P.PORTO

ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

NORTE2020

PORTUGAL 2020

UNIÃO EUROPEIA

Fundo Social Europeu

Trabalho Prático

Programação Dispositivos Móveis

Hugo Silva-8180378

Marco Carneiro-8180382

Conteúdo

Introdução

Este documento foi realizado para descrever o trabalho elaborado para a disciplina de Programação para Dispositivos Moveis e tem como objetivo demonstrar toda a elaboração do nosso projecto. Ao longo do relatório irá ser apresentada uma visão do produto e análise do problema, os requisitos do produto bem como detalhes de implementação e funcionalidades obrigatórias e de bonificação. Por fim será feita uma conclusão à cerca do projecto.

Abreviações:

RF-> Requisitos Funcionais;

RNF-> Requisitos não Funcionais;

Visão do Produto e análise do problema

O produto que foi proposto desenvolver é uma aplicação em android que permita registar as atividades realizadas por um utilizador. A aplicação deve permitir ao utilizador saber a sua localização atual, deve permitir ao utilizador ter acesso a dados meteorológicos sobre a localização onde se encontra (através da API openweathermaps) deve oferecer ao utilizador um histórico das suas atividades bem como a localização em que decorreram. O produto deve também oferecer a possibilidade de ver estatísticas importantes numa atividade como por exemplo a duração e o número de passos e a distância percorrida.

Requisitos do Produto

RF1->A aplicação deve permitir registo de contas e login;

RF2-> A aplicação deve pedir permissões para aceder à localização do dispositivo, internet e sensores;

RF3->Deve ser permitido guardar dados das atividades na base de dados local do dispositivo (Room);

RF4-> Deve ser possível fazer pedidos via REST (Retrofit);

RF5-> A aplicação deve permitir escolher o tipo de atividade (Ex: corrida, caminhada);

RF6->A aplicação deve verificar a velocidade a que decorre a atividade (deve pausar o tempo da atividade quando a velocidade for superior aos limites, por exemplo se o utilizador vai no carro não deve ser contabilizado como tempo de atividade);

RF7-> A aplicação deve oferecer o histórico de atividades ao utilizador através de uma RecyclerView;

RF8-> A aplicação deve contar o número de passos que o utilizador realiza durante a atividade;

RF9-> A aplicação deve registar a hora de início e de fim de cada atividade bem como a data;

RF10-> A aplicação deve guardar na Room as coordenadas num espaço de tempo predefinido (por exemplo de 1 em minutos);

RF11-> A aplicação deve enviar notificações;

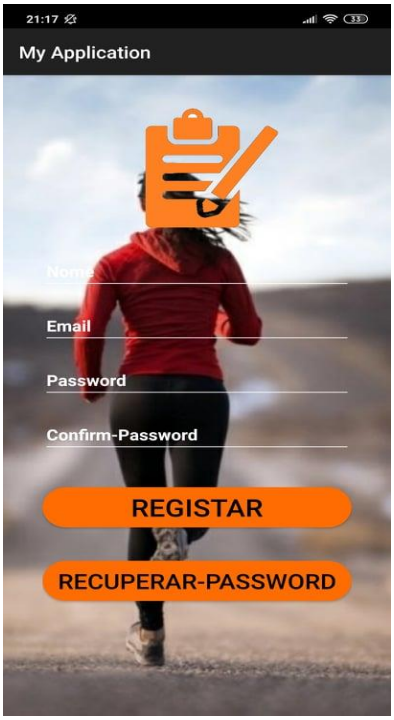
RNF1-> A aplicação deve apresentar um layout simples e intuitivo;

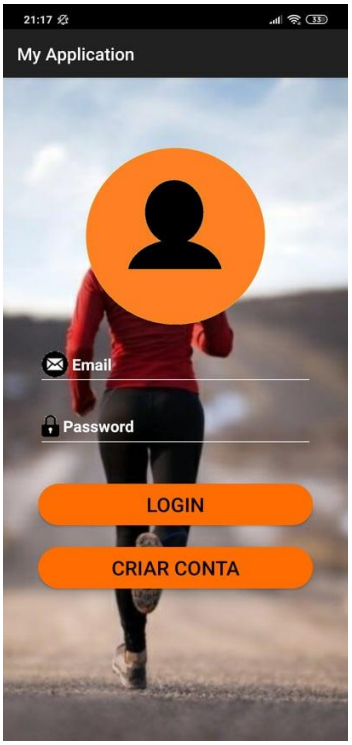
RNF2-> A aplicação deve poder ser utilizada em qualquer circunstância.

Login e Registo

A funcionalidade de registo e login foi feita com recurso a uma biblioteca de suporte (Firebase) para isso a primeira coisa que foi realizado foi a conexão do nosso projecto com o Firebase, realizado este passo seguimos a documentação que está nas referências deste relatório. Para o registo é pedido o nome, email e password. Para efetuar login o utilizador tem de colocar o email e a sua password. Para esta funcionalidade adicionamos a permissão para a app poder usar a internet.

Na imagem acima vemos a tabela de autenticação do nosso projecto no Firebase.





Nas duas imagens em cima representam a interface de registo e login respetivamente.

Sensor para contagem de passos

O que se seguiu na elaboração do nosso projecto foi a contagem de passos dados pelo utilizador. A primeira coisa foi o estudo de que forma seria implementado o sensor. Depois do estudo decidimos implementar o sensor através de uma AsyncTask, esta decisão é justificada com o facto de caso o utilizador ponha o telemóvel em modo suspensão ou utilize outra aplicação os passos sejam contados na mesma em background.

```
public class InitActivityTask extends AsyncTask<Void, Atividade, Atividade> implements SensorEventListener {
```

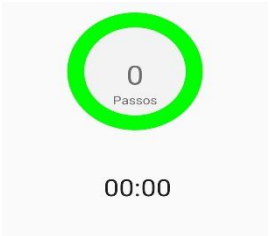
Na nossa AsyncTask implementamos o SensorEventListener e de seguida instanciamos e registamos o nosso sensor no construtor da nossa classe.

```
this.model=model;
mSensorManager = (SensorManager) mContext.getSystemService(mContext.SENSOR_SERVICE);
mStep = mSensorManager.getDefaultSensor(Sensor.TYPE_STEP_DETECTOR);
mSensorManager.registerListener((SensorEventListener) this, mStep, SensorManager.SENSOR_DELAY_NORMAL);
```

Com a implementação do SensorEventListener obtemos o método onSensorChanged que regista qualquer alteração no nosso sensor. Como o nosso objeto é contar passos incrementamos uma variável que sempre que o sensor recebe alterações (sempre que é dado um passo) aumenta 1 valor.

