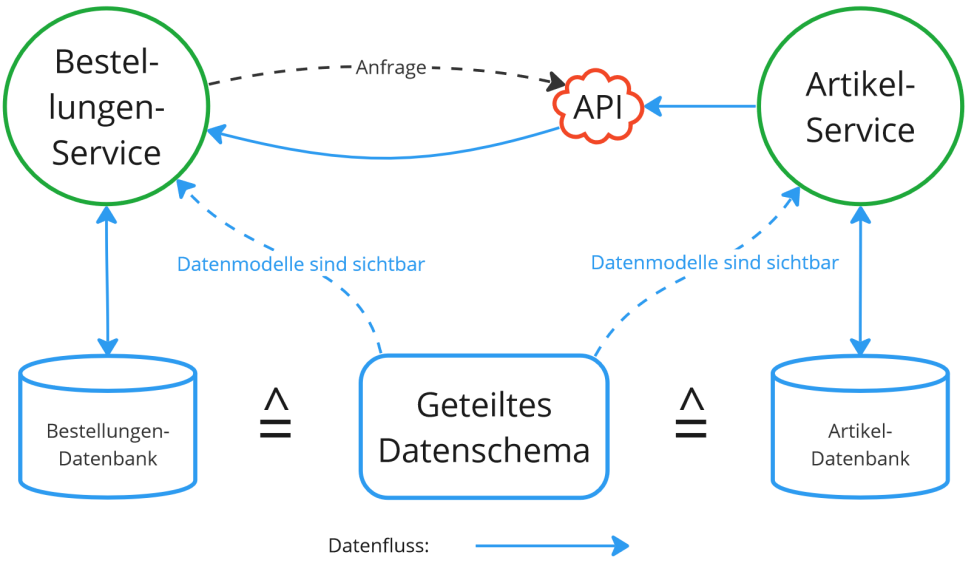

Abb. 3.1.: Logischer Entwurf des Versuchsaufbaus

3.1.1. Datenbank

Es wurde sich dazu entschieden zwei MariaDB/MySQL-Datenbankserver einzurichten, da dies die im Partnerunternehmen genutzte Technologie ist. Diese werden über ein Docker-

File mit Konfigurationsparametern in einem lokalen Docker-Netzwerk bereitgestellt und sind dann von den verschiedenen Services nutzbar. Die Testfälle werden so alle mit den gleichen Bedingungen getestet.

Für das Datenmodell wurde sich für einen leicht abgeänderten Ausschnitt aus dem Datenmodell des Partnerunternehmens entschieden, welcher auch ein paar relevante Anwendungsbeispiele für die Testfälle bietet. Das Datenmodell ist auf zwei verschiedene Datenbanken aufgeteilt. Die Bestellungen-Datenbank besteht aus Bestellungen und Bestellpositionen. Dabei verweisen die Bestellungen auf ein bis zwei Bestellpositionen. Die Bestellpositionen wiederum, verweisen auf eindeutige Artikel-"Stock Keeping Units" (SKUs)[1]. Die Artikel-Datenbank beinhaltet Artikel- und Preis-Entitäten. Die Artikel verweisen auf jeweils eine Preis-Entität (vgl. Abbildung 3.2).

Wie in Abbildung 3.1 zu sehen, ist die Artikel-Datenbank an die Artikel-Services und die Bestellungen-Datenbank an die Bestellungen-Services angebunden. So soll der Fall simuliert werden, in welchem ein Service der Bestellungen verarbeitet, Daten von einem anderen Service benötigt, der wiederum Artikel verwaltet.

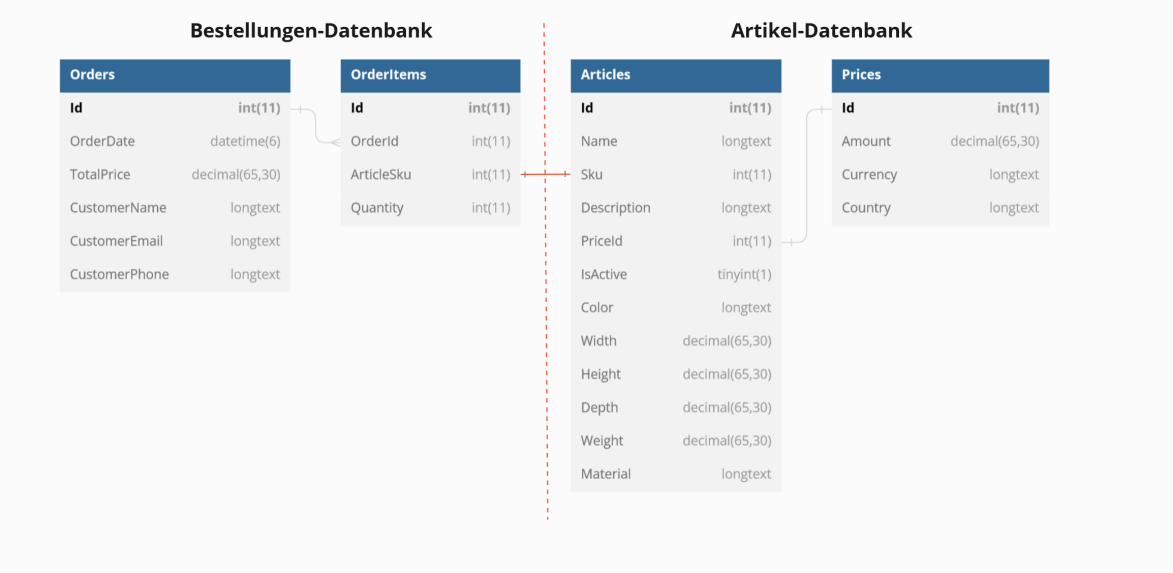

Abb. 3.2.: Relationendiagramm

3.1.2. Artikel-Services

Für jede der drei Technologien wurde ein Artikel-Service implementiert. Die Services haben jeweils eine Datenbankverbindung zu der Datenbank mit den Artikeln, über welche sie direkt

[1] auch Artikelnummern genannt

über den `DBContext`[2] Daten schreiben und lesen können. Sie werden fertig kompiliert im Lokalen Netzwerk ausgeführt (diskutiert in Abschnitt 3.4. Alle drei Services stellen auf ihre unterschiedlichen Weisen Endpunkte nach Außen zur Verfügung (APIs). Diese Endpunkte decken in allen Services die selben Funktionalitäten ab. Dabei finden die Datenabfragen aller Services zur Datenbank immer direkt über das `DBContext`-Objekt statt. Die Unterschiede der Services liegen also hauptsächlich in der Serialisierung und Deserialisierung von Daten und der Art und Weise der Bereitstellung von Endpunkten nach außen.

Jeder Service bietet folgende Endpunkte. Alle Endpunkte geben die vollständigen Objekte mit den vollständigen Informationen zurück[3].

- GetArticles(int X, string? includedData) - Gibt die ersten X Artikel zurück. Wird eine includedData-Zeichenkette angegeben, werden nur die darin spezifizierten Attribute des Artikelobjekts übertragen. Die Take(X)-Funktion[4] wird genutzt.
- GetArticlesWithPrice(int X) - Gibt die ersten X Artikel mit dem jeweils zugehörigen Preis-Objekt zurück. Die Take(X)-Funktion wird genutzt.
- GetArticleBySku(int sku) - Gibt den Artikel mit der angegebenen SKU zurück. Die First(Bedingung)-Funktion[5] wird genutzt.
- GetArticleWithPriceBySku(int sku) - Gibt den Artikel mit der angegebenen SKU zusammen mit dem dazugehörigen Preis zurück. Die First(Bedingung)-Funktion wird genutzt.
- GetPriceById(int id) - Gibt das Preis-Objekt mit der angegebenen Id zurück. Die First(Bedingung)-Funktion wird genutzt.

Hinweis: *Es wurde sich bewusst dazu entschieden die ersten beiden Endpunkte so zu implementieren, dass immer nur die ersten X Artikel zurück gegeben werden (Die Daten sind in der Datenbank nach ihren Ids sortiert). Dies ist eine Vereinfachung gegenüber echten Unternehmensprozessen, da dort meistens eine Liste von Bestellungen oder Artikeln anhand von Eigenschaften (beispielsweise anhand eines Statuses) aus der Datenbank abgefragt werden. Diese Vereinfachung hat allerdings keine Auswirkungen auf die Aussagekräftigkeit der ermittelten Ergebnisse, da sie nur dafür sorgt, dass Daten schneller aus der Datenbank gelesen werden können. Der Fokus liegt im Rahmen dieser Arbeit auf der Geschwindigkeit der Datenübertragung zwischen den Web-Services und nicht auf dem Datenaustausch zwischen dem Service und der Datenbank.*

[2] Objekt aus EF Core, über welches die Datenbankverbindung genutzt werden kann
[3] Ausgenommen der erste Endpunkt, welcher dafür eine Filter-Möglichkeit anbietet
[4] Gibt die ersten X Elemente aus der Datenbank-Tabelle zurück
[5] Gibt das erste Element aus der Datenbank-Tabelle zurück, welches eine Bedingung erfüllt

3.1.3. Bestellungen-Services

Für die Abfrage der Artikel-Daten vom Artikel-Service wurde zu jedem Artikel-Service ein Bestellungen-Service implementiert. In dem Testaufbau fungieren die Bestellungen-Services als Clients. Die Clients senden Anfragen und empfangen Daten von dem Artikel-Service mit der jeweils passenden Technologie. Außerdem haben die Bestellungen-Services eine Datenbankverbindung zu der Bestellungen-Datenbank, über welche sie direkt Bestelldaten beziehen können. Die Bestellungen-Services müssen also auch dafür sorgen, dass die Daten aus beiden Quellen sinnvoll zusammengeführt werden. Um eine Vergleichbarkeit herzustellen, haben alle Bestellungen-Services eine möglichst gleiche Routine implementiert, die nacheinander Testfälle ausführt, Daten sammelt und ausgibt. Mehr technische Details zu den Tests finden sich im nächsten Abschnitt.

3.1.4. Datenerhebung

Eine besonders hilfreiche Benchmarking-Bibliothek, die im Folgenden für die Durchführung der Testfälle genutzt wird ist "BenchmarkDotNet"[14]. Sie führt einige Warmup-Routinen durch bevor die eigentlichen Testfälle ausgeführt werden um Caches für den echten Testlauf vorzubereiten. Zudem werden automatisch Ausreißer-Messwerte herausgerechnet. Diese können durch alle möglichen Faktoren auftreten, sind aber nicht repräsentativ für das Ergebnis und können den Durchschnitt der Testdurchläufe verfälschen. Es werden auch automatisch mehrere Iterationen durchgeführt. Durch all diese Faktoren wird eine hohe Vergleichbarkeit erreicht.

Die Messungen wurden vor der Anfrage des Clients an den Server gestartet und wurden beendet, sobald alle Antwortobjekte auf der Client-Seite zusammengesetzt waren. Vorher wurde bereits die Verbindung zwischen dem Client und dem Server konfiguriert. Diese Konfiguration hat also keinen Einfluss auf die Messergebnisse.

3.2. Metriken

Für den Vergleich zwischen den Datenübertragungstechnologien fokussiert sich diese Arbeit auf folgende Metriken:

- Serialisierung von C#-Objekten auf der Server-Seite
- Deserialisierung von C#-Objekten auf der Client-Seite
- Rundlaufzeit (Round-Trip-Time)

Diese Metriken werden für verschiedene Szenarien erhoben um einen Vergleich zwischen den drei Technologien zu ermöglichen. Die Geschwindigkeiten der Serialisierung und Deserialisierung sind relevant, um den Unterschied zwischen GraphQL und REST welche das JSON-Format nutzen und gRPC, welche das Protocol-Buffer-Format nutzt zu erforschen.

Nachdem genauer auf die Serialisierung und Deserialisierung für die Technologien eingegangen wurde, wird die Rundlaufzeit im Mittelpunkt stehen.

3.3. Testfälle

Unter Verwendung der im vorherigen Abschnitt beschriebenen Metriken wurden verschiedene Testfälle konzipiert um die drei Technologien in verschiedenen Szenarios miteinander zu vergleichen. Diese sind im Folgenden aufgeführt.

