

ECE 558: Embedded System Programming Final Project Report

Young Guns FA (YGFA)

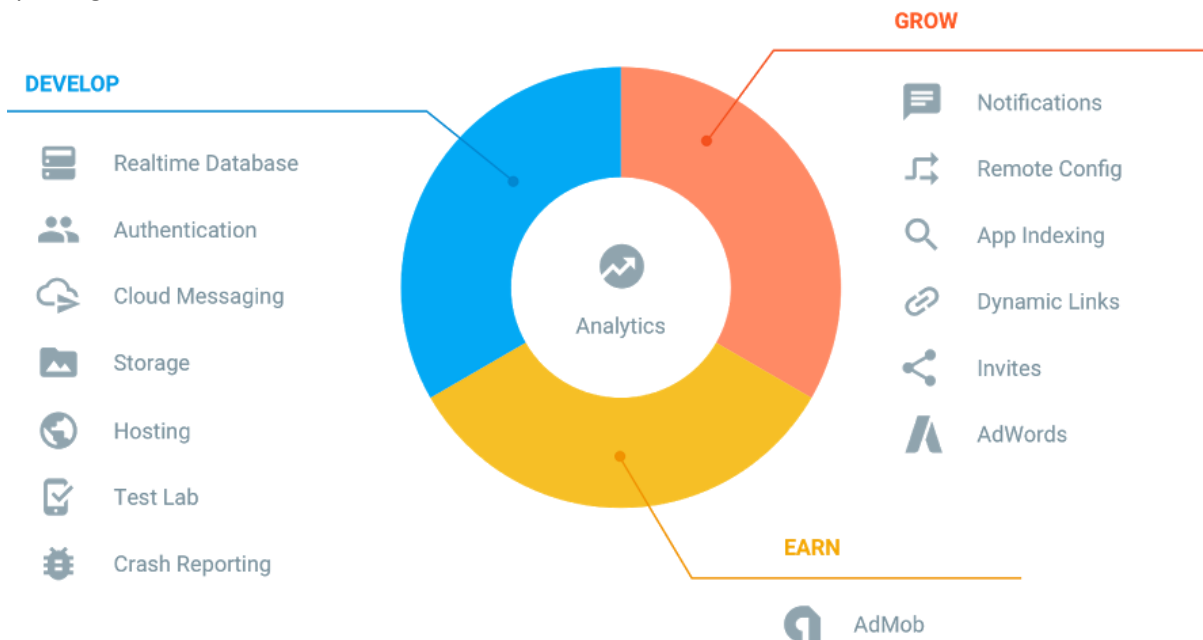
Project By: Arjun Sridhar, Chetan Bornarkar, Joel Jacob, Kaustubh Agashe

INTRODUCTION:

Young Guns FA is an android application built for a football academy which informs the user about the schedule of matches, events, training timings according to the user's age group and branch location. Through the app, the user can capture images, view them, upload them to the cloud and share the images with other users in real time. The app also includes one click navigation to the location of the venue.

What is Firebase?

Firebase is Google's mobile platform. We can develop high-quality apps and grow our user base on iOS, Android, or the Web. Firebase is made up of complementary features that we can mix-and-match to fit our needs including: Free & unlimited Analytics, A Real-time Database, User Authentication, Crash Reporting, Push Notifications, and more



For our purposes, we utilized **the Real-time DataBase, Authentication, Cloud Messaging (Notifications) and Storage**. We were able to obtain **analytics** of our user base as well as **crash reports**, both which proved to be a bonus to using this platform. The app can be made monetizable very easily by setting up AdMob and displaying Google ads.

References:

- <https://firebase.google.com/>

- <https://firebase.google.com/features/>
- <https://techcrunch.com/2016/11/07/googles-firebase-developer-platform-gets-better-analytics-crash-reports-and-more/>

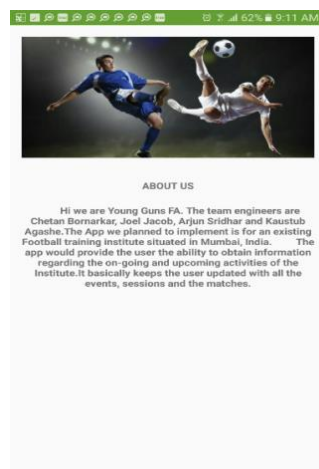
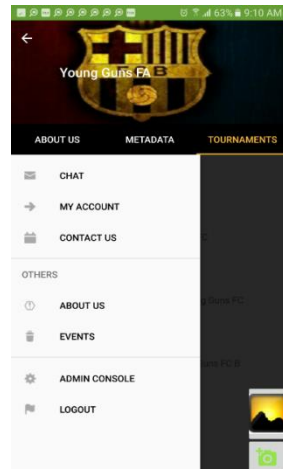
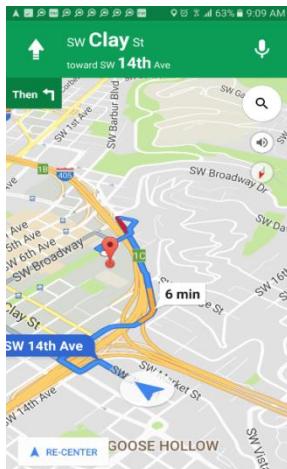
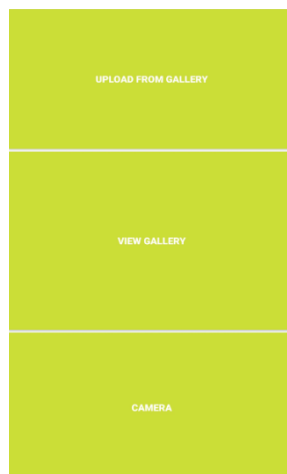
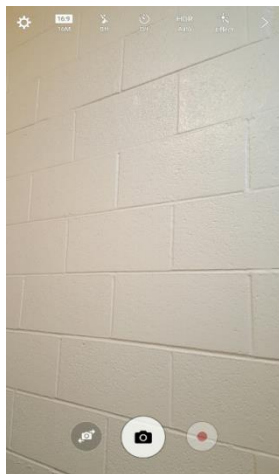
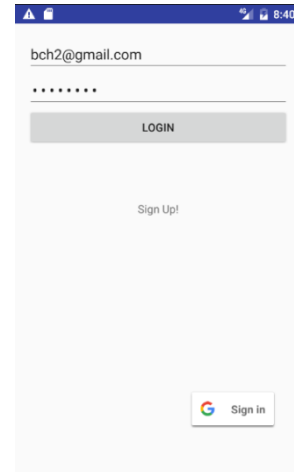
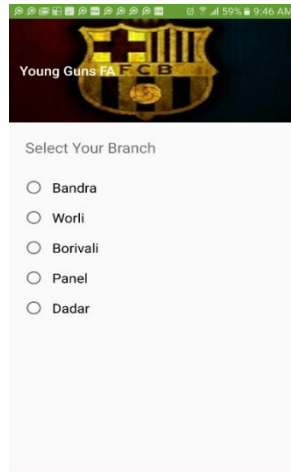
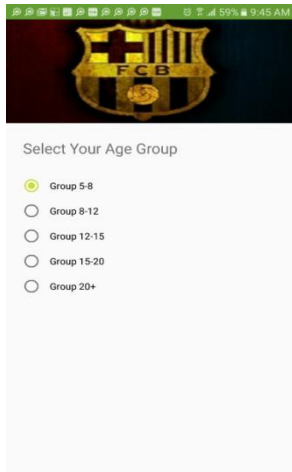
DESIGN FLOW:

