Ludwig-Maximilians-Universität München
Institut für Informatik
Lehrstuhl für Mobile und Verteilte Systeme
Prof. Dr. Claudia Linnhoff-Popien

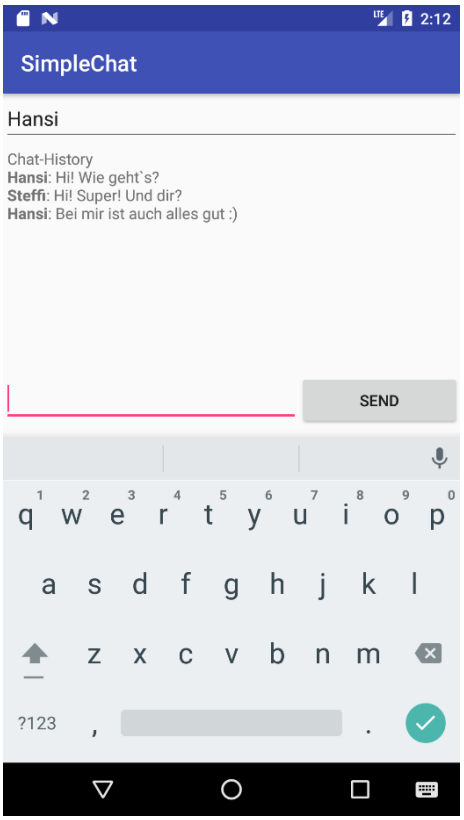
mobile and distributed systems group

Praktikum Mobile und Verteilte Systeme
Sommersemester 2019
Übungsblatt 2

SimpleChat – Thema der heutigen Übung ist die Implementierung eines einfachen Chat-Clients, der über einen Server Nachrichten austauschen kann. Hierfür stehen prinzipiell 2 Ansätze zur Verfügung, von denen eine implementiert werden muss. Der Client baut entweder einen bidirektionalen Kommunikationskanal mit dem Server über WebSockets auf, oder frägt in regelmäßigen Abständen beim Server an, ob neue Nachrichten zur Verfügung stehen (polling).

Aufgabe 1: Android Chat-Client

Erstellen Sie zunächst ein neues Projekt *SimpleChat* in Android Studios und implementieren Sie eine einfache Activity, welche die UI des Chatfensters realisiert. Ihre Implementierung könnte bspw. wie folgt aussehen:



Sie brauchen also jeweils ein Feld für den Nutzernamen, die empfangenen Nachrichten und für die Nachricht, die Sie senden wollen. Zudem einen Button zum Senden der Nachricht.

Aufgabe 2: Server

In dieser Aufgabe sollen Sie mittels dem Spring-Framework (https://spring.io/) einen Server aufsetzen, der mit Ihrem Chat-Client kommunizieren kann und die Nachrichten empfängt bzw. ausliefert. Hierfür stehen Ihnen, wie anfangs erwähnt, 2 Möglichkeiten offen:

a) Zum einen können Sie WebSockets verwenden und einen bidirektionalen Kommunikationskanal öffnen, so dass der Server beim Eintreffen einer Nachricht aktiv an alle verbundenen Clients die Nachricht weiterleiten kann.

b) Sie können aber auch einen reinen RESTful Webservice verwenden. Hierzu brauchen Sie zwei REST-konforme Dienste bzw. Schnittstellen (POST zum Senden und GET zum Empfangen von Nachrichten). Anders als bei Sockets müssen Sie sich hier eine Update-Strategie überlegen, so dass ein Client sich über neue Nachrichten informiert und diese beim Server abholt (polling).

Entscheiden Sie sich bewusst für eine Variante und implementieren Sie die entsprechende Server-Logik. Die folgenden Unterpunkte geben Ihnen einige Hinweise zur Implementierung mit dem Spring-Framework.