3.3.1. Serialisierung und Deserialisierung

Als erstes sollen die Serialisierungs- und Deserialisierungsgeschwindigkeit der Datenformate JSON und Protocol Buffer verglichen werden. Dazu wurde ein Testaufbau erstellt bei dem ein einfaches C#-Objekt (zu sehen in Anhang: Listing 4) in ein JSON- beziehungsweise Byte-Array-Format serialisiert und anschließend wieder deserialisiert wird.

Für die JSON Serialisierung existieren im DotNet-Framework zwei große Bibliotheken. Zum einen “Newtonsoft Json” und zum anderen “System.Text.Json” von Microsoft. Im ersten Schritt werden also diese beiden Bibliotheken miteinander verglichen.

Danach wird das selbe C#-Objekt mit der Protobuf-Serialisierung und Deserialisierung getestet und der JSON-Perfomance gegenüber gestellt.

3.3.2. Datenabfragen

Nun soll der tatsächlichen Datenübertragungsweg gemessen werden. Im Folgenden werden die Testfälle für verschiedene Arten von Datenabfragen vorgestellt. Für jeden Testfall wird ein gestaffelter Test durchgeführt. Es wird jeweils mit einem Artikel beziehungsweise einer Bestellung begonnen und dann werden Steigerungen der Datenabfrage in Zehnerpotenzen bis zu 10.000 durchgeführt.

Test 1: Artikel

Zuerst werden die Technologien im Bezug auf einfache Abfragen, welche auf Seiten des Artikel-Services auf eine Tabelle begrenzt sind verglichen. Am einfachsten ist hier eine Abfrage von

einer bestimmten Anzahl an Elementen aus der Artikel-Tabelle. Der Client des Bestellungen-Services wird in verschiedenen Testrunden unterschiedlich viele Datensätze abfragen um die Performance in Abhängigkeit zur Datenmenge analysieren zu können.

Bei diesem Test wird folgendes Verhalten erwartet:

- Mit steigender Anzahl an Artikeln steigt die Rundlaufzeit
- GraphQL ist etwas langsamer als REST und gRPC auf Grund von der langsameren Server-seitigen Auflösung der Anfrage
- Bei großen Datenmengen sollte gRPC die kürzeste Rundlaufzeit aufweisen da dann die schnellere Übertragungs- und Serialisierungsgeschwindigkeit am stärksten ins Gewicht fällt

Test 2: Reduzierter Artikel

Es bietet sich an, den ersten Test noch einmal für ein reduziertes Artikelmodell mit nur einer Id und dem Namen des Artikels durchzuführen. Bei diesem Test lassen sich gut die Unterschiede in den Technologien zeigen, da GraphQL nativ die Möglichkeit mitbringt, nur bestimmte Attribute von angefragten Daten zurück zu geben. Für gRPC und REST muss hier eine eigene Filter-Logik eingebaut werden.

Bei diesem Test wird folgendes Verhalten erwartet:

- Mit steigender Anzahl an Artikeln steigt die Rundlaufzeit
- GraphQL sollte hier deutlich schneller sein, da diese Technologie als einzige der drei Technologien nativ reduzierte Daten abfragen kann
- Bei großen Datenmengen sollte gRPC zumindest eine kürzere Rundlaufzeit als REST aufweisen da dann die schnellere Übertragungs- und Serialisierungsgeschwindigkeit am stärksten ins Gewicht fällt
- Für REST und gRPC wird auf Grund der zusätzlichen Filter-Logik für kleinere Datenmengen eine langsamere Rundlaufzeit als bei dem vorherigen Testfall erwartet

Test 3: Artikel mit jeweils einem Preis (Viele kleine Abfragen)

Der dritte Test soll nun eine komplexere Abfrage abbilden. Genauer wird das klassische N+1 Problem abgebildet (siehe Quelle [15]) indem eine bestimmte Anzahl Artikel mit den dazugehörigen Preisen abgefragt werden. Für gRPC und REST wird als erstes der getArticles(int

X)-Endpunkt für eine Liste von Artikeln angefragt. Mit Hilfe der Preis-Ids aus dieser Antwort wird dann je Artikel der dazugehörige Preis am getPriceById(int id)-Endpunkt abgerufen. Diese Art der Datenabfrage kann man als "chatty"[6] bezeichnen, da viele kleine Anfragen zwischen Client und Server ausgelöst werden. GraphQL wird für diesen Testfall nicht betrachtet, da es typischerweise alle Daten in einem Request abfragt.

Bei diesem Test wird folgendes Verhalten erwartet:

- Mit steigender Anzahl an Artikeln steigt die Rundlaufzeit
- gRPC kann kaum von der schnelleren Übertragungs- und Serialisierungsgeschwindigkeit profitieren, da nur sehr viele kleine Anfragen ausgelöst werden
- Allgemein wird auf Grund der vielen Anfragen eine lange Antwortzeit erwartet

Test 4: Artikel mit jeweils einem Preis (Eine große Abfrage)

Der vierte Testfall liefert das gleiche Ergebnis wie der dritte Testfall. Nun sollen die Daten aber über eine große Abfrage auf ein Mal abgefragt werden. Dazu wird der GetArticlesWithPrice(int X)- Endpunkt angefragt. Auf der Artikel-Service-Seite von REST und gRPC kommt die include()-Funktion der Datenabfragesprache "Language-Integrated Query" (LINQ) zum Einsatz. Diese Art der Datenabfrage kann man "bulky"[7] nennen, da zwar nur wenige, dafür aber sehr große Anfragen zwischen Client und Server ausgelöst werden.

Bei diesem Test wird folgendes Verhalten erwartet:

- Mit steigender Anzahl an Artikeln steigt die Rundlaufzeit
- Bei großen Datenmengen sollte gRPC die kürzeste Rundlaufzeit aufweisen da dann die schnellere Übertragungs- und Serialisierungsgeschwindigkeit am stärksten ins Gewicht fällt
- GraphQL sollte langsamer als gRPC und REST sein, da die Technologie nicht ohne Weiteres effizient für Abfragen mit N+1-Problem funktioniert. GraphQL kann hier nämlich nicht automatisch von LINQ-Optimierungen profitieren (weitere Erläuterung in der Auswertung in Abschnitt 4.2).
- REST und gRPC werden mit einer Abfrage schneller sein, als im vorherigen Testfall, da deutlich weniger Übertragungs-Overhead durch die vielen Abfragen anfällt und die include()-Funktion weitere Optimierungen bietet

[6] deutsch: gesprächig
[7] deutsch: umfangreich, sperrig

- Allgemein wird auf Grund der großen Datenmenge eine lange Antwortzeit erwartet

Test 5: Bestellungen mit dazugehörigen Artikeln und Preis

Im fünften Test sollen nun alle Daten abgefragt werden. Dazu werden zunächst die Bestellungen mit Bestellpositionen aus der Bestellungen-Tabelle bezogen. Daraufhin werden dann die Artikel mit Preisen vom Artikel-Service abgefragt, die durch die Bestellpositionen über die Artikel-Sku referenziert werden. Dies ist ein sehr typischer Anwendungsfall im Partnerunternehmen. Die Artikel und Preise werden in einer Anfrage pro Sku an dem Endpunkt GetArticleWithPriceBySku(int sku) abgefragt. Hierbei ist die Herausforderung zu bewältigen, die Daten aus beiden Datenbanken zusammenzuführen.

Bei diesem Test wird folgendes Verhalten erwartet:

- Mit steigender Anzahl an Bestellungen steigt die Rundlaufzeit
- gRPC kann kaum von der schnelleren Übertragungs- und Serialisierungsgeschwindigkeit profitieren, da nur sehr viele kleine Anfragen ausgelöst werden
- Allgemein wird auf Grund der großen Datenmenge eine lange Antwortzeit erwartet

3.4. Technische Umstände der Testumgebung

Alle Testfälle wurden auf dem selben Notebook durchgeführt. Allerdings ist damit nicht sicher gestellt, dass alle Ergebnisse unter den exakt selben Bedingungen entstanden sind. Es wurden allerdings vor jedem Testdurchlauf alle nicht benötigten Prozesse beendet und eine Benchmarking-Bibliothek genutzt, welche automatisiert Warmup-Routinen durchführt um Caches vorzubereiten, die Testfälle mehrfach durchzuführen und auch eine Statistik anzulegen (siehe Abschnitt 3.1.4). Innerhalb eines Testdurchlaufs wurde jeder Test also mehrfach durchgeführt, ein Fehler und eine Standardabweichung berechnet und es wurden Ausreißer-Werte entfernt. Dazu wurden alle Testfälle mehrfach durchgeführt und dann die Ergebnisse ausgewählt, die die kleinsten Standardabweichungen aufwiesen. Die Ergebnisse der mehrfachen Testdurchläufe unterschieden sich allerdings nur geringfügig voneinander, weswegen die Ergebnisse insgesamt ein hohe Aussagekraft haben.

Die Services für die Testfälle laufen auf dem selben Notebook unter localhost und kommunizieren über das lokale Netzwerk. Gleichzeitig laufen die beiden Datenbanken in virtuellen Docker-Umgebungen auch auf dem selben Notebook. Die Systemeigenschaften des Notebooks finden sich in Anhang A.1.

Dieser Aufbau entspricht nicht den realen Konditionen in der Relaxdays GmbH. Dort laufen die Datenbanken und Webservices auf externen Servern, die unabhängig von einander Operationen durchführen können. Hier ist eine echte Parallelverarbeitung möglich, da physisch verschiedene Prozessoren existieren. Auf dem Notebook hingegen, teilen sich alle benötigten Prozesse den selben Prozessor. Dies verlangsamt die Prozesslaufzeiten, da der Prozessor nur eine Operation gleichzeitig bearbeiten kann. In diesem Fall wurde aber mit einem Notebook mit einem Prozessor mit 8 Kernen gearbeitet. Das bedeutet, dass immer noch ein relativ hoher Grad an Parallelverarbeitung stattfinden kann, solange der Arbeitsspeicher nicht überfüllt wird. Während der Testdurchläufe wurde der Arbeitsspeicher nicht voll belastet. Trotzdem ist dieser Aufbau nicht realitätsgetreu.

Im Rahmen dieser Arbeit müssen keine konkreten Laufzeitanforderungen erfüllt werden, sondern drei Technologien sollen zueinander in ein Verhältnis gesetzt werden. Das bedeutet, solange alle Technologien während ihrer Testdurchläufe in den selben Konditionen operieren, ist durchaus eine aussagekräftige Datenerfassung möglich. Dieser Punkt gibt allerdings Anstoß, die grundlegenden Aussagen in dieser Arbeit in einem realitätsnäheren Anwendungsfall zu überprüfen und so Daten zu sammeln, welche direkt mit den realen Daten vergleichbar wären (siehe: Abschnitt 5.3).

Zusätzlich zu den Gedanken zu der Prozessornutzung ist auch die Nutzung von lokal laufenden Docker-Containern für die Datenbanken kritisch zu betrachten. Der Datenaustausch zwischen den Services und den Datenbanken ist so keinen realen Netzwerkbedingungen wie Schwankungen in der Verbindung ausgesetzt. Diese Arbeit fokussiert sich allerdings auf den Datenaustausch zwischen den Web-Services und dafür ist dieser Aufbau sogar besser geeignet, da unbekannte Variablen wie die Netzwerkschwankungen die Ergebnisse des Vergleichs nicht verunreinigen. Mit dem gewählten Aufbau laufen alle Technologien bestmöglich in den selben Konditionen.

4. Auswertung

Im diesem Kapitel erfolgt eine Auswertung der experimentell ermittelten Ergebnisse.

4.1. Serialisierung und Deserialisierung

Zuerst sollen die Ergebnisse der Tests für die Serialisierung und Deserialisierung dargestellt werden. Die Tabelle A.2 im Anhang zeigt die Serialisierungs- und Deserialisierungsgeschwindigkeiten der zwei JSON-Bibliotheken und des Protocol-Buffer-Serialisierers.

Zur Verdeutlichung der Geschwindigkeiten in beide Richtungen wurde das Diagramm 4.1 erstellt.