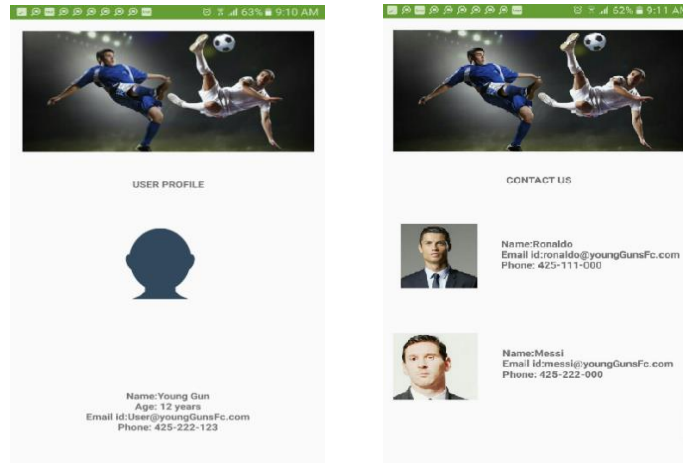
1. When the app starts the first time, the login/Sign in activity screen is shown, requesting the user to enter their credentials. It also has a Google sign in button which allows the user to sign in with their Google. If the user does not have an account, then they have the option of creating a new account using the Sign-Up button. If the user presses the sign-up button, then go to **step 2** else go to **step 5**.
2. **Sign Up Button:** The user can create a **new** account onto the online database by entering their email account and password.
3. After they enter the email Id and password, they are asked to choose their branch where they wish to go **train and practice**. They are also asked for their **age group**; this information is used to extract specific events and match schedules for that particular age group from the **database**.
4. The user is then shown the login page where they have to enter the credentials to their newly created account.
5. Once the user has their account setup, they are shown a slider tab view consisting of "Contact Us", "Metadata", "Tournaments". This information is extracted from the database.
6. On the tournament tab the users can see their next match, a description of the match and its location.
7. The user can **tap** the event to navigate to the training location, a **turn-by-turn Google maps navigation activity** shows up and directs the user to their destination.
8. There is a **camera** button on the main screen which when pressed allows the user to capture images from the phones camera. This image is stored in the locally on the phones photo Gallery. If the user wishes, they can then **upload** the images to the cloud and other app users will be able to view them on their photo galleries.
9. The button "**Upload from Gallery**" opens up the phone gallery and when the user selects an image, it uploaded to the online Firebase Storage space. The "**View Gallery**" button populates the images from the cloud in a photo gallery grid.
10. A user changes the details of the events from the app **console** database in firebase, the changes will **propagate** to **everyone** using the app. **Notifications** can also be sent in similar fashion.

BLOCK DIAGRAM:

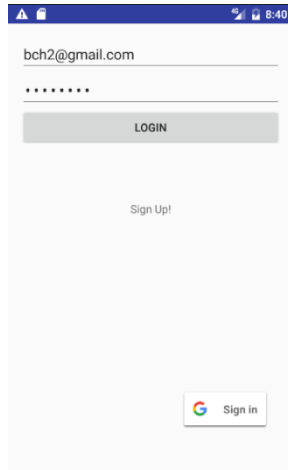


APP UI:

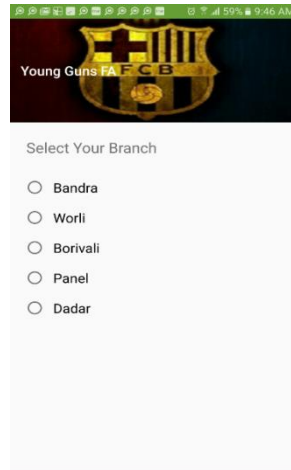




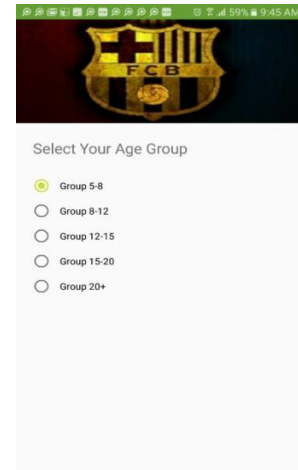
SIGN UP AND SIGN IN ACTIVITY:



(1st: Login Page)



(2nd: If user selects sign up

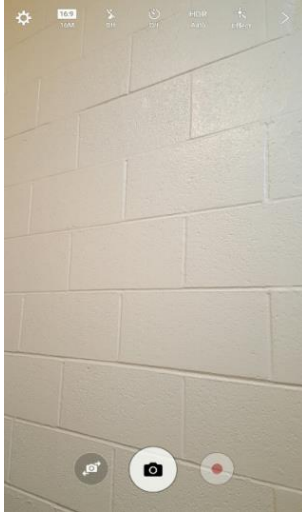


(3rd: Select the age group)

Then he has to select branch)

The sign up/sign in activity is the first activity seen when the app is run. If the user has a google account they can use it to login into the application. If the user does not have a google account then we have a feature to create an account onto the online database. The user can do so by pressing the sign up button after which the app will allow them to enter their desired email address and password. After that they will be prompted to select the branch in which they wish to go to and the age group they belongs to. Once all the data is entered into the database the app goes back to the login page waiting for the user to enter their credentials.

