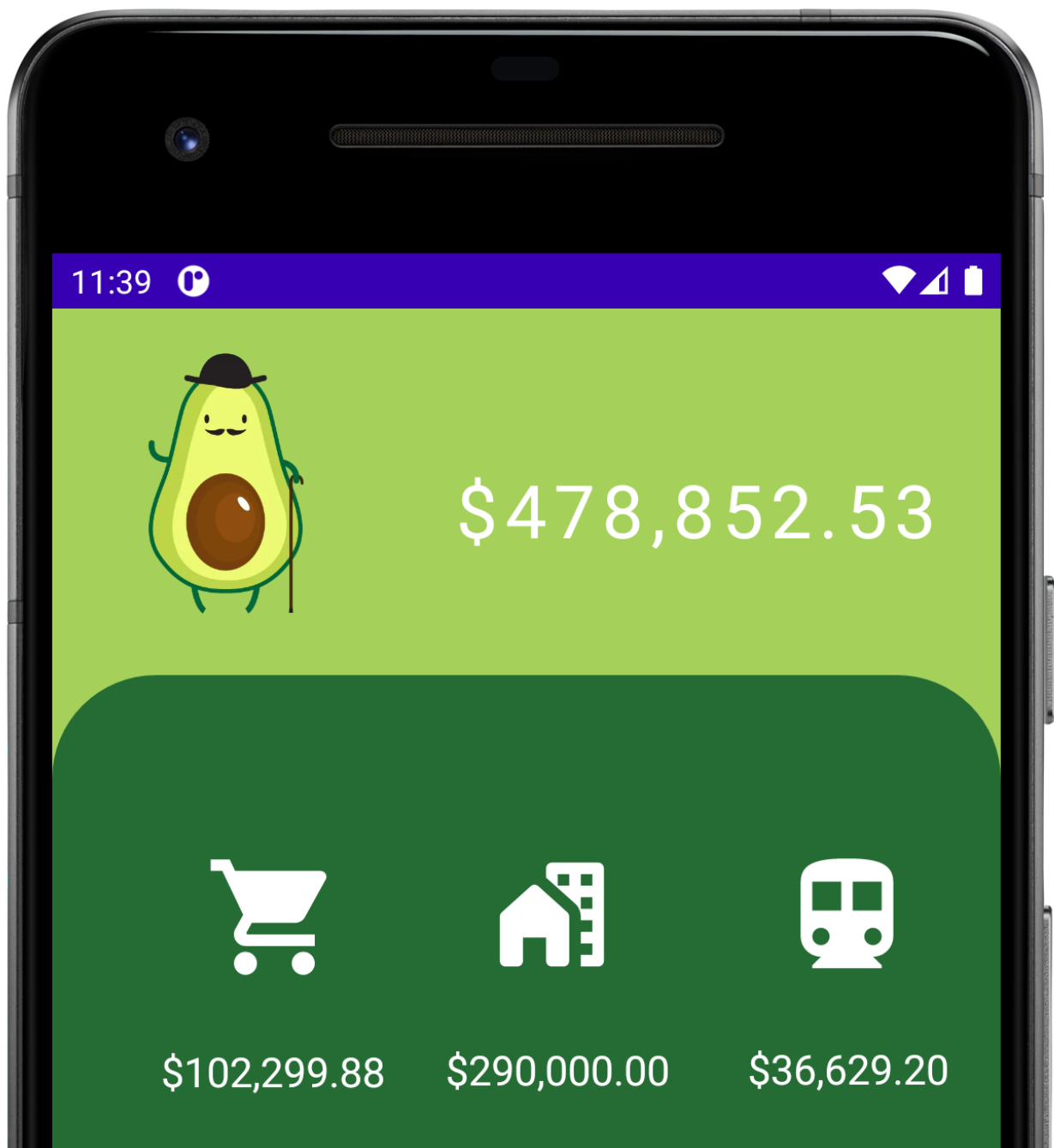


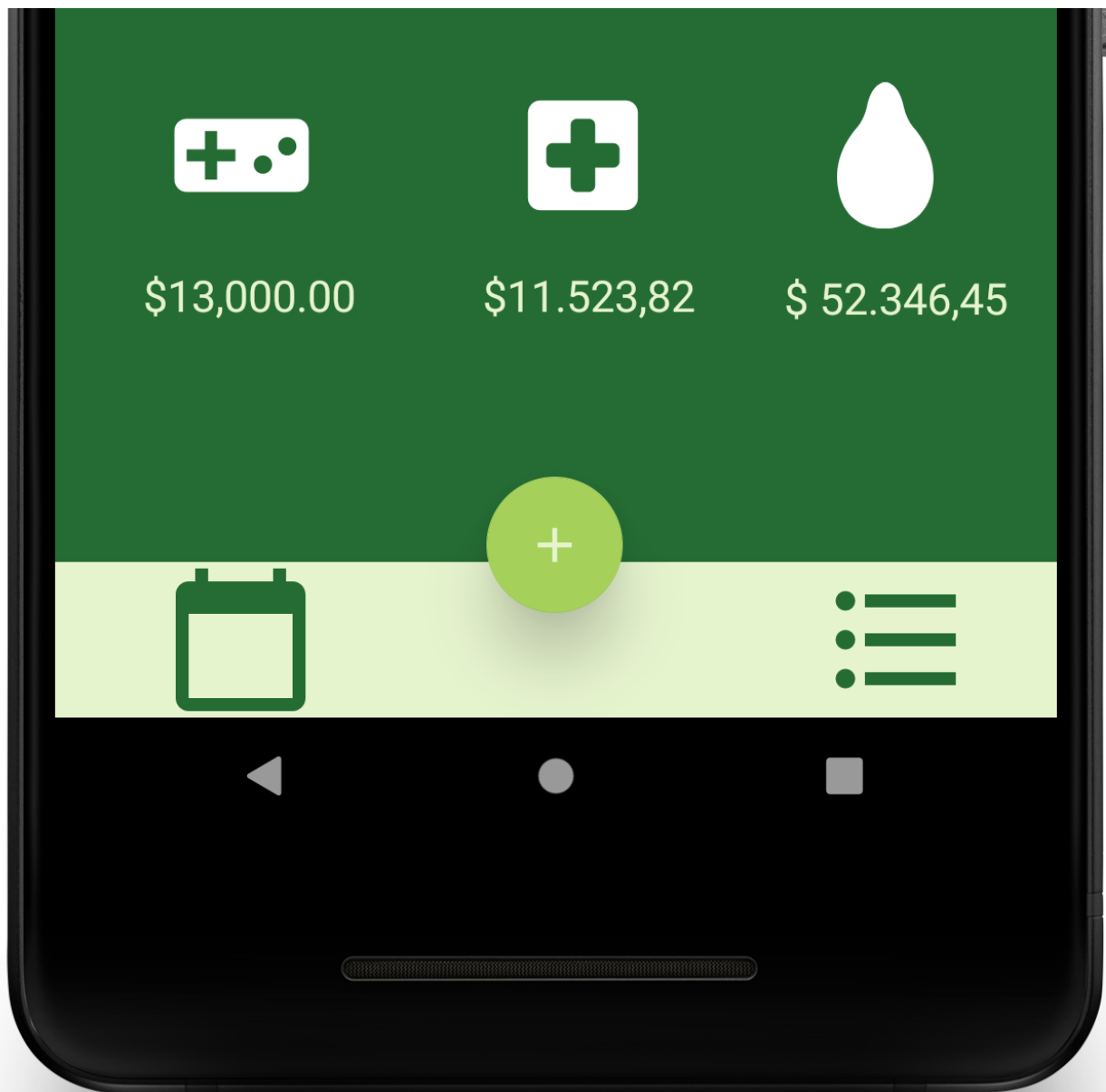
MODULO 3 | Desarrollo de Aplicaciones Móviles Android Java | Ignacio Cavallo

