



Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Теоретическая информатика и компьютерные технологии

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ
НА ТЕМУ:

«Разработка мобильного приложения «Консилиум»

Студент ИУ9-82
(Группа)

(Подпись, дата) А. А. Белогуров
(И.О.Фамилия)

Руководитель ВКР

(Подпись, дата) А.Б. Домрачева
(И.О.Фамилия)

Консультант

(Подпись, дата) _____
(И.О.Фамилия)

Консультант

(Подпись, дата) _____
(И.О.Фамилия)

Нормоконтролер

(Подпись, дата) _____
(И.О.Фамилия)

2018 г.

Аннотация

Объем дипломной работы составляет 55 страниц, на которых размещены 17 рисунков, 8 таблиц и 16 листингов. При написании диплома использовался 21 источник.

Исследуемой областью являлось взаимодействие врачей и пациентов при помощи мобильных устройств на операционной системе Android.

В дипломную работу входит введение, четыре главы и заключение.

В введении раскрывается проблематика работы и ее актуальность, определяется цель и постановки задачи.

В заключении описываются результаты проделанной работы и дальнейшие перспективы разработки.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	3
1 Обзор предметной области и изучение существующих аналогов	5
1.1 Социальные медицинские сети	5
1.2 Требования к разрабатываемому приложению «Консилиум»	6
2 Проектирование и разработка Android-приложения и серверной части.....	9
2.1 Серверная часть.....	9
2.2 Проектирование базы данных	13
2.3 Клиентская часть.....	18
2.4 Разработка приложения на примере пользователя.....	22
2.5 Проектирование пользовательского интерфейса.....	33
3 Оптимизация работы приложения	37
3.1 Реактивное программирование.....	37
3.2 Сжатие изображений	39
3.3 Просмотр STL-моделей	41
4 Тестирование.....	45
4.1 Тестирование бизнес-процессов серверной части.....	45
4.2 Тестирование Android-приложения	50
ЗАКЛЮЧЕНИЕ.....	53
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	54

ВВЕДЕНИЕ

Современный человек активно пользуется портативными устройствами. В том числе это связано с тем, что в 90-х годах прошлого столетия стало возможным уместить большое количество различных схем и плат в небольшом по размеру общем корпусе, помещавшемся в кармане рубашки или штанов. В то время подобные устройства обладали маленьким объемом памяти и ограниченными возможностями.

С течением времени множество компаний начали вкладывать в это направление огромные суммы денег, что привело к появлению первых мобильных операционных систем в начале двадцать первого века. С каждым годом производители добавляли как можно больше разных возможностей, чтобы пользование мобильными устройствами становилось легче и проще, но, с другой стороны, их возможности могли заменить персональный компьютер.

Как правило, использование портативной техники не ограничивалось одним лишь браузером для просмотра веб-страниц. В огромном потоке информации стало актуальным использовать одно приложение для одной конкретной задачи, в котором будет собрана вся информация по данному направлению, от новостей и обучающих программ до всевозможных игр.

Кроме того, с интереса к портативным устройствам началось развитие социальных сетей. Из названия можно понять, что они служат для того, чтобы соединять большие группы людей, объединенных общей идеей – социумы. Это основные функции данных сервисов, но в зависимости от целей их создателей могут варьироваться между собой.

Как правило, социальные сети чаще имеют развлекательный характер, нежели профессиональный. Однако, если сделать возможным не только общение между коллегами, но и обмен информацией, характерной для конкретной области, то такие сервисы помогут повысить продуктивность работников, а также автоматизировать сложные задачи. Это особенно актуально в сфере медицины для привлечения к обсуждению

профессиональных проблем как можно большего числа специалистов, когда все процессы должны проходить быстро и давать точные результаты. Для этого в рамках данной дипломной работы будет создано приложение для операционной системы Android под названием «Консилиум».

В ходе работы должны быть выполнены следующие задачи:

- обзор предметной области и изучение существующих аналогов с целью определения требований к разрабатываемому ПО;
- разработка Android-приложения и серверной части и проектирование пользовательского интерфейса;
- оптимизация работы приложения;
- тестирование работоспособности.

1 Обзор предметной области и изучение существующих аналогов

1.1 Социальные медицинские сети

Перед началом разработки своей социальной медицинской сети необходимо провести поиск и сравнительный анализ других таких сервисов. Это поможет выявить их достоинства, которые можно перенять в свое приложение, и недостатки, чтобы их не допустить в дальнейшей разработке.

Поиск таких сервисов оказался довольно затруднительным, так как даты их создания приходятся на период 2010-2011 годов. Оказалось, что многие из них уже не функционируют, а пользователей насчитывают не более тысячи: Медтусовка [1], Alma Mater [2]. Это может говорить о том, что такие социальные сети создавались больше для учебных и тестовых целей, нежели для полноценной работы с пользователями. Большинство же работающих сервисов представляют собой медицинские порталы, в которых присутствуют записи к врачам, вызов их на дом и аналогичное взаимодействия пациента с медицинскими работниками.

Из по-настоящему функционирующих социальных медицинских сетей было найдено около 4 сервисов с большой базой врачей: Medring [3], Врачи РФ [4], Доктор на работе [5], Враче вместе [6]. Все они направлены на лучшее взаимодействие сотрудников медицинской сферы между собой, но у каждой присутствуют свои особенности. Также добавим в этот список дипломную работу 2017 года «Разработка медицинской социальной сети» Щедромирского С.В. для стационарных компьютеров. Проведём их сравнение в таблице 1, где доступные возможности будут помечены символом «+», а недоступные или неизвестные — «-».

Таблица 1 – Сравнительный анализ социальных медицинских сетей

Сервис	Диалоги	Кол-во пользователей	Моб. приложение	Доп. возможности
Medring	-	-	-	Расшифровка анализов, группы по интересам
Врачи РФ	-	500 000 +	iOS/Android	Публикации научных статей и исследований
Доктор на работе	+	550 000 +	iOS/Android	Публикации новостей мира медицины
Врачи вместе	-	100 000 +	-	Онлайн-лекции, новости медицины
ВКР Мед. Социальная сеть 2017	+	-	-	Просмотр DICOM и STL файлов

Как видно из таблицы выше, большая часть сервисов кроме профессиональной деятельности затрагивает также показ новостей из мира медицины, что позволяет узнавать о новых разработках и открытиях. Не все социальные сети имеют аналоги на мобильных платформах, что также является причиной их низкой популярности. Для регистрации в некоторых из них требуется подтверждение своей профессии. После исследования аналогов можно переходить к составлению и описанию возможностей разрабатываемой социальной медицинской сети.

1.2 Требования к разрабатываемому приложению «Консилиум»

Первым делом стоит продумать навигацию по Android-приложению. Самыми распространёнными вариантами является Navigation Drawer, который представляет собой список и выдвигается при свайпе слева направо, а также вкладки категорий, которые можно пролистывать пальцем. Остановимся на

первом из них, так как он является наиболее эффективным и простым в понимании.

При помощи Navigation Drawer можно будет перемещать по пяти главным категориям:

- профиль пользователя;
- все пользователи;
- избранные пользователи;
- диалоги;
- загруженные файлы.

Все категории, кроме первой, будут представлены в качестве списков. При нажатии на элемент списка будет открывать специализированное окошко с подробной информацией. Можно будет загрузить файлы наиболее популярных типов: JPG, TXT, PDF и другие. Не лишним будет добавить формат файлов STL, которые служат для хранения трёхмерных объектов для последующей их печати на 3D принтерах. С помощью него врачи могут легко визуализировать импланты любой сложности и размеров, что играет важную роль перед подготовкой к операции. Внутри приложения будет реализован просмотр файлов STL, PDF и JPG с дополнительными функциями приближения и отдаления. Кроме того, диалоги могут строиться между любыми двумя пользователями, одно сообщение может содержать текст или один файл, который можно выбрать с внутренней памяти телефона или из уже ранее загруженных файлов в приложении. Популярную категорию «друзья» предлагается заменить на «избранных пользователей», так как социальная медицинская сеть предусматривает больше профессиональную деятельность. Тогда у каждого пользователя будет свой список «избранных пользователей», чтобы он мог иметь к ним быстрый доступ. Список всех возможностей приложения не окончательный и может варьироваться в процессе разработки.

Теперь стоит определить ограничения, которые могут быть наложены на мобильное приложение. Во-первых, для авторизации необходимо будет ввести свое имя, логин с паролем, а также вид деятельности. Все пользователи будут

разделены на две группы: врачи и пациенты. Так как администрирование приложения на мобильном устройстве является не лучшей идеей, то права всех пользователей будут равны. Во-вторых, доступ к интернету должен быть надёжным в процессе всего взаимодействия с сервисом. Из-этого вытекает, что надо как можно больше ограничить размер пересылаемых данных между сервером и клиентом. Для этого размер всех загружаемых файлов будет ограничен отметкой в 10MB. Он позволит передавать как фотографии больших разрешений, так и сложные STL-модели. В-третьих, все фотографии на сервере будут сжиматься для уменьшения загружаемых данных с целью их компактного отображения в приложении.

Таким образом, сформулируем требования к разрабатываемому ПО:

- сокращённая авторизация;
- размер загружаемых файлов не должен превышать 10MB;
- необходимо надёжное соединение с интернетом;
- встроенный браузер должен иметь поддержку WEBGL для просмотра STL файлов.

После описания возможностей разрабатываемой социальной медицинской сети и наложения на неё некоторых ограничений можно переходить к построению архитектуры и проектированию отдельных компонентов всего проекта.

2 Проектирование и разработка Android-приложения и серверной части

2.1 Серверная часть

Разрабатываемое приложение представляет собой социальную сеть, которая должна быть доступна с любого портативного устройства на операционной системе Android. Следовательно, пользователь должен иметь возможность ввести свои данные в виде логина и пароля в приложении и получить доступ к своему аккаунту независимо от устройства, расположения и других условий. Обеспечить такой доступ к данным позволяет клиент-серверная архитектура приложения.

Как правило, сервер – это программное обеспечение, запущенное на отдельной вычислительной машине, которое обеспечивает взаимодействие между клиентом и базой данных. Клиент – это так же программное обеспечение, но к нему имеет доступ непосредственно сам пользователь и которое визуализирует данные, полученные с сервера. В качестве клиента может выступать любой сайт, приложение для мобильных или стационарных устройств и другие. Таким образом, клиентское приложение является скорее оберткой данных и не выполняет никаких сложных вычислений. Вся ресурсозатратная работа выполняется на самом сервере, что позволяет разделить нагрузку между приложениями. Но в этом можно заметить и главный недостаток использования такой архитектуры, а именно то, что повышение вычислительной мощности, быстродействия и эффективности работы сервера влечет за собой большие, а порой даже огромные материальные затраты. Это является следствием того, что все данные пересылаются через интернет, следовательно, необходимо обеспечить хорошее местоположение серверной части и надежное соединение с сетью, что позволит обеспечить минимальные задержки, а также серверное оборудование является более дорогим, по сравнению с обычными офисными или домашними решениями.

Для передачи данных между клиентом и сервером используется протоколы передачи данных, самым популярным из которых является HTTP и HTTPS. Вторым из названных протоколов является расширением первого, за исключением того, что он позволяет устанавливать защищенное соединение с использованием разных видов шифрования передаваемых сообщений. Для разрабатываемой социальной сети будет использоваться протокол HTTPS, так как пользователи будут отправлять свои личные данные, которые необходимы для доступа в приложение, передавать текстовые и файловые сообщения другим пользователям. Такого рода данные необходимо шифровать, чтобы злоумышленники не получили к ним доступ.

Сами сообщения, которыми обмениваются сервер и клиент, могут пересылаться в разных форматах передачи данных. Наиболее популярными форматами является JSON и XML. В данном случае, будет использоваться первый формат, так как он является более читаемым, занимает меньше места и требует меньше ресурсов для обработки.

В качестве сервера может выступать любое программное обеспечение, которое никак не должно зависеть от клиента. Это может быть разработка с помощью нативных инструментов выбранного языка программирования или использование фреймворков и библиотек, которые представляют уже готовую базу для быстрой разработки компонент и сервисов, характерных для серверной части приложения. В данном случае будет использован фреймворк Spring Boot 2.0 [7], который является более упрощенной версией Spring и язык программирования Java. Он представляет собой подключаемые библиотеки с помощью *pot.xml*, которыми могут выступать подключение к сети, использование различных баз данных, включая NoSQL решения, а также авторизация пользователя и другие. Spring является программным обеспечением с открытым исходным кодом, что позволяет детально изучить каждый используемый в проекте компонент. Так же стоит отметить о такой вещи, как Dependency Injection, известной как внедрение зависимостей, которая представляет доступ другим компонентам при помощи аннотаций, что

существенно упрощает процесс как написания кода, так и его тестирования. Не стоит забывать о не менее важном преимуществе, а именно большая база готовых проектов, которые выдерживают огромную нагрузку. Этот факт говорит о том, что данный фреймворк является хорошо расширяемым и гибким. Благодаря этим и другим преимуществам, Spring стал одним из самых популярных средств для проектирования и создания серверной части разных приложений.

Как правило, архитектура Spring-приложения основывается на трёх так называемых слоях (Рисунок 1).

