

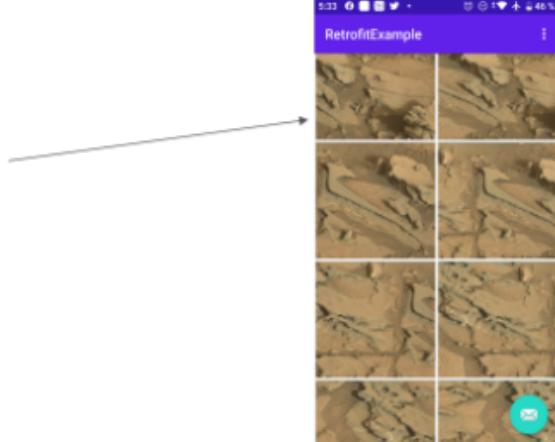
## Evidencia Semana 15

### Resumen Evidencia día 2 semana 15

El día de hoy se comienza a desarrollar el siguiente ejercicio de consumo de Servicio Rest propuesto en clases.

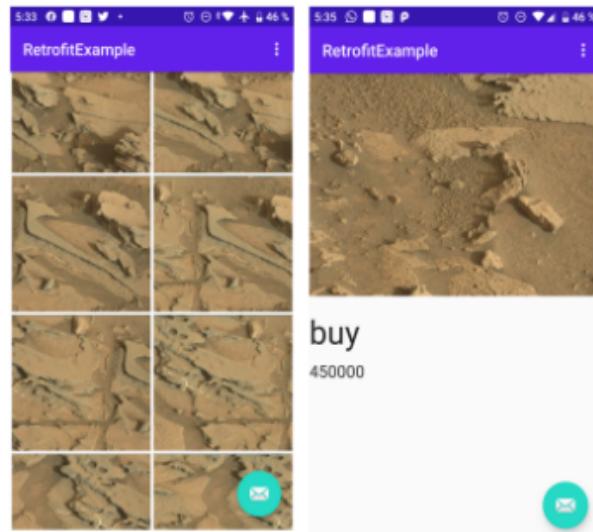
#### Ejercicio: Integrar Retrofit

1. Vamos a conectarnos a un servicio web y traernos los datos hacia nuestra app.
2. Retrofit manejará la conexión y a través de Gson se transformarán los datos a objeto.
3. Mostraremos el listado de objetos en una vista.
4. Tenemos un repositorio con el proyecto inicial, donde ya están construidas las vistas.



#### Una vez construida la app debería verse así:

1. Nos conectaremos a una API que nos va a entregar un listado de elementos.
2. Tendremos que analizar los elementos que nos envíaran.  
TIP: utiliza POSTMAN  
a. <https://android-kotlin-fun-mars-server.appspot.com/realestate>
3. Descarga el proyecto base desde el siguiente repositorio.



Las primeras actividades incluyen agregar las dependencias en el Gradle, para este ejercicio: Retrofit, Lycycle, Corrutinas, Room y Picasso y el ViewBinding.

Se crean las vistas, que para este ejercicio serían 2 fragments y se comienza a trabajar en la creación de la entidad que se convertirá en la base de datos que se genera a través del JSON.

Se crea el DAO y el Cliente de la BD y trabajamos con la siguiente URL:  
<https://android-kotlin-fun-mars-server.appspot.com/realestate> (no es necesaria la APIkey)

```

// 20211127204939
// https://android-kotlin-fun-mars-server.appspot.com/realestate

[
  {
    "price": 450000,
    "id": "424905",
    "type": "buy",
    "img_src": "http://mars.jpl.nasa.gov/msl-raw-images/msss/01000/mcam/1000MR0044631300503698E01_DXXX.jpg"
  },
  {
    "price": 800000,
    "id": "424906",
    "type": "rent",
    "img_src": "http://mars.jpl.nasa.gov/msl-raw-images/msss/01000/mcam/1000ML0044631300305227E03_DXXX.jpg"
  },
  {
    "price": 11000000,
    "id": "424907",
    "type": "rent",
    "img_src": "http://mars.jpl.nasa.gov/msl-raw-images/msss/01000/mcam/1000MR0044631290503689E01_DXXX.jpg"
  },
  {
    "price": 800000,
    "id": "424908",
    "type": "rent",
    "img_src": "http://mars.jpl.nasa.gov/msl-raw-images/msss/01000/mcam/1000ML0044631290305226E03_DXXX.jpg"
  },
  {
    "price": 12000000,
    "id": "424909",
    "type": "rent",
    "img_src": "http://mars.jpl.nasa.gov/msl-raw-images/msss/01000/mcam/1000MR0044631280503688E0B_DXXX.jpg"
  }
]