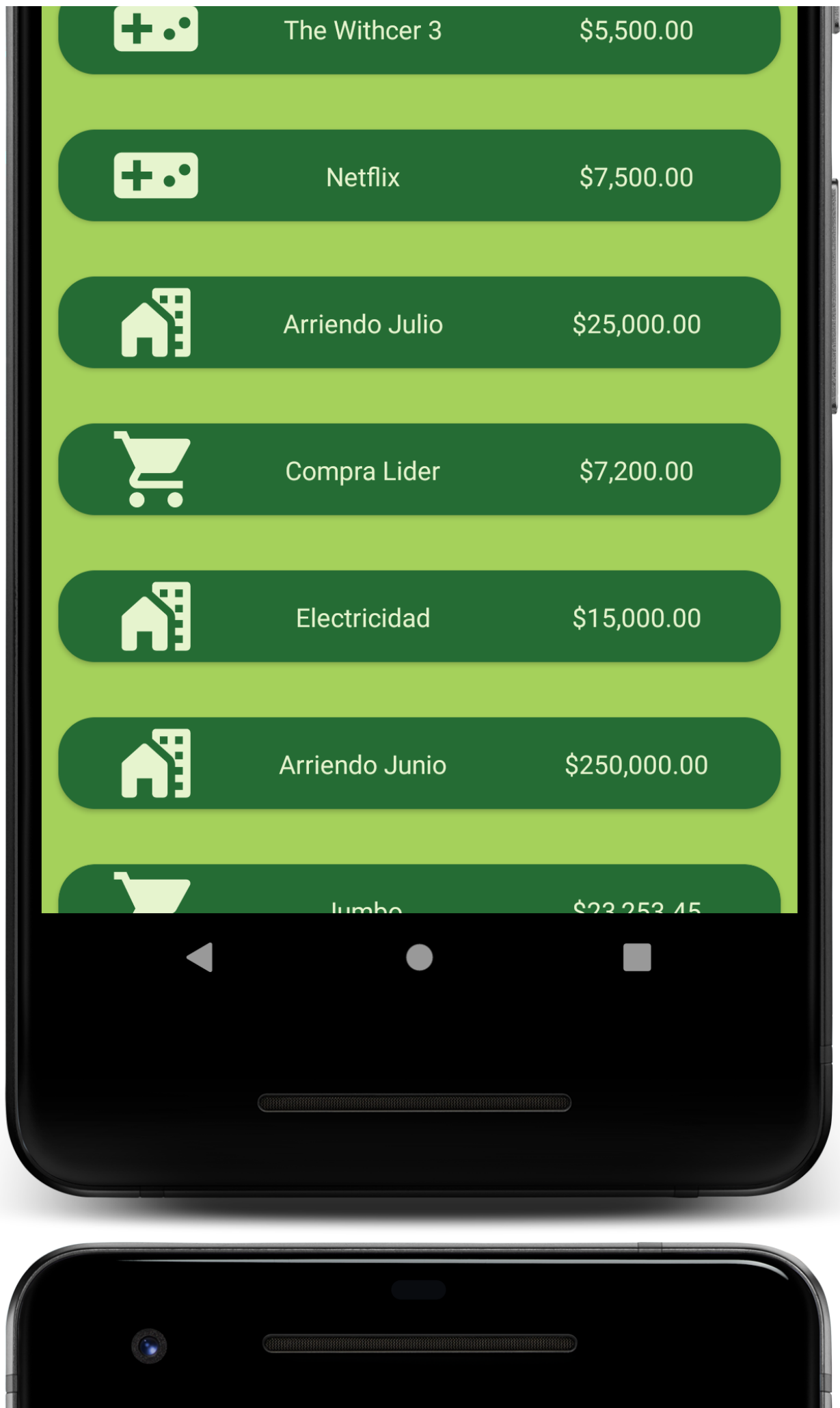
https://github.com/cavigna/modulo_desarrollo_de_aplicaciones_moviles_android_java

EXAMEN | Clase 55 | 19-07

PaltaApp







11:39



Gastos Mensuales



Julio

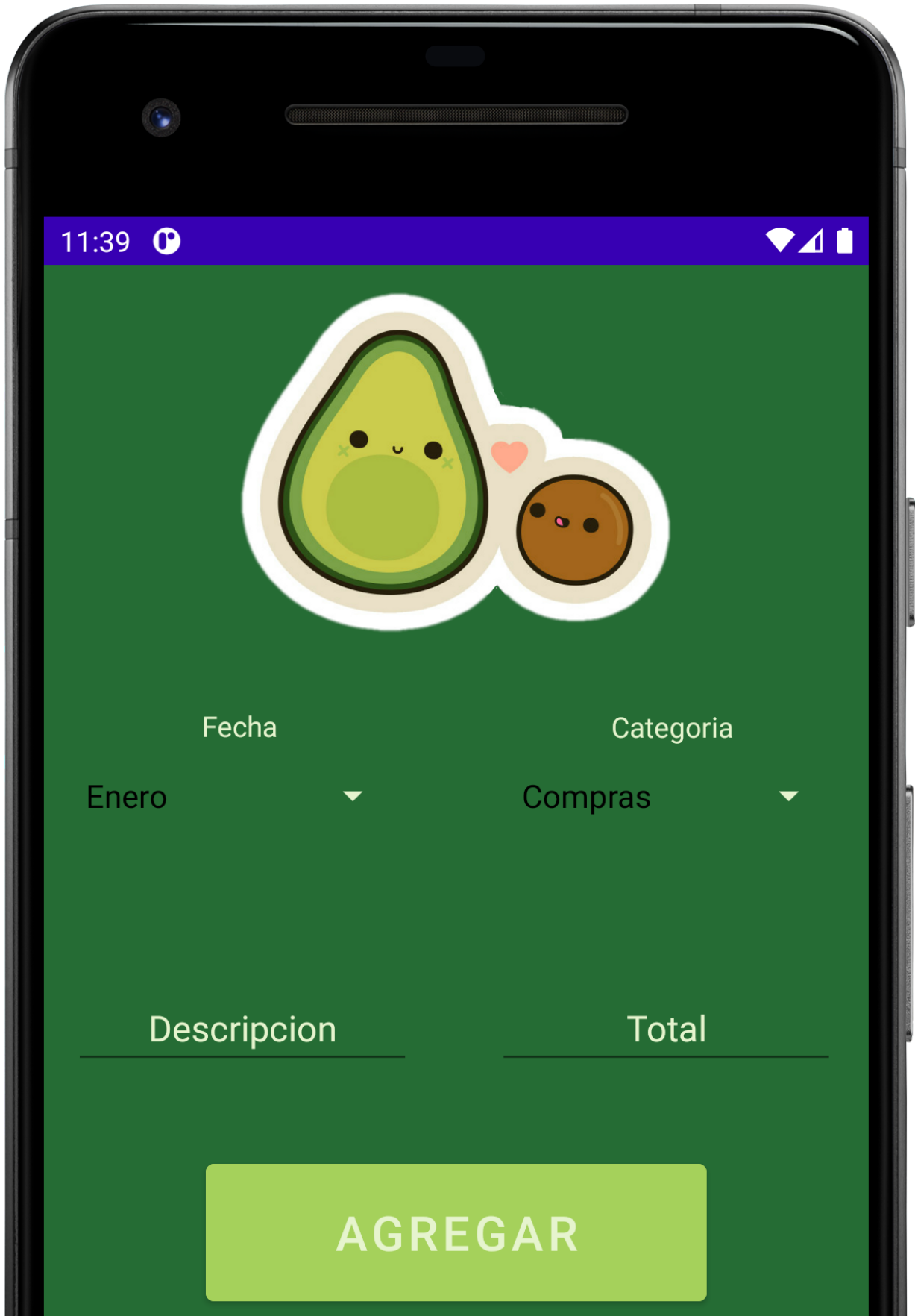
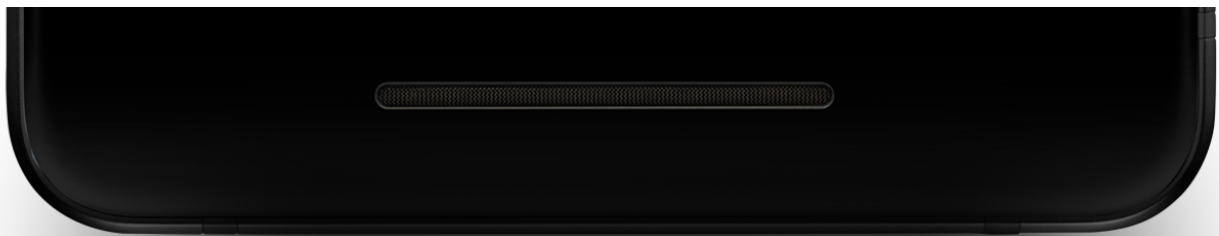
\$65,200.00