Server aufsetzen mit Spring

Da der Sever komplett unabhängig von Ihrem Android-Client ist, verwenden wir zur Implementierung nicht Android Studios, sondern IntelliJ (https://www.jetbrains.com/idea/).
Erstellen Sie hierin ein neues Projekt *SimpleChatServer* und verwenden Sie *Gradle* als Build-Manager. Passen Sie Ihre build.gradle Datei wie folgt an:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.5.5.RELEASE")
    }
}

apply plugin: 'java'
apply plugin: 'idea'
apply plugin: 'org.springframework.boot'

jar {
    baseName = 'simpleChatServer'
    version =  '0.1.0'
}

repositories {
    mavenCentral()
}

sourceCompatibility = 1.8
targetCompatibility = 1.8

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    testCompile('org.springframework.boot:spring-boot-starter-test')
}
```

Gradle sorgt nun dafür, dass beim Build Ihres Projekts die richtigen Pakete eingebunden werden und achtet auch auf Abhängigkeiten. Sie müssen nur darauf achten, dass Ihr Projekt mit Gradle synchronisiert ist (IntelliJ gibt hier Hilfestellung).

Wenn Sie Gradle richtig konfiguriert und mit ihrem Projekt synchronisiert haben, kennt Ihr Compiler nun die org.springframework.boot Library. Erstellen Sie in Ihrem Verzeichnis unter src/main/java das Paket *myServer* und darin eine Java-Klasse Application.java. Diese sollte wie folgt aussehen:

```java
package myServer;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Die *Application.java* beinhaltet die Main-Methode als Einstiegspunk, die von Gradle durch die Annotation `@SpringBootApplication` als Einstiegspunkt der Server-Applikation automatisch erkannt wird. Der Server ist in dieser Minimalkonfiguration bereits ausführbar. Öffnen Sie hierzu in IntelliJ ein Terminal (Console) und geben Sie unter dem aktuellen Projektpfad den Befehl *gradlew bootrun* ein. Ihr Projekt sollte nun gebaut und der Server gestartet werden. Wenn Sie die Zeile *Tomcat started on port(s): 8080 (http)* lesen, wissen Sie, dass die Ausführung erfolgreich war und der Server bei Ihnen lokal auf dem Port 8080 läuft. Sie können dies auch im Browser überprüfen, indem Sie *http://localhost:8080/* eingeben. Mit *strg+c* in der Console beenden Sie den laufenden Server.
Hinweis: Sie können den Server-Port frei wählen, indem Sie unter /src/main/resources eine neue Datei *application.properties* anlegen und `server.port=PORTNUMMER` eintragen.

Nachdem Sie nun einen minimalistischen Tomcat-Server starten und beenden können, beginnen Sie nun damit die Server-Logik für Ihre Chat-Anwendung zu schreiben. Je nachdem für welche Variante Sie sich entschieden haben, finden Sie die nachfolgendenden Hinweise zur Implementierung:

Variante A: Realisierung über WebSockets

Um unter Spring mit WebSockets zu arbeiten, müssen Sie zunächst die entsprechende Library einbinden. Fügen Sie hierzu in Ihrer build.grade Datei die entsprechende Zeile hinzu:

```
dependencies {
    …
    compile("org.springframework.boot:spring-boot-starter-websocket")
```

Erzeugen Sie anschließend eine neue Java-Datei, welche die Konfiguration des WebSockets übernimmt. Hierzu müssen Sie die Annotationen `@Configuration` und `@EnableWebSocket` verwenden und das Interface *WebSocketConfigurer* implementieren.

Hierfür muss die Methode `registerWebSocketHandlers(WebSocketHandlerRegistry registry)` überschrieben werden. Die dabei übergebene *registry* bietet die Methode `addHandler(WebSocketHandler handler, String path)`. Somit können Sie Ihren eigenen Handler registrieren und einen Pfad als Endpunkt bestimmen, bspw.: „/chat“.

Erstellen Sie nun Ihren eigenen Handler als separate Java-Klasse, welcher die Aufgabe der Nachrichtenvermittlung an alle angemeldeten Clients übernimmt. Leiten Sie Ihre neue Klasse von der Klasse `TextWebSocketHandler` ab und überschreiben Sie die drei Methoden sinnvoll:

- `afterConnectionEstablished(WebSocketSession session)`
  - Was soll passieren, wenn ein Client die Verbindung zum Server aufgebaut hat?
- `afterConnectionClosed(WebSocketSession session, CloseStatus status)`
  - Was soll passieren, wenn ein Client die bestehende Verbindung beendet hat?
- `handleTextMessage(WebSocketSession session, TextMessage message)`
  - Was soll mit einer Nachricht passieren, die dem Server übermittelt wurde?