```

## código DAO

```

package com.example.terrenosmarte.dao
import androidx.lifecycle.LiveData
import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import com.example.terrenosmarte.model.Terrain

@Dao
public interface TerrainDAO {
    @Insert( onConflict = OnConflictStrategy.REPLACE )
    suspend fun insertAllTerrains( terrainList : List<Terrain> )
    @Query("SELECT id, img_src, type, price FROM terrain_table")
    fun getAllTerrainsFromDB():LiveData<List<Terrain>>
    @Query("SELECT id, img_src, type, price FROM terrain_table WHERE id = :id")
    fun getTerrainById( id : Long ) : LiveData<Terrain>
    @Query("SELECT id, img_src, type, price FROM terrain_table LIMIT 1")
    fun getFirstTerrain() : LiveData<Terrain>
}

```

## Código Cliente Retrofit

```

package com.example.terrenosmarte.cliente
import com.example.terrenosmarte.service.TerrainService
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory

class TerrainClient {
    companion object {

```

```

val BASE_URL = "https://android-kotlin-fun-mars-server.appspot.com/"
private val cliente = TerrainClient
fun getClient( url : String ) : TerrainService {
    val retrofit = Retrofit.Builder().baseUrl( url
        ).addConverterFactory(GsonConverterFactory.create()).build()
    return retrofit.create(TerrainService::class.java)
}
}
}
}

```

## Código Entidad

```

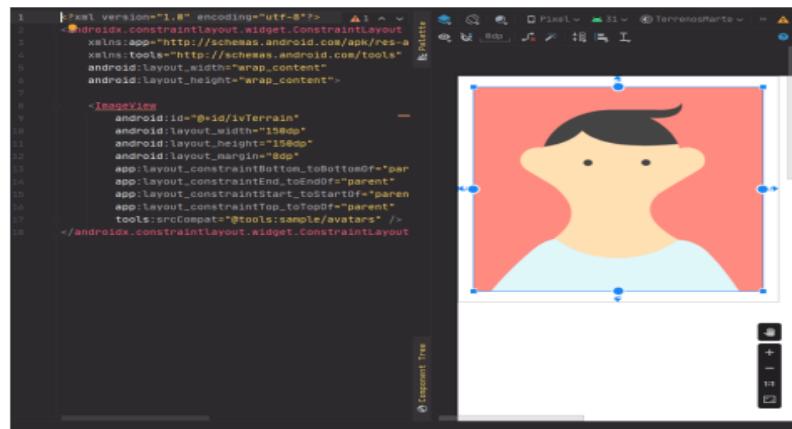
package com.example.terrenosmarte.model
import androidx.room.Entity
import androidx.room.PrimaryKey
    @Entity(tableName = "terrain_table")
    data class Terrain (
        @PrimaryKey val id: Long,
        val img_src: String,
        val price: Int,
        val type: String)