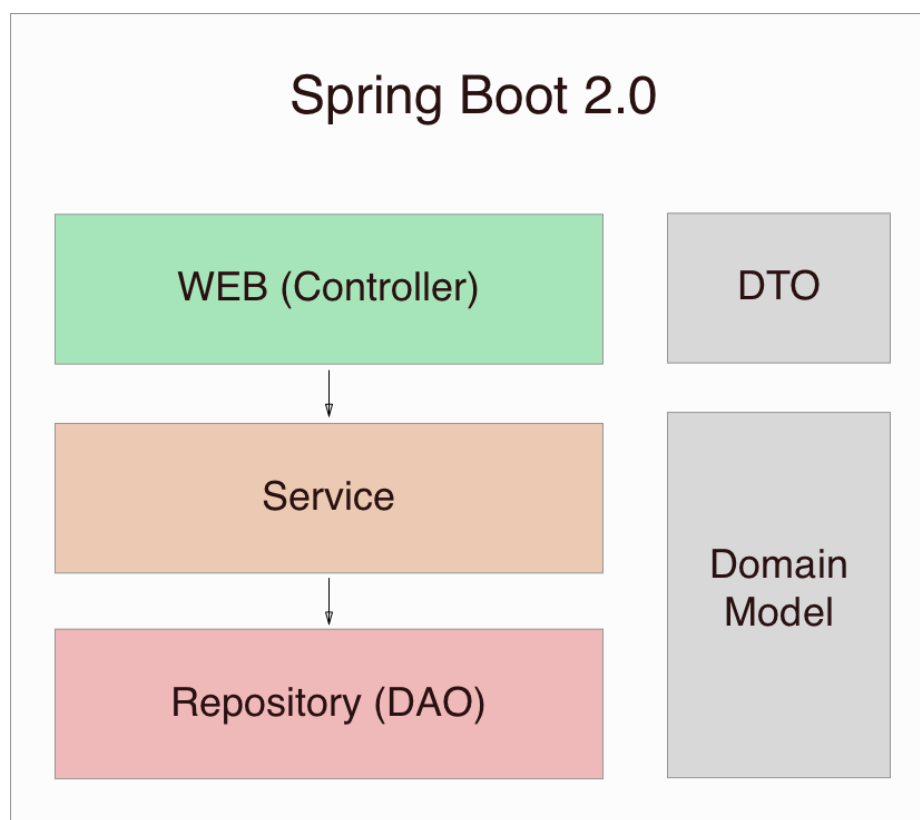


Рисунок 1 – Архитектура Spring Boot приложения

Разберём более подробно каждый из них.

- а) **слой репозитория** (*repository*) или **слой DAO** (*Data access object*). Это классы, которые отвечают за саму передачу данных с БД. Они помечаются аннотацией `@Repository` и, как правило, создаются для каждой сущности в базе данных. Один метод в репозитории – это одна транзакция, которая может быть реализована с помощью сторонних библиотек, или написана собственноручно. Для упрощения разработки будет использоваться

комбинация Hibernate + JPA, которая позволяет определить сущности базы данных с помощью классов-моделей, а JPA сгенерирует основные элементарные операции, выполняемые с таблицей в БД. Это избавит от написания похожих SQL запросов для каждой сущности. Однако, если средствами JPA невозможно решить поставленную задачу, сам запрос можно определить в аннотации метода, который будет выполняться при каждом вызове данной функции.

- б) Следующим слоем является **сервисы** (*service*). Данные классы помечаются аннотацией *@Service* и отвечают за бизнес-процессы приложения. Изначально создается интерфейс, в котором определяются методы, где один метод отвечает за один единственный процесс. Такой подход упрощает разработку и отладку больших и сложных приложений, а также их становится легко расширять и поддерживать. Далее определяются класс, который реализует ранее определённый интерфейс. Реализуемые методы должны быть простыми и содержать минимум действий.
- в) Последний слой – **контроллеры** (*controller*). Такие классы помечаются аннотацией *@RestController*, а каждый метод аннотацией *@RequestMapping*, который содержит информацию о типе запроса, его пути и пересылаемых данных. Слой контроллеров является последним связующим звеном между сервером и клиентом. Например, если сервер должен вернуть всех пользователей, то сперва вызывается метод в контроллере, который вызывает соответствующий метод сервиса, а тот в свою очередь вызывает метод репозитория, который получает всех пользователей из базы данных.

Такой подход крайне рекомендуется при разработке серверных приложений. Кроме всего, на диаграмме (Рисунок 1) также изображен слой с пометкой DTO. Он используется для того, чтобы отдавать более полную информацию о запрашиваемых объектах. Так как слой сервисов и репозитория использует классы-модели, то в слое контроллеров присутствуют конвертеры, которые переводят данные в классы DTO. Это позволяет избежать дополнительных запросов к серверу о вспомогательных данных изначально

запрашиваемого объекта, а отдать сразу полную информацию о нём. На основе этих трёх слоев и будет строиться архитектура серверной части разрабатываемой социальной сети.

2.2 Проектирование базы данных

Следующим важным шагом будет определение сущностей социальной сети, выбор и проектирование базы данных. Результатом будет являться ER-модель данных.

С помощью нее можно выделить основные сущности и связи между ними посредством первичных и внешних ключей. Также для каждого объекта можно определить его атрибуты, которые будут служить для его описания и выступать в качестве полей таблицы. После создания модели данных «сущность-связь» могут быть порождены все существующие модели данных: иерархическая, сетевая, реляционная и другие.

В качестве базы данных будет выбрана объектно-реляционная СУБД PostgreSQL [6] с открытым исходным кодом. Во-первых, она является самой популярной базой данных, которые разработчики используют в своих проектах. Во-вторых, PostgreSQL поддерживает огромный набор вспомогательных возможностей для хранения данных, вплоть до создания новых типов и поддержки JSON. Кроме всего, она позволяет хранить огромные массивы данных и обладает отказоустойчивостью. Все эти возможности PostgreSQL вместе со Spring позволяют создать очень гибкое и быстрое серверное приложение для любых нужд.

Далее необходимо выделить все сущности и их атрибуты. В социальной сети центральным объектом будет служить «User», который будет отвечать за представление одного пользователя в социальной сети. В качестве идентификатора и первичного ключа будет использоваться UUID (universally unique identifier), который состоит из 16-байтного номера и будет генерироваться базой данных при вставке новой сущности. Для всех остальных таблиц будет использован аналогичный подход. В качестве атрибутов

определим имя пользователя, его логин и пароль, который будет хешироваться и представлен в виде нечитаемой строки. Следовательно, для сущности «User» были выделены следующие атрибуты:

- идентификатор;
- полное имя;
- логин;
- пароль.

Для описания более подробной информации о пользователе, будет использоваться сущность «User profile», которая будет ссылаться на таблицу «User» и иметь связь «один к одному», так как у одного пользователя может быть только один профиль с дополнительной информацией. Так же будет присутствовать связь с сущностью, обозначающей загружаемые файлы в базу данных, которая будет определена ниже. Она нужна для обозначения фотографии пользователя. Подводя итог, можно сделать вывод, что данная сущность будет содержать следующие атрибуты:

- идентификатор;
- профессия;
- дополнительная информация;
- роль;
- идентификатор пользователя;
- идентификатор файла для фотографии профиля.

Непосредственно для файлов определим сущность «File entity», которая будет содержать информацию о файле. Объявим её дочерней к «User» со связью «один ко многим», так как один пользователь может загружать много файлов. Из атрибутов будут выделены только основные характеристики загружаемого объекта:

- идентификатор;
- название;
- данные файла;
- время создания;

- тип файла;
- идентификатор автора.

Основная функция социальной сети – это общение между людьми. Как правило, диалоги могут вести между собой два пользователя и более. В нашем случае для мобильного приложения ограничимся именно двумя пользователями для простоты реализации и уменьшения потребляемых ресурсов на стороне клиента. Для этих целей определим сущность «Chat room», которая будет содержать информацию о двух собеседниках при помощи их идентификаторов:

- идентификатор;
- идентификатор первого пользователя;
- идентификатор второго пользователя.

Для обозначения пересылаемых сообщений в чате будет использоваться сущность «Chat message». Одно сообщение может содержать текст или один файл, который будет ссылаться на сущность «File entity». Остальные атрибуты будут содержать информации об авторе и диалоге:

- идентификатор;
- текст сообщения;
- дата;
- идентификатор диалога;
- идентификатор автора;
- идентификатор файла.

Аналогичной важной функцией социальной сети является добавление и разделение пользователей по разным группам, в том числе и друзьям. Для медицинских целей это не лучшая идея, поэтому было решено оставить только группы избранных пользователя. Это сделано для того, чтобы врач мог в любой момент найти интересующего его пациента. Созданная сущность «Favorite users» будет схожа по структуре к диалогам и содержать атрибуты только о двух пользователях:

- идентификатор;

- идентификатор первого пользователя;
- идентификатор второго пользователя.

Следующим этапом будет преобразование ER-модели данных в реляционную. Для этого обозначим для всех сущностей возможности нулевых значений, определим типы полей и выделим первичные и внешние ключи при помощи таблиц (Таблицы 2-7). В качестве типов будут использоваться те значения, которые характерны для базы данных PostgreSQL. При обращении к БД из Java-кода, все типы будут преобразованы к их аналогам.

Таблица 2 – Описание сущности «User»

Атрибут	Тип	Ключ
Id	UUID	Primary key
Username	VARCHAR(100)	-
Name	VARCHAR(100)	-
Password	VARCHAR(100)	-

Таблица 3 – Описание сущности «User profile»

Атрибут	Тип	Ключ
Id	UUID	Primary key
User id	UUID	Foreign key
Avatar file id	UUID	Foreign key
Profession	VARCHAR(100)	-
Description	VARCHAR(1000), NULLABLE	-
Role	VARCHAR(100)	-

Таблица 4 – Описание сущности «File entity»

Атрибут	Тип	Ключ
Id	UUID	Primary key
Author id	UUID	Foreign key
Title	VARCHAR(100)	-

Data	BLOB, NULLABLE	-
Update time	TIMESTAMP	-
File type	VARCHAR(100)	-

Таблица 5 – Описание сущности «Chat room»

Атрибут	Тип	Ключ
Id	UUID	Primary key
First user id	UUID	Foreign key
Second user id	UUID	Foreign key

Таблица 6 – Описание сущности «Chat message»

Атрибут	Тип	Ключ
Id	UUID	Primary key
Author id	UUID	Foreign key
File entity id	UUID	Foreign key
Text	VARCHAR(1000), NULLABLE	-
Date	TIMESTAMP	-
File type	VARCHAR(100)	-

Таблица 7 – Описание сущности «Favorite users»

Атрибут	Тип	Ключ
Id	UUID	Primary key
From user id	UUID	Foreign key
To user id	UUID	Foreign key

Последним шагом является построение диаграммы, в которой каждый прямоугольник обозначает сущность и его атрибуты. Связи между сущностями обозначаются линиями, концы которых помечены метками для обозначения типов связи (Рисунок 2).

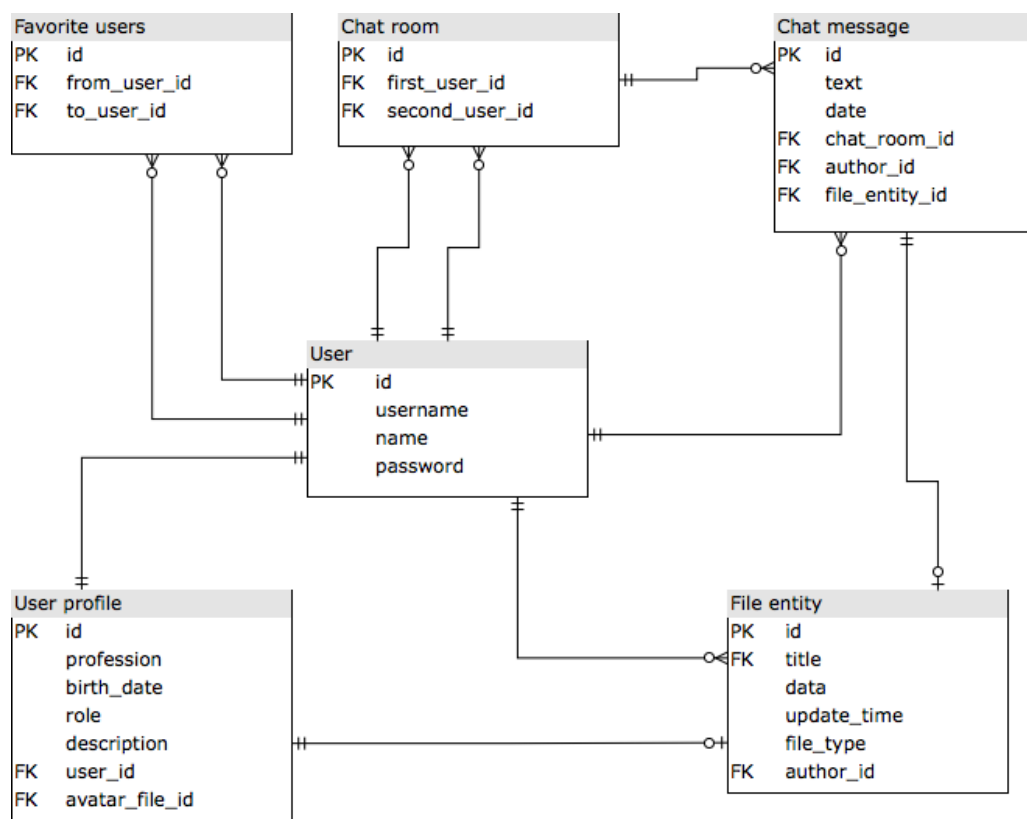


Рисунок 2 – Реляционная модель базы данных для медицинской социальной сети

Реляционная модель позволяет отобразить структуру базы данных при помощи одной диаграммы. Это способствует лучшему пониманию всех внутренних процессов, происходящих между сущностями, а также быстрому выявлению и исправлению найденных проблем и недостатков.

