СОДЕРЖАНИЕ

[1 Теоретическое введение 2](#_Toc120150495)

[2 Выполнение поставленных задач в практической работe №4 3](#_Toc120150496)

[2.1 Ролевая модель разрабатываемого приложения 3](#_Toc120150497)

[2.1.1 Планировка структуры системы и построение UserFlow приложения 3](#_Toc120150498)

[2.1.2 Определение разновидностей и задач каждой группы пользователей в ролевой модели приложения 4](#_Toc120150499)

[2.1.3 Изменение внешнего представления и логики приложения в зависимости от выбранной ролевой модели 4](#_Toc120150500)

[2.2 Реализация авторизации посредством общения с удалённым открытым OAuth 2.0 провайдером 11](#_Toc120150501)

[2.2.1 Выбор OAuth 2.0 провайдера для реализации удалённой авторизации 11](#_Toc120150502)

[2.2.2 Реализация открытия формы или приложения OAuth 2.0 провайдера 16](#_Toc120150503)

[2.2.3 Получение access\_token или access\_key от OAuth 2.0 провайдера 17](#_Toc120150504)

[2.2.4 Реализация логики работы приложения после проведения удалённой авторизации через протокол OAuth 2.0 17](#_Toc120150505)

[3 Демонстрация работы приложения 21](#_Toc120150506)

[4 Приложение 23](#_Toc120150507)

# Теоретическое введение

В наше время очень распространены различные маркетплейсы, в них продают множество товаров различного рода происхождения и порой покупать тот или иной продукт довольно страшно, поэтому мы заходим читать отзывы различных людей, которые часто состоят из пары слов и не могут дать представление о предмете вожделения, происходит из-за шаблонного представления отзывов, состоящих из 3 пунктов: достоинства, недостатки и комментарии, при таких инструментах сложно сделать идейный отзыв о товаре после которого тот или иной читатель сможет понять стоит ли тот или иной товар покупки.

Программа «Отзовик» позволяет проводить авторизаию, через популярную социальную сеть VK, а также предоставляет возможность создавать текст с возможностью добавлять фотографии под каждый абзац для большего погружения читателя, а также просмотр отзывов других пользователей, и поиск предмета обозревания.

# Выполнение поставленных задач в практической работe №4

## Ролевая модель разрабатываемого приложения

### Планировка структуры системы и построение UserFlow приложения

В ходе планирования UserFlow была сконструирована диаграмма UserFlow показанная на рисунке 1.

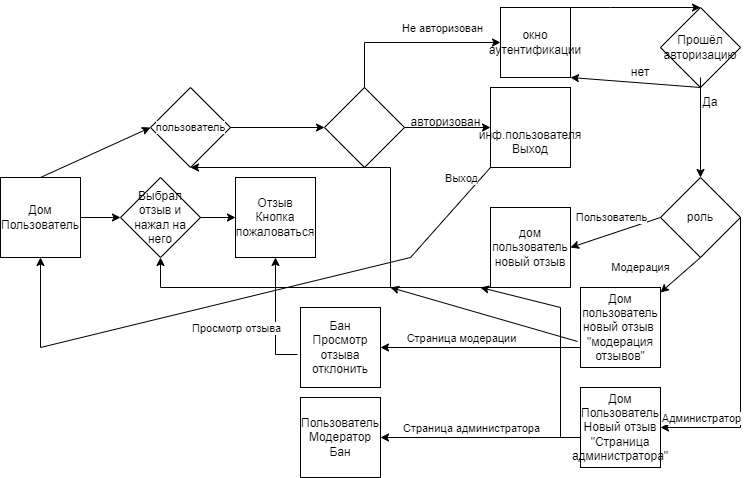


Рисунок 1 – UserFlow разрабатываемой программы

Пользователь начинает работу приложения с состояния неавторизованного пользователя, которому из доступного функционала доступна лишь страница со списком отзывов на различные товары, показываемая на диаграмме как кнопка «Дом», и переход на страницу пользователя, где его перенаправляют на страницу авторизации, при нажатии на кнопку, условно обозначенную на диаграмме «Пользователь». Если пользователь не смог авторизоваться, то он остаётся на том же окне, в случае успешного прохождения идентификации пользователя, Если же авторизация происходит впервые то необходимые данные нового пользователя заносятся в таблицу пользователей локально базы данных. У пользователя меняется интерфейс в зависимости от прав доступа пользователя определённых его ролью и получает соответствующий доступ к «ролевым страницам» пользователя, но создавать отзывы можно, будучи любым авторизованным пользователем. В ролевых страницах привилегированный пользователь может производить из соответствующие манипуляции, как показано на диаграмме. При бане пользователя/отзыва соответствующее поле удаляется из таблицы. Пользователь после авторизации проходит на главный экран. Также любой пользователь может просматривать пользовательские отзывы. При переходе на страницу профиля будучи авторизованным пользователь перейдёт на соответствующую страницу со списком всех своих отзывов хранящихся в локальной базе данных пользователя. В будущих итерациях блокирующиеся поля не должны удалятся, а должны быть помечены, чтобы сохранять историю действий ответственных на действия привилегированных пользователей.