Hinweis: Die Klasse soll wie ein Broker funktionieren, d.h. wenn eine Nachricht beim Server ankommt, soll diese entsprechend an alle verbundenen Clients sofort ausgeliefert werden.

Weiter Informationen über WebSockets und Spring sind auch online zu finden. Einen guten Überblick gibt bspw. https://docs.spring.io/spring/docs/current/spring-framework-reference/html/websocket.html Bitte beachten Sie allerdings, dass die Verwendung von STOMP als Messaging Protokoll, was häufig mit Spring verwendet wird, in Verbindung mit Android Clients schwierig ist, da es clientseitig (noch) kaum unterstützt wird.

Variante B: Realisierung über einen RESTful Webservice

Erzeugen Sie bei dieser Variante eine neue Java-Klasse, welche den RESTController implementiert (Annotation mittels @RestController). Sie benötigen für Ihre Chat-Anwendung zwei Schnittstellen zum Erhalt und Ausliefern von Textnachrichten (POST und GET).
Wie Sie in Spring einen RESTful Webservice erstellen, entnehmen Sie bitte dem offiziellen Tutorial: https://spring.io/guides/gs/rest-service/

Zudem sollten Sie ein Model zur Datenhaltung (Speicherung der Nachrichten) in einer eigenen Java-Klasse implementieren. Mittels der Annotation *@Autowired* können Sie dem Framework sagen, dass es genau eine Instanz des Models beim Starten des Servers erstellt. Damit haben Sie Zugriff auf Ihr Model und können bei einer GET-Anfrage die Nachrichten ausliefern bzw. bei einem POST-Aufruf eine Nachricht im Model abspeichern.

Beachten Sie bitte, dass Spring die Jackson JSON Library verwendet, um Objekte einer Klasse in einen gültigen JSON-String zu konvertieren und schließlich zu übertragen. Sie sollten daher noch eine Java-Klasse für eine Nachricht definieren, welche als Instanzvariablen *String Nutzername* und *String Nachricht* beinhaltet. Damit können Sie ganz einfach Nachrichten als Instanzen verwalten und als JSON-Strings übertragen.

Aufgabe 3: Client und Server zusammenbringen

Egal für welche Variante Sie sich entschieden haben, Sie sollten nun Ihren funktionsfähigen Server mittels *gradlew bootrun* starten können. Wenn der Server läuft müssen Sie nun Ihren Android-Client so erweitern, dass dieser Nachrichten an den Server schicken und empfangen kann.

Variante A: Realisierung über WebSockets

Wenn Sie bei Ihrem Server WebSockets verwendet haben, müssen Sie nun auf Client-Seite die entsprechende abstrakte Klasse *WebSocketClient* aus dem Paket *org.java_websocket.client* implementieren und sich mit der entsprechenden Instanz verbinden. Damit Ihr Compiler Bescheid weiß, fügen Sie zunächst folgende Zeile in Ihre build.gradle Datei (*Module: app*) hinzu:

```
dependencies {
    …
    implementation "org.java-websocket:Java-WebSocket:1.3.0"
```

Implementieren Sie nun ein *WebSocketClient*, indem Sie die entsprechenden abstrakten Methoden der Klasse überschreiben:

- `onOpen(ServerHandshake serverHandshake)`
- `onMessage(String s)`
- `onClose(int i, String s, boolean b)`
- `onError(Exception e)`

Die Klasse *WebSocketClient* verlangt im Konstruktor eine gültige URI, die auf den von Ihnen definierten Endpunkt an Ihrem Server verweist bspw.: http://10.153.199.132:8080/chat. Beachten Sie hierzu, dass Sie die IP-Adresse Ihres lokalen Servers einstellen müssen. Ein einfaches localhost:8080 funktioniert nicht, da der Android-Emulator dann sich selbst referenziert und nicht die Host-Maschine.

