**Технически университет – гр. Варна**

**Факултет по изчислителна техника и автоматизация**

**Катедра „Софтуерни и интернет технологии“**



ДИПЛОМНА РАБОТА

**Тема:**

„Проектиране и реализация на система за мигновени съобщения“

**Изготвил:** Константин Костов

**Специалност:** Софтуерни и интернет технологии

ТУ Варна, 2017 г. Ръководител: доц. Христо Ненов

**Съдържание:**

1. **Увод …………………………………………………………………… 3**
2. **Анализ на проблема и избор на технологии …………………….. 5**
3. **Проектиране на системата ................................................................ 16**

**3.1. Потребителски интерфейси ………………………………..…. 16**

**3.1.1. Сървърен интерфейс……………………………...…… 16**

**3.1.2. Клиентски интерфейс……………………………...….. 16**

**3.2. Архитектура на системата …………………………………..... 23**

**3.3. Трансфер на информацията между клиент и сървър …...… 28**

**3.3.1. TCP Сокети……………………………………..…………… 28**

**3.3.2. Протокол за комуникация на апкликационно ниво …………………………………………………………………...…... 29**

**3.4. Бази от данни………………………………………...…………… 30**

* + 1. **Сървърна база от данни…………………………..……….. 30**
    2. **Клиентска база от данни …………………………………... 31**

1. **Реализиране на системата……………...…………………………... 33**
   1. **Сървърна реализация ……………………………………….. 33**
   2. **Клиентска реализация ………………………………..……... 40**
   3. **Апликационен протокол за комуникация ………………... 47**
   4. **База от данни………………………………………..………… 49**

**4.4.1 Сървърна база ……………………………………………… 49**

**4.4.2. Клиентска база …………………………………………….. 50**

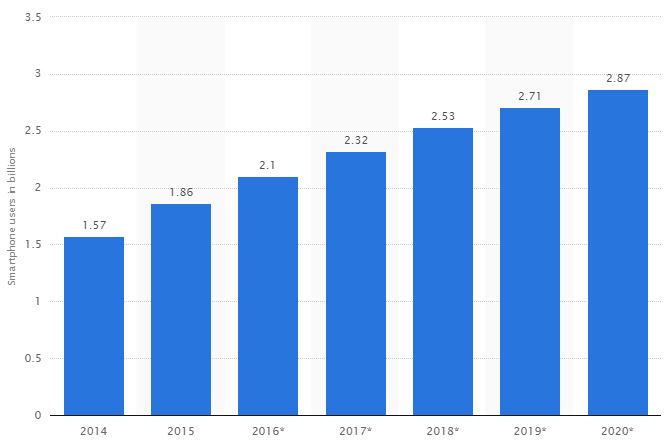
1. **Тестове и резултати ……………………...……………………….… 53**
2. **Заключение ………………………………...………………………... 56**

**Библиография…………………………...…………………………......... 57**

**Приложение……………………………………...………………….........58 Сорс код ………………………………………...……………………. 59**

**1. Увод**

С развитието на технологиите се подобряват и методите за комуникация между хората. Докато само до преди около век на писмо да пътува от Европа до Америка са му били нужни седмици, дори месеци сега може да отнеме само няколко секунди. Един от методите за постигане на тази цел е с помощта на интернет. Чрез чат система в реално време се предоставя бърз, удобен и лесен начин за комуникация на големи разстояния с изключително голяма надеждност. В зависимост от типа информация, която се предава сигурността на кореспонденцията може да бъде гарантирана чрез криптографски алгоритми.

****С развитието на транспорта отпадат до голяма степен ограниченията наложени от големите разстояния и с глобализацията - границите между държавите. Така се предоставя възможност на хората по-лесно да общуват помежду си, а на бизнеса да се свърже по-добре с клиентите си. По данни на сайта “Statista” в световен мащаб към момента има над 2 милиарда активни смарт телефони ( виж таблица № 1.1). Чийто брой се очаква да нарасне до над 2.8 милиарда до 2020 година.

**Таблица 1.1.**

**Брой на смарт телефони по света.** (източник: Statista)

Чрез мобилно приложение се позволява на хората да имат директна връзка с компаниите, дори когато са в движение.

Целта на настоящата разработка е да предостави точно такъв канал за връзка чрез приложение за клиентски достъп. Във времена на висока конкуренция липсата на адаптивност и отговаряне на нуждите на потребителите често води до големи загуби, ако не дори до фалит. Така например ако застрахователна компания предостави чат система, се създава допълнителна стойност на продукта или услугата, който продава. Задачата на дипломната работа е да намали разходите, като същевременно се увеличи полезността. Това най-общо е основната цел, към която се стреми всеки бизнес, независимо от областта му на работа.

**2. Анализ на проблема и избор на технологии**

Доста настоящи приложения дават възможност за обратна връзка. Най– често те съдържат някакъв вид форма за обратна връзка, а съдържанието и се изпраща като имейл. Това е достатъчно в случаите, когато потребителят не очаква да получи отговор или поне не веднага. При нужда от такъв, той ще бъде изпратен на имейла предоставен от потребителя. Когато обаче е нужен бърз отговор, една чат система за мигновени съобщения перфектно решава проблема. Чрез нея потребителя ще може да задава стойте въпроси към call център или техническата поддръжка на компанията и в реално време да получи адекватен отговор.

Погледнато от бизнес гледна точка това носи след себе си само позитивни неща. Така например ще се облекчи натоварването в call центъра на компанията. Поради факта, че един човек може да води комуникация с повече от един клиент чрез лични съобщения към него, докато при директен разговор по телефона това е невъзможно.

Съществува въпроса дали да не се избере вече готов или полуготов продукт на 3-та компания.

**Недостатъци при използване на външен софтуер за комуникация между клиент и компания:**

При използването на чат система на трети лица, потребителя трябва да бъде информиран, че посредством предоставения софтуер от тях, той може да осъществи контакт с представители на фирма Х. Това от своя страна води до допълнителни разходи за маркетинг, без да има гарантирана полза. В защита на метода може да се каже, че ако клиента има вече такъв инсталиран софтуер, то не би му се налагало да инсталира още един. Като допълнение можем да добавим, че няма да бъде нужно на клиента да се запознава с нов продукт.

Настъпва обаче проблема за заплащането на този продукт или услуга. Факт е, че в днешно време повечето системи не изискват закупуването на еднократен лиценз, а месечна или годишна такса. Друг недостатък е, че някои трети компании таксуват според броя клиенти, които ползват продукта. За фирми с много индивидуални клиенти това би довело до значителни разходи, които не са обосновани или биха довели до значително поскъпване на продуктите и/или услугите, които предлагат.

Разбира се съществуват и евтини или привидно безплатни програми, чрез които задачата може да бъде изпълнена. Те обаче показват реклами на потребителите, за да могат и да имат някакви приходи, чрез които да съществуват. Това може да доведе до раздразнение на потребители и прекратяване на взаимоотношенията не само с фирмата посредник, но и с основната фирма.

**Недостатъци при използването на SDK чат:**

При разработване на едно приложение може да бъде закупен лиценз за използването на framework за чат системата. Това би намалило времето за производство, но би оскъпил продукта.

**Собствена реализация:**

Реализацията на чат система носи след себе си редица предимства. Погледнато от страната на разработчиците при нужда от промяна или отстраняване на бъгове ще се извършат по- бързо понеже вече са запознати с кода на проекта. Докато при използването на библиотека за чат система модифицирането и персонализирането до голяма степен са или забранени или сложни за реализиране.

Когато се използва собствен сървър вместо споделен или изчислителни ресурси в облака, се оскъпява крайния продукт, но за сметка на това, може да се прави по - добър анализ на потреблението и потребителските предпочитания. Което разбира се води до подобряване на продукта, така че да отговаря повече на изискванията на неговите ползватели.

**Избор на технологии:**

**Избор на сървър:**

За по- висока производителност ще се направи собствена имплементация на сървър написан на програмния език Java. Липсата на създаден вече продукт дава повече свобода на разработчиците да имплементират свое решение на проблема. Java се е доказала като надежден и сигурен език за програмиране, с достатъчна висока производителност. Писането и на сървъра, и на клиентката част на този език дава възможност един и същи екип да реализира и двата проекта. Това премахва времето за търсенето на два различни екипа, които да изпълнят задачите.

Комуникацията между клиентите и сървър трябва да е надеждна. Всяка информация която е изпратена от сървъра трябва да пристига в същата последователност до клиента, без загуба на данни. Такава връзка е TCP. Тя предлага канал от точка до точка по интернет. За да се осъществи връзка по TCP, системата ще използва сокети.

Като допълнение се използва собствен протокол на апликационно ниво за организация на информацията преди да се кодира като текст.

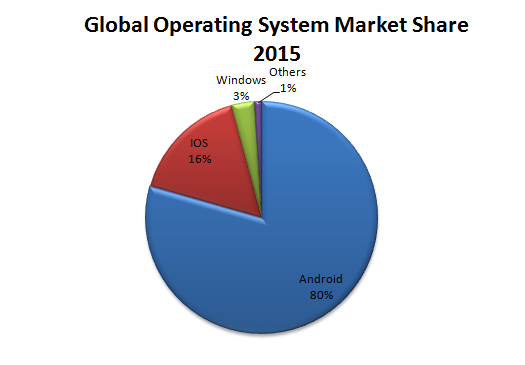
**Избор на база от данни:**

В днешно време е безсмислено да има система, която да няма поне един вариант на база. За база на сървъра е избрана релационна база от данни. Тя е консистентна, бърза и лесна за имплементиране. Конкретната имплементация ще бъде на MySQL. Тя е доказала свойте предимства и към момента е с най-голям пазарен дял, което още веднъж показва нейните възможности.

За база на клиента ще се използва специално дефиниран от ОС файл за запаметяване на специфични за клиента настройки и предпочитания. Тази система е плоска БД с максимален размер до 4 KB. Ще се използва и вградената релационна база с малки системни изисквания – SQLite.

**Избор на клиентска платформа:**

За клиентска платформа най-подходяща в случая е Android. Това да дължи на факта, че има голям процентен дял на пазара на мобилни устройства (виж фигура 2.1).



**Фигура 2.1. Пазарен дял на мобилните операционни системи.** (източник: forbes.com)

Също така масовия потребител в България използва Android телефон.

Тази операционна система предлага много възможности, който са само в полза на потребителя. Най-значимият е, че се поддържа от изключително много устройства с различен размер и форма по целия свят. За да се гарантира успеваемост на разработвания продукт е нужно той да има някой определени характеристики. Основните включват многоезичност на приложенията, поддръжка на различни по големина екрани и различни по версия платформа.

**Фундаменталност на Андройд системата:**

Приложенията за Android са написани на програмния език Java. Android SDK инструменти компилират кода заедно с всички данни и ресурсни файлове в APK, което представлява голям архив с разширение на файла .apk. Този файл съдържа всичката нужна информация за инсталирането на приложението върху устройството на потребителя.

Всяко изпълняващо се приложение работи независимо от останалите. Така се изпълняват следните изисквания за сигурност.

* Андройд е многопотребителска Linux система, където всяко приложение е отделен потребител.
* По подразбиране ОС раздава на всяко приложение уникално Linux потребителско ID. (Това ID се използва само от ОС и е неизвестно за приложението). Системата задава права на всички файлове в приложението и само потребителят с определено ID може да ги достъпи.
* Всяко приложение има своя виртуална машина и кода който се изпълнява е изолиран от останалия свят.
* По подразбиране всяко приложение е отделен Linux процес. OC стартира процеса, когато някой от компонентите имат нужда да бъдат изпълнени и го прекратяват, когато вече не е нужен или системата има нужда от допълнителни ресурси за по – високо приоритетни приложения.

ОС работи на принципа на най-малко привилегирования. Т.е. всяко приложение по подразбиране има достъп до компонентите, който са му нужни, за да си свърши поставената задача и не повече. Това създава сигурна среда която гарантира че приложение не може да достъпи ресурси, които не са и предоставени. Въпреки това е възможно две приложения да споделят данни или да достъпват системни услуги.

* Могат да се създадат приложения, които да споделят едно и също Linux ID, което позволява да достъпват взаимно файловете си. За да се пестят системни ресурси приложения с общо ID могат да работят в един и същи процес и да споделят една виртуална машина. За тази цел те трябва да бъдат подписани с един и същи сертификат.
* За някой ресурси или данни като например контактите от телефона, SMS-и, външна карта памет, камера, Bluetooth и др. потребителят експлицитно трябва да разреши ползването им. Настоящата реализация ще изисква достъп до контактите, за да се осигури по-бърз и лесен вход в система.

**Многоезичност:**

Android позволява много лесна скалируемост относно добавянето на нови езици. Без промени по функционалността може да се персонализира продукта да отговаря на изискванията на дадена културна общност. Android извършва тази настройка според избрания от потребителя език на операционната система. Ресурсите за всеки език се записани в ресурсните директории на проекта.

**Фонов режим:**

Голяма част от приложението зависи от това какво се изпълнява на фонов режим. Service-а е входна точка за поддържане на приложение работещо на заден фон поради обективни причини. Използва се за изпълняване на задачи, които са дълги по време или изискващи значителна изчислителна мощ. Service-a не предоставя потребителски интерфейс. Например може да се използва за слушане на музика докато потребителя използва друго приложение или може да сваля информация от интернет без да блокира потребителски вход. Друг компонент като Activity може да стартира service и дори да се обвърже с него, за да обменят информация. Има точно две различни типа услуги, различаващи се по това как казват на системата да управлява приложението.

* Започнатите услуги (Started services) казват на системата да ги поддържа работещи докато не приключат работата която изпълняват. Така те могат да изпълняват песен или да синхронизират информация с облака дори след като потребителят спре приложението.
* Обвързаните услуги (bound services ) се изпълняват само защото приложение или системата им е казала че искат да извършат дадена работа. Това на практика е услуга предоставяща API за даден процес. Така ОС разбира, че процес А е обвързан с процес В и знае че трябва да поддържа процес В жив за А. Особено когато процес А от значение за потребителя, това означава, че и процес В става от значение за потребителя. По този начин се получава един доста добър метод за изграждане на сложни системни концепции. Live wallpapers, слушатели за нотификации, screen savers, входни методи и много други базови системи функции са изградени като услуги към които по късно приложенията имплементират и се свързват. Точно такъв bound service служи за основа върху, която се изгражда услугата за връзка към сървър, посредством сокет.

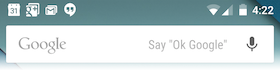
**Broadcast receivers:**

Broadcast receivers са компонент, който позволява на системата да доставя събития на приложения извън обикновения начин на комуникация, отговаряйки на събития, които са глобални за цялата система. Понеже те представляват един много добре дефиниран вход в приложението, ОС може да доставя известия дори, когато то не работи. Например приложение може да запише час за изпълняване на аларма за публикуване на нотификация за информиране на потребителя за предстоящо събитие. При предаването на алармата на Broadcast receiver, се сваля отговността от въпросното приложение, което премахва нуждата то да продължава да работи и може да бъде спяно, за да пести батерия и други ресурси, докато не се задейства алармата. Много от алармите са системни – известяване, че екрана се е изключил, че нивото на батерията е ниско, че снимка е направена. Приложения също могат да създават събития, например че дадена информация е била изтеглена и може да бъде използвана от други. Въпреки че повечето receiver-ри нямат графичен интерфейс те могат да създават нотификации показващи се в горния край на екрана за настъпването на събитие.

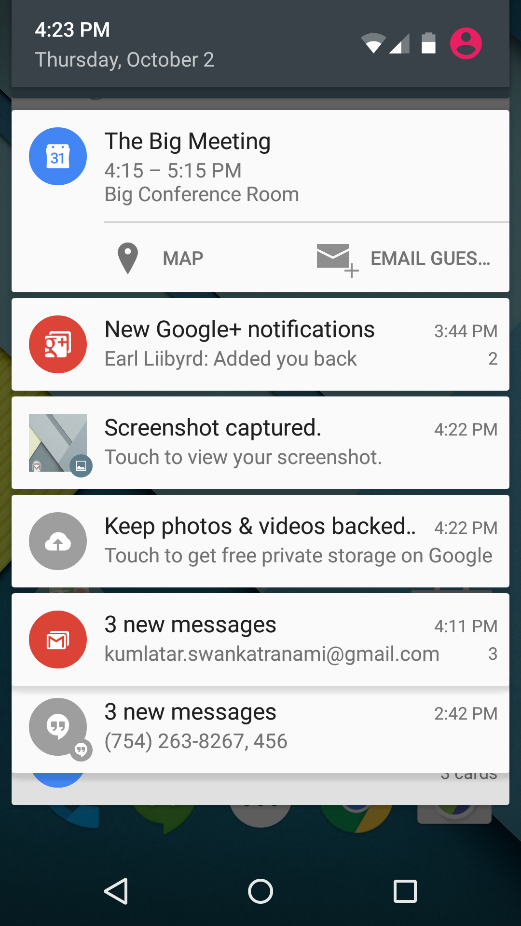
**Нотификации:**

Нотификациите са жизненоважни в днешно време. Чрез тях се постига информиране на потребителя в реално време, без да не налага периодичното проверяване за настъпването на дадено събитие.

Нотификацията е съобщение което може да бъде показано на потребителя извън нормалния потребителски интерфейс. Когато се укаже на ОС да създаде нотификация, първоначално тя се появява в зоната за нотификации. За да види детайлна информация, потребителят трябва да отвори нотификационното чекмедже. И зоната, и чекмеджето са системно контролирани и клиента може да ги отвори по всяко време. Виж фигури №2.2 и №2.3.



**Фигура 2.2.** **Нотификации в нотификационната зона** (източник: android.com)



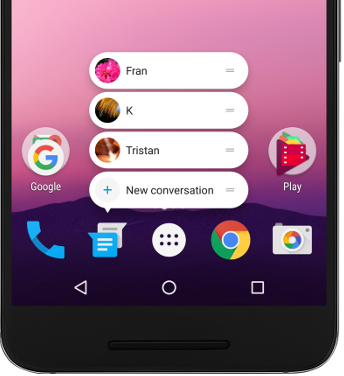
**Фигура 2.3.** **Нотификации в чекмедже** (източник: android.com)

Нотификациите са важна част от интерфейса на Android, затова е важно да се спазват препоръките за техния дизайн. Въпреки че не е задължително, за да се ускори работата на потребителя може да се създадат събития за всяка нотификация. Поне една е препоръчително да се направи, която да отваря приложението, за което е предназначена. Понеже системата, която се имплементира е чат в реално време, за по-малко натискане на излишни бутони, може да се добавят към нотификацията бутони за отлагане на алармата или отговаряне директно с текстово съобщение.

