Verteilte Systeme: Chat Applikation unter Einsatz einer Message-Oriented-Middleware

Anja Wolf, Marvin Staudt, Andreas Westhoff und Johannes Knippel

Abstract—Aufbau einer Chat Applikation unter Einsatz einer Message Oriented Middleware und deren Anwendung.

Index Terms—Verteilte Systeme, JMS, Wildfly, MOM, JavaEE, JPA, Kafka, Docker, Angular, GraphQL, ReST

1 Einleitung

Chat-Applikationen basieren in den meisten Anwendungsfällen auf speziellen Middleware Technologien, die auf verteilten Systemen aufgebaut und implementiert sind. Diese Studienarbeit beschreibt die Chat-Anwendung, die im Rahmen des Praktikums im Fach *Verteilte Systeme* im Wintersemester 2018/2019 bearbeitet wurde.

Ziel der Studienarbeit ist, es innovative Lösungsansätze und Konzepte im Umfeld der verteilten Systeme zu erforschen, zu erproben und diese zu bewerten.

Im Fokus der Arbeit steht eine einfache Chat-Anwendung, die dazu dient, die Entwicklung und Funktionsweise von verteilten Systemen auf Basis von Message Oriented Middleware (MOM), näher zu vertiefen.

Im Folgenden wird die Konzeption der Anwendung sowie die benötigte Infrastruktur erläutert. Anschließend wird die Umsetzung und die damit verbundene Konfiguration der einzelnen Komponenten der Anwendung näher beschrieben. Abschließend folgt ein Fazit.

22. Dezember, 2018

2 Konzeption

Zu entwickeln galt es ein verteiltes System auf Basis von Middleware Technologien, welches im Anschluss durch bestimmte Benchmarkzyklen analysiert werden soll. Genauer handelt es sich um spezielle MOM, die eine klare und strukturierte Methode der Kommunikation zwischen disparaten Software Entitäten zur Verfügung stellt. [1] Im Fokus der Entwicklung steht die Installation und die Inbetriebnahme der verwendeten Technologien, die Nutzung von Representational State Transfer (ReST), Message Queues, Topics, GraphQL, Apache Kafka, Transaktionsverarbeitung sowie die Leistungsbewertung der entstandenen Applikation.

Die darzustellende Anwendung beinhaltet einen einfachen Chat-Prozess. Dieser setzt sich aus zwei Teilen zusammen. Einem Chat-Client, der es einem Benutzer ermöglicht über eine graphische Nutzer-Oberfläche mit anderen Chat-Beteiligten zu kommunizieren und einem Chat-Server, der sich um die Verwaltung der Chat-Nutzer und deren Nachrichten kümmert.

Möchte sich ein Benutzer am Server registrieren, verwendet er eine dafür implementierte Client-Anwendung. Ist er registriert beziehungsweise am System angemeldet, können Nachrichten von anderen bereits registrierten Benutzern empfangen und an diese verschickt werden. Der Server bestätigt jede Nachrichtenanfrage eines registrierten Benutzers (Clients). Ist ein Benutzer nicht angemeldet, so werden die Nachrichten auch nicht verarbeitet. Zur Auswertung und Datenhaltung werden Informationen zu den Nachrichten vom Server in dafür vorgesehene Datenbanken gespeichert. Eingehende Nachrichten werden in Verbindung mit den Benutzernamen und dessen genutzten Threads in einer hier benannten Trace-Datenbank gespeichert. In einer zweiten separaten Count-Datenbank wird ein Zähler mit der Anzahl der Nachrichten pro Nutzername geführt. Aufgrund der Softwarearchitektur erstrecken sich manche Aktionen über verteilte Komponenten, die es über Transaktionen abzusichern gilt. Sollte eine Komponente, wie beispielsweise eine der beiden Datenbanken, ausfallen, so müssen alle Schritte der jeweiligen Transaktion zurückrückrollbar gehalten werden.

