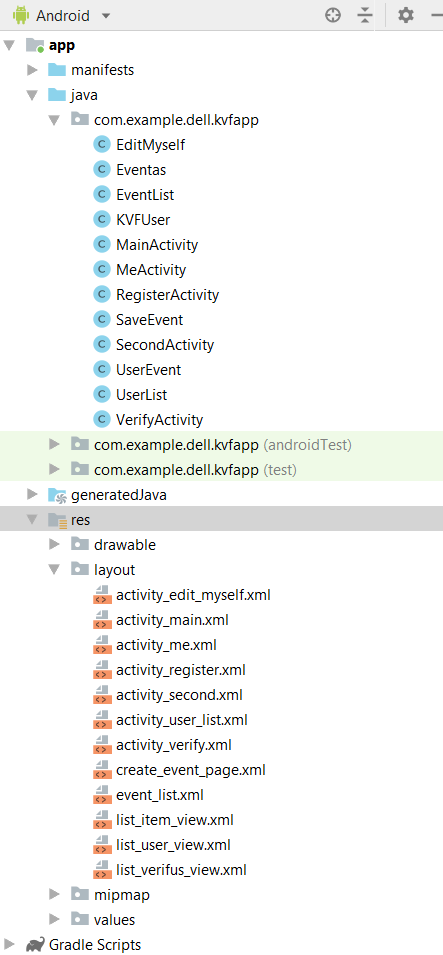
Criterion C: Development

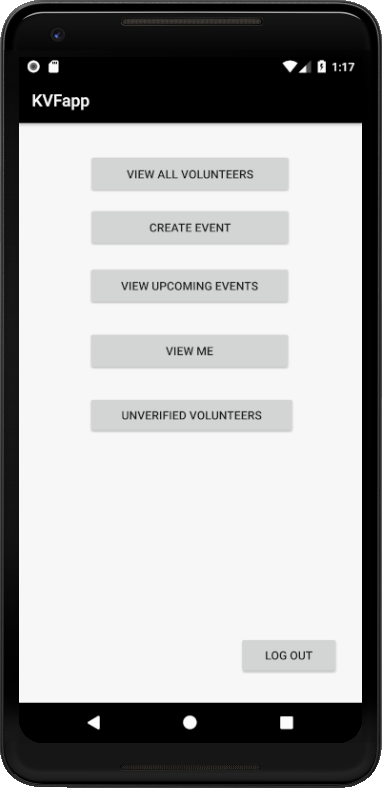
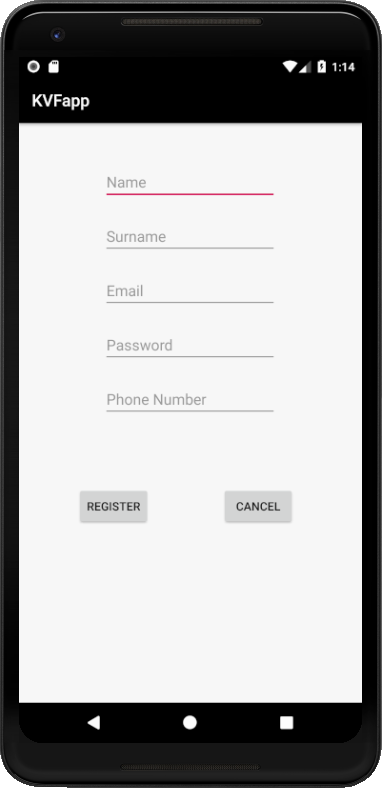
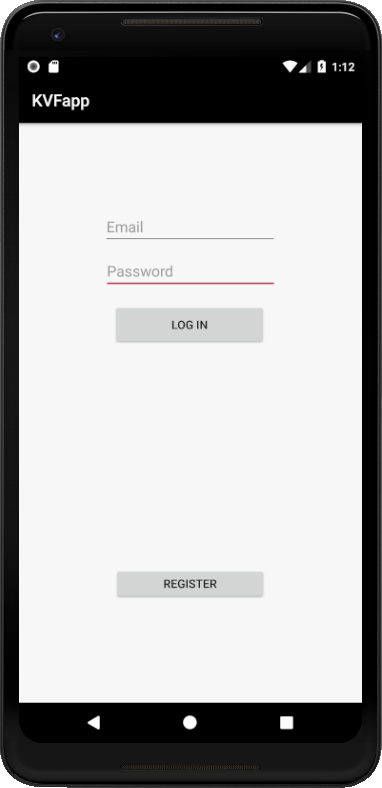
**List of techniques used:**