### Определение разновидностей и задач каждой группы пользователей в ролевой модели приложения

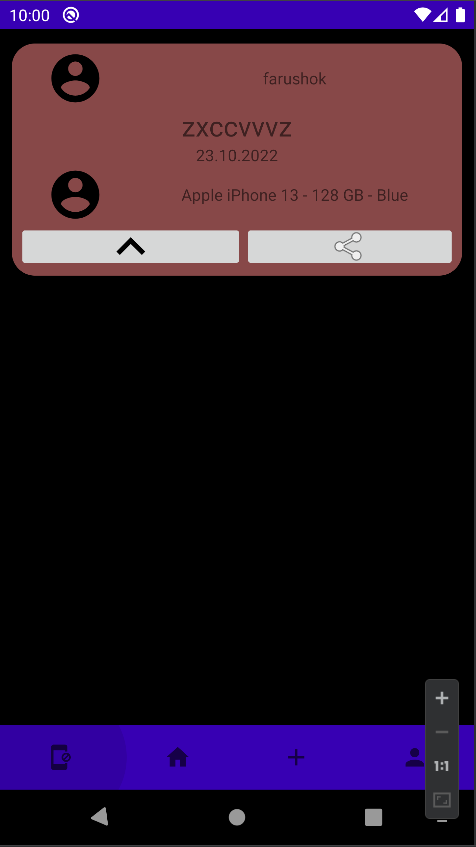
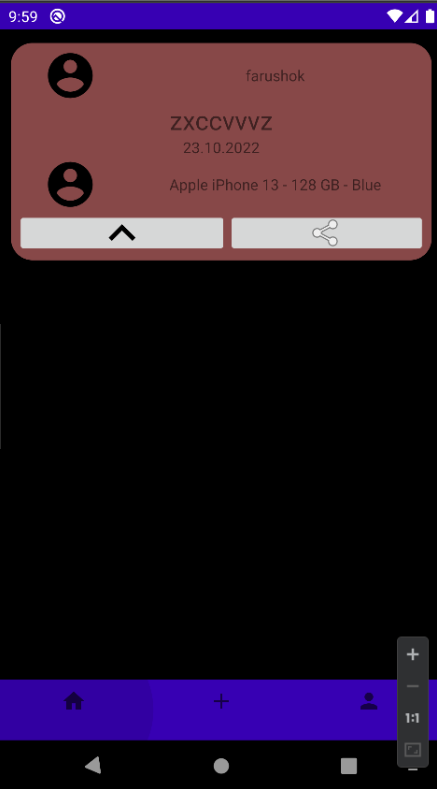
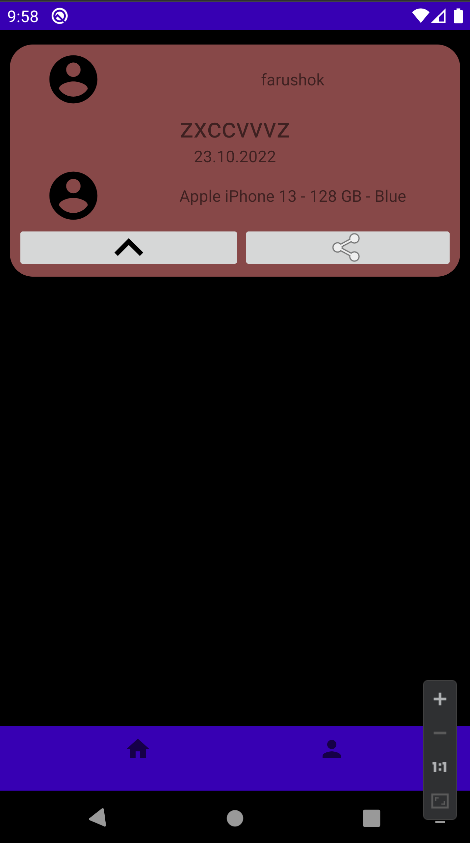
Для выполнения задуманного в предыдущем пункте UserFlow необходимо реализовать такие группы пользователь, как:

* user/unauthorized – обычный пользователь/неавторизованный, численность: 0 – ∞;
* moderator – модератор занимается модерацией отзывов, на которые были получены жалобы, с возможностью удалить его либо же отклонить полученные жалобы численность: 1 – 20;
* admin – администратор занимается переназначением ролей модератор или пользователь и баном пользователей, численность 1 – 5.

Каждый пользователь имеет свою роль, доступ к данным определяется по таблице в базе данных.

### Изменение внешнего представления и логики приложения в зависимости от выбранной ролевой модели

Для изменения пользовательского интерфейса в зависимости от роли были определены минимальные данные доступные каждому пользователю и соответствующие им элементы на разметке. Далее для каждой роли были добавлены скрытые элементы, которые прорисовываются в зависимости от роли, так, например, обычный пользователь и администратор не видят кнопки перехода на страницу модерации, а модератор не видит страницу администрирования, неавторизованный же не видит никакие свойственные элементы ранее названных ролей, как показано на рисунке 2.



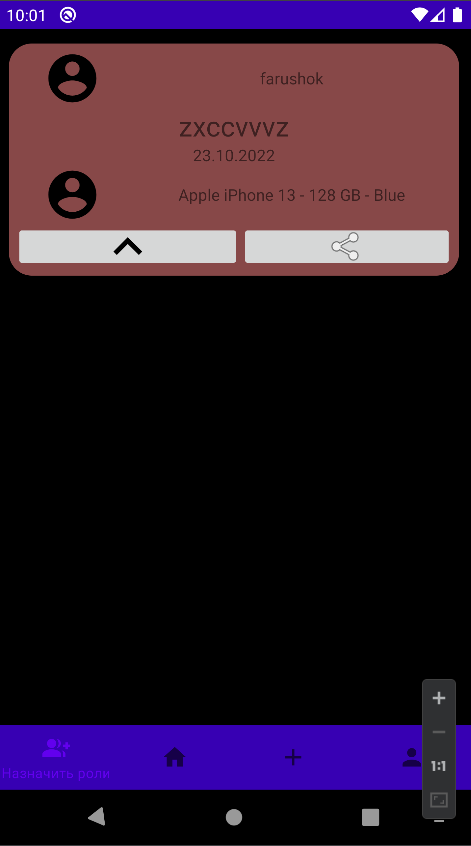


Рисунок 2 – изменение интерфейса в зависимости от роли

Для хранения информации о доступе была добавлена сущность «PermissionEntity», изображённой на рисунке 3, в локальную базу данных наследующаяся от базовой модели-класса Permission.