E também atualizada a view que mostra ao utilizador o número de passos atuais.

```
@Override
public void onSensorChanged(SensorEvent event) {
    passos++;
    view4.setText(" " + passos);
}
```




```
<!-- Sensores -->
<uses-feature
    android:name="android.hardware.sensor.stepdetector"
    android:required="true" />
<uses-feature
    android:name="android.hardware.sensor.stepcounter"
    android:required="true" />
```

Uses-feature para utilização do sensor para detetar passos.

Pedidos à API(Openweathermaps)

O primeiro passo foi a obtenção da chave para podermos utilizar a API, de seguida implementamos as dependências do Retrofit no nosso projecto.

Com as dependências implementadas criamos a interface responsável por fazer o pedido GET à API, este método recebe como parâmetro a latitude e longitude bem com a chave para a API.

```
public interface OpenweatherApi {

    @GET("data/2.5/weather?")
    Call<WeatherResponse> getCurrentWeatherData(@Query("lat") String lat, @Query("lon") String lon, @Query("APPID") String app_id);

}
```

De seguida criamos o modelo de dados que permitirá converter o gson recebido da API em objetos java. Para isso fomos à documentação da API ver a resposta recebida e à documentação da API para ver que classes e atributos que devíamos criar para que fosse possível a conversão.

```
public class Main {

    public float temp;

    public float humidity;

    public float pressure;

    public float temp_min;

    public float temp_max;


}
```

```
public class Sys {

    public String country;

    public long sunrise;

    public long sunset;
}
```

Em cima nas duas imagens vemos dois exemplos das classes que criamos para a conversão.

De seguida na Atividade que suporta o fragmento em que são apresentados os dados criamos a instância da API.

```
//Instancia para chamada á api
private Retrofit getRetrofit() {
    return new Retrofit.Builder().baseUrl("http://api.openweathermap.org/")
            .addConverterFactory(GsonConverterFactory.create()).build();
}
```

Em seguida criamos o método para executar o pedido assíncrono, método este que recebe como parâmetro as TextViews, que irão mostrar os dados recebidos, e recebe também as coordenadas de onde se encontra o

dispositivo para que no pedido à API seja feito de acordo com a localização do utilizador.

```
@Override
public void getDetails(final TextView view, final TextView view2, final TextView view3, final ImageView img) {
    getApi().getCurrentWeatherData(lan, lon, key)
            .enqueue(new Callback<WeatherResponse>() {
                @Override
                public void onResponse(Call<WeatherResponse> call, Response<WeatherResponse> response) {
                    //Resposta da api
                    points = response.body();
                    // Saca imagem do tempo da api
                    String baseurl = "http://openweathermap.org/img/wn/";
                    String temp = null;
                    TarefaDwoload download = new TarefaDwoload(img,  context: User_home.this);
                    temp = baseurl + points.weather.get(0).icon.toString() + "@2x.png";
                    download.execute(temp);

                    view.setText(points.name);
                    double newvalue = (points.main.temp - 273.15);
                    String resultado = String.format("%.0f", newvalue);
                    view2.setText((" " + resultado + " C°"));
                    view3.setText("" + points.weather.get(0).description);

                }

                @Override
                public void onFailure(Call<WeatherResponse> call, Throwable t) {
                    view.setText(t.toString());

                }

            });
}
```

Durante esta chamada é também executado numa AsyncTask o download do icon referente ao tempo (Sol, nuvem, nuvem com chuva), para isso é passo a ImageView que vai ser atualizada com o icon e o link para que o download seja executado.

```
public class Auxiliar {
    public  static Bitmap baixarImagem(String endereco) throws IOException {
        // URL endereco;
        InputStream inputStream = null;
        Bitmap imagem;

        URL url = new URL(endereco);
        HttpURLConnection conexao = (HttpURLConnection) url.openConnection();
        InputStream input = conexao.getInputStream();
        imagem = BitmapFactory.decodeStream(input);
        return imagem;
    }
}
```

A imagem em cima demonstra a classe que executa o download do icon do tempo da API.

```
click.getDetails(text, text2, text3, img);
```

Neste trecho de código é onde o Fragment faz o pedido assíncrono à API, para a realização do pedido criamos uma interface que permite que o Fragment execute um método que se encontra na Atividade.

```
public interface Ges {
    void getDetails(TextView view,TextView view2,TextView view3, ImageView img);
}
```

A imagem em cima demonstra o código da interface que cria o método que é executado (na primeira imagem desta página) no Fragment.

Esta imagem demonstra o layout final da chamada à API em que mostra o nome do local onde nos encontramos, a temperatura e uma descrição do tempo.

Criação da Tabela Atividade (Room)

A primeira coisa realizada foi a implementação das dependências do Room no nosso projecto. Em seguida e com base nos exercícios realizados nas aulas criamos os vários componentes (Ex: Live Data, Entidades, DAO ViewModel e Base de dados) necessários para a criação da base de dados local no nosso telemóvel.

Entidade

Representa uma tabela na base de dados, para isso criamos uma entidade (Atividade) com os seguintes atributos:

```
@PrimaryKey(autoGenerate = true)
@NonNull
public int id;

private String id_user;
private String tipo;
private String data;
private int steps;
private String timestart;
private String timeend;
```

Dos atributos a salientar o "id_user" que recebe o id do utilizador do Firebase. Objetivo? Cada utilizador tem as suas atividades e caso haja um dispositivo utilizado por mais que uma pessoa então cada pessoa apenas deve ter acesso às suas atividades.