1. Visual design (GUI)
   1. *Android studio* 3.3.2 for Windows 64-bit
   2. *XML* 1.0
2. Algorithmic thinking
   1. Java JDK 8, Android Development Libraries
   2. Object-Oriented programming (encapsulation)
   3. Modularity
   4. HashMap (implementation of Map interface in Java), java.util package
   5. Abstract data types, ArrayList, java.util library
3. Database technology
   1. NoSQL cloud real-time *Firebase* database
4. Development and testing
   1. *Stack Overflow*
   2. Pixel XL API 26 emulator
   3. Nexus One API 26 emulator
5. Android phones (Samsung Galaxy J5, Samsung Galaxy S8)The application was divided into several fragments, using Java classes and XML design files. Since modularity and fragments’ structures allow each part to be tested and debugged individually, this development choice ensured easier maintenance of the project.

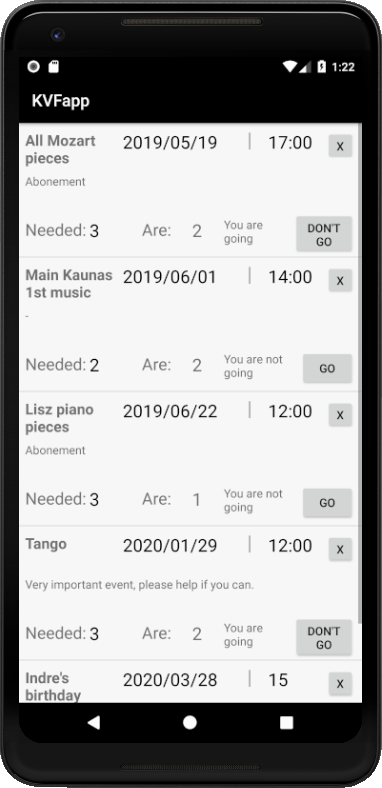
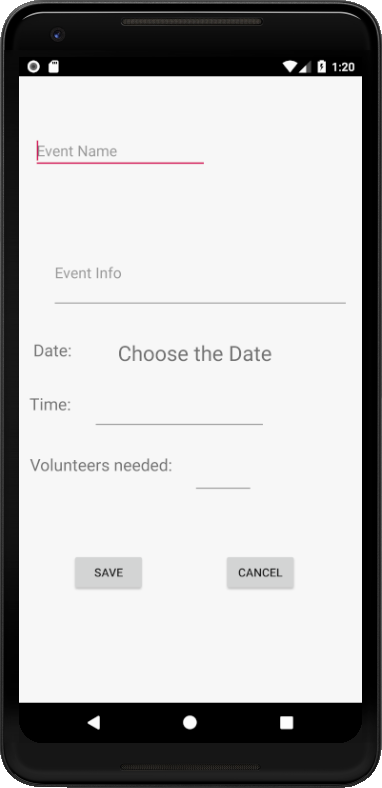
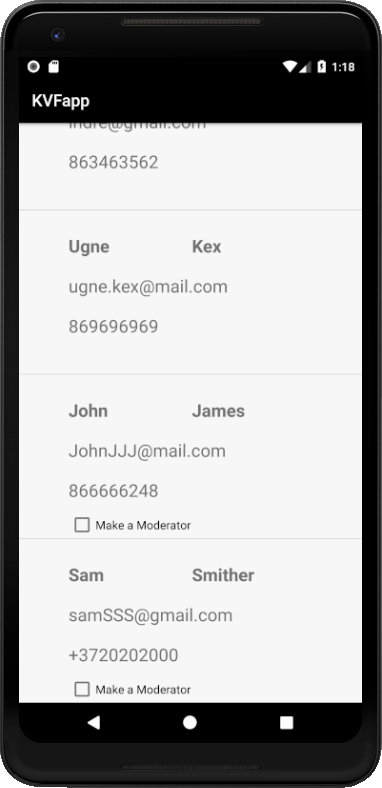