Die in den Datenbanken gespeicherten Informationen kön-

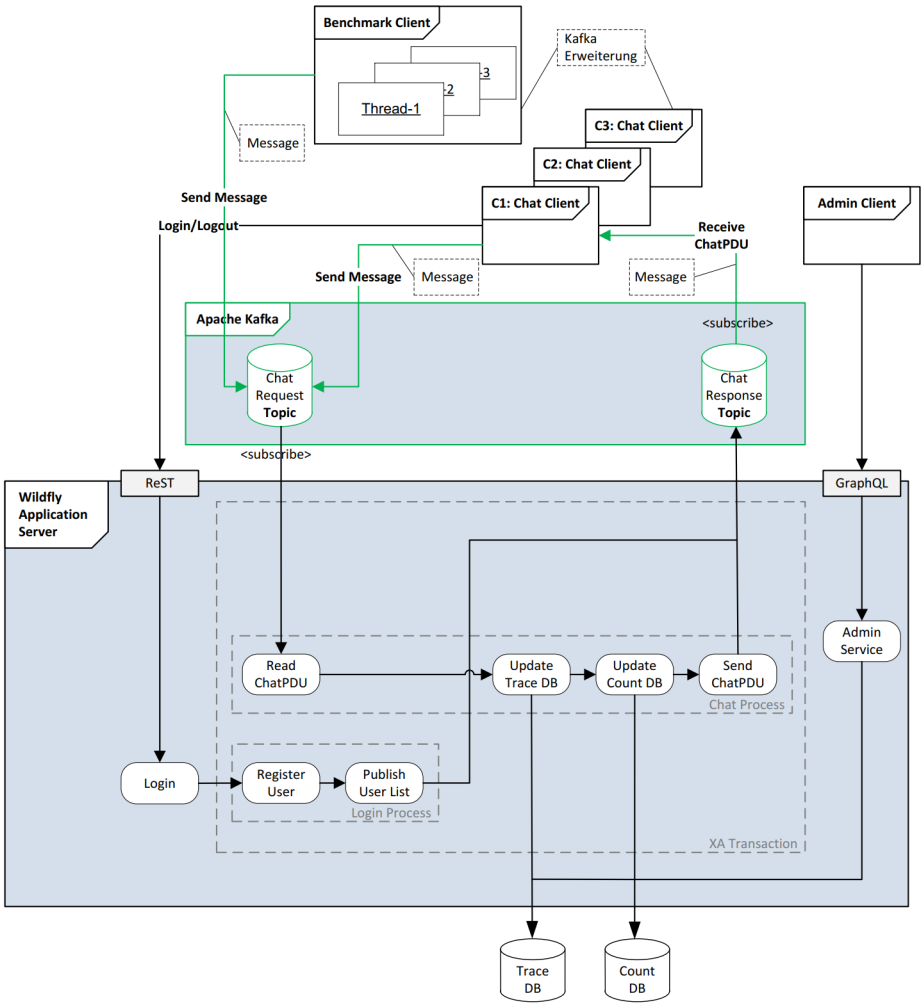

Fig. 1. Grobe Skizze der Architektur der Chat-Anwendung mit Erweiterung um Apache Kafka

nen über eine implementierte GraphQL[1] Schnittstelle ausgelesen werden.

Für Administratoren des Systems wurde eine Client Anwendung basierend auf dem Webapplikations-Framework Angular[2] entwickelt. Diese nutzt die GraphQL Schnittstelle, um Daten über die Chat-Anwendungen auszulesen und anzuzeigen.

Zur Bestimmung von Performanceergebnissen des entstehenden Systems wird ein Benchmarking-Client genutzt, der eine wählbare Anzahl an Anwendern simuliert und Nachrichten an den Server schickt. Dabei werden verschiedene Kennzahlen, wie beispielsweise die Dauer eines Roundtrips der Nachricht, erhoben.

1. http://graphql.org/
2. https://angular.io/

3 Infrastruktur

3.1 Github

Für ein gemeinsames Arbeiten am Quellcode wurde die Versionsverwaltungsplatform Github verwendet. Es wurden zwei Repositories für Server und Client angelegt. Für das Aufsetzten des Servers wurde ein neues Projekt ersetllt, wohingegen für die Arbeit auf Client-Seite das bereits bestehende Projekt chatApplication erweitert wurde.

3.2 Application Server Wildfly 13

Als Application Server wurde Wilfdly 13.0.0.Final genutzt.
....

3.3 Java Message Service (JMS)

..

3.4 Apache Kafka

..

3.5 Maria DB

Datenbanken im Chatprozess um Nachrichten zu speichern und protokollieren. MariaDB verwendet. Docker ...

3.6 Representational State Transfer (ReST)

...

3.7 GraphQL

....

4 UMSETZUNG

Einleitung

4.1 Aufbau Server

4.2 Aufbau Client

4.3 JMS Konfiguration

4.4 Kafka Konfiguration

4.5 Login über ReST

4.6 Chat Prozess

4.6.1 JMS Message

4.6.2 Kafka

4.6.3 Persistieren der Informationen in den Datenbanken Trace und Count

Während des *Chat Process* werden mit Hilfe der beiden Datenbankprozesse **Count** und **Trace** die benötigten Informationen zu den Nachrichten der *chatuser* in die jeweilige Datenbank gespeichert und protokolliert. Hierfür werden zwei Datenbanksysteme benötigt, welche unabhängig voneinander ausgeführt werden können. Um dies zu gewährleisten wurde jeweils eine Datenbank in einem Docker-Container ausgeführt. Zum einen persistiert die Datenbank **Trace** alle gewünschten Informationen zu einer Chatnachricht. Die **Count** Datenbank dient zum ermitteln der Anzahl der versendeten Nachrichten eines *chatusers*. Dementsprechend wird sobald ein *chatuser* die erste Nachricht versendet ein Eintrag erstellt. Bei jeder weiteren Nachricht wird dieser automatisch mit einem Zähler inkrementiert.

Die Count-Datenbankinstanz erzeugt zur Laufzeit eine Tabelle mit dem Namen *countdata*. Die Struktur besteht dabei aus den folgenden fünf Attributen: *ID, username, clientthread, serverthread und message*. (Vgl. Tabelle 1)

Die Trace-Datenbankinstanz erzeugt zur Laufzeit eine Tabelle mit dem Namen *trace*. Die Struktur besteht dabei aus den folgenden drei Attributen: *ID, username und counting*. (Vgl. Tabelle 2)

TABLE 1
Struktur der Count-Tabelle.

Attribut	Funktion
`ID`	Eindeutige ID
`username`	Nutzername
`counting`	Anzahl der gesendeten Nachrichten

TABLE 2
Struktur der Trace-Tabelle.