DAO (Data Access Objects)

Contêm os métodos para acesso à base de dados, nesta interface foram criados todas as queries necessárias quer para inserir, quer para fazer Update quer para mostrar todas as atividades (Histórico de Atividades). A implementação da interface foi feita pela biblioteca DAO

```
@Dao
public interface AtividadeDao {
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    void insereActivity(Atividade... atividades);

    @Delete
    void deleteActivity(Atividade... contacts);

    @Query("Select * from atividade  ")
    LiveData<List<Atividade>> getallAtividades(  );

    @Query("Select * from atividade WHERE id_user = :id_user and tipo=:tipo and timestart= :timestart  and steps= :steps")
    Atividade get( String id_user ,String tipo,String timestart, int steps );

    @Query("UPDATE atividade SET steps= :passos , timeend= :hora_fim, duracao=:duracao WHERE id= :id")
    void update( int passos ,String hora_fim,int duracao, int id);


}
```

Criação da Base de dados

Na imagem em baixo mostra a criação da base de dados e a respetivas tabelas.

```
@Database(entities = {Atividade.class,Coordenadas.class}, version = 1, exportSchema = false)
public abstract class ActivityDataBase extends RoomDatabase {
```

A imagem que se segue mostra o método em que a base de dados é instanciada.

```
public static ActivityDataBase getInstance(final Context context) {
    if (INSTANCE == null) {
        synchronized (ActivityDataBase.class) {
            if (INSTANCE == null) {
                INSTANCE = Room.databaseBuilder(
                        context.getApplicationContext(), ActivityDataBase.class,  name: "tell.db")
                        .addMigrations(MIGRATION_1_2)
                        .build();
            }
        }
    }
    return INSTANCE;
}
```

Criação do Repositório

A criação do Repositório teve como objetivo instanciar os métodos criados na DAO, aqui foi instanciada a interface DAO bem como os métodos que lhe pertencem.

```
public void insereAtividade(Atividade p) { new InsertAsync(atividadeDao).execute(p); }

public void delete(Atividade c) { new DeleteAsync(atividadeDao).execute(c); }
```

Estes métodos são executados em AsyncTask. Motivo? Operações sobre a base de dados não são permitidas na thread da UI!

Já o método de obter todas as atividades foi criado com o Live Data porque assim permite a atualização de componentes em tempo real usando o padrão de software Observable.

ViewModel

De seguida foi criado o ViewModel, os ViewModel são retidos automaticamente durante as mudanças de configuração, de modo que os

dados retidos estejam imediatamente disponíveis para a próxima atividade ou instância de fragmento, dado a utilidade do ViewModel resolvemos adicioná-lo ao nosso projecto.

```
public class ActivityViewModel extends AndroidViewModel {
    private LiveData<List<Atividade>> allAtivities;
    private ActivityRepositorio repositorio;
    private Atividade activity;
   private int ola;


    public ActivityViewModel(@NonNull Application application) {
        super(application);
        repositorio=new ActivityRepositorio(application);
        allAtivities = repositorio.getallAtividades();
    }
```

Este ViewModel tem como atributos uma Live Data para receber o conjunto de atividades e o repositório para fazer o acesso às queries da base de dados.

RecyclerView e Adapter

Quando obtemos a lista de atividades através do Live Data temos a mostrar ao utilizador. Para isso criamos um RecyclerView (é um componente da biblioteca de suporte que facilita a apresentação de grandes quantidades de dados).

Ao criarmos um RecyclerView criamos os seguintes componentes:

- **LayoutManager**-> Responsável pelo posicionamento das views.
- **Adapter->** Serve de ponte entre o modelo de dados e a RecyclerView.
- **ViewHolder->** Possui uma referência para uma view a ser mostrada no ecrã.

```
public class ActivityAdapter extends RecyclerView.Adapter<ActivityAdapter.ActivityHolder> {
    private List<Atividade> atividades;
    private Context mContext;
    private Click_atividades button;


    public ActivityAdapter(Context context, List<Atividade> atividades) {
        this.mContext = context;
        this.atividades = atividades;
    }

    public void setContactos(List<Atividade> palavras) {
        this.atividades = palavras;
        notifyDataSetChanged();
    }

    @Override
    public void onAttachedToRecyclerView(@NonNull RecyclerView recyclerView) {
        super.onAttachedToRecyclerView(recyclerView);
        try {
            button = (Click_atividades) mContext;
        } catch (ClassCastException e) {
            throw new ClassCastException("Nao consigo converter" + mContext.getPackageName());
        }
    }
```

```
public class ActivityHolder extends RecyclerView.ViewHolder implements View.OnClickListener
    TextView tipo, data, titulo, hora_fim;

    Button btn;


    public ActivityHolder(@NonNull View itemView) {
        super(itemView);
        titulo = itemView.findViewById(R.id.titulo);
        tipo = itemView.findViewById(R.id.time_start);
        //data = itemView.findViewById(R.id.contact_name);
        hora_fim = itemView.findViewById(R.id.km);
        //id = itemView.findViewById(R.id.id);

        itemView.setOnClickListener(this);


    }

    @Override
    public void onClick(View v) {
        int position = getLayoutPosition();
        Atividade atividade = atividades.get(position);
        button.details_atividade(atividade);
        //button.delete(atividade);
    }
}
```

Aqui é apresentado a classe ViewHolder que está situada dentro do adapter.