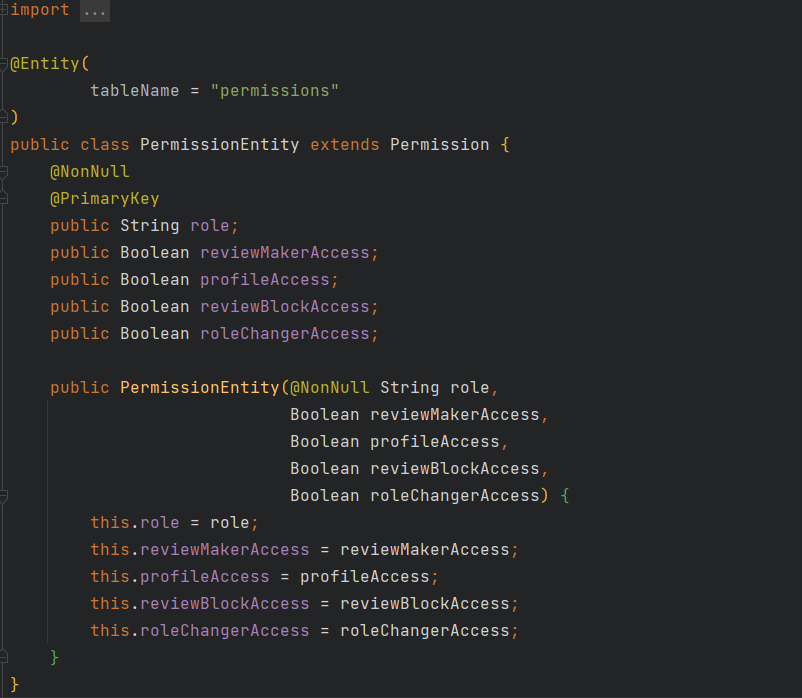


Рисунок 3 – класс-сущность «PermissionEntity»

Для более простого построения был создан Builder класс в родителе для более простого и понятного построения экземпляра сущности во время инициализации и записи базовых моделей как показано на рисунке 4.



Рисунок 4 – инициализация базовых ролей

И для связи между этими объектами был создан класс-посредник с помощью функционала Room, позволяющий при запросе поля одной сущности также и получить связанную с ним поле(-я). Класс изображён на рисунке 5.

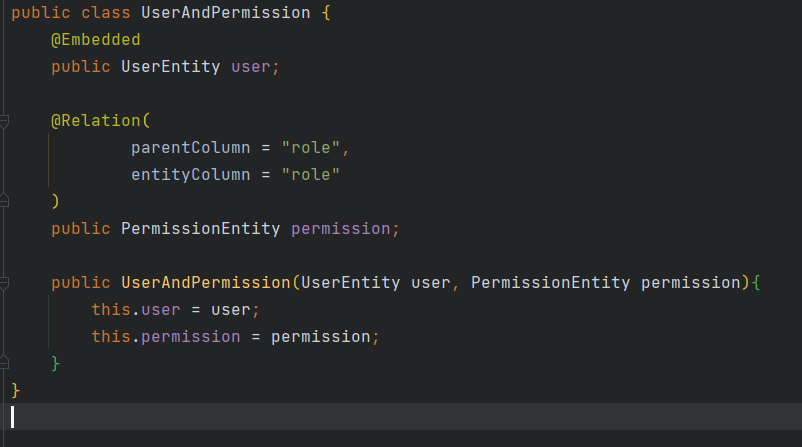


Рисунок 5 – класс-посредник

Под аннотацией «@Embedded» указывается через которую ищется другая сущность, находящейся под аннотацией «@Relation», в которой указывается общие столбцы сущностей. Для получения данных в подобном формате необходимо в DAO интерфейсе над указать @Transaction. Данный класс используется для изменения интерфейса без повторного запроса через ViewModel из View как показано на рисунках 6 и 7.

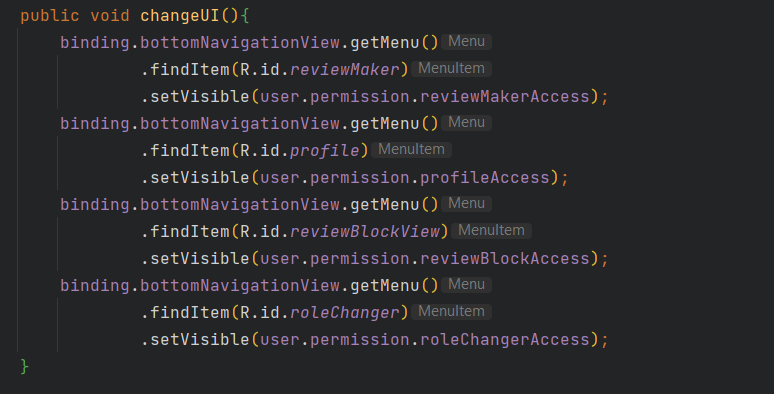


Рисунок 6 – изменение кнопок нижней навигационной панели

Производится поиск элемента нижней панели по Id и выставления параметра visible() в значение хранящееся в сущности PermissionEntity под каждой подобный элемент.

Так же для возможности модерации была создана сущность ReportEntity хранящая в качестве первичного ключа ForeignKey отзыва, на который производится жалоба и число жалоб. Модератор на своей странице, показанной на рисунке 7.



Рисунок 7 – страница модератора

Модератор может выбрать действия предварительно ознакомившись с отзывом при нажатии на область над кнопками. При нажатии на кнопку «Бан» выбранный отзыв будет удалён с БД также вместе с полем жалобы в таблице. Для администратора страница выглядит похожим образом, как показано на рисунке 8.

