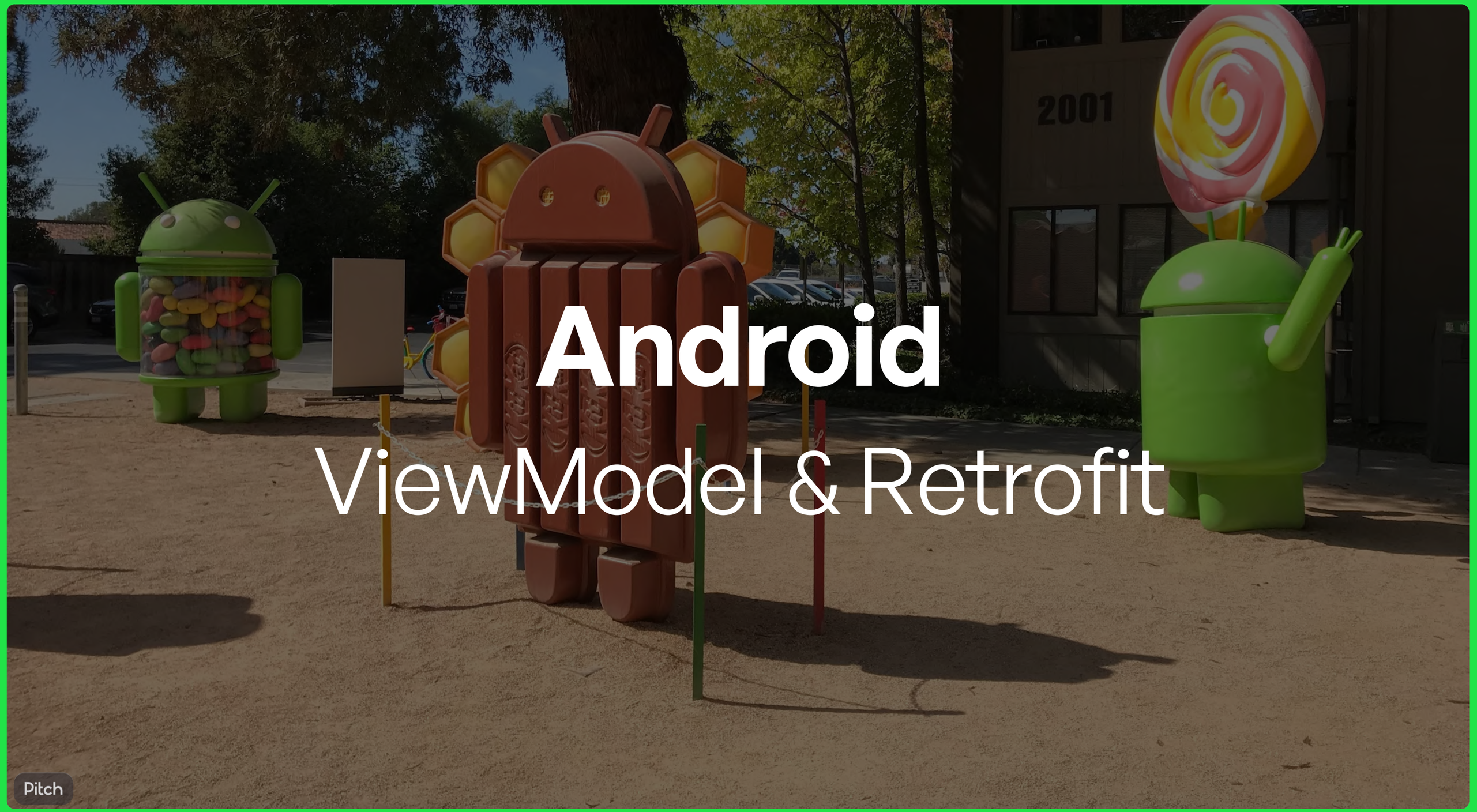
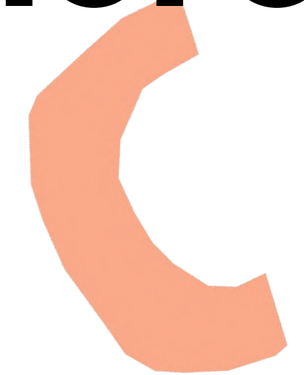
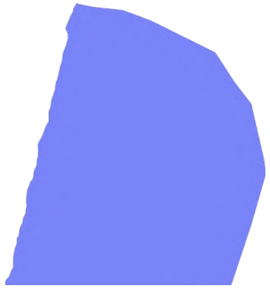