2.3 Клиентская часть

Последним важным этапом перед началом разработки является выбор архитектуры мобильного приложения, который в свою очередь будет играть роль клиента. В постановке задачи была определена платформа, а именно Android-устройства.

Как правило, разработка приложений для операционной системы Android строится на основе двух компонентов. Первая из них, это *layout* файлы, которые основываются на языке разметки XML и отвечают за то, как именно приложение будет выглядеть на экране устройства. Они ответственны за отрисовку всех элементов, их расположения и стили. Вторым компонентом

являются файлы *Activity/Fragment*, которые отвечают за то, что будет отображаться на экране. Это обычные классы, которые наследуются от Android-компонент и реализуют определённый набор методов и могут быть реализованы на языке программирования Java или Kotlin. С помощью такой комбинации становится возможным создать приложение любой сложности. Но за второй компонентой в больших приложениях может стоять нечто большее, чем один файл. Как правило, это классы, отвечающие за внутреннее хранилище, взаимодействие с сетью, логику с данными и другое. Очень долгое время для Android-приложений не было чёткой инструкции и рекомендаций, как правильно все должно взаимодействовать между собой. Создавались сторонние библиотеки для этих целей, но в них были много недостатков и проблем. И только в 2017 году на конференции Google I/O 2017 создатель операционной системы Android представил новую архитектуру приложений, которую должны использовать разработчики в своих проектах.

Android Architecture Components [9] – это новая библиотека, которая представляет набор компонент для управления жизненными циклами Activity и Fragment. Она позволяет сохранять состояния объектов, избегать утечек памяти и легко загружать данные для их отображения в UI (Рисунок 3).

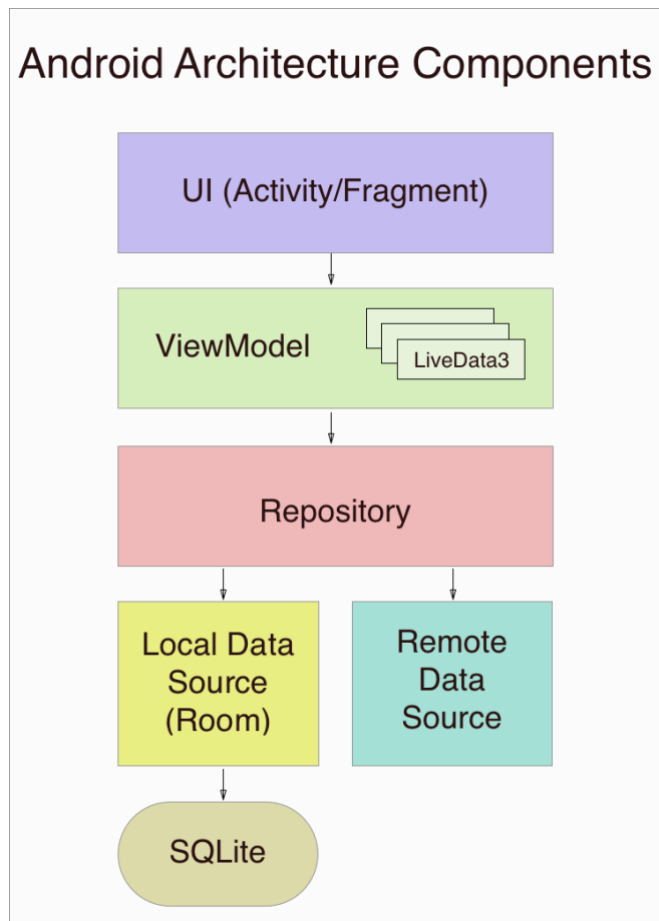


Рисунок 3 – Схема Android Architecture Components

Разберем более подробно каждый из слоев. Самый верхний – это классы, отвечающие за то, какие данные будут отображаться. Он был разобран выше.

Следующий – слой `ViewModel`. Он помогает избежать одну из самых больших проблем Android-приложения. Оно может находиться в разных состояниях своего жизненного цикла, от создания представления, до перехода в ждущий режим, когда приложение уходит в фон при звонке на мобильное устройство или другие глобальные действия. При смене таких состояний `Activity` или `Fragment` не запоминают состояние новых объектов и создают их заново. Существуют инструменты, которые позволяют их сериализовать и восстановить, но чем больше таких объектов будет использоваться, тем больше ресурсов понадобится на работу с ними. Именно поэтому `ViewModel` позволяет обернуть любые данные в класс `LiveData`, который будет сохранять их состояния в течение всего действия `Activity` и `Fragment`. Это мощный инструмент, который позволяет очень просто обойти недостатки данной

платформы и избавиться от утечек памяти, вызванных созданием и удалением объектов на разных состояниях жизненного цикла.

Дальше идет слой Repository, который очень похож на одноимённый слой в архитектуре сервера, но имеет некоторые отличия. Репозиторий может обращаться как к внутреннему хранилищу, который в свою очередь будет представлен компонентом Room для транзакций с базой данных SQLite и внешнему хранилищу. Он будет представлен библиотекой Retrofit 2 [10] для обмена данными с сервером при помощи HTTP-запросов. Так как, структура каждого из них довольно сложная, то Room и Retrofit 2 будут более подробно рассмотрены в реализации приложения.

Полученная архитектура социальной медицинской сети представлена ниже (Рисунок 4).

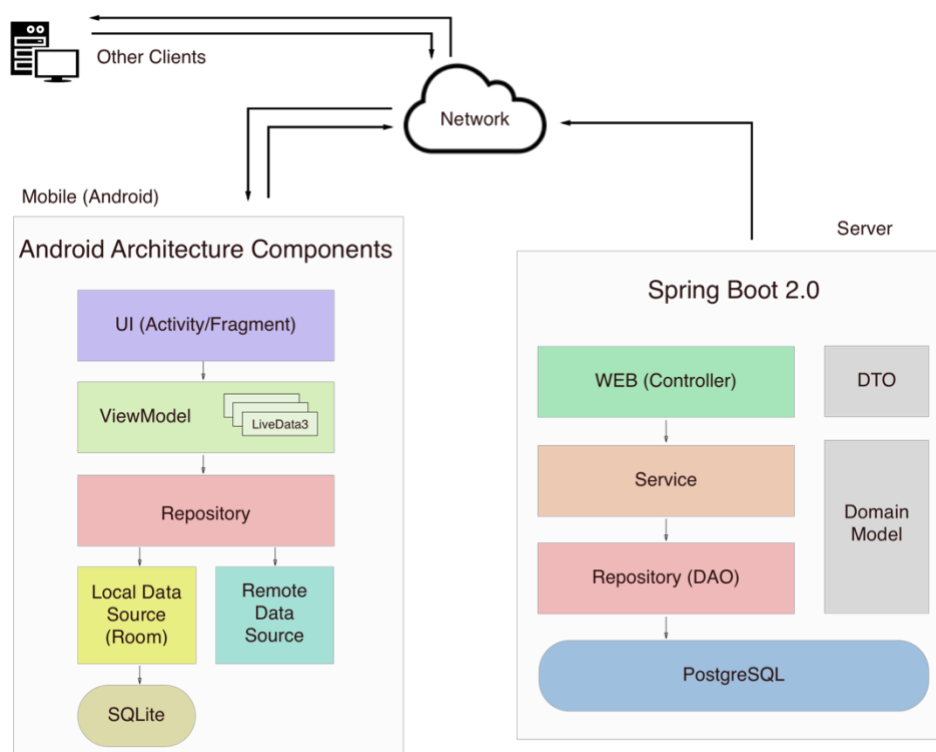


Рисунок 4 – Архитектура социальной медицинской сети

Ключевым моментом архитектуры клиент-сервер является то, что каждая часть может быть построена каким угодно образом и разрабатываться параллельно двумя разными командами. Необходимо согласовать только формат сообщений, которыми будут обмениваться клиент и сервер.

2.4 Разработка приложения на примере пользователя

Так как разработка всех компонентов, соответствующим сущностям в базе данных, довольна однотипна для выбранной архитектуры, то далее будет представлены все действия, связанные с пользователями, начиная с базы данных и заканчивая отображением информации на экране мобильного устройства.

Для управления и изменения состояний таблиц будет использована библиотека Liquibase [11]. Она помогает осуществлять быстрые миграции баз данных. Для этого для одной операции создается *changeSet* на языке XML, который может выполняться при определённых условиях. При создании таблицы «User» необходимо, чтобы такой таблицы не существовало, далее постепенно определяются поля и их свойства, которые соответствуют ER-модели данных (Листинг 1).

Листинг 1 – Создание таблицы «User» при помощи Liquibase

```
1. <changeSet id="create-table-user" author="belogurow">
2.   <preConditions onFail="MARK_RAN">
3.     <not>
4.       <tableExists tableName="user"/>
5.     </not>
6.   </preConditions>
7.   <createTable tableName="user">
8.     <column name="id" type="UUID">
9.       <constraints primaryKey="true"/>
10.    </column>
11.
12.    <column name="username" type="VARCHAR(100)">
13.      <constraints nullable="false" unique="true"/>
14.    </column>
15.
16.    <column name="name" type="VARCHAR(100)">
17.      <constraints nullable="false"/>
18.    </column>
19.
20.    <column name="password" type="VARCHAR(100)">
21.      <constraints nullable="false"/>
22.    </column>
23.
24.  </createTable>
25. </changeSet>
```

Далее необходимо создать класс-модель для представления пользователя. Для того, чтобы JPA смогло автоматически построить SQL запросы для данной модели, все поля обозначаются с помощью аннотаций Hibernate (Листинг 2).

Листинг 2 – Класс User

```
1.  @Entity
2.  @Table(name = "user", schema = "public")
3.  public class User implements Serializable {
4.
5.      @Id
6.      @NotNull
7.      @GeneratedValue(generator="uuid")
8.      @GenericGenerator(name = "uuid", strategy = "uuid2")
9.      @Column(name = "id", unique = true)
10.     private UUID id;
11.
12.     @NotNull
13.     @Column(name = "username", unique = true)
14.     private String username;
15.
16.     @NotNull
17.     @Column(name = "name")
18.     private String name;
19.
20.     @NotNull
21.     @Column(name = "password")
22.     private String password;
23.
24.     public User() {
25.     }
26.
27.     // getters and setters
28.     // ...
29. }
```

Следующим шагом является создание репозитория `UserRepository` и определения в нём вспомогательных SQL запросов для поиска пользователя по логину и условия, существует ли пользователь по передаваемому логину (Листинг 3).

Листинг 3 – Интерфейс UserRepository

```
1.  @Repository
2.  public interface UserRepository extends JpaRepository<User, UUID> {
3.
4.      boolean existsByUsername(String username);
5.
6.      User findByUsername(String username);
7.  }
```

Далее создадим сервис `UserService`, в котором заменим методы для вставки новой сущности на регистрацию и вход в социальную сеть, так как слой сервисов прежде всего отвечает за бизнес-логику. Остальные методы будут обращаться к аналогичным методам в репозитории (Листинг 4).