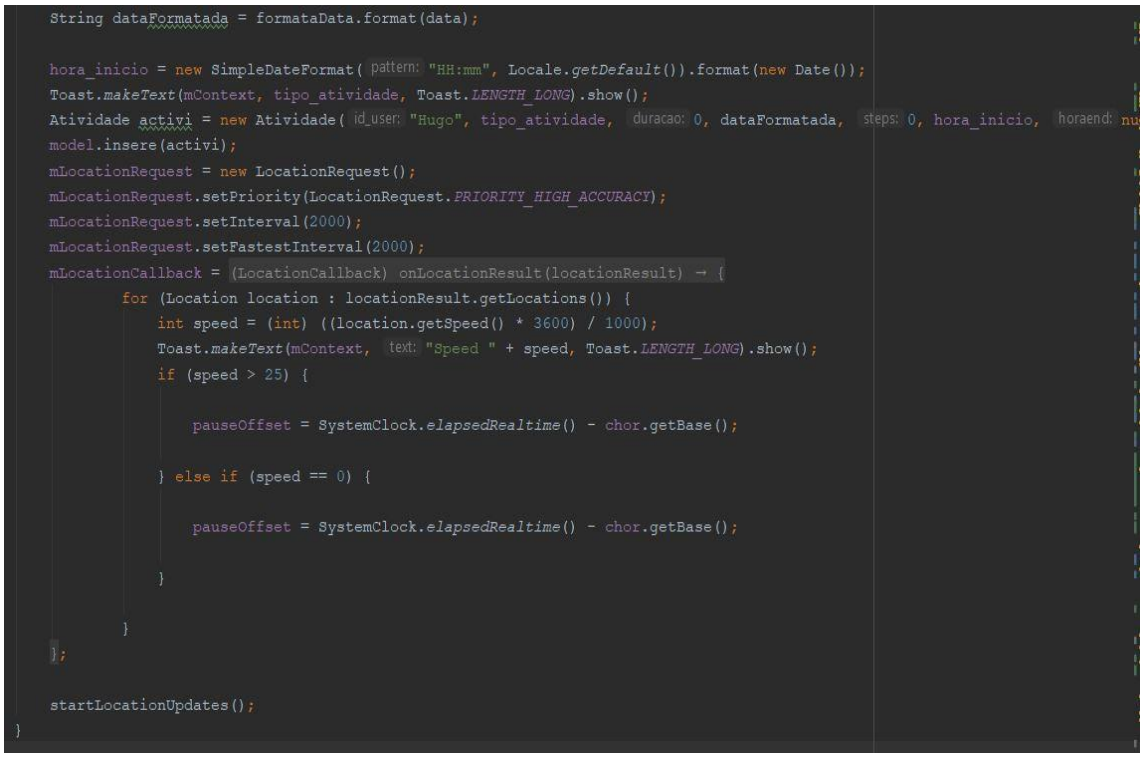
Início de Atividades

Com a base de dados criada e a tabela “atividades” criada criamos o botão que permite iniciar atividade. Assim que a atividade é iniciada é criado na base de dados um objeto da entidade “Atividade” sendo registados a hora de início a data e o id do utilizador que iniciou a atividade. Este processo é realizado através de uma AsyncTask.

Com o início da atividade é também enviado uma notificação a informar que iniciou uma atividade. Para que o envio da notificação fosse possível criamos um canal e de seguida criamos a notificação.

```
//Channel Notification
private void createNotificationChannel() {
    // Create the NotificationChannel, but only on API 26+ because
    // the NotificationChannel class is new and not in the support library
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        CharSequence name = "Hugo";
        String description = "Atividade";
        int importance = NotificationManager.IMPORTANCE_DEFAULT;
        NotificationChannel channel = new NotificationChannel(CHANNEL_ID, name, importance);
        channel.setDescription(description);
        // Register the channel with the system; you can't change the importance
        // or other notification behaviors after this
        NotificationManager notificationManager = getSystemService(NotificationManager.class);
        notificationManager.createNotificationChannel(channel);
    }
}
```

Quando é iniciada uma atividade é executada a AsyncTask que permite registar passos dados pelo utilizador, permite verificar a velocidade a que decorre a atividade e faz o Update à tabela "atividade" durante a realização de uma atividade. A verificação de velocidade consiste em criar um tempo de exercício real ou seja caso o utilizador esteja no carro ou a realizar uma atividade numa velocidade superior ao normal (pessoa no máximo consegue cerca de 35km\h em corrida sprint) o tempo é descontado à duração da atividade ou seja se a atividade demorou 10 minutos mas 4 foram de carro só são contabilizados 6 minutos de atividade.



```
    String dataFormatada = formataData.format(data);

    hora_inicio = new SimpleDateFormat( pattern: "HH:mm", Locale.getDefault()).format(new Date());
    Toast.makeText(mContext, tipo_atividade, Toast.LENGTH_LONG).show();
    Atividade activi = new Atividade( id_user: "Hugo", tipo_atividade, duracao: 0, dataFormatada, steps: 0, hora_inicio, horaend: nu
    model.insere(activi);
    mLocationRequest = new LocationRequest();
    mLocationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
    mLocationRequest.setInterval(2000);
    mLocationRequest.setFastestInterval(2000);
    mLocationCallback = (LocationCallback) onLocationResult(locationResult) → {
            for (Location location : locationResult.getLocations()) {
                int speed = (int) ((location.getSpeed() * 3600) / 1000);
                Toast.makeText(mContext, text: "Speed " + speed, Toast.LENGTH_LONG).show();
                if (speed > 25) {

                    pauseOffset = SystemClock.elapsedRealtime() - chor.getBase();

                } else if (speed == 0) {

                    pauseOffset = SystemClock.elapsedRealtime() - chor.getBase();

                }

            }
    };

    startLocationUpdates();
}
```

O trecho de código da imagem de cima refere-se ao "onPreExecute" da AsyncTask e mostra a inserção de uma atividade na Room através do model que instanciamos. Neste método também é instanciado o LocationCallback que é usado para receber notificações da velocidade do dispositivo para depois verificar se é uma velocidade dentro dos limites definidos ou não.

Neste método é também iniciado a verificação da velocidade através do método "startLocationUpdates".

```java
@Override
protected Atividade doInBackground(Void... voids) {
    try {
        atividade = model.onlyActivity( id_user: "Hugo", tipo_atividade, hora_inicio, steps: 0);
        publishProgress(atividade);

        while (isCancelled() == false) {
            Thread.sleep( millis: 2000);
            publishProgress(atividade);
        }

    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    publishProgress(atividade);


    return atividade;
}
```

Este é o método doInBackground, este método é executado numa nova Thread depois do método onPreExecute (). A primeira execução a ser realizada neste método é a obtenção do id da atividade inserida no onPreExecute, é como este id que depois iremos atualizar os dados da atividade como o número de passos e hora de fim.

Problema Encontrado:

Com a criação deste método surgiu um problema que era poder haver vários ids retornados pelo método "onlyActivity". Caso o utilizador executasse duas atividades (do mesmo tipo) no mesmo minuto o método que apenas deveria retornar um id poderia retornar dois diferentes.

Exemplo: Objeto atividade -> id_user: "1234", tipo_atividade: "corrida", hora início: "12:32", número_passos: "0";

Outra atividade criada no mesmo minuto admitindo que o utilizador não realizou passos:

Objeto atividade -> id_user: "1234", tipo_atividade: "corrida", hora_início: "12:32", número_passos: "0";

Assim o método onlyAtividade terias duas opções de retorno.

Solução:

Para que tal não acontecesse definimos que o utilizador só poderia terminar a atividade passado pelo menos um minuto assim é evitado conflitos no id retornado pelo método "onlyActivity".