*Figure 1. Project structure.*

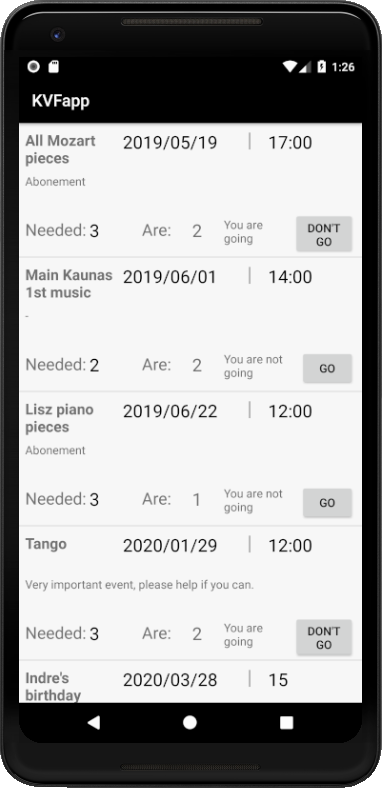
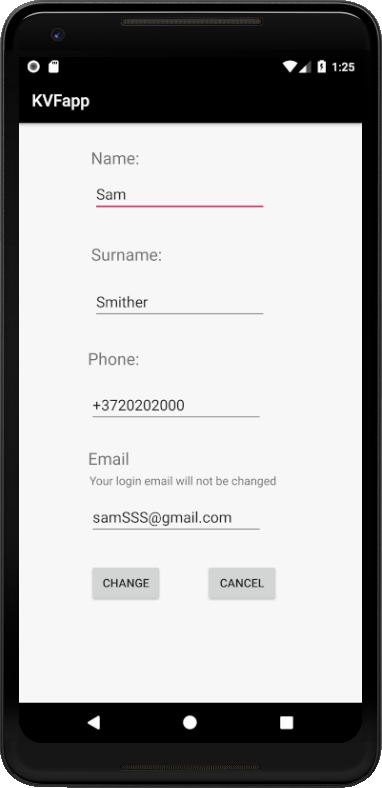
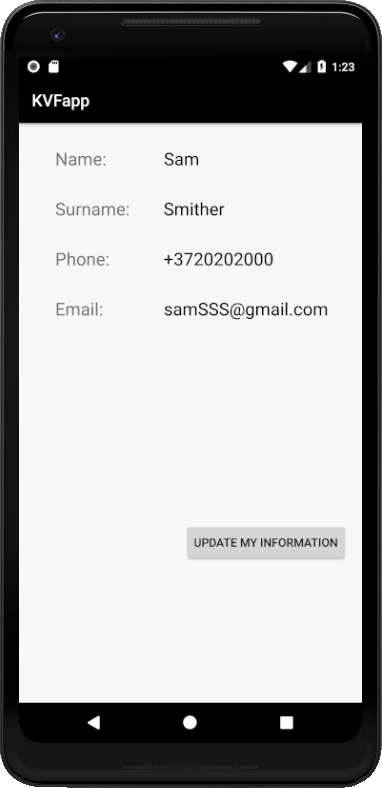
Images below show the graphical interface of KVF application. Interface was created by using *Android Studio* XML files and every visual interface has its own Java class. After discussing with **the client**, the program had to be divided in two parts - Volunteer and Moderator part. *GUI 4*, *GUI 5*, *GUI 6, GUI 7* and *GUI 13* show the Moderator side of the application, while *GUI 10* and *GUI 11* show the Volunteer side of the app and *GUI 11* and *GUI 12* show the not verified user side of the application. Some screens are the same for all sides - volunteer, moderator and no-verified user (*GUI 7,*  *GUI 8)*. *GUI 1* and *GUI 2* are login and register pages, therefore are not affected by any side.



