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

Johannes Knippel, Marvin Staudt, Andreas Westhoff und Anja Wolf

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

Schönes Fazit

- 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`	Chat-Nutzer Name
`counting`	Zähler der gesendeten Nachrichten

TABLE 2
Struktur der Trace-Tabelle.

Attribut	Funktion
`ID`	Eindeutige Id
`username`	Chat-Nutzer Name
`clientthread`	Thread-Name des Clients
`serverthread`	Identifikation des Servers
`message`	Die 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

4.8 Benchmark-Client

Ergebnisse etc.

4.9 GraphQL

4.9.1 Was ist GraphQL?

4.9.2 Die implementierte GraphQL-Schnittstelle

evtl Herausforderung

4.9.3 GraphQL vs. ReST

4.10 Admin-Client

- Fitzli

```
{"data":
 {"allTraces":[
        {"userName":"Marvin",
         "message":"Hallo Andreas"},
        {"userName":"Andreas",
         "message":"Hallo Marvin"},
        {"userName":"Alexandra",
         "message":"Wie geht es dir?"}]
 }
}
```

Representational State Transfer (ReST)

`application/JSON`

5 Fazit

Schönes Fazit

- Fitzli

Blbalalb

References

[1] *Basic Tutorial: Introduction* https://www.howtographql.com/basics/0-introduction/

[2] *GraphQL is the better REST* https://www.howtographql.com/basics/1-graphql-is-the-better-rest/

[3] Blogpost: *REST versus GraphQL*, E. Chimezie, 2017 https://blog.pusher.com/rest-versus-graphql/

[4] Blogpost: *GraphQL vs Rest: Overview*, P. Sturgeon, 2017 https://philsturgeon.uk/api/2017/01/24/graphql-vs-rest-overview/