За системи в реално време друго условие е съобщенията да се приемат на момента, затова е препоръчително да са с по- висок приоритет.

**Apps Shortcuts:**

С последната версия към днешна дата (Android 7.1) са добавени apps shortcuts. Реално това позволява на потребителите да заредят ключови функционалности с натискането само на един бутон. За да се извърши това действие е нужно продължително натискане на стартовата икона на приложението, и след това да се избере от показалите се преки пътища действие отговарящо на желанието на клиента. Виж фигура № 2.4.



**Фигура 2.4: Преки пътища** (източник: android.com)

**Съвместимост:**

Към момента се предвижда приложението да работи само в портретен формат. Използването му в формат пейзаж би затруднило потребителя понеже, клавиатурата ще скрива голяма част от екрана и ще остане малко “използваемо“ пространство.

Поддържането на твърде много версии на Android би било твърде неизгодно. От една страна това ще оскъпи проекта и увеличило времето му за изпълнение. От друга страна ще се увеличи броя на хората, които биха използвали приложението. Границата, където се срещат двете идеи, е поддържането на достатъчно версии, мнозинството от хора да могат да инсталират приложението, без това да води до бавно реагиращо приложение или компенсиране от сигурността. Това ще означава от версия Android 4.4 (API 19) до най-новата. Пилотната версия е желателно да бъде достъпна само за ограничен брой клиенти от една единствена държава.

**Избор на метод за комуникация:**

Връзката между клиент и сървър ще се осъществява чрез интернет, като двата най-разпространени метода, които позволяват трансфер на съобщения са чрез протоколите на апликационно ниво HTTP и XMPP.

HTTP протокола е stateless и би изразходвал значително батерия за установяване всеки път на връзка със сървъра. От друга страна има голям overhead и се изпраща големо количество ненужна информация.

XMPP протокола е създаден пред 1999 и е базиран на xml. Отново както при HTTP се изпраща значително количество мета данни, които не носят полза.

При използването на собствен апликационен протокол, може предварително да го пригоди специално за конкретните нужди на системата. Когато и двете комуникиращи страни са запознати с метода на изпращане и получаване на данни, отпада ограничението да ползват една и съща платформа. Така клиентската система може да бъде различно имплементирана система от сървърната.

**3. Проектиране на системата**

**3.1. Потребителски интерфейси**

Системата ще включва множество екрани, като тук са изброени само свързаните с изпълняването на задачата.

**3.1.1. Сървърен интерфейс**

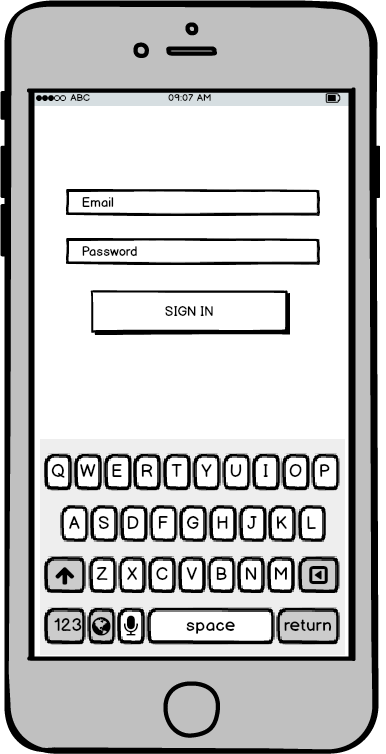
Сървъра сам по себе си няма да има интерфейс, освен конзола, където да се показват логове за състоянието на сървъра. Тези логове ще включват информация за броя на клиентите, които са свързани към него и в реално време да се изписват типа на заявка, която се обработва от него в момента.   
За интерфейс към сървъра ще се ползва клиентското приложение, с разликата, че ще има администраторски екран, който да позволява разговори към много клиенти едновременно.

**3.1.2 Клиентски интерфейс**

* **Splash screen**

Към момента на се предвижда да има начален зареждащ екран (splash screen). Такъв екран не е харесван от потребителите, понеже ги кара да чакат, без да знаят дали приложението ще оправдае очакванията им (ако за първи път стартират приложението). В случай че потребителят използва приложението по няколко пъти на ден (приложения от типа имейл клиент или новинарско приложение) постоянното наблюдаване на началния екран, би ги отегчило и ще ги накара да си помислят, извършва много работа на заден фон при стартиране. Препоръчителния минимален период за визуализация на екрани от такъв тип е 2 секунди. По- малко може да не е достатъчно на клиента да види информацията върху целия екран. Повече от 2 секунди може да използват само ако има предварително зареждане на множество информация като например текстури на игра, които не могат да бъдат заредени на по- късен етап или поетапно зареждане.

* **Login screen**



**Фигура 3.1. Прототип на екран за вход**

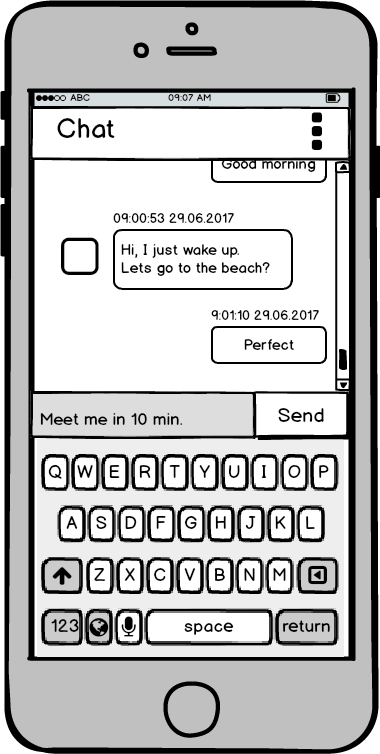
При стартиране на приложението директно ще се зарежда логин екрана. В него ще могат да бъдат въвеждани клиентския имейл и парола. Когато някое от полетата не отговаря на едно или повече изисквания, да се информира потребителя и да се чака за повторен опит за вход. След преминаване на предварителната проверка за валидност на данните, ще бъде направена заявка към сървъра, за да се валидира клиента. А след обработка на заявката от сървъра, се изпраща отговор към клиента. Сървъра от своя страна сравнява получените стойности с тези записани в неговата база от данни. При успех като резултат се изпраща низ за достъп ( access token), който за в бъдеще може да бъде използван вместо имейл и парола. Този низ ще се подновява всеки път, когато клиента се впише в системата чрез парола или не може да предостави низа. Последния остава скрит от потребителя и останалите приложения на телефона, като се записва във временната плоска база от данни с права за четене само от приложението собственик на файла.

Понеже се предполага, че чат системата ще бъде част от по- голямо приложение, екрана за регистриране на потребител не е имплементиран. Това се дължи на факта, че най- вероятно не всеки потребител ще може да се регистрира, а само ползвателите на дадения продукт, който вече ще имат регистрация било то в сайта на продукта или другаде.

* **Основен екран на чат системата**

Основния екран на системата се дели на два вида в зависимост от типа потребител, който се е вписва в приложението. Всяко приложение поддържа и двата изгледа на екрана, което е направено с цел да не се разработват две почти идентични приложения отделно.

**А) Базов екран**

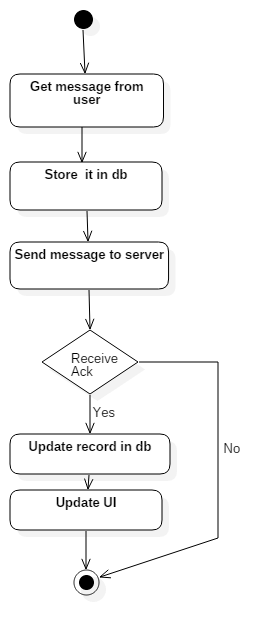
****

**Фигура 3.2. Прототип на чат екран**

Базовия екран е достатъчен, за да отговори на нуждите на потребителя т.е. да предостави начин за комуникация с представители на компанията собственик на приложението.

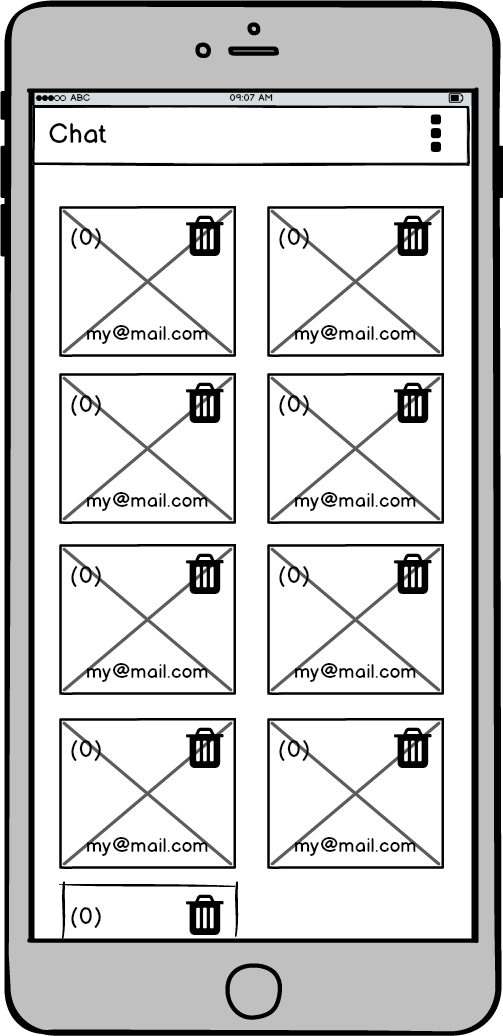
Екрана трябва да съдържа поле за въвеждане на текст от потребителя и бутон за неговото изпращане. При отварянето на екрана да се показва и софтуерната клавиатура от телефона. Над полето за въвеждане на текст да се съдържа списък с изпратените съобщения от клиента, ако има такива и отговорите, които той е получил. Всяко съобщение да се асоциира с време на изпращане или получаване.

Ключов фактор е потвърждението от сървъра, че съобщението на клиента е прието. Нека успешно изпратените съобщения да се оцветяват със син фон. Този цвят е избран понеже той се асоциира с надеждност и сигурност, а неуспешно изпратените съобщения да се оцветяват с червеникав цвят. Червения цвят дразни окото и ще привлече вниманието на потребителя. По подразбиране всички изходящи от телефона съобщения се приемат за неуспешно изпратени, докато не пристигне позитивен отговор от сървъра. Тогава те да променят фона си от червен на син. Виж фигура № 3.3. за последователността при изпращане на съобщение.



**Фигура 3.3. Диаграма на активностите при изпащане на съобщение**

**Б) Администраторски екран**

****

**Фигура 3.4. Прототип на администраторски екран**

Администраторския екрана се различава по това, че чрез негова помощ ще може да се води комуникация с повече от един клиент. Клиентите ще бъдат подредени в 2 колони в списък (виж фигура № 3.4). Всеки клиент ще се визуализира на дисплея като правоъгълник, в който да се зареди неговата профилна снимка или аватара му. Тя да служи за фон на останалите елементи като:

* Иконка за брой непрочетени съобщения
* Иконка за изтриване на чата с клиента
* Имейл на клиента

Разположението на елементите е следното. В горния ляв ъгъл да се изписва в скоби броя на непрочетени съобщения например „(3)“. В горния десен ъгъл да се изобрази иконка за изтриване. Спрямо конвенцията за дизайн на Google, тя трябва да бъде кошче за боклук в червен цвят. За по-голяма сигурност и предпазване от неволното и натискане, да се появява диалогов прозорец питащ потребителя дали наистина това е действието, което иска да извърши. Отново според конвенциите на Google, възможните варианти да бъдат два: позитивен (да бъде озаглавен „DELETE”) и негативен (да бъде озаглавен „CANCEL”). При натискане извън диалоговия прозорец, той да се затвори и да се извърши действието асоциирано с негативния отговор.

Екрана ще има app bar, с падащо меню, от което ще може да бъде добавян нов клиент, с който да се води комуникация. Ако последния вече е добавен, да не се добавя наново. С цел по – добра навигация и намаляване на излишните взаимодействия между потребителя и приложението, клиентите, с които води чат съобщения да се подреждат по време на последна комуникация. Т.е. най – горе на екрана са най – скорошните клиенти, докато надолу са от по – ранен етап. При изтриване на клиент или добавяне на нов в списъка, последния да се пренарежда автоматично.

**Информационни съобщения:**

Потребителят трябва да бъде запознат с всяко едно действие, което се извършва от приложението. Затова било то при успех или неуспех, той трябва да бъде информиран с подходящо съобщение.

От критично значение е показването на съобщение, когато приложението няма връзка с интернет. То да се появява под app bar-a, като дълга лента, заемаща цялата ширина на екрана. В него да е изписано, че няма връзка с интернет, а фона на лентата да бъде жълта на цвят, за да привлече вниманието на потребителя.

Аналогично пак чрез същата лента да се информира клиента, когато не може да се установи връзка със сървъра поради някакъв проблем.

В случаите, когато се извършва работа на заден фон, но не може да се продължи напред, докато не бъдат завършени изчисленията, да се зарежда на екрана зареждащата иконка по подразбиране. За различните версии на Android тя е различна на вид, въпреки това трябва да наподобява въртящ се кръг или аналогична картинка.

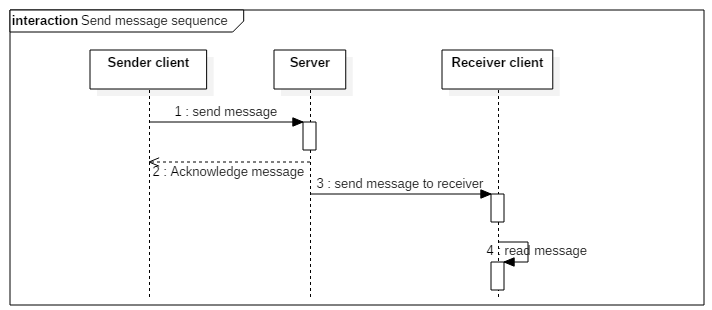
Друг начин да се информира потребителя е чрез т.н. toast съобщения. Те представляват малка тъмносива лента с кратък текст, която се появява в долния край на екрана над всички останали елементи, включително софтуерна клавиатура, като продължителността им на показване е 2-3 секунди. Този тип известия да се използва в случаите, когато потребителя иска да започне разговор с друг потребител, но въведения имейл е грешен или не съществува в базата от данни.

По – информиращи да са грешките при опит за вход в системата. Така например, когато потребителя се опита да влезе в системата, при непълен имейл, не само да се информира, но и да се пренесе фокуса върху текстово поле.

Аналогично да бъде съобщението за грешка при използване на твърде къса или твърде дълга парола.

**3.2 Архитектура на системата**

Основната задача на сървъра е да приема чат съобщения от клиента изпращач и да ги доставя на клиента получател. (Възможно е това да бъде един и същи клиент). Виж фигура № 3.5. за последователност на действията при изпащане на съобщение.



**Фигура 3.5. Диаграма на последователностите при изпращане на съобщение**

При получаване на съобщение от клиента, сървъра проверява първо какъв тип съобщение е. Според него се определя и какви проверки за валидност на данните да бъдат извършени. Най – честата заявка ще бъде лично съобщение към клиент. С цел повишаване на сигурността и намаляване възможността от спам, сървъра експлицитно изчислява идентификационния номер на клиента изпращач. Това се постига с помощта на сокетите. Така няма да се позволи на клиент X, да се представи за клиент Y и да изпрати съобщение на получател Z.

Друг фактор, който трябва да се вземе под внимание е, че задължително да се достави съобщението до получателя. За тази цел се сървъра проверява дали получателя е на онлайн в момента. Ако не е, съобщението към клиента се запазва в базата от данни за по нататъшно изпращане.

При вход на клиент, сървъра проверява в базата дали има неизпратени към него съобщение. Ако има, те биват изпратени. Така се гарантира, че всеки клиент, които се появи на линия ще получи съобщенията. Ако клиента никога повече не влезе в системата, с изтриване на акаунта му трябва да се изтрият и съобщенията към него.

Трябва да се информира клиента при неуспешен опит за изпращане на съобщение към сървъра. Потребителя да може да направи повторен опит за изпращане, като натисне върху балона със съобщение.

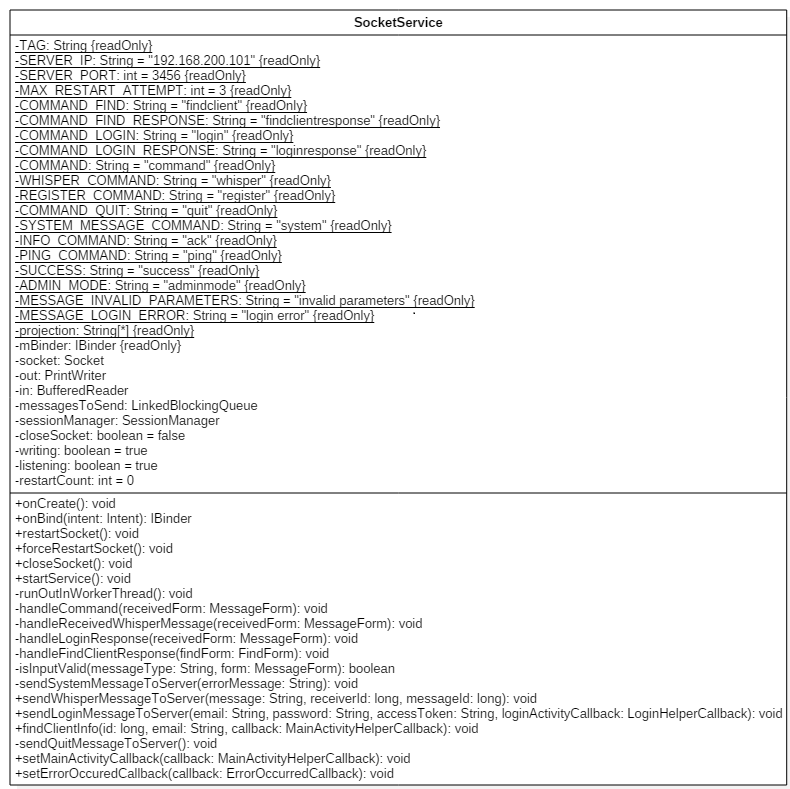
Системата трябва да поддържа изпращане на съобщения от клиент към поддръжка (call center), както и изпращане на съобщения от един акаунт на поддръжката към много клиенти.