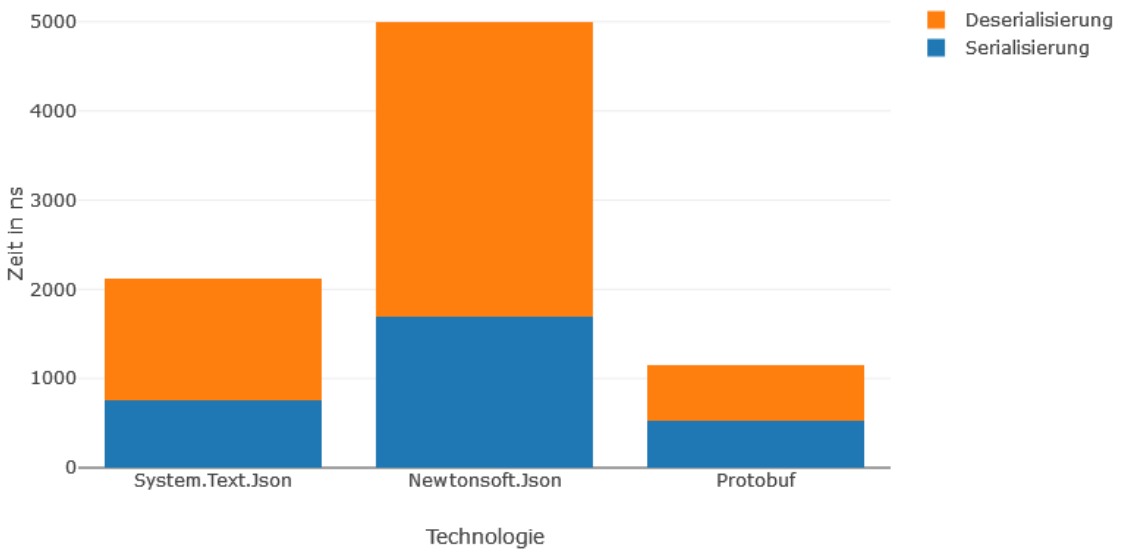

Abb. 4.1.: Serialisierung und Deserialisierung

Es lässt sich deutlich erkennen, dass die System.Text.Json-Bibliothek schneller als die Newtonsoft.Json Implementierung ist. Vor allem macht sich der Unterschied in der Deserialisierung bemerkbar, bei welcher System.Text.Json ungefähr 2,4-mal schneller ist, aber auch die Serialisierung ist ungefähr doppelt so schnell. Vergleicht man auch die Protocol-Buffer Serialisierung und Deserialisierung, ist diese wiederum noch einmal annähernd doppelt so schnell wie System.Text.Json.

Für die weiteren Testfälle wurde für die REST- und GraphQL-Tests folglich auch die System.Text.Json-Bibliothek genutzt um die beste Rundlaufzeit zu ermöglichen.

4.2. Datenabfragen

In diesem Abschnitt werden die Ergebnisse der verschiedenen Datenabfragen-Testfälle beschrieben.

Hinweis: *Es ist bei den folgenden Abbildungen zu beachten, dass die Zeit-Achse in den Diagrammen logarithmisch dargestellt ist, um in einem Diagramm alle Messwerte darstellen zu können. Hierbei geht vor allem bei hohen Messwerten ein Teil der visuell wahrnehmbaren Proportionalität verloren. Es wird empfohlen die Zahlen in den Diagrammen sowie die Tabellen, die zu jedem Diagramm im Anhang zu finden sind genau zu betrachten.*

Test 1: Artikel

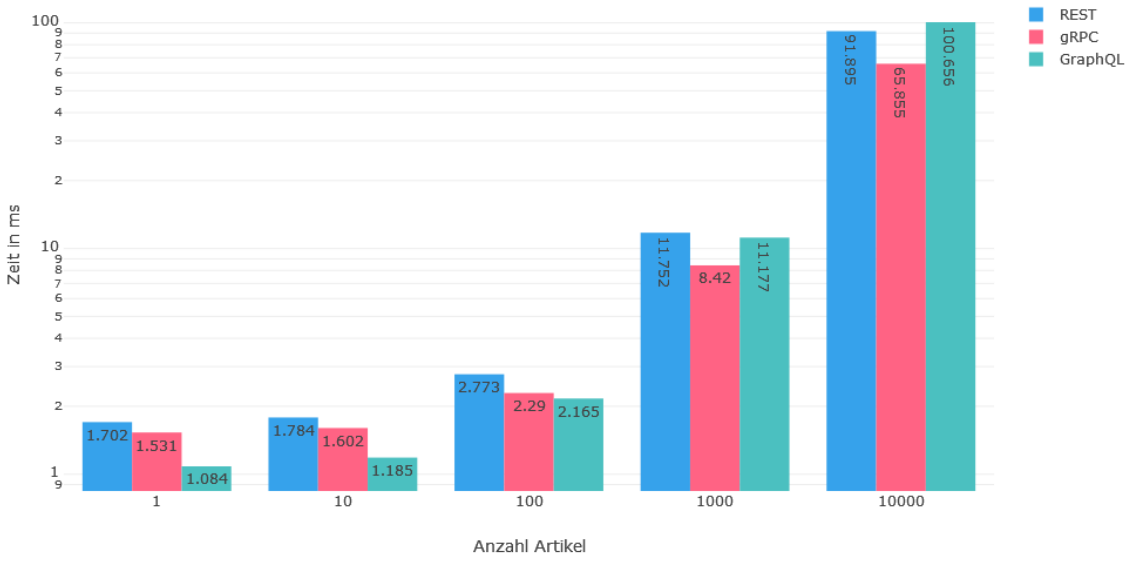

Abb. 4.2.: Ergebnis Test 1: Liste von Artikeln

Für alle beobachteten Technologien steigen, wie in Abbildung 4.2 zu erkennen, die Rundlaufzeiten mit einer zunehmenden Datenmenge. Bei den kleineren Anfragen mit 1 bis 1.000 Artikeln liefert die GraphQL-Implementierung deutlich bessere Ergebnisse als die andern bei-

den Technologien, bei größeren Mengen nimmt die relative Rundlaufzeit allerdings deutlich zu und final ist GraphQL für 10.000 Artikel sogar die langsamste Technologie. REST ist bis auf den letzten Messwert durchgehend am langsamsten. gRPC ist konstant schneller als REST. Am Anfang ist diese Differenz relativ klein, allerdings zeichnet sich ab 1.000 Artikeln eine bessere Skalierung für größere Anfragen von gRPC gegenüber REST ab. Bei 10.000 Artikeln ist gRPC mit 66 Millisekunden fast 30% schneller als REST mit 92 Millisekunden. Auffällig ist auch, dass sich die Ergebnisse von 1 und 10 Artikeln nur minimal unterscheiden.

Test 2: Reduzierter Artikel

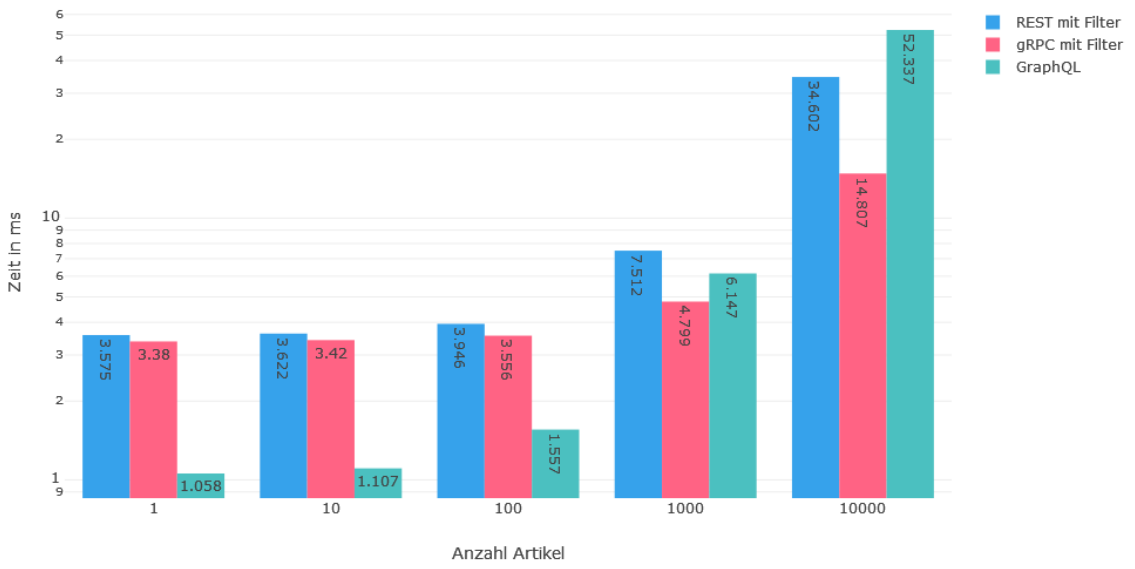

Abb. 4.3.: Ergebnis Test 2: Liste von reduzierten Artikeln

Für die Messpunkte von 1 bis 100 Artikeln fällt hier die große Differenz zwischen den Ergebnissen von gRPC und REST auf der einen Seite und den Ergebnissen von GraphQL auf der anderen Seite auf (vgl. Abbildung 4.3). Für die Messpunkte für 1 und 10 Artikel ist GraphQL mit 1,1 ms ungefähr 3 mal so schnell wie REST und gRPC mit 3,6 und 3,4 ms. REST und gRPC sind in diesem Bereich ungefähr um das Doppelte langsamer als im ersten Testfall während GraphQL sogar etwas schneller als vorher ist.

Bei den höheren Datenmengen schlägt dieser Trend allerdings um. Bei der Erhöhung von 100 Artikeln auf 1.000 Artikel verdoppelt sich die Rundlaufzeit von REST fast auf 7,1 ms, während sie sich bei GraphQL sogar vierfach auf 6,1 ms erhöht. Währenddessen skaliert

gRPC deutlich besser und erhöht die Rundlaufzeit nur um 27% auf 4,8 ms. Insgesamt ist gRPC durchweg schneller als REST und für die Messpunkte 1.000 und 10.000 ist gRPC sogar bedeutend schneller. Es zeigt sich wieder eine deutlich bessere Skalierung für gRPC.

Test 3: Artikel mit jeweils einem Preis (Viele kleine Abfragen)

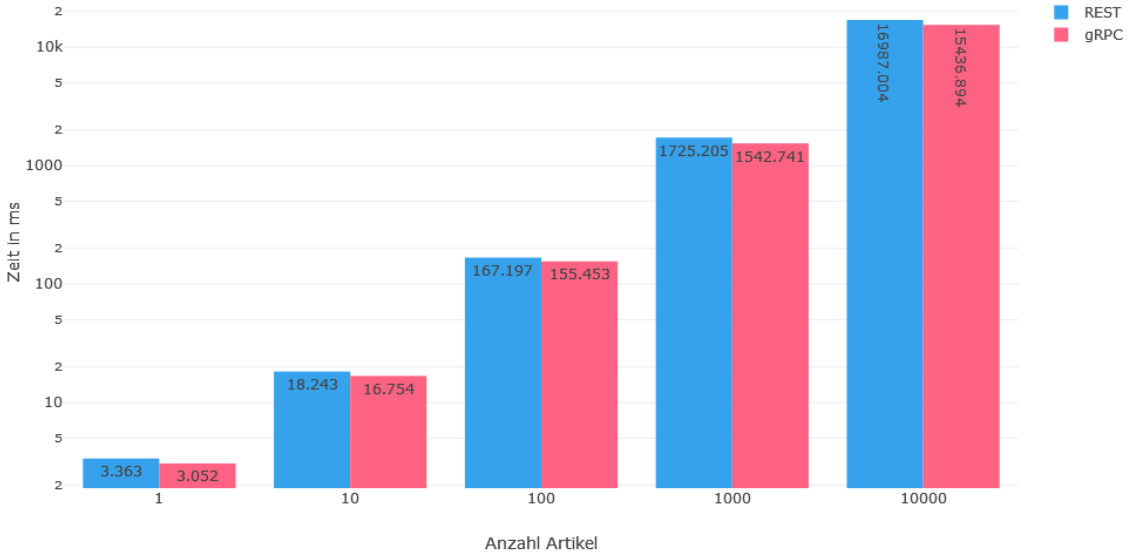

Abb. 4.4.: Ergebnis Test 3: Artikel mit Preis (“chatty”)

Die Messergebnisse in Abbildung 4.4 zeigen eine Zunahme der Rundlaufzeit für die Datenabfrage mit zunehmender Anzahl von Artikeln sowohl bei REST als auch bei gRPC. Es wurde beobachtet, dass gRPC in allen Fällen geringfügig schnellere Abfragezeiten als REST aufweist. Diese Differenz wird etwas größer mit einer Steigerung der Artikelmenge. Insgesamt sind die Rundlaufzeiten für beide Technologien sehr lang. Bereits für 1.000 Artikel dauert eine solche Abfrage 1,54 Sekunden bei gRPC und 1,725 Sekunden bei REST. Bei 10.000 Artikeln mit Preisen müssen insgesamt 17 Sekunden bei REST und 15,4 Sekunden bei gRPC gewartet werden.