THE CAMERA ACTIVITY:



(Camera View)

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode == REQUEST_TAKE_PHOTO && resultCode == RESULT_OK) {
        // Show the thumbnail on ImageView
        Uri imageUri = Uri.parse(mCurrentPhotoPath);
        File file = new File(imageUri.getPath());
        try {
            InputStream ims = new FileInputStream(file);
            ivPreview.setImageBitmap(BitmapFactory.decodeStream(ims));
        } catch (FileNotFoundException e) {
            return;
        }

        // ScanFile so it will be appeared on Gallery
        MediaScannerConnection.scanFile(Camera.this,
            new String[]{imageUri.getPath()}, null,
            new MediaScannerConnection.OnScanCompletedListener() {
                public void onScanCompleted(String path, Uri uri) {
                }
            });
    }
}
```

```
private File createImageFile() throws IOException {
    // Create an image file name
    String timeStamp = new SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date());
    String imageFileName = "JPEG_" + timeStamp + "_";
    File storageDir = new File(Environment.getExternalStoragePublicDirectory(
        Environment.DIRECTORY_DCIM), "Camera");
    File image = File.createTempFile(
        imageFileName, /* prefix */
        ".jpg", /* suffix */
        storageDir /* directory */
    );

    // Save a file: path for use with ACTION_VIEW intents
    mCurrentPhotoPath = "file:" + image.getAbsolutePath();
    return image;
}

private void dispatchTakePictureIntent() throws IOException {
    Intent takePictureIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    // Ensure that there's a camera activity to handle the intent
    if (takePictureIntent.resolveActivity(getPackageManager()) != null) {
        Uri photoURI = FileProvider.getUriForFile(Camera.this, BuildConfig.APPLICATION_ID + ".provider", createImageFile());
        takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT, photoURI);
        startActivityForResult(takePictureIntent, REQUEST_TAKE_PHOTO);
    }
}
```

The camera is invoked using the android inbuilt functions. Using the app's camera, the user can take pictures and preview them. The Uri of the image is used in all the app features related to images. Our camera implementation takes care of adding the images captured to the phone gallery, we use **MediaScannerConnection.scanFile()** for this purpose. The method **dispatchTakePictureIntent()** is used to invoke the camera and take photos. The **createImageFile()** method is used to name the image, create a File with the image name, format and the path and it returns the file.

PHOTO GALLERY:



(Gallery View)

```
ArrayList<ImageModel> data = new ArrayList<>();
ArrayList<String> IMGS = ViewActivity.getUrlList();

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_gallery);

    for (int i = 0; i < IMGS.size(); i++) {
        ImageModel imageModel = new ImageModel();
        imageModel.setName("Image " + i);
        imageModel.setUrl(IMGS.get(i));
        data.add(imageModel);
    }
}
```

```
@Override
public void onBindViewHolder(RecyclerView.ViewHolder holder, int position) {

    Glide.with(context).load(data.get(position).getUrl())
        .thumbnail(0.5f)
        .override(200,200)
        .crossFade()
        .diskCacheStrategy(DiskCacheStrategy.ALL)
        .into((MyItemHolder) holder).mImg;
}
```

We were able to utilize a gallery viewer application created by an app developer named **Suleiman** that was open sourced on **GitHub**. It makes use of the **Glide** framework which is an **image loading and caching** library for Android focused on **smooth scrolling**.

The photos that were uploaded to Firebase online storage had their URL's saved in the Real-time database. We query the database when the main activity initiates and load an array with the URL's of the images we desire to display. We then pass that array to the Glide API which in turns calls its **onBindViewHolder()** method to display the grid of scrollable images.

We have the option to cache the images into internal memory and also modify how big the picture thumbnail should be to help save on system resources.

Sources:

- <https://github.com/Suleiman19/Gallery>
- <https://github.com/bumpstech/glide>

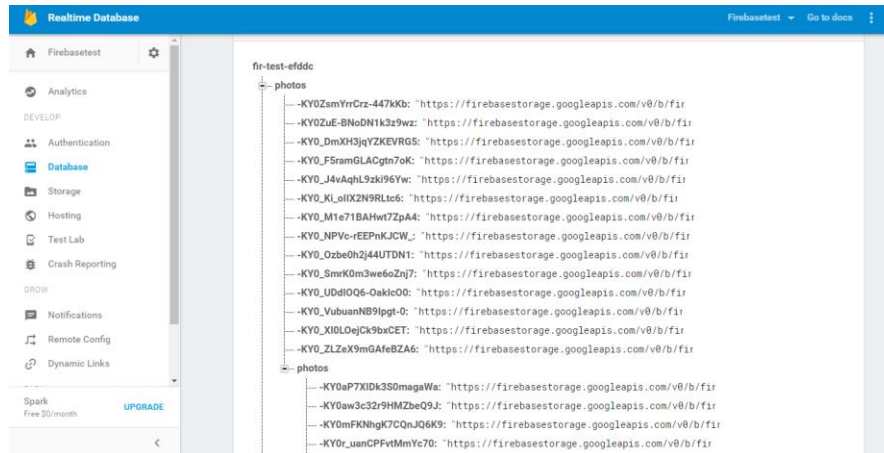
DOWNLOADING IMAGES VIA THE REAL-TIME DATABASE:

```
public void DownloadUpdate() {
    mStorageReference = FirebaseStorage.getInstance().getReference();
    mData = FirebaseDatabase.getInstance();
    mDatabase = mData.getReference();

    mDatabase.child("photos").addChildEventListener(new ChildEventListener() {
        @Override
        public void onChildAdded(DataSnapshot dataSnapshot, String s) {
            urlList.add((String) dataSnapshot.getValue());
        }