Класа ServerImplementation трябва да имплементира интерфейса Server. Добавя се помощен клас ConnectionThread, който представлява конекция към един клиент. За да комуникират двата обекта се използват публични методи или Callback Listener. Виж фигура №6.



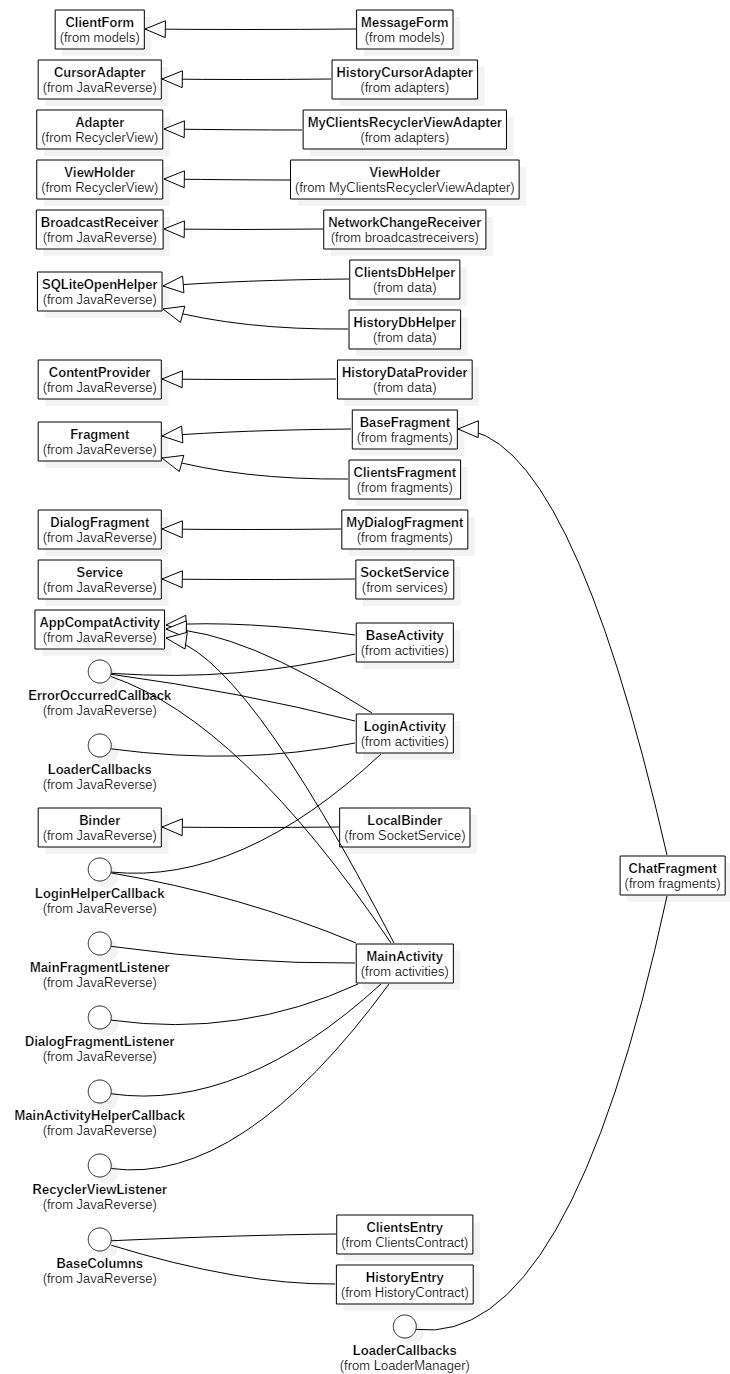
**Фигура 3.6. Клас диаграма на сървърна имплементация**

При клиента основния контролер се имплементира чрез класа MainActivity. Той наследява дефинирания от Android AppCompatActivity, като презаписва основните му методи. Този клас комуникира, както със сокет услугата, така и с различни фрагменти (View модели в MVC архитектурата). Пълния набор от методи, които съдържа класа са показани във фигура №3.7.



**Фигура 3.7. Клас диаграма на MainActivity**

Типовата архитектура в клиентското приложение е доста разнообразна. Голям набор от класовете дефинирани от Android Framework, трябва да бъдат наследени и презаписани техни методи с конкретна за приложението логика. В последствие наследяващите класове трябва да имплементират различни интерфейси, за да се осъществи междунишкова комуникация или възприемане на действие по дисплея от страна на потребителя. Пълната схема може да бъде видяна във фигура № 3.8.



**Фигура 3.8. Типова йерархия на класовете в клиентското приложение**

**3.3. Трансфер на информацията между клиент и сървър**

**3.3.1 TCP Сокети**

Основната функция на тази система е пренос на потребителски съобщения от и до сървъра. Това се извършва с помощта на TCP сокети. Чрез този тип сокети се осигурява тунелна връзка между клиент и сървър.

Сървъра не трябва да позволява на клиент да изпраща каквато и да е информация към друг клиент, преди той да се валидирал и неговата валидност не бъде потвърдена. При многократно получаване на съобщения различни от съобщения за вход или съобщения с невалидни данни, този потребител да се третира като атакуваш и да се прекъсне по нататъшна комуникация с него.

Понеже сокетите осигуряват тунелна връзка след като знаем идентичността на дадения потребител, не е нужно да изискваме неговите параметри за вход при последващи съобщения до прекъсване на връзката. Така ще се намали трафика и ще се увеличи производителността, понеже няма да има нужда да се прави проверка за валидация на клиента, както при HTTP заявките от тип POST, PUT, DELETE.

В случай че след валидация, клиента отново изпрати логин параметри, поради възможност те да не отговарят на параметрите, с които се влязъл в системата първия път, за по- голяма сигурност тази заявка да бъде игнорирана от сървъра.

Клиентското приложение трябва също да отвори сокет. Това да се осъществява още при стартиране на приложението. Поради ограничените ресурси от към батерия и / или интернет връзка (мобилни данни) сокета да се затваря автоматично при унищожаване на приложението. Поради факта, че всеки отворен сокет сървъра го асоциира само с един клиент, когато потребителя се отпише от приложението и желае да влезе с друг акаунт, да се рестартира сокета. В противен случай клиента никога няма да получи потвърждение за успешно влизане в системата, дори да са му валидни входните данни.

**3.3.2** **Протокол за комуникация на апликационно ниво**

Поради факта, че се използват TCP сокети, трябва да се дефинират правилата за трансфер на съобщения. Това се извършва с помощта на собствен протокол, който е базиран на форматиран чрез json текст.

Така модел обекти имплементирани на обектно- ориентиран език ще могат да се конвертират към json обекти, който преди изпращане ще бъдат преобразувани в низове. С това се постига независимост между клиент и сървър, които могат да бъдат написани при нужда на различни програмни езици. В json обектите променливите се запазват по метода ключ – стойност. Ключа трябва да бъде известен и на двете страни за да могат да преобразуват обекта, в случая, отново в Java обект.

Минималния набор от ключове, който са необходими за коректна работа са следните:

* command
* id
* email
* password
* accesstoken
* isAdmin
* messageId
* receiverId
* senderId

А за определяне на типа заявка, която трябва да се изпълни се определят следни командни стойности:

* login
* loginresponse
* findclient
* findclientresponse
* whisper
* register
* quit
* system (изпозлва се за системни съобщения)
* ack (info)

Понеже всяко съобщение е независимо от останалите и е критично важно да бъде доставено до крайната цел, трябва да има отговор, с който да се потвърждава приемането му от получателя. Това се осъществява чрез ключа “command” и нейната стойност – “ack”.

**3.4. База от данни**

**3.4.1. Сървърна база от данни**

Информацията от сървъра се съхранява на релационна база от данни (MqSQL). Нейната роля е да съхранява всички потребители и неуспешно изпратените съобщения.

* **Таблица на потребителите**

Потребителите се съхраняват в една обща таблица. Параметрите, които се пазят за тях са:

* идентификатор (id), който служи за основен идентификатор на таблицата, но също така и за идентификатор на клиента
* имейл, с който е регистриран потребителя
* парола за вход в системата
* низ за автентичност (access token)
* тип на акаунт – потребител или администратор

Само идентификатора е автоматично генериран от базата, докато останалите полета са отговорност на сървъра.

* **Таблица за съобщенията**

С цел оптимизация, да се запазват само съобщенията, които към момента на получаване от сървъра на са могли да бъдат предадени към крайния клиент. Параметрите, които се пазят за тях са:

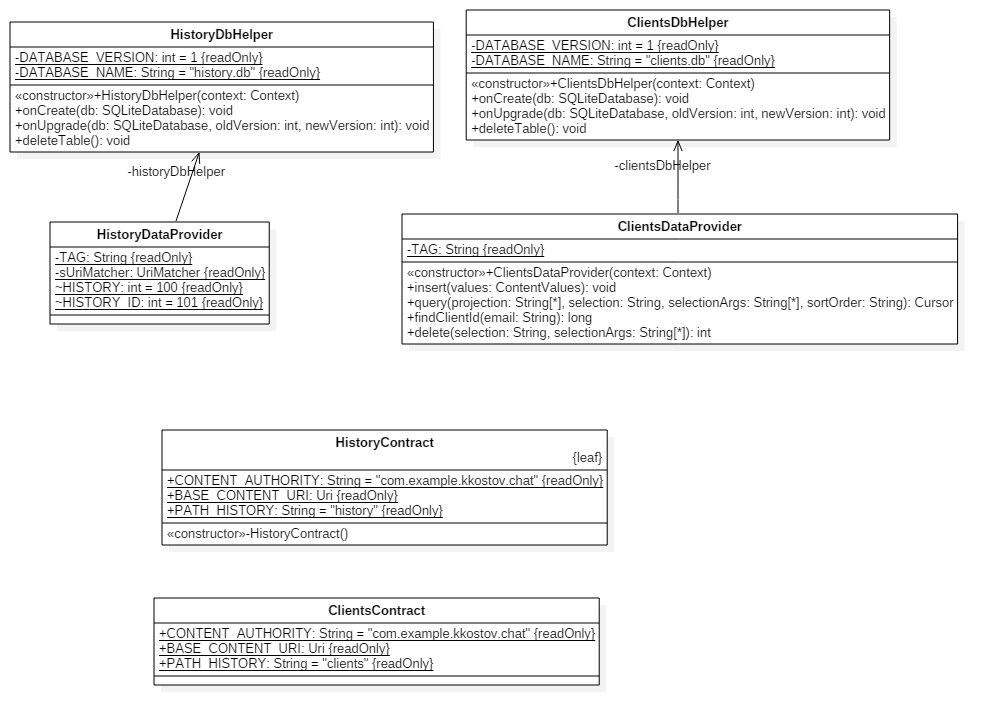
* идентификатор (id) - основен идентификатор на таблицата
* идентификатор на изпращача
* идентификатор на получателя
* съобщението, което трябва да бъде изпратено
* дата и час на приемане на съобщението от сървъра
* флаг за състоянието на съобщението дали е прието от получателя

Отново само основния идентификатор е отговорност на базата да го генерира автоматично, докато останалите полета се попълват от сървъра.

**3.4.2. Клиентска база от данни**

* **SQLite база от данни.**

Да се използва вградената във всички Андройд устройства релационна база от данни. Фигура № 3.9. описва програмно отношенията между система и база от данни.



**Фигура 3.9. Диаграма на класовете на SQLite база от данни**

1. **Таблица за съобщения**

Понеже данните, които ще се запаметяват са еднородни една таблица за съобщения е достатъчна, за да се запазват всички получени и изпратени съобщения. Полетата, които ще съдържа таблицата са следните:

* идентификационен номер (id) – това поле се явява основен идентификатор в таблицата и същевременно номер на съобщението.
* идентификатор на изпращача
* идентификатор на получателя
* съобщението, което е получено или изпратено
* флаг за указване дали съобщението е входящо или изходящо
* флаг за указване дали съобщението е прието от сървъра. (Когато клиента приема съобщение от сървъра, този флаг винаги е вдигнат).
* флаг за указване дали съобщението е прочетено от потребителя
* дата и час на приемане или изпращане на съобщението

Само идентификационния номер е автоматично генериран от базата от данни.

**2) Таблица за клиенти**

За да се намали дублирането на информация се запазват само по веднъж идентификаторите на клиентите и техните имейли, а при нужда да се прочетат данните се използва референция към тях. Полетата, които ще се съдържат са:

* идентификационен номер (id) – който е основен номер в таблицата
* идентификатор на клиента
* имейл на клиента
* **Плосък файл с данни**

За запазване на потребителските настройки и предпочитания да се използва системния файл shared preferences за запаметяване на ключ- стойност данни. Права за достъп до този файл да има само приложението собственик (създател) на файла.

**4. Реализиране на системата**

**4.1 Сървърна реализация**

Сървъра е имплементиран на езика Java. „Main“ метода на програмата приема като аргументи максималния брой на разрешени свързвания (връзки с клиенти) и номер на порт, който да бъде отворен за сокета. Самия сървър сокет е имплементиран чрез интерфейс, което създава допълнителна гъвкавост за имплементиране. Ползите от това са следните.

Първо - улеснява се възможността за промяна на бизнес логиката, без това да нарушава настоящата имплементация.

Второ – лесно може да бъде стартиран повече от един сървър на една машина с цел облекчаване на натовараемостта и пълноценното използване на ресурсите и.

Интерфейса има един метод, който трябва да бъде наследен. Чрез него се създава и отваря сървърен сокет (дефиниран в JRE 1.8). Той е оптимизиран да работи с множество клиенти. При започването му на работа, метода, в който се извика се блокира до осъществяването му на връзка с клиент. По – голяма производителност всеки нов клиент се създава в отделна нишка. При създаването на нишката се подава като параметър на конструктора callback listener.

В „run“ метода на новосъздадената нишката, се отварят входа и изхода за комуникация с клиента. Входа и изхода се създават в две отделни нишки, понеже имплементираните операции за изход са блокиращи.

// Open send and receive streams to client.

out = **new** PrintWriter(clientSocket.getOutputStream(), **true**);

in = **new** BufferedReader(**new**

InputStreamReader(clientSocket.getInputStream()));

isConnected = **true**;

runOutInWorkerThread();

където „runOutInWorkerTherad()” e ,

Thread outThread = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

// Send message to server

**while** (isConnected) {

**try** {

String msg = messagesToSend.take();

out.println(msg);

} **catch** …

}

outThread.start();

“messagesToSend” е обект от тип LinkedBlockingQueue. Чрез блокиране на нишката не се изразходва излишно процесорно време за проверка на опашката дали има съобщения за изпращане.

В случай че клиента прекрати връзката, се затваря сокета при сървъра, а след това използваните нишки преустановяват работа и биват унищожени, за да освободят ресурси.

За разлика от Web сокетите, където след предаване на информацията връзката се прекъсва, тук тя остава отворена. При осъществяване на връзка между сървър и клиент, сървъра чака за удостоверяващи идентичността на клиента данни, изпратени във предварително дефинирания формат. В конкретната имплементация е зададен максимален брой от 10 опита за вход. След това се приема, че клиента е атакуваш и сървъра затваря сокета от своята страна. При приемане на валидни параметри, се извиква чрез listener-a метод за вход на клиент, като извикващата нишка чака отговора на извиканата. След получаване на отговор се проверява флага за валидация на клиента. Ако опита за вход не е бил успешен отново се чака повторен опит, а ако е бил, се продължава нататък, като се дава възможност да бъдат изпълнявани и други типове заявки от клиента.

Заявките, които се разпознават от сървъра са:

* Login
* Register
* Whisper
* Find client
* Quit

При получаване на някои от гореизброените, след проверка за валидност на параметрите се извиква чрез listener-a съответния метод.

За изпращането на по- малко мета информация заявките (входящи и изходящи) се форматират като Json и след това се изпращат като низове. За тази цел има няколко помощни метода, които чрез overloading приемат съществените данни, а мета информацията я добавят имплицитно. Пример да това е добавянето на командната стойност „whisper“ при изпращане на лично съобщение.

Интерфейса, който се използва за комуникация между клиентките нишки и основата нишка има следните методи:

* Register client
* Login client
* Find client in db
* Does user exists
* Disconnect unvalidated client
* Disconnect validated client
* Send message to receiver

Основната нишка (в която се съдържа server socket обекта) следи клиентските нишки, като ги дели да две групи: не валидирани и валидирани.

Първите се съхраняват в синхронизиран списък, понеже се предполага във всеки един момент от времето ще бъдат значително по- малко от валидираните. Валидираните от своя стана се съхраняват в синхронизиран map за по – бързо търсене, като ключа, по който се търси е клиентския номер, понеже за него е сигурно, че е уникален.

Основната нишка override-ва методите на интерфейса от клиентската нишка. Така когато се извика метод от клиентската нишка, тя се обработва в контекста на основната. Например “loginClient(ClientForm form, ConnectionThread connectionThread)”.

// Validate required fields

// …

ClientDAO clientDao = **new** ClientDAOImpl();

ClientForm clientFromDb = **null**;

// Find client in db by what is provided

**if** (form.getId() > 0) {

clientFromDb = clientDao.findClient(form.getId());

} **else** {

clientFromDb = clientDao.findClient(form.getEmail());

}

// If client exists

**if** (clientFromDb != **null** && clientFromDb.getId() > 0) {

// Validate credentials

**if** ((clientFromDb.getEmail().equals(form.getEmail()) &&

clientFromDb.getPassword().equals(form.getPassword())) || clientFromDb.getAccessToken().equals(form.getAccessToken())) {

// Generate new access token for client and save it

String accessToken = generateAccessToken();

clientFromDb.setAccessToken(accessToken);

**long** rowId = clientDao.update(clientFromDb);

**if** (rowId <= 0) {

printMessageOnConsole("Error saving client access token");

// Client is valid raise his flag in order to know and

// notify him.

connectionThread.setValidated(**false**);

connectionThread.setLoginResponseReady(**true**);

connectionThread.notify();

connectionThread.

sendErrorMessageToMyself(***MESSAGE\_LOGIN\_ERROR***);

**return**;

}

// Find client in unvalidated list

**int** index = findIndexInUnvalidatedConnections(connectionThread);

**if** (index < 0) {

printMessageOnConsole("Error finding client in unvalidated list");

connectionThread.closeConnection(); **return**;

}

// Move client to valid connections

**long** clientId = clientFromDb.getId();

ConnectionThread connection = unvalidatedConnections.remove(index);

connection.setUserId(clientId);

validConnections.put(clientId, connection);

// Client is valid raise his flag in order to know and // notify him.