Junio

\$397,408.61

Mayo

\$16,243.92



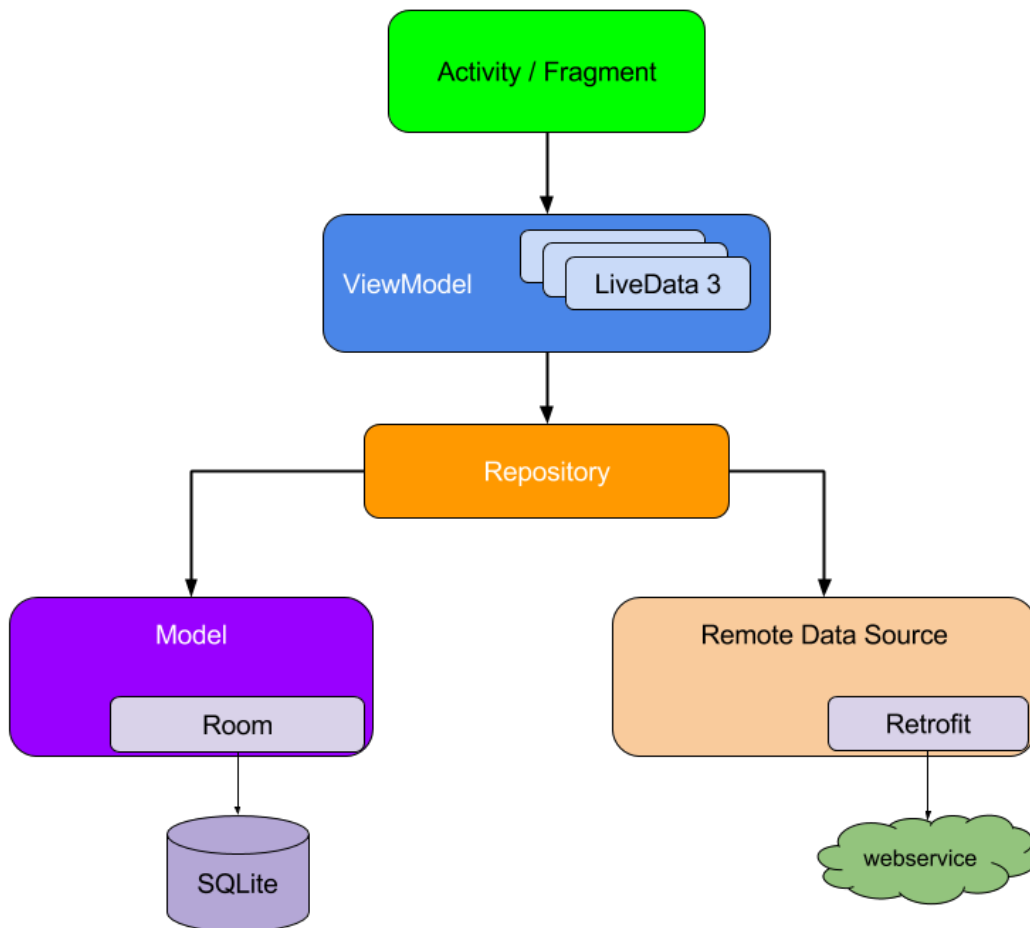


Siguiendo los lineamientos del profesor, construí una app que llevara la cuenta de los gastos, y que el usuario pueda agregarlos o eliminarlos junto a una vista que permita comparar mes a mes y le agregue otra vista con los últimos gastos.

Como desafío me propuse usar lo aprendido en clase, junto a otras herramientas que no fueron impartidas, pero que son imprescindibles en el desarrollo de aplicaciones en Android, entre ellas destaco a **RecyclerView**, **arquitectura MVVM**, y **el uso de Jetpack Room** en oposición a SQLite directo. Por ende, explicaré lo siguiente:

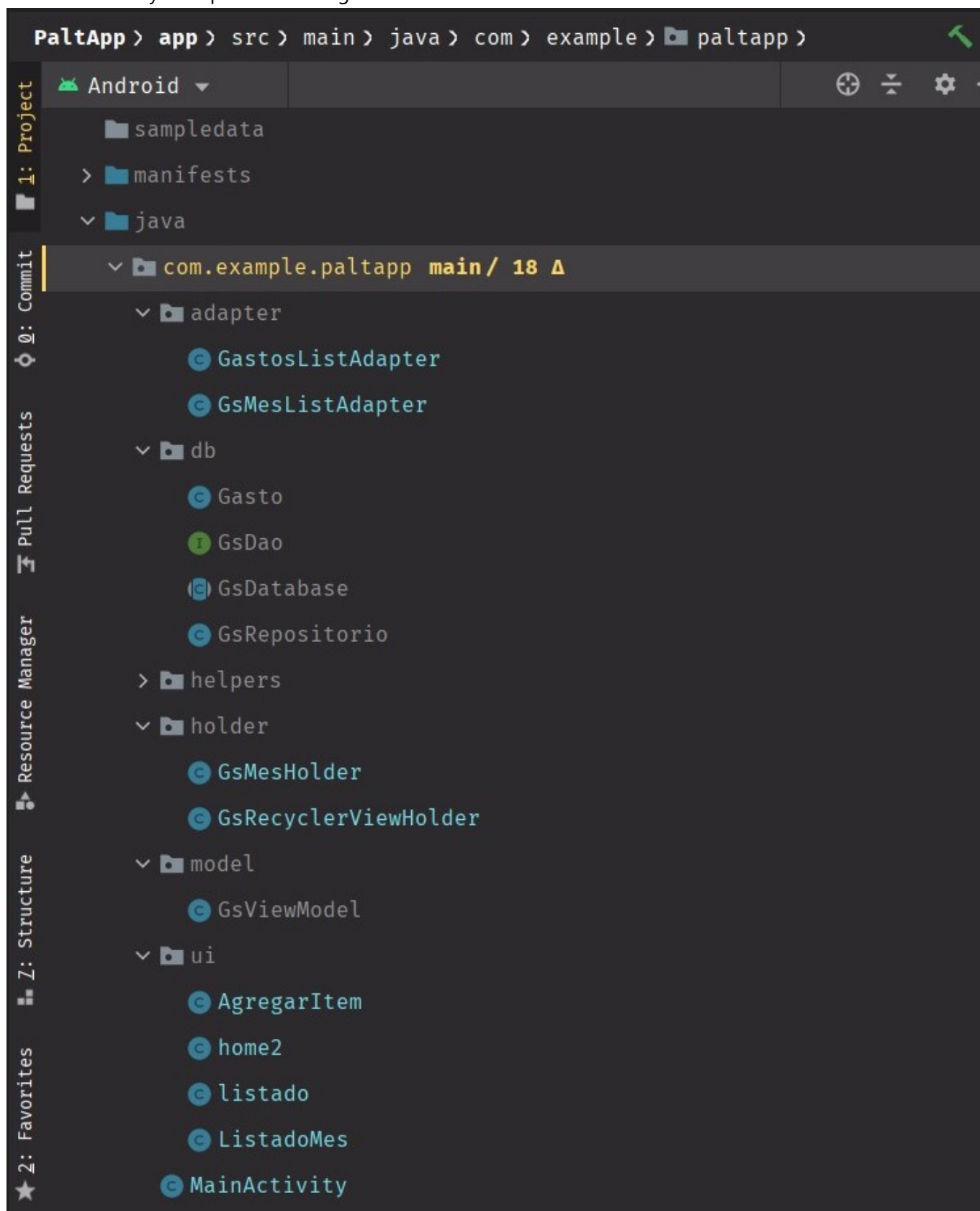
1. [Diseñar la App con Arquitectura ModelView ViewModel \(MVVM\)](#)
2. [Android Room para SQL](#).
3. [Recyclerview](#)

Model View ViewModel



Consiste en separar la UI (Interfaz Gráfica), de los View Model (Intermediario entre Business Logic y UI) de los Model (Business Logic) por medio de un Repositorio (Este último no es Obligatorio, pero sí, sugerido). Esto requiere que en las Activities solo exista código relativo a la UI, o la interfaz gráfica, pero que no, haya nada relativo a la lógica de la App. A su vez, se requieren Holders, ViewModels y Adaptadores para los RecyclerView.