Attribut	Funktion
`ID`	Eindeutige ID
`username`	Nutzername
`clientthread`	Identifikation des clients
`serverthread`	Identifikation des Servers
`message`	versendete Nachricht

Um im ersten Schritt einen Zugriff von wildfly auf die in den beiden Docker-Containern ausgeführten Datenbanken zu erhalten muss die Konfigurationsdatei des Applikations-Servers *standalone.xml* um die beiden Datenbankinstanzen erweitert werden. Hierbei werden zwei *XA-Datasources* eingetragen. Diese Art von Datenbanken erlaubt es mehrere *Datasources* gleichzeitig innerhalb einer Transaktion verwenden zu können. Folgende Informationen wurden hierbei hinterlegt: *IP-Adresse, Port, User* und *Passwort*. Wichtig ist hierbei die Verwendung der richtigen Syntax der jeweiligen wildfly-Version, da andernfalls die Verbindung zwischen Applikationsserver und Datenbank nicht erfolgreich hergestllt werden kann.

Um nun im Folgenden die Daten in die jeweiligen Tabellen persisitieren zu können wurde die *Java Persistence API* **(JPA)** verwendet. Dies vereinfacht den Prozess der Zuordnung und Auslieferung der Objekte zu den richtigen Einträgen in den Datenbankinstanzen. Dabei bilden die zwei Persistence Entities der Klassen **Trace.java** und **Count.java** aus dem Package *databases* die Tabellen *countdata* und *trace* ab. Die

Objekte der Klassen werden automatisch erkannt und die Struktur der Tabellen festgelegt.

In dieser Studienarbeit wurde der Ansatz der *Enterprise JavaBean* **(EJB)** mit einer container-verwalteten Transaktion verwendet. Dementsprechend wurden beide Datenbanken als *persistence-unit* in dem *persistence.xml*-file angegeben. Weiterhin wurden hier die IP-Adressen, welche *Docker* vergeben hat, zusammen mit den jeweiligen Ports hinterlegt. Zusätzlich wurde der *User* und das *Passwort* angegeben, um einen Zugang zur Datenbank zu ermöglichen. Wichtig hierbei ist auch die Angabe der Klasse, welche die *Entity* auf eine die jeweilige Datenbanktabelle wiederspiegeln soll.

Mit Hilfe von Hibernate, welches ebenfalls in der *persisstence.xml* mit den entsprechenden Eigenschaften definiert wurde, wird die Schnittstelle für den Zugriff auf die Datenbanken festgelegt. Dadurch wird es möglich, Datenbankverbindungen aufzubauen und diese entsprechend zu verwalten. Dabei werden *Structured Query Language* **(SQL)** Abfragen weitergeleitet und nach Ausführung dem Applikationsserver wieder zur Verfügung gestellt.

In dieser Studienarbeit wurden zwei **JPA**-Implementierungen eingebunden. Dementsprechend werden für beide zur Laufzeit jeweils automatisch ein *Entity-Manager* erstellt. Dieser dient als Komponente zur erfolgreichen Persistierung von Daten in die Datenbank zur Verfügung. Daraus ergeben sich Methoden, wie z. B. das Speichern, das Finden oder Bearbeiten bereits gespeicherter Daten oder das Löschen eines Datensatzes. Dies wird automatisch unter Verwendung der richtigen Annotationen zur Laufzzeit durchgeführt.

Dementsprechend wird es ermöglicht eine einheitliche Schnittstelle für den *ChatProcess* zur Verfügung zu stellen. Mit den Methoden *updateCount(Count count)* und *create(Trace trace)*, welche der *ChatProcess* aufruft, werden Daten in die beiden Datenbankinstanzen gespeichert bzw. aktualisiert.

Auch werden hierbei die Methoden zum Abfragen und zum Zurücksetzen der Datenbanken für den Admin Client (vgl. Kapitel 4.10)bereitgestellt. Die beiden Methoden *List<Count> findAll()* und *List<Trace> findAll()* geben jeweils den gesamten Inhalt der in den Datenbanken gespeicherten Informationen aus. Die Methode *clear()* ermöglicht das separate Löschen aller Daten aus den beiden Datenbanken.

Im Folgenden Kapitel 4.7 werden die Transaktionsverwaltung und der Zusammenhang des Persistieren dieser Informationen in die beiden Datenbankinstanzen genauer erläutert.

4.7 Transaktionsverwaltung

Um eine übergreifende XA-Transaktion unter Einbeziehung der Zugriffe auf die JMS-Queue, das Topic sowie auf die Datenbanken zu realisieren übernimmt das *Transactions subsystem* des Wildfly Application Servers die komplette Transaktionsverwaltung. Die beiden MariaDB-Container, welche im Docker ausgeführt werden, unterstützen den Prozess der XA-Transaktionen. Auch wurden die beiden Datenbanken als *XA-Datasources* in der Konfigurationsdatei *standalone.xml* des Application Servers definiert.

Dementsprechend wird ein Transaktionsmanager mit dem *TransactionsManagementType.CONTAINER* vorgeschaltet. Das Starten, Zurückrollen oder Abschließen einer Transaktion wird direkt unter dessen Verwaltung gestellt. Hierfür sind die Annotationen *@Transactional* vor die jeweiligen Methoden bzw. Klassen gesetzt.

Ein aktives Eingreifen ist aus diesem Grund nicht notwendig. Wird eine Methode ohne Komplikationen durchlaufen, schließt der Transaktionsmanager die Transaktion automatisch mit *COMMIT* ab. Wird jedoch ein Fehler geworfen, werden alle darin enthaltenen Methoden mit einem *ROLLBACK* zurückgerollt.

