МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт кибернетики
Базовая кафедра №252 – информационной безопасности

КУРСОВАЯ РАБОТА

По дисциплине
«Методы программирования»

Тема курсовой работы
«Приложение-мессенжер»

Студент группы ККСО-03-19	Николенко В.О.	_____ *(подпись)*
Руководитель курсовой работы	Чуваев А.В.	_____ *(подпись)*

Работа представлена к защите «__» _____ 2021 г.

Допущен к защите «__» _____ 2021 г.

Москва – 2021

СОДЕРЖАНИЕ

1. ВВЕДЕНИЕ

В наше время обмен информацией стал носить глобальный характер, и в связи с этим у людей с каждым годом появляются всё более высокие требования к приложениям и программам, позволяющим им коммуницировать друг с другом.

Мы уже не можем представить себе скорость интернета в 5 кбит, как это было в конце 1990-х - начале 2000-х, не можем вообразить, как будем пользоваться приложением с кривым и страшным интерфейсом, приложением с большим количеством ошибок в коде, которые могут приводить к массовым сбоям, или вообще выходу из строя всей системы обмена сообщениями. Более того, много кто начал задумываться о безопасности того приложения, посредством которого тот или иной гражданин отправляет информацию на другой конец провода. Иначе говоря, никто бы не хотел, чтоб его личную переписку прочитал кто-либо посторонний, враждебно настроенный. Все эти запросы породили клиентоориентированную индустрию-гиганта, которая корпит над своими детящими.

Я решил хотя бы отчасти прикоснуться к одной из отраслей этой индустрии - Android разработке. И далее я продемонстрирую свою работу, которая шла на проятжении 3 месяцев.

2. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

2.1. Концепция

За основу мессенжера были взяты такие базовые возможности, как:

1) страница с авторизацией и возможностью как войти уже зарегистрированному пользователю, так и зарегистрировать нового.
2) чат двух пользователей.
3) возможность разлогиниться.
4) отдельная возможность для пользователей отправлять друг другу картинки, хранящиеся у них на устройствах.

В программе активно используется функционал RealTime Data Base в FireBase от корпорации Google, иначе говоря, при отправке и получении сообщений и данных пользователей вызваются методы, используемые корпорацией Google и которые, к сожалению, находятся в закрытом доступе. Так что их описание в дальнейшем будет довольно-таки поверхностным.

2.2. Разбор используемых сторонних методов из Java библиотек

Предлагаю ознакомиться с некоторыми методами, которые были использованы в проекте внутри классов исполняемой программы:

• **onCreate** - первый метод, с которого начинается выполнение activity. В этом методе activity переходит в состояние Created. Этот метод обязательно должен быть определен в классе activity. В нем производится его первоначальная настройка. В частности, создаются объекты визуального интерфейса. Этот метод получает объект Bundle, который содержит прежнее состояние activity, если оно было сохранено. Если activity заново создается, то данный объект имеет значение null. Если же activity уже ранее была создана, но находилась в приостановленном состоянии, то bundle содержит связанную с activity информацию.

После того, как метод onCreate() завершил выполнение, activity переходит в состояние Started, и и система вызывает метод onStart()

• Метод **findViewById()** позволяет получить ссылку на View, которая размещена в разметке через его идентификатор.

Чтобы разместить пользовательский интерфейс, необходимо вызвать метод **setContentView()**, т.к. изначально экран активности пуст. У метода есть две перегруженные версии. Можно передать в параметре либо экзем-

пляр компонента (View), либо идентификатор ресурса (наиболее распространённый способ).

• Интерфейс **setOnClickListener()**. Кнопка присваивает себе обработчика с помощью метода setOnClickListener (View.OnClickListener l), т.е. подойдет любой объект с интерфейсом View.OnClickListener. Мы можем указать, что наш класс Activity будет использовать интерфейс View.OnClickListener.

• Дочерняя активность может произвольно возвратить назад объект Intent, содержащий любые дополнительные данные. Вся эта информация в родительской активности появляется через метод обратного вызова **Activity.onActivityResult()**, наряду с идентификатором, который она первоначально предоставила.

Если дочерняя активность завершится неудачно или будет закрыта пользователем без подтверждения ввода через кнопку Back, то родительская активность получит результат с кодом RESULT_CANCELED.

Метод принимает несколько параметров:

- код запроса - тот код, который использовался для запуска дочерней активности, возвращающий результат.
- результирующий код - код результата, поступающий от дочерней активности, как правило, RESULT_OK или RESULT_CANCELED
- данные - намерение может включать в себя различные данные в виде параметра extras внутри намерения.