Test 4: Artikel mit jeweils einem Preis (Eine große Abfrage)

Die Messergebnisse in Abbildung 4.5 zeigen eine Zunahme der Rundlaufzeit für die Datenabfrage mit zunehmender Anzahl von Artikeln für alle drei Technologien. Es fällt auf, dass

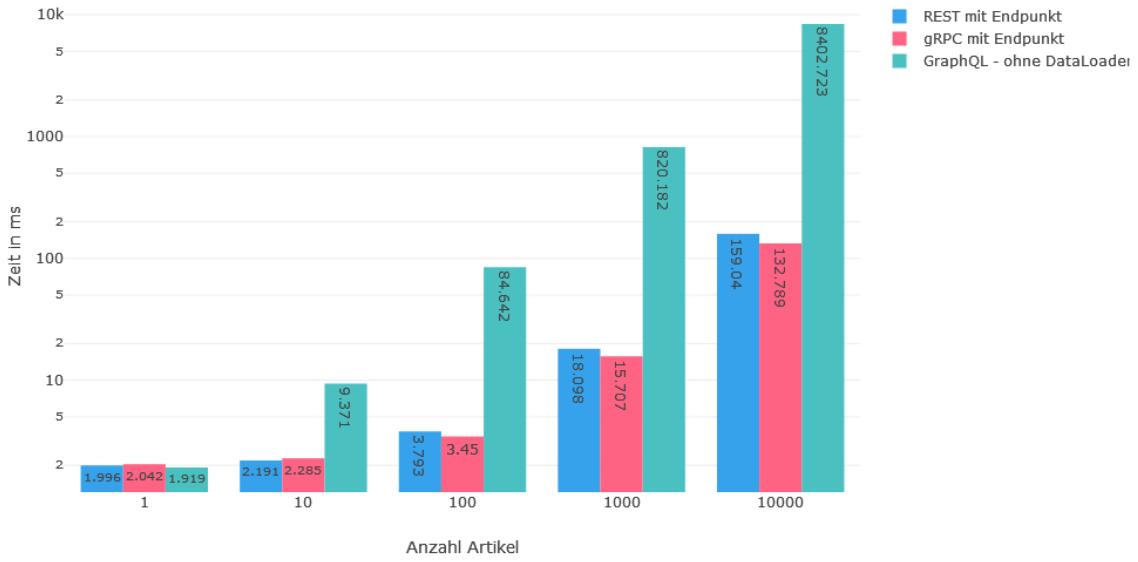

Abb. 4.5.: Ergebnis Test 4: Artikel mit Preis (“bulky”)

GraphQL sehr viel länger als die anderen beiden Technologien braucht. Bereits bei 100 Artikeln sind REST und gRPC ungefähr um das 24-fache und bei 10.000 Artikeln ungefähr um das 56-fache schneller. Die GraphQL-Implementation ohne `Data-Loader` scheint bei der Verarbeitung großer Datenmengen deutlich ineffizienter zu sein wenn das N+1-Problem vorliegt. Ansonsten kann beobachtet werden, dass REST und gRPC ähnliche Leistungswerte aufweisen, wobei gRPC in jedem Messpunkt leicht bessere Zeiten erzielt.

Ein Vergleich zwischen diesem und dem vorherigen Testfall ist interessant. Wird für REST und gRPC ein Endpunkt implementiert, der alle benötigten Daten in einer Anfrage liefert, ist ein sehr beachtlicher Performancegewinn zu verzeichnen. REST und gRPC benötigen nur noch einen Bruchteil der Zeit für die selben Daten. Im Messpunkt von 100 Artikeln ist zum Beispiel eine Verbesserung von 167,2 ms auf 3,8 ms bei REST und eine Verbesserung von 155,5 ms auf 3,5 ms bei gRPC zu beobachten. Trotz der im Vergleich sehr langsamen Performance von GraphQL in Abbildung 4.5, ist auch festzustellen, dass GraphQL sogar ohne `Data-Loader` ungefähr doppelt so schnell ist, wie gRPC und REST ohne einen dedizierten Endpunkt im dritten Testfall.

Auf Grund dieser nicht konkurrenzfähigen Ergebnisse der GraphQL-Implementierung, wurde diese um einen Data-Loader erweitert[16][17]. Die Ergebnisse dieses abgeänderten Tests

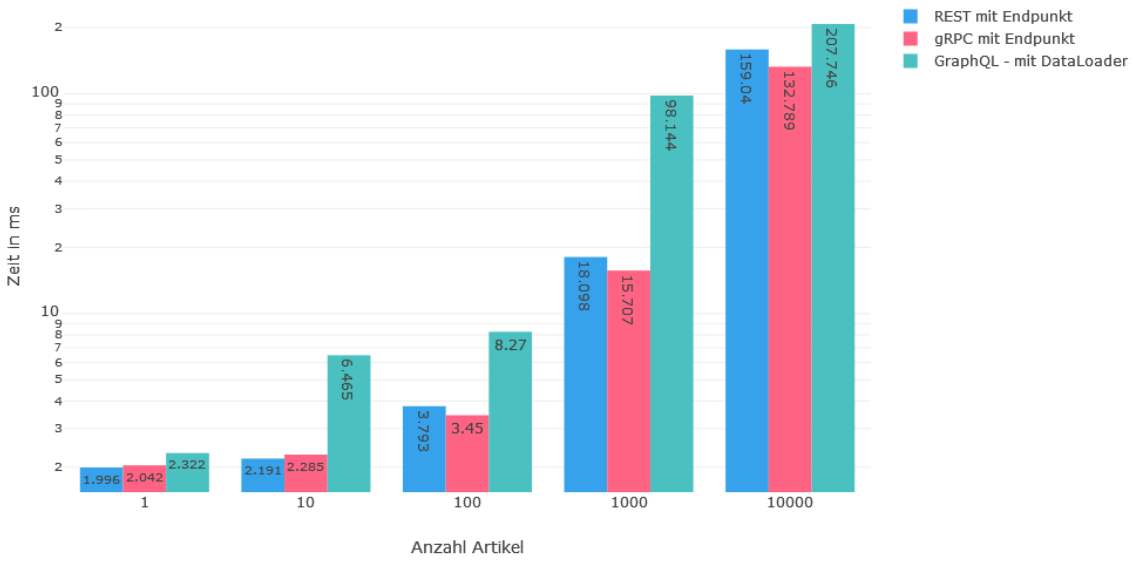

Abb. 4.6.: Ergebnis Test 4 mit verbessertem GraphQL

finden sich in Abbildung 4.6. Man kann eine drastische Verbesserung der Rundlaufzeit von GraphQL im Vergleich zur Implementierung ohne Data-Loader feststellen. Für 100 Artikel ist das neue Messergebnis ungefähr 10 mal, für 1.000 Artikel 8 mal und für 10.000 Artikel sogar 40 mal schneller. GraphQL ist zwar weiterhin langsamer als REST und gRPC mit dedizierten Endpunkten, kann nun aber näher an deren Leistungswerte heranreichen.

Vergleicht man nun wieder dieses Ergebnis mit dem dritten Testfall, ist GraphQL mit dem Data-Loader für 100 Artikel ungefähr 19 mal schneller als REST und gRPC ohne dedizierten Endpunkt. Allerdings sind gRPC und REST für die selbe Anzahl an Artikeln noch immer mehr als doppelt so schnell, wenn sie einen dedizierten Endpunkt implementiert haben.

Test 5: Bestellungen mit dazugehörigen Artikeln und Preis

Bei diesem Test (vgl. Abbildung 4.7) fällt als erstes das Ausreißer-Ergebnis von gRPC bei 10.000 Bestellungen auf[1]. gRPC benötigt in diesem Messpunkt mit 226 Millisekunden ungefähr die 3,4-fache Zeit von REST und GraphQL, welche untereinander ungefähr gleich auf liegen. Dass gRPC bei größeren Datenmengen im Vergleich langsamer wird, zeichnet sich bereits bei 1.000 Bestellungen ab. Die Werte aller drei Technologien ergeben in jedem Mess-

[1]Man beachte hier die logarithmische Zeitachse

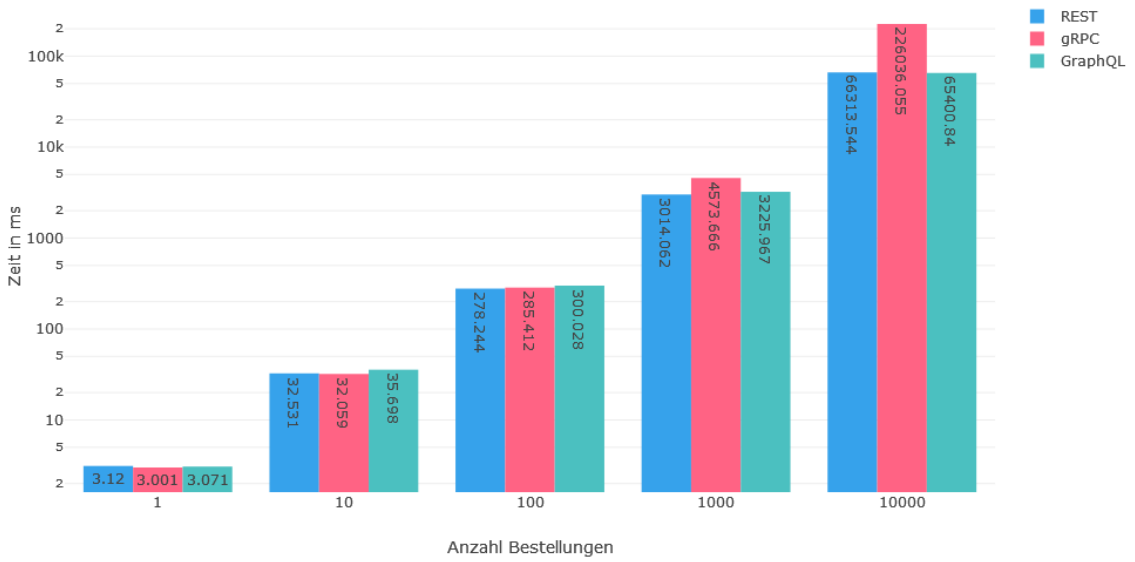

Abb. 4.7.: Ergebnis Test 5: Bestellungen mit dazugehörigen Artikeln und Preis

punkt bis zu 100 Bestellungen sehr ähnliche Rundlaufzeiten. Für REST und GraphQL setzt sich dieser Trend auch in den weiteren Messpunkten fort. Es fällt auf, dass sich die Rundlaufzeiten mit jedem Messpunkt bis auf den letzten ungefähr verzehnfachen. Der letzte Testfall bring allerdings ungefähr eine 22-fach längere Verarbeitungszeit mit sich. Allgemein sind die Laufzeiten für alle Messpunkte verglichen mit den anderen Testfällen sehr langsam.

4.3. Diskussion

Die im letzten Abschnitt dargestellten Ergebnisse werden in diesem Abschnitt kontextualisiert und diskutiert.