```

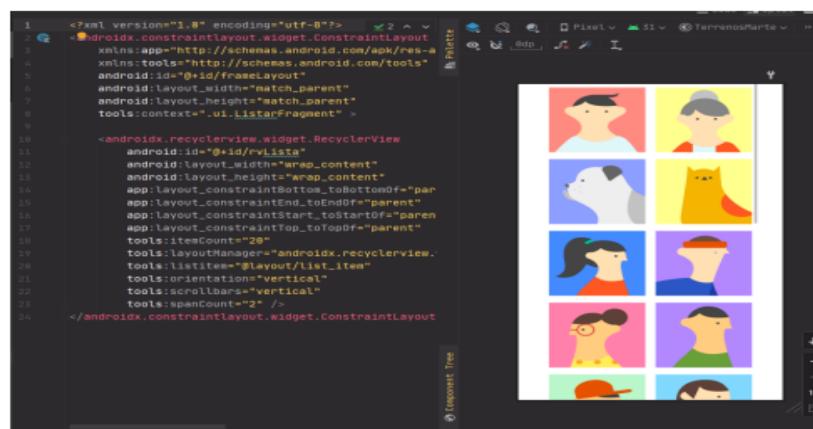
## Resumen Evidencia día 3 semana 15

Hoy se trabaja en la continuación del ejercicio del día anterior, se resolvieron dudas de algunos compañeros,y se trabaja en el diseño de las vistas y el DataBase.

Vista item\_layout



Vista Fragmento Listar



## Código DataBase

```
package com.example.terrenosmarte.room
import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import com.example.terrenosmarte.dao.TerrainDAO
import com.example.terrenosmarte.model.Terrain

private const val DATA_BASE_NAME = "mars_db"
@Database(entities = [Terrain::class], version= 3 )
public abstract class MarsDB : RoomDatabase()
{
    abstract fun getTerrainDao() : TerrainDAO
    companion object
    {
        @Volatile
        private var INSTANCE : MarsDB? = null
        fun getDatabase( context : Context) : MarsDB
        {
            val tempInstance = INSTANCE
            if ( tempInstance != null ){
                return tempInstance
            }
            synchronized(this){
                val instance = Room.databaseBuilder( context,
                    MarsDB::class.java,DATA_BASE_NAME).fallbackToDestructiveMigration().build()
                INSTANCE = instance
                return instance
            }
        }
    }
}
```

## Código Service

```
package com.example.terrenosmarte.service
import com.example.terrenosmarte.model.Terrain
import retrofit2.Call
import retrofit2.http.GET
interface TerrainService {
    @GET("realestate")
    fun getTerrains() : Call<ArrayList<Terrain>>
}
```

## Resumen Evidencia día 4 semana 15

El día de hoy se realiza un trabajo de investigación y presentación sobre Hilos-Executor-Corutinas.

### Resumen

Un hilo es un único flujo secuencial de control dentro de un programa. En cualquier momento durante el tiempo de ejecución del hilo, hay un único punto de ejecución. Sin embargo, un hilo en sí mismo no es un programa; un hilo no puede ejecutarse por sí solo.

Más bien, se ejecuta dentro de un programa.

No maneja un solo hilo, sino varios hilos o subprocessos que se ejecutan al mismo tiempo y realizan diferentes tareas en un solo programa.

Un hilo es similar a un proceso real en que ambos tienen un único flujo secuencial de control. Sin embargo, un hilo se considera ligero porque se ejecuta dentro del contexto de un programa completo y aprovecha los recursos asignados para ese programa y el entorno del programa.

Como flujo secuencial de control, un hilo debe crear algunos de sus propios recursos dentro de un programa en ejecución. Por ejemplo, un hilo debe tener su propia pila de ejecución (stack) y contador de programa (register).

### **Executor**

Objeto que ejecuta tareas ejecutables enviadas. Esta interfaz proporciona una forma de desacoplar el envío de tareas de la lógica de cómo se ejecutará cada tarea. Normalmente se utiliza un Executor en lugar de crear hilos explícitamente.

### **Corrutinas**

Una corrutina es en sí misma un hilo liviano. Es decir, se vale de hilos o pool de hilos para ejecutar código de manera concurrente (destino llegar a UI) y de forma muy eficiente optimizando el uso de recursos.

La creación de muchos hilos atenta contra el rendimiento en algunas situaciones, ya que estos subprocessos pueden imponer una carga excesiva sobre el “garbage collector” y el “Object Allocation” de la JVM.