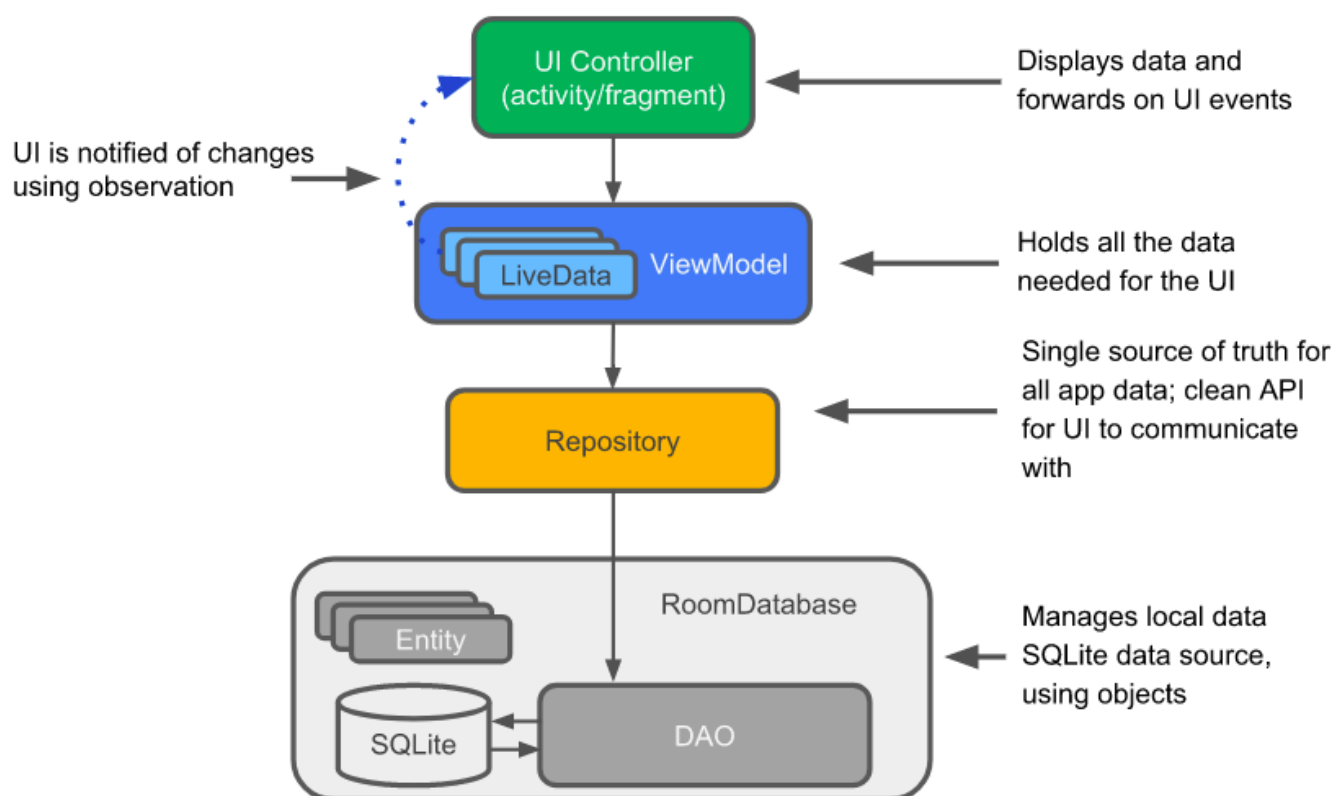
Por ende mi Proyecto quedó con la siguiente estructura:



- **UI:** Incluye las Activities, solo con los aspectos visuales.
- **DB:** Incluye la Clase base **Gasto.java**, Su respectivo DAO **GsDao.java**, la construcción de la db **GsDatabase.java** y el repositorio **GsRepositorio.java**
- **Holder:** Aquí encontramos los holders que se conecta de forma directa con la lógica de la App en el ViewModel. Son una de las tantas capas de abstracción que sugiere *Google*

- **ViewModel o Model:** Acá está la *verdad de La Milanese*, es el lugar donde ocurre gran parte de la lógica.
- **Adapter :** Son los adaptadores requeridos para las vistas que contienen listados.

Se buscó replicar este diagrama:

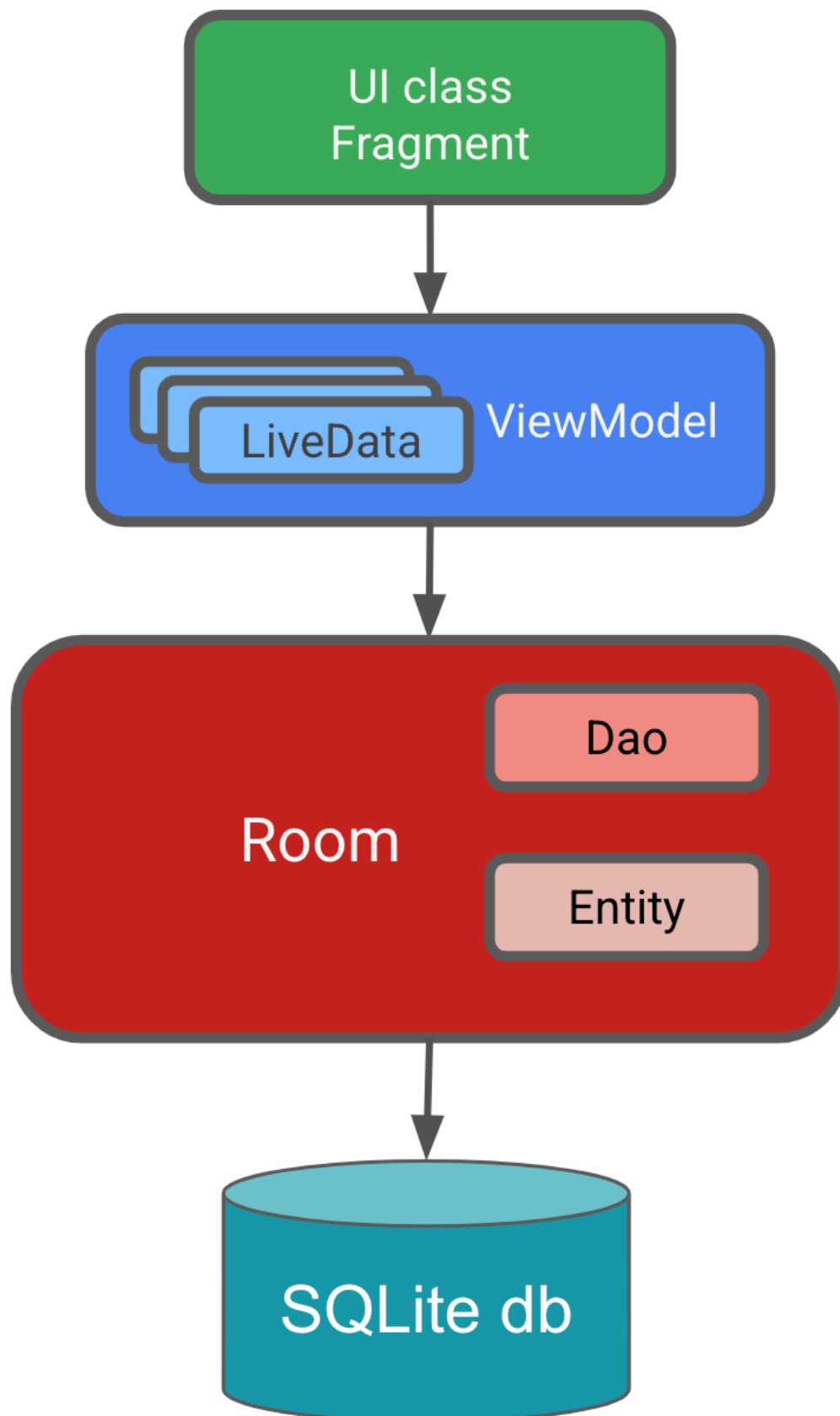


Android Room

¿Pero que es este famoso Android Room?