Die Ergebnisse der Tests zur Serialisierung und Deserialisierung von JSON- beziehungsweise Protocol-Buffer-Dateien zeigen eine sehr deutliche Abstufung der Geschwindigkeiten untereinander. Das Argument des Protocol-Buffer-Datenformats schneller zu sein, kann hier eindeutig bestätigt werden. Trotzdem ist zu beachten, dass alle Verfahren im Nanosekundenbereich arbeiten und dadurch die Auswirkungen der kleineren Datenpakete und schnelleren Serialisierung von gRPC erst bei sehr großen Datenmengen zum Tragen kommen.

Die Unterschiede zwischen NewtonSoft.Json und System.Text.Json liegen nicht nur in der Geschwindigkeit der Serialisierung und Deserialisierung. System.Text.Json ist zwar perfor-

manter, was an hoch optimiertem Code und einer effizienteren Arbeitsspeichernutzung liegt, dafür bietet NewtonSoft.Json mehr Funktionalitäten und setzt auf mehr Flexibilität in der Anwendung[18][19].

Die Daten die bei dem ersten Datenabfragen-Testfall ersichtlich wurden zeigen, dass der eigentliche zusätzliche Overhead von GraphQL bei Datenabfragen nach einfachen Listen keine Auswirkungen hat. GraphQL bildet einen Anfrage-Graphen, der dann für die Antwort aufgelöst wird. Wird wie hier nur ein Datenknoten ohne Verschachtelung aufgelöst, ist diese Abfrage für kleinere Listen sehr effizient und somit sogar schneller als REST und gRPC. Die immer besser werdende Rundlaufzeit von gRPC im Vergleich zu REST ist mit der effizienteren Übertragung und Serialisierung von gRPC zu erklären. Bei großen Datenpaketen fallen diese Faktoren besonders stark ins Gewicht.

Werden anstatt vollständiger Datensätze nur Datensätze mit bestimmten Attributwerten abgefragt (Testfall 2), muss GraphQL insgesamt weniger Attributwerte auflösen. Da GraphQL diese Funktionalität als einzige dieser Technologien nativ beherrscht, ist es wenig überraschend zu sehen, dass sich die guten Werte von GraphQL noch ein mal verbessern. Möchte man bei REST und gRPC nur bestimmte Attributwerte in der Server-Antwort haben, muss für diese Technologien ein gesonderter Endpunkt mit spezieller Filter-Logik implementiert werden. Die Auswirkungen dieses zusätzlichen Overheads sieht man sehr deutlich in den Messwerten von 1 bis 100 Artikeln, in welchen sich die Ergebnisse zu dem vorangegangenen Test sogar verschlechtern. Erst ab dem Messpunkt von 1.000 Artikeln sind die Ergebnisse von REST und gRPC im Vergleich besser als vorher. An diesem Punkt, übertrifft die eingesparte Übertragungs- und Serialisierungszeit auf Grund der reduzierten Datenmenge die benötigte Zeit für die Filterlogik. Es lässt sich also festhalten, dass es sich bei REST und gRPC im Allgemeinen nicht lohnt, spezielle Datenfilter einzubauen. Der Mehraufwand einer solchen Implementation würde den potentiellen Geschwindigkeitsgewinn bei großen Datenmengen nicht rechtfertigen.

Die Ergebnisse des dritten Testfalls scheinen erst einmal unintuitiv, denn gRPC ist auch für sehr große Datenmengen nur geringfügig schneller als REST. Die nur gering bessere Rundlaufzeit von gRPC gegenüber REST ist dadurch zu erklären, dass gRPC nur bei größeren Datenpaketen einen merklichen Vorteil gegenüber REST hat. In diesem Testfall wurden allerdings sehr viele einzelne Anfragen nach Artikeln und Preisen durchgeführt. Bei vielen kleinen Datenpaketen kann die optimierte Übertragungsgeschwindigkeit und vor Allem die Serialisierung und Deserialisierung von gRPC ihre Vorteile nicht ausnutzen. Dieses Verhalten für kleine Anfragen lässt sich auch in den vorherigen Tests beobachten. Schaut man sich den

ersten Testfall und dort den Messwert für einen Artikel an, sieht man einen sehr ähnlichen, kleinen Unterschied zwischen den Rundlaufzeiten. Im Grunde ist der dritte Testfall ähnlich zu einer Aneinanderreihung dieses Messpunktes. Es ergibt also Sinn, dass sich dieser Trend fortsetzt.

Die allgemein sehr langen Antwortzeiten sind hier mit dem Overhead einer HTTPS-Anfrage zu erklären. Da sehr viele kleine Anfragen ausgelöst werden, wird jedes mal eine neue HTTPS-Anfrage gesendet und diese Übertragungszeiten addieren sich auf.

Zwar wird im vierten Testfall im Gegensatz zu REST und gRPC wieder kein gesonderter Endpunkt für GraphQL benötigt, aber die Übertragungsgeschwindigkeiten des ersten Tests ohne DataLoader sind so schlecht, dass GraphQL für diesen Anwendungsfall trotzdem nicht konkurrenzfähig wäre. Die langsame Geschwindigkeit von GraphQL ohne `Data-Loader` ist dadurch zu erklären, dass zwar nur eine Anfrage über das Netzwerk benötigt wird, dann aber auf der Server-Seite viele kleine Datenabfragen in Richtung Datenbank durch die Server-seitigen Resolver ausgelöst werden. Das N+1 Problem wird also nur in Richtung der Datenbankanbindung verschoben. REST und gRPC können hier von Batching-Optimierungen profitieren, die durch die Server-seitige Nutzung von LINQ ohne zusätzlichen Aufwand aktiv werden. Man sieht diesen starken Unterschied direkt, sobald mehr als ein Artikel mit Preis angefragt wird.

Ist aber ein `Data-Loader` für GraphQL implementiert, sorgt dieser dafür, dass die Preis-Abfragen, die vorher für jeden Artikel einzeln an die Datenbank gesendet wurden, nun für alle Artikel gebündelt gesendet werden. Dieser Batching-Prozess erhöht die Rundlaufzeit drastisch und das N+1 Problem ist gelöst. Zwar ist die GraphQL Rundlaufzeit auf Grund des zusätzlichen Server-seitigen Overheads weiterhin langsamer als die von gRPC und REST, kann nun aber deutlich besser mit ihnen konkurrieren. Es empfiehlt sich also, in jeder größeren Implementation von GraphQL einen solchen `Data-Loader` zu implementieren.

Die sehr langen Zeiten, die die Anfragen im fünften Testfall allgemein benötigen, haben mehrere Gründe. Zum einen müssen zuerst die Bestellungen mit den dazugehörigen Bestellpositionen über die direkte Datenbankverbindung des Bestellungen-Services abgefragt werden. Die Bestellungen haben beim Erstellen der Datenbank alle über eine Zufallsfunktion 1 bis 2 Bestellpositionen erhalten, was bedeutet, dass zu jeder Bestellung im Schnitt 1,5 Bestellpositionen gehören. Die Geschwindigkeiten von nur diesem ersten Schritt finden sich in Abbildung A.1. Wie man sieht, sind diese Werte relativ gering und haben nur wenig Einfluss auf das Ergebnis.

Im nächsten Schritt werden die Artikel mit ihren jeweiligen Preisen über die SKUs in

den Bestellpositionen abgefragt. Dies geschieht durch separate Anfragen an den Endpunkt GetArticleWithPriceBySku(int sku). Da mit dieser Implementierung stets nur ein Artikel auf einmal angefragt wird, kann keine der Technologien von Batching-Optimierungen profitieren.

Ein weiterer Faktor und großer Unterschied ist die Funktion die auf der Server-Seite zum Abfragen der Daten von der Artikel-Datenbank genutzt wird. Es werden nicht wie bei anderen Testfällen nur die ersten X Artikel über die Take()-Funktion abgerufen, sondern jedes mal wird das Ergebnis über die jeweilige SKU von der First()-Funktion gesucht. Die First()-Funktion durchsucht so lange die Datenbanktabelle, bis ein passender Eintrag zu der SKU gefunden wurde. Dieses Verfahren ist offensichtlich deutlich langsamer. Das Verhalten der First()-Funktion liefert auch eine Erklärung für die 22-fache Erhöhung der Antwortzeit bei Verzehnfachung der Bestellungen im letzten Messpunkt. Bei 1.000 Bestellungen, werden am Ende ungefähr 1.500 Artikel abgefragt, allerdings ist die Datenbank so aufgebaut, dass die SKUs der Artikel in der Artikeldatenbank genau wie die Bestellpositionen sortiert sind. Sollen also die die ersten 1.000 Bestellungen mit Artikeln abgefragt werden, so werden letztendlich auch genau die obersten ungefähr 1.500 Artikel aus der Artikel-Datenbank benötigt. Diese oberen Artikel in der Datenbank werden durch die First-Funktion deutlich schneller gefunden als die Artikel, die weiter hinten liegen. Das Ergebnis wird also durch den Aufbau der Test-Datenbank verfälscht. Die Aussagekräftigkeit dieses Testfalls sollte durch eine zufällige Verteilung der Artikel in der Artikel-Datenbank erhöht werden können.

Das sehr langsame Ergebnis von gRPC bei 10.000 Bestellungen ist sicherlich auf ähnliche Ursachen zurückzuführen, ist aber noch ein mal deutlich stärker betroffen als REST und GraphQL. Dieses Verhalten trat bei allen Iterationen des Testdurchlaufs auf. An dieser Stelle wird dieses Ergebnis nicht weiter erforscht, da es eine Vielzahl von Ursachen haben kann, welche im Rahmen dieser Bachelorarbeit nicht voll umfänglich erforscht werden können. Es wird in der Gesamtbewertung nicht berücksichtigt. Hier würde sich allerdings im Rahmen von Optimierungsarbeiten eine weitere Nachforschung anbieten (siehe Abschnitt 5.3).

Die Auswertung der Testfälle ergibt, dass gRPC bei üblichen Datenmengen im Unternehmen (ungefähr 1-1000 Datensätze) ähnliche Geschwindigkeiten wie REST aufweist. Jedoch zeigt sich bei größeren Datenpaketen pro Anfrage eine deutliche Geschwindigkeitsverbesserung von gRPC im Vergleich zu REST auf Grund der effizienteren und damit schnelleren Serialisierung und Datenübertragung. Soll eine Anwendung durchschnittlich sehr hohe Datenmengen pro Anfrage übertragen, ist gRPC also sehr empfehlenswert.

REST liefert allgemein sehr solide Ergebnisse. Für Datensätze pro Anfrage im Bereich von 1 - 1000 ist die Geschwindigkeit ungefähr gleich schnell wie gRPC. Allerdings wurde festge-

stellt, dass REST bei größeren Datenmengen pro Anfrage nicht so gut skaliert wie gRPC. REST hat allerdings den Vorteil, dass es die deutlich weiter verbreitete Technologie ist und viele Bibliotheken und Optimierungsmöglichkeiten für REST angeboten werden.

Es stellt sich heraus, dass die Nutzung von GraphQL insbesondere dann empfehlenswert ist, wenn eine sehr flexible API bereitgestellt werden soll. Besonders für Datenabfragen ist Flexibilität ein wichtiger Faktor denn oft werden nur ganz spezifische Datensätze benötigt. Durch die Verwendung von GraphQL können Anfragen genau auf die benötigten Daten zugeschnitten werden, was eine effizientere Datenübertragung ermöglicht und Over- und Underserving von Daten vermeidet. Diese gewonnene Flexibilität geht allerdings auf Kosten der Rundlaufzeit. Die Server-seitige Umwandlung der Anfrage in einen Graphen und die anschließende Auflösung der einzelnen Attribute stellt einen zusätzlichen Overhead dar, der vor allem komplexere Anfragen verlangsamt.

GraphQL weist in der Auswertung wesentliche Unterschiede zu den anderen beiden Technologien auf. Die Ergebnisse zeigen, dass GraphQL eine bessere Leistung im Vergleich zu REST und gRPC bietet, wenn entweder keine vollständigen Datensätze benötigt werden oder wenn REST und gRPC mehrere Anfragen durchführen müssten, um die benötigten Daten zu liefern. Dies lässt sich besonders gut im Vergleich der Ergebnisse zwischen den Testfällen 1 und 2 aber auch 3 und 4 erkennen. Werden nur Daten ohne Verschachtlungen in die Tiefe abgefragt, ist GraphQL sogar die schnellste Alternative.