### **Ventajas**

**Desarrollador** → Implementación de código asincrónico con un estilo idéntico al código sincrónico.

**Rendimiento** → permiten la ejecución de miles y hasta millones de subprocessos concurrentemente con un uso de recursos eficiente.

**Implementación** → manipulación sencilla de los hilos de ejecución valiéndose de los Dispatchers.

**Versatilidad** → las corrutinas se pueden suspender y reanudar a mitad de la ejecución.

### **Elementos de las Corrutinas**

**Dispatcher:** Sistema que planifica la gestión de los trabajos y el inicio y fin de los procesos, estableciendo prioridades y asignando procesadores a cada tarea.

El contexto determina en qué hilo se ejecutarán las corrutinas. Hay cuatro opciones:

**Dispatchers.Default:** para trabajo intenso de CPU (por ejemplo, ordenar una lista grande)

**Dispatchers.Main:** Se limita al subprocesso principal que opera con objetos de UI. Por lo general, estos despachadores son de un solo subprocesso. Para trabajar con esto, se debe agregar la dependencia al proyecto:

**org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.9**

**Dispatcher.Unconfined:** ejecuta corrutinas sin limitarlas a ningún hilo específico

**Dispatchers.IO:** para trabajos pesados de IO (por ejemplo, consultas de bases de datos de larga duración).

## Resumen Evidencia Dia 5 semana 15

Resumen Presentación de Ignacio

Código sincrónico

Una función sincrónica, bloqueará el hilo con el cual esté trabajando. Suponiendo que lo haga del hilo de la UI, el usuario experimentará un ANR hasta que termine la ejecución.

Código Asíncrono

Aunque sospechosamente se parecen las funciones, esta posee una sutil diferencia, el modificador suspend. Una corriente puede suspender la ejecución sin bloquear un hilo e incluso moverse a otro hilo. A esto se le llama el punto de suspensión.

Anatomía de una corriente

Existen 3 Componentes claves:

1. Scope
2. Context
3. Builder

**Scope:** Las Corrientes siguen un patrón de diseño llamado Concurrencia Estructurada, implicando que estas solo pueden ser ejecutadas en un Scope determinado definiendo el ciclo de vida de la misma. Por ende el Scope nos permitirá:

- Cancelar las Corrientes
- Definir su ciclo de vida.
- Notificar Errores.

**Context:** Una Corriente siempre se ejecuta dentro de un contexto y este a su vez contiene un dispatcher (despachante), que indica qué subprocesso se utilizará. Estos pueden ser:

- Dispatchers. Main
- Dispatchers. IO
- Dispatchers. Default

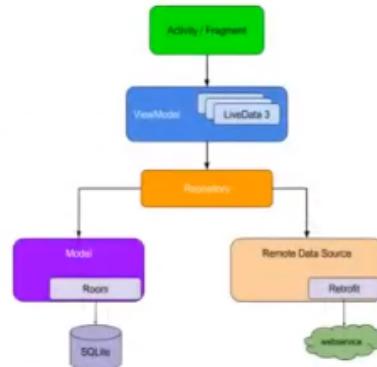
**Builder:** Launch/Async

## Lección 12, Consolidación: Retrofit + Room+MVVM

Repaso y revisión de conceptos MVVM, Retrofit con corrientes, LiveData, ViewModel.

### Ejercicio de consolidación (Requerimientos generales)

1. El proyecto debe estar en un repositorio en la plataforma GitHub.
2. Debe construir el proyecto con base en los requerimientos específicos solicitados, pero puede añadir su capa de customización.
3. Está considerado construirlo en horas de clases, para responder las consultas, pero si trabaja fuera del horario no existe problema.
4. En cada clase debe subir un commit y push hasta donde haya avanzado.