Листинг 4 – Реализация интерфейса UserService

```
1. @Service
2. public class UserServiceImpl implements UserService {
3.
4.     @Autowired
5.     private PasswordEncoder passwordEncoder;
6.
7.     @Autowired
8.     private UserRepository userRepository;
9.
10.    @Override
11.    public User login(User user) throws CustomException {
12.        Optional<User> userFromDB =
13.            this.findByUsername(user.getUsername());
14.
15.        if (userFromDB.isPresent()) {
16.            if (passwordEncoder.matches(user.getPassword(),
17.                userFromDB.get().getPassword())) {
18.                // Login Success
19.                return userFromDB.get();
20.            } else {
21.                // Incorrect password
22.                throw new
23.                    CustomException(ErrorCode.PASSWORD_INCORRECT);
24.            }
25.        } else {
26.            // Incorrect login
27.            throw new CustomException(ErrorCode.LOGIN_INCORRECT);
28.        }
29.    }
30.
31.    @Override
32.    public User registration(User user) throws CustomException {
33.        return this.save(user);
34.    }
35.
36.    @Override
37.    public User save(User user) throws CustomException {
38.        Optional<User> userFromDB =
39.            this.findByUsername(user.getUsername());
40.
41.        if (userFromDB.isPresent()) {
42.            throw new CustomException(ErrorCode.LOGIN_EXISTS);
43.        } else {
44.            // Create new user
45.            user.setPassword(passwordEncoder.encode(user.getPassword()));
46.            user.setUserRole(UserRole.USER);
47.            return userRepository.saveAndFlush(user);
48.        }
49.    }
50.
51.    @Override
52.    public Optional<User> findByUsername(String username) {
53.        if (this.existsByUsername(username)) {
54.            return Optional.of(userRepository.findByUsername(username));
55.        }
56.        return Optional.empty();
57.    }
```

```
57.     }  
58.     // ...  
59.     }
```

Разберём более подробно реализацию сервиса. Аннотации `@Autowired` (Листинг 4 строки 4, 7) являются частью `Dependency Injection`, и при компиляции проекта Spring сам создаст экземпляры для объектов, помеченных этой аннотацией. Не менее интересным является реализация метода `login()` для входа пользователей в социальную сеть. На вход метод получает экземпляр класса `User` с заполненными полями `login` и `password`. Первым делом, необходимо узнать, если ли пользователь с таким именем в базе данных (Листинг 4 строка 12). Для этого используется класс `Optional`, который был добавлен в Java 8 версии. Он представляет собой обертку для любого объекта, созданный для защиты от `null` значений. Таким образом, прежде чем обращаться напрямую к значению объекта, необходимо узнать при помощи метода `isPresent()` существует ли объект на самом деле. Такой подход был создан для избавления от популярной ошибки `NullPointerException`. Таким образом, если пользователь с таким именем существует и его пароль правильный, то метод возвращает экземпляр метода `User` из `Optional` при помощи метода `get()` с полностью заполненными полями.

Также возможны случаи, когда такого пользователя может не существовать или пароль не будет совпадать с правильным. Для этого был создан класс `CustomException` (Листинг 5), который при возникновении исключения будет передаваться напрямую в клиент с кодом и сообщением ошибки.

Листинг 5 – Класс `CustomException` для представления пользовательских исключений

```
1.  public class CustomException extends Exception {  
2.  
3.      private final ErrorCode errorCode;  
4.  
5.      public CustomException(ErrorCode errorCode) {  
6.          super(errorCode.toString());  
7.          this.errorCode = errorCode;  
8.      }  
9.
```

```
10.    public CustomException(String message, Throwable cause, ErrorCode code) {  
11.        super(message, cause);  
12.        this.errorCode = code;  
13.    }  
14.  
15.    public CustomException(String message, ErrorCode code) {  
16.        super(message);  
17.        this.errorCode = code;  
18.    }  
19.    public CustomException(Throwable cause, ErrorCode code) {  
20.        super(cause);  
21.        this.errorCode = code;  
22.    }  
23.  
24.    public ErrorCode getErrorCode() {  
25.        return errorCode;  
26.    }  
27. }
```

Последним этапом перед отправкой данных с сервера является прохождение их через слой контроллера. Ранее было сказано, что все классы-модели на данном слое преобразуются в классы DTO. На рисунке 5 будет представлено отличие в структурах данных классов на основе UML-диаграммы.

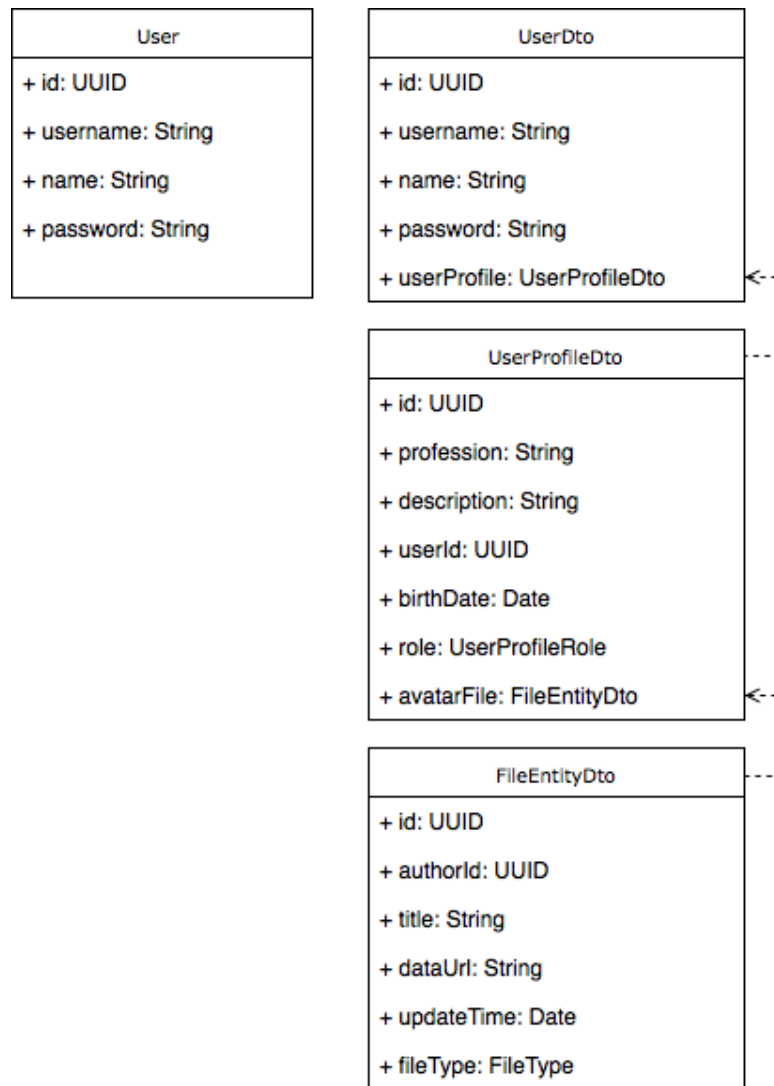


Рисунок 5– Сравнение структур классов User и UserDto

Как видно на рисунке выше, во втором случае будет возвращено с сервера намного больше информации в JSON-формате, что поможет сэкономить количество запросов, отправляемых с клиента.

В слое контроллера аннотации *@RequestMapping* у каждого метода объявляют путь, по которому будет доступен запрос. Например, метод *registration()* (Листинг 6 строки 9-16) будет доступен по следующему адресу – “URL/registration”, где URL – это адрес, по которому доступен сервер. В качестве тела запроса передается экземпляр класса *User*. Spring берет много работы на себя, поэтому не требуется собственноручно разбирать приходящий JSON и настраивать многочисленные соединения, все это работает изначально. Каждый метод логируется при помощи класса *Logger*, который при

возникновении ошибок или для отладки работы сервера поможет проследить последовательность всех действий. Также обратим внимание на метод *findAll()*, который возвращает всех пользователей социальной сети (Листинг 6 строки 35-43). Он написано в функциональном стиле при помощи Stream API. Изначально, для конвертации результата сервиса пришлось бы писать цикл, который бы конвертировал каждый экземпляр класса *User* в *UserDto*. Теперь этого можно избежать, и при помощи метода *map()* провести конвертацию на прямую. Так как, в конце необходимо вернуть список полученных элементов, то для этого используется метод *collect()*. При помощи Stream API стало возможным упростить большие участки кода и увеличить их быстродействие.

Листинг 6 – Класс UserController

```
1.  @RestController
2.  public class UserController {
3.
4.      private static Logger LOGGER =
5.  LoggerFactory.getLogger(UserController.class);
6.
7.      private ConvertUser2UserDto convertUser2UserDto;
8.      private UserService userService;
9.
10.     @RequestMapping(value = "/registration", method = RequestMethod.POST)
11.     public ResponseEntity registration(@RequestBody User user) throws
12. CustomException {
13.         LOGGER.info("registration({})", user);
14.
15.         return ResponseEntity.ok(convertUser2UserDto.convert(
16.             userService.registration(user)));
17.     }
18.
19.     @RequestMapping(value = "/login", method = RequestMethod.POST)
20.     public ResponseEntity login(@RequestBody User user) throws CustomException {
21.         LOGGER.info("login({})", user);
22.
23.         return ResponseEntity.ok(convertUser2UserDto.convert(
24.             userService.login(user)));
25.     }
26.
27.     @RequestMapping(value = "/users/{id}", method = RequestMethod.GET)
28.     public ResponseEntity findUserById(@PathVariable(value = "id") UUID id) {
29.         LOGGER.info("findUserById({})", id);
30.
31.         return userService.findById(id)
32.             .map(user ->
33.                 ResponseEntity.ok(convertUser2UserDto.convert(user)))
34.             .orElseGet(() -> ResponseEntity.notFound().build());
35.     }
36.     @RequestMapping(value = "/users", method = RequestMethod.GET)
```

```

37.    public ResponseEntity findAll() {
38.        LOGGER.info("findAll()");
39.
40.        return ResponseEntity.ok(userService.findAll()
41.            .parallelStream()
42.            .map(user -> convertUser2UserDto.convert(user))
43.            .collect(Collectors.toList()));
44.    }
45. }

```

На этом разработка серверной части для представления пользователя закончилась. Перейдем к его представлению на стороне клиента, а именно на Android-приложении. Сперва определим, как выполняются запросы при помощи библиотеки Retrofit 2 [10]. Изначально для таких целей служили классы *AsyncTask*, но при малейшем шаге в сторону или модификации запроса, все приходилось переписывать заного, что часто являлось причиной возникновения утечек памяти. Retrofit 2 в свою очередь позволяют обернуть все запросы в интерфейс и определить их структуру при помощи аннотаций, что позволяет им быть очень гибкими в настройке (Листинг 7). Стоит отметить, что Google в документации по Android Architecture Components советуют для взаимодействия с интернетом использовать именно эту библиотеку.

Листинг 7 – Интерфейс UserWebService

```

1.    public interface UserWebService {
2.        @POST("/login")
3.        Flowable<UserDto> login(@Body User user);
4.
5.        @POST("/registration")
6.        Flowable<UserDto> registration(@Body User user);
7.
8.        @GET("/users/{id}")
9.        Flowable<UserDto> getUserById(@Path("id") UUID id);
10.
11.        @GET("users")
12.        Flowable<List<UserDto>> getUsers();
13.    }

```

Каждый метод интерфейса возвращает класс *UserDto*, обернутый в класс *Flowable*, который является частью библиотеки для реактивного программирования RxJava 2 [12]. Она будет рассмотрена подробнее в главе про оптимизацию работы приложения. Следуя архитектуре приложения, следующим слоем является репозиторий (Листинг 8), который является

промежуточным пунктом между отправлением запросов и их обработке во *ViewModel*.

Листинг 8 – Класс RemoteUserRepository

```
1. @Singleton
2. public class RemoteUserRepository {
3.     private static final String TAG = RemoteUserRepository.class.getSimpleName();
4.
5.     public Flowable<UserDto> registration(User user) {
6.         Log.d(TAG, "registration: " + user.toString());
7.         return App.sWebUserService.registration(user);
8.     }
9.
10.    public Flowable<UserDto> login(User user) {
11.        Log.d(TAG, "login: " + user.toString());
12.        return App.sWebUserService.login(user);
13.    }
14.
15.    public Flowable<List<UserDto>> getUsers() {
16.        Log.d(TAG, "getUsers: ");
17.        return App.sWebUserService.getUsers();
18.    }
19.
20.    public Flowable<UserDto> getUserById(UUID id) {
21.        Log.d(TAG, "getUserById: " + id);
22.        return App.sWebUserService.getUserById(id);
23.    }
24. }
```

Далее в слое *ViewModel* рассмотрим метод для регистрации пользователя (Листинг 10), так как все остальные методы являются однотипными. Но не стоит забывать, что не все запросы могут быть успешными, поэтому прежде, чем передавать результат в слой UI, необходимо обернуть полученный результат запроса с сервера в дополнительный класс, который при успешном выполнении будет возвращать результат объекта, а при ошибке – её подробное описание (Листинг 9).

Листинг 9 – Класс Resource для обёртки результата выполненного запроса

```
1. public class Resource<T> {
2.     private final NetworkStatus status;
3.     private final T data;
4.     private final String message;
5.
6.     private Resource(NetworkStatus status, T data, String message) {
7.         this.status = status;
8.         this.data = data;
9.         this.message = message;
10.    }
11.
12.    public static <T> Resource<T> success(T data) {
```

```

13.     return new Resource<>(SUCCESS, data, null);
14. }
15.
16.     public static <T> Resource<T> error(ResponseBody errorBody) {
17.         try {
18.             JSONObject object = new JSONObject(errorBody.string());
19.             return new Resource<> (ERROR, null, object.getString("message"));
20.         } catch (JSONException | IOException e) {
21.             e.printStackTrace();
22.             return Resource.error();
23.         }
24.     }
25.
26.     public static <T> Resource<T> error(@NonNull String message) {
27.         return new Resource<>(ERROR, null, message);
28.     }
29.
30.     private static <T> Resource<T> error() {
31.         return new Resource<>(ERROR, null, "Api error. See logs.");
32.     }
33.
34.     // getters and setters
35.     // ...
36. }

```

В классах *ViewModel* все объекты необходимо оборачивать в класс *LiveData*, которые в свою очередь будут содержать результат запроса в обертке класса *Resource*. Получается довольно сложная структура, но это скорее недостатки языка программирования Java. В теле метода регистрации вызывается соответствующий метод репозитория. кроме всего при помощи библиотеки RxJava 2 все происходит в отдельном потоке, что позволяет не блокировать экран мобильного устройства и продолжать реагировать на остальные действия. Именно поэтому для передачи результата в *LiveData* используется метод *postValue()*, при выполнении всех действий в основном потоке необходимо использовать метод *setValue()*.