Der relevanteste Unterschied zwischen GraphQL und den beiden ähnlich funktionierenden Technologien REST und gRPC, ist die Bereitstellung spezifischer Daten an ihren jeweiligen Endpunkten. Wie in den Testfällen deutlich wurde, ist eine Abfrage von Artikeln mit dazugehörigen Preisen mit jeder Technologie möglich, wenn zu jedem Datenobjekt, welches an der API zur Verfügung gestellt werden soll genau ein Endpunkt existiert. So kann bei REST und gRPC zuerst der Artikel und dann über einen Schlüssel in einer zweiten Abfrage der Preis abgefragt werden. Bei GraphQL kann direkt in einer Anfrage an den Artikel-Endpunkt der Preis mit abgefragt werden und es wird kein gesonderter Preis-Endpunkt benötigt.

Angenommen, es solle nun auch der Bestand des Artikels am Endpunkt zur Verfügung gestellt werden, bräuchten REST und gRPC dann einen weiteren Endpunkt für den Bestand. Dies würde einen weiteren Implementationsaufwand bedeuten. GraphQL würde in diesem Fall keinen neuen Endpunkt benötigen. Nur die Query auf der Client-Seite müsste angepasst werden.

Man erkennt, dass GraphQL in dieser Hinsicht schon einen gewissen Vorteil in Skalierung und Flexibilität hat. Zusätzlich ist GraphQL in diesem Szenario deutlich performanter, wie im Vergleich zwischen den Testfällen 3 und 4 zu erkennen ist.

Es wurde jedoch auch festgestellt, dass REST und gRPC durch die Bereitstellung spezifischer Endpunkte, welche diese Daten in einer einzigen Anfrage zurückgeben, deutlich geringere Antwortzeiten als GraphQL aufweisen (siehe Testfall 4). Es ergibt also viel Sinn für jeden spezifischen Fall einen eigenen Endpunkt zu implementieren. Für dieses Beispiel wären dann schon 4 Endpunkte (`getArticle()`, `getArticleWithPrice()`, `getArticleWithStock()`, `getArticleWithPriceAndStock()`) nötig, um in allen Fällen von dieser höheren Geschwindigkeit gegenüber GraphQL profitieren zu können.

Eine Implementierung eines speziellen Endpunktes für jede neue Art der Datenabfrage für REST und gRPC ist also aufwändig und skaliert sehr schlecht. Das Problem ist nur dann nicht besonders ausschlaggebend, wenn nur wenige unterschiedliche Kombinationen von Daten vom Client benötigt werden und nur selten neue Datensätze hinzu kommen.

Zusammenfassend lässt sich sagen, dass die Auswahl der geeigneten Technologie von den Anforderungen des konkreten Anwendungsfalls abhängt. Im Grunde wurde ein Grundkonflikt zwischen Flexibilität und Performanz festgestellt. GraphQL liefert Flexibilität, ist dafür aber im Allgemeinen weniger performant, REST und gRPC bieten in einem etwas starreren Rahmen aber eine schnellere Rundlaufzeit. Für durchschnittlich größere Datenmengen pro Anfrage überzeugt gRPC.

In Bezug auf die Leistung bei unternehmenstypischen Datenmengen sind REST und gRPC zwar ähnlich gut geeignet, da REST aber bereits im Unternehmen implementiert ist und eine Einführung von gRPC eine neue, tiefgreifende und den Mitarbeitern unbekannte Implementierung erfordern würde, werden die Vorteile von gRPC bei großen Datenmengen als nicht ausreichend betrachtet, um einen Wechsel der Technologie zu gRPC zu empfehlen.

Für eine Entscheidung zwischen GraphQL und REST im Kontext der Relaxdays GmbH müssen die Faktoren Flexibilität, Performanz, Funktionalität und Implementierungsaufwand abgewogen werden.

Entweder hält man an der bereits im Unternehmen genutzten REST-Technologie fest, welche sehr solide Testergebnisse liefert aber gesonderte Endpunkte für einige typische Anforderungen benötigen würde, um eine gute Leistung zu ermöglichen. Der Implementierungsaufwand wäre für REST zwar nicht zu unerheblich, ist allerdings insgesamt als nicht sehr hoch einzuschätzen, da die Entwickler bereits den Umgang mit der Technologie kennen. Wie oben beschrieben, stellt die fehlende Flexibilität allerdings ein Problem dar, wenn das Datenmodell angepasst wird.

Alternativ implementiert man mit GraphQL eine neue Technologie, die zwar für größere Datenmengen oft leicht schlechtere Antwortzeiten mitbringt, dafür aber sehr flexible Daten-

abfragen ohne weitere Server-seitige Implementationen ermöglicht und bei kleinen Anfragen sogar mit sehr schnellen Antwortzeiten überzeugt. Natürlich wären für GraphQL erst aufwendige Vorbereitungen nötig, um eine dem Unternehmen unbekannte Technologie breit und einheitlich umzusetzen und die Mitarbeiter zu schulen. Hinzu kommt, dass GraphQL in C# noch nicht sehr lange existiert und damit noch nicht viele Bibliotheken existieren, die wichtige Funktionen hinzufügen.

4.4. Herausforderungen

Bei der Implementierung der verschiedenen Technologien unter Verwendung des Frameworks ASP .NET Core in C#, taten sich einige Herausforderungen hervor. Diese sollen hier heraus gestellt werden, da sie durchaus relevant für die Umsetzung der Technologien in der Relaxdays GmbH sind.

4.4.1. Typ-Sicherheit

Hauptsächlich kamen diese Herausforderungen auf, während versucht wurde, das EF-Core-Datenmodell mit seinen Datenobjekten in die, für die jeweilige Technologie wichtigen Datenobjekte zu übersetzten und Typ-Sicherheit zu erhalten. Dadurch, dass festgelegte C#-Klassen als grundlegendes Datenschema dienen sollen, wie es in der Relaxdays GmbH überall der Fall ist, gab es Konflikte mit der Standard-Nutzungsweise der verschiedenen Technologien. Im Folgenden werden diese genauer erläutert.

REST

Bei REST gab es in dieser Hinsicht keine Probleme in der Implementierung, da diese Technologie sehr gut in C# integriert ist. Um die Technologie nutzen zu können, muss man nur einen sogenannten Controller implementieren, der die verschiedenen Endpunkte definiert. Hierbei kann einfach auf die im EF-Core Datenmodell definierten C#-Klassen referenziert werden um Typ-Sicherheit zu gewährleisten. Auf der Client-Seite ist das etwas anders. Hier muss die Anfrage-URL aus dem Namen des Endpunkts und den benötigten Parametern zusammen gesetzt werden. Diese URL ist am Ende ein zusammengesetzter String ohne eine festgelegte Form. Erst zur Laufzeit, wenn dieser Endpunkt angefragt wird, kann ein Fehler auffallen. Wenn der Anfrage-String korrekt war, kommt die Antwort des Endpunkts als JSON-String zurück. Um wieder ein typisiertes Objekt zu erhalten, muss ein JSON-Deserialisierer diesen String in ein vorher festgelegtes Objekt umwandeln. Sollte der String nicht die korrekte Struktur des erwarteten Objektes erhalten, kommt es auch zu einem Laufzeitfehler. Um die Deserialisierung in die Klassen, die in dem Grundschema definiert sind erfolgreich durchzuführen, müssen die Attribute der Klassen nur mit einer Annotation versehen werden. Diese

Annotation ist nicht einmal nötig, wenn sich der Name in der C#-Definition und der Name in dem serialisierten JSON-String gleichen.

gRPC

Würde man ein neues Projekt mit gRPC aufbauen, dann würde man alle Endpunkte über eine gRPC-eigene .proto-Datei definieren. In einer solchen Datei wären dann Deklarationen von Funktionen und Klassen enthalten. Dieses Schema wird dann für die Client-Anwendung und für die Endpunkte zur Verfügung gestellt und so kann beides Typ-sicher implementiert werden. Bei Relaxdays bilden aber wie bereits erwähnt C#-Klassen das grundlegende, geteilte Datenschema. Es musste nun also eine Möglichkeit gefunden werden, diese beiden Darstellungsformen zu vereinen. Für diese Herausforderung wurde eine Bibliothek genutzt (protobuf-net[20]), welche es ermöglicht, normale C#-Klassen mit Protobuf-Annotationen zu versehen und dann für die gRPC Übertragung zu nutzen (vgl. Code in Listing 4.1). Ähnlich dazu konnte auch ein annotiertes Interface für den gRPC-Endpunkt definiert werden. Mit diesen annotierten Klassen kann dann auch auf der Client-Seite mit gesicherten Datentypen gearbeitet werden. Jedoch hat diese Methode einen kleinen Nachteil: Alle Anfragen und Antworten an den Endpunkt müssen als annotierte Objekte definiert werden. In dem zu Grunde liegenden Testaufbau war es nötig, dass sowohl ein spezielles Anfrage-Objekt für Integer-Anfragen als auch ein spezielles Antwort-Objekt für einzelne Artikelantworten und Listen von Artikelantworten implementiert werden mussten. Damit geht Flexibilität verloren.

```
1  [ProtoContract]
2  [ProtoInclude(6, typeof(Price))]
3  public class Article
4  {
5      [ProtoMember(1)]
6      public int Id { get; set; }
7
8      [ProtoMember(2)]
9      public string? Name { get; set; }
10
11     [ProtoMember(3)]
12     public int Sku { get; set; }
13
14     [ProtoMember(4)]
15     public string? Description { get; set; }
16
17     [ForeignKey(nameof(Price))]
18     [ProtoMember(5)]
19     public int PriceId { get; set; }
20
21     [ProtoMember(6)]
```

```
22      public Price? Price { get; set; }
23
24      //...
25  }
```

Listing 4.1: Protobuf-Annotationen in der Artikel-Klasse

GraphQL

GraphQL basiert auf einem GraphQL-Schema welches das Datenmodell der Schnittstelle definiert. Darin enthalten sind die Modelle der Objekte und Funktionsdefinitionen unterteilt in Lese- und Schreiboperationen (Queries und Mutations). Normalerweise muss dafür eine eigene .graphql-Datei erstellt werden, die sich in ihrem Aufbau stark an dem JSON-Format orientiert (vergleiche Listing 3). Da in dem konkreten Anwendungsfall meiner Arbeit aber die C#-Klassen die Grundlage für dieses Schema bilden sollen, musste auch hier eine Lösung gefunden werden. Auf der einen Seite gibt es Code-Generatoren, die aus bestehenden Klassen .graphql-Dateien erzeugen können. Dies könnte in einer Deployment-Pipeline als automatischer Schritt erfolgen. In dem hier beschriebenen Testaufbau sollte aber auf Code-Generierung verzichtet werden und alle verschiedenen Technologien sollten die selben Klassen nutzen.

Die größte Bibliothek ("GraphQL" [21]), die GraphQL in .NET nutzbar macht und von der GraphQL Foundation entwickelt wird, bietet hierfür eine Lösung. Für jede Klasse, deren Objekte über die GraphQL-Schnittstelle ausgetauscht werden sollen, muss eine neue Klasse definiert werden, die die Grundklasse als generisches Argument erhält. In dem Konstruktor dieser neuen GraphType-Klassen werden dann Funktionen für die Attribut-Zugriffe definiert. Im Grunde wird hier eine Abbildung der GraphType-Klasse auf die Grundklasse vorgenommen. Diese auf den Grundklassen basierende GraphType-Klassen können dann in der Definition von Funktionen der Schnittstelle verwendet werden. Eine solche Klasse ist beispielhaft in Listing 4.2 dargestellt. Es ist also durchaus möglich, Typ-Sicherheit mit Verweis auf die Grundklassen auf der Seite des Endpunktes herzustellen. Allerdings fällt in der Implementierung auf diese Weise ein nicht zu vernachlässigender und potentiell auch fehleranfälliger Aufwand an.