Room es una librería que abstrae el uso de SQLite al implementar una capa intermedia entre esta base de datos y el resto de la aplicación. De esta forma se evitan los problemas de SQLite sin perder las ventajas de su uso. Room funciona con una arquitectura cuyas clases se marcan con anotaciones preestablecidas. Por otro lado, la mayoría de las consultas a la base de datos sí se comprueban en tiempo de compilación.

Entonces tenemos:



Entity

En *PaltApp* la entidad consiste en una clase con la anotación: `@Entity`, en mi caso fue la de `Gasto.java`:

```

package com.example.paltapp.db;

import androidx.room.Entity;
import androidx.room.PrimaryKey;

@Entity(tableName = "tabla_gastos")
public class Gasto {

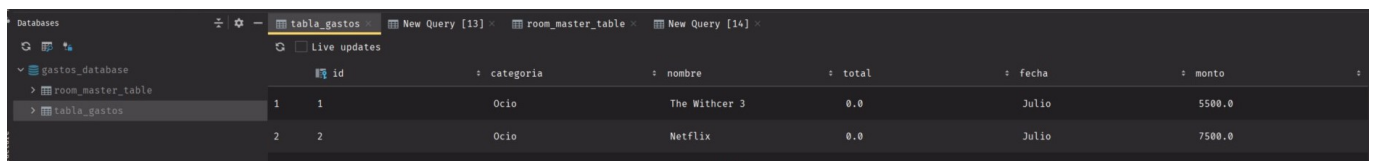
    @PrimaryKey(autoGenerate = true)
    private int id;
    private String categoria;
    private String nombre;
    private String fecha;
    private double monto;
    private double total;

    // Constructor
    public Gasto( String categoria, String nombre, String fecha, double monto) {
        this.nombre = nombre;
        this.categoria = categoria;
        this.fecha = fecha;
        this.monto = monto;
    }

    // GETTERS AND SETTERS.
    /*.....*/

```

Esta clase cumple la función de un mapeado del comando **CREATE** de una tabla en SQLite. Por ende, Cada uno de los Atributos será una de las columnas de la tabla:



| | id | categoria | nombre | total | fecha | monto |
|---|----|-----------|---------------|-------|-------|--------|
| 1 | 1 | Ocio | The Withcer 3 | 0.0 | Julio | 5500.0 |
| 2 | 2 | Ocio | Netflix | 0.0 | Julio | 7500.0 |

DAO

Una vez que tengamos la entidad debemos crear un **Data Access Object (DAO)**. Este representa un mapeado de operaciones SQL a métodos.

Lo que quiere decir que crearemos, leeremos, actualizaremos y eliminaremos registros desde ellos.

```

package com.example.paltapp.db;

import androidx.lifecycle.LiveData;
import androidx.room.Dao;
import androidx.room.Delete;
import androidx.room.Insert;
import androidx.room.OnConflictStrategy;

```

```

import androidx.room.Query;
import androidx.room.RawQuery;
import androidx.room.Update;

import java.util.List;

@Dao
public interface GsDao {

    @Insert(onConflict = OnConflictStrategy.IGNORE)
    void insertGasto(Gasto gasto);

    @Update
    void updateGasto(Gasto gasto);

    @Delete
    void deleteGasto(Gasto gasto);

    @Query("DELETE FROM tabla_gastos")
    void deleteAllGasto();

    //seleccionar todos los gastos. Usado en La lista de todos los gastos
    @Query("SELECT * FROM tabla_gastos")
    LiveData<List<Gasto>> selectAllGastos();

    //suma todos los gastos. Será usado en la Main activity para ver el total
    @Query("SELECT SUM(monto) AS value FROM tabla_gastos")
    LiveData<Double> sumAllGastos();

    //Suma Gs y lo ordena por fecha. Sirve para la vista de comparación Mensual
    @Query("SELECT id, categoria, fecha, nombre, monto, SUM(monto) as total FROM"
+
        " tabla_gastos group by fecha")
    LiveData<List<Gasto>> gastosMes();

```

Cabe destacar el uso de **LiveData**:

LiveData es una clase de contenedor de datos observable. A diferencia de un observable regular, LiveData está optimizado para ciclos de vida, lo que significa que respeta el ciclo de vida de otros componentes de las apps, como actividades, fragmentos o servicios. Esta optimización garantiza que LiveData solo actualice observadores de componentes de apps que tienen un estado de ciclo de vida activo.

Es decir, es una clase que nos permite observar de forma asincrónica los datos de nuestra db. Otra consideración es la diferencia entre:

```
LiveData<List<Gasto>>
```

Nos devuelve un LiveData en un formato de Lista, por que en caso que el resultado de la Query sea más de una, necesitaremos una lista.

```
LiveData<Double> sumAllGastos();
```

En este caso la Query, es solo un número, por ende, es simplemente un Double.

Room DataBase

Es el punto de entrada principal para comunicar el resto de paltapp con el esquema relacional de datos. Esta clase nos oculta la implementación de SQLiteOpenHelper para facilitarnos el acceso a la base de datos.

```
package com.example.paltapp.db;

import android.content.Context;

import androidx.annotation.NonNull;
import androidx.room.Database;
import androidx.room.Room;
import androidx.room.RoomDatabase;
import androidx.sqlite.db.SupportSQLiteDatabase;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

@Database(entities = Gasto.class, version = 2, exportSchema = false)
public abstract class GsDatabase extends RoomDatabase {

    public abstract GsDao gsDao();

    private static GsDatabase minstance; // singleton
    private static final String DB_NAME = "gastos_database";

    private static final int NUMBER_OF_THREADS = 4;

    static final ExecutorService databaseWriteExecutor =
        Executors.newFixedThreadPool(NUMBER_OF_THREADS);

    public static GsDatabase getDatabase(final Context context) {

        if (minstance == null) {
            synchronized (GsDatabase.class){
                if (minstance == null){
                    minstance = Room.databaseBuilder(context.getApplicationContext(),
                        GsDatabase.class, DB_NAME)
                        .fallbackToDestructiveMigration()
                        .addCallback(sRoomDatabaseCallback)
                        .build();
                }
            }
        }
    }
}
```

```

    }

    return minstance;
}

private static RoomDatabase.Callback sRoomDatabaseCallback = new
RoomDatabase.Callback() {

    @Override
    public void onCreate(@NonNull SupportSQLiteDatabase db) {
        super.onCreate(db);
        databaseWriteExecutor.execute(() -> {
            GsDao dao = minstance.gsDao();

            //Llena la base de Datos al ser construida. Simplemente para
testing
            Gasto gasto = new Gasto("Ocio", "Netflix", "Julio",
                                7500);
            dao.insertGasto(gasto);
            /* ...*/
        });
    }
};
}

```

Puntos a recordar:

- `@Database(entities = Gasto.class, version = 2, exportSchema = false)`: Hace referencia a la Entidad, y la versión implica modificaciones. Empezé con una version 1, pero luego tuve que agregar una columna total para poder hacer queries(si no, hay ciertas consultas que dan error), entonces tuve que hacer una version 2.
- `private static GsDatabase minstance;`: Es un singleton garanrizando que una clase solo tenga una instancia. Solo realizamos una única instancia de la db.

- ```

 if (minstance == null) {
 synchronized (GsDatabase.class){
 if (minstance == null){
 minstance =
Room.databaseBuilder(context.getApplicationContext(),
 GsDatabase.class, DB_NAME)
 .fallbackToDestructiveMigration()
 .addCallback(sRoomDatabaseCallback)
 .build();
 }
 }
 }

```

Si no existe una instancia de la db, entonces, construimos una incluyendo un callBack que permite crear dummy data(para que la db este con algunos datos.)