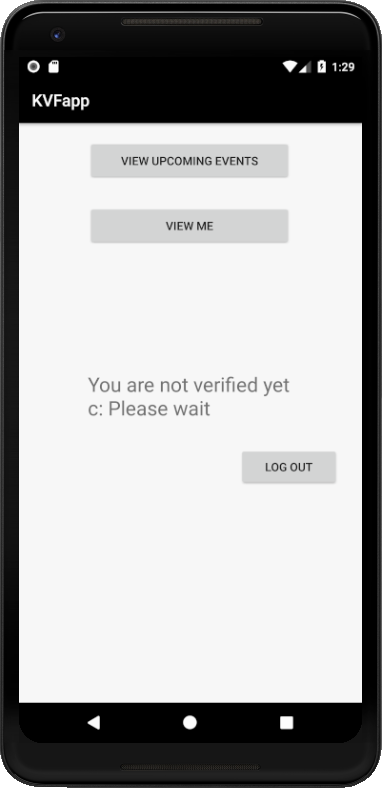
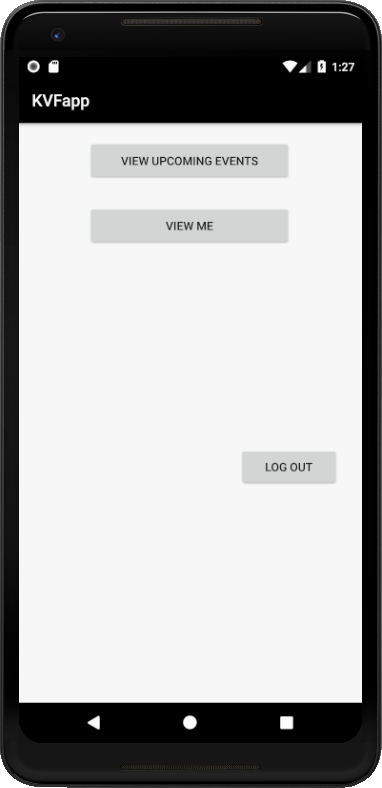
*GUI 1. Login page. GUI 2. Register page. GUI 3. Main screen of Moderator.*



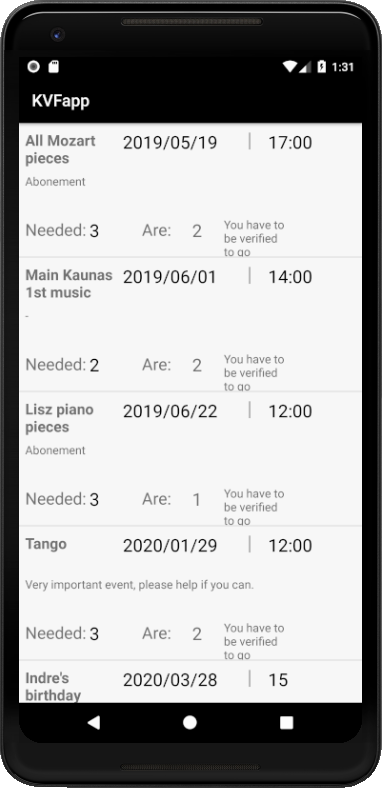
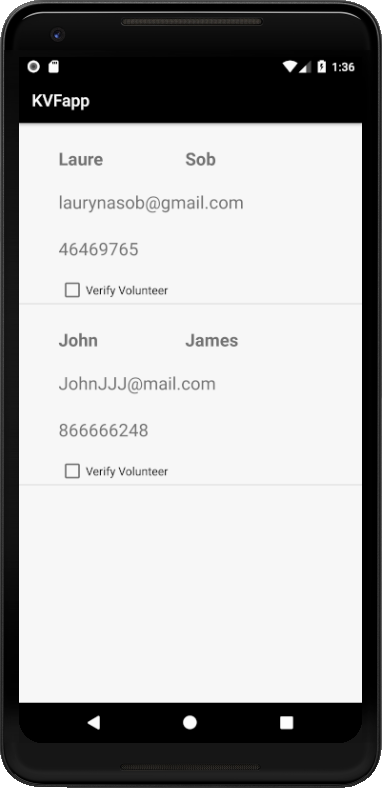
*GUI 4. Volunteer list view. GUI 5. Creating event view. GUI 6. Events page for Moderators.*