//...

// Send response to sender

MessageForm messageForm = **new** MessageForm();

messageForm.setCommand(***COMMAND\_LOGIN\_RESPONSE***);

messageForm.setIsAdmin(clientFromDb.getIsAdmin());

messageForm.setAccessToken(accessToken);

messageForm.setId(clientId);

connectionThread.sendMessageToMyself(messageForm);

// If there are unsent messages to client,

// send them to him now

sendAllUnsendMessagesToReceiver(connectionThread);

printMessageOnConsole("Client with id " + clientId + " has loged in");

printMessageOnConsole("Number of unvalidated clients is: " + unvalidatedConnections.size()

+ " and valid is: " + validConnections.size());

} **else** {

// Client is valid raise his flag in order to know and // notify him.

// …

// Send response to sender

MessageForm messageForm = **new** MessageForm();

messageForm.setCommand(***COMMAND\_LOGIN\_RESPONSE***);

messageForm.setMessage(***MESSAGE\_INVALID\_PARAMETERS***);

messageForm.setAccessToken(**null**);

messageForm.setId(0);

connectionThread.sendMessageToMyself(messageForm);

printMessageOnConsole("Client with thread id " + connectionThread.getId() + " has invalid credentials");

}

} **else** {

printMessageOnConsole("Client with thread id " + connectionThread.getId() + " does not exist in db");

// Client is valid raise his flag in order to know and notifyhim

// …

// Send response to sender

MessageForm messageForm = **new** MessageForm();

messageForm.setCommand(***COMMAND\_LOGIN\_RESPONSE***);

messageForm.setMessage(***MESSAGE\_INVALID\_PARAMETERS***);

messageForm.setAccessToken(**null**);

messageForm.setId(0);

connectionThread.sendMessageToMyself(messageForm);

} }

Логин заявката първо проверява дали всички задължителни полета имат стойност. След това се търси клиента в базата от данни по имейл или по идентификационния номер. Ако не бъде открит се изпраща към изпращача адекватен отговор. В случай че съществува такъв клиент се сравняват имейла и паролата или низа за валидация за съответствие. При съвпадение се генерира нов низ за валидация, който първо се запазва в базата от данни преди да бъде изпратен към клиента. На клиентката нишка се вдига флага, че е валидирана, след което се събужда.

Метода за намиране на клиент, търси в базата от данни и при успех връща информация за клиента, ако не връща „нулев клиент“.

Заявката за изпращане на лично съобщение е малко по – сложна. Първоначално тя се обработва от клиентската нишка, която проверява дали всички задължителни полета имат стойности. След това получателя се търси в базата от данни, дали съществува такъв клиент. Ако няма се връща негативен отговор. Ако има се изпраща “ack” съобщение към изпращача, за да се информира, че заявката му е успешно приета. После работата се поема от основната нишка.

**private** **void** handleWhisperCommand(String messageFromClient) {

Gson json = **new** Gson();

MessageForm receivedForm = json.fromJson(messageFromClient, MessageForm.**class**);

// Send message only if it is valid. If not inform the sender.

**if** (!isInputValid(***WHISPER\_COMMAND***, receivedForm)) {

sendErrorMessageToMyself("Invalid parameters");

**return**;

}

**if** (listener.doesUserExists(receivedForm.getReceiverId())) {

// The server sets the client id of the sender

// explicitly, not the client.

receivedForm.setSenderId(**this**.userId);

// Send info msg to server to know that his msg was received.

MessageForm infoForm = **new** MessageForm();

infoForm.setCommand(***INFO\_COMMAND***);

infoForm.setMessageId(receivedForm.getMessageId());

infoForm.setMessage(***SUCCESS***);

sendMessageToMyself(infoForm);

listener.sendMessageToReceiver(receivedForm);

} **else** {

sendErrorMessageToMyself("Receiver does not exist");

}

}

Тук има раздвояване на логиката. Първия вариант е оптимистичния. При него търсим дали клиента е на онлайн и ако е, директно изпращаме съобщението. Ако не е, го запаметяваме в базата от данни.

@Override

**public** **void** sendMessageToReceiver(MessageForm form) {

// If receiver is online send now the message else store in db and

// wait to come online.

ConnectionThread connectionThread =

validConnections.get(form.getReceiverId());

**if** (connectionThread != **null**) {

sendMessageToOnlineReceiver(form);

} **else** {

// Store the message to send when he comes online

storeUnsentMessageToReceiver(form);

}

}

За връзка към базата се използва DAO дизайн шаблона. Чрез него си гарантираме независимост между програмата и базата от данни. Текущата имплементация е чрез JDBC. Имплементирани са CRUD операциите, като преди всяка операция се отваря нова връзка към базата, а след завършването връзката се затваря.

За обмен между базата и сървъра се използва DTO обекти, а между сървър и клиент различни форми, които преди изпращане се конвертират към низове.

Изходящите заявки, който сървъра изпраща биват:

* Whisper to client
* Ack
* Login response
* Find client response

**4.2 Клиентска реализация**

Клиентското приложение също трябва да създаде и отвори сокет. Понеже операцията е дълга и продължителна се създава собствен Bound service, който наследява Service. За да се разкачи обаче от UI нишката, е нужно той да работи в отделна нишка. Това се случва в метода startService(), където се създават две нишки. Една за вход и една за изход. Подобно както при сървъра при приемане на съобщение първо се обработва от какъв тип е, след което се преминава към неговото изпълнение.

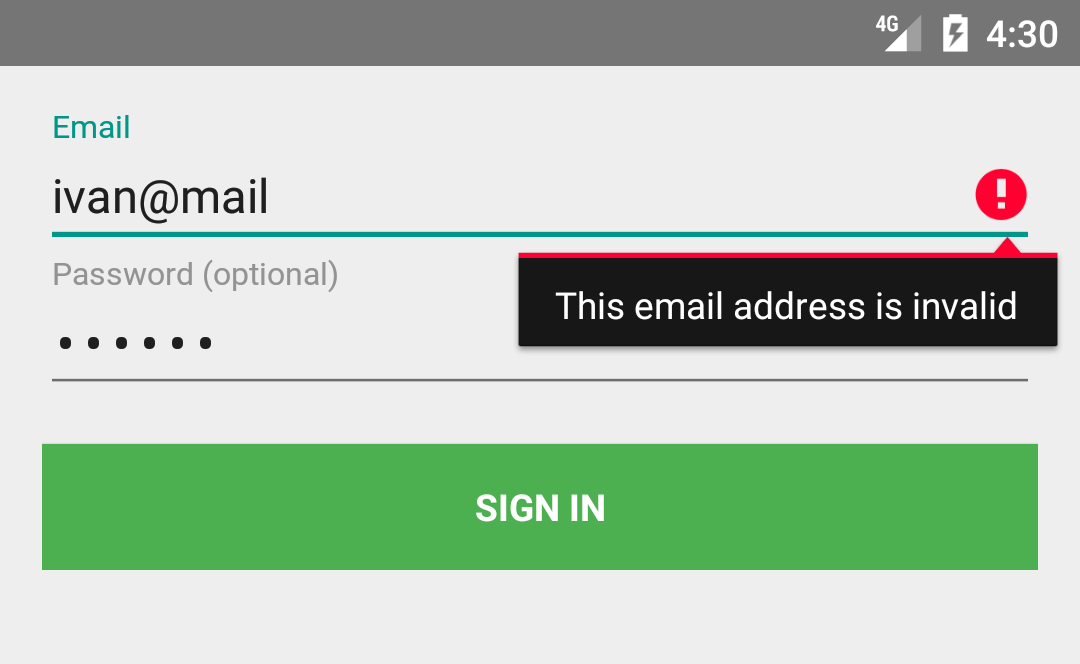
Така например при обработка на лично съобщение и след проверка на валидността, се подготвя информацията за записване в локалната база от данни. Ако данни за изпращача не се съдържат в локалната база, се прави запитване до сървъра да ни даде информация за клиента. След това се записва както съобщението така и данните за клиента изпращач.

За разлика от сървъра обаче при спиране на работата на service-a, автоматично се изпраща едно последно съобщение към сървъра, което е за изход. Не е задължително, но е препоръчително да се изпрати такова съобщение, за да може да бъде нормално отписан от списъка с клиенти на линия.

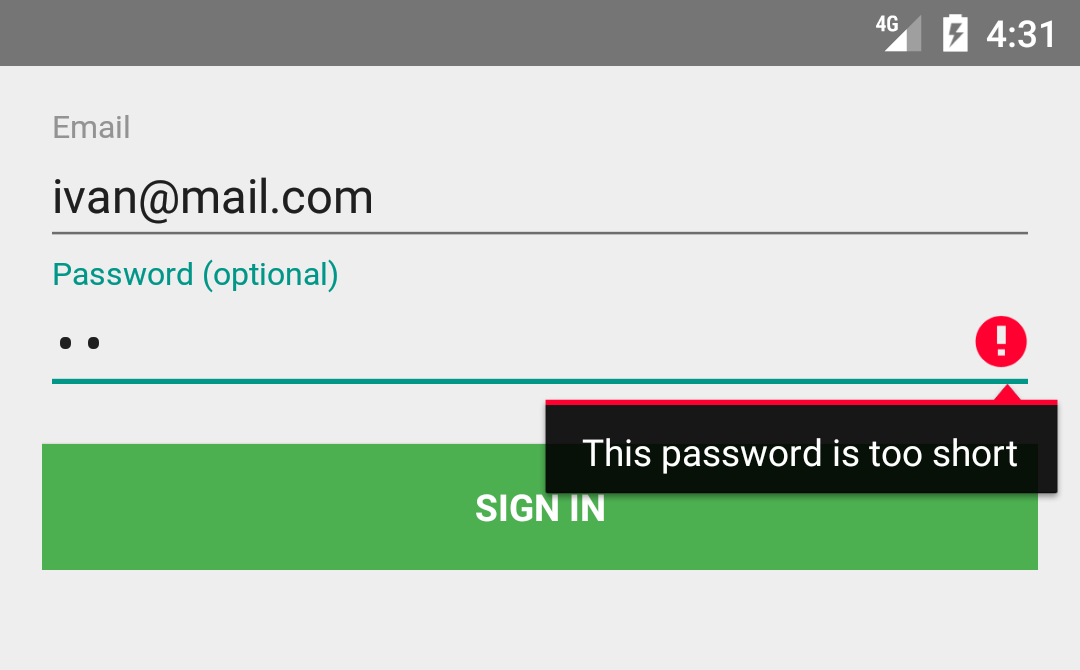
Service-a комуникира с UI нишката чрез множество callback listeners.

Приложението има два основни класа наречени LoginActivity и MainActivity, който се явяват контролери в MVC модела. При стартиране на приложението, се зарежда LoginActivity. В първия от множеството методи, които се изпълняват преди да стане видим екрана се създава SocketService, ако вече не е създаден и се свързва към него. Следват различни инициализации на визуални елементи, както и връзка към базата от данни.

Когато service-a е готов за работа, при наличието на валидиращ низ (access token) се прави заявка към сървъра и се чака неговия отговор. Ако е положителен директно се зарежда основния екран (MainActivity) . Ако не е или няма запаметен такъв низ се показва на екрана логин формата. При попълването на последната се окаже, че някои от полетата не отговарят на предусловията, се показва адекватно съобщение, до полето с проблем.



**Фигура 4.1. Съобщение за невалиден имейл**



**Фигура 4.2. Съобщение за твърде кратка парола**

За улеснение на потребителя, при одобрен достъп до контактите в телефона, може да бъде избран имейл от вече запомнените.

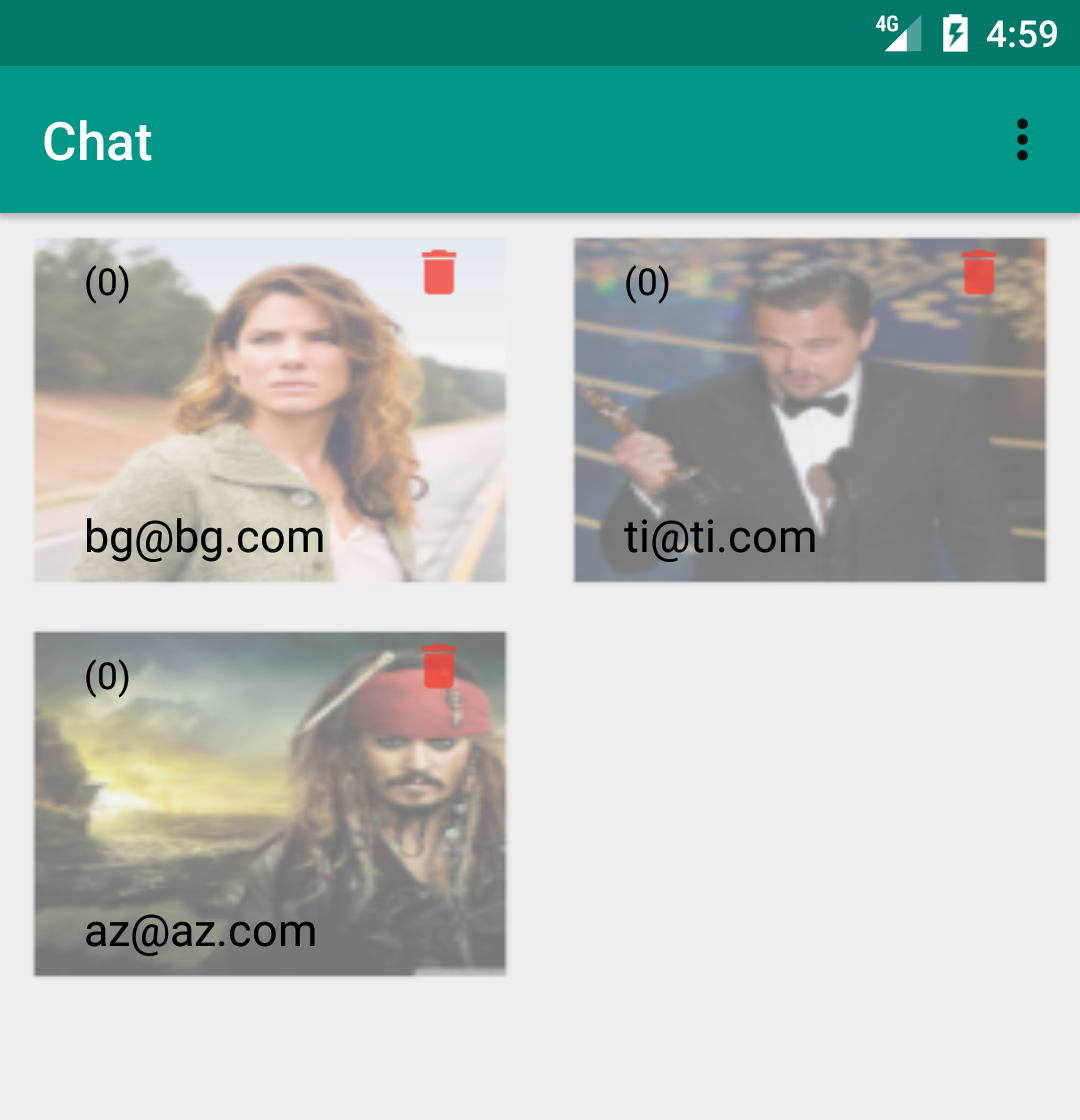
LoginActivity имплементира няколко интерфейса за да се осигури комуникация между UI нишката и работните нишки в Service-a. Особеност на Android е, че всички промени по UI елементите са възможни единствено от основната нишка.

LoginActivity.**this**.runOnUiThread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **if** (show) {  
 LoginActivity.**this**.showProgress(**false**);

// …   
 }   
 }  
});

MainActivity-то също се обвързва със сокет услугата (SoketService) и я създава , ако вече не е създадена. При стартиране на MainActivity, според типа на акаунта могат да се заредят два различни екрана.

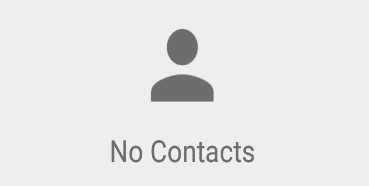
Разликата между обикновения потребителки акаунт и администраторския е, че администраторския има един екран в повече. Чрез него той може да комуникира, с който и да е клиент, докато на потребителския профил – само е предварително дефиниран. Ако акаунта е администраторски, се зарежда списък от потребители, с които той може да води комуникация. Погледни фигура № 4.3.



**Фигура 4.3. Екрана снимка на списък с клиенти**

Списъка е под формата на решетка с 2 колони. Той е имплементиран чрез наследяване на класа RecyclerViewAdapter. Ползата при използване на този тип адаптер е, че се зареждат в паметта само броя елементите, които се показват на екрана плюс още 4 елемента. По два над и под екрана в случай, че потребителя скролира нагоре или надолу. Елементите, които са излезли извън видимата част на дисплея вместо да бъдат унищожени се преизползват, като само се подменя информацията, която визуализират. Това се постига чрез презаписване на метода дефиниран от RecyclerViewAdapter-a – onBindViewHilder(final ViewHolder holder, int position). В този метод са изпозват слушатели (listeners), за да се засече интерактивност от стана на потребителя. При промяна на елементите в списъка, за да се променят и показаните елементи на екрана, е създаден помощен метод, който бива извикван от ClientsFragment ( обект, който се явява View в MVC модела). С него се зачиства списъка на адаптера, записват се новите елементи и след това се информират всички наблюдатели чрез notifyDateSetChanged(), че данните са се променили.

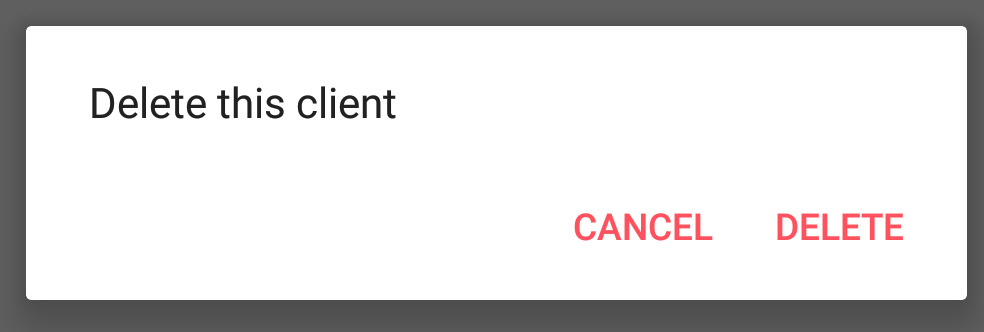
Ако няма налични клиенти, които да се визуализират в центъра на екрана се зарежда иконка по подразбиране за контакти дефинирана от Google и надпис под него че няма запаметени контакти. Виж фигура № 4.4.