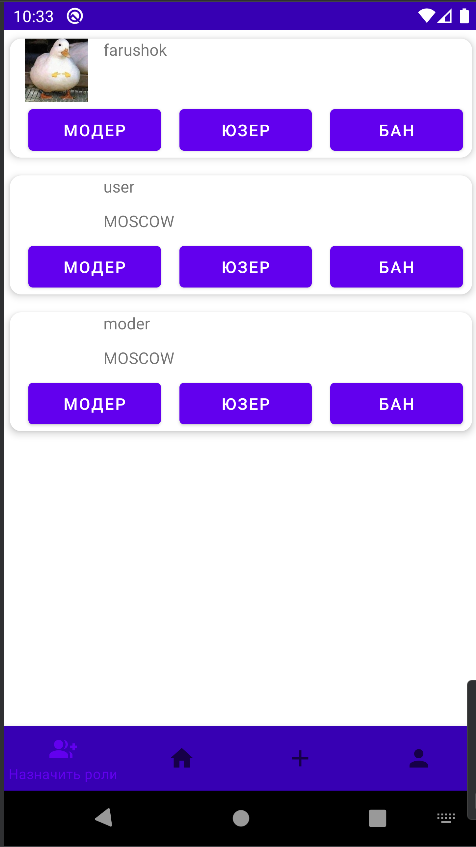


Рисунок 8 – страница администратора

Администратор при нажатии на кнопки может выполнять соответствующие действия над учётными записями пользователя, кнопка «Бан» работает аналогично модерации.

## Реализация авторизации посредством общения с удалённым открытым OAuth 2.0 провайдером

### Выбор OAuth 2.0 провайдера для реализации удалённой авторизации

Для реализации поставленных задач в качестве провайдера OAuth 2.0 была выбрана социальная сеть «ВКонтакте», по причине распространённости в странах СНГ и документации, представленной на русском языке.

Для начала работы с провайдером были исследованы разделы изображённые на рисунках 8-12.

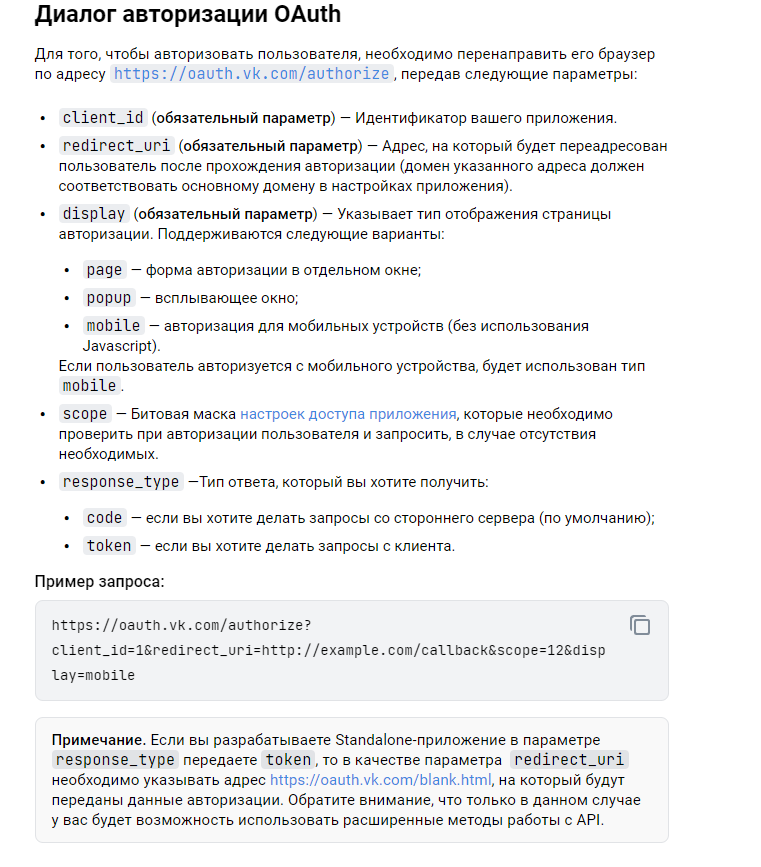


Рисунок 8 – документация вызова диалога авторизации

Client\_id был получен после создания приложения в соответствующем разделе в соц.сети, redirect\_uri был указан согласно требования провайдера blank.html.

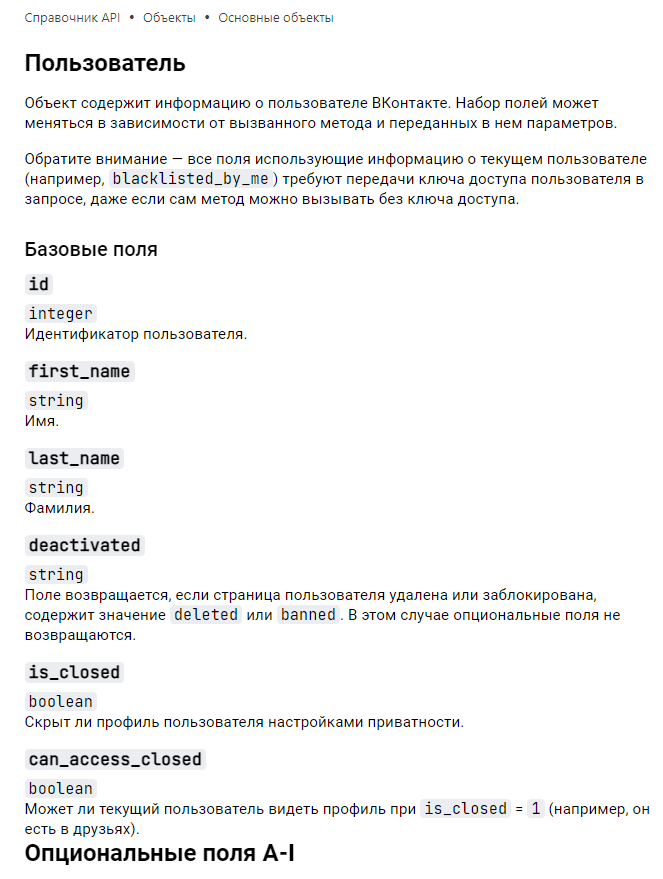


Рисунок 9 – модель пользователя

Из модели запрашивается фотография и короткое имя, которые будут использованы для авторизации нового пользователя в разрабатываемом приложении.