Ainda no método doInBackground é executado um ciclo a cada dois segundos que chama o método publishProgress (utilizado para apresentar resultados intermédios na UI enquanto a tarefa continua a ser executada) em que nesse método é executado o Update à base de dados e é chamado o "startLocationUpdates" (verificando a velocidade a que o dispositivo de encontra). Este ciclo é terminado quando o utilizador clica em terminar a atividade (ou seja, a AsyncTask é cancelada).

```
@Override
protected void onProgressUpdate(Atividade... values) {
    super.onProgressUpdate(values);

    long elapsedMillis_second = SystemClock.elapsedRealtime() - chor.getBase();
    duracao=(int) ((elapsedMillis_second / (1000 * 60)) % 60);
    long elapsedMillis = SystemClock.elapsedRealtime() - chor.getBase() - pauseOffset;
    chor.getOnChronometerTickListener();
    startLocationUpdates();
    minutes = (int) ((elapsedMillis / (1000 * 60)) % 60);
    //Updates à base de dados
    String hora_fim = new SimpleDateFormat( pattern: "HH:mm", Locale.getDefault()).format(new Date());
    Update_Activity_Task upda_final = new Update_Activity_Task(atividade.getId(), hora_fim, passos, minutes, model);
    upda_final.execute();
}
```

O trecho de código em cima demonstra a forma como é feito o Update à base de dados. Para isso criamos uma AsyncTask auxiliar visto que todas as operações sobre a base de dados não são permitidas na thread da UI. Neste método é também obtido a duração da atividade. Para obter a duração da atividade é feito a conta do cronometro e retirado o tempo em que a velocidade foi superior ao limite ou foi 0 (tempo esse que o utilizador não este a realizar realmente atividade).

Fim de uma Atividade

Assim que o utilizador clica no botão “Terminar atividade” a task é cancelada. A task possui um método onCancelled que é invocado quando a AsyncTask é cancelada.

Neste método é feito o Update final à base de dados em que são guardados o número de passos e a hora de fim, é também terminado o método que verifica a velocidade e o método que conta os passos.

```
@Override
protected void onCancelled() {
    chor.stop();
    stopLocationUpdates();
    mSensorManager.unregisterListener((SensorEventListener) this);
    //Update final
    String hora_fim = new SimpleDateFormat( pattern: "HH:mm", Locale.getDefault()).format(new Date());
    Update_Activity_Task upda_final = new Update_Activity_Task(atividade.getId(), hora_fim, minutes, passos, model);
    upda_final.execute();


}
```

Com o fim da atividade é também enviada uma notificação ao utilizador com informações sobre a atividade.

Esta notificação contém um botão que permite regressar à aplicação, o canal utilizado para esta notificação é o mesmo utilizado para a notificação enviada quando o utilizador inicia a atividade.

```
private void createNotification_EndActivity(String tipo_atividade, int duracao) {

    Intent botao = new Intent( action: "acao update");
    PendingIntent botaoPendingIntent = PendingIntent.getBroadcast( context: this, requestCode: 0, botao, PendingIntent.FLAG_ONE_SH
    NotificationManager mNotifyMgr = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
    mNotifyMgr.cancel(notificationId);
    Intent resultIntent = new Intent( packageContext: this, User_home.class);
    PendingIntent resultPendingIntent = PendingIntent.getActivity( context: this, requestCode: 0, resultIntent, PendingIntent.FLAG
    builder = new NotificationCompat.Builder( context: this, CHANNEL_ID)
            .setSmallIcon(R.drawable.ic_stat_accessibility)
            .setContentTitle(tipo_atividade + " terminada!")
            .setAutoCancel(true)
            .addAction(R.drawable.ic_stat_accessibility, title: "Iniciar Outra atividade", resultPendingIntent)
            .setContentText("Acabou de terminar uma " + tipo_atividade)
            .setStyle(new NotificationCompat.BigTextStyle()
                    .bigText("Acabou de terminar uma " + tipo_atividade + " que demorou " + duracao + " minutos "))
            .setPriority(NotificationCompat.PRIORITY_DEFAULT);
    builder.setContentIntent(resultPendingIntent);

}
```

A ação realizada no botão é executada através de um PendingIntent.

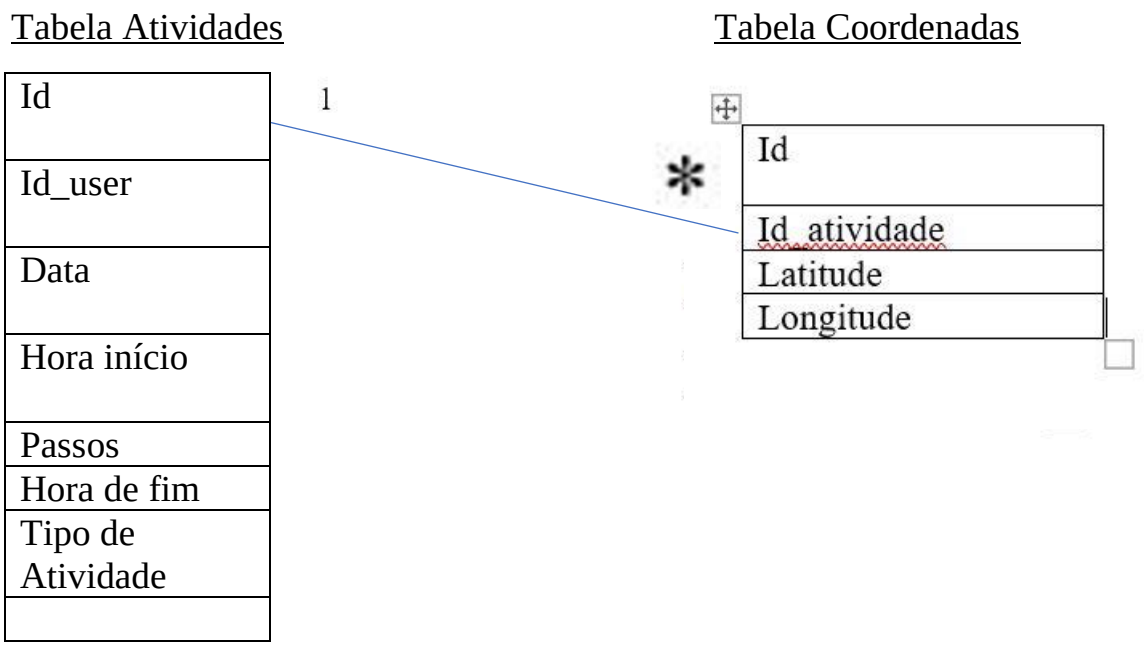
Com o início e o fim da atividade realizados e inseridos na base de dados falta guardar as localizações, ou seja, os pontos pelos quais o utilizador passou durante a atividade.