**Фигура 4.4. Изображение за липса на контакти**

При получаване не съобщение от клиент, който не е в списъка автоматично се изтегля информацията за него от сървъра и се визуализира в горния ляв ъгъл на екрана. А броя не непрочетени съобщения е изписан в скоби в горния ляв ъгъл на снимката му.

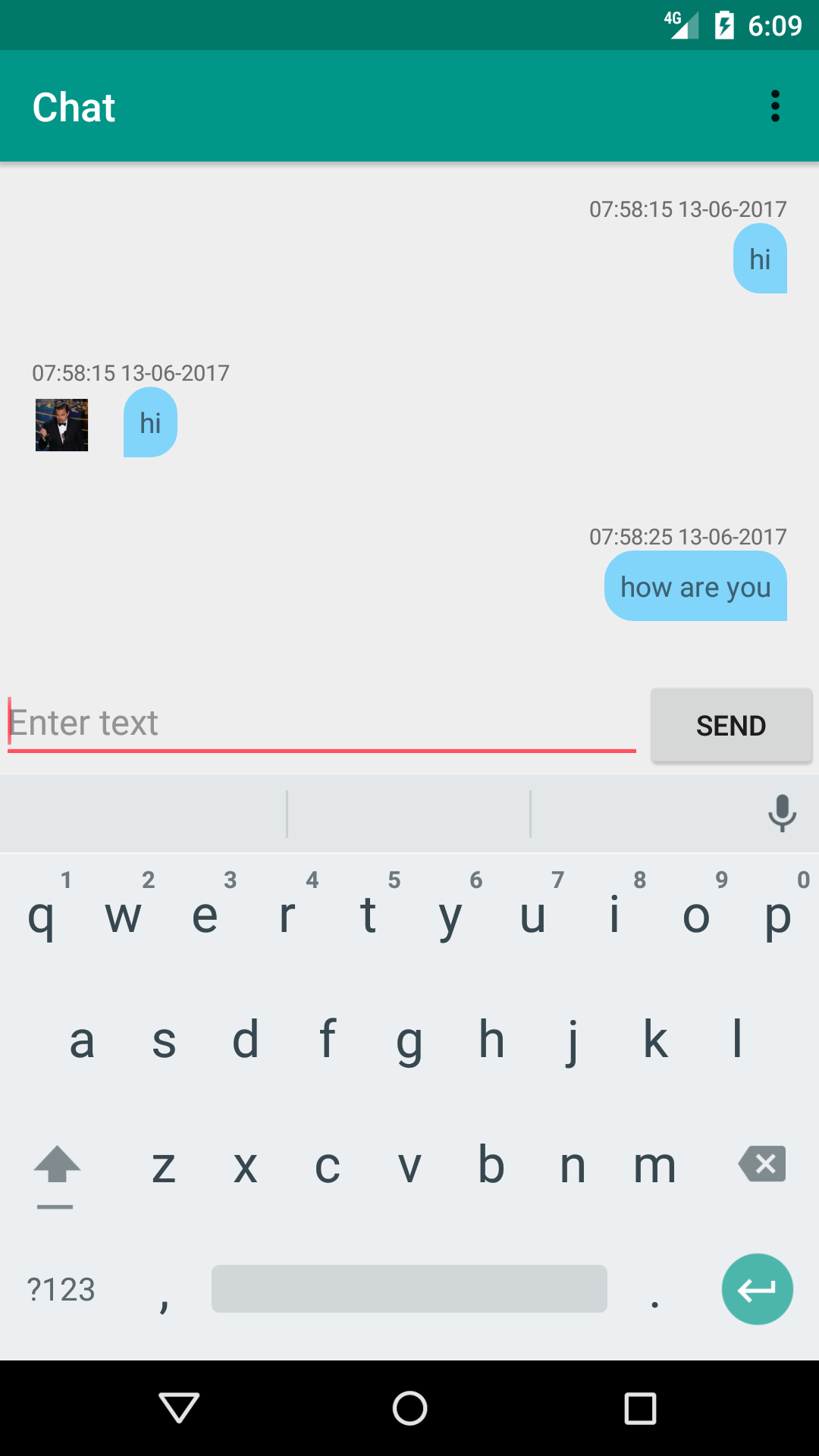
За всеки клиент потребителя може да избере между две опции. Първата е натискане върху бутона за изтриване ( кошчето за боклук), при което се появява малък диалогов прозорец в центъра на екрана. Виж фигура № 4.5.



**Фигура 4.5. Диалогов прозорец за изтриване на клиент**

Понеже конвенцията на Google изисква бутоните да не са едносрични отговори (Yes / No), са изписани действията, които могат да бъдат извършени. Позитивния отговор изтрива клиента от локалната база и го премахва от екрана на дисплея, като останалите елементи се пренареждат. При избор на негативен отговор или премахване на диалога ( чрез натискане извън диалоговия прозорец), прозореца се затваря без да се извърши допълнително действие.

Втория и по – често срещан случай е потребителя да натисне върху, която и да е част от аватара на клиента. Това задейства отварянето на нов прозорец с размерите на целия дисплей, в който може да се води комуникация с избрания клиент. Погледни фигура № 4.6.



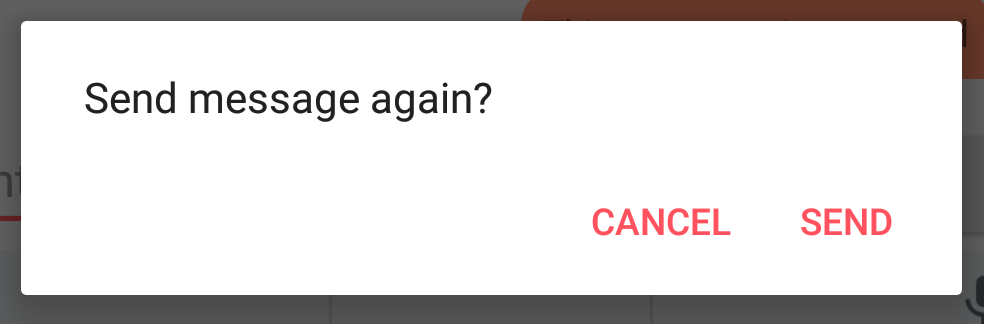
**Фигура 4.6. Екранна снимка на чат**

Съобщенията от потребителя се подреждат в дясната част на екрана, докато получените – в лявата. За да не се получи объркване от лявата стана на входящото съобщение се изобразява снимката на потребителя, който го е написал.

От имплементационна гледна точка това е постигнато чрез наследяването на клас CursorAdapter. Подобно при RecyclerAdapter-a, тук пак има статичен клас ViewHolder, в които се съдържат визуалните елементи и bindView(View view, final Context context, Cursor cursor) за поставяне на данни във UI елементите. Разликата е в това, че когато базата от данни, от която се зарежда информацията за визуализиране е според изискванията на Google ( наименованието на колоната за първичен ключе е „\_id”), и имаме имплементиран клас ContentProvicer отговарящ конкретно за чат историята, можем да се предостави на Android framework-а да се грижи за актуализирането на списъка при промяна на някое от полетата в базата. Като допълнение е указано в xml-a на ListView-то да подреждат елементите от дъното чрез атрибута „android:stackFromBottom=’true’”.

За да имат текстовите съобщения форма на балон се прави специален фон за входящи и изходящи съобщения. Фона е под формата на правоъгълник с 3 заоблени ъгъла. Заоблянето е със радиус 15dp (dencity pixels). Единствения ъгъл, който не се заобля е долу в дясно за изходящи или долу в ляво за входящи съобщения.

Когато поради една или друга причина не бъде доставено съобщението до сървъра чрез натискане и задържане върху съобщението, се показва нов диалогов прозорец подканващ потребителя да го изпрати наново.



**Фигура 4.7. Диалогов прозорец за повторно изпращане на съобщение**

При проблем при свързването към сървъра или няма интернет връзка се изписва информационно съобщение в горния край на дисплея. Погледни фигура № 4.8.



**Фигура 4.8. Информираща лента за проблем при свързване със сървъра**

При възстановяване на връзката със сървъра това съобщение се скрива. Това се постига чрез т.нар. broadcast receivers. Чрез тях клиентското приложение се „абонира“ за да получи известие от Android системата, при отсъствие на интернет връзка. Конкретната имплементация при получаване на известие, чрез callback listener съобщава на Activity-то, което е видимо в момента на получаване, да покаже на екрана съобщението.

Засега приложението поддържа само един език – английски. Лесно може да бъден добавен втори – български, понеже стринговите ресурси са изнесени отделно от бизнес логиката на приложението във външен xml файл.

**4.3 Апликационен протокол за комуникация**

Чрез протокола са имплементирани следните заявки, които сървъра разпознава.

1. Лично съобщение до сървъра (Whisper to server)

**In message**  **Out message**

- command: whisper -command: ack

- message: msg -message: success

- messageId: -messageId:

- receiverId:

1. Съобщение за вход (login)

**In message**  **Out message**

* command: login - command: loginresponse
* accessToken: - accessToken:
* email: - id:
* password: - isAdmin: 0 или 1

1. Съобщение за намиране на клиент по имейл или клиентски идентификатор (id) (Find client).

**In message**  **Out message**

* command: findclient -command: findclientresponse
* email: -email:
* id: -id:

Ако търсения клиент не бъде открит в базата от данни, като резултат за “id” се връша „-1“ и като имейл – “null”.

1. Лично съобщение от сървъра до клиента

**In message**  **Out message**

- command: whisper

- senderId:

- receiverId:

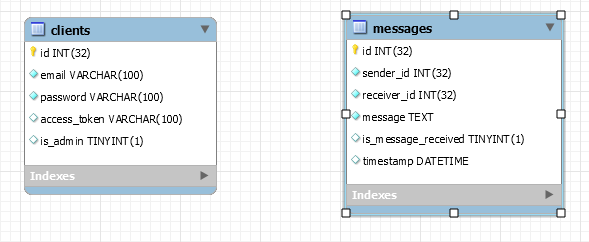
- message:

Командните стойности са константи, които са предварително дефинирани. Останалите стойности са променливи и се попълват динамично от сървъра.

**4.4 Бази от** **данни**

**4.4.1 Сървърна база**

Базата, която се използва от сървъра е MySQL. Състой се от 2 таблици – клиенти (clients) и съобщения (messages).



**Фигура 4.9. Схема на таблиците ползвани от сървъра**

В таблицата клиенти, стойността на id се генерирана от базата, докато останалите полета са отговорност на използващата програма. Колоните имейл и парола се с максимална дължина от 100 символа, като е задължително те да бъдат въведени. При използването на тип VARCHAR, неизползваните стойности за символите не заемат пространство на файловата система. Полето “access\_token” е с стойност по подразбиране “null”, а флага от тип число е със стойност 0.

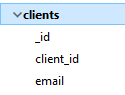
В таблицата съобщения, стойността на id се генерира от базата. Същото важи и за полето „timestamp”, на което се присвоява датата и часа от системата в момента на създаване на нов запис. Полетатa „sender\_id”, „receiver\_id” и “message” са задължителни за въвеждане, понеже без някое от тях информацията не е достатъчно уникална и не може да бъде използвана по предназначение. Полето “is\_message\_received” е флаг, с числова стойност по подразбиране 0.

**4.4.2 Клиентска база от данни**

Базата от данни, която се съхранява при клиента е от 2 типа. Първата е вградената релационна база от данни – SQLite, а втората, отново вградена, плосък файл от тип ключ – стойност.

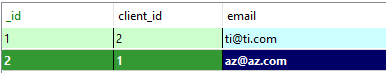
* **SQLite база**

Базата съдържа две таблици – клиенти (clients) и история (history).

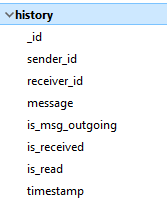


**Фигура 4.10. Схема на таблицата ползвана от клиента**

Клиентската таблица е малка по размер, като в нея се съхраняват само информацията за контактите, с които потребителя води комуникация (виж фигура № 4.10). Тази информация бива идентификационен номер на клиента (client\_id) и неговия имейл (email). Таблицата съдържа и колона с идентификатори на редовете ( primary key), които се генерират от базата.



**Фигура 4.11. Примерни записи в таблица с клиенти**

****

**Фигура 4.12. Схема на таблица чат история ползвана от клиента**

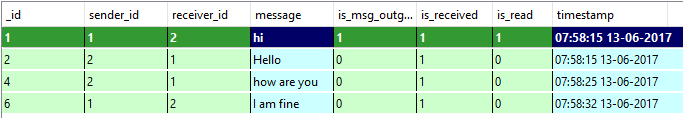
Историята на чата се запазва в тази таблица. Тя има идентификатор (primary key), който се генерира от базата. Колоните “sender\_id” и “receiver\_id” са числови стойности, като не могат да бъдат оставени празни. Погледни фигура № 4.12.

Важно е да се отбележи, че от съображения за сигурност сървъра определя автоматично идентификатора на изпращача. Това се дължи на факта, че потребител с root права на телефона, може да редактира локалната таблица и да изпраща неограничено спам съобщения, прикрит като друг потребител. Или атакуващ да прихване трафика между устройството и сървъра и да промени от чие име се изпраща съобщението.

За да може да бъде направена хронологична справка на комуникацията между два клиента, в локалната база от данни в полето “sender\_id” се записва идентификатора на потребителя. Не се предвижда, идентификатора да бъде променян в бъдеще, но ако това се случи, с приоритет е таблицата, съхранявана при сървъра, което би довело до липса на история на съобщенията.

Колоната „message” е от тип текст и също не може да има стойност null. Флаговото поле “is\_message\_outgoing” – показва дали съобщението е входящо или изходящо, със стойност по подразбиране 0, което означава че е входящо. Полето “is\_received” се отнася до това дали съобщението е прието от сървъра, а не от получателя. В случай че съобщението е изходящи за клиента, този флаг винаги е вдигнат (стойността е 1). Колоната “is\_read” e помощна и се използва за определяне на броя получени, но непрочетени съобщения от потребителя. Когато потребителя е писателя на съобщението този флаг в вдигнат ( стойността е 1). В последната колона се съхранява времето на получаване или изпращане на съобщение. Стойността е във формата час:минута:секунда ден, месец, година и се въвежда от клиентското приложение.

При извършване на заявка от тип справка, върнатия резултат е от тип Cursor.



**Фигура 4.13. Примерни записи в таблица история**

За всяка таблица е създаден и договор клас (Contract), в който са описани пътищата до таблицата и нейните редове, както и наименованията на всяка колона. За да може да използваме функционалността, която ни предоставя Android framework-a основния идентификатор на таблицата (primary key) трябва да бъде наименуван „\_id”.

* **Shared preferences**

Потребителските настройки и информация се запазват в системата чрез “SharedPreferences” интерфейса. Те биват: идентификационен номер на клиента, неговия имейл, ключ за удостоверяване и флаг дали акаунта на клиента е администраторски. Само приложението създател има права за достъп.

1. **Тестове и резултати**

По време на реализацията на проекта бяха правени множество от тестове. В резултат, на което бяха открити и отстранени различни проблеми, включително такива с критична важност. Изброени са някой по- сериозни изключения, които бяха прихванати:

* NullPointerException – това беше един от най- често срещаните проблеми. Това се дължеше на факта, че и клиента, и сървъра са многонишкови процеси извършващи меджунишкова комуникация. Важно е да се инстанцира даден обект преди да бъде използван или да бъде извикан някой от публичните му методи. Някои от изключенията могат да бъдат избегнати чрез проверка дали обекта е “null” преди да бъде извикан негов метод. Това е решение за проблема предимно в случайте, когато не може да се задължи ползвателя ( разработчика) да инстанцира обекта и да го остави изцяло във валидно състояние. Такъв е случая със SocketService-a в Android приложението, където трябва да се уведоми Activity-то за настъпване на такова събитие. Но според жизнения цикъл на този тип service, той може да продължи работа си дори след като Activity-то бъде унищожено.
* CalledFromWrongThreadException: Only the original thread that created a view hierarchy can touch its views. – Тази грешка се получава, когато нишка – работник чрез callback listener се опита да промени някой от UI елементите. Дори имплементирането на интерфейса за бъде в Activity клас ( Activity класовете работят на основната нишка наричана още UI нишка), метода се изпълнявано в контекста на нишката работник, откъдето идва и грешката. Има два начина за отстраняване на проблема. Единия е като се взема контекста на activity-то, например “MyActivity.this.updateView()”. Другия и по- сигурен начин е чрез изричното оповестяване за работа на основната нишка с помощта на runnable обект – “MyActivity.this.runOnUIThread(new Runnable () { …}”
* NetworkOnUIThreadException – Това е обратен на вариант на предходното изключение. В него всички дейности, които извършват работа по интернет, трябва да работят в отделна нишка. В по – предните версии на Android се е създавало само предупреждение, докато в най-новите - Exception.
* SQLiteException - причините за този тип грешки са много на брой. Някой от оправените включваха от прости синтактични грешки като сгрешено име на колона до невалиден SQL код.
* IOException – е често срещана грешка и при клиента, и при сървъра. Тя задължително трябва да бъде хванат и обработена, за да може да продължи работата на програмата нормално. Получава се, някои от буферите за вход и изход на сокета бъде затворен. По – рядко срещано е, когато сокета не може да бъде инициализиран. В случай че бъде прихваната такава грешка, входа и изхода трябва да се затворят, ако вече не са, а след това и самия сокет.

Отстранени бяха и няколко логически грешки:

Първата грешка се появяваше в клиентското приложение. Поради завишаване на сигурността, идентификационния номер на изпращача се определя не от клиента, а от сървъра. Поради тази причина, когато се запаметяваше изходящо съобщение в локалната база при клиента, номера на изпращача се инициализираше със стойността по подразбиране „0“. Това не създаваше проблем, до момента, в който потребителя не направи преглед на чат хронологията. Реално всички съобщения, които той е изпратил не могат да бъдат визуализирани, понеже не отговарят на изискванията. Сега при запис на съобщение в локалната база се записва и идентификатора на клиента.

Друга грешка, която беше открита е, че след критична промяна по бизнес логиката от страна на клиента, вече се позволяваше на потребител да търси друг по идентификатор или имейл. Метода при сървъра обработващ тази заявка, не беше актуализиран да позволи такова търсене, което водеше винаги до грешка.