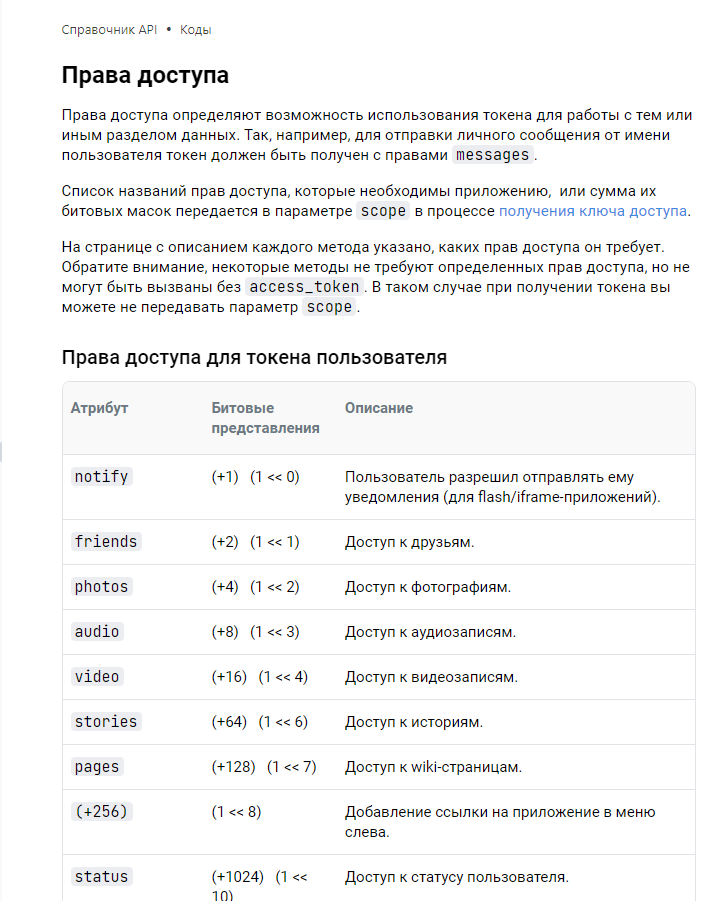


Рисунок 10 – документация прав доступа

Для формирования нового пользователя из перечисленных прав в разрабатываемом приложении используются photos и offline.