Planeamento de Relação entre Tabelas Atividade e Coordenadas

Como o Room não permite guardar listas (a não ser que implementamos conversores (que não usamos durante as aulas)) tivemos de recorrer uma tabela para guardar as coordenadas.

Para criar a tabela das coordenadas realizamos os mesmos processos utilizados para a criação da tabela de atividades. Criação de Entidade, criação de DAO e criação do ViewModel.

Com as duas tabelas criadas tivemos de as relacionar.



A tabela coordenadas tem um atributo que recebe o id da atividade assim relacionamos as duas tabelas numa relação de um para muitos, ou seja, uma atividade pode ter várias coordenadas e uma coordenada só esta associada a uma atividade.

Registo de Coordenadas

Criada a tabela e os métodos para a inserção na base de dados criamos um LocationRequest e um LocationCallback. No LocationRequest definimos que a atualização é feita de 30 em 30 segundos. LocationCallback é o listener que vai receber a lista das atualizações.

```
//Localização
mLocationRequest = new LocationRequest();
mLocationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
mLocationRequest.setInterval(30000);
mLocationRequest.setFastestInterval(30000);
mLocationCallback = (LocationCallback) onLocationResult(locationResult) → {
        for (Location location : locationResult.getLocations()) {
            Toast.makeText(mContext, text: " " + location.getLatitude(), Toast.LENGTH_LONG).show();
            list_latitude.add(location.getLatitude());
            list_longitude.add(location.getLongitude());

        }
};
```

No LocationCallback criamos dois arrays que irão guardar a latitude e a longitude. Assim que o utilizador inicia a atividade é registado a localização periodicamente de 30 em 30 segundos. Este método é executado no Fragmento que inicia a atividade logo tem como principal desvantagem caso o utilizador suspenda o telemóvel ou saia da aplicação enquanto uma atividade decorre não será guardado a localização.

Assim que o utilizador termina a atividade é parado o método que executa a atualização da localização e a lista de pontos é inserida na tabela coordenadas, o id da atividade em curso já foi obtido através de uma AsyncTask.

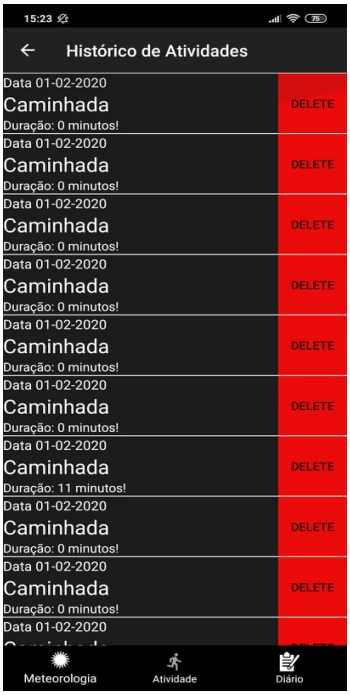
```
int contador_insercao = 0;
while (contador_insercao < list_latitude.size()) {
    Coordenadas c = new Coordenadas(list_longitude.get(contador_insercao), list_latitude.get(contador_insercao), id, contador_insercao);
    model_coordenadas.insere(c);
    contador_insercao++;
}
click.mudafragment();
```

Nesta inserção é também inserido um número que corresponde à ordem dos pontos para prevenir que quando forem chamados (através do query) os pontos tenham uma ordem evitando assim mostrar os pontos por uma ordem incorreta.

Listagem de Atividades

Com a inserção de atividades e coordenadas realizadas implementamos a parte de mostrar ao utilizador o histórico de atividades realizadas. Para isso criamos um layout para a RecyclerView que já tínhamos criado depois foi chamar o método que também já tínhamos implementado no ViewModel e que usa o Live Data este método seleciona tudo da tabela atividade em que id do user é igual ao id do user que fez o login (mostrar apenas as atividades de cada utilizador).

```
@Query("Select * from atividade  where id_user= :id_user ")
LiveData<List<Atividade>> getallAtividades( String id_user );
```



A imagem em cima mostra o layout já com a lista de atividades realizadas pelo utilizador, aqui o utilizador tem duas operações que pode realizar ver os detalhes da atividade ou eliminá-la através do button a vermelho chamado de “delete”. Estas ações foram definidas no Adapter das atividades.

```
itemView.setOnClickListener(this);

btn.setOnClickListener(this);
```

Caso o utilizador clique em apagar uma atividade é chamada uma interface que permite aceder aos métodos na atividade em que o adapter está inserido.

```
if (v.getId() == btn.getId()) {
    button.delete_atividade(atividade);
} else {
    button.details_atividade(atividade);
}
```

Nessa atividade estão os métodos que permitem apagar atividade através do ViewModel e o método que permite abrir o fragmento com os detalhes de cada atividade (Assumindo que está no modo telemóvel).

Caso o utilizador clique para ver os detalhes da atividade é passado ao fragmento details (Fragmento responsável por apresentar os detalhes da atividade) a atividade correspondente e uma lista de pontos que foram obtidas da tabela coordenadas com base no id da atividade (relação entre tabelas que foi apresentada na 18).

```
@Override
public void details_atividade(final Atividade ca) {
    if (isSmartphone) {
        getSupportActionBar().setTitle("Detalhes da  Atividade!");
        model_coordenadas.allCoordenadas(ca.getId()).observe( owner: this, (Observer) (coordenadas) → {
                Toast.makeText( context: List_Ativities.this,  text: " " + coordenadas.size(), Toast.LENGTH_LONG).show();
                Fragment_Details adeus = new Fragment_Details(ca, coordenadas,  ativity: List_Ativities.this);
                //Passa Argumentos pro fragmento
                FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();
                transaction.replace(R.id.placeholder, adeus);
                transaction.commit();
        });
```

O Fragment Details depois mostra os detalhes de cada atividade e contem também um mapa que irá mostrar os pontos obtidos.

```
@Override
public void details_atividade(final Atividade ca) {
    if (isSmartphone) {
        getSupportActionBar().setTitle("Detalhes da  Atividade!");
        model_coordenadas.allCoordenadas(ca.getId()).observe( owner: this, (Observer) (coordenadas) → {
                Toast.makeText( context: List_Ativities.this,  text: " " + coordenadas.size(), Toast.LENGTH_LONG).show();
                Fragment_Details adeus = new Fragment_Details(ca, coordenadas,  ativity: List_Ativities.this);
                //Passa Argumentos pro fragmento
                FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();
                transaction.replace(R.id.placeholder, adeus);
                transaction.commit();
        });
```