Beispeilsweise wird beim unerwarteten Abbruch während des Schreibvorgangs von Daten in einer der beiden Datenbanken eine Exception nötig, die automatisch alle Änderungen mit einem *ROLLBACK* rückgängig macht. Werden zum Beispiel erfolgreich Daten in die erste Datenbank geschrieben und ein unerwarteter Fehler tritt beim Einfügen der Daten in die zweite Datenbank auf, muss mittels einer Exception durch den Transaktionsmanager ein Zurückrollen der beiden Schreibvorgänge durchgeführt werden. Im Anschluss sollte ein erneuter Zustellversuch dieser Nachricht die Folge sein.

Auch ein aktives Auslösen eines Rollbacks im *ChatProcess* ist mit der Methode *setRollbackOnly()* des *MessageDrivenContext* der Message Driven Bean im Falle einer *JMSException* möglich.

Im Falle eines Sendeversuchs einer *Message*, welche von einem nicht eingeloggten *user* versendet wird, ist diese Exception allerdings nicht notwendig. Dies ist darauf zurückzuführen, dass in der Klasse *ChatProcess* im Vorfeld überpüft wird, ob der *user* eingeolgt ist. Ist dieser nicht eingeloggt, wird kein Sendeprozess durchgeführt und somit findet kein Schreiben in die Datenbank statt.

4.8 Benchmark-Client

Der Benchmarking-Client dient zur Simulation eines Chat-Szenarios. Dabei kann hinsichtlich der Performance beim Versand von Chat-Nachrichten getestet werden. Diese wird anhand der Round Trip Time (RTT) einer Nachricht gemessen. Diese misst die Zeit für einen Chat-Request mit vollständiger Bearbeitung (vgl. Abbildung 2).

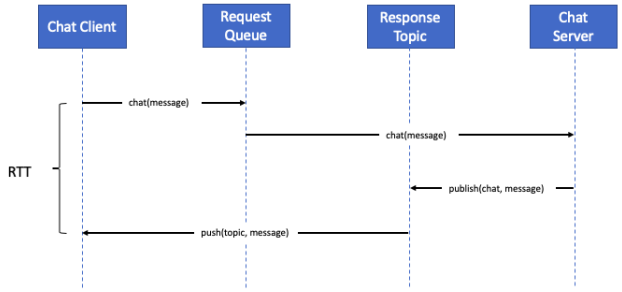

Fig. 2. Definition der Round Trip Time

Grundlage für die Performance-Tests ist das bereits vorhandenen Java-basiertes Benchmarking-Framework. Dieses wurde um eine JMS sowie eine Apache Kafka Anbindung erweitert um Leistungstest für das Versenden von Nachrichten über die beiden Message Broker durchzuführen.

Damit dies möglich ist, mussten mehrere Änderungen im Quellcode vorgenommen werden. Zunächst wurden zwei zusätzliche ImplementationTypes, *JMSImplementation* und *KafkaImplementation*, definiert. Desweiteren wurden die Klasse *BenchmarkingClientFactory* um JMS sowie Kafka spezifische Parameter erweitert.

Für die JMS Implementierung wurden außerdem die Klassen *JmsChatClient*, *JmsSimpleMessageListenerThread* sowie *JmsBenchmarkingClientImpl* neu implementiert. Für die Kafka Implementierung wurden zusätzlich die Klassen *KafkaChatClient*, *KafkaReceiver* und *KafkaBenchmarkingClientImpl* implementiert.

Die Anzahl der Clients, sowie die Menge an Nachrichten bzw. deren Byte-Größe kann verändert werden. Diese Einstellungen können für TCP in der Benutzeroberfläche, für JMS bzw. Kafka in der Klasse *BenchmarkUserInterfaceParameters* angepasst werden.

Vor der Durchführung des Leistungstest wurde die Anzahl der Nachrichten konstant auf 100 je Client mit einer Größe von je 50 Byte festgelegt. Die Anzahl der Clients wurde in 10er Schritten verändert. Der erste Test wurde mit 10 Clients durchgeführt. Danach wurde bis zu einem Maximalwert von 50 Clients erhöht. Jeder der Clients loggt sich dabei ein, versendet die definierte Anzhal an Nachrichten, empfängt Nachrichten der anderen Clients und loggt sich wieder aus.

Das Benchmarking wurde in zwei verschiedenen Ausprägungen durchgeführt. Sowohl JMS als auch Kafka wurden hinsichtlich ihrer Performance getestet. Beide male wurde der Test mit der Klasse *BenchmarkingUserInterfaceSimulation* gestartet, angepasst an JMS bzw. Kafka. Die Ergebnisse des Benchmarks werden in der Konsole ausgegeben. Eine Anbindung an das bereits vorhandene User Interface hat nicht stattgefunden. Die Ergebnisse der beiden Testdurchläufe können Tabelle 3 entnommen werden.

TABLE 3
Ergebnisse (RTT's) der beiden Benchmark-Läufe

Anzahl Clients	JMS [in ms]	Kafka [in ms]
10	168	36.024
20	189	49.759
30	222	67.164
40	227	90.335
50	322	10.3483

Im ersten Durchlauf des Leistungstest wurden die durchschnittliche RTT von JMS gemessen. Bei einer linear ansteigenden Anzahl an Clients steigt auch diese linear an (vgl. Abbildung 3).