Листинг 10 – Метод регистрации пользователя в классе *UserViewModel*

```

1.     public LiveData<Resource<UserDto>> registration(User user) {
2.         Log.d(TAG, "registration: " + user.toString());
3.
4.         final MutableLiveData<Resource<UserDto>> liveData = new MutableLiveData<>();
5.         mCompositeDisposable.add(
6.             mRemoteUserRepository.registration(user)
7.                 .subscribeOn(Schedulers.io())
8.                 .observeOn(AndroidSchedulers.mainThread())
9.                 .subscribe(
10.                    userResult -> {
11.                        Log.d(TAG, "registrationRx-result: " + userResult.toString());
12.                    }

```



```

13.         liveData.postValue(Resource.success(userResult));
14.     },
15.     error -> {
16.         Log.d(TAG, "registration-error:" + error.getMessage());
17.
18.         if (error instanceof HttpException) {
19.             liveData.postValue(Resource.error                ((HttpException)
error).response().errorBody());
20.         } else {
21.             liveData.postValue(Resource.error(
22. error.getMessage());
23.         }
24.     })
25. );
26. return liveData;
27. }

```

Последним шагом является отображение полученных данных на UI, для этого в *Activity*, которое отвечает за регистрацию пользователя объявляется метод (Листинг 11), который подписывается на изменения LiveData. Изменения приходят при вызове описанных выше методов *postValue()* или *setValue()*. Если запрос выполнен успешно, то полученные данные передаются дальше, иначе при всплывающего сообщения Toast внизу экрана отображается полученная ошибка. Также приложение должно быть интерактивным и отображать все изменения, происходящие внутри, поэтому процесс отправки запроса и получения результата отображается при помощи вращающегося индикатора выполнения (ProgressBar).

Листинг 11 – Передача результатов запроса регистрации в Activity

```

1. private void initDataListener() {
2.     signUpObserver = userResource -> {
3.         mProgressBarSignUp.setVisibility(View.VISIBLE);
4.
5.         if (userResource == null) {
6.             Toast.makeText(this, R.string.received_null_data,
7. Toast.LENGTH_LONG).show();
8.             mProgressBarSignUp.setVisibility(View.GONE);
9.             return;
10.        }
11.
12.        switch (userResource.getStatus()) {
13.            case SUCCESS:
14.                uploadUserProfile(userResource.getData().getId());
15.                break;
16.            case ERROR:
17.                Toast.makeText(this, userResource.getMessage(Toast.LENGTH_LONG).show();
18.                break;
19.            default:
20.                Toast.makeText(this, R.string.unknown_status, Toast.LENGTH_LONG).show();

```

```
21.     }  
22.  
23.     mProgressBarSignUp.setVisibility(View.GONE);  
24. };  
25. }
```

Выше представлены все действия, с которыми связан класс пользователя. Реализация работы с другими сущностями выполняется при помощи всех тех же инструментов, поэтому на первых взгляд может показаться, что разработка такого комплекса программного обеспечения является довольно простой, но это таковым не является. Реализации каждой компоненты предшествует выполнение большого количества аналитической работы, формирование всевозможных правил поведения пользователя с приложением, и только затем их разработка и обработка граничных случаев, что делает разработку существенно сложнее и дольше.

2.5 Проектирование пользовательского интерфейса

Важным этапом в разработке Android-приложения является проектирование его интерфейса и выявление подробностей для UI (user interface) и UX (user experience) дизайна.

При первом запуске приложения пользователя должно встречать окно аутентификации, где будет предложено ввести свой логин или пароль (Рисунок 6). Если пользователь еще не зарегистрирован, то по нажатии на соответствующую кнопку в интерфейсе приложения откроется окно для авторизации (Рисунок 7).

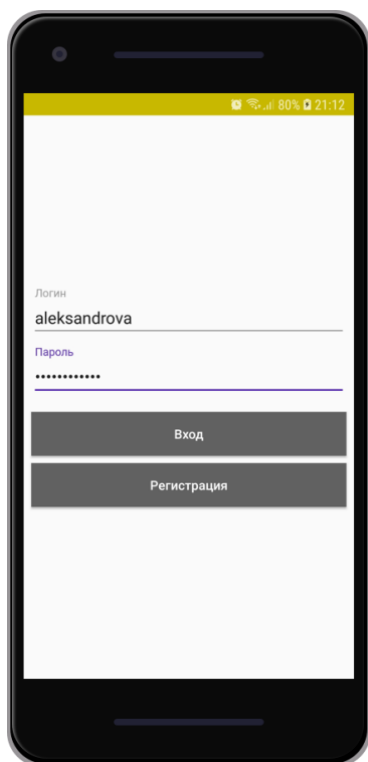


Рисунок 6 – Процесс аутентификации

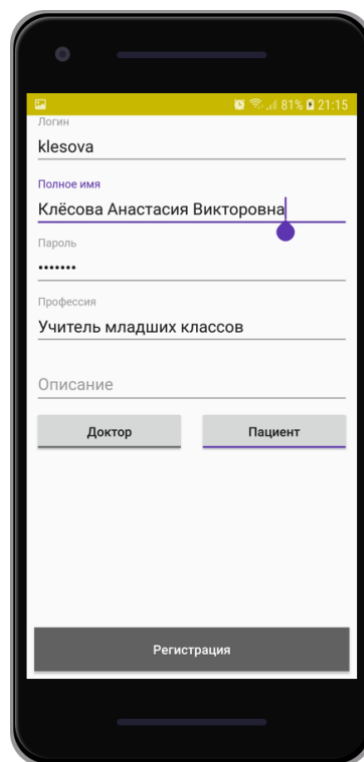


Рисунок 7 – Процесс авторизации

После прохождения процесса аутентификации или авторизации, пользователь попадает в основное меню приложения «Консилиум». Как было описано выше, для навигации по приложению будет использован Navigation Drawer, который будет составлен из 5 пунктов (Рисунок 8). Для его реализации будет использована сторонняя библиотека Material Drawer [13] от Mike Penz, которая позволяет создать выдвигающееся меню при помощи одного кода. В шапке Navigation Drawer будет содержаться имя и логин текущего пользователя, а также фото профиля, если оно было загружено ранее.

Все пункты навигации, кроме первого, будут открывать новое окно, которое будет содержать список элементов выбранной категории. Такие списки позволяет создавать компонент RecyclerView. Не менее интересным выглядит окно диалога между двумя пользователями (Рисунок 9), так как он также был реализован при помощи выше сказанного компонента. Диалог – это тот же список, но с разными элементами. Все сообщения текущего пользователя помещаются справа, а пользователя, с которым ведётся беседа – слева экрана.

Внизу окна можно ввести любое сообщение или при помощи иконки скрепки прикрепить файл. Если был отправлен файл типа JPG, STL или PDF, то по нажатию на него, откроется новое окно с подробной информацией о выбранном файле.

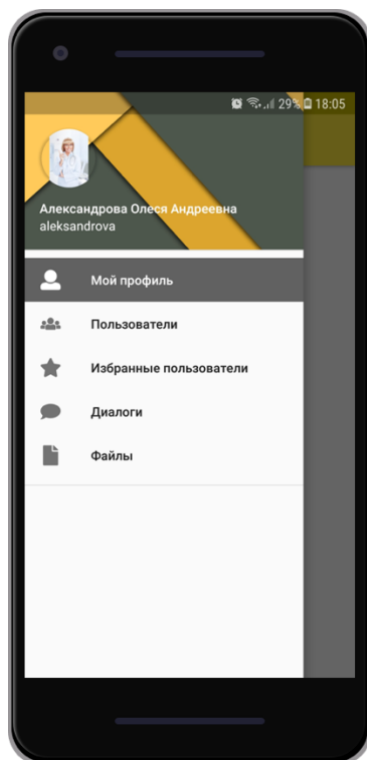


Рисунок 8 – Навигация по приложению

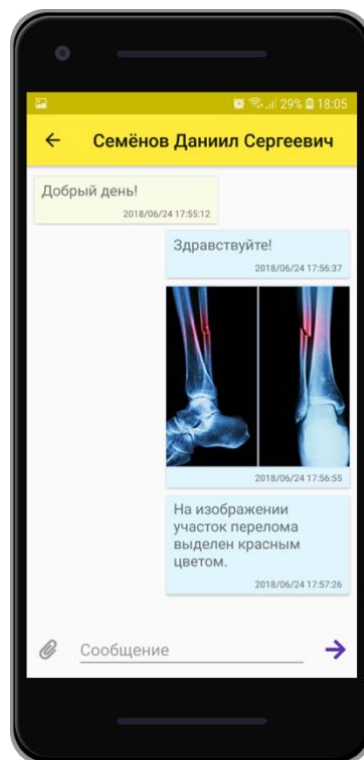


Рисунок 9 – Диалог между пользователями

Для того, чтобы пользователь мог загружать свои файлы в социальную медицинскую сеть, была использована отдельная категория «Файлы». Для загрузки нового файла необходимо нажать на иконку «+», которая позволит загрузить из памяти устройства файл определённых типов и задать ему любое имя. После выбора файла, кнопка «Загрузить» становится активной. По нажатию на неё файл отправляется на сервер, а пользователю предлагается продолжить загрузку новых файлов или закончить. Весь процесс изображен на рисунке 10.

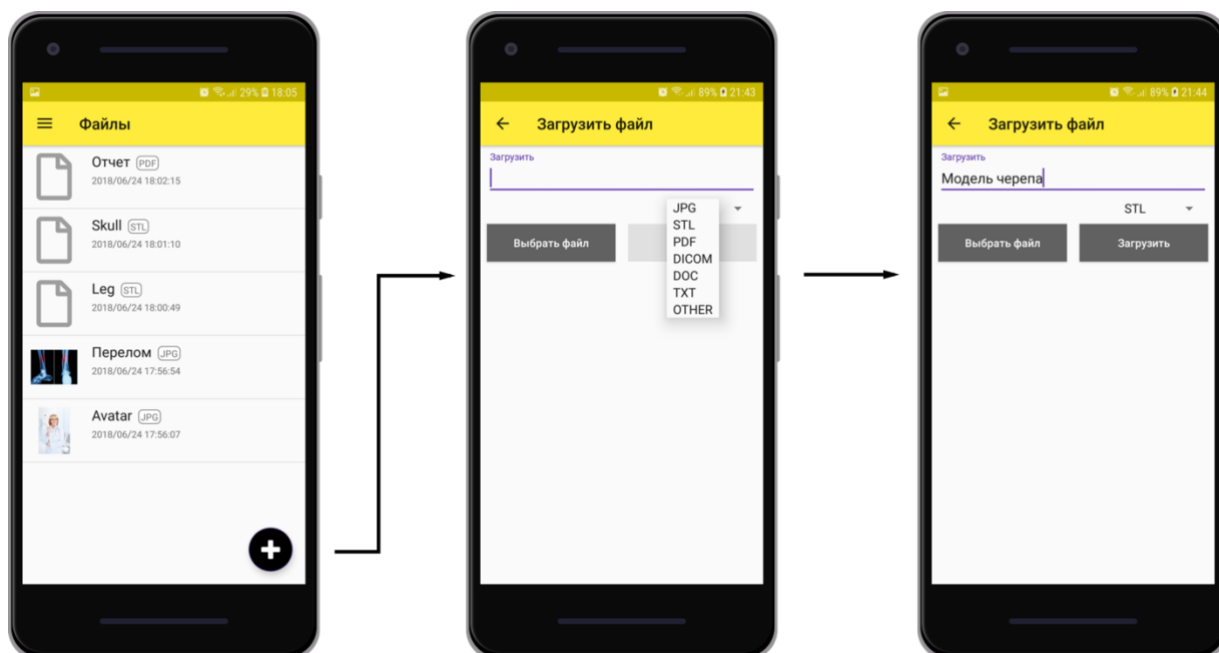


Рисунок 10 – Процесс загрузки нового файла для текущего пользователя

Таким образом, были изображены основные экраны социальной медицинской сети «Консилиум». Все они используют одни и те же компоненты, но в разном исполнении. Это позволяет продемонстрировать гибкость интерфейса Android-приложения.

3 Оптимизация работы приложения

3.1 Реактивное программирование

В современном мире Android-разработки стало очень популярным строить свои приложения на основе принципов реактивного программирования. Разберём более подробно, что это такое и как оно поможет улучшить разрабатываемое приложение.

Реактивное программирование строится на трех основных понятиях [14]:

- a) **Асинхронность** (*Asynchronous*). Это означает, что разные части программы могут работать одновременно.
- b) **Все основано на событиях** (*Event-Based*). Выполнение части кода базируется на событиях, которые произошли во время выполнении программы. Например, нажатие на кнопку запустит событие, которое получит обработчик и выполнит некоторую работу.
- c) **Наблюдаемые последовательности** (*Observable sequences*). Определение будет рассмотрено ниже.