• Т.к. в нашем приложении могут так или иначе появиться двовольно объёмные списки, было принято решение использовать **ViewHolder**. Это одна из методик для улучшения производительности работы больших списков. Метод findViewById() достаточно тяжёлый в плане потребления ресурсов, так что нужно избегать его, если в нём нет прямой необходимости. ViewHolder сохраняет ссылки на необходимые в элементе списка шаблоны. Этот ViewHolder прикреплён к элементу методом setTag(). Каждый элемент списка может содержать применённую ссылку. Если элемент очищен, мы можем получить ViewHolder через метод getTag().

• **LayoutInflater** используется для создания нового View (или же Layout) из одного из XML-макетов.

2.3. Описание функционала классов исполняемой программы

Программа содержит следующие классы:

Главная Activity - **ChatActivity**;

Внутри реализованы такие алгоритмы как: ограничение объёма возможного сообщения, активация/деактивация кнопки отправки сообщения (то есть, если EditText пуст, то и кнопка неактивна), очистка EditText в XML вёрстке после нажатия кнопки отправки сообщения, при условии отсутствия имени пользователя (или же на начальных этапах, когда в приложении не было авторизации, для тестировки и отладки) устанавливается значение по умолчанию - default. Помимо всего прочего тут создан метод, благодаря, которому осуществляется выход на страницу регистрации, чтоб заново зайти или зарегистрироваться.

Message - класс, определяющий параметры нашего сообщения, внутри конструктор для полей:

- String text - текст самого сообщения;
- String name - имя пользователя (отображается под текстом сообщения);
- String sender - id пользователя-отправителя;
- String recipient - id пользователя-получателя;
- String imgUrl - если картинки нет, то содержит null, если есть, то Uri нашего изображения, которое содержится в RealTimeDataBase;
- boolean myMessage - переменная отвечающая за отображение пузыря сообщения и раположение текста (если myMessage = true, то пузырь и текст внутри находятся справа, иначе - слева);

User - класс, определяющий параметры нашего пользователя (те, что он ввёл при регистрации и имя, которое он может менять при входе в приложение), внутри конструктор для полей:

- String name - имя пользователя;
- String email - email пользователя;
- String id - его уникальный идентификатор;
- int avatarMockUpResource - заглушка для аватара;

MessageAdapter - адаптер, содержащий объекты "Message". Именно в нём у нас определяется позиция сообщения, то есть, каким по счёту его отображать в общем списке (понятно, чем позже было отправлено сообще-

ние, тем ниже в списке оно будет). Так же внутри метода идёт определение того, какой тип информации послал нам пользователь: текст или картинку.

SignInActivity - класс отвечающий за авторизацию пользователя. Авторизация может проходить по одному из двух путей: вход для уже зарегистрированного пользователя и регистрация нового пользователя. В первом случае нам предлагается заполнить три поля: электронный почтовый адрес, который сравнивается алгоритмами Google и проверяется, существует ли он в базе или нет, пароль, который так же, как и электронный почтовый адрес, проверяется на сервере на соответствие введённому и имя пользователя. Во втором случае добавляется еще и строка подтверждения пароля, для уверенности в том, что он был введёт так, как задумывалось. Внутри стоит несколько проверок, например, длина пароля должна содержать 8 и более символо, строка с электронным адресом не должна быть пустой.

UserListActivity - класс, в котором реализованы переходы внутрь чатов двух пользователей (а не чату из вообще всех пользователей, как это было на ранних этапах разработки), так же тут есть проверка, направленная на исключение текущего пользователя из списка чатов (хотя, его можно было бы оставить, как это сделали ВКонтакте и Телеграм, у них он назван "Сохраненные сообщения").

3. ПРАКТИЧЕСКАЯ ЧАСТЬ

3.1. Часть I - FireBase и RealTime DataBase

В силу того, что почти весь функционал чата построен на основе Fire Base от корпорации Google, рассмотрим для начала некоего рода код, который играет роль правил для всех пользователей, на самой странице FireBase:

```
{
  "rules": {
    ".read": "auth != null",
    ".write": "auth != null"
  }
}

```

Тут мы для "*.read*" и "*.write*" в качестве разрешений для прочтения и записи ставим параметр "$auth != null$", что позволяет читать сообщения и/или писать в нашу базу данных от FireBase. То есть неавторизованный пользователь не сможет ни прочитать какое-то сообщение, ни написать что-либо в нашу базу данных.