*GUI 7. User information screen. GUI 8. Edit User information. GUI 9. Events page for volunteers.*



*GUI 10. Main screen for volunteers. GUI 11. Not verified users main screen.*

*GUI 12. Not Verified users Event screen. GUI 13. List of not verified Users.*

**Algorithmic thinking**

**Object oriented programming**

In order to ensure faster development and code reuse, OPP was necessary. *Private* instance variables provided encapsulation, which enhances security, as the variables can be modified or accessed only by using Public accessor or mutator methods, thus cannot be called outside the class.

The code, presented below was used to get and modify the values of the events (Figure 2). Event needed to be represented as an object with special properties, thus *Eventas* class was created. This provided code reuse as well as made an object insert in the database easier.

| //Class of Event object, which will be used to create different events which will be put into the database  **public class** Eventas  {  *//private instance variables*  **private** String **title**;  **private** String **info**;  **private** String **eventDate**;  **private** String **eventTime**;  **private** String **needVolunteers**;  **private** String **id**;  //constructor methods  **public** Eventas(String t, String i, String d, String m, String s, String Are, String id)  {  **this**.**title** = t;  **this**.**info** = i;  **this**.**eventDate** = d;  **this**.**eventTime** = m;  **this**.**needVolunteers** = s;  **this**.**areVolunteers** = Are;  **this**.**id** = id;  **this**.**VolID** = **null**;  }  **public** Eventas()  {  **this**.**title** = **null**;  **this**.**info** = **null**;  **this**.**eventDate** = **null**;  **this**.**eventTime** = **null**; |
| --- |
| *Figure 2. Eventas class code.* |

| **this**.**needVolunteers** = **null**;  **this**.**id** = **null**;  **this**.**VolID** = **null**;  }  //Accessor and Modifier methods  **public** String getTitle() {**return title**;}  **public** String getInfo() {**return info**;}  **public** String getEventDate() {**return eventDate**;}  **public** String getEventTime() {**return eventTime**;}  **public** String getNeedVolunteers() {**return NeedVolunteers**;}  **public** String getId() {**return id**;}    **public void** setTitle(String t) {**title** = t;}  **public void** setInfo(String i) {**info** = i;}  **public void** setEventDate(String ed) {**eventDate** = ed;}  **public void** setEventTime(String et) {**eventTime** = et;}  **public void** setNeedVolunteers(String nv) {**needVolunteers** = nv;}  **public void** setId(String i){**id** = i;}  } |
| --- |
| *Figure 2. Continued.* |

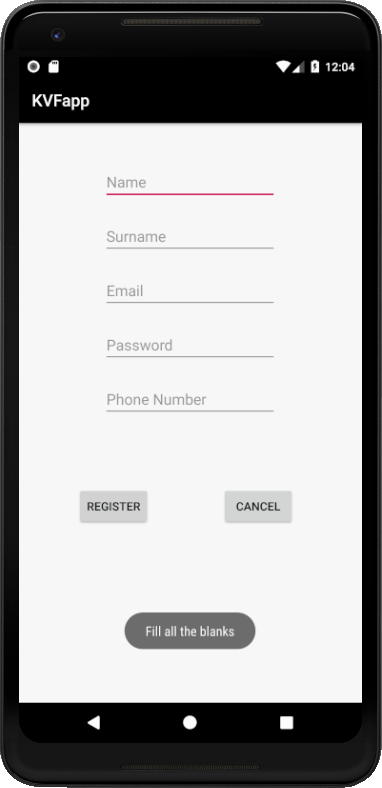
**Registration**

One of the client’s requests was authentication, therefore *StartRegister* method was created. The application has to present essential information about every volunteer, such as name and surname, as well as contact information: a phone number and email address, while every volunteer has to have their own account. In order to ensure security, the firebase authentication system was used, which provides backend services as well as Software Development Kits for easier implementation.

All the information except password was saved in the firebase database. Passwords are coded and stored into firebase Authentication module’s database.