Ggf. müssen Sie zusätzlich im Konstruktor die Version des WebSockets einstellen, damit es mit Ihrem Server kompatibel ist, bspw.: *new WebSocketClient(uri, new Draft_17())*.

Rufen Sie beim Start Ihrer Activity die Methode *connect()* auf Ihrer erzeugten WebSocketClient Instanz auf, um eine Verbindung aufzubauen.

Über die Methode *send(String msg)* können Sie Textnachrichten, die der Nutzer eingibt über die WebSocketClient Instanz an den Server schicken. Achten Sie hier wiederum darauf, dass Sie gültige JSON-Strings als Nachrichten versenden, die Ihr Server verarbeiten kann. Hierfür können Sie JSON Bibliotheken wie Jackson oder das bereits im Android SDK integrierte org.json verwenden.

Logischerweise bekommen Sie dann auch JSON-Strings als Nachricht vom Server in der Methode *onMessage(String s)* geschickt, die Sie in Ihrer Activity in der Chat-History anzeigen sollen. Dies geht nur über den UiThread! Wenn Sie erfolgreich Nachrichten übertragen und anzeigen können sind Sie fertig. Testen Sie Ihre Applikation dann auch wenn möglich mit mehreren Geräten und schrieben Sie sich gegenseitig Nachrichten.

Variante B: Realisierung über einen RESTful Webservice

Wenn Sie einen RESTful Webservice implementiert haben, müssen Sie nun die entsprechenden Methoden für das Senden einer Nachricht und den Empfang von Nachrichten auf dem Client realisieren. Hierzu sollten Sie eine eigene Java-Klasse schreiben, welche diese Funktionen prinzipiell übernimmt.

Verwenden Sie eine *HttpURLConnection* aus dem Paket java.net und verbinden sie sich mit den jeweiligen REST-Schnittstellen auf Ihrem Server. Beachten Sie hierzu, dass Sie die IP-Adresse Ihres lokalen Servers einstellen müssen. Ein einfaches localhost:8080 funktioniert nicht, da der Android-Emulator dann sich selbst referenziert und nicht die Host-Maschine.

Für das Senden von Nachrichten sollten Sie POST verwenden und Ihre Nachricht in einen gültigen JSON-String konvertieren und versenden. Hierfür können Sie JSON Bibliotheken wie Jackson oder das bereits im Android SDK integrierte org.json verwenden. Definieren Sie zusätzlich den Content-Type als „application/json“ und als charset „utf-8“.

Den JSON-String sollten Sie mittels *DataOutputStream* über Ihre geöffnete Verbindung an den Server versenden. Vergessen Sie nicht den Stream nach Nutzung wieder zu schließen.
Denken Sie auch daran, Server-Anfragen außerhalb Ihres UI-Threads laufen zu lassen, bspw. durch Verwendung eines AsyncTask. Für das Empfangen von Nachrichten schicken Sie eine GET-Anfrage an Ihre definierte REST-Schnittstelle. Definieren Sie wieder den Content-Type als „application/json“ und als charset „utf-8“.

Nun brauchen Sie eine InputStreamReader, welcher den Datenstrom vom Server ausliest. Vergessen Sie nicht auch diesen Stream nach Nutzung wieder zu schließen.

Damit haben Sie die wichtigsten Verbindungen implementiert und können Daten versenden und empfangen. Allerdings muss der Client (anders als bei Verwendung von WebSockets) selbst aktiv nach Nachrichten beim Server anfragen. Daher: Überlegen Sie sich eine Strategie, wie und wie oft Sie nach Nachrichten fragen und diese dann in der Chat-History anzeigen wollen! Sie können bspw. einen eigenen Service in Ihrer App etablieren, der jede Sekunde nach allen bzw. nach neuen Nachrichten fragt. Dies bleibt Ihnen überlassen. Wenn Sie erfolgreich Nachrichten übertragen und anzeigen können sind Sie fertig. Testen Sie Ihre Applikation dann auch wenn möglich mit mehreren Geräten und schrieben Sie sich gegenseitig Nachrichten.