        @Override
        public void onChildChanged(DataSnapshot dataSnapshot, String s) {
        }
    });
}
```

Accessing the Firebase Database is done through a **"parent-child"** hierarchy. We reference the online database and traverse the database until we reach the **"photos"** child. We setup the **onChildAdded()** listener to monitor any changes made to the database in real-time



(This screenshot displays the online **FireBase DataBase** via the **console** where user management and other Administration tasks can be accomplished)

If a new **URL string** is added into the database via the **upload activity** or through the **online console**, the listener will add that new entry into the **urlList** array and that image will be populated in the Photo Gallery.

UPLOAD PHOTOS TO FIREBASE STORAGE:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == REQUEST_TAKE_PHOTO && resultCode == RESULT_OK) {
        try {
            Bitmap help1 = MediaStore.Images.Media.getBitmap(getContentResolver(), mfileURI);

            mProgressDialog.setMessage("Uploading.....!");
            mProgressDialog.show();
            MediaStore.Images.Media.insertImage(getContentResolver(), help1, "", "");

            StorageReference filepath = mStorageReference.child("photos").child(mfileURI.getLastPathSegment());

            filepath.putFile(mfileURI).addOnSuccessListener(new OnSuccessListener<UploadTask.TaskSnapshot>() {
                @Override
                public void onSuccess(UploadTask.TaskSnapshot taskSnapshot) {

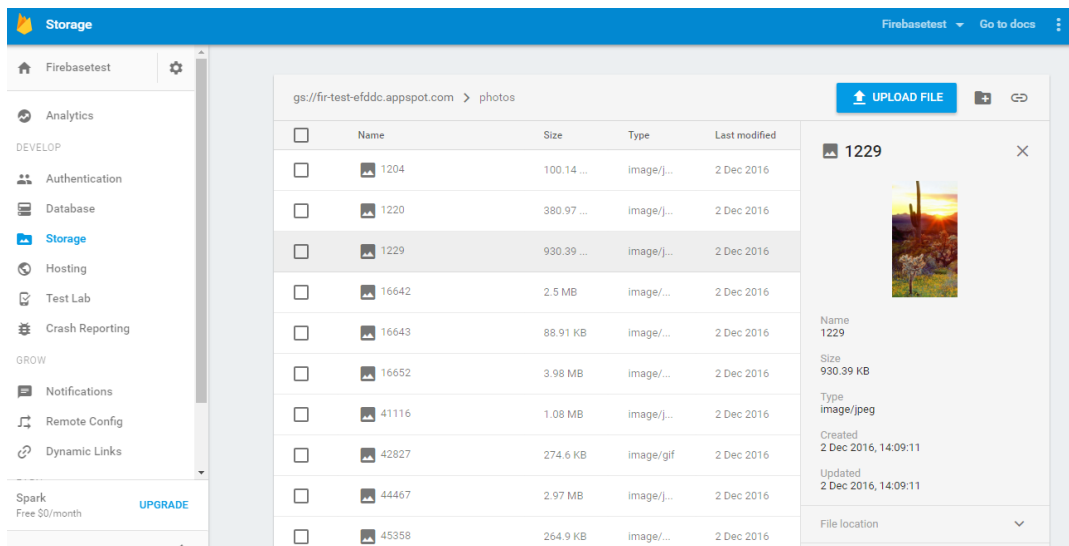
                    Uri myUri = taskSnapshot.getDownloadUrl();
                    String downloadurl = myUri.toString();

                    mDatabase.child("photos").push().setValue(downloadurl);

                    Toast.makeText(PhotoUpload.this, "Upload done", Toast.LENGTH_LONG).show();
                    mProgressDialog.dismiss();

                }
            });
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

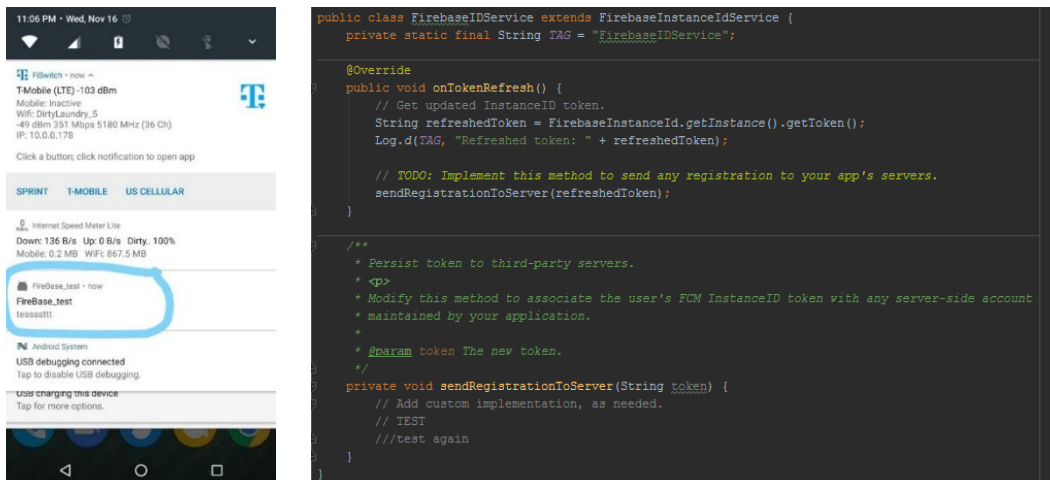
Similar to the database access explained above, to upload a photo the user picks from the phone gallery, we traverse to the **"photos"** child reference and upload the photo using the **.putFile()** method call.



Photos uploaded to Google FireBase Storage

We then download the URL of the file and save it to the database using the **.push().setValue** method. The URL now resides on the database to be accessed by any activity that may require it in the future.

NOTIFICATIONS (BACKGROUND SERVICE)



(Test Notification)

FireBase allows us to send notifications to users using their API via the online console.

It does this by creating a service **FirebaseIDService**, that runs in background after the app is installed.

```

@Override
public void onMessageReceived(RemoteMessage remoteMessage) {
    // ...