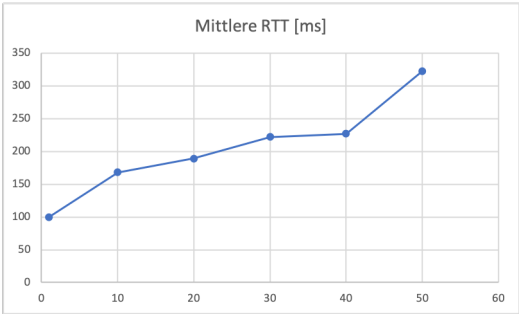

Fig. 3. Benchmark JMS

Beim zweiten Durchlauf des Benchmarks wurde die durchschnittliche RTT von Kafka gemessen. Auch hier steigt diese linear mit einer lienar ansteigenden Anzahl an Clients (vgl. Abbildung 4).

Vergleicht man beide Testdurchläufe wird deutlich, dass sich durch die Umstellung von JMS auf Kafka die RTT des Chat Prozesses erhöht. Wärhend bei JMS Nachrichten selbst bei einer Anzahl an 50 Clients Nachrichten noch unter

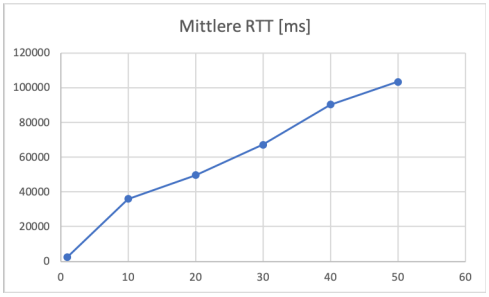

Fig. 4. Benchmark Kafka

einer Sekunde verschickt werden, sind die RTT Zeiten bei Kafka selbst bei einer geringen Anzahl an Clients bereits im höheren Sekunden bereich.
Aus den Testergebnissen lässt sich Folgendes schließen: JMS verschickt Nachrichten um ein vielfaches schneller als Kafka. Desweiteren steigt mit der Anzahl der angemeldeten Chat Clients und zu versendenden Nachrichten die RTT einer einzelnen Chat Nachricht.

4.9 GraphQL

Im Folgenden Abschnitt wird die GraphQL-Schnittstelle und die Vorgehensweise der Implementierung in dieser Studienarbeit dargestellt. Weiterhin werden mittels eines Vergleichs die Vor- und Nachteile von *ReST* und *GraphQL* diskutiert. Abschließend werden die aufgetretenen Herausforderungen in diesem Projekt angeführt.

4.9.1 Was ist GraphQL?

GraphQL bietet mit Hilfe eines neuen API Standards eine effiziente und leistungsfähige Alternative zu *ReST* an. *GraphQL* ist eine Abfragesprache *(Query Language)*, welche über die API angesprochen werden kann. Dabei ermöglicht es das Ausführen von Abfragen mit Hilfe eines Typsystems, welches im Vorfeld definiert wurde, zur Laufzeit des Servers. Zusätzlich stellt *GraphQL* Spezifikationen und Werkezeuge, welche lediglich über **einen** HTTP-Endpunkt angesprochen werden, zur Verfügung. [2]

Die im Folgenden beschriebenen zwei Schritte dienen der Erklärung des Grundkonzepts, welches hinter *GraphQL* steht, am Beispiel der Implementierung in diesem Projekt.

1. Da *GraphQL* ein eigenes Typsystem hat, welches das Schema der API definiert, muss dieses im ersten Schritt festgelegt werden. Hierfür wird die *Schema Definition Language* **(SDL)** verwendet. Hier ein Beispiel, welches im Rahmen dieser Studienarbeit angewendet wurde, um den einfachen Typ *Trace* zu definieren.

```
type query {
        allTrace: [TraceTO]
        allCount: [CountTO]
}

type TraceTO {
        id: ID!
        username: String
        clientthread: String
        message: String
        serverthread: String
}
```

   Ein *GraphQL-Service* wird erstellt, indem Typen und Felder für diesen Typen definiert werden, um im Anschluss für jedes Feld bzw. für jeden Typen Funktionen festlegen zu können.

2. Der *GraphQL-Service* kann nun mit der folgenden Query vom Client ausgeführt werden. [3]

```
query = {
        allTrace {
                id
                username
                clientthread
                message
                serverthread
        }
}
```

   Dabei kann der Client direkt festlegen welche Attribute ausgegeben werden sollen bzw. entsprechende auch weglassen. Es müssen dementsprechend nicht alle Attribute, welche im zuvor festgelegten Schema definiert wurden, auch wieder abgerufen werden. Die hinterlegte Methode zu dieser Abfrage wird in dem Mapper-Repository der Klasse *TraceMRepository* mit der Methode *public List<TraceMapper> findAll()* definiert. Diese Methode wird in der Klasse *Query* mit der folgenden Methode aufgerufen und ausgeführt. [3]

```
public List<TraceMapper> findAll() {
  return TraceMRepository.findAll();
}
```

Der Server gibt die angefragten Daten im Anschluss als JSON-Objekt wie folgt zurück.

```
{
 "data": {
  "allTrace": [
```

```
    {
     "id": "1",
     "username": "Marvin",
     "clientthread":
        "Thread[AWT-EventQueue-0,6,main]",
     "message": "Hallo, wie geht's?",
     "serverthread": "JMS"
    }
   ]
 }
}
```