## Repositorio | Model

Estas dos clases las presento juntas, ya que el repositorio no es obligatorio, es una recomendación de Google a la hora de implementar MVVM, generando una capa mas de abstracción. Veremos que el código se repite en estas clases

### Repositorio

```
package com.example.paltapp.db;

import android.app.Application;

import androidx.lifecycle.LiveData;

import java.util.List;

public class GsRepositorio {

 private GsDao gsDao;
 private LiveData<List<Gasto>> mAllGastos;
 private LiveData<List<Gasto>> mgastosMes;

 private LiveData<Double> msumaGsCate;
 private LiveData<Double> msumaAllGs;

 private LiveData<Double> msumaGsCat2;

 private String categoria;

 public GsRepositorio(Application application) {
 GsDatabase database = GsDatabase.getDatabase(application);
 gsDao = database.gsDao();
 mAllGastos = gsDao.selectAllGastos();
 msumaGsCate = gsDao.sumGastosCategoria();
 msumaAllGs = gsDao.sumAllGastos();
 msumaGsCat2 = gsDao.sumGsCat2(categoria);

 mgastosMes = gsDao.gastosMes();
 }

 public LiveData<List<Gasto>> selectAllGastos() {
 return mAllGastos;
 }
}
```

```

 public LiveData<Double> sumaGastosCategoria() {return msumaGsCate;}

 public LiveData<Double> sumaAllGs() {return msumaAllGs; }

 public LiveData<List<Gasto>> gastosMes() {
 return mgastosMes;
 }

 public LiveData<Double> sumaGsCat2(String categoria){
 return gsDao.sumGsCat2(categoria);
 }

 // SQLite QUERIES definidas en DAO que se ejecuntan en el REPO.

 public void insert(Gasto gasto) {
 GsDatabase.databaseWriteExecutor.execute(()
 -> gsDao.insertGasto(gasto));
 }

 public void update(Gasto gasto) {
 GsDatabase.databaseWriteExecutor.execute(()
 -> gsDao.updateGasto(gasto));
 }

 public void delete(Gasto gasto) {
 GsDatabase.databaseWriteExecutor.execute(()
 -> gsDao.deleteGasto(gasto));
 }

 public void deleteAll() {
 GsDatabase.databaseWriteExecutor.execute(()
 -> gsDao.deleteAllGasto());
 }
}

```

## Model

```

package com.example.paltapp.model;

import android.app.Application;

import androidx.annotation.NonNull;
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.LiveData;

import com.example.paltapp.db.Gasto;
import com.example.paltapp.db.GsRepositorio;

import java.util.List;

public class GsViewModel extends AndroidViewModel {

```



```

//private List<Gasto> mselectAll;
private GsRepositorio mRepository;
private final LiveData<List<Gasto>> mAllGasto;
private LiveData<Double> mSumaGsCat;
private LiveData<Double> mSumAllGastos;

private LiveData<List<Gasto>> mSumMes;

private LiveData<Double> mSumaGsCat2;
public String categoria;

public GsViewModel(@NonNull Application application) {
 super(application);
 mRepository = new GsRepositorio(application);
 mAllGasto = mRepository.selectAllGastos();

 mSumaGsCat = mRepository.sumaGastosCategoria();
 mSumAllGastos = mRepository.sumaAllGs();

 mSumaGsCat2 = mRepository.sumaGsCat2(categoria);

 mSumMes = mRepository.gastosMes();
}

public LiveData<List<Gasto>> SumMes() {
 return mSumMes;
}

public LiveData<Double> getSumaGsCat2(String categoria) {
 return mRepository.sumaGsCat2(categoria);
}

public LiveData<List<Gasto>> getmAllGasto() {
 return mAllGasto;
}

public LiveData<Double> getmSumaGsCat() {
 return mSumaGsCat;
}

public LiveData<Double> SumAllGastos() {
 return mSumAllGastos;
}

public void insert(Gasto gasto){mRepository.insert(gasto);}

public void update(Gasto gasto){mRepository.update(gasto);}

public void delete(Gasto gasto){mRepository.delete(gasto);}

public void deleteAll(){mRepository.deleteAll();}

```

```
public void selectAll(){mRepository.selectAllGastos();}

}
```

Bueno, tratemos de desenmarañar este enredo. Tomemos como guía este esquema:

```
Dao ==> Repositorio ==> Model
```

Y esta query, para seguir el recorrido:

```
SELECT * FROM tabla_gastos
```

## DAO

```
@Query("SELECT * FROM tabla_gastos")
LiveData<List<Gasto>> selectAllGastos();
```

## REPOSITORIO

```
// public class GsRepositorio {
// private GsDao gsDao;
// private LiveData<List<Gasto>> mAllGastos;
// /*...*/

// public GsRepositorio(Application application) {
// GsDatabase database = GsDatabase.getDatabase(application);
// gsDao = database.gsDao();

// mAllGastos = gsDao.selectAllGastos();

// }

// public LiveData<List<Gasto>> selectAllGastos() {return mAllGastos;
// }
```

## MODEL

```
//public class GsViewModel extends AndroidViewModel {
// private final LiveData<List<Gasto>> mAllGasto;
// /* ... */
```

```
// public GsViewModel(@NonNull Application application) {
// super(application);
// mRepository = new GsRepositorio(application);
// mAllGasto = mRepository.selectAllGastos();}

public LiveData<List<Gasto>> getmAllGasto() {
 return mAllGasto;
}
```

Como podemos ver, partimos desde el Dao, lo pasamos al repositorio y finalmente al Model.

Dao ==> Repositorio ==> Model

Dao

```
@Query("SELECT * FROM tabla_gastos")
LiveData<List<Gasto>> selectAllGastos();
```



Repositorio

```
public LiveData<List<Gasto>> selectAllGastos() {return mAllGastos;}
```



Model

```
public LiveData<List<Gasto>> getmAllGasto() {return mAllGasto;}
```

Pero para para!, y que hago con tantas capas de abstracción??. De que me sirve?. Calmate un poco, que en el próximo apartado vamos a ver cuando y como se usa.