Для использования принципов реактивного программирования в Java необходимо подключить библиотеку RxJava 2, которая обладает открытым исходным кодом и позволяет писать код в функциональном стиле. Она позволяет расширить шаблон наблюдателя и связать последовательности данных или событий вместе декларативно, в то же время абстрагируя проблемы, которые могут быть связаны с безопасностью и низкоуровневой передачей потоков, параллельностью структур данных и неблокирующими устройствами ввода-вывода.

Шаблон проектирования Наблюдатель (*Observer*) – ключ к пониманию данной архитектуры, где на объект подписываются «слушатели» (*listeners*), которые уведомляют и вызывают соответствующие методы при каждом изменении состояния объекта.

На рисунке 11 более подробно показан механизм работы реактивных потоков. Любой объект, который далее будет называться подписчиком

(*subscriber*), может подписаться на любой другой наблюдаемый объект (*observable*).

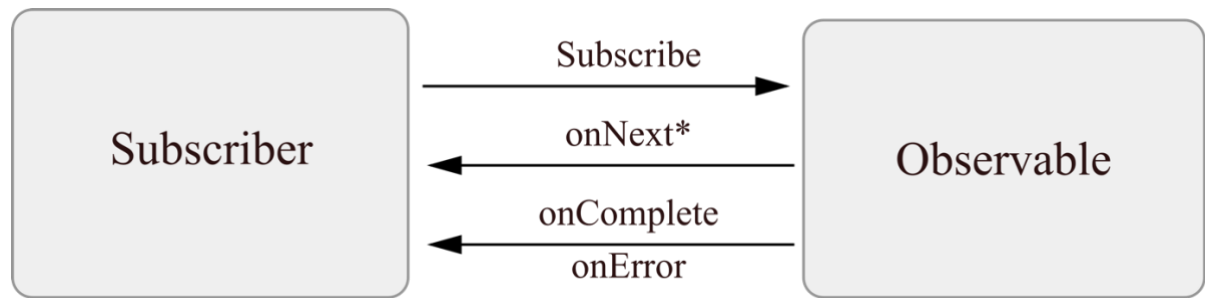


Рисунок 11 – Механизм работы реактивных потоков [14]

При появлении события подписчику предоставляется три метода, которые можно значительно сократить при использовании лямбда-выражений:

- *onNext(T t)* – вызывается если объект получен успешно;
- *onError(Throwable t)* – вызывается, если во время события произошла ошибка;
- *onComplete()* – вызывается при завершении обработки всех объектов.

Таким образом, получаемый объект необходимо обернуть в одну из структур RxJava 2. Самыми популярными из них являются *Flowable* и *Observable*. Они испускают 0 или *N* элементов, а затем завершаются успешно или с ошибкой. Единственное их отличие в том, что *Flowable* обладает механизмом *Backpressure*.

Backpressure описывает ситуацию, когда объекты поступают намного быстрее, чем оператор успевает их обработать. Без данного механизма такие ситуации могут привести к потере части объектов или к возникновению ошибки *OutOfMemoryError*.

Как было описано в предыдущем разделе, отправленные HTTP запросы при помощи библиотеки Retrofit 2 могут использовать так же принципы реактивного программирования при помощи комбинации с RxJava 2. Пример такого взаимодействия можно увидеть в листинге 7, где все методы возвращают *Flowable*. Далее в листинге 10 в строках 7-8 происходит

взаимодействие с потоками. Чуть ниже лямбда-выражения *userResult* и *error* являются аналогами методов *onNext()* и *onError()*.

Собрав всю выше описанную информацию воедино можно сделать вывод о том, что использование принципов реактивного программирования помогает использовать дополнительные возможности и упрощать уже существующие функции. Главным недостатком RxJava 2 является её сложность в освоении и изучении всех внутренних процессов, без понимания которых можно навредить быстродействию своего приложения. Аналогом этой библиотеки в языке программирования Kotlin являются *coroutines*, которые намного проще в освоении и позволяют использовать можно возможностей для взаимодействия с потоками.

3.2 Сжатие изображений

Следующим важным этапом в оптимизации работы приложения является сжатие картинок, при первой загрузке на сервер. Оно нужно для того, чтобы при их просмотре на стороне клиента загружалось меньше данных. Необходимо найти ту грань, чтобы новое изображение можно было комфортно просматривать без потери важных элементов, и его размер потреблял заметно меньше ресурсов, чем размер до сжатия.

Во время поиска решений этой задачи была найдена библиотека ImageIO из стандартного набора Java JDK, которая предоставляет набор инструментов для работы с изображениями. Процесс начинался с выбора режима сжатия и его численного значения, затем открытия потока данных *OutputStream* и записи в него нового изображения. Через некоторое время обнаружились и недостатки данной библиотеки. Все дело в том, что специальный стандарт EXIF позволяет добавлять к различным медиа файлам дополнительную информацию, включая его ориентацию, дату и устройство съёмки, а также многое другое. При взаимодействии с ImageIO эти данные стираются и восстановить их невозможно. Таким образом, при загрузке фотографии в вертикальной

ориентации и её последующим сжатии, результат мог иметь горизонтальную ориентацию. Поэтому пришлось искать данной библиотеке замену.

Выбор пал на Thumbnailator [15] версии 0.4.8. Она предоставляет более удобный интерфейс для взаимодействия с изображением, и что самое главное – сохраняет информацию EXIF. Для этого было написан отдельный класс (Листинг 11), который на вход принимает только числовое значение степени сжатия изображения. В дальнейшем данный класс можно вынести в отдельный пакет и использовать в других проектах.

Листинг 11 – Класс ImageCompression

```
1. public class ImageCompression {  
2.  
3.     public static byte[] compress(byte[] data, float imageQuality) throws IOException {  
4.         try (InputStream inputStream = new ByteArrayInputStream(data);  
5.             OutputStream outputStream = new ByteArrayOutputStream()) {  
6.  
7.             Thumbnails.of(inputStream)  
8.                 .scale(0.9)  
9.                 .outputQuality(imageQuality)  
10.                .toOutputStream(outputStream);  
11.  
12.            return ((ByteArrayOutputStream) outputStream).toByteArray();  
13.        }  
14.    }
```

В методе сжатия используется конструкция *try-with-resources*, которая позволяет обернуть в *try*-блок объекты *closeable*. При выходе из блока для них автоматически будет вызван метод *close()*. Данная конструкция позволяет правильно освободить ресурсы и не беспокоиться об возникновении утечек памяти. Сама библиотека довольно проста в использовании (Листинг 10 строки 7-10). Размер исходного изображения становится равным параметру *imageQuality*, а его разрешение становится равным 90% от исходного. Результат сжатия продемонстрирован на рисунке 12.

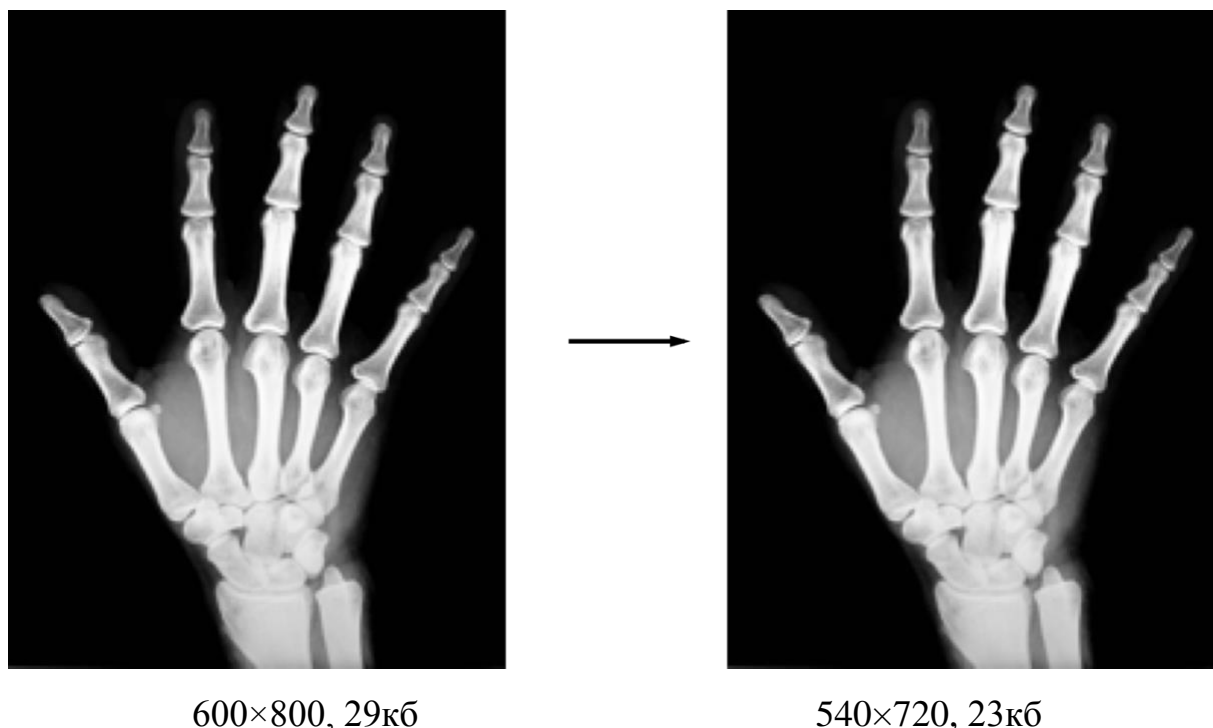


Рисунок 12 – Сжатие изображения на примере снимка кисти ладони [16]

Как видно выше, результат сжатой картинке визуально не отличить от изначальной. Однако, её размер и разрешение поменялись в меньшую сторону. При больших изображениях это поможет сэкономить большое количество ресурсов при отправке HTTP запросов.

3.3 Просмотр STL-моделей

Приложение социальной медицинской сети «Консилиум» предполагает просмотр загруженных STL-моделей прямо на экране мобильного устройства. Стандартных средств в операционной системе Android для таких целей не существует.

В браузерах для просмотра STL-файлов используются разные библиотеки, наиболее популярной из которых является ThreeJS [18]. Она позволяет создавать сцены любых размеров, прикреплять к ним разные источники света, загружать объекты и организовывать их взаимодействие между собой. ThreeJS поддерживает файлы разных форматов и STL в том числе. Таким образом, внутри Android-приложения необходимо открыть WEB

страницу, которая с помощью данной библиотеки будет показывать загруженную модель.

Для таких случаев на экране приложения можно разместить элемент под названием `WebView` [17], который позволяет загружать страницы из интернета или те, которые помещены в корень самого приложения. Грубо говоря, открывается встроенный браузер устройства, ему передается адрес страницы, который он загружает и отображает прямо внутри приложения.

Далее необходимо определить, как передавать загружаемую STL-модель с сервера прямо в `WebView`. Для этого было найдено два решения. Первый подразумевает передачу URL-адреса в этот компонент, чтобы сама страница посылала запрос на сервер и на прямую загружала полученный объект. Второй предполагает загрузку модели в кэш приложения, и далее передачу расположения файла в `WebView`. Был выбран второй метод, так как загрузку файла можно будет проводить с помощью уже используемых библиотек, а отображаемый компонент освободится от лишней трудоёмкой работы.

Просто так сохранить объект в кэш не получится. Для этого необходимо разрешения от пользователя или так называемые *user-permission* – это всплывающие окна, в которые спрашивается разрешения на доступ к дополнительным функциям мобильного устройства. Изначально, все *user-permission* объявлялись в файле *AndroidManifest.xml*, который содержит всю информацию о компонентах приложения. Начиная с версии Android 6.0 некоторые разрешения необходимо спрашивать напрямую у пользователя (Листинг 12).

Листинг 12 – Запрос разрешения на доступ к внутреннему хранилищу приложения

```
1. private void askPermissions() {  
2.     if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {  
3.         requestPermissions(new String[] {Manifest.permission.READ_EXTERNAL_STORAGE},  
4.         REQUEST_CODE_ASK_PERMISSIONS);  
5.         requestPermissions(new String[] {Manifest.permission.WRITE_EXTERNAL_STORAGE},  
6.         REQUEST_CODE_ASK_PERMISSIONS);  
7.     }  
8. }
```

Такой доступ спрашивается при первом открытии окна с просмотром STL-файла. Если доступ разрешён, то файл сохраняется в кэш приложения, и его путь передается напрямую в WebView (Листинг 13). Если доступ отклонили, то вернуть его всегда можно в настройках приложения.

Листинг 13 – Просмотр STL-файлов при помощи WebView

```
1. private void showWebView(String stl) {  
2.     mWebView.loadUrl("file:///android_asset/StlViewer.html");  
3.     mWebView.getSettings().setJavaScriptEnabled(true);  
4.     mWebView.getSettings().setSaveFormData(true);  
5.     mWebView.getSettings().setAllowFileAccessFromFileURLs(true);  
6.     mWebView.getSettings().setBuiltInZoomControls(true);  
7.  
8.     mWebView.setWebViewClient(new WebViewClient() {  
9.         @Override  
10.        public void onPageFinished(WebView view, String url) {  
11.            hideProgressBar();  
12.            Log.d(TAG, "onPageFinished: " + stl);  
13.  
14.            mWebView.loadUrl("javascript:loadStl(\"" + stl + "\");");  
15.        }  
16.    }
```

В строчках 2-6 выше представленного листинга выставляются настройки для работы JavaScript-компонентов. Момент доступности компонента логируется и при помощи метода *loadUrl()* (листинг 13 строка 14) напрямую вызывается метод из HTML *loadStl()*, в который передаётся расположения файла на устройстве. Далее происходит отображение полученной STL-модели (Рисунок 13).