4.9.2 Die implementierte GraphQL-Schnittstelle

Die im Zuge dieser Studienarbeit implementierte *GraphQL-Schnittstelle* ist auf die Bibliothek *com.graphql-java* in der Version *3.0.0* aufgebaut. Die Bibliothek ist in der *pom.xml*-Datei des Maven-Projekts hinterlegt. Um einen GraphQL Endpunkt mit der folgenden *URL* erreichbar zu machen wurde eine Klasse mit dem Namen *GraphQLEndpoint* im Package *graphql* erstellt. Diese erweitert die Klasse *SimpleGraphQLServlet*.

```
localhost:8080/server-1.0-SNAPSHOT/graphql
```

Diese Klasse beschreibt von welchem Schema dieser *GraphQL*-Service definiert wird. Das Schema hierzu muss im folgenden Verzeichnis hinterlegt werden.

```
chat/server/src/main/resources/schema.graphqls
```

Das Schema wird einführend wie folgt definiert:

```
schema {
        query: Query
        mutation: Mutation
}
```

In der Klasse *Query* werden die Methoden definiert, die alle Daten aus den beiden Datenbanken *Count* und *Trace* ausgeben. In der Klasse *Mutation* werden die Methoden definiert, welche der Admin-Client im Anschluss ausführen kann, um die beiden Datenbanken zu entleeren. *Query* und *Mutation* müssen beide jeweils im Schema *schema.graphqls* als Typen definiert werden. Im Anschluss müssen die jeweiligen Methoden in den beiden Klassen definiert werden.
In diesen Methoden wurde die gesamte Logik der Abfrage bzw. das Manipulieren der Daten hinterlegt. Die beiden *GET-Methoden*, die alle Daten aus den Datenbanken ausgeben werden als *GET-Reqeust* auf den Server abgesetzt. Die beiden Methoden, welche zum Löschen der Datenbanken aufgerufen werden können, wurden in der Klasse *Mutation* definiert. Dabei muss ein *POST-Request* abgesetzt werden. Zusätzlich wird ein **JSON-Body** mit dem folgenden Inhalt übergeben.

```
{"query":"mutation{clearCount}"}
```

Der abgebildete *JSON-Body* entleert die *Count-Datenbank*. Als Rückmeldung wird ein *Boolean-Wert* vom Applikationsserver zurückgegeben. War das Zurücksetzen der Datenbank erfolgreich wird *true* ausgeben. Das Vorgehen für die *Trace-Datenbank* ist identisch.

Herausforderungen im Laufe der Implementierung:

- Sehr wichtig bei der Implementierung einer *GraphQL-Schnittstelle* ist es Typen, Felder und Methoden strikt so zu benennen, wie es in der Datei *schema.graphqls* definiert wurde. Andernfalls werden diese nicht erkannt und es wird eine Exception beim Abfragen geworfen.
- Fehlermeldungen werden bei *GraphQL* nicht wie bei *ReST* mittels HTTP-Statuscodes ausgegeben. Tritt eine Exception auf, wird diese im Body zurückgegeben.
- Die bereits integrierten *JPA-Klassen*, welche zur Persistierung der Daten in die Datenbanken benötigt werden, konnten durch den GraphQL-Endpunkt in unserem Fall nicht mit der Annotation *@EJB* angesprochen werden. Daher mussten zusätzlich *Mapper-Klassen* für *Count* und*Trace* implementiert werden. Die beiden Mapper stellten dementsprechend die Schnittstelle zu den Datenbanken und *GraphQL*, um einen Datenaustausch zu ermöglichen.

4.9.3 GraphQL vs. ReST

ReST ist ein Architekturkonzept für netzwerkbasierte Software und gilt als Bindeglied, um eine Kommunikation zwischen Client und Server zu ermöglichen. Der Fokus liegt dabei darauf, eine *API* über längere Zeit "haltbar" zu machen, anstatt die Leistung hier zu optimieren. Ziel dieser Architektur ist es, eine vereinfachte Architektur der entkoppelten Schnittstelle zu ermöglichen und eine erhöhte Visibilität der Interaktionen zu erreichen. Darunter leidet selbstverständlich die Effizienz, da Informationen lediglich in einem standardisierten Format abgerufen werden können und nicht auf die speziellen Bedürfnisse angepasst werden können. [4]
GraphQL ist im Gegensatz dazu eine Abfragesprache, eine

Spezifikation und eine Sammlung von Tools, die über einen **einzigen** Endpunkt mittels *HTTP* ausgeführt werden können. Der große Fokus liegt hierbei auf der Optimierung von Leistung und Flexibilität.
Ein Grundprinzip von **ReST** beruht sich darauf, dass einheitliche Schnittstellen der Protokolle, wie z. B. *HTTP-Inhaltstypen* oder *Statuscodes*, verwendet werden. **GraphQL** definiert hingegen seine eigenen Konventionen. GraphQL gibt beispielsweise zu **jeder** Anfrage den Response Status-Code *200 OK* zurück. ReST verwendet meist den *HTTP Response Status*. Hier wird lediglich bei erfolgreicher Bearbeitung der Anfrage der Statuscode 200 zurückgegeben. Andernfalls kann zum Beispiel auch der Status-Code *400 Bade Reqeuest* oder *500 Internal Server Error* zurückgegeben werden. [4]
Ein weiterer Unterschied besteht darin, dass bei GraphQL zu Beginn ein Schema definiert werden muss, welches nach den jeweiligen Bedürfnissen direkt angepasst werden kann. Dieses Schema definiert die Art der Daten, welche bei clientseitiger Abfrage ausgegeben werden. Dementsprechend ist die gegenseitige Unabhängigkeit des Servers und des Clients sichergestellt.
Zusammenfassend kann nicht ohne eine Bedarfsanalyse des jeweiligen Projektes zu einer der beiden Technologien geraten werden. GraphQL kann einige Schwachstellen von ReST beheben, allerdings müssen die Anforderungen der Anwendung genau geprüft werden, um eine endgültige Aussage treffen zu können. Dementsprechend kann nicht pauschal gesagt werden, welche der beiden Technologien bevorzugt werden soll. [5]