Далее в разделе аутентификации мы выбираем параметр "*Email/password*", как самый простой и распространённый тип авторизации в любого рода приложениях.

Для работоспособности кода добавим зависимость в "*build.gradle* ($Module : app$)":

```
dependencies {
  //...
  implementation platform('com.google.firebase:
firebase-bom:29.0.2')
    implementation 'com.google.firebase:firebase-auth
'
}

```

3.2. Часть II - Программная реализация

Так как листинг программы целиком слишком велик, предлагаю рассмотреть конкретные участки программы внутри классов по отдельности и разобраться, что и каким образом они делают.

Класс SignInActivity

В классе "*SignInActivity*" в методе "*onCreate*" уже после инициализации переменных и связывания их при помощи id с элементами в XML-разметке мы пишем код следующего рода:

```
1     //...
2     //here we set OnClickListener
3     //for loginSignUpBtn to hear
4     //the button activity
5     //...
6     loginSignUpBtn.setOnClickListener(new View.
    OnClickListener()
7     {
8       @Override
9       public void onClick(View view)
10      {
11        loginSignUpUser(emailEditText.getText().
    toString().trim(), passwordEditText.getText().
    toString().trim());
12      }
13    });
14
15    if(auth.getCurrentUser() != null)
16    {
17      startActivity(new Intent(SignInActivity.this,
    UserListActivity.class));
18    }
19
```

И для того чтоб всё заработало мы создаём метод "*loginSignUpUser*" в котором пишем конструкцию "*if/else*", благодаря чему мы разделяем пользователей, которые только лишь логинятся и которые собираются впервые зайти в приложение и регистрируются. В силу того, что по своей сути что внутри "*if*", что внутри "*else*" конструкции примерно похожи, рассмотрим только конструкцию предназначенную для регистрации пользователя:

```
1     //...
2     //here we check if our two
3     //passwords are equal
4     //...
5     if(!passwordEditText.getText().toString().trim().
```

```
equals(confirmPasswordEditText.getText().toString().
trim()))
  {
    Toast.makeText(this, "Password mismatch", Toast.
LENGTH_SHORT).show();
  }
  //...
  //check if password less than 8 symbols
  //...
  else if(passwordEditText.getText().toString().trim
().length() < 8)
  {
    Toast.makeText(this, "Password must contain 8 or
more symbols", Toast.LENGTH_SHORT).show();
  }
  //...
  //check if string with email
  //is empty and shows a message
  //for our user
  //...
  else if(emailEditText.getText().toString().trim().
equals(""))
  {
    Toast.makeText(this, "No email to sign up", Toast
.LENGTH_SHORT).show();
  }
  else
  {
    //...
    //here is onCompleteListener
    //...
  }

```

Сам "*onCompleteListener*" в "*else*" выглядит следующим образом:

```
  auth.createUserWithEmailAndPassword(email, password
)

  .addOnCompleteListener(this, new OnCompleteListener
```

```
<AuthResult>()
  {
    @Override
    public void onComplete(@NonNull Task<AuthResult>
task)
    {
      if (task.isSuccessful())
      {
        //...
        //Log in success, we create a new user
        //and send to UserListActivity.class
        //our new intent whith tag (key) "userName"
        //...
        FirebaseUser user = auth.getCurrentUser();
        createUser(user);
        Intent intent = new Intent(SignInActivity.
this, UserListActivity.class);
        intent.putExtra("userName", nameEditText.
getText().toString().trim());
        startActivity(intent);
      }
      else
      {
        // If sign in fails, display a message to the
 user.
        //Log.w(TAG, "createUserWithEmail:failure",
task.getException());
        Toast.makeText(SignInActivity.this, "
Authentication failed.", Toast.LENGTH_SHORT).show();
      }
    }
  });

```

Так же для упрощения задачи разделения логинящихся и регистрирующихся пользователей был написан ещё один метод:

```
  public void toggleLoginMode(View view) {
    if(loginModeActive) {
      loginModeActive = false;
```

```
4         loginSignUpBtn.setText("Sign up");
5         toggleLoginSignUpTextView.setText("Or, log in")
  ;
6         confirmPasswordEditText.setVisibility(View.
  VISIBLE);
7       } else {
8         loginModeActive = true;
9         loginSignUpBtn.setText("Log in");
10        toggleLoginSignUpTextView.setText("Or, sign up
  ");
11        confirmPasswordEditText.setVisibility(View.
  VISIBLE);
12        confirmPasswordEditText.setVisibility(View.GONE
  );
13      }
14    }
15
```

Этот метод по своей сути нужен лишь для визуального комфорта пользователя. Он отвечает за отображение 3 строчек (строки пароля, email и имени пользователя) на странице регистрации для тех пользователей которые хотят залогиниться и 4 строчки (строки пароля, подтверждения пароля, email и имени пользователя) для тех, кто хочет зарегистрироваться.