Dazu kommt, dass die Flexibilität auf der Client-Seite relativ beschränkt ist. Es gibt Werkzeuge wie "GraphQL Playground"[22] oder "Altair"[23], mit welchen man einen GraphQL-Endpunkt mit Echtzeit-Syntax- und Typ-Prüfung manuell testen kann. Will man aber automatisiert in einem Programm mit einem GraphQL Endpunkt kommunizieren, muss man vorher Queries als Zeichenkette ohne Typ-Prüfung zur Compile-Zeit definieren.

Um dieses Problem zu lösen existiert bereits eine Bibliothek (GraphQLinq[24]), welche aber noch nicht sehr viel genutzt wird. Mit GraphQLinq lassen sich aus dem GraphQL-Schema des Endpunktes, welches jederzeit von einem aktiven GraphQL-Endpunkt abgerufen werden

kann, Dateien generieren, welche die Klassen des Schemas abbilden. Mit deren Hilfe ist es dann möglich, stark typisierte Queries mit Hilfe der bereits bekannten und in C# viel genutzten LINQ-Abfragesyntax zu schreiben. Bei dieser Methode ist allerdings zwingend zu beachten, dass Breaking-Changes[2] hier zu Fehlern führen können, wenn sich das Datenschema des Endpunktes ändert, die Clients aber noch nicht die neu generierten Klassen benutzen. Breaking-Changes müssten mit dieser Methode also vermieden oder die API müsste versioniert werden. Es wäre lohnenswert, sich diese Bibliothek genauer anzuschauen und zu prüfen, in wie weit sie in dem benötigten Anwendungsfall genutzt werden kann (siehe Abschnitt 5.3).

```
1  public sealed class ArticleType : ObjectGraphType<Article>
2  {
3      public ArticleType(ArticleDbContext articleDbContext,
           IDataLoaderContextAccessor dataLoader)
4      {
5          Field(x => x.Id);
6          Field(x => x.Name);
7          Field(x => x.Description, true);
8          Field(x => x.Sku);
9          Field(x => x.PriceId);
10         Field<PriceType>(
11             "Price",
12             resolve: context =>
13             {
14                 var loader = dataLoader.Context!.GetOrAddBatchLoader<int,
                       Price>(
15                     "GetPriceById",
16                     articleDbContext.GetPricesByIdAsync);
17                 return loader.LoadAsync(context.Source.PriceId);
18             });
19         Field(x => x.IsActive, true);
20         Field(x => x.Color, true);
21         Field(x => x.Width, true);
22         Field(x => x.Height, true);
23         Field(x => x.Depth, true);
24         Field(x => x.Weight, true);
25         Field(x => x.Material, true);
26     }
27 }
```

Listing 4.2: GraphQL-ArticleType Klasse, welche die C#-Artikel Klasse abbildet

[2]Tiefergreifende Änderungen des Datenmodells, welche nicht mit vorher funktionierenden Queries kompatibel sind

4.4.2. Datenbank Schlüsselverweise

Das Auftrennen der monolithischen Datenbank in mehrere kleinere Datenbanken führt dazu, dass Fremdschlüssel-Beziehungen zwischen Daten, die sich danach in unterschiedlichen Datenbanken befinden, nicht mehr funktionieren können. Damit wird das Referenzieren von Datensätzen zueinander in vielen Fällen unmöglich. Auch Datenbankfunktionen wie On-Delete-Cascade können nicht funktionieren.

Im Rahmen dieser Arbeit konnte das Problem zwar teilweise mit einem Verweis von der Bestellposition auf die eindeutige SKU des bestellten Artikels gelöst werden, in der Regel gibt es aber keinen alternativen, eindeutigen Schlüssel der genutzt werden kann um die entsprechenden Einträge zu referenzieren. Dieses Problem ist von hoher Relevanz wenn diese Strategie der Datenbankauftrennung in dem realen System der Relaxdays GmbH umgesetzt werden soll denn in der realen Datenbank gibt es viel mehr Tabellen und Beziehungen als in dem hier betrachteten, reduzierten Aufbau. Potentiell könnte das Problem also an vielen Stellen auftreten. Dieser Punkt sollte im Rahmen weiterer Forschung betrachtet werden (siehe Abschnitt 5.3).

5. Fazit

Im Folgenden werden die Ergebnisse übersichtlich zusammengefasst und ein Fazit im Bezug auf die Forschungsfrage gegeben.

5.1. Zusammenfassung

In dieser Bachelorarbeit wurde ein umfassender Technologievergleich zwischen den Kommunikationstechnologien REST, GraphQL und gRPC im Kontext einer bestehenden E-Commerce Architektur durchgeführt. Das Ziel bestand darin, die am besten geeignete Technologie für den Austausch von Datenbank-Daten zwischen Microservices mit jeweils eigener Datenbank zu ermitteln. Verschiedene experimentelle Testfälle wurden entwickelt, um reale Unternehmensprozesse mit unterschiedlichen Datenmengen zu simulieren. Die Technologien wurden anhand ihrer Übertragungsleistung sowie ihrer Umsetzbarkeit im bestehenden Unternehmenskontext verglichen.

Die Auswertung der Testfälle ergibt, dass gRPC bei üblichen Datenmengen im Unternehmen (ungefähr 1-1000 Datensätze) ähnliche Rundlaufzeiten wie REST aufweist. Erst bei größeren Datenpaketen pro Anfrage zeigt sich eine deutliche Verbesserung der Geschwindigkeit im Vergleich zu REST. Da REST aber bereits im Unternehmen implementiert ist und eine Einführung von gRPC eine neue und den Entwicklern unbekannte Implementierung erfordern würde, wurden die Vorteile von gRPC bei großen Datenmengen als nicht ausreichend betrachtet, um einen Wechsel der Technologie zu gRPC zu rechtfertigen.

REST stellt sich in allen Testfällen als solide Technologie heraus, welche durchgehend gute Ergebnisse liefert, allerdings nicht ganz so gut skaliert wie gRPC. Der große Vorteil dieser Technologie ist die sehr gute Integration in den C#- und .Net-Kontext und die daraus resultierende einfache Handhabung für die Entwickler.

GraphQL hingegen weist deutliche Unterschiede zu den anderen beiden Technologien auf. Es ist auf Grund der flexiblen Funktionsweise besonders gut für sehr spezifische Anfragen geeignet. Die Ergebnisse zeigen, dass GraphQL eine bessere Leistung im Vergleich zu REST und gRPC bietet, wenn entweder keine vollständigen Datensätze benötigt werden oder wenn REST und gRPC mehrere HTTPS-Anfragen durchführen müssten, um die benötigten Daten zu liefern.

Werden für REST und gRPC jedoch speziellen Endpunkte implementiert, die diese Daten in einer einzigen Anfrage zurückgeben, sind REST und gRPC wieder performanter. Die Im-

plementierung dieser speziellen Endpunkte für jede neue Art der Datenabfrage ist allerdings sehr aufwendig und skaliert schlecht.

Ein weiterer wichtiger Faktor zur Entscheidung zwischen REST und GraphQL ist auch der Umfang der Funktionalitäten der beiden Technologien. Für REST in C# gibt es eine große Anzahl an Bibliotheken, die eine Nutzung vereinfachen können, für GraphQL als jüngere, noch nicht sehr breit genutzte Technologie gibt es weniger fertige Bibliotheken, die direkt genutzt werden können. Wichtige Bibliotheken, die beispielsweise das Typ-sichere Client-seitige Konsumieren eines GraphQL-Endpunktes vereinfachen, sind noch nicht ausgereift.

Zusammenfassend lässt sich feststellen, dass die Nutzung von GraphQL insbesondere dann empfehlenswert ist, wenn eine sehr flexible API bereitgestellt werden soll. Besonders für Datenabfragen ist Flexibilität ein wichtiger Faktor denn oft werden nur ganz spezifische Datensätze benötigt. Die für solche spezifischen Datenabfragen benötigte Implementierung und Dokumentation neuer REST- oder gRPC-Endpunkte für jeden spezifischen Anwendungsfall wäre sehr aufwändig und skaliert schlecht. Durch die Verwendung von GraphQL können Anfragen präzise auf die benötigten Daten zugeschnitten werden, was eine effizientere Datenübertragung ermöglicht. In Bezug auf die Leistung bei unternehmenstypischen Datenmengen waren REST und gRPC ähnlich gut geeignet, wobei REST aufgrund seiner bereits vorhandenen Verbreitung im Unternehmen einen gewissen Vorteil hat.

5.2. Ergebnis

Diese Bachelorarbeit hat gezeigt, dass die Auswahl der geeigneten Kommunikationstechnologie stark von den spezifischen Anforderungen und Gegebenheiten des Unternehmens abhängt. Bei einer bestehenden REST-Implementierung kann eine Umstellung auf gRPC aufgrund der besseren Leistung bei großen Datenmengen möglicherweise gerechtfertigt sein. Für Unternehmen, die eine hohe Flexibilität bei Datenabfragen benötigen und bereit sind, eine neue Technologie einzuführen, bietet sich GraphQL als vielversprechende Alternative an. Letztendlich ist es wichtig, die individuellen Anforderungen und Ressourcen des Unternehmens sorgfältig zu berücksichtigen, um die bestmögliche Technologieentscheidung zu treffen.

Für den konkreten Fall der Relaxdays GmbH ist die Frage nach der besten Technologie für diesen Anwendungsfall nur differenziert zu beantworten. Im Grunde müssen die Faktoren Flexibilität, Performanz, Funktionalität und Implementierungsaufwand im konkreten Unternehmenskontext abgewogen werden.

Da die gRPC-Technologie zwar eine sehr gute Leistung für sehr große Datenmengen auf-

weist, im Unternehmen aber noch nicht bekannt ist und REST für die eher kleineren, unternehmenstypischen Datenmengen eine ähnliche Leistung bietet, wird der hohe Aufwand einer möglichen Implementierung in näherer Zukunft als nicht sinnvoll bewertet.

GraphQL ist die beste Wahl, wenn die Relaxdays GmbH in Zukunft viel Wert auf Flexibilität in ihren Datenabfragen legen möchte. Diese Option wäre mit einem nicht zu unterschätzenden, initialen Implementierungsaufwand verbunden, hätte dann aber wenige Abhängigkeiten und könnte ohne zusätzliche Endpunkte für viele verschiedene Anfragen eine gute Antwortgeschwindigkeit liefern. Als jüngere Technologie die noch nicht sehr breit genutzt wird, hat GraphQL allerdings einen Nachteil in der Funktionalität. Wichtige Bibliotheken für die Nutzung der Technologie in C# sind noch nicht sehr ausgereift.

Der Faktor, dass REST bereits allen Entwicklern bekannt ist und schon breit im Unternehmen eingesetzt wird, wird als sehr bedeutend eingeschätzt. REST bietet auch eine allgemein sehr solide Leistung für alle Testfälle, ist allerdings weniger flexibel als GraphQL und liefert nur Datensätze zurück, die als Antworten in speziellen Endpunkten implementiert wurden. Wenn Datenzugriffe über die verschiedenen Datencluster hinweg nicht sehr häufig aufträten und somit nicht sehr viele Endpunkte implementiert werden müssten, wäre REST die beste Technologie für die Relaxdays GmbH.

5.3. Ausblick

In Zusammenhang mit den in dieser Arbeit erforschten Themen haben sich einige weitere Themenfelder aufgetan, welche für eine tiefere Betrachtung in Frage kommen.

In Abschnitt 2.2 wurden verschiedene Optionen für die Datenhaltung in Microservice-Architekturen dargestellt. Da dies nicht der Hauptfokus dieser Arbeit war, könnte eine weiterführende Untersuchung prüfen, welche Struktur sich am besten eignet. Beispielsweise bietet GraphQL inhärent eine Möglichkeit, einen Aufbau wie in Abbildung 2.3 zu realisieren[6].

Aus der Erkenntnis, dass Fremdschlüsselverweise beim Auftrennen der Datenbank verloren gehen, ergibt sich weiterer Forschungsbedarf. Es sollte untersucht werden, wie man Schlüssel über die Grenzen der kleineren Datenbanken definiert und verwaltet.

Die Vorarbeiten in dieser Arbeit können auch als Ansatzpunkt dienen, zu erforschen wie die API-Datenabfrage an einen anderen Web-Service abstrahiert und benutzerfreundlich gestaltet werden könnte. Denkbar wäre zum Beispiel eine direkte Integration der Datenabfrage in die LINQ-Syntax.