Наблюдаваше се увисване на работа на сървъра при опит за логин с невалидни параметри, понеже нишката с отворен сокет към клиента и беше указано да чака, като в определени ситуации, никога не беше събуждана. Като допълнение същата нишка, „заспиваше“, точно преди да отвори изхода към клиента, което доведе до затруднение при откриването на проблема. Проблема беше отстранен, когато при всички случаи се събуждаше нишката, независимо дали потребителя е успял да влезе в системата или не.

Друг проблем, който беше причинен от човешка грешка, беше разменянето на идентификаторите на изпащача и получателя, при обработка на съобщението от сървъра. След коригиране и присвояване на правилните стойности, системата работи както е проектирана.

Сървърният софтуер беше инсталиран на машина с хардуерни параметри CPU 4x2.4GHz и 8 GB RAM. Системата беше тествана под максимално натоварване. Бяха създадени 2000 клиента и един сървър, които едновременно комуникираха помежду си, като работата се извършваше с добра производителност. Създаването на повече от горепосочения брой клиенти, надхвърляше хардуерните възможности на компютъра.

Голям набор от тестове бяха извършени. Те служат, за да потвърдят, че системата е устойчива дори, когато не протекат по предполагаемия начин. Така например беше проверена системата, при опит за вход с невалидни данни, при липса на интернет връзка, при липса на сървър, който да обработи заявките и много други. Намерените проблеми бяха отстранени, за да се гарантира комфорт на работата на потребителя.

Системата беше тествана против изпращане на празни съобщения, както и изпращането на много големи такива. Тестовете преминаха успешно.

Към момента не се наблюдават проблеми, нито при клиентския софтуер, нито при сървърния.

1. **Заключение**

Анализиран беше проблема за комуникация между клиент и компания. Бизнес логиката намери адекватно решение. Проектирана система дава софтуерно решение за справяне с него. Настоящата реализация дава имплементация, с която да се покаже, че е възможно да бъде направен такъв продукт. Той изцяло отговаря на предварително дефинирани изисквания и е напълно функциониращ. Тестовете показват, че софтуера е устойчив на сривове и е надежден.

**Моменти за бъдещо развитие:**

* Развитието на машинното самообучение води до още една огромна положителна промяна. То може да се използва като посредник между потребители и поддръжка в call център. Понеже човешката психология е настроена негативно към комуникиране с машина, тя трябва да бъде сведена до минимум. Същевременно с това машината или по- точно алгоритъма за машинно самообучение трябва да може да отговори на някои основни въпроси свързани с клиента и да предаде тази информация към поддръжката в централата на компанията. Така компанията ще е запозната, кой клиент точно се обажда и най- вероятните причини за неговото обаждане, още преди той да се е свързал с тях.
* Като допълнение може да се добави възможността за файлов трансфер. Изпращане на снимки или видеа към компания, може да бъде много по-описателно. В определени случай те могат да служат и като доказателство.
* За улеснение на администраторите, чрез app shortcuts да може директно да се води кореспонденция с някои от последно свързалите се клиенти.
* Значително удобство може да бъде, използването на нотификации, с което да се премахне нуждата от ръчно проверяване за нови съобщения, когато приложението не работи.

**Библиография:**

1. **Божикова, В., Алексиева, В.** Програмни спецификации. ТУ-Варна, 2015
2. **Русков, Т., Вълчанов, Х.** Операционни системи. ТУ-Варна, 2013
3. **Рускова, Н., Вълчанов, Х.** Разпределено програмиране. ТУ-Варна, 2012
4. Андройд компоненти. 2017. <https://developer.android.com/guide/components.html> (10.06.2017)
5. Брой на смарт телефони по света. 2017. <https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/> (15.06.2017)
6. Пазарен дял на мобилните операционни системи. 2015. <https://www.forbes.com/sites/greatspeculations/2015/07/14/reasons-why-nokia-may-be-planning-to-re-enter-the-smartphone-business/#50cbd4b344e9> (10.06.2017)
7. Java database connector. 2017. <https://www.ibm.com/developerworks/rational/library/769.html> (12.06.2017)
8. TCP Сокети 2017. <https://docs.oracle.com/javase/tutorial/networking/sockets/> (04.06.2017)

**Приложение:**

**Сорс код:**

**Сървър:**

Интерфейси:

MessageDAO.java

**public** **interface** MessageDAO {

**long** insert(MessageDTO dto);

**long** update(MessageDTO dto);

**void** delete(MessageDTO dto);

List<MessageDTO> query(MessageDTO dto);

List<MessageDTO> queryAllUnsendMessagesForReceiver(**long** receiverId);

}

ClientDAO.java

**public** **interface** ClientDAO {

**long** insert(ClientForm form);

**long** update(ClientForm form);

**void** delete(ClientForm form);

List<ClientForm> query(ClientForm form);

ClientForm findClient(**long** id);

ClientForm findClient(String email);

}

ChatServer.java

**public** **interface** ChatServer {

/\*

\* Start Server and listen for clients

\*/

**public** **void** startServer();

}

Имплементация:

ServerImplementation.java

public final class ServerImplementation implements ChatServer, ClientConnectionCallback {

private static final int DEFAULT\_PORT\_NUMBER = 3456;

private static final String SYSTEM\_MESSAGE\_COMMAND = "system";

private static final String ADMIN\_MODE = "adminmode";

private static final String MESSAGE\_INVALID\_PARAMETERS = "invalid parameters";

private static final String MESSAGE\_LOGIN\_ERROR = "login error";

private static final String COMMAND\_FIND = "findclient";

private static final String COMMAND\_FIND\_RESPONSE = "findclientresponse";

private static final String COMMAND\_LOGIN\_RESPONSE = "loginresponse";

private static final String WHISPER\_COMMAND = "whisper";

private int portNum;

private int maxConnections;

private List<ConnectionThread> unvalidatedConnections;

private ConcurrentMap<Long, ConnectionThread> validConnections;

public ServerImplementation(int portNum, int maxConnections) {

// Check if provided port num is between the min and max port range

if (portNum > 0 && portNum < 65545) {

this.portNum = portNum;

} else {

this.portNum = DEFAULT\_PORT\_NUMBER;

}

if (maxConnections > 0 && maxConnections < Integer.MAX\_VALUE) {

this.maxConnections = maxConnections;

} else {

printMessageOnConsole("Error. Max connections must be between 0 and ~ 2 000 000 000");

System.exit(1);

}

// unvalidatedConnectionsCount = new AtomicInteger(0);

unvalidatedConnections = Collections.synchronizedList(new ArrayList<>());

validConnections = new ConcurrentHashMap<>();

}

@Override

public void startServer() {

ServerSocket serverSocket = null;

boolean socketIsListening = true;

try {

serverSocket = new ServerSocket(portNum);

printMessageOnConsole("Server is listening on port: " + portNum);

while (socketIsListening) {

// Because sum may overflow we cast to long

if ((long) unvalidatedConnections.size() + validConnections.size() <= maxConnections) {

ConnectionThread connection = new ConnectionThread(serverSocket.accept(), this);

connection.start();

// When first connect client has not validated itself

unvalidatedConnections.add(connection);

printMessageOnConsole("Number of unvalidated clients is: " + unvalidatedConnections.size()

+ " and valid is: " + validConnections.size());

} else {

printMessageOnConsole("Max number of connections reached: " + maxConnections);

}

}

} catch (IOException e) {

System.out.println("Error could not listen on port : " + portNum);

e.printStackTrace();

} finally {

try {

socketIsListening = false;

serverSocket.close();

} catch (IOException e) {

System.out.println("Error closing serverSocket");

e.printStackTrace();

}

}

}

/\*\*

\* Generates access token for every client

\*/

private String generateAccessToken() {

int randomNum = ThreadLocalRandom.current().nextInt(0, Integer.MAX\_VALUE - 1);

return String.valueOf(randomNum);

}

private void printMessageOnConsole(String message) {

System.out.println(message);

}

/\*\* Callback methods are implemented below \*/

@Override

public synchronized void disconnectUnvalidatedClient(long threadId) {

boolean isRemoved = false;

for (int i = 0; i < unvalidatedConnections.size(); i++) {

if (threadId == unvalidatedConnections.get(i).getId()) {

unvalidatedConnections.remove(i);

isRemoved = true;

break;

}

}

// Write log

if (isRemoved) {

printMessageOnConsole("Disconnected client with threadId: " + threadId);

} else {

printMessageOnConsole("Error disconnecting unvalidated client. Client was not found in list. ");

}

printMessageOnConsole("Size of unvaldiated list is: " + unvalidatedConnections.size());

}

@Override

public synchronized void disconnectValidatedClient(long clientId) {

ConnectionThread removedClient = validConnections.remove(clientId);

if (removedClient != null) {

printMessageOnConsole("Disconnected client with clientId: " + clientId);

printMessageOnConsole("Valid clients count: " + validConnections.size());

} else {

printMessageOnConsole("Error disconnecting client. Client was not found in map. ");

printMessageOnConsole("Valid clients count: " + validConnections.size());

}

}

/\*\* Client has passed validation and can now be moved to trusted clients \*/

@Override

public synchronized void registerClient(ClientForm form, ConnectionThread connectionThread) {

// Validate required fields

if (((form.email == null || form.email.trim().length() == 0)

&& (form.password == null || form.password.trim().length() == 0))) {

connectionThread.sendErrorMessageToMyself(MESSAGE\_INVALID\_PARAMETERS);

return; }

ClientDAO clientDao = new ClientDAOImpl();

// Set access token on user to authenticate with it

form.setAccessToken(generateAccessToken());

// Save client in db

long clientId = clientDao.insert(form);

// Is save is successful

if (clientId > 0) {

// Find client in unvalidated list

int index = findIndexInUnvalidatedConnections(connectionThread);

if (index < 0) {

printMessageOnConsole("Error finding client in unvalidated list");

connectionThread.closeConnection();

return;

}

// Move to valid connections

ConnectionThread connection = unvalidatedConnections.remove(index);

connection.setUserId(clientId);

validConnections.put(clientId, connection);

// Client is valid raise his flag in order to know and notify him.

connectionThread.setValidated(true);

connectionThread.setRegisterResponseReady(true);

connectionThread.notify();

// Send response to sender

MessageForm messageForm = new MessageForm();

messageForm.setCommand(SYSTEM\_MESSAGE\_COMMAND);

connectionThread.sendMessageToMyself(messageForm);

printMessageOnConsole("Client with id " + clientId + " has registered");

printMessageOnConsole("Number of unvalidated clients is: " + unvalidatedConnections.size()

+ " and valid is: " + validConnections.size());

}

else {

connectionThread.sendErrorMessageToMyself(MESSAGE\_LOGIN\_ERROR);

}

}

/\*\* If client pass validation , will now be moved to trusted clients \*/

@Override

public synchronized void loginClient(ClientForm form, ConnectionThread connectionThread) {

// Validate required fields

if (!isLoginFormValid(form)) {

// Client is valid raise his flag in order to know and notify

// him.

connectionThread.setValidated(false);

connectionThread.setLoginResponseReady(true);

connectionThread.notify();

connectionThread.sendErrorMessageToMyself(MESSAGE\_INVALID\_PARAMETERS);

return;

}

ClientDAO clientDao = new ClientDAOImpl();

ClientForm clientFromDb = null;

// Find client in db by what is provided

if (form.getId() > 0) {

clientFromDb = clientDao.findClient(form.getId());

} else {

clientFromDb = clientDao.findClient(form.getEmail());

}

// If client exists

if (clientFromDb != null && clientFromDb.getId() > 0) {

// printMessageOnConsole("Emails are: " + clientFromDb.getEmail()+ "

// --- " + form.getEmail());

// Validate credentials

if ((clientFromDb.getEmail().equals(form.getEmail())

&& clientFromDb.getPassword().equals(form.getPassword()))

|| clientFromDb.getAccessToken().equals(form.getAccessToken())) {

// Generate new access token for client and save it in db

String accessToken = generateAccessToken();

clientFromDb.setAccessToken(accessToken);

long rowId = clientDao.update(clientFromDb);

if (rowId <= 0) {

printMessageOnConsole("Error saving client access token");

// Client is valid raise his flag in order to know and

// notify him.

connectionThread.setValidated(false);

connectionThread.setLoginResponseReady(true);

connectionThread.notify();

connectionThread.sendErrorMessageToMyself(MESSAGE\_LOGIN\_ERROR);

return;

}

// Find client in unvalidated list

int index = findIndexInUnvalidatedConnections(connectionThread);

if (index < 0) {

printMessageOnConsole("Error finding client in unvalidated list");

connectionThread.closeConnection();

return;

}

// Move client to valid connections

long clientId = clientFromDb.getId();

ConnectionThread connection = unvalidatedConnections.remove(index);

connection.setUserId(clientId);

validConnections.put(clientId, connection);

// Client is valid raise his flag in order to know and notify

// him.

connectionThread.setValidated(true);

connectionThread.setLoginResponseReady(true);

connectionThread.notify();

// Send response to sender

MessageForm messageForm = new MessageForm();

messageForm.setCommand(COMMAND\_LOGIN\_RESPONSE);

messageForm.setIsAdmin(clientFromDb.getIsAdmin());

messageForm.setAccessToken(accessToken);

messageForm.setId(clientId);

connectionThread.sendMessageToMyself(messageForm);

// If there are unsent messages to client,

// send them to him now

sendAllUnsendMessagesToReceiver(connectionThread);

printMessageOnConsole("Client with id " + clientId + " has loged in");

printMessageOnConsole("Number of unvalidated clients is: " + unvalidatedConnections.size()

+ " and valid is: " + validConnections.size());

} else {

// Client is valid raise his flag in order to know and notify

// him.

connectionThread.setValidated(false);

connectionThread.setLoginResponseReady(true);

connectionThread.notify();

// Send response to sender

MessageForm messageForm = new MessageForm();

messageForm.setCommand(COMMAND\_LOGIN\_RESPONSE);

messageForm.setMessage(MESSAGE\_INVALID\_PARAMETERS);

messageForm.setAccessToken(null);

messageForm.setId(0);

connectionThread.sendMessageToMyself(messageForm);

printMessageOnConsole("Client with thread id " + connectionThread.getId() + " has invalid credentials");

}

} else {

printMessageOnConsole("Client with thread id " + connectionThread.getId() + " does not exist in db");

// Client is valid raise his flag in order to know and notify

// him.

connectionThread.setValidated(false);

connectionThread.setLoginResponseReady(true);

connectionThread.notify();

// Send response to sender

MessageForm messageForm = new MessageForm();

messageForm.setCommand(COMMAND\_LOGIN\_RESPONSE);

messageForm.setMessage(MESSAGE\_INVALID\_PARAMETERS);

messageForm.setAccessToken(null);

messageForm.setId(0);

connectionThread.sendMessageToMyself(messageForm);

// connectionThread.sendErrorMessageToMyself(MESSAGE\_LOGIN\_ERROR);

}

}

@Override

public void findClientInDb(FindForm form, ConnectionThread connectionThread) {

// Find client in db by email or id

ClientDAO clientDao = new ClientDAOImpl();

ClientForm clientFromDb = null;

if (form.getEmail() != null && form.getEmail().trim().length() != 0) {

clientFromDb = clientDao.findClient(form.getEmail());

} else {

clientFromDb = clientDao.findClient(form.getClientId());

}

// Response if client is found

if (clientFromDb != null) {

FindForm returnForm = new FindForm();

returnForm.setCommand(COMMAND\_FIND\_RESPONSE);

returnForm.setClientId(clientFromDb.getId());

returnForm.setEmail(clientFromDb.getEmail());

connectionThread.sendMessageToMyself(returnForm);

} else {

FindForm returnForm = new FindForm();

returnForm.setCommand(COMMAND\_FIND\_RESPONSE);

returnForm.setClientId(-1); // set id as invalid

returnForm.setEmail(null);

connectionThread.sendMessageToMyself(returnForm);

}

}

@Override

public boolean doesUserExists(long clientId) {

ClientDAO clientDao = new ClientDAOImpl();

ClientForm clientFromDb = clientDao.findClient(clientId);

if (clientFromDb != null) {

return true;

}

return false;

}

public boolean doesUserExists(String email) {

ClientDAO clientDao = new ClientDAOImpl();

ClientForm clientFromDb = clientDao.findClient(email);

if (clientFromDb != null) {

return true;

}

return false;

}

@Override

public void sendMessageToReceiver(MessageForm form) {

// If receiver is online send now the message else store in db and

// wait to come online.

ConnectionThread connectionThread = validConnections.get(form.getReceiverId());

if (connectionThread != null) {

sendMessageToOnlineReceiver(form);

} else {

// Store the message to send when he comes online

storeUnsentMessageToReceiver(form);

}

}

/\*\* Helper methods \*/

/\*\*

\* Store in db messages

\*/

private void storeUnsentMessageToReceiver(MessageForm form) {

MessageDAO messagesDao = new MessageDAOImpl();

// Convert data to dto

MessageDTO dto = new MessageDTO();

dto.setSenderId(form.getSenderId());

dto.setReceiverId(form.receiverId);

dto.setMessage(form.getMessage());

dto.setIsReceivedByClient(0); // 0 = false

messagesDao.insert(dto);

}

private void sendAllUnsendMessagesToReceiver(final ConnectionThread client) {

MessageDAO messageDao = new MessageDAOImpl();

List<MessageDTO> list = messageDao.queryAllUnsendMessagesForReceiver(client.getUserId());

for (MessageDTO dto : list) {

// Convert data to MessageForm

MessageForm messageForm = new MessageForm();

messageForm.setMessageId(dto.getId());

messageForm.setCommand(WHISPER\_COMMAND);

messageForm.setSenderId(dto.getSenderId());

messageForm.setReceiverId(dto.getReceiverId());

messageForm.setMessage(dto.getMessage());

client.sendMessageToMyself(messageForm);

// Update db that the msg was send to receiver

dto.setIsReceivedByClient(1); // 1 = true

messageDao.update(dto);

}

}

private boolean isLoginFormValid(ClientForm form) {

boolean isValid = false;

if (!((form.email == null || form.email.trim().length() == 0)

&& (form.password == null || form.password.trim().length() == 0))) {

isValid = true;

} else if (!(form.accessToken == null || form.accessToken.trim().length() == 0)) {

isValid = true;

}

return isValid;

}

public void sendMessageToOnlineReceiver(MessageForm form) {

ConnectionThread connection = validConnections.get(form.getReceiverId());

connection.sendMessageToMyself(form);

}

/\*\*

\* Finds and return the id of element. It there is not such element it

\* returns -1.