Для создания же пользователя используется приватный метод:

```
1     private void createUser(FirebaseUser firebaseUser)
  {
2       User user = new User();
3       user.setId(firebaseUser.getUid());
4       user.setEmail(firebaseUser.getEmail());
5       user.setName(nameEditText.getText().toString().
  trim());
6       usersDatabaseReference.push().setValue(user);
7     }
8
```

Класс ChatActivity

Один из интересных алгоритмов, найденных в документациях и на сайтах типа coderoad это "*addTextChangedListener*", который активирует и деактивирует кнопку отправки. Когда пользователь что-то ввёл, хотя бы 1

символ, в TextEdit, кнопка загорается фиолетовым в знак того, что теперь сообщение введено и его можно отправить.

```
1     messageEditText.addTextChangedListener(new
  TextWatcher() {
2       @Override
3       public void beforeTextChanged(CharSequence
  charSequence, int i, int i1, int i2) {}
4
5       @Override
6       public void onTextChanged(CharSequence
  charSequence, int i, int i1, int i2) {
7         if(charSequence.toString().trim().length() > 0)
   {
8           sendMessageButton.setEnabled(true);
9         } else {
10          sendMessageButton.setEnabled(false);
11        }
12      }
13
14      @Override
15      public void afterTextChanged(Editable editable)
  {}
16    });
17
```

Тут мы при помощи метода "$toString()$" приводим пременную "$charSequence$" к строковому типу, а затем при помощи "$trim()$" обрезаем пробелы в начале и в конце, если таковые имеются, чтоб сравнение строк стало более достоверным. И в случае, если длина этой строки у нас > 0, кнопка активируется.

Вызываемый ниже метод не позволяет пользователю ввести более 500 символов в одном сообщении, это создано исключительно из эстетических целей, чтоб текст пользователя не перекрывал экран с остальными сообщениями.

```
1     messageEditText.setFilters(new InputFilter[] {new
  InputFilter.LengthFilter(500)});
2
```

Далее рассмотрим метод отправки сообщений:

```
1     sendMessageButton.setOnClickListener(new View.
```

```
OnClickListener() {
    @Override
    public void onClick(View view) {

      Message message = new Message();
      message.setText(messageEditText.getText().
toString());
      message.setName(userName);
      message.setSender(auth.getCurrentUser().getUid
());
      message.setRecipient(recipientUserId);
      message.setImgUrl(null);

      messagesDatabaseReference.push().setValue(
message);

      messageEditText.setText("");
    }
  });

```

Тут происходит присваивание полям нового экземпляра класса "*Message*" значений (при помощи "сеттеров которые позволяют обратиться к приватным значениям заданным в классе, мы устанавливаем значения text, name, sender, recipient и ImgUrl = null для message).

В случае, если мы загружаем какой-то файл, а в нашем случае это картинка, FireBase предлагает нам использовать следующий алгоритм действий:

```
 Task<Uri> urlTask = uploadTask.continueWithTask(new
 Continuation<UploadTask.TaskSnapshot, Task<Uri>>()
{
    @Override
    public Task<Uri> then(@NonNull Task<UploadTask.
TaskSnapshot> task) throws Exception {
      if (!task.isSuccessful()) {
        throw task.getException();
      }

      // Continue with the task to get the download
URL
```

```
9           return imageReference.getDownloadUrl();
10        }
11      }).addOnCompleteListener(new OnCompleteListener<Uri
    >() {
12        @Override
13        public void onComplete(@NonNull Task<Uri> task) {
14          if (task.isSuccessful()) {
15            Uri downloadUri = task.getResult();
16            Message message = new Message();
17            message.setImgUrl(downloadUri.toString());
18            message.setName(userName);
19            message.setSender(auth.getCurrentUser().
    getUid());
20            message.setRecipient(recipientUserId);
21            messagesDatabaseReference.push().setValue(
    message);
22          } else {
23            // Handle failures
24            // ...
25          }
26        }
27      });
28
```

Тут мы так же инициализируем переменные класса "$Message$", но в строковую переменную "$ImgUrl$" кидаем $downloadUri.toString()$, благодаря чему мы потом можем видеть картинку с телефона другого пользователя, ведь мы по сути подгружаем её с сервера, куда она была отправлена (алгоритм схож с алгоритмом из web-разработки).