    // TODO(developer): Handle FCM messages here.
    // Not getting messages here? See why this may be: https://goo.gl/39bRNJ
    Log.d(TAG, "From: " + remoteMessage.getFrom());

    // Check if message contains a data payload.
    if (remoteMessage.getData().size() > 0) {
        Log.d(TAG, "Message data payload: " + remoteMessage.getData());
    }

    // Check if message contains a notification payload.
    if (remoteMessage.getNotification() != null) {
        Log.d(TAG, "Message Notification Body: " + remoteMessage.getNotification().getBody());
    }

    mTitle.add(remoteMessage.getNotification().getTitle());
    mBody.add(remoteMessage.getNotification().getBody());
    sendNotification(remoteMessage.getNotification().getTitle(), remoteMessage.getNotification().getBody());
}

//This method is only generating push notification
private void sendNotification(String title, String messageBody) {
    Intent intent = new Intent(this, MainActivity.class);
    intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
    PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, intent,
        PendingIntent.FLAG_ONE_SHOT);

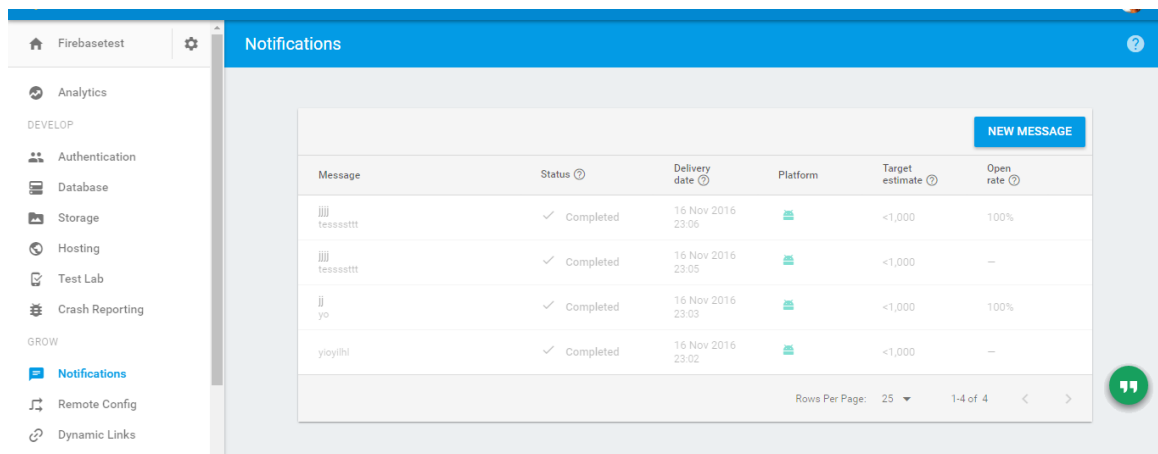
    Uri defaultSoundUri= RingtoneManager.getDefaultUri(RingtoneManager.TYPE_NOTIFICATION);
    NotificationCompat.Builder notificationBuilder = new NotificationCompat.Builder(this)
        .setSmallIcon(R.mipmap.ic_launcher)
        .setContentTitle(title)
        .setContentText(messageBody)
        .setAutoCancel(true)
        .setSound(defaultSoundUri)
        .setContentIntent(pendingIntent);

    NotificationManager notificationManager =
        (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);

    notificationManager.notify(0, notificationBuilder.build());
}

```

When a notification is sent via the online console, the **onMessageReceived()** activity is called and it creates the notification on the notification bar. If the app is running in foreground, the **sendNotification()** method is called. The user receives a notification sound and the phone vibrates. If the app is killed and not running, the user will receive a silent notification. If they click on the notification it will open up the app and launch its main activity.



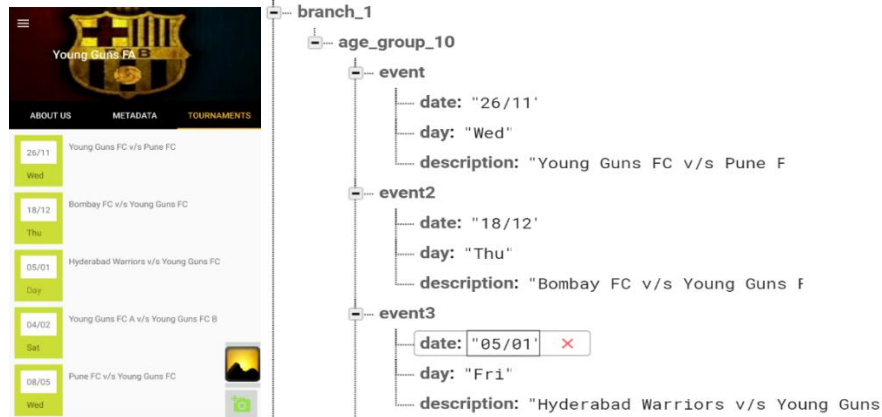
The screenshot shows the Firebase console interface with the 'Notifications' tab selected. A table lists four messages, all with a status of 'Completed' and a delivery date of 16 Nov 2016. The messages are from 'fesssttt', 'fesssttt', 'jiyo', and 'yioyihl'. The table includes columns for Message, Status, Delivery date, Platform, Target estimate, and Open rate. A 'NEW MESSAGE' button is visible in the top right corner of the table area.

| Message | Status | Delivery date | Platform | Target estimate | Open rate |
|----------|-------------|-------------------|----------|-----------------|-----------|
| fesssttt | ✓ Completed | 16 Nov 2016 23:06 | Android | <1,000 | 100% |
| fesssttt | ✓ Completed | 16 Nov 2016 23:05 | Android | <1,000 | — |
| jiyo | ✓ Completed | 16 Nov 2016 23:03 | Android | <1,000 | 100% |
| yioyihl | ✓ Completed | 16 Nov 2016 23:02 | Android | <1,000 | — |

*The console also allows us to send system **notifications** to users even when the app is running in the background*

A new message can be created via the console and broadcasted as a notification to every user who has the app installed on their phone.

THE CALENDAR INTERFACED WITH THE DATABASE:



(The screenshot above shows the events and matches schedule, which is read from the database shown in the figure to right)

```
public CalViewHolder(View itemView) {
    super(itemView);
    mDateTextView = (TextView) itemView.findViewById(R.id.tvCalendarDateId);
    mDayTextView = (TextView) itemView.findViewById(R.id.tvCalendarDay);
    mEventTextView = (TextView) itemView.findViewById(R.id.tvCalendarEventId);
}