The code presented below was used to registrate the users and check whether they provided all the needed information. In order to ensure security, the password should be at least 7 characters long which is checked by using if cycle. If the user does not meet these requirements, the registration is not completed and the notification message pops, which explain what is needed for the registration to be successful.

| //registration process  **private void** startRegister(){  **final** String name = **mNameT**.getText().toString();  **final** String email = **mEmailT**.getText().toString();  **final** String phone = **mPhoneT**.getText().toString();  **final** String surname = **mSurnameT**.getText().toString();  String password = **mPassT**.getText().toString();  //checking if no empty spaces are left  **if**(!TextUtils.*isEmpty*(name)&& !TextUtils.*isEmpty*(surname)&&!TextUtils.*isEmpty*(email)&&!TextUtils.*isEmpty*(password)&&!TextUtils.*isEmpty*(phone)) {  //password length has to be longer than 6 characters  **if** (password.length() > 6) {  **mProgress**.setMessage(**"Signing Up"**);  **mProgress**.show();  //creating user  **mAuth**.createUserWithEmailAndPassword(email, password).addOnCompleteListener(**new** OnCompleteListener<AuthResult>() {  @Override  **public void** onComplete(@NonNull Task<AuthResult> task) {  **if** (task.isSuccessful()) {  String id = **mAuth**.getCurrentUser().getUid();  //new User object is created  KVFUser NewUser = **new** KVFUser(name, surname, email, phone, id );  //User is put into the database  **mDatabase**.child(id).setValue(NewUser);  **mProgress**.dismiss();  Intent mint = **new** Intent(RegisterActivity.**this**, SecondActivity.**class**);  mint.addFlags(Intent.***FLAG\_ACTIVITY\_CLEAR\_TOP***);  startActivity(mint);  } **else** {**mProgress**.dismiss();  //if the registration is unsuccessful, the notification message pops  Toast.*makeText*(RegisterActivity.**this**, **"Something is wrong"**, Toast.***LENGTH\_SHORT***).show();}  }  });  }**else** Toast.*makeText*(RegisterActivity.**this**, **"Password is too short"**, Toast.***LENGTH\_SHORT***).show();  }  **else** {  **mProgress**.dismiss();  Toast.*makeText*(RegisterActivity.**this**, **"Fill all the blanks"**, Toast.***LENGTH\_SHORT***).show();  }  } |
| --- |
| *Figure 3. Registration process code.* |



*GUI 14. Example of notification message.*

**Login**

The login is handled by the *checkLogin* method. If no details are put, the message pops, informing the user to fill in the details. After the user puts his their details, the Firebase Authentication module database is searched by the *checkUserExist* method. *OnDataChange* method is used to retrieve the data from the database. If such user exists and the password is correct the user is redirected to the main page (*SecondActivity*).

| **private void** checkLogin(){  String email = **mEmailField**.getText().toString().trim();  String pass = **mPasswordField**.getText().toString().trim();  //checking whether the details are put  **if** (!TextUtils.*isEmpty*(email) && !TextUtils.*isEmpty*(pass)){  **nProgress**.setMessage(**"Logging In"**);  **nProgress**.show();  **mAuth**.signInWithEmailAndPassword(email, pass).addOnCompleteListener(**new** OnCompleteListener<AuthResult>() { |
| --- |
| *Figure 4. Login process code.* |

| @Override  **public void** onComplete(@NonNull Task<AuthResult> task) {  **if**(task.isSuccessful()){  //checking if user with such details exist  checkUserExist();  **nProgress**.dismiss();  }**else**{  **nProgress**.dismiss();  Toast.*makeText*(MainActivity.**this**, **"Error Login"**, Toast.***LENGTH\_LONG***).show();  }  }  });  }  **else** {**nProgress**.dismiss();  Toast.*makeText*(MainActivity.**this**, **"Fill in all the blanks"**, Toast.***LENGTH\_LONG***).show();  }  }  **private void** checkUserExist(){  **final** String uid = **mAuth**.getCurrentUser().getUid();  **mDatabase**.addValueEventListener(**new** ValueEventListener() {  @Override  //searching the database for the user  **public void** onDataChange(@NonNull DataSnapshot dataSnapshot) {  **if**(dataSnapshot.hasChild(uid)){  Intent loginIntent = **new** Intent(MainActivity.**this**, SecondActivity.**class**);  loginIntent.addFlags(Intent.***FLAG\_ACTIVITY\_CLEAR\_TOP***);  startActivity(loginIntent);  } **else** Toast.*makeText*(MainActivity.**this**, **"Hm, maybe create an account first..?"**, Toast.***LENGTH\_LONG***).show();  }  @Override  **public void** onCancelled(@NonNull DatabaseError databaseError) {  }  });  } |
| --- |
| *Figure 4. Continued.* |