4.10 Admin-Client

Um eine Web-Anbindung zu illustrieren sollte im Rahmen dieser Studienarbeit ein Admin-Client implementiert werden. Dabei war die Art der Implementierung nicht maßgeblich vorgegeben, lediglich dass ein aktuelles Webapplikations-Framework verwendet wird.

Für die Implementierung der Admin Anwendung wurde das auf TypeScript- und komponentenbasierte Framework Angular in der Version 7 verwendet. Die Anwendung dient der administrativen Verwaltung der Daten des Chats und zeigt Details zu den im Server gespeicherten Statistiken an. Sie ist direkt an die GraphQL-Schnittstelle des Chat-Servers angebunden und verwaltet Daten aus den Datenbanken Trace und Count sowie die Chat-User Daten. Zudem können über die Anwendung die verschiedenen Daten aus den Datenbanken gelöscht werden.

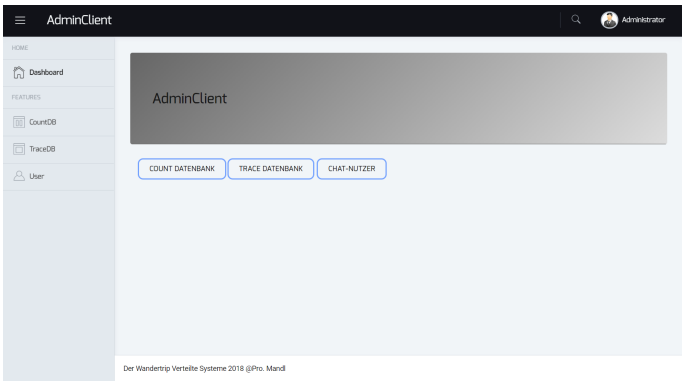

Fig. 5. Startseite/Dashboard des Admin-Clients.

Die Angular-Anwendung besteht aus vier verschiedenen Komponenten, die jeweils über einen eigenen Link, im Angularjargon auch Routen genannt, erreichbar sind. Für eine einfachere Bedienbarkeit wurde eine übersichtliche Headerzeile implementiert, die den Titel, ein Suchfeld und den Administrator als Profil anzeigt. Für eine einfache und schnelle Navigation zwischen den Komponenten wurde eine faltbare Sidebar implementiert, die die vier Komponenten übersichtlich mit Icons am rechten Bildschirmrand anzeigt.
Die Dashboard-Komponente stellt die Willkommensseite für die Anwendung dar (siehe Abbildung 5). Über die Willkommens- beziehungsweise Startseite gelangt man durch Button zu den anderen Komponenten der Anwendung. Die CountDB-Komponente ist unter der Route `/countdb` zu erreichen. Sie zeigt eine Auflistung aller Zähler-Einträge pro Chat-User an. Die TraceDB-Komponente wird über die Route `/tracedb` angesprochen. Diese veranschaulicht die Trace-Einträge für jede gesendete Nachricht in Tabellenform. Die Chat-User-Komponente erlaubt die Verwaltung von eingeloggten Nutzern. Sie ist über die Route `/user` erreichbar. Sollte ein ungültiger Link im Browser eingefügt werden, so gelangt man automatisch zurück zum Dashboard.

Im Folgenden werden die einzelnen Komponenten näher beschrieben sowie die Verbindung zur GraphQL Schnittstelle erläutert.

4.10.1 Count-Administration

Im Reiter der Count-Administration kann die Anwendung alle Einträge des Zählers für alle ausgeführten Chat-Requests eines Chat-Users anzeigen und bei Bedarf auch wieder löschen. Abbildung 6 zeigt die Komponente des

Admin-Clients, die für die Verwaltung der Zähler zuständig ist.

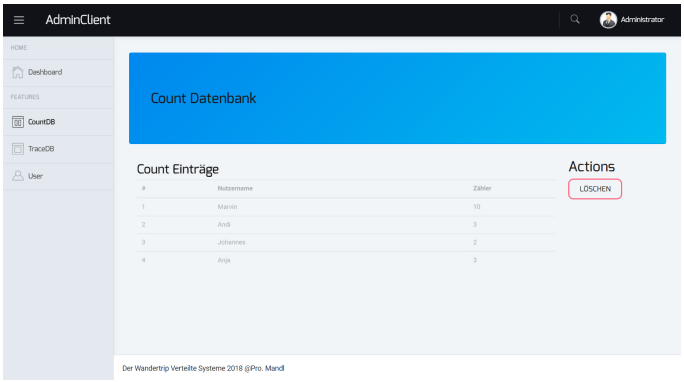

Fig. 6. Count-Administration des Admin-Clients.