public void bindCalEvent(CalendarEventsDataTableParm event) {
    mDateTextView.setText(event.getDate().toString());
    mDayTextView.setText(event.getDay());
    mEventTextView.setText(event.getEventDetails());
}
```

(The code snippet above shows how the event data is populated in the view)

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
    View view = inflater.inflate(R.layout.calendar_event_container, container, false);
    mCalEventRecyclerView = (RecyclerView) view.findViewById(R.id.calendar_event_container);
    mCalEventRecyclerView.setLayoutManager(new LinearLayoutManager(getActivity()));

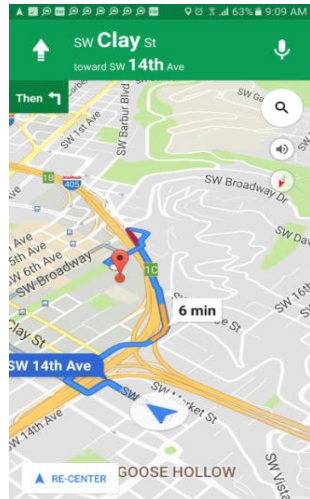
    updateUI();
    return view;
}

private void updateUI()
{
    CalendarEventDate_Db mCalEventsDb = CalendarEventDate_Db.get();
    mCalendarEventsDataTableParms = CalendarEventDate_Db.get().getCalendarEvents();
    if(mCalEventAdapter == null)
    {
        mCalEventAdapter = new CalEventAdapter(mCalendarEventsDataTableParms);
        mCalEventRecyclerView.setAdapter(mCalEventAdapter);
    }
    else
        mCalEventAdapter.notifyDataSetChanged();
}
```

(The snippet above shows that when the tournament view is inflated it reads the data from the database)

The calendar event reads the events, training and matches schedule from the firebase database and populates the recycler view. The above image shows the tournament events/matches as it is stored in the database. This is a fragment which is a part of the view pager activity.

TURN-BY-TURN NAVIGATION USING GOOGLE MAPS:



```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_maps);
    // Obtain the SupportMapFragment and get notified when the map is ready to be used.
    SupportMapFragment mapFragment = (SupportMapFragment) getSupportFragmentManager()
        .findFragmentById(R.id.map);
    mapFragment.getMapAsync(this);

    String location = "google.navigation:q=Peter+Stott+Portland+Oregon";

    Uri gmmIntentUri = Uri.parse(location);
    Intent mapIntent = new Intent(Intent.ACTION_VIEW, gmmIntentUri);
    mapIntent.setPackage("com.google.android.apps.maps");
    startActivity(mapIntent);
}

/**
 * Manipulates the map once available.
 * This callback is triggered when the map is ready to be used.
 * This is where we can add markers or lines, add listeners or move the camera. In this case,
 * we just add a marker near Sydney, Australia.
 * If Google Play services is not installed on the device, the user will be prompted to install
 * it inside the SupportMapFragment. This method will only be triggered once the user has
 * installed Google Play services and returned to the app.
 */

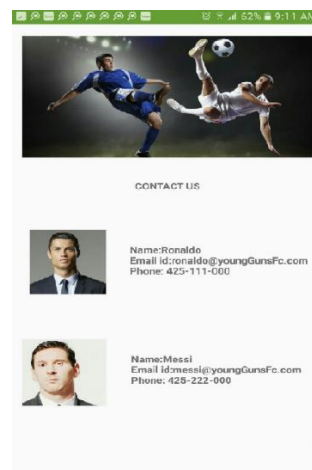
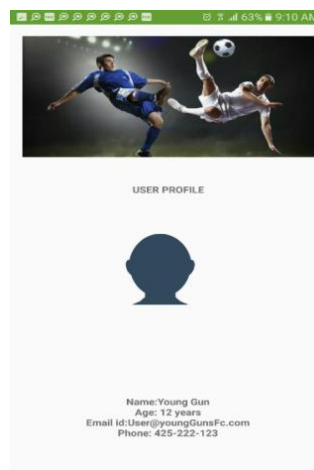
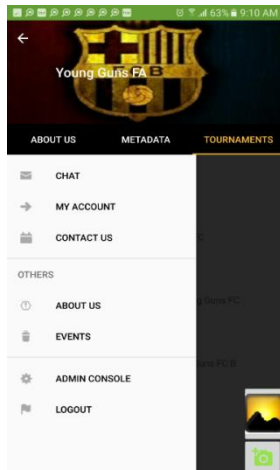
@Override
public void onMapReady(GoogleMap googleMap) {
    mMap = googleMap;

    // Add a marker in Sydney and move the camera
    LatLng sydney = new LatLng(45, -122);
    mMap.addMarker(new MarkerOptions().position(sydney).title("Marker in Portland"));
    mMap.moveCamera(CameraUpdateFactory.newLatLng(sydney));
}
```

We utilize the **MAPS API** to implement turn-by-turn navigation within the Calendar event UI fragment in the app. When we initiate the activity, we send in a search location (“Peter Stott Football Ground”) as an intent to the Maps API. The Google Maps application opens up and automatically navigates to the parceled location.

When we hit the back key, the app returns to the main activity, right where we left off. We need to setup a Maps API Key and store it in an .xml file within the Android resource folder for proper functioning.

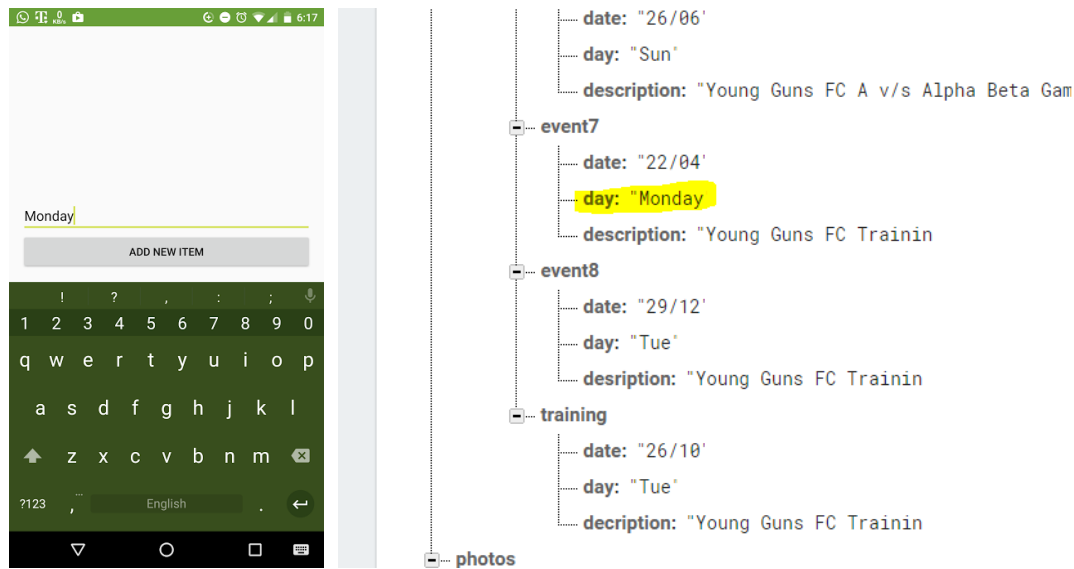
MISCELLANEOUS ACTIVITIES:



(The screenshots above show the miscellaneous activities which are seen from the navigation view. The one to right is the “Contact us” and the other is “User Profile” activity)

The miscellaneous activities are inflated when the user presses their respective button in the navigation view which consists of My Account Info, Contact Us, User Profile, LogOut, Administrative console, Events, About Us.