\*/

private synchronized int findIndexInUnvalidatedConnections(ConnectionThread connectionThread) {

for (int i = 0; i < unvalidatedConnections.size(); i++) {

if (connectionThread.getId() == unvalidatedConnections.get(i).getId()) {

return i;

}

}

return -1;

}

}

ConnectionThread.java

public final class ConnectionThread extends Thread {

private static final String COMMAND = "command";

private static final String COMMAND\_PREFIX = ":";

private static final String CLIENT\_IDENTIFICATION\_COMMAND = ":meet";

private static final String GET\_ALL\_CLIENTS\_COMMAND = ":who";

private static final String COMMAND\_QUIT = "quit";

private static final String SYSTEM\_MESSAGE\_COMMAND = "system";

private static final String COMMAND\_FIND = "findclient";

private static final String COMMAND\_FIND\_RESPONSE = "findclientresponse";

private static final String COMMAND\_LOGIN = "login";

private static final String WHISPER\_COMMAND = "whisper";

private static final String REGISTER\_COMMAND = "register";

private static final String INFO\_COMMAND = "ack";

private static final String PING\_COMMAND = "ping";

private static final String SUCCESS = "success";

private static final String DEFAULT\_THREAD\_NAME = "Guest";

private static final int INCORRECT\_ATTEMPTS\_АLLOWED = 10;

private ClientConnectionCallback listener;

private Socket clientSocket;

private PrintWriter out;

private BufferedReader in;

private LinkedBlockingQueue<String> messagesToSend = new LinkedBlockingQueue<>();

private Long userId = 0L;

private boolean isConnected;

private AtomicBoolean hasClientValidated;

private boolean isLoginResponseReady;

private boolean isRegisterResponseReady;

/\*\*

\* Set userId after client has logged in or registered and is now thrusted

\* connection

\*/

public synchronized void setUserId(Long userId) {

this.userId = userId;

}

public synchronized long getUserId(){

return this.userId;

}

public void setValidated(boolean isValidated) {

this.hasClientValidated.set(isValidated);

}

/\*\* Returns weather the client has connected successfully \*/

public boolean isConnected() {

return isConnected;

}

/\*\*

\* Set this to true then the main method wants to notify thread about login

\* result

\*/

public synchronized void setLoginResponseReady(boolean isLoginResponseReady) {

this.isLoginResponseReady = isLoginResponseReady;

}

public synchronized void setRegisterResponseReady(boolean isRegisterResponseReady) {

this.isRegisterResponseReady = isRegisterResponseReady;

}

public ConnectionThread(Socket clientSocket, ClientConnectionCallback listener) {

// Change the build in name of thread to default name.

this.setName(DEFAULT\_THREAD\_NAME);

this.clientSocket = clientSocket;

this.listener = listener;

this.isConnected = false;

this.hasClientValidated = new AtomicBoolean(false);

isLoginResponseReady = false;

isRegisterResponseReady = false;

}

public void run() {

printMessageOnConsole(

"A new client with thread id: " + Thread.currentThread().getId() + " has connected to the Server");

try {

// Open send and receive streams to client.

out = new PrintWriter(clientSocket.getOutputStream(), true);

in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

isConnected = true;

runOutInWorkerThread();

// Wait client to login or register before he can do anything else

waitForClientToLoginOrRegister();

String msgFromClient;

try {

// gson = new Gson();

while (isConnected) {

if ((msgFromClient = in.readLine()) != null) {

printMessageOnConsole("Read msg: " + msgFromClient);

// MessageForm messageForm =

// gson.fromJson(msgFromClient, MessageForm.class);

// Check what type of command the user send

handleCommand(msgFromClient);

}

}

} catch (IOException e) {

// Suppress exception if trying to read from closed stream

}

} catch (IOException | InterruptedException e) {

printMessageOnConsole("Error opening send and receive streams to client or Inerrupeted exception!");

e.printStackTrace();

} finally {

printMessageOnConsole(this.getId() + " thread is in finally");

try {

in.close();

} catch (Exception e) {

printMessageOnConsole("Error closing input stream to client.");

e.printStackTrace();

}

try {

clientSocket.close();

} catch (IOException e) {

e.printStackTrace();

}

// If the client has been validated this thread is given given

// userId.

// If not use the build in thread id to remove the connection

if (userId <= 0) {

listener.disconnectUnvalidatedClient(this.getId());

} else {

listener.disconnectValidatedClient(userId);

}

}

}

private void runOutInWorkerThread() {

Thread outThread = new Thread(new Runnable() {

@Override

public void run() {

// Send message to server

while (isConnected) {

try {

String msg = messagesToSend.take();

printMessageOnConsole("Sending to client msg: " + msg);

out.println(msg);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

if (out != null) {

out.close();

}

}

});

outThread.start();

}

/\*\*

\* This method is executing in a worker thread

\*/

private void handleCommand(final String messageFromClient) {

// String commandValue =parseCommandValue (messageFromClient);

// Check what type the command is

if (messageFromClient.contains(WHISPER\_COMMAND)) {

handleWhisperCommand(messageFromClient);

} else if (messageFromClient.contains(COMMAND\_FIND)) {

handleFindCommand(messageFromClient);

} else if (messageFromClient.contains(SYSTEM\_MESSAGE\_COMMAND)) {

printMessageOnConsole("System message from client : " + messageFromClient);

} else if (messageFromClient.contains(COMMAND\_QUIT)) {

handleQuitCommand();

} else {

sendErrorMessageToMyself("Invalid command");

}

}

private void handleFindCommand(String messageFromClient) {

Gson json = new Gson();

FindForm findForm = json.fromJson(messageFromClient, FindForm.class);

// Validate required fields

if ((findForm.getEmail() == null || findForm.getEmail().trim().length() == 0) && findForm.getClientId() <= 0) {

sendErrorMessageToMyself("Invalid parameters. You shoud provide client id or client email.");

return;

}

listener.findClientInDb(findForm, this);

}

private void handleWhisperCommand(String messageFromClient) {

Gson json = new Gson();

MessageForm receivedForm = json.fromJson(messageFromClient, MessageForm.class);

// Send message only if it is valid. If not inform the sender.

if (!isInputValid(WHISPER\_COMMAND, receivedForm)) {

sendErrorMessageToMyself("Invalid parameters");

return;

}

if (listener.doesUserExists(receivedForm.getReceiverId())) {

// The server sets the client id of the sender

// explicitly, not the client.

receivedForm.setSenderId(this.userId);

// Send info msg to server to know that his msg was received.

MessageForm infoForm = new MessageForm();

infoForm.setCommand(INFO\_COMMAND);

infoForm.setMessageId(receivedForm.getMessageId());

infoForm.setMessage(SUCCESS);

sendMessageToMyself(infoForm);

listener.sendMessageToReceiver(receivedForm);

} else {

sendErrorMessageToMyself("Receiver does not exist");

}

}

private void waitForClientToLoginOrRegister() throws IOException, InterruptedException {

int incorrectAttempts = 0;

int disconnectAfter = 10 \* 1000; // 10 seconds

long timeEnd = Calendar.getInstance().getTimeInMillis() + disconnectAfter;

String lineRead;

try {

// while (timeEnd > Calendar.getInstance().getTimeInMillis()) {

Gson json = new Gson();

while (!hasClientValidated.get() && incorrectAttempts < INCORRECT\_ATTEMPTS\_АLLOWED) {

if ((lineRead = in.readLine()) != null) {

printMessageOnConsole("Client thread id: " + this.getId() + " login msg: " + lineRead);

// Trim the string in order to remove tailing characters

// that break the parsing

ClientForm clientForm = json.fromJson(lineRead.trim(), ClientForm.class);

// Check if there is such user.

if (clientForm.getCommand().equals(COMMAND\_LOGIN)) {

synchronized (this) {

listener.loginClient(clientForm, this);

// wait for response from the main thread

while (!isLoginResponseReady) {

this.wait();

}

isLoginResponseReady = false; // reset the flag

}

} else if (clientForm.getCommand().equals(REGISTER\_COMMAND)) {

synchronized (this) {

listener.registerClient(clientForm, this);

// wait for response from the main thread

while (!isRegisterResponseReady) {

this.wait();

}

// reset the flag

isRegisterResponseReady = false;

}

} else {

// Handle invalid command

incorrectAttempts++;

MessageForm errorForm = new MessageForm();

errorForm.setCommand(SYSTEM\_MESSAGE\_COMMAND);

errorForm.setMessage("Invalid command. User must first login or register");

sendMessageToMyself(errorForm);

}

} else {

printMessageOnConsole("Input is null");

if (out == null){

printMessageOnConsole("Output is null");

}

// closeConnection();

// listener.disconnectUnvalidatedClient(this.getId());

break;

}

} // end while

// }

} catch (JsonSyntaxException jsonException) {

// Handle invalid command

incorrectAttempts++;

sendErrorMessageToMyself("Error. Expecting json, received string.");

}

if (!hasClientValidated.get()) {

closeConnection();

listener.disconnectUnvalidatedClient(this.getId());

}

}

/\*\*

\* Closes socket and tells the server to remove him from collection

\*

\*/

private void handleQuitCommand() {

printMessageOnConsole("Thread id " + this.getId() + " handle quit command");

closeConnection();

}

private boolean isInputValid(String messageType, MessageForm form) {

if (messageType.equals(WHISPER\_COMMAND)) {

// Validate we have receiver to send to

if (form.receiverId <= 0) {

sendErrorMessageToMyself("Reveicer ID id null or 0");

return false;

}

// Validate that the form has msg to send

if (form.message == null || form.message.trim().length() == 0){

sendErrorMessageToMyself("Message is null or empty");

return false;

}

// Validate that the message has msg id

if (form.messageId <= 0) {

sendErrorMessageToMyself("Message ID is null or empty");

return false;

}

return true;

}

return false;

}

public void printMessageOnConsole(String message) {

System.out.println(message);

}

/\*\*

\* @param errorText

\* - Add only the text you want to send and this method handles

\* the rest.

\*/

public void sendErrorMessageToMyself(String errorText) {

Gson json = new Gson();

MessageForm errorForm = new MessageForm();

errorForm.setCommand(SYSTEM\_MESSAGE\_COMMAND);

errorForm.setMessage(errorText);

String response = json.toJson(errorForm);

messagesToSend.add(response);

}

/\*\*

\*

\* @param form

\* - the form to be populated with values is, responsibility of

\* the person who called this method.

\*/

public void sendMessageToMyself(MessageForm form) {

Gson json = new Gson();

String response = json.toJson(form);

//printMessageOnConsole("sendMessageToMyself: " + response);

messagesToSend.add(response);

}

/\*\*

\* @param form

\* - the form to be populated with values is, responsibility of

\* the person who called this method.

\*/

public void sendMessageToMyself(ClientForm form) {

Gson json = new Gson();

String response = json.toJson(form);

messagesToSend.add(response);

}

/\*\*

\*

\* @param form

\* - the form to be populated with values is, responsibility of

\* the person who called this method.

\*/

public void sendMessageToMyself(FindForm form) {

Gson json = new Gson();

String response = json.toJson(form);

messagesToSend.add(response);

}

public void closeConnection() {

isConnected = false;

}

/\*

\* Callback interface

\*/

public interface ClientConnectionCallback {

void registerClient(ClientForm client, ConnectionThread connectionThread);

void loginClient(ClientForm client, ConnectionThread connectionThread);

void findClientInDb(FindForm form, ConnectionThread connectionThread);

boolean doesUserExists(long clientId);

void disconnectValidatedClient(long userId);

void disconnectUnvalidatedClient(long threadId);

/\*\*

\* Pass message to server who is responsible for sending it to the

\* receiving client

\*/

void sendMessageToReceiver(MessageForm form);

}

}

**Клиентски сорс код:**

MainAcitivity.java  
**public class** MainActivity **extends** AppCompatActivity **implements** MainFragmentListener,  
 DialogFragmentListener, MainActivityHelperCallback, RecyclerViewListener,  
 LoginHelperCallback, ErrorOccurredCallback {  
  
 **public static final** String ***INTENT\_MESSAGE\_RESTART\_SERVICE*** = **"intent\_message\_restart"**;  
  
 **private static final** String ***TAG*** = MainActivity.**class**.getSimpleName();  
 **private static final long *SUPPORT\_ID*** = 1;  
 **private static int** *COLUMN\_COUNT* = 2;  
 **private** SocketService **mService**;  
 **private boolean mBound** = **false**;  
 **private** String **receiverEmail** = **null**;  
  
 **private** TextView **tvNoService**;  
  
  
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.***activity\_main***);  
  
 NetworkChangeReceiver receiver = **new** NetworkChangeReceiver();  
 receiver.setListener(MainActivity.**this**);  
  
 **tvNoService** = (TextView) MainActivity.**this**.findViewById(R.id.***tvNoService***);  
  
 Toolbar myToolbar = (Toolbar) findViewById(R.id.***my\_toolbar***);  
 setSupportActionBar(myToolbar);  
  
  
 *// Bind to SocketService* Intent intent = **new** Intent(**this**, SocketService.**class**);  
 bindService(intent, **mConnection**, Context.***BIND\_AUTO\_CREATE***);  
  
 FragmentTransaction ft = getFragmentManager().beginTransaction();  
 SessionManager sessionManager = SessionManager.*getInstance*(getApplicationContext());  
  
 *// Depending weather the client is admin load the appropriate fragment* **if** (sessionManager.isAdminModeEnabled()) {  
 ClientsFragment clientsFragment = ClientsFragment.*newInstance*(*COLUMN\_COUNT*);  
 ft.addToBackStack(ClientsFragment.**class**.getSimpleName())  
 .add(R.id.***flMainActivityFrame***, clientsFragment, ClientsFragment.**class**.getSimpleName()).commit();  
 } **else** {  
  
 ChatFragment chatFragment = ChatFragment.*newInstance*(***SUPPORT\_ID***);  
 ft.addToBackStack(ChatFragment.**class**.getSimpleName())  
 .add(R.id.***flMainActivityFrame***, chatFragment, ChatFragment.**class**.getSimpleName()).commit(); } }  
 @Override  
 **public boolean** onCreateOptionsMenu(Menu menu) {  
 *// Inflate the menu; this adds items to the action bar if it is present.* getMenuInflater().inflate(R.menu.***menu\_main***, menu);  
 **return true**;  
 }  
  
 @Override  
 **public boolean** onOptionsItemSelected(MenuItem item) {  
 *// Handle action bar item clicks here. The action bar will  
 // automatically handle clicks on the Home/Up button, so long  
 // as you specify a parent activity\_main in AndroidManifest.xml.* **int** id = item.getItemId();  
  
 **if** (id == R.id.***action\_new\_chat***) {  
 *// Open dialog for user to enter receivers email* askUserForReceiverEmail();  
  
 *// Show spinner until server calback returns the receiverId* findViewById(R.id.***loadingPanel***).setVisibility(View.***VISIBLE***);  
 }  
 **if** (id == R.id.***action\_signout***) {  
 *// Remove stored user data* SessionManager.*getInstance*(getApplicationContext()).removeAllData();  
  
  
 Intent intent = **new** Intent(**this**, LoginActivity.**class**);  
 intent.putExtra(***INTENT\_MESSAGE\_RESTART\_SERVICE***, **true**);  
 startActivity(intent);  
  
 finish();  
 }

**return super**.onOptionsItemSelected(item);  
 }  
  
  
 **private void** askUserForReceiverEmail() {  
 MyDialogFragment newFragment = **new** MyDialogFragment();  
 newFragment.show(getFragmentManager(), **"dialog"**);  
 }  
  
 @Override  
 **public void** onBackPressed() {  
 **int** index = getFragmentManager().getBackStackEntryCount() - 1;  
 FragmentManager.BackStackEntry backEntry = getFragmentManager().getBackStackEntryAt(index);  
 String tag = backEntry.getName();  
  
 Log.*d*(***TAG***, **"onBackPressed fragment count: "** + index + **" tag: "** + tag);  
 **if** (index == 0) {  
 *// No fragment to remove so close the app* finish();  
 } **else** {  
 getFragmentManager().popBackStack();  
 } }  
  
 @Override  
 **protected void** onDestroy() {  
 **super**.onDestroy();  
 Log.*d*(***TAG***, **"onDestroy: will tell the service to stop"**); *// Unbind from the service* **if** (**mBound**) {  
 unbindService(**mConnection**);  
 **mBound** = **false**;  
 }  
 }  
  
 @Override  
 **public void** hasConnection(**boolean** hasConnection) {  
  
 Log.*d*(***TAG***, **"hasConnection: "** + hasConnection);  
  
 **if** (**tvNoService** != **null**) {  
 **this**.showNoServiceUI(!hasConnection);  
 }  
 }  
  
 **public void** showNoServiceUI(**final boolean** show) {  
 Log.*d*(***TAG***, **"showNoServiceUI: "** + show);  
  
 MainActivity.**this**.runOnUiThread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **if** (show) {  
 **tvNoService**.setText(R.string.***no\_service***);  
 **tvNoService**.setVisibility(View.***VISIBLE***);  
 } **else** {  
 **tvNoService**.setVisibility(View.***GONE***);  
 } } }); }  
  
  
 **private void** markMessagesAsRead(**long** senderId) {  
  
 ContentValues contentValues = **new** ContentValues();  
 contentValues.put(HistoryEntry.***COLUMN\_IS\_MESSAGE\_READ\_BY\_ME***, 1); *// 1 = true* String selection = HistoryEntry.***COLUMN\_SENDER\_ID*** + **" = ? "**;  
 String selectionArgs[] = {String.*valueOf*(senderId)};  
  
  
 **int** rowsUpdated = getApplicationContext().getContentResolver().update(  
 HistoryEntry.***CONTENT\_URI***,  
 contentValues,  
 selection,  
 selectionArgs);  
 Log.*d*(***TAG***, **"marked msg as read count: "**); }  
  
 @Deprecated  
 **private boolean** hasForIntenetConnection() {

ConnectivityManager cm =  
 (ConnectivityManager) ((Context) **this**).getSystemService(Context.***CONNECTIVITY\_SERVICE***);  
  
 NetworkInfo activeNetwork = cm.getActiveNetworkInfo();  
 **boolean** isConnected = activeNetwork != **null** &&  
 activeNetwork.isConnectedOrConnecting();  
  
 **return** isConnected;  
 }  
  
  
 */\*\*  
 \* Defines callbacks for service binding, passed to bindService()  
 \*/* **private** ServiceConnection **mConnection** = **new** ServiceConnection() {  
  