**User participation in the event**

One of **the client’s** criteria was to show how many volunteers will attend a particular event. In order to do that, the list of users, who checked in, is retrieved from the firebase database by *onDataChange* method (figure 5). Since the value got was one String, it had to be splitted into separate substrings which hold the id values of the event and volunteers. Every list has the same id as the event which the volunteers are attending to. *UserList* object was created to store the id of the event and the list of volunteers attending the particular event. The Array List of *UserList* objects is created which stores the event id and the id of the users who are attending to the event.

| //database path reference  **aDatabase** = FirebaseDatabase.*getInstance*().getReference().child(**"VolGO"**);  **aDatabase**.addValueEventListener(**new** ValueEventListener() {  @Override  **public void** onDataChange(@NonNull DataSnapshot dataSnapshot) {  //loop to retrieve all childs from the database  **for** (DataSnapshot ds: dataSnapshot.getChildren()) {  ds.getValue(**true**);  //splitting the database value into separate strings  String temp = ds.getValue().toString();  temp = temp.substring(1,temp.length()-1);  ArrayList putList = **new** ArrayList<String>();  String[] arr = temp.split(**","**);  **for**(**int** c = 0; c<arr.**length**; c++)  {  arr[c]=arr[c].substring(arr[c].indexOf(**'='**)+1);  putList.add(arr[c]);  }  //creating a new object and putting it into the Array list  **EventasList**.add(**new** UserEvent(ds.getKey(), putList));  }  }  @Override  **public void** onCancelled(@NonNull DatabaseError databaseError) {  }  }); |
| --- |
| *Figure 5. Code of retrieving list of users which participate in the event.* |

The code below shows the algorithm which checks whether the particular user is attending to the event as well as calculates the number of volunteers attending the particular event by calculating the length of the *UserID* Array list (figure 6). Firstly, if the event has any volunteers at all. If it does have volunteers, it is checked whether the user is in the array by looping through the Array list. Boolean *IsVol* is used to determine whether the user is going to the event.

| **int** EventIdInt = -1;  **int** VolLength = 0;  **boolean** IsVol = **false**;  //checking if a list with the same id as the event exists  **if**(**EventasList**.size()!=0) {  **for** (**int** h = 0; h < **EventasList**.size(); h++) {  **if** (**EventasList**.get(h).getEventID().equals(**TruListEvent**.get(position).getId())) {  EventIdInt = h;  //calculates the number of volunteers  VolLength = **EventasList**.get(EventIdInt).getUserID().size();  }  }  //checking if the user is checked in the event  **if**(EventIdInt>=0) {  **for** (**int** j = 0; j < VolLength; j++) {  **if** (**EventasList**.get(EventIdInt).getUserID().get(j).equals(**UserID**)) {  IsVol = **true**;  }  }  }  }  **final int** EventasListID = EventIdInt;  VolAre.setText(**""** + VolLength); |
| --- |
| *Figure 6 .Code which checks whether user participates in the event.* |

**Moderator**

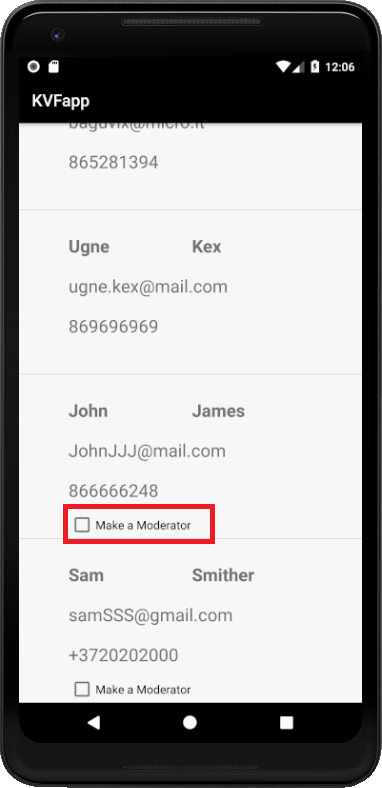
Another client’s criteria was to divide the program into two parts - to volunteer and moderator parts. In order to do it, I used boolean *isMod* which was stored in the database (figure 7).