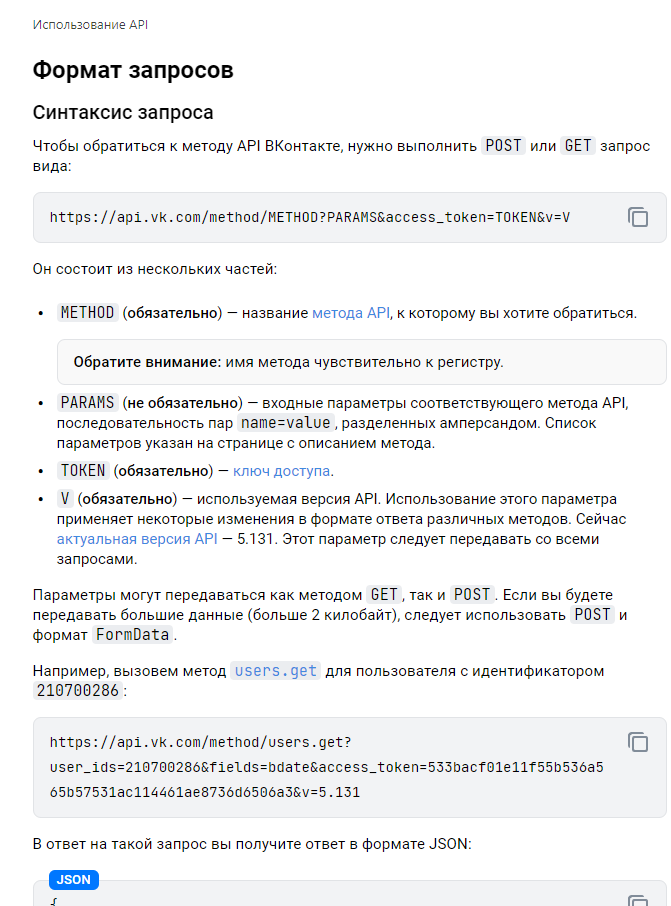


Рисунок 11 – документация формата запросов

Для понимая способа взаимодействия были изучены форматы запроса данных

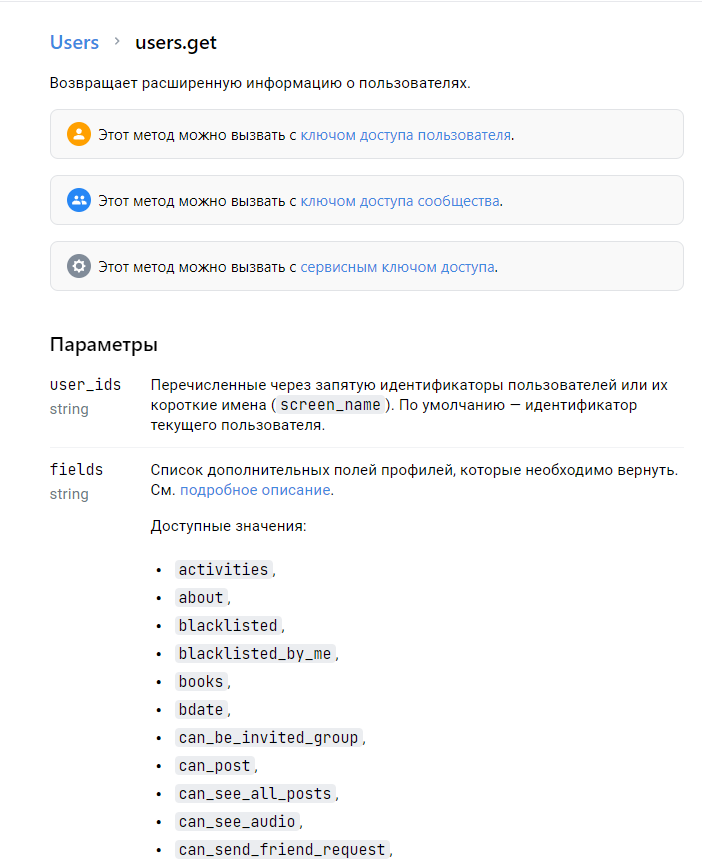


Рисунок 12 – метод users.get

Для успешного запроса и формирования POJO класса был использован метод users.get откуда берётся и скачивается изображение профиля пользователя и его короткое имя для записи в качестве имени пользователя в БД

### Реализация открытия формы или приложения OAuth 2.0 провайдера

Для реализации открытия формы был использован WebView для отображения страниц по указываемому запросу, куда передаётся переопределённый WebViewClient, отвечающий за обработку ссылок, как показано на рисунке 13.

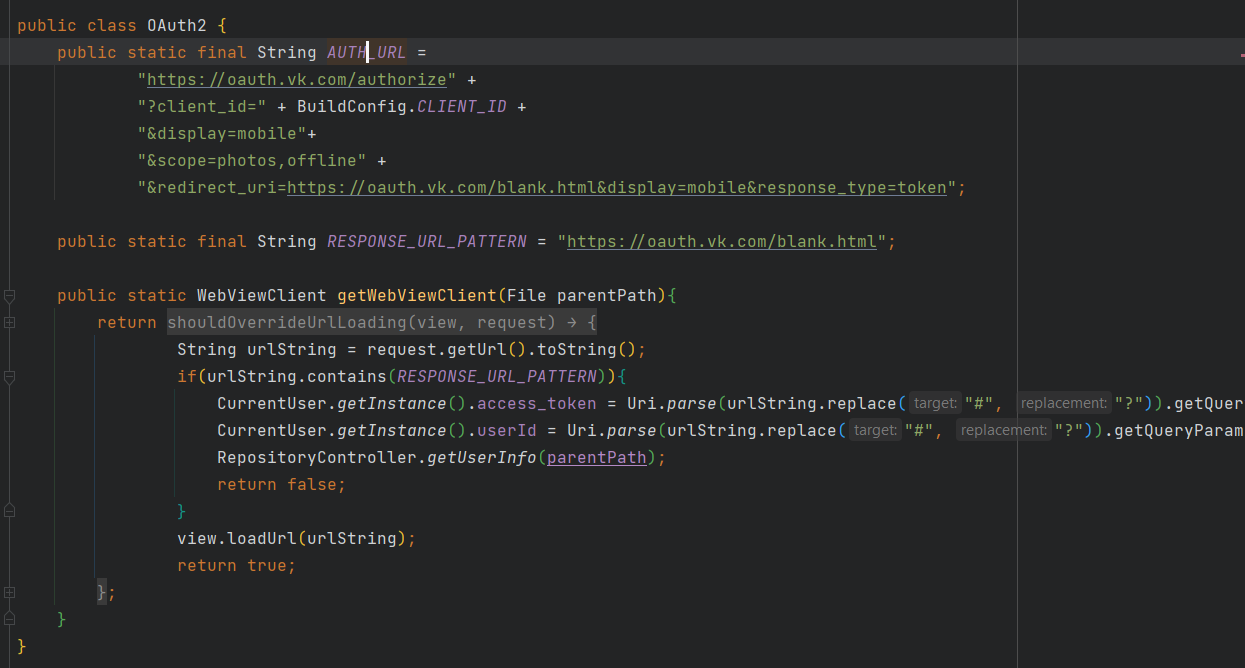


Рисунок 13 – обработка ссылки

Реализованный клиент передаётся в WebView созданного для этого OAuthFragment, как показано на рисунке 14, содержащий лишь только WebView. И загружающий в него первую ссылку на запрос авторизации куда помещается client\_id спрятанный в local.properties.

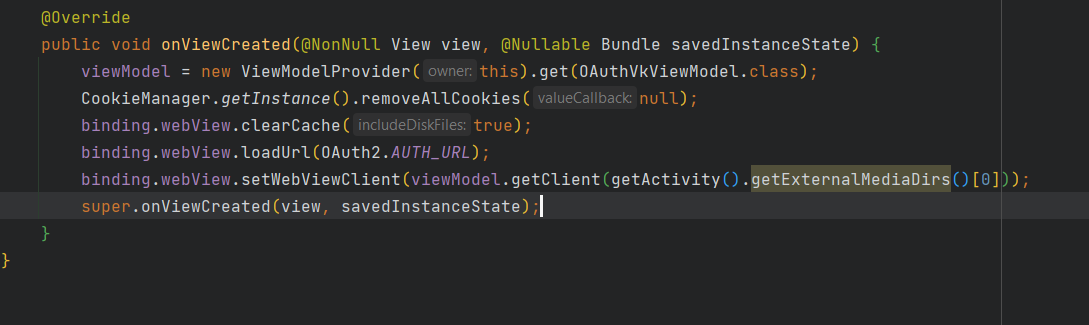


Рисунок 14 – заполнение WebView

Для возможности последующих авторизаций новых выставляется параметр clearCache в true.

### Получение access\_token или access\_key от OAuth 2.0 провайдера

В случае если произошла переадресация на указанную в документации страницу, при выполнении логики WebClient, полученные оттуда токен сохраняются поля класса CurrentUser хранящий информацию о активном пользователе, а также user\_Id используемый для реализации методов VK API.