Die GraphQLinq-Bibliothek, die sehr wichtig für die Nutzung von GraphQL in C# wäre, konnte im Rahmen dieser Arbeit nicht genau betrachtet werden. Es ergibt Sinn, diese Bibliothek und ihre Funktionalitäten im Detail zu betrachten.

Bisher nicht vergleichend betrachtet wurden die Paketgrößen der übertragenen Daten für die verschiedenen Technologien. Diese Metrik kann für einige Anwendungsfälle allerdings auch sehr relevant sein um beispielsweise Anforderungen in der Bandbreite für die verschiedenen Technologien zu vergleichen.

Da in der Implementierung des Testaufbaus für diese Arbeit festgestellt wurde, dass einzelne Optimierungen durch Einstellungen der Technologien große Auswirkungen auf deren Leistung haben können, könnte eine weitere Erforschung der Einstellungs- und Optimierungsmöglichkeiten verbesserte Ergebnisse liefern (siehe beispielsweise die Quellen [25] oder [26]).

Der Testaufbau, der im Rahmen dieser Arbeit genutzt wurde um den Technologievergleich durchzuführen, basiert teilweise auf Vereinfachungen und Abstraktionen gegenüber den realen Bedingungen in der Relaxdays GmbH. Dadurch lassen sich zwar die Testergebnisse der einzelnen Technologien untereinander vergleichen, allerdings sind sie nicht mit realen Leistungsdaten vergleichbar. Diese Erkenntnis sollte zum Anlass genommen werden, um die Technologien mit den realen Bedingungen und Datensätzen der Relaxdays GmbH weiterführend zu testen.

A. Anhang

```
{
    "Id" : 1,
    "Name" : "Chair",
    "Sku" : 123456,
    "Description" : "You can sit on it.",
    "PriceId" : 1,
    "IsActive" : true,
    "Color" : "black",
    "Width" : 0.3,
    "Height" : 1.5,
    "Depth" : 0.3,
    "Weight" : 2.5,
    "Material" : "Wood"
}
```

Listing 4: JSON-Darstellung des für den Serialisierungs- und Deserialisierungstest genutzten Objektes

Prozessor:	AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz
Anzahl CPU Kerne:	8
Anzahl Threads:	16
Installierter RAM:	16,0 GB
Systemtyp:	64-Bit-Betriebssystem, x64-basierter Prozessor

Tabelle A.1.: Notebook Eigenschaften

Methode	Durchschnitt (ns)	Fehler (ns)	Standardabweichung (ns)
SerializeJsonSTJ	752.9	3.67	3.43
SerializeJsonNSJ	1,689.8	15.32	14.33
SerializeProtobuf	518.0	1.64	1.53
DeserializeJsonSTJ	1,366.5	7.99	7.47
DeserializeJsonNSJ	3,307.2	24.80	23.19
DeserializeProtobuf	638.3	6.35	5.63

Tabelle A.2.: Leistungsvergleich zwischen Serialisierungs- und Deserialisierungsmethoden

NumberOfArticles	REST (ms)	gRPC (ms)	GraphQL (ms)
1	1.702	1.531	1.084
10	1.784	1.602	1.185
100	2.773	2.290	2.165
1000	11.752	8.420	11.177
10000	91.895	65.855	100.656

Tabelle A.3.: Test 1 Ergebnisse

NumberOfArticles	REST (ms)	gRPC (ms)	GraphQL (ms)
1	3.575	3.380	1.058
10	3.622	3.420	1.107
100	3.946	3.556	1.557
1000	7.512	4.799	6.147
10000	34.602	14.807	52.337

Tabelle A.4.: Test 2 Ergebnisse

NumberOfArticles	REST (ms)	gRPC (ms)
1	3.363	3.052
10	18.243	16.754
100	167.197	155.453
1000	1,725.205	1,542.741
10000	16,987.004	15,436.894

Tabelle A.5.: Test 3 Ergebnisse

NoA	REST (ms)	gRPC (ms)	GraphQL o. DL (ms)	GraphQL m. DL (ms)
1	1.996	2.042	1.919	2.322
10	2.191	2.285	9.371	6.465
100	3.793	3.450	84.642	8.270
1000	18.098	15.707	820.182	98.144
10000	159.040	132.789	8,402.723	207.746

Tabelle A.6.: Test 4 Ergebnisse

NumberOfArticles	REST (ms)	gRPC (ms)	GraphQL (ms)
1	3.120	3.001	3.071
10	32.531	32.059	35.698
100	278.244	285.412	300.028
1000	3,014.062	4,573.666	3,225.967
10000	66,313.544	226,036.055	65,400.840

Tabelle A.7.: Test 5 Ergebnisse

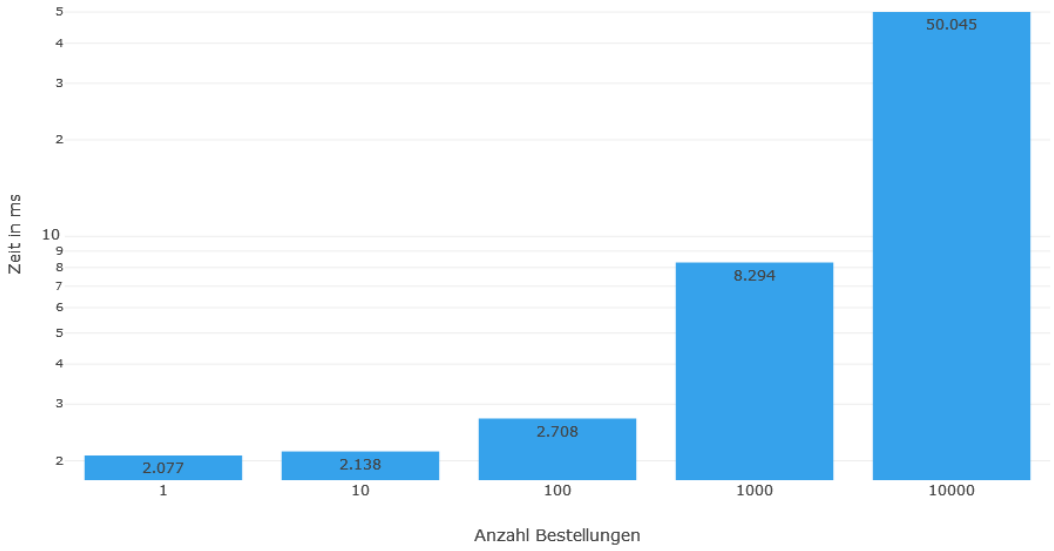

Abb. A.1.: Ergänzung Test 5: Abfragen von Bestellungen mit Bestellpositionen über direkte Datenbankanbindung

Literaturverzeichnis

[1] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham, 2017. ISBN 978-3-319-67425-4. doi: 10.1007/978-3-319-67425-4_12. URL https://doi.org/10.1007/978-3-319-67425-4_12.

[2] Joydip Kanjilal. Top 10 Microservices Design Principles | Developer.com. *Developer*, February 2023. URL https://www.developer.com/design/microservices-design-principles.

[3] Johannes Thönes. Microservices. *IEEE Software*, 32(1):116–116, 2015. doi: 10.1109/MS.2015.11.

[4] Christof Fetzer. Building critical applications using microservices. *IEEE Security & Privacy*, 14(6):86–89, 2016. doi: 10.1109/MSP.2016.129.

[5] Mohamed El Kholy and Ahmed El Fatatry. Framework for interaction between databases and microservice architecture. *IT Professional*, 21(5):57–63, 2019. doi: 10.1109/MITP.2018.2889268.

[6] Bogdan Nedelcu. Behind GraphQL: A first-principles approach. *Medium*, December 2021. URL https://medium.com/@bogdanned/system-design-with-graphql-601655a0c21f.

[7] About gRPC, May 2023. URL https://grpc.io/about. [Online; accessed 5. May 2023].

[8] Protocol Buffers Overview, April 2023. URL https://protobuf.dev/overview. [Online; accessed 5. May 2023].

[9] GraphQL A query language for your API, May 2023. URL https://graphql.org. [Online; accessed 5. May 2023].

[10] D. Roth, R. Anderson, and S. Luttin. Uebersicht ueber ASP.NET Core, March 2023. URL https://learn.microsoft.com/de-de/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-7.0. [Online; accessed 5. May 2023].

[11] Arthur Vickers. Überblick über Entity Framework Core – EF Core, July 2023. URL https://learn.microsoft.com/de-de/ef/core. [Online; accessed 12. Jul. 2023].

[12] What is a Container? | Docker, February 2023. URL https://www.docker.com/resources/what-container. [Online; accessed 5. May 2023].

[13] About MariaDB Server - MariaDB.org, May 2023. URL https://mariadb.org/about. [Online; accessed 5. May 2023].

[14] BenchmarkDotNet-Library. BenchmarkDotNet, June 2023. URL https://github.com/dotnet/BenchmarkDotNet. [Online; accessed 13. Jun. 2023].

[15] J. D. Lien. What is the N+1 query problem and how to solve it?, January 2023. URL https://planetscale.com/blog/what-is-n+1-query-problem-and-how-to-solve-it. [Online; accessed 15. Jul. 2023].

[16] GraphQL .NET, June 2023. URL https://graphql-dotnet.github.io/docs/guides/dataloader. [Online; accessed 19. Jun. 2023].

[17] Marinko Spasojevic. Advanced GraphQL Queries, Error Handling, Data Loader - Code Maze. *Code Maze*, January 2022. URL https://code-maze.com/advanced-graphql-queries.

[18] Marlon Dedakis. System.Text.Json vs Newtonsoft.Json, July 2023. URL https://www.linkedin.com/pulse/systemtextjson-vs-newtonsoftjson-marlon-dedakis. [Online; accessed 14. Jul. 2023].

[19] Genevieve Warren. Migrate from Newtonsoft.Json to System.Text.Json - .NET, July 2023. URL https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/migrate-from-newtonsoft?pivots=dotnet-7-0. [Online; accessed 14. Jul. 2023].

[20] protobuf net. protobuf-net, June 2023. URL https://github.com/protobuf-net/protobuf-net. [Online; accessed 9. Jun. 2023].

[21] GraphQL 7.4.1, June 2023. URL https://www.nuget.org/packages/GraphQL. [Online; accessed 9. Jun. 2023].

[22] GraphQL Playground, July 2023. URL https://www.apollographql.com/docs/apollo-server/v2/testing/graphql-playground. [Online; accessed 13. Jul. 2023].

[23] Samuel Imolorhe. Getting Started. *Altair GraphQL Client*, July 2023. URL https://altairgraphql.dev/docs.

[24] Giorgi Dalakishvili. GraphQLinq, June 2023. URL https://github.com/Giorgi/GraphQLinq. [Online; accessed 13. Jun. 2023].

[25] Wojciech Trocki. GraphQL performance explained - Wojciech Trocki - Medium. *Medium*, December 2021. URL https://medium.com/@wtr/graphql-performance-explained-cb4b43412fb4.

[26] Debendr Dash. Tips And Tricks To Improve WEB API Performance, July 2023. URL https://www.c-sharpcorner.com/article/Tips-And-Tricks-To-Improve-WEB-API-Performance. [Online; accessed 15. Jul. 2023].

[27] Paul de Vrieze and Lai Xu. Resilience analysis of service-oriented collaboration process management systems. *SOCA*, 12(1):25–39, March 2018. ISSN 1863-2394. doi: 10.1007/s11761-018-0233-5.

[28] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai Josuttis. Microservices in Practice, Part 1: Reality Check and Service Design. *IEEE Software*, 34 (1):91–98, January 2017. ISSN 1937-4194. doi: 10.1109/MS.2017.24.

[29] D. Taibi, V. Lenarduzzi, and Claus Pahl. *Architectural Patterns for Microservices: A Systematic Mapping Study*. SCITEPRESS, Portugal, 2018. ISBN 978-989-758295-0. URL https://bia.unibz.it/esploro/outputs/conferenceProceeding/Architectural-Patterns-for-Microservices-A-Systematic-Mapping-Study/991005773017601241.