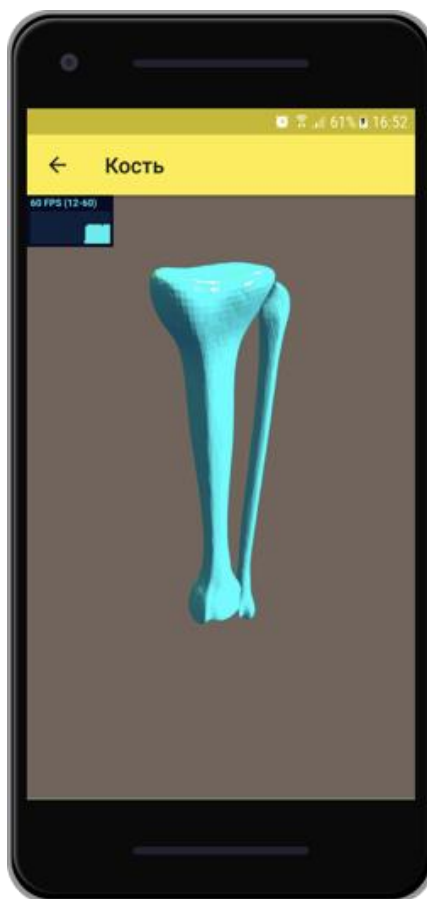


Рисунок 13 – Отображение STL-модели кости

При помощи функции перелистываний по экрану полученный объект можно как угодно поворачивать, приближать и уменьшать. Чтобы кэш приложения не рос при каждом отображении нового файла, его нужно очищать. Для этого используем метод жизненного цикла экрана, который вызывается перед закрытием экрана. Таким образом, был создан компонент для просмотра любых STL-моделей внутри Android-приложения при помощи компонента WebView и библиотеки ThreeJS.

4 Тестирование

4.1 Тестирование бизнес-процессов серверной части

Не менее важным этапом в разработке любого проекта является его тестирование. Оно помогает выявить слабые места приложения, определить граничные значения, и проверить наиболее частые сценарии пользования. Как правило, тесты разделяются на три главные части, которые могут быть выражены концепцией под названием «пирамида тестов» (Рисунок 14).

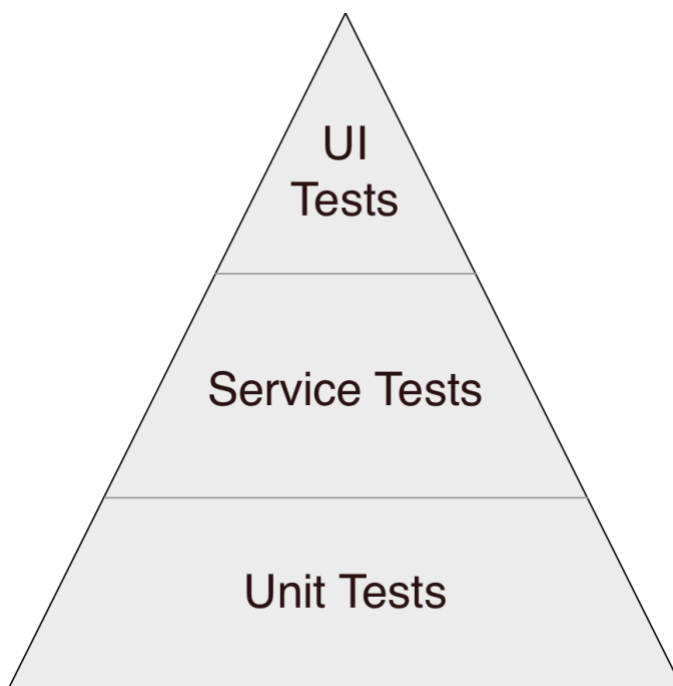


Рисунок 14 – «Пирамида тестов»

Она говорит о том, что чем ниже слой расположен в пирамиде, тем больше различных ситуаций он может покрыть и тем легче его разрабатывать. Начнём снизу-вверх. Первых из них называется *Unit* или модульными тестами. Они используются для проверки на корректность данных посредством обычных сравнений. Такие тесты легко писать, они быстро исполняются и потребляют мало ресурсов. Если все *Unit*-тесты прошли успешно, то изменения в коде не повлияли на написанную ранее часть. Провалившийся же тест сразу укажет на место, где возникла ошибка.

- a) На первых этапах разработки **модульное тестирование** обязательно для крупных проектов, оно поможет сэкономить огромное количество времени и обеспечить корректное поведение больших участков кода.
- b) Вторым слоем является **интеграционное** или **сервисное тестирование**. Оно используется, для тестирования взаимодействия разрабатываемого проекта с другими сервисами, которыми могут быть база данных, файловое хранилище и другие. Это означает, что перед запуском своего проекта, надо обеспечить работу и постороннего компонента. Интеграционные тесты находятся на более высоком уровне, чем модульные. Такие интеграции сложнее отслеживать, их выполнение занимает больше времени, но зато они обеспечивает уверенность в корректной работе с другими сервисами.
- c) На верхушке пирамиды находятся **тесты для пользовательского интерфейса**. Они позволяют выполнить имитацию пользования сайтом или приложение с помощью фантомных нажатий на экран. Как правило, для этого используются сторонние библиотеки. Далее необходимо описать сценарии взаимодействия с приложением, которые подразумевают изменения элементов интерфейса. В нашем случае такой вид тестов не понадобится, так как серверная часть проекта не предусматривает использование графического интерфейса.

Так как в предыдущей главе подробно описывалась разработка компонентов для сущности *User*, то далее необходимо провести тестирование разработанных классов. Для этого надо определить, какие именно части архитектуры приложения необходимо протестировать. Так как сервис обращается к репозиторию, а он в свою очередь к базе данных, то интеграционное тестирование слоя сервисов позволит определить корректность записи и взятия данных из БД.

Для этого будут написаны как успешные Unit-тесты для каждого метода сервиса, так и те, которые должны вызывать ошибки при обращении к базе данных. Начнём с метода регистрации пользователя (Листинг 14).

Листинг 14 – Тестирование метода сервиса для регистрации пользователя

```

1.  @Test
2.  public void registration() throws CustomException {
3.      User user = UserFamilyCreator.createUser("registration");
4.
5.      User userResult = userService.registration(user);
6.
7.      assertNotNull(userResult.getId());
8.      assertEquals(user.getUsername(), userResult.getUsername());
9.  }
10.
11. @Test
12. public void registrationLoginExists() throws CustomException {
13.     User user = UserFamilyCreator.createUser("registration");
14.
15.     User registration = userService.registration(user);
16.
17.     assertNotNull(registration.getId());
18.     assertEquals(user.getUsername(), registration.getUsername());
19.
20.     User user1 = UserFamilyCreator.createUser(user.getUsername());
21.
22.     try {
23.         User registration1 = userService.registration(user);
24.     } catch (CustomException e){
25.         assertEquals(e.getErrorCode(), ErrorCode.LOGIN_EXISTS);
26.     }
27. }

```

Так как это интеграционное тестирование, то во время тестов отдельно поднимается база данных PostgreSQL для тестовых данных. Все сущности в ней идентичны основной БД, так как все миграции проходят с помощью библиотеки LiquiBase. Далее каждый метод помечается аннотацией *@Test*, внутри которого статические методы *assert* выполняют проверки новых данных. Так как все тесты подразумевают использование однотипных данных, то был создан класс *UserFamilyCreator*, которые создает экземпляр класса *User* с заданным логином. Это позволит избежать повторяющихся участков кода. Для того, чтобы протестировать уникальность логина нового пользователя, второй тест создаёт двух новых пользователей и в *try*-блоке проверяет возникновение ошибки с соответствующим кодом во время сохранения сущности в БД. (Листинг 14 строки 20-26).

Для обработки исключительных ситуаций во время тестов также можно использовать в теле аннотации *@Test* название класса ошибки. Продемонстрируем это на методе изменения пользователя (Листинг 15). Для этого создадим две новых сущности, и во второй поменяем логин, который

будет совпадать с логином первой сущности. В процессе изменения база данных должна выбросить исключение *DataIntegrityViolationException*, которая оповещает об ошибке при попытке вставить новые данные или изменить уже существующие.

Листинг 15 – Тестирование метода сервиса для изменения пользователя

```
1.  @Test(expected = DataIntegrityViolationException.class)
2.  public void updateFail() throws CustomException {
3.      User user1 = UserFamilyCreator.createUser("updateFail1");
4.      User userResult1 = userService.save(user1);
5.
6.      assertNotNull(userResult1.getId());
7.      assertEquals(user1.getUsername(), userResult1.getUsername());
8.
9.      User user2 = UserFamilyCreator.createUser("updateFail2");
10.     User userResult2 = userService.save(user2);
11.
12.     assertNotNull(userResult2.getId());
13.     assertEquals(user2.getUsername(), userResult2.getUsername());
14.
15.     userResult2.setUsername(userResult1.getUsername());
16.     userService.update(userResult2);
17. }
```

Реализация большинства *Unit*-тестов однотипна и похожа между собой. Поэтому остальные методы в записке приводиться не будут. Перейдём к тестированию слоя контроллеров. Так как он обращается к слою сервисов, а тестировать его повторно второй раз смысла нет, то необходимо поставить так называемые «заглушки» на вызываемые методы. Для этого используется библиотека Mockito [19]. С помощью неё можно напрямую вернуть результат без вызова тела метода. Увидеть пример такого тестирования можно на методе поиска пользователя по уникальному идентификатору (Листинг 16).

Листинг 16 – Тестирование метода контроллера для поиска пользователя по идентификатору

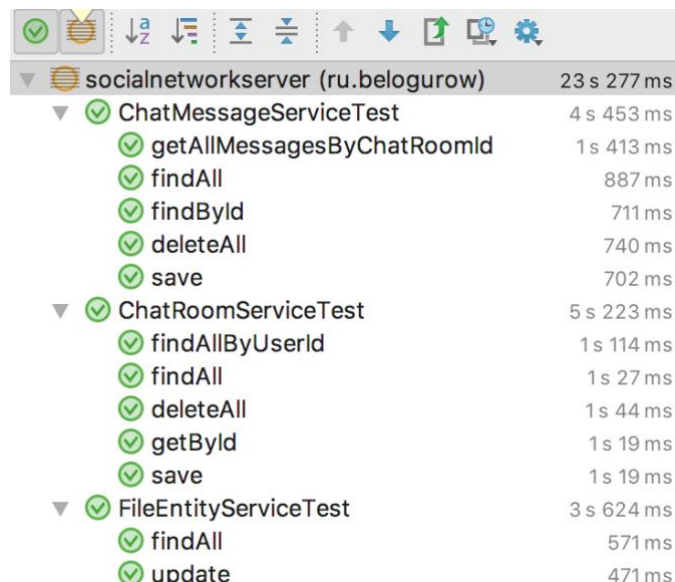
```
1.  @Test
2.  public void findUserByIdOk() throws Exception {
3.      User user = UserFamilyCreator.createUserWithId("getUser");
4.
5.      when(userDatabaseService.findById(user.getId()))
6.          .thenReturn(Optional.of(user));
7.
8.      mockMvc.perform(get("/users/{id}", user.getId()))
9.          .andExpect(MockMvcResultHandlers.print())
10.         .andExpect(status().isOk())
11.         .andExpect(content().contentType(
```

```

12.         MediaType.APPLICATION_JSON_UTF8))
13.         .andExpect(jsonPath("$.id", is(user.getId().toString())))
14.         .andExpect(jsonPath("$.username", is("getUser")))
15.         .andExpect(jsonPath("$.name", is("getUser")))
16.         .andExpect(jsonPath("$.userRole",
17.             is(UserRole.USER.toString())));
18.
19.     verify(userDatabaseService, times(1))
20.         .findById(user.getId());
21.     verifyNoMoreInteractions(userDatabaseService);
22. }

```

На строке 5 ставится заглушка, которая говорит, что при вызове метода сервиса необходимо вернуть экземпляр класса *User* в обёртке *Optional*. Далее посылается запрос и проверяется корректность полученного результата в формате JSON. На последних строках говорится, что метод сервиса для поиска пользователя вызывался один раз и других взаимодействием с этим классом не было. С помощью библиотеки Mockito можно выстраивать сложные связи и проводить независимое тестирование слоёв приложения между собой. Таким образом, прохождение всех написанных тестов проекта отображено на рисунке 15.