Данные приходят формате «базовая строка»#переменные&… для того чтобы можно было легче вытянуть данные необходимо использовать Uri и вставить в метод parse полученную строку с заменённой решёткой на знак вопроса являющийся разделителем. И вызвать параметр, вытягивающий значения параметров.

### Реализация логики работы приложения после проведения удалённой авторизации через протокол OAuth 2.0

Для реализации последующей логики были созданы классы сервис и базовый класс с помощью Retrofit а также POJO класс для работы с полученными данными. На рисунке 15 изображён сервис для работы с API.



Рисунок 15 – сервис для реализации

Также в запросе необходимо указать версию API которая также помещена в local.properties. Для получения ответа в виде класса был написан VkResponse хранящий в себе информацию о городе имени и ссылке на фотографию, как показано на рисунке 16.



Рисунок 16 – POJO класс для сервиса

Чтобы вызвать реализованный сервис был создан базовый класс для работы с API, изображённый на рисунке 16

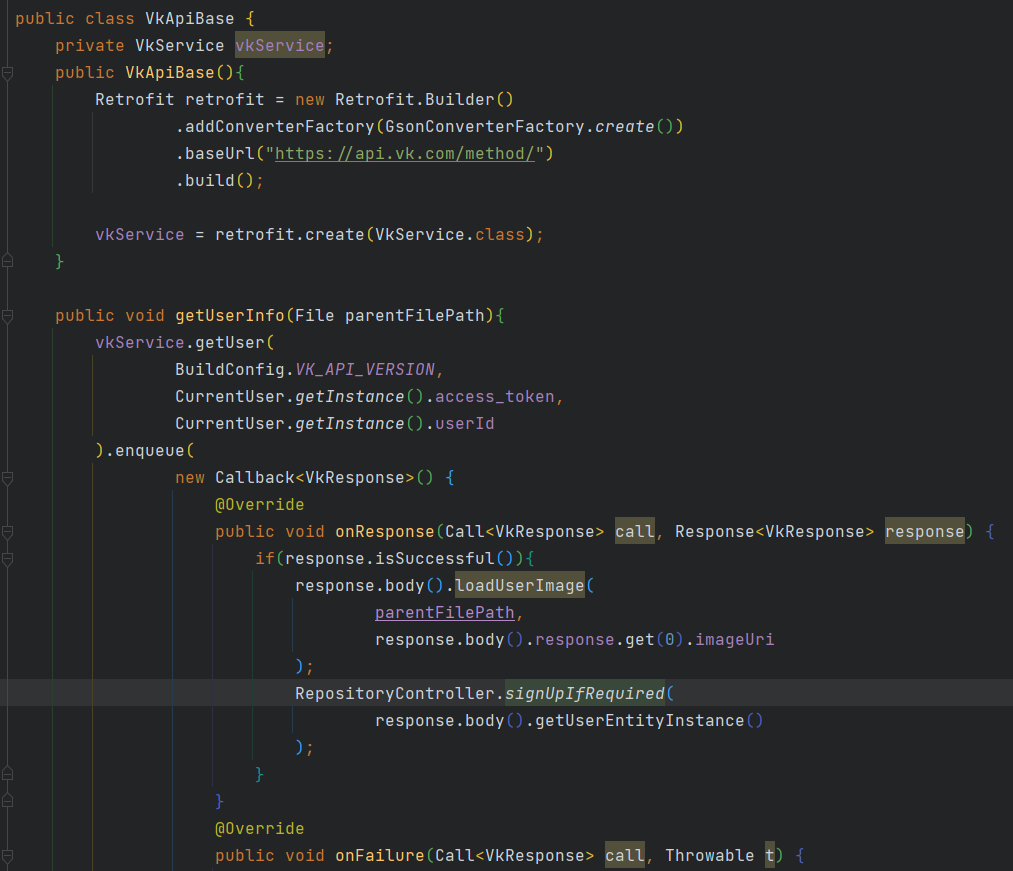


Рисунок 16 – базовый класс для работы с VK API

При вызове метода обработки ответа необходим экземпляр класса File куда сохранить изображение пользователя. Далее происходит авторизация либо регистрация нового пользователя вызовом метода signUpIfRequired(), с параметром UserEntity, как показано на рисунке 17.

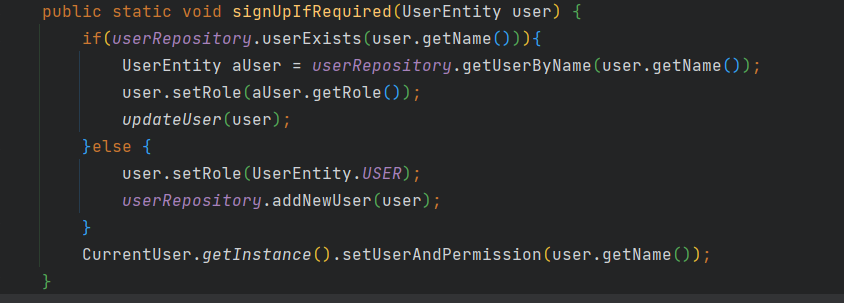


Рисунок 17 – авторизация пользователя

Если пользователь существует, то информация о нём обновляется, а при его отсутствии в БД заносится новый пользователь путь вызова синхронного метода записи нового пользователя в БД. Метод производится синхронно поскольку требуется прохождение полной авторизации, прежде чем, программа сможет работать с пользователем. Т.К. для всех последующих процессов обращение идёт не к провайдеру, а к базе данных. Далее занесённый пользователь изменит состояние класса CurrentUser чем вызовет изменения в интерфейсе, поскольку поле содержащее текущего пользователя является LiveData<> как показано на рисунке 18.

