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

Noch neu Schreiben

Diese Studienarbeit beschreibt die Chat-Applikation, die im Rahmen des Praktikums im Fach *Verteilte Systeme* im Wintersemester 2017/2018.

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 Middleware-Technologien, vor allem mit 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.

21. Dezember, 2018

2 Konzeption

Schöne Konzeption

- Fitzli

3 Infrastruktur

Github, Wildfly 13, DBs in Docker

- Fitzli

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 Zugriff 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 ist 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

Ergebnisse etc.

4.9 GraphQL

Im Folgenden Abschnitt wird die GraphQL-Schnittstelle und die Vorgehensweise der Implementierung mit den aufgetretenen Herausforderungen in diesem Projekt erläutert. Abschließend werden mittels eines Vergleichs die Vor- bzw. Nachteile von *ReST* und *GraphQL* diskutiert.

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. [1]

Die im Folgenden beschriebenen drei 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. [2]

```
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. [2]

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

Der Server gibt die angefragten Daten im Anschluss durch eines JSON-Body 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 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 ausgebenw 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. [3]
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*. [3]
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. [4]

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.

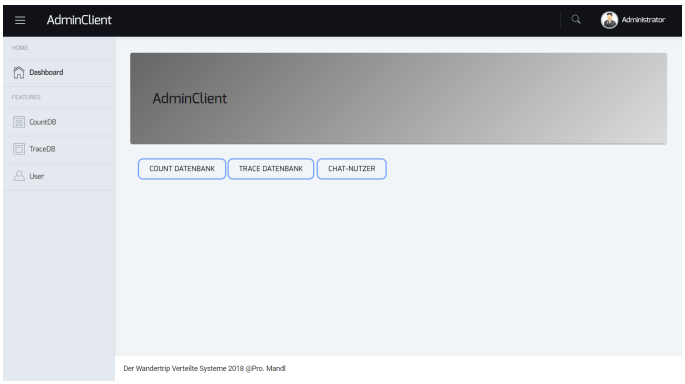

Fig. 1. Startseite des Admin-Clients.

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 1). 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 2 zeigt die Komponente des Admin-Clients, die für die Verwaltung der Zähler zuständig ist.

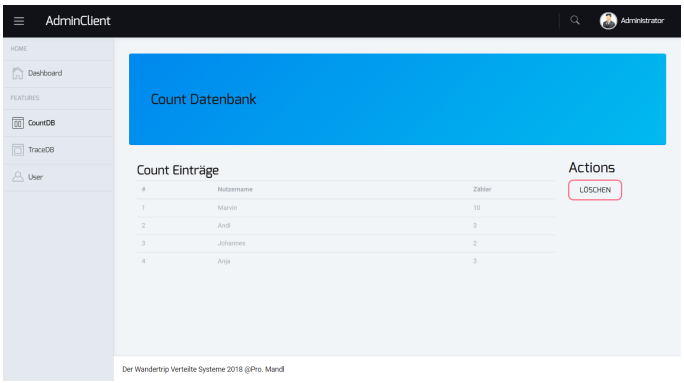

Fig. 2. Count-Verwaltung des Admin-Clients.

Diese Komponente ruft die GraphQL Schnittstelle unter `/graphql?=query={allCounts{id userName counter}}` mit einem HTTP GET Aufruf auf um alle Zähler Einträge darstellen zu können. Zum Löschen aller Zähler Einträge wird die GraphQL Schnittstelle mit einem POST Request unter `/graphql` aufgerufen. Der Body dieser Anfrage muss die Mutation im JSON Format enthalten. `{"query": "mutation={clearCounts}"}`

4.10.2 Trace-Administration

Der Admin-Client kann Einträge der versendeten Nachrichten eines Chat-Nutzers anzeigen und diese ebenfalls löschen. Abbildung 3 zeigt die Komponente des Admin-Clients, die für die Verwaltung der protokollierten Nachrichten zuständig ist.

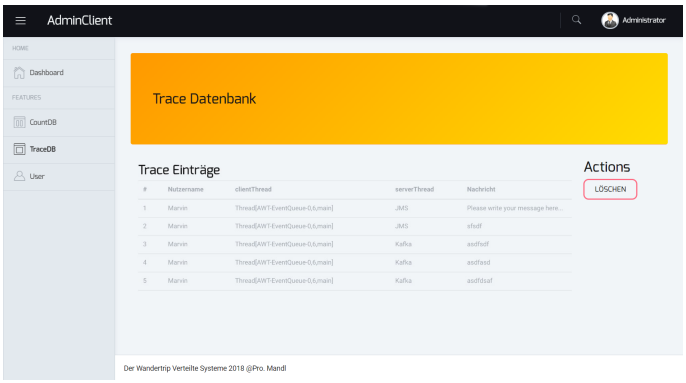

Fig. 3. Trace-Verwaltung des Admin-Clients.

Die Trace-Verwaltung ruft die GraphQL Schnittstelle unter `/graphql?=query={allTraces{id userName clientThread message}}` mit einem HTTP GET Aufruf auf. Zum Löschen aller Einträge wird die GraphQL Schnittstelle mit einem POST Request unter `/graphql` aufgerufen. Der Body dieser Anfrage muss die Mutation im JSON Format enthalten. `{"query": "mutation={clearTraces}"}`

4.10.3 Chat-User-Administration

Der Admin-Client kann alle angemeldeten Chat-Nutzer anzeigen und diese ebenfalls löschen. Beim Löschen werden die Nutzer zuvor von der Chat-Anwendung abgemeldet. Abbildung 4 zeigt die Komponente des Admin-Clients, die für die Verwaltung der Chat-Nutzer zuständig ist.

Die Chat-Nutzer Komponente ruft die GraphQL Schnittstelle unter `/graphql?query={allChatUsers{name}}` mit einem HTTP GET Aufruf auf. Zum Löschen aller Einträge wird die GraphQL Schnittstelle mit einem POST Request unter `/graphql` aufgerufen. Im Body des Requests muss die angeforderte Mutation (`{"query": "mutation{clearChatUsers}"}`) mit den Contentype `application/JSON` enthalten sein.

Representational State Transfer (ReST)

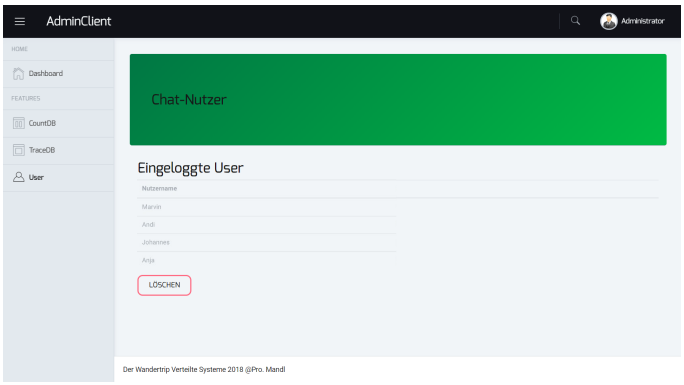

Fig. 4. Chat-Nutzer-Verwaltung des Admin-Clients.

5 Fazit

Schönes Fazit

- evtl Herausforderungen

References

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

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

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

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