RecyclerView

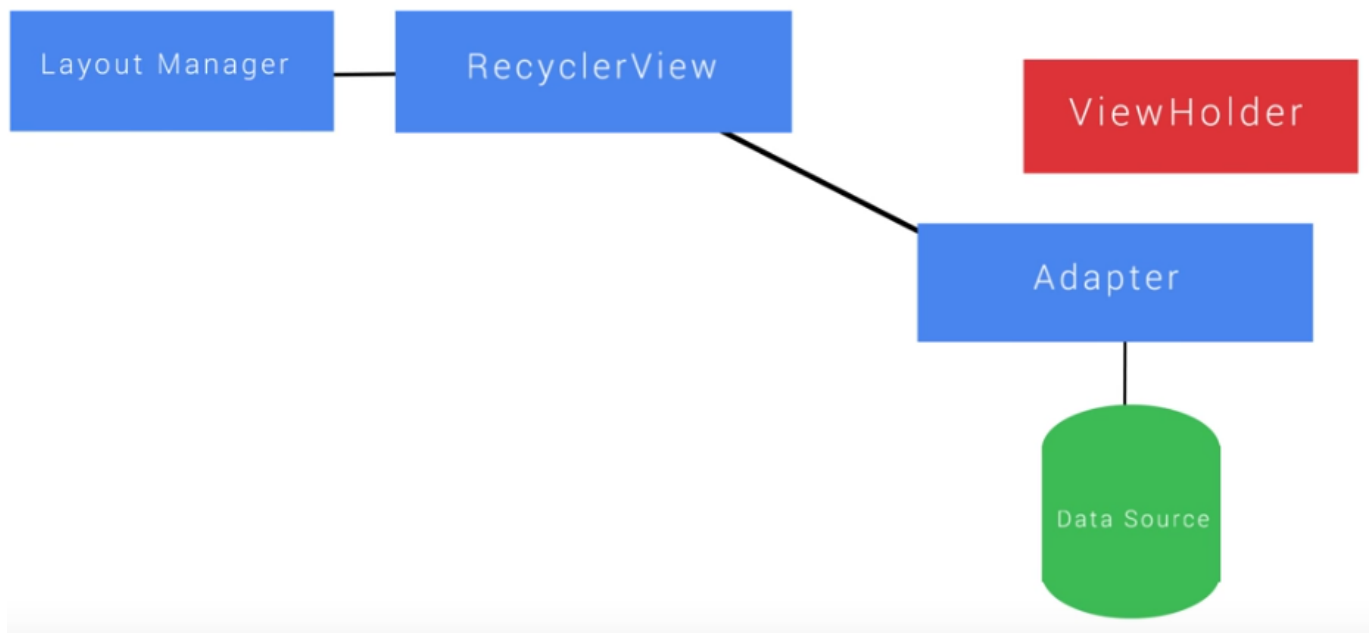
## Cómo crear listas dinámicas con RecyclerView



RecyclerView facilita que se muestren de manera eficiente grandes conjuntos de datos. Tú proporcionas los datos y defines el aspecto de cada elemento, y la biblioteca RecyclerView creará los elementos de forma dinámica cuando se los necesite.

Como su nombre lo indica, RecyclerView *recicla* esos elementos individuales. Cuando un elemento se desplaza fuera de la pantalla, RecyclerView no destruye su vista. En cambio, reutiliza la vista para los elementos nuevos que se desplazaron y ahora se muestran en pantalla. Esto mejora en gran medida el rendimiento y la capacidad de respuesta de tu app y reduce el consumo de energía.

Bueno, ya sabemos que es, veamos un diagrama que nos explique como se implementa:



Listo, y ahora qué?.

Ahora veamos como se implementó.

## ViewHolder

```
package com.example.paltapp.holder;
/*...*/

public class GsRecyclerViewHolder extends RecyclerView.ViewHolder {
 private TextView textView_categoria, textView_monto;
 private ImageView imageView_categoria;
 private NumberFormat format = NumberFormat.getCurrencyInstance();

 public GsRecyclerViewHolder(@NonNull View itemView) {
 super(itemView);
 textView_categoria = itemView.findViewById(R.id.textView_categoria_list);
 textView_monto = itemView.findViewById(R.id.textView_monto_item);
 imageView_categoria = itemView.findViewById(R.id.image_categoria);
 }

 public void bindCategoria(String cate){
 textView_categoria.setText(cate);}

 public void bindMonto(Double amount){
 textView_monto.setText(format.format(amount)); }

 //método que cambia el ícono en función a la caegoria

 public void bindImage(String imagenCatgeoria){
 switch (imagenCatgeoria){
 case "Compras":
```

```

imageView_categoria.setImageResource(R.drawable.ic_baseline_shopping_cart_24);
 break;
 case "Hogar":// transporte, ocio, salud, otros ingreso

imageView_categoria.setImageResource(R.drawable.ic_round_home_work_24);
 break;
 case "Transporte":

imageView_categoria.setImageResource(R.drawable.ic_baseline_directions_transit_24)
;
 break;
 case "Ocio":

imageView_categoria.setImageResource(R.drawable.ic_baseline_videogame_asset_24);
 break;
 case "Salud":

imageView_categoria.setImageResource(R.drawable.ic_round_local_hospital_24);
 break;
 case "Otros":
 imageView_categoria.setImageResource(R.drawable.ic_avocado_24);
 break;
 default:
 imageView_categoria.setImageResource(R.drawable.ic_avocado_24);
 break;

 }

}

public static GsRecyclerViewHolder create(ViewGroup parent) {
 View view = LayoutInflater.from(parent.getContext())
 .inflate(R.layout.row_item, parent, false);
 return new GsRecyclerViewHolder(view);
}
}

```

Acá hay dos cosas a destacar:

```

public void bindCategoria(String cate){
 textView_categoria.setText(cate);}

public void bindMonto(Double amount){
 textView_monto.setText(format.format(amount)); }

```

**Te acordás de las capas de abstracción? Bueno, este es otro paso más para llegar a usarlas finalmente. Si otra capa más, pero es la última, te lo prometo.**

Aca hacemos un *bind*, una unión entre la UI, en este caso un textView, y definimos el texto con un setText en función al elemento de la lista.

## Adapter

```
package com.example.paltapp.adapter;

/*...*/

public class GastosListAdapter extends ListAdapter<Gasto, GsRecyclerViewHolder> {

 public GastosListAdapter(@NonNull DiffUtil.ItemCallback<Gasto> diffCallback)
 {
 super(diffCallback);
 }

 @NonNull

 @Override
 public GsRecyclerViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int
viewType) {
 return GsRecyclerViewHolder.create(parent);
 }

 @Override
 public void onBindViewHolder(@NonNull GsRecyclerViewHolder holder, int
position) {
 Gasto current = getItem(position);
 holder.bindCategoria(current.getNombre());
 holder.bindMonto(current.getMonto());
 holder.bindImage(current.getCategoria());
 }

 public static class GsDiff extends DiffUtil.ItemCallback<Gasto>{

 @Override
 public boolean areItemsTheSame(@NonNull Gasto oldItem, @NonNull Gasto
newItem) {
 return oldItem == newItem;
 }

 @Override
 public boolean areContentsTheSame(@NonNull Gasto oldItem, @NonNull Gasto
newItem) {
 return oldItem.getId() == newItem.getId();
 }
 }
}
```

Destacados:

```
public class GastosListAdapter extends ListAdapter<Gasto, GsRecyclerViewHolder>
```

Este adaptador extiende de la clase List adapter que requiere una clase, en nuestro caso la `@Entity` creada en `Gasto.class` junto a un ViewHolder, que lo vamos a explicar más adelante

```
@Override
public void onBindViewHolder(@NonNull GsRecyclerViewHolder holder, int
position) {
 Gasto current = getItem(position);
 holder.bindCategoria(current.getNombre());
 holder.bindMonto(current.getMonto());
 holder.bindImage(current.getCategoria());
}
```

**Te acordás de las capas de abstracción? Bueno, este es otro paso más para llegar a usarlas finalmente. Si, te mentí, te dije que era la última, pero no lo es. Que querés que le haga?, anda a quejarte a Google.**

En definitiva lo que hacemos acá es hacer un binding, una union entre nuestro viewmodel y la entidad con nuestro holder. Recordá que este es un **adaptador**, y como un enchufe, ponemos un un adaptador entre el cable y el toma corriente. En esta analogía, el cable es ViewModel y el toma corriente el ViewHolder. Primero hacemos referencia al Gasto. Recordá que está definido como

```
public LiveData<List<Gasto>> getmAllGasto()
```