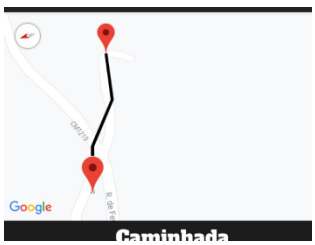
Na imagem ao lado é mostrado o layout do fragmento que mostra os detalhes de cada atividade. Este Fragmento é de referir que contem um método que calcula a distância aproximada entre pontos. Esta distância é calculada primeiro ponto e o último, mas sim de ponto a ponto de modo a apresentar valores mais perto dos reais percorridos pelo utilizador.


```
while (i < c.size() - 2) {
    Location start = new Location( provider: "Start Point");
    start.setLatitude(list_coordenadas.get(i).getLatitude());
    start.setLongitude(list_coordenadas.get(i).getLongitude());
    Location finish = new Location( provider: "Finish Point");
    finish.setLatitude(list_coordenadas.get(i + 1).getLatitude());
    finish.setLongitude(list_coordenadas.get(i + 1).getLongitude());
    float distance = start.distanceTo(finish);
    distancia = distance + distancia;
    i++;
}
DecimalFormat df = new DecimalFormat( pattern: "##.##");
txt.setText(" " + df.format( number: distancia / 1000) + " km!");
```

Método que percorre toda a lista de coordenadas e calcula a distância entre os pontos, quando termina atualiza uma TextView com o valor calculado em kms.

```
@Override
public void onMapReady(GoogleMap googleMap) {
    this.mGoogleMap = googleMap;
    //Criar a linha das coordenadas
    if (list_coordenadas.size() != 0) {
        LatLng latLng = new LatLng(list_coordenadas.get(0).getLatitude(), list_coordenadas.get(0).getLongitude());
        mGoogleMap.addMarker(new MarkerOptions().position(latLng).title("Inicio").snippet("Inicio da atividade"));
        PolylineOptions rectOptions = new PolylineOptions();
        for (Coordenadas lng : list_coordenadas) {
            LatLng latLng2 = new LatLng(lng.getLatitude(), lng.getLongitude());
            rectOptions.add(latLng2);
        }
        LatLng lastLng = new LatLng(list_coordenadas.get(list_coordenadas.size() - 1).getLatitude(), list_coordenadas.get(list_coordenadas.size
        mGoogleMap.addMarker(new MarkerOptions().position(lastLng).title("FIM").snippet("Fim da atividade"));

        mGoogleMap.moveCamera(CameraUpdateFactory.newLatLngZoom(lastLng, v: 14));

        Polyline polyline = mGoogleMap.addPolyline(rectOptions);
        calculaDistanciaEmKM(list_coordenadas,distancia);
    } else {
```

O Fragmento também possui uma PolylineOptions que permite criar uma linha entre os pontos guardados na lista, o primeiro local e último são marcados com Markers.



Modo Tablet

Durante o trabalho implantamos também o modo tablet, em termos de layout foram todos alterados quer seja para aparecer em letras maiores quer seja para as imagens serem maiores. Em termos de alteração de código apenas foi mudado a forma como é apresentada a lista de atividades juntamente com os detalhes em simultâneo. Toda a gestão de Fragmentos é feita na mesma atividade.

Quando a atividade a primeira coisa é verificar se foi inicializado num tablet ou num telemóvel

```
if (findViewById((R.id.list_ativity)) != null) {
    isSmartphone = true;

} else {
    isSmartphone = false;
}
```

Este método verifica se o fragmento de telemóvel foi instância e apartir daí conseguimos verificar em qual dos dispositivos está a ser executada a nossa aplicação.

```
} else {
    model.allAtivities().observe( owner: this, (Observer) (atividades) → {

        //Fragment Detalhes da atividade
        Fragment_listof_ativities list = new Fragment_listof_ativities(atividades, ativi: List_Ativities.this);
        FragmentManager manager = getSupportFragmentManager();
        FragmentTransaction transaction = manager.beginTransaction();
        transaction.replace(R.id.fragment, list);
        transaction.addToBackStack(null);
        transaction.commit();

        details = (Fragment_Details)
                getSupportFragmentManager().findFragmentById(R.id.fragment2);


    });

}
```

Caso esteja no modo tablet é passado a lista com as atividades ao fragmento responsável pela listagem e obtido o id do fragmento detalhes.

De seguida quando é obtido um evento de click são obtidas as coordenadas de acordo com o id da atividade (de igual forma como na versão para telemóvel) para em vez de ser criado um fragmento a substituir outro é apenas atualizado o fragmento details uma vez que estão os dois fragmentos em simultâneo.

```
} else {
    model_coordenadas.allCoordenadas(ca.getId()).observe( owner: this, (Observer) (coordenadas) → {
            //Fragment Detalhes da atividade
            details.setAllCoordenadas(coordenadas);
            details.setAtividade(ca);


    });
```

```
public void setAtividade(Atividade atividade) {
    duracao.setText("Decorreu durante: " + atividade.getDuracao() + " minutos!");
    text.setText(atividade.getTipo());
    passos.setText(" " + atividade.getSteps() + " Passos!");
    data.setText(" " + atividade.getData());
    time_start.setText(" " + atividade.getTimestart());
    time_end.setText(" " + atividade.getTimeend());

}
```

A imagem em cima mostra o código em que é atualizado as TextsView com os valores recebidos como parâmetro.