Die Count-Komponente ruft die GraphQL Schnittstelle unter `/graphql?query={allCount{id username counting}}` mit einem HTTP GET Request und einem Observable Object `<CountResponse>` auf, um alle Zähler Einträge darstellen zu können. Hierbei wird das erhaltene JSON Objekt an die TypeScript Klasse der CountResponse geparsed, die dann im weiteren Schritt im HTML-Code zur Anzeige genutzt wird. Zum Löschen aller Einträge wird die GraphQL Schnittstelle mit einem POST Request unter `/graphql` aufgerufen. Der Body dieser Anfrage muss die Mutation `{"query": "mutation{clearCount}"}` im JSON Format enthalten, um eine erfolgreiche Löschung der Einträge zu erzielen.

4.10.2 Trace-Administration

Analog zur Count-Administration kann die Anwendung in der Trace-Verwaltung Einträge von versendeten Nachrichten eines Chat-Users anzeigen und diese bei Bedarf auch wieder löschen. In Abbildung 7 ist die Trace-Komponente mit einigen Einträgen zu sehen. Sie kann neben dem Inhalt einer Nachricht sowohl den serverThread als auch den clientThread anzeigen lassen, die eine genaue Analyse der versendeten Nachrichten erlaubt.

Die Trace-Administration ruft die GraphQL Schnittstelle unter `/graphql?query={allTrace{id username clientthread message serverthread}}` mit einem HTTP GET Aufruf und einem Observable Object `<TraceResponse>` auf. Wie auch in der Count-Komponente wird das erhaltene JSON Objekt an die TypeScript Klasse der TraceResponse geparsed, die dann wiederum für die HTML-seitige Anzeige genutzt wird. Zum Löschen aller Einträge wird die GraphQL Schnittstelle

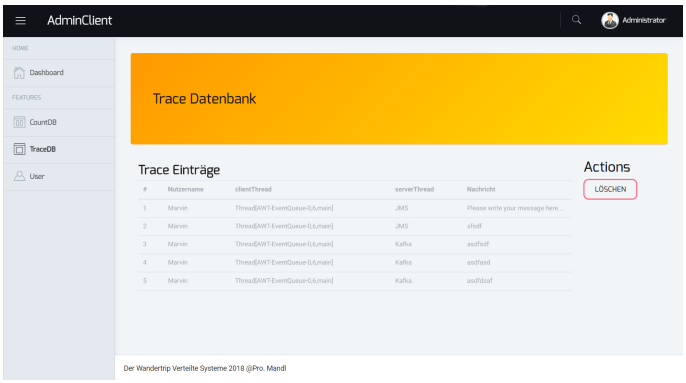

Fig. 7. Trace-Administration des Admin-Clients.

mit einem POST Request unter `/graphql` aufgerufen. Der Body dieser Anfrage muss die Mutation `{"query": "mutation{clearTrace}"}` im JSON Format enthalten, um eine erfolgreiche Löschung der Einträge zu erreichen.

4.10.3 Chat-User-Administration

Im Reiter der Chat-User-Administration kann die Anwendung alle angemeldeten User anzeigen und bei Bedarf können diese auch wieder abgemeldet und gelöscht werden. In Abbildung 8 ist die User-Komponente mit beispielhaften eingeloggten Benutzern zu sehen.

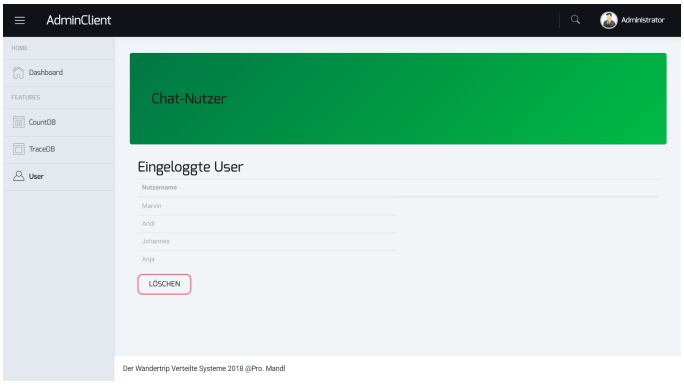

Fig. 8. Chat-User-Administration des Admin-Clients.

Die Chat-User werden über über die ReST Schnittstelle beim Laden des Reiters unter `/rest/users/currentusers` mit einem HTTP GET Request aufgerufen. Auch hier wird das erhaltene JSON Objekt mithilfe des Observable Objects `<ChatUserResponse>` an die korrespondierende TypeScript Klasse geparsed, die dann wiederum im HTML-Code zur Visualisierung verwendet wird. Zum Löschen aller Einträge wird die ReST Schnittstelle mit einem DELETE Request unter `/rest/users/logout/<user>` aufgerufen.

Hierbei wird über die chatUserList iteriert, die im vorherigen GET Request mit den eingeloggten Usern befüllt wurde.

5 Fazit

Schönes Fazit

- evtl Herausforderungen

References

[1] *Middleware for Communications* https://bit.ly/2A7Y7u0https://bit.ly/2A7Y7u0

[2] *Introduction to GraphQL* https://graphql.org/learn/

[3] *The Fullstack Tutorial for GraphQL* https://www.howtographql.com

[4] *GraphQL vs ReSt: Overview* https://philsturgeon.uk/api/2017/01/24/graphql-vs-rest-overview/

[5] *ReST vs. GraphQL: Critical Review* https://blog.goodapi.co/rest-vs-graphql-a-critical-review-5f77392658e7