Entonces cuando definimos

```
Gasto current = getItem(position);
```

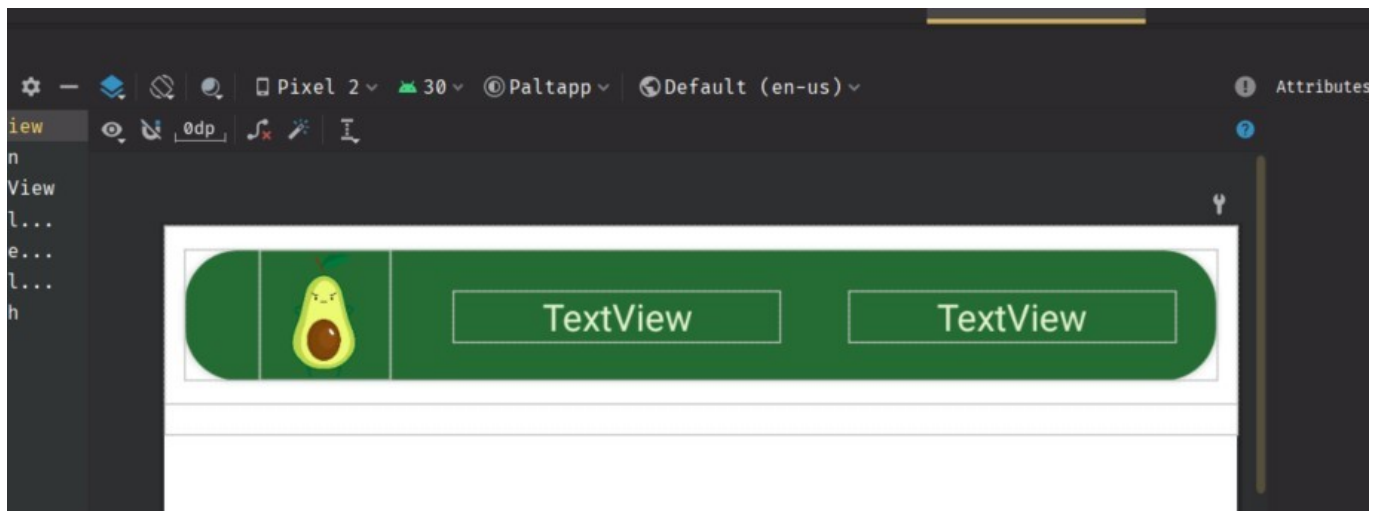
estamos haciendo referenica al Gasto que esta dentro de:

```
LiveData<List<Gasto>>
```

Si lo pensamos como un for loop de un array, comienza por el primer elemento, que sería `getItem(position)`, o sea la posición 0. Entonces current es el Gasto por el cual pasa el loop, en la primera iteración como dijimos antes será 0. Y luego obetenemos los datos con los getters que definimos en la `@Entity` Gasto.



Listo....No entendí nada, tanto quilombo, como queda la cosa??. Tranqui, te lo muestro en unas imagenes



Ves que hay dos TextView, estan serán obtenidas y seteadas en: **Adapter**

```
@Override
public void onBindViewHolder(@NonNull GsRecyclerViewHolder holder, int
position) {
 Gasto current = getItem(position);
 holder.bindCategoria(current.getNombre());
 holder.bindMonto(current.getMonto());
 holder.bindImage(current.getCategoria());
}
```

## Holder

```
public void bindCategoria(String cate){
 textView_categoria.setText(cate);}

public void bindMonto(Double amount){
 textView_monto.setText(format.format(amount)); }
```

Pero pero, y la app, como quedó?. Tranqui panki, queda otra cosa más! Viste que habia que separar la UI de la logica, veamos la UI.

## Activity

```
package com.example.paltapp.ui;

public class listado extends AppCompatActivity {
```

```

private GsViewModel mGsViewModel;
private GastosListAdapter adapter;
RecyclerView recyclerView;

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

 //Conexión con el ViewModel
 mGsViewModel = new ViewModelProvider(this).get(GsViewModel.class);
 //Conexión con el RecyclerView
 recyclerView = findViewById(R.id.recyclerview);

 adapter = new GastosListAdapter(new GastosListAdapter.GsDiff());

 recyclerView.setAdapter(adapter);

 recyclerView.setLayoutManager(new LinearLayoutManager(this));

 mGsViewModel.getMAllGasto().observe(this, gastos -> {
 adapter.submitList(gastos);
 });
}
}

```

Hacemos las conexiones con el ViewModel y con el recyclerView, y por supuesto un observer que va a "observar" a la LiveData.

**Y si, al fin! TODAS ESAS CAPAS DE ABSTRACCION PARA LLEGAR A ESTO:**

```

mGsViewModel.getMAllGasto().observe(this, gastos -> {
 adapter.submitList(gastos);
});

```

Después quedaría así:



## Tus ultimos movimientos



The Withcer 3

\$5,500.00



Netflix

\$7,500.00



Arriendo Julio

\$25,000.00



Compra Lider

\$7,200.00



Electricidad

\$15,000.00



Arriendo Junio

\$250,000.00

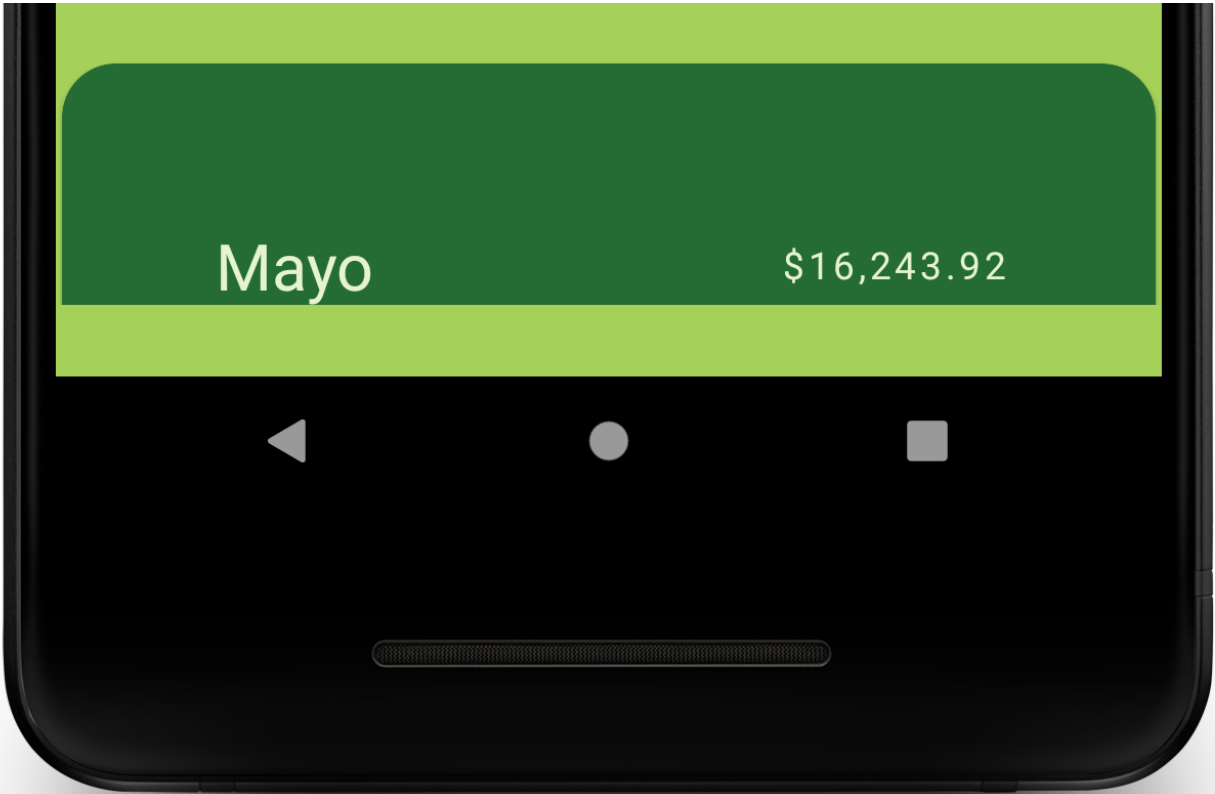


lumbo

\$22,252.45

RESULTADO







todo el código acá: [https://github.com/cavigna/Android\\_Development/tree/main/PaltApp](https://github.com/cavigna/Android_Development/tree/main/PaltApp)