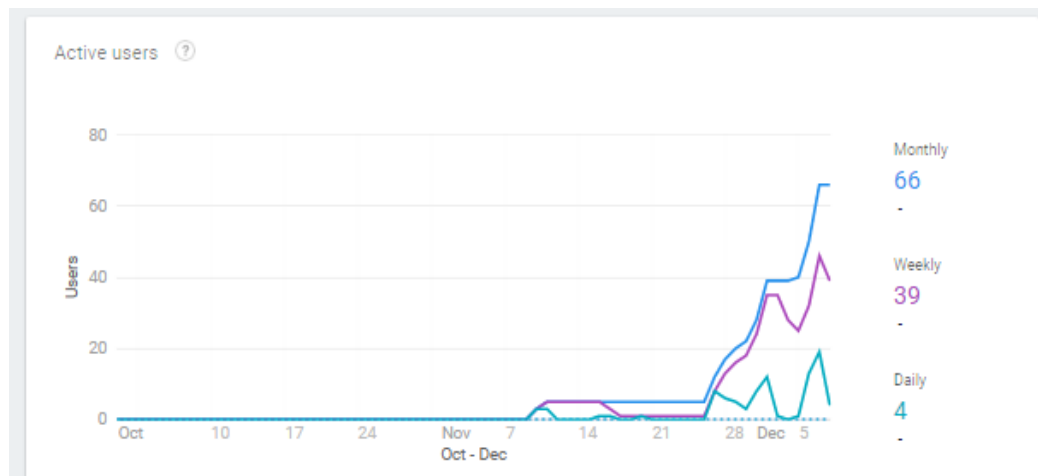
ADMIN CONSOLE PROTOTYPE (EXTENDED FEATURE):



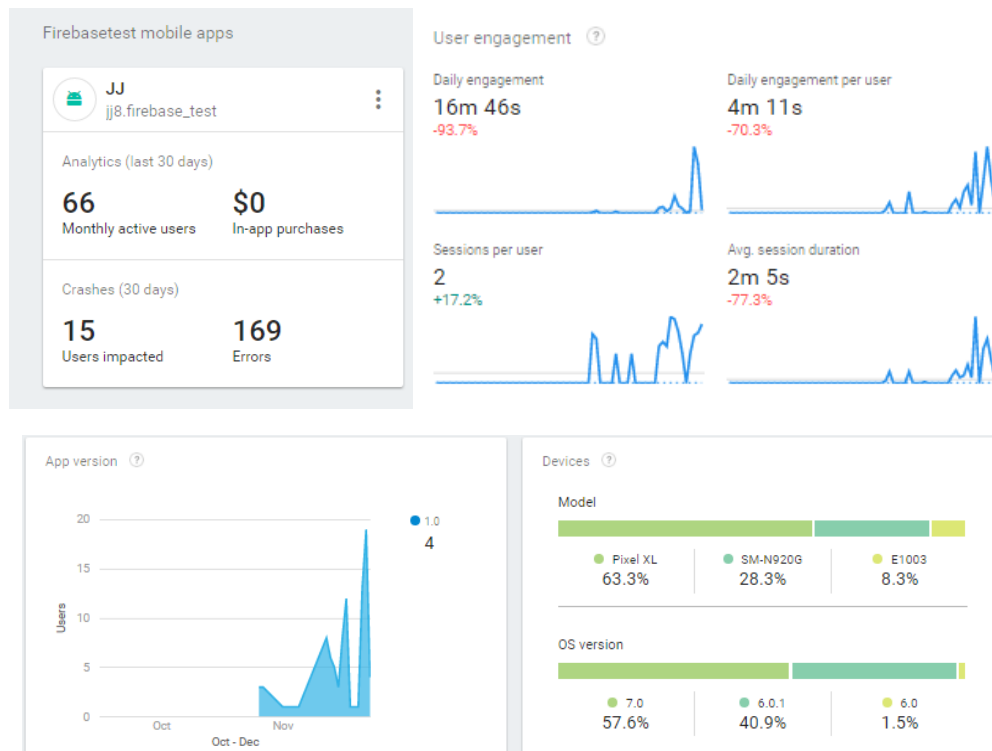
We were able to implement an additional prototype feature that allows app administrators to modify the online database via the app itself. The admin need not go to browser console to update the database. If a change is made to the database via the app admin panel, it will propagate to every user who has the app installed in real-time.

APP ANALYTICS AND CRASH REPORTING (EXTENDED FEATURES):