Класс UserAdapter

Сам по себе класс не очень интересен с точки зрения каких-то алгоритмов или нововведений. Если вкратце, то "$UserAdapter$" нам нужен для создания очереди из пользователей, которые извлекаются из массива $ArrayList < User > users$, и при этом располагаются сначала те, кто был зарегистрирован в первую очередь (сверху вниз). При этом у каждого пользователя есть 2 поля: имя и заглушка для аватара, которые инициализируются внутри.

Класс MessageAdapter

Этот класс такой же по своему назначению, как и предыдущий, даже если смотреть на порядок появления сообщений (сначала более старые и так сверху вниз). Интерес представляют только две конструкции "$if/else$":

```
1     int viewType = getItemViewType(position);
2     if(viewType == 0) {
3       layoutResource = R.layout.my_message_item;
4     } else {
5       layoutResource = R.layout.your_message_item;
6     }
7
8     boolean isText = newMessage.getImgUrl() == null;
9
10    if(isText) {
11      viewHolder.messageTextView.setVisibility(View.
    VISIBLE);
12      viewHolder.photoImageView.setVisibility(View.GONE
    );
13      viewHolder.messageTextView.setText(newMessage.
    getText());
14    } else {
15      viewHolder.messageTextView.setVisibility(View.
    GONE);
16      viewHolder.photoImageView.setVisibility(View.
    VISIBLE);
17      Glide.with(viewHolder.photoImageView.getContext()
    ).load(newMessage.getImgUrl()).into(viewHolder.
    photoImageView);
18    }
19
```

Первая конструкция нужна для отображения нужного нам пузыря, где будет располагаться непосредственно сам текст сообщения. То есть, если viewType == 0, то моё сообщение отображается справа и пузырь ориентирован вправо, иначе (для собеседника от моего лица) его расположение слева и смотрит он так же влево.

Вторая конструкция нужна для определения, какой перед нами тип данных: картинка или текст. В зависимости от этого мы убираем с экрана либо ImageView, либо TextView.

Класс UserListActivity

Тут присутствует важная для функционала конструкция - "*onUserClickListener*":

```
1     userAdapter.setOnUserClickListener(new UserAdapter.
    OnUserClickListener() {
2       @Override
3       public void onUserClick(int position) {
4         goToChat(position);
5       }
6     });
7
8      private void goToChat(int position) {
9       Intent intent = new Intent(UserListActivity.this,
     ChatActivity.class);
10      intent.putExtra("recipientUserId", userArrayList.
    get(position).getId());
11      intent.putExtra("userName", userName);
12      startActivity(intent);
13    }
14
```

Здесь происходит переход к конкретному собеседнику по клику пользователя по средством передачи переменной *position*, по своей сути это индекс элемента массива "*userArrayList*", у которого мы после получаем *id* и к которому впоследствие будут идти сообщения.

Так же при создании пользователя была добавлена пустая иконка аватара. И при каждом последующем создании очередного пользователя на экране будет высвечиваться картинка и имя. Для этого нам потребуется заглушка, которую мы устанавливаем следующим образом:

```
1     public void onChildAdded(@NonNull DataSnapshot
    snapshot, @Nullable String previousChildName) {
2       User user = snapshot.getValue(User.class);
3       if(!user.getId().equals(auth.getCurrentUser().
    getUid())) {
4         user.setAvatarMockUpResource(R.drawable.
    user_image);
5         userArrayList.add(user);
6         userAdapter.notifyDataSetChanged();
7       }
```

```
8       }
9
```

Вся работа целиком лежит в Git репозитории[1].

[1]https://github.com/GrandF17/Chat-app

4. ЗАКЛЮЧЕНИЕ

В заключение я бы хотел сказать, что после проделанной работы мне уже не кажется сложным, или невозможным написать свой собственный мессенжер. Многие трудные моменты, с которыми я столкнулся во время реализации, остались в прошлом, и я прекрасно осознаю, что этот проект можно улучшить и сделать надёжным в несколько сотен раз. Возможно, я буду работать над ним и в будущем, что довольно-таки вероятно.

5. СПИСОК ЛИТЕРАТУРЫ

1) Блинов И.Н., Романчик В.С. Java: методы программирования.
2) Google. Библиотека алгоритмов FireBase.
3) JavaRush.ru
4) Адитья Бхаргва. Грокаем алгоритмы.
5) Образовательный портал Udemy.