 @Override  
 **public void** onServiceConnected(ComponentName className,  
 IBinder service) {  
 *// We've bound to SocketService, cast the IBinder and get LocalService instance* SocketService.LocalBinder binder = (SocketService.LocalBinder) service;  
 **mService** = binder.getService();  
 **mBound** = **true**;  
  
 *// Set the listener to receive callback from service* **mService**.setMainActivityCallback((MainActivityHelperCallback) MainActivity.**this**);  
 **mService**.setErrorOccuredCallback((ErrorOccurredCallback) MainActivity.**this**);  
  
 }  
  
 @Override  
 **public void** onServiceDisconnected(ComponentName arg0) {  
 **mBound** = **false**;  
 }  
 };  
  
 */\* Callback methods are implemented below \*/* @Override  
 **public void** sendMessageButtonClicked(String text, **final long** receiverId) {  
 **long** myId = SessionManager.*getInstance*(getApplicationContext()).getClientId();  
  
 *// Prepare data to save in db* ContentValues contentValues = **new** ContentValues();  
 contentValues.put(HistoryEntry.***COLUMN\_SENDER\_ID***, myId); *// set my id for later to show in chat fragment the chat history* contentValues.put(HistoryEntry.***COLUMN\_RECEIVER\_ID***, receiverId);  
 contentValues.put(HistoryEntry.***COLUMN\_MESSAGE***, text);  
 contentValues.put(HistoryEntry.***COLUMN\_IS\_MESSAGE\_OUTGOING***, 1); *// We set to true because we are sending to server* contentValues.put(HistoryEntry.***COLUMN\_IS\_MESSAGE\_RECEIVED***, 0); *// At the moment we assume that is has not been send* contentValues.put(HistoryEntry.***COLUMN\_IS\_MESSAGE\_READ\_BY\_ME***, 1); *// This is always true because we are the ones who wrote it* contentValues.put(HistoryEntry.***COLUMN\_TIMESTAMP***, TimeUtility.*getCurrentTimeStamp*()); *// Time we send the message* Log.*d*(***TAG***, **"sendMessageButtonClicked Will save text and isOutgoung: "** + text + **" --- "** + contentValues.getAsInteger(HistoryEntry.***COLUMN\_IS\_MESSAGE\_OUTGOING***));  
  
 *// Save message in db* Uri newRecordUri = getContentResolver().insert(HistoryEntry.***CONTENT\_URI***, contentValues);  
 **long** newRecordId = ContentUris.*parseId*(newRecordUri);  
 Log.*d*(***TAG***, **"sendMessageButtonClicked: inserted : "** + newRecordId);  
 **mService**.sendWhisperMessageToServer(text, receiverId, newRecordId);  
 }  
  
 @Override  
 **public void** markMessagesWithClientAsRead(**long** receiverId) {  
 MainActivity.**this**.markMessagesAsRead(receiverId);  
 }  
  
 @Override  
 **public void** sendMessageAgain(**final** String message, **final long** receiverId, **final int** messageId) {  
  
 *// Use the Builder class for convenient dialog construction* AlertDialog.Builder builder = **new** AlertDialog.Builder(MainActivity.**this**);  
 builder.setMessage(R.string.***send\_again***)  
 .setPositiveButton(R.string.***send***, **new** DialogInterface.OnClickListener() {  
 **public void** onClick(DialogInterface dialog, **int** id) {  
  
 **mService**.sendWhisperMessageToServer(message, receiverId, messageId);  
 }  
 })  
 .setNegativeButton(R.string.***cancel***, **new** DialogInterface.OnClickListener() {  
 **public void** onClick(DialogInterface dialog, **int** id) {  
 *// Remove loading spinner* MainActivity.**this**.findViewById(R.id.***loadingPanel***).setVisibility(View.***GONE***);  
 dialog.dismiss();  
 }  
 })  
 .setOnDismissListener(**new** DialogInterface.OnDismissListener() {  
 @Override  
 **public void** onDismiss(DialogInterface dialog) {  
 *// Remove loading spinner* MainActivity.**this**.findViewById(R.id.***loadingPanel***).setVisibility(View.***GONE***);  
 dialog.dismiss();  
 }  
 });  
 Dialog dialog = builder.create();  
 dialog.show();  
 }  
  
 */\*  
 Callback method from DialogFragmentListener  
 \*/* @Override  
 **public void** onDialogPositiveButtonClicked(String email) {  
  
 *// Ask server to give you the receivers id* **mService**.findClientInfo(0, email, MainActivity.**this**);  
  
 *// When user adds new email contact return him to ClientsFragment* Fragment f = MainActivity.**this**.getFragmentManager().findFragmentById(R.id.***flMainActivityFrame***);  
 **if** (f **instanceof** ChatFragment) {  
 onBackPressed();  
 }  
 }  
  
 @Override  
 **public void** OnDialogNegativeButtonClicked() {  
 Log.*d*(***TAG***, **"OnDialogNegativeButtonClicked: "**);  
 *// Remove loading spinner* MainActivity.**this**.findViewById(R.id.***loadingPanel***).setVisibility(View.***GONE***);  
 }  
  
 @Override  
 **public void** findClientResponse(**final long** clientId) {  
  
 *// Because this was called from another thread, run on UI thread explicitly.* runOnUiThread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 *// Validate that the client exist* **if** (clientId <= 0) {  
 Toast.*makeText*(MainActivity.**this**, **"There is no user with such email"**, Toast.***LENGTH\_LONG***).show();  
 *// Remove loading spinner* findViewById(R.id.***loadingPanel***).setVisibility(View.***GONE***);  
 **return**;  
 }  
  
 *// Remove loading spinner* findViewById(R.id.***loadingPanel***).setVisibility(View.***GONE***);  
  
 updateClientsFragmentUI();} }); }  
 */\*\*  
 \* Update the clients and the number of their messages that are unread by the user of this app.  
 \*/* @Override  
 **public void** updateClientsFragmentUI() {  
 *//Find ClientsFragment and tell it to requery the db.* **final** Fragment fragment = getFragmentManager().findFragmentByTag(ClientsFragment.**class**.getSimpleName());  
 **if** (fragment **instanceof** ClientsFragment) {  
 *// Because the method invocation originated from the service, explicitly tell to run on UI thread* MainActivity.**this**.runOnUiThread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 ((ClientsFragment) fragment).getClientsFromDbViaHandler();  
 }  
 });  
 }  
 }  
  
 @Override  
 **public void** onClientClickedInList(ClientListItemContent client) {

ChatFragment chatFragment = ChatFragment.*newInstance*(client.getClientId());  
 FragmentTransaction ft = getFragmentManager().beginTransaction();  
 ft.addToBackStack(ChatFragment.**class**.getSimpleName())  
 .replace(R.id.***flMainActivityFrame***, chatFragment, ChatFragment.**class**.getSimpleName()).commit();  
  
 *// Mark messages ar read when the chat fragment is opened.* markMessagesAsRead(client.getClientId());  
 }  
  
  
 */\*\*  
 \* Show dialog and delete if positive  
 \*/* @Override  
 **public void** onDeleteClientClicked(**final** ClientListItemContent client) {  
  
  
 *// Use the Builder class for convenient dialog construction* AlertDialog.Builder builder = **new** AlertDialog.Builder(MainActivity.**this**);  
 builder.setMessage(R.string.***delete\_client***)  
 .setPositiveButton(R.string.***delete***, **new** DialogInterface.OnClickListener() {  
 **public void** onClick(DialogInterface dialog, **int** id) {  
  
 ClientsDataProvider clientsDataProvider = **new** ClientsDataProvider(MainActivity.**this**);  
  
 String selection = ***COLUMN\_CLIENT\_ID*** + **" = ?"**;  
 String selectionArgs[] = {String.*valueOf*(client.getClientId())};  
 clientsDataProvider.delete(selection, selectionArgs);  
  
 updateClientsFragmentUI();  
 }  
 })  
 .setOnDismissListener(**new** DialogInterface.OnDismissListener() {  
 @Override  
 **public void** onDismiss(DialogInterface dialog) {  
 *// Remove loading spinner* MainActivity.**this**.findViewById(R.id.***loadingPanel***).setVisibility(View.***GONE***);  
 dialog.dismiss();  
 }  
 })  
 .setNegativeButton(R.string.***cancel***, **new** DialogInterface.OnClickListener() {  
 **public void** onClick(DialogInterface dialog, **int** id) {  
 dialog.dismiss();  
 }  
 });  
 Dialog dialog = builder.create();  
 dialog.show();  
 }}

ChatFragment.java

*/\*\*  
 \* A placeholder fragment containing a simple view.  
 \*/***public class** ChatFragment **extends** BaseFragment **implements** LoaderManager.LoaderCallbacks<Cursor> {  
 **private static final** String ***TAG*** = ChatFragment.**class**.getSimpleName();  
 **private static final** String ***RECEIVER\_ID\_BUNDLE\_KEY*** = **"receiverid"**;  
 **private final long MY\_ID** = SessionManager.*getInstance*((Context) getActivity()).getClientId();  
  
 **private** HistoryCursorAdapter **cursorAdapter**;  
  
 *// Views* **private** ListView **listView**;  
 **private** EditText **etUserInput**;  
 **private** Button **btnSendMessage**;  
  
 **private long receiverId** = 0;  
  
  
 **private static final** String[] ***HISTORY\_COLUMNS*** = {  
 HistoryEntry.***TABLE\_NAME*** + **"."** + HistoryEntry.***\_ID***,  
 HistoryEntry.***COLUMN\_RECEIVER\_ID***,  
 HistoryEntry.***COLUMN\_MESSAGE*** };  
  
 **public static** ChatFragment newInstance(**long** receiverId) {  
 ChatFragment fragment = **new** ChatFragment();  
 Bundle args = **new** Bundle();  
 args.putLong(***RECEIVER\_ID\_BUNDLE\_KEY***, receiverId);  
 fragment.setArguments(args);  
 **return** fragment;  
 }  
  
 @Override  
 **public void** onCreate(@Nullable Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
  
 *// Get the id from bundle* **receiverId** = getArguments().getLong(***RECEIVER\_ID\_BUNDLE\_KEY***, 0);

*// If finding receiver id was not successful exit fragment* **if** (**receiverId** <= 0) {  
 Toast.*makeText*((Context) getActivity(), **"Error getting receiver info"**, Toast.***LENGTH\_LONG***).show();  
 getActivity().onBackPressed();  
 }  
  
 *// Prepare the loader. Either re-connect with an existing one, or start a new one.* getLoaderManager().initLoader(0, **null**, **this**);  
  
 }  
  
 **private void** getReceiverIdFromDb(**final** String email) {  
 *// Query the db on worker thread* **new** Handler().post(**new** Runnable() {  
 @Override  
 **public void** run() {  
 ClientsDataProvider clientsDataProvider = **new** ClientsDataProvider((Context) getActivity());  
 **receiverId** = clientsDataProvider.findClientId(email);  
 }  
 });  
 }  
  
 @Override  
 **public** View onCreateView(LayoutInflater inflater, ViewGroup container,  
 Bundle savedInstanceState) {  
 View rootView = inflater.inflate(R.layout.***fragment\_chat***, container, **false**);  
**cursorAdapter** = **new** HistoryCursorAdapter(getActivity(), **null**, **true**);  
  
 *// Get a reference to the ListView, and attach this adapter to it.* **listView** = (ListView) rootView.findViewById(R.id.***lvChat***);  
 **listView**.setDivider(**null**);  
 **listView**.setAdapter(**cursorAdapter**);  
  
 **listView**.setOnItemLongClickListener(**new** AdapterView.OnItemLongClickListener() {  
 @Override  
 **public boolean** onItemLongClick(AdapterView<?> parent, View view, **int** position, **long** id) {  
  
 Cursor cursor = (Cursor) **listView**.getItemAtPosition(position);  
 *// Get data* **int** isReceived= cursor.getInt(cursor.getColumnIndex(HistoryEntry.***COLUMN\_IS\_MESSAGE\_RECEIVED***));  
 **int** isOutgoing = cursor.getInt(cursor.getColumnIndex(HistoryEntry.***COLUMN\_IS\_MESSAGE\_OUTGOING***));  
 String message = cursor.getString(cursor.getColumnIndex(HistoryEntry.***COLUMN\_MESSAGE***));  
 **long** receiverId = cursor.getLong(cursor.getColumnIndex(HistoryEntry.***COLUMN\_RECEIVER\_ID***));  
 **int** messageId = cursor.getInt(cursor.getColumnIndex(HistoryEntry.***\_ID***));  
  
 **if**(isReceived == 0 && isOutgoing == 1) {  
 **mListener**.sendMessageAgain(message, receiverId, messageId);  
 }  
 **return false**;  
 }  
 });  
 **return** rootView;  
 }

@Override  
 **public void** onViewCreated(View view, @Nullable Bundle savedInstanceState) {  
 **super**.onViewCreated(view, savedInstanceState);  
  
 **etUserInput** = (EditText) getActivity().findViewById(R.id.***etUserInput***);  
 **btnSendMessage** = (Button) getActivity().findViewById(R.id.***btnSendMessage***);  
  
 **btnSendMessage**.setOnClickListener(**new** View.OnClickListener() {  
 @Override  
 **public void** onClick(View v) {**if** (**etUserInput**.getText().toString().trim().length() == 0) {  
 **return**;  
 }  
  
 String text = **etUserInput**.getText().toString().trim();  
 Log.*d*(***TAG***, **"onClick send btn: "** + text);  
 **mListener**.sendMessageButtonClicked(text, **receiverId**);  
 **etUserInput**.setText(**null**);  
 }  
 });  
 }  
  
 @Override  
 **public void** onDestroy() {  
 **super**.onDestroy();  
 **mListener**.markMessagesWithClientAsRead(**receiverId**);  
 }  
  
 */\*\*  
 \* Implementation below for Loader callback methods  
 \*/* @Override  
 **public** Loader<Cursor> onCreateLoader(**int** id, Bundle args) {  
 *// This is called when a new Loader needs to be created.  
  
 // WHERE (senderId = 5 and receiverId = 1) OR ( senderId = 1 and receiverId = 5)  
 // One time the user of this app is the sender and on the receiver.* String selection = **"( "** + HistoryEntry.***COLUMN\_SENDER\_ID*** + **" = ? "** + **" AND "** +  
 HistoryEntry.***COLUMN\_RECEIVER\_ID*** + **" = ? "** +  
 **" ) "** + **" OR ( "** +  
 HistoryEntry.***COLUMN\_SENDER\_ID*** + **" = ? "** + **" AND "** +  
 HistoryEntry.***COLUMN\_RECEIVER\_ID*** + **" = ? "** +  
 **" )"**;  
 String selectionArgs[] = {String.*valueOf*(**MY\_ID**), String.*valueOf*(**receiverId**), String.*valueOf*(**receiverId**), String.*valueOf*(**MY\_ID**)};  
 String sortOrder = HistoryEntry.***\_ID*** + **" ASC"**;  
  
 **return new** CursorLoader(getActivity(),  
 HistoryEntry.***CONTENT\_URI***,  
 **null**, *// return all columns* selection,  
 selectionArgs,  
 sortOrder);  
 }  
  
 @Override  
 **public void** onLoadFinished(Loader<Cursor> loader, Cursor cursor) {  
 Log.*d*(***TAG***, **"onLoadFinished: rows count:"** + cursor.getCount() + **" poss: "** + cursor.getPosition());  
 **cursorAdapter**.swapCursor(cursor);  
 }

@Override  
 **public void** onLoaderReset(Loader<Cursor> loader) {  
 **cursorAdapter**.swapCursor(**null**);  
 }  
  
 **public interface** MainFragmentListener {  
  
 **void** sendMessageButtonClicked(String text, **final long** receiverId);  
  
 **void** markMessagesWithClientAsRead(**long** receiverId);  
  
 **void** sendMessageAgain(String message, **long** receiverId, **int** messageId) ;  
  
 }  
}

View елементи:

­fragment\_chat.xml

*<?***xml version="1.0" encoding="utf-8"***?>*<**LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:orientation="vertical"**>  
  
 <**ListView  
 android:id="@+id/lvChat"  
 android:layout\_width="match\_parent"  
 android:layout\_height="0dp"  
 android:layout\_weight="1"  
 android:transcriptMode="alwaysScroll"  
 android:stackFromBottom="true"**/>  
  
 <**LinearLayout  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:orientation="horizontal"**>  
  
 <**EditText  
 android:id="@+id/etUserInput"  
 android:layout\_width="0dp"  
 android:layout\_height="wrap\_content"  
 android:layout\_weight="1"  
 android:maxLength="300"  
 android:hint="Enter text"** />  
  
 <**Button  
 android:id="@+id/btnSendMessage"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:layout\_gravity="bottom"  
 android:text="Send"** />  
 </**LinearLayout**>  
</**LinearLayout**>

list\_item\_chat\_box.xml

*<?***xml version="1.0" encoding="utf-8"***?>*<**LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:orientation="vertical"  
 android:padding="@dimen/fragment\_margin"**>  
  
 <**TextView  
 android:id="@+id/tvTime"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:textSize="11sp"** />  
  
 <**LinearLayout  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:orientation="horizontal"**>  
  
 <**ImageView  
 android:id="@+id/receiverIcon"  
 android:layout\_width="30dp"  
 android:layout\_height="30dp"  
 android:layout\_marginTop="4dp"  
 android:src="@mipmap/ic\_2"** />  
  
 <**TextView  
 android:id="@+id/tvReceiversMessage"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:layout\_gravity="center\_vertical"  
 android:layout\_marginEnd="@dimen/big\_margin"  
 android:layout\_marginLeft="@dimen/fragment\_margin"  
 android:layout\_marginRight="@dimen/big\_margin"  
 android:layout\_marginStart="@dimen/fragment\_margin"  
 android:padding="8dp"  
 android:background="@drawable/rounded\_corners\_without\_left\_send\_color"  
 android:text="msg from receiver"** />  
  
  
 *<!-- Empty view for pushing the senders messages to the right side of screen-->* <**View  
 android:layout\_width="0dp"  
 android:layout\_height="1dp"  
 android:layout\_weight="1"** />  
  
 <**TextView  
 android:id="@+id/tvMyMessage"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:layout\_gravity="right|end"  
 android:layout\_marginLeft="@dimen/big\_margin"  
 android:layout\_marginStart="@dimen/big\_margin"  
 android:padding="8dp"  
 android:text="msg from sender"**>  
 </**TextView**>  
  
 </**LinearLayout**>  
</**LinearLayout**>