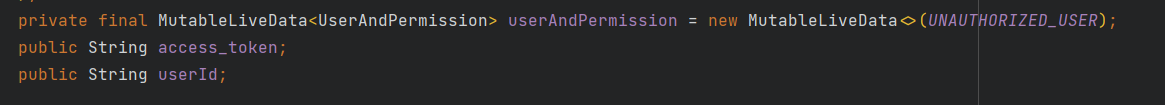


Рисунок 18 – поле текущего пользователя

# Демонстрация работы приложения

Для демонстрации были запущены пользователи с разных ролей (рисунок 19) и показана таблица доступа на рисунке 20.

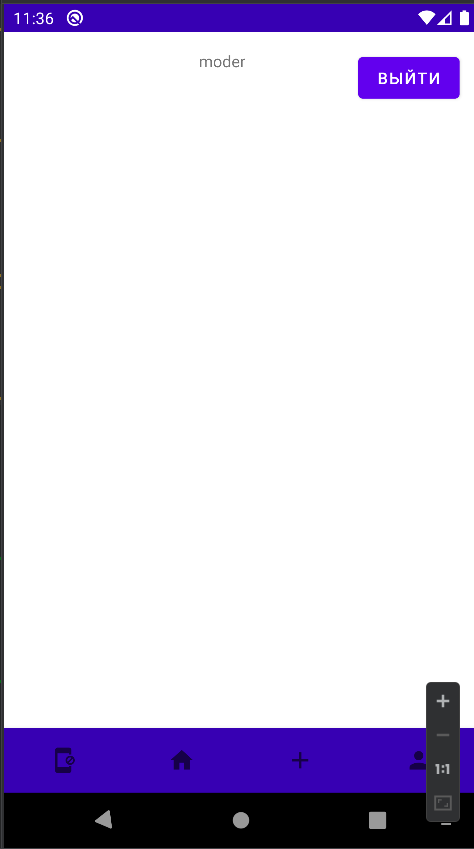
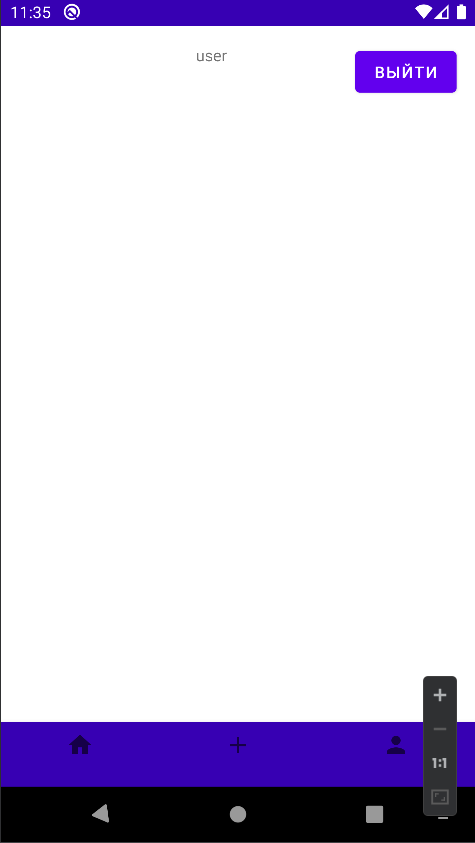
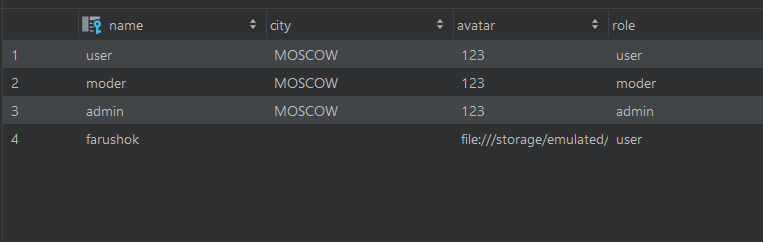


Рисунок 19 – интерфейс ролей



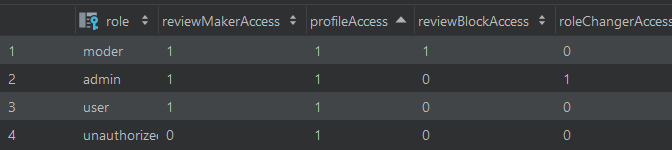


Рисунок 20 – демонстрация работы ролевой модели

Как видно из рисунков интерфейс пользователей соответствует их ролям.

Дале необходимо произвести авторизацию через провайдера как показано на рисунке 21.

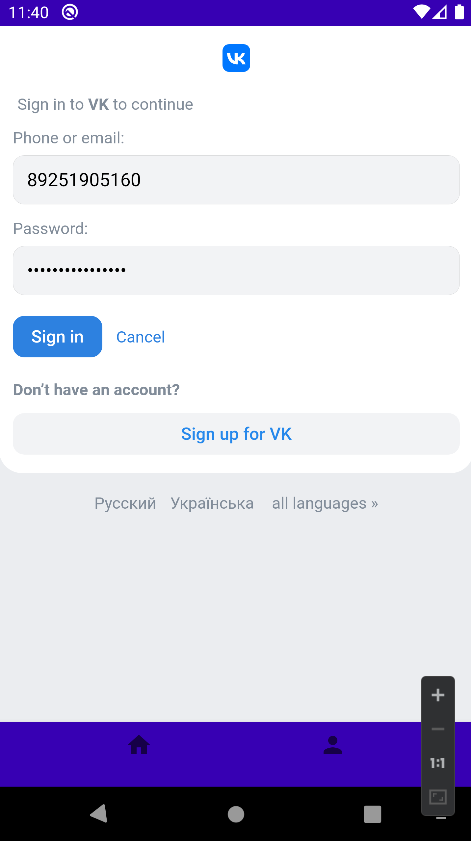


Рисунок 21 – демонстрация работы авторизации через провайдера

Исходя из результатов авторизации через провайдера программа выполняет поставленные задачи.

# Приложение

GitHub проекта: <https://github.com/Joombey/ReviewerJAVA/tree/master>