Test Class	Duration
socialnetworkserver (ru.belogurow)	23 s 277 ms
ChatMessageServiceTest	4 s 453 ms
getAllMessagesByChatRoomId	1 s 413 ms
findAll	887 ms
findById	711 ms
deleteAll	740 ms
save	702 ms
ChatRoomServiceTest	5 s 223 ms
findAllByUserId	1 s 114 ms
findAll	1 s 27 ms
deleteAll	1 s 44 ms
getById	1 s 19 ms
save	1 s 19 ms
FileEntityServiceTest	3 s 624 ms
findAll	571 ms
update	471 ms

Рисунок 15 – Результат прохождения всех тестов

Зелёная галочка обозначает успешное прохождение тестов. Подводя итог, можно сказать, что этап написания и проведения тестов не менее важен, чем этап разработки всего приложения. Зачастую для этого выделяют большие

команды и ресурсы. Как правило, нужно стремиться к 100% покрытию всего кода тестами. Это поможет избавить приложение от слабых мест и сделать его более стабильным, что поможет ему эффективнее выполнять свою работу.

4.2 Тестирование Android-приложения

Клиентская часть проекта также нуждается в тестировании. Как правило, это происходит в течении всего процесса разработки с помощью реального устройства или эмулятора, на котором запускается тестовая версия приложения. Здесь также применима «пирамида тестов» (Рисунок 14), но в силу нехватки ресурсов и времени остановимся на ручном тестировании всего приложения.

Разработка клиентской части и тестирование велось на трёх Android-устройствах. Их спецификации подробно представлены в таблице 8. Стоит отметить, что из-за разных прошивок приложение в некоторых местах может работать нестабильно, поэтому чем больше устройств будет использовано при тестировании, тем меньше проблем возникнет в дальнейшем. Это является одной из проблем разработки Android-приложений, так как каждый год выходит огромное количество телефонов и планшетов от разных производителей, следовательно, приходится активно адаптироваться под них.

Таблица 8 – Список используемых устройств во время разработки и тестирования

Модель	Версия ОС	Оперативная память	Потребляемая оп. память приложения
Samsung Galaxy S7	Android 8.0	4 GB	150 MB
Samsung Galaxy S4	Android 6.0	2 GB	120 MB
Xiaomi Redmi 4 Pro	Android 7.1	3 GB	130 MB

Визуально очень трудно определить, насколько хорошо приложение работает на устройство. Для этого в Android Studio 3.0 был добавлен Android Profiler [20]. С помощью него можно в режиме реального времени отслеживать

показатели CPU, оперативной памяти и сетевую активность. Android Profiler может показывать трасировку выполняемого кода, распределение памяти внутри приложения и следить за отправляемыми файлами через интернет. На рисунке 16 представлен Android Profiler устройства Samsung Galaxy S7 во время обычных переходов между экранами приложения.

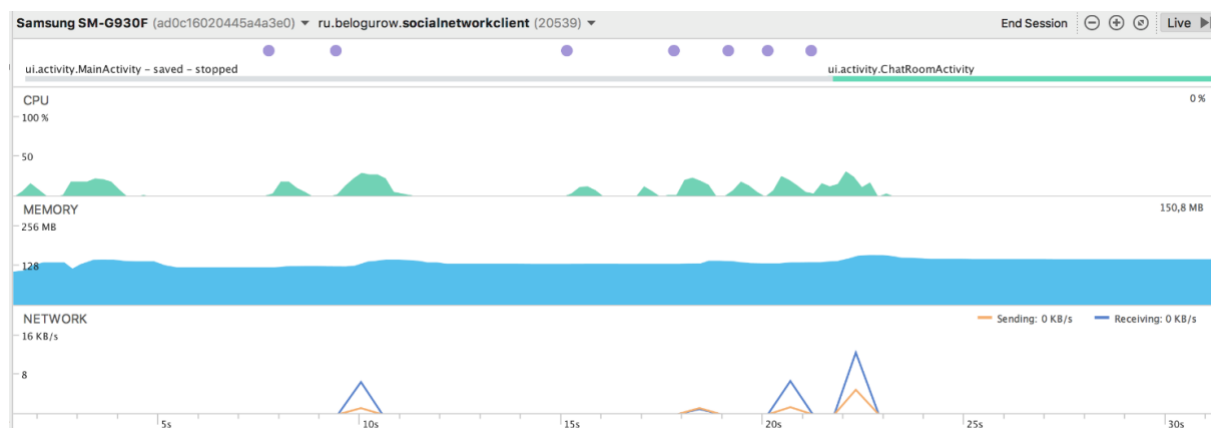


Рисунок 16 – Отображение работы приложения при помощи Android Profiler

В слое *CPU* зелёным цветом обозначаются места, когда приложению понадобилось затронуть ресурсы центрального процессора устройства. Это может быть выделение нового потока и другие похожие действия. В слое *Memory* показывается кривая распределения потребляемой оперативной памяти. Ровная линия и отсутствие больших ям может говорить о том, что приложение работает стабильно и без сбоев. На рисунке 16 неровности появляются только во время открытия новых экранов или загрузки больших файлов и изображений, это может происходить при сложной логике взаимодействия всех компонентов. Последний слой *Network* отображает время и размер отправляемых и принимаемых HTTP-запросов. Жёлтой линией показывается количество отправленных байт, а синий – количество принятых от сервера. По нажатию на каждый слой откроется новое окно с более подробной информацией о выбранном компоненте. Таким образом, Android Profiler представляет мощный инструмент для отслеживания внутренних процессов приложения. С помощью него можно быстро находить утечки памяти, отслеживать их появление и быстро устранять их.

Для более подробной отладки приложения использовалась сторонняя библиотека Stetho [21] от компании Facebook. Она позволяет при помощи инструмента Chrome DevTools браузера Google Chrome управлять внутренней базой SQLite, просматривать иерархию всех элементов и следить за всеми HTTP-запросами, включая просмотр их тел ответа на языке JSON (Рисунок 17). Stetho представляет удобный интерфейс для отслеживания всех этих процессов и подробную информацию о каждом из них. Для работы с ней необходимо добавить зависимость в приложение, подключить мобильное устройство по проводу к компьютеру, на котором ведётся разработка, и открыть ChromeDevTools.

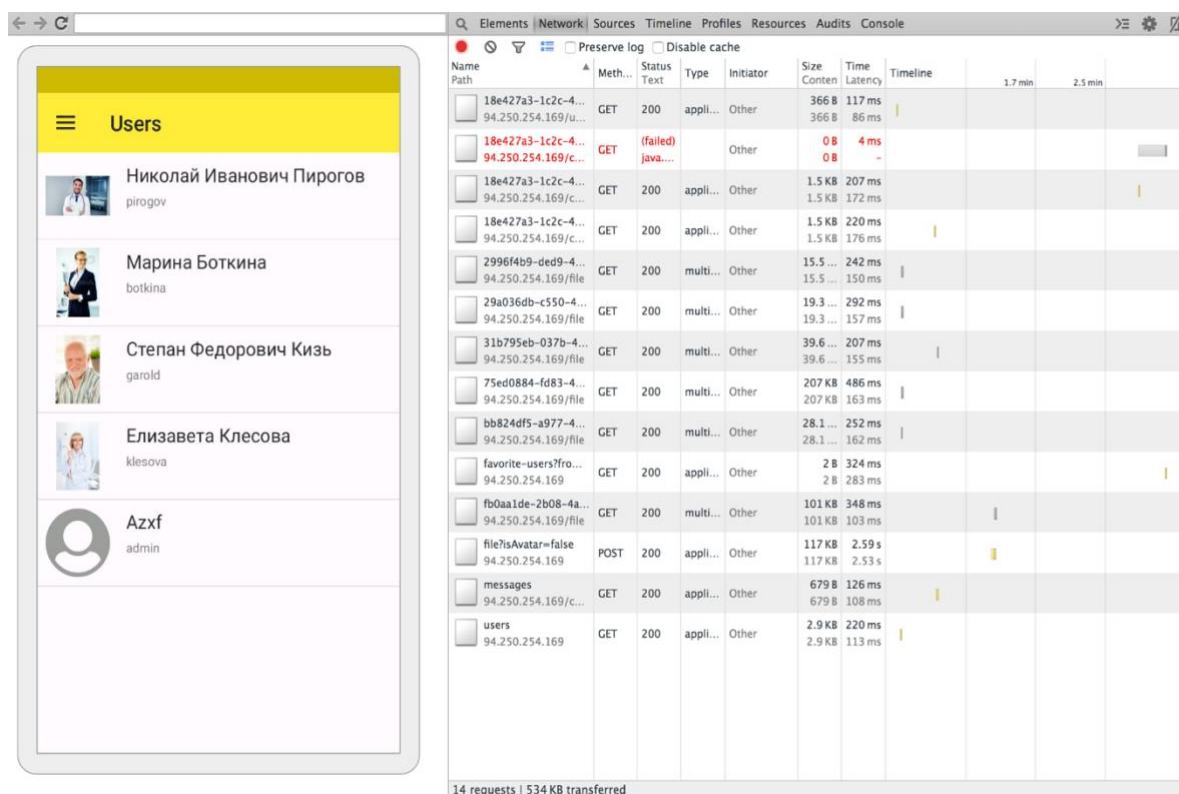


Рисунок 17 – Отображение списка отправленных HTTP-запросов при помощи библиотеки Stetho

С помощью предоставленных инструментов можно эффективно протестировать и отследить работу приложения на конкретных устройствах.

ЗАКЛЮЧЕНИЕ

В ходе написания дипломного проекта был создан мобильный программный комплекс «Консилиум». Была разработана серверная часть продукта на базе Spring Boot 2.0, а также его непосредственная клиентская часть в виде Android-приложения. Для хранилища данных использовалась СУБД PostgreSQL.

В рамках работы над проектом:

- проведён обзор программ-аналогов, который помог выявить преимущества и недостатки таких сервисов;
- сформулированы возможности Android-приложения на основе ранее проведённого анализа;
- разработана архитектура клиент-серверного приложения;
- спроектирован интерфейс мобильного приложения;
- обоснован выбор защищённого протокола HTTPS для передачи данных между клиентом и сервером;
- реализована авторизация и аутентификация пользователей, просмотр их профиля и добавления пользователей в избранный список;
- реализована загрузка файлов с устройства и просмотр форматов STL, PDF, JPG;
- спроектирован чат между пользователями, который использует технологию WebSocket, и позволяет обмениваться сообщениями, состоящих из текста или одного файла;
- проведена оптимизация внутренних процессов Android-приложения при помощи сторонних библиотек;
- проведено ручное тестирование и отладка клиентской части проекта и тестирование отдельных слоёв серверной части.

В качестве перспектив разработки приложения можно рассматривать расширение функциональности и перенос социальной сети на другие мобильные платформы и создание полноценного сайта.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Медтусовка [Электронный ресурс]. - <http://medtusovka.ru>
2. Alma Mater [Электронный ресурс]. - <http://www.alma-mater.su>
3. Медицинская социальная сеть [Электронный ресурс]. - <https://medring.ru/>
4. Врачи РФ [Электронный ресурс]. - <https://vrachirf.ru/>
5. Доктор на работе [Электронный ресурс]. - <https://www.doktornarabote.ru/>
6. Врачи вместе – профессиональная социальная сеть для врачей.
[Электронный ресурс]. - <https://vrachivmeste.ru/>
7. Spring Boot [Электронный ресурс]. - <https://spring.io/projects/spring-boot>
8. PostgreSQL: The world's most advanced open source relational database
[Электронный ресурс]. - <https://www.postgresql.org/>
9. Android Architecture Components | Android Developers [Электронный
ресурс]. - <https://developer.android.com/topic/libraries/architecture/>
10. Retrofit [Электронный ресурс]. - <http://square.github.io/retrofit/>
11. Liquibase | Database Refactoring [Электронный ресурс]. -
<https://www.liquibase.org/>
12. RxJava – Reactive Extensions for the JVM [Электронный ресурс].
<https://github.com/ReactiveX/RxJava>
13. MaterialDrawer [Электронный ресурс]. -
<https://mikepenz.github.io/MaterialDrawer/>
14. RxJava Anatomy: What is RxJava, how RxJava is designed, and how RxJava
works [Электронный ресурс]. - [https://medium.com/mindorks/rxjava-anatomy-
what-is-rxjava-how-rxjava-is-designed-and-how-rxjava-works-d357b3aca586](https://medium.com/mindorks/rxjava-anatomy-what-is-rxjava-how-rxjava-is-designed-and-how-rxjava-works-d357b3aca586)
15. Thumbnailator - a thumbnail generation library for Java [Электронный
ресурс]. - <https://github.com/coobird/thumbnailator>
16. Кости пальцев кисти [Электронный ресурс]. -
http://www.medclub.ru/anatomy/finger_bones.html
17. Three.js – JavaScript 3D library [Электронный ресурс]. - <https://threejs.org>

18. WebView | Android Developers [Электронный ресурс]. - <https://developer.android.com/reference/android/webkit/WebView>
19. Mockito framework site [Электронный ресурс]. - <http://site.mockito.org>
20. Measure App performance with Android Profiler | Android Developers [Электронный ресурс]. - <https://developer.android.com/studio/profile/android-profiler>
21. Stetho – a debug bridge for Android applications [Электронный ресурс]. - <http://facebook.github.io/stetho/>