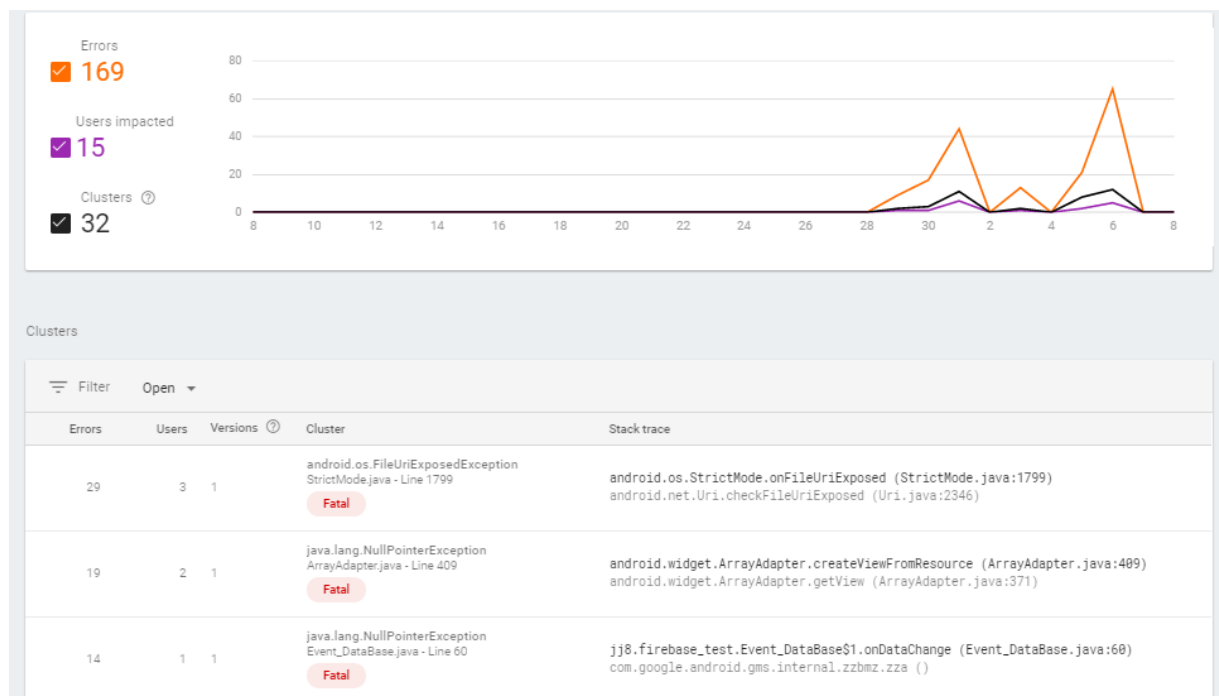
FireBase allowed us to obtain useful analytics based on app usage, some screenshots attached below.



This graph aptly represents the number of user over time



Active users, Device and OS details and some engagement data.



App crashes were documented and this data proves useful in debugging and isolating recurring issues.

BUILD.GRADLE (APP LEVEL)

```
//
compile 'com.android.support:appcompat-v7:25.0.1'
compile 'com.google.firebase:firebase-auth:10.0.1'
compile 'com.google.android.gms:play-services-auth:10.0.1'
compile 'com.google.firebase:firebase-core:10.0.1'
compile 'com.google.firebase:firebase-messaging:10.0.1'
compile 'com.google.android.gms:play-services-location:10.0.1'
compile 'com.google.android.gms:play-services-places:10.0.1'
compile 'com.google.firebase:firebase-database:10.0.1'
compile 'com.google.android.gms:play-services:10.0.1'
compile 'com.github.bumptech.glide:glide:3.7.0'
compile 'com.android.support:support-v4:25.0.1'
compile 'com.android.support:design:25.0.1'
compile 'com.google.firebase:firebase-storage:10.0.1'
compile 'com.android.support:recyclerview-v7:25.0.1'

compile 'com.google.firebase:firebase-auth:10.0.1'
compile 'com.google.android.gms:play-services-auth:10.0.1'
compile 'com.google.firebase:firebase-core:10.0.1'
compile 'com.android.support:design:25.0.1'
compile 'com.google.firebase:firebase-messaging:10.0.1'
compile 'com.github.alamkanak:android-week-view:1.2.6'

compile 'com.github.hotchemi:permissionsdispatcher:2.2.0'
annotationProcessor 'com.github.hotchemi:permissionsdispatcher-processor:2.2.0'

testCompile 'junit:junit:4.12'
// code added from google calendar api tutorials
compile 'pub.devrel:easypermissions:0.1.5'
compile ('com.google.api-client:google-api-client-android:1.22.0') {
    exclude group: 'org.apache.httpcomponents'
}
compile ('com.google.apis:google-api-services-calendar:v3-rev225-1.22.0') {
    exclude group: 'org.apache.httpcomponents'
}
```

We had to compile quite a few extra libraries and a few repositories to allow all the connected components to function, attached is a screen capture of the main libraries that must be compiled for this app to function.

WORK DISTRIBUTION:

The four of us worked on the various app modules individually.

- Chetan worked on the Event Scheduler which connected with the real-time database. Created a tabbed activity using fragments and view pagers and overall app integration.
- Kaustubh worked on the app design and layout. (Tabbed activities and a swipe able interface)
- Arjun got the Camera working and was able to upload/download to the app and save images to the gallery via the Database and Firebase Storage.
- Joel worked on the Firebase features like the real-time Database, Sign in/Sign up activities with Google Authentication, Notifications, Maps API, and setting up the photo gallery to view downloaded images and overall app integration.

RESULTS AND CHALLENGES FACED:

We were able to deliver on our proposal and were able to implement the features we had hoped to see working in the application. 😊

That being said, we faced quite a few hurdles on the way,

- Creating an entry in the database and retrieving corresponding data from firebase storage required a little research.
- Retrieving images from the firebase database required a URL which had to be generated. This task was challenging because we had to convert the URL to URI to read the image.
- Ideally the creation of image gallery should have happened in the tabbed activity. Since that required converting multiple activities into fragments, for lack of time, we decided to create a new activity instead. With time, we would be able to implement the activity in fragments.
- We had to make sure the database was not called multiples times, otherwise the gallery would duplicate images, this required us instantiating the database in the main activity.
- SHA1 keys for debug and release version of the app proved to be a little difficult to obtain.
- Permissions for hardware had to be manually granted for post Marshmallow devices (Android 6.0, SDK 23). This is because permission management has changed in newer Android versions. This is a relatively easy fix and can be implemented without much work.

FUTURE WORK:

- Some activities could be converted to fragments thus making it work on bigger phones and tablets.
- The current database has only one age group and one branch. We can create new age groups and different branches in the database.
- We can create a messaging function quite easily. The backend has already been set up because of the notifications system we have implemented.
- Modifying the camera to capture and store videos is extremely simple. We were already able to capture and save videos to the phone gallery during testing.

REFERENCES:

- <https://firebase.google.com/>
- <https://firebase.google.com/features/>
- <https://techcrunch.com/2016/11/07/googles-firebase-developer-platform-gets-better-analytics-crash-reports-and-more/>
- <https://github.com/Suleiman19/Gallery>
- <https://github.com/bumptech/glide>
- <https://firebase.google.com/docs/android/setup>
- <https://www.sitepoint.com/creating-a-cloud-backend-for-your-android-app-using-firebase/>
- <https://github.com/sitepoint-editors/Firebase-with-Android>
- <https://www.simplifiedcoding.net/android-push-notification-tutorial-using-firebase/>
- <http://stackoverflow.com/>