*Figure 7. Database*

One of **the client’s** requests was the ability for a moderator to turn a volunteer into the moderator. Checking the checkbox changes the boolean *IsMod* value in the database and thus turns volunteer into moderator. If the user in the list is already a moderator, the checkbox *checkBox* (element of GUI) is invisible. Also, the name of the child in the database is the id of the object.

| Boolean Mod;  Mod = **TruListUser**.get(position).getIsMOD();  //checking whether the user is a moderator  **if**(Mod == **true**)  {  checkBox.setVisibility(checkBox.***INVISIBLE***);  }  checkBox.setOnClickListener(**new** View.OnClickListener() {  @Override  **public void** onClick(View view) {  String id = **TruListUser**.get(b).getId();  **TruListUser**.get(b).setIsMOD(**true**);  KVFUser NewUser = **TruListUser**.get(b);  //changing the *IsMod* value in the database  **uDatabase**.child(id).setValue(NewUser);  **TruListUser**.clear();  //notification message  Toast.*makeText*(UserList.**this**, **"The person became a moderator."**, Toast.***LENGTH\_SHORT***).show();  }  }); |
| --- |
| *Figure 8. Code which checks if user is a moderator.* |



*GUI 15. Example of CheckBox.*

**Sorting**

The events have to be sorted by dates in the descending order, so the nearest events could be displayed first. In order to achieve this, *SortMe* method was implemented, using the Bubble sort, as it is more effective than the Linear sort. Since the dates in the database were stored in the String format, they had to be splitted into Day, Month and Year ints. Since the long variable have more storage for number values, dates were turned they were turned into one long variable.

| **public** ArrayList<Eventas> SortMe(ArrayList<Eventas> e)  {  //looping through the array  **for**(**int** fr = 0; fr<e.size()-1; fr++)  {  **for** (**int** sr = 0; sr<(e.size()-fr-1);sr++) {  //splitting the date into day, month and year stings  String StringDate1 = e.get(sr).getEventDate();  String StringDate2 = e.get(sr+1).getEventDate();  String StringYear1 = StringDate1.substring(0, 4);  String StringYear2 = StringDate2.substring(0, 4);  String StringMonth1 = StringDate1.substring(5, 7);  String StringMonth2 = StringDate2.substring(5, 7);  String StringDay1 = StringDate1.substring(8);  String StringDay2 = StringDate2.substring(8);  //parsing String to int  **int** Year1 = Integer.*parseInt*(StringYear1);  **int** Year2 = Integer.*parseInt*(StringYear2);  **int** Month1 = Integer.*parseInt*(StringMonth1);  **int** Month2 = Integer.*parseInt*(StringMonth2);  **int** Day1 = Integer.*parseInt*(StringDay1);  **int** Day2 = Integer.*parseInt*(StringDay2);  //turning the date into long variable  **long** Date1 = Year1\*10000+Month1\*100+Day1;  **long** Date2 = Year2\*10000+Month2\*100+Day2;  //comparing two dates  **if** (Date1>Date2) {  Eventas temp = **new** Eventas();  temp = e.get(sr);  e.set(sr, e.get(sr + 1));  e.set(sr + 1, temp);}}}  **return** e;} |
| --- |
| *Figure 9. Sorting algorithm.* |

**Testing**

Testing and debugging was performed with Android SDK virtual devices and Samsung Galaxy J5 and Samsung Galaxy S8 phones. Android SDK virtual devices were used during the development process to test separate parts of the program. Real devices were used at the end of the process, during the beta testing.

Word count: 944