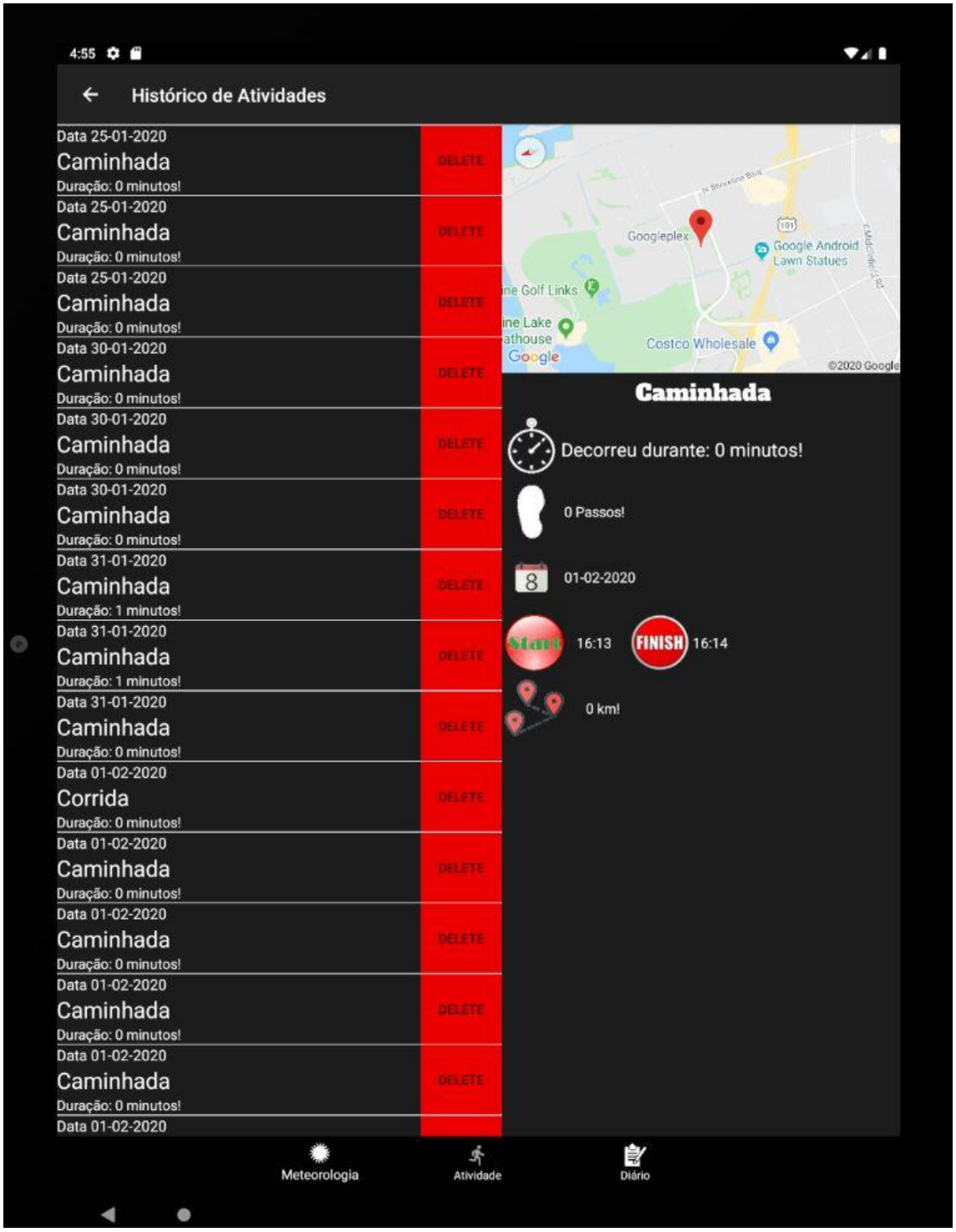
```
public void setAllCoordenadas(List<Coordenadas> list_coordenadas) {
    this.list_coordenadas = list_coordenadas;
    if (list_coordenadas.size() != 0) {
        SupportMapFragment m = (SupportMapFragment) getChildFragmentManager().findFragmentById(R.id.map);
        m.getMapAsync((OnMapReadyCallback) this);
    }

}
```

Nesta imagem podemos ver o método que permite a atualização do mapa a cada click.

Definições

Implementamos também as definições para o utilizador selecionar de quanto em quanto tempo pretender fazer Update à localização.

Anexos



Layout modo tablet.

Layout durante a realização de uma atividade.

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation 'com.google.firebase:firebase-auth:17.0.0'
    //implementation 'com.android.support:design:28.0.0'
    testImplementation 'junit:junit:4.12'
    implementation 'androidx.legacy:legacy-support-v4:1.0.0'
    //implementation 'com.google.android.gms:play-services-maps:11.6.0'
    implementation 'com.google.android.gms:play-services-location:17.0.0'
    implementation 'com.google.android.gms:play-services-maps:17.0.0'
    // https://mvnrepository.com/artifact/com.google.android.gms/play-services-gcm
    implementation group: 'com.google.android.gms', name: 'play-services-gcm', version: '17.0.0'

    androidTestImplementation 'androidx.test:runner:1.2.0'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
    implementation 'com.google.firebase:firebase-firestore:21.3.1'
    implementation group: 'androidx.room', name: 'room-runtime', version: '2.2.1'
    androidTestImplementation group: 'androidx.room', name: 'room-testing', version: '2.2.1'
    annotationProcessor 'androidx.room:room-compiler:2.2.1'
    implementation 'androidx.recyclerview:recyclerview:1.1.0'
    implementation 'androidx.lifecycle:lifecycle-extensions:2.1.0'
    ///Template Graphico layout steps
    implementation 'com.github.j4velin.EazeGraph:EazeGraph:1.0.3'
    implementation 'com.google.android.material:material:1.0.0'
    implementation 'com.squareup.retrofit2:retrofit:2.5.0'
    implementation 'com.squareup.retrofit2:converter-gson:2.5.0'
}
```

Dependências do projecto.

Conclusão

Com a realização deste projecto aprofundamos os nossos conhecimentos sobre programação para Android, concluímos também que conseguimos apresentar um produto funcional e que permite registar atividades a qualquer altura do dia. Durante a realização do trabalho podemos também verificar que existem várias bibliotecas de suporte que dão grande ajuda no desenvolvimento de aplicações para android como por exemplo o Firebase e a API Google Play Services de Localização.

Referências

https://developers.google.com/android/reference/com/google/android/gms/maps/model/PolygonOptions (Utilizado para obter informações de como fazer a linha entre mapas).

https://www.youtube.com/watch?v=tbh9YaWPKKs (Utilizado para obter informação sobre login e registo com Firebase).

https://www.youtube.com/watch?v=TwHmrZxiPA8(Utilizado para obter informação sobre login e registo com Firebase).

https://developer.android.com/reference/android/os/AsyncTask(Documentação sobre AsyncTask).

https://pt.stackoverflow.com/questions/64260/gostaria-de-achar-a-distancia-em-km-entre-dois-markers-como-posso-fazer (Tutorial como obter distancia entre dois pontos).

https://www.youtube.com/watch?v=RLnb4vVkftc (Para tempo cronómetro).

-Documentação das aulas teóricas e exercício realizados durante as aulas.