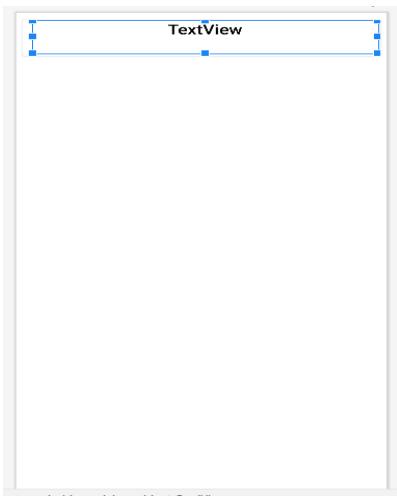


### Ejercicio de consolidación (Requerimientos generales)

6. Al hacer click sobre un elemento, debe mostrar un nuevo Fragmento con otro RecyclerView mostrando imágenes.
7. Al hacer un onLongClick sobre un elemento, debe guardar el objeto en la persistencia de datos como favoritos.
8. Debe haber un fragmento que muestre los objetos favoritos seleccionados en un RecyclerView. (Implementar un fab o menu).
9. En el fragmento de favoritos, debe ser posible eliminar un elemento o todos los elementos.

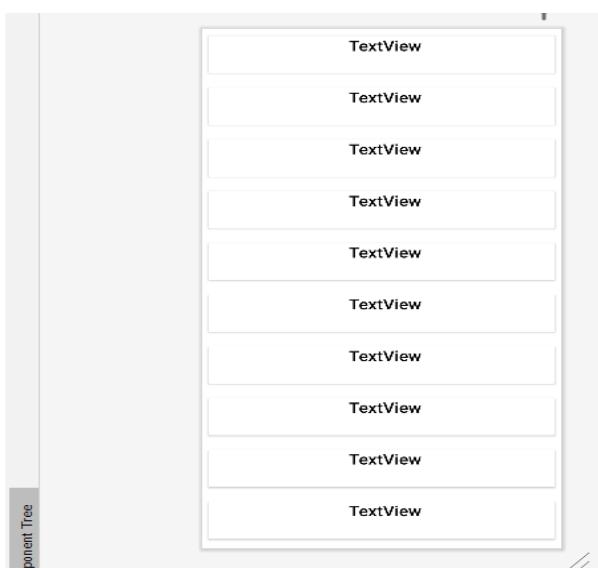
Se trabaja sobre la misma app de la Api de perros, pero ahora contendrá una actividad, que mostrará un recyclerView por razas, que al hacer clic en el ítem, lleve a una segunda actividad que muestre las imágenes y una tercera actividad con la lista de los favoritos guardados.

se crean el item\_layout para las razas y se une al recyclerView del main Activity:



```
        android:layout_height="match_parent"
        tools:context=".MainActivity"

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/rvListaRazas"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:listitem="@layout/raza_item_layout"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```



Se agrega una nuevo get de consulta en el ApiService

```
interface MascotaService {

    @GET("breed/{raza}/images")
    fun getMascotas(@Path("raza")raza:String): Call<MascotaJson>

    @GET("breeds/list")
    fun getRazas(): Call<RazaJson>
}
```

Se crea el modelo RazaModel con json, la entidad y un nuevo DAO, RazaDao:

```
data class RazaJson(  
    val message: List<String>  
)  
@Entity  
(tableName = "raza")  
class RazaModel(id:Int,var descripcion:String) {  
  
    @PrimaryKey  
(autoGenerate = false)  
    var id:Int = id  
}  
  
@Dao  
interface RazaDao {  
  
    @Insert(onConflict = REPLACE)  
    fun agregar(razaModel: RazaModel)  
  
    @Insert(onConflict = REPLACE)  
    fun agregarAll(razas: List<RazaModel>)  
  
    @Query("select id,descripcion from raza")  
    fun listar() : LiveData<List<RazaModel>>  
}
```

Reflexión: Los conceptos y ejercicios vistos durante esta semana, si bien aún son un poco difíciles de integrar a mi conocimiento, se debe repasar los videos de clases anteriores y practicar con distintas APIs, las llamadas y respuestas.