Android

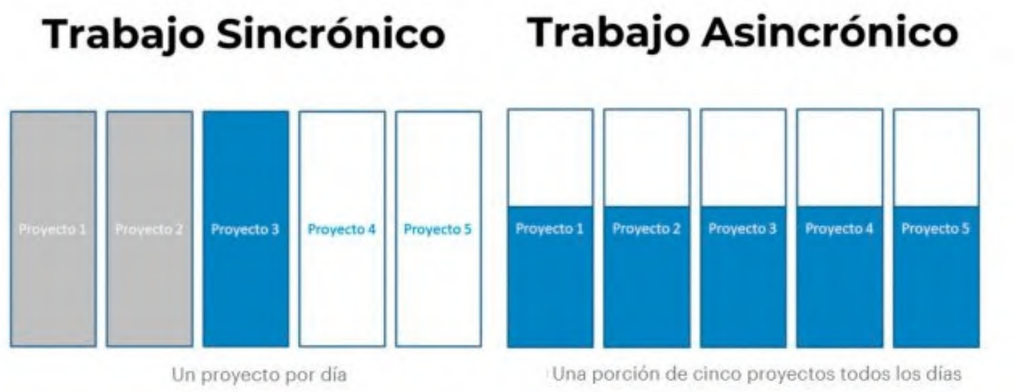
ViewModel & Retrofit

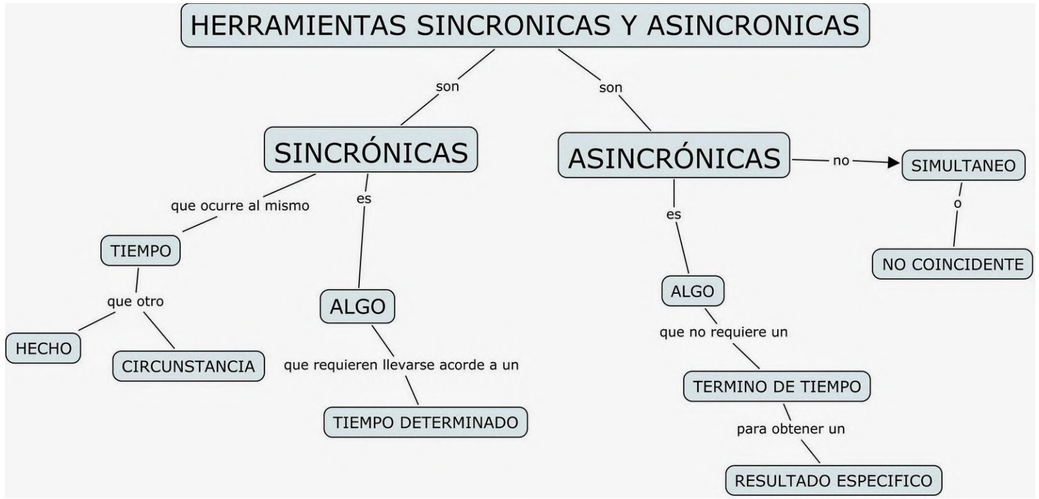
Sincronía y Asincronía



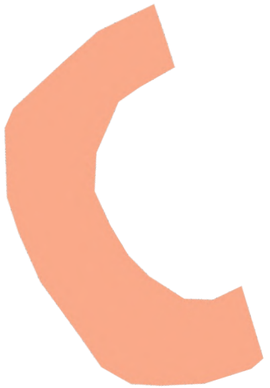


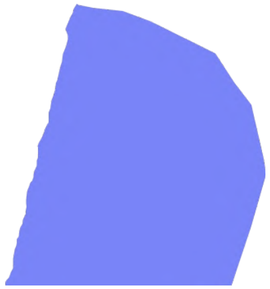
Definición de sincronia y asincronia





Programación Reactiva





Programación Reactiva

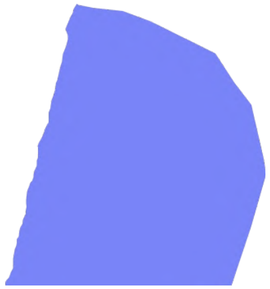
En la programación clásica, todo se basa en acciones que nosotros realizamos activamente. Lo que hacemos en cada caso es ir y preguntarle «oye, ¿qué datos tienes para mí ahora mismo?

Si quiero algo:
- Yo lo pido
- Yo lo busco
- Yo lo traigo y lo trato
- Yo lo muestro

En la programación reactiva, le damos la vuelta a la idea. La forma de trabajar sería «estoy interesado en tus datos, cada vez que haya un cambio infórmame».

Si quiero algo:
- Yo lo pido y espero a que me lo traigan
- Alguien se encarga de ir buscarlo
- Alguien se encarga de traelo y tratarlo
- Alguien se encarga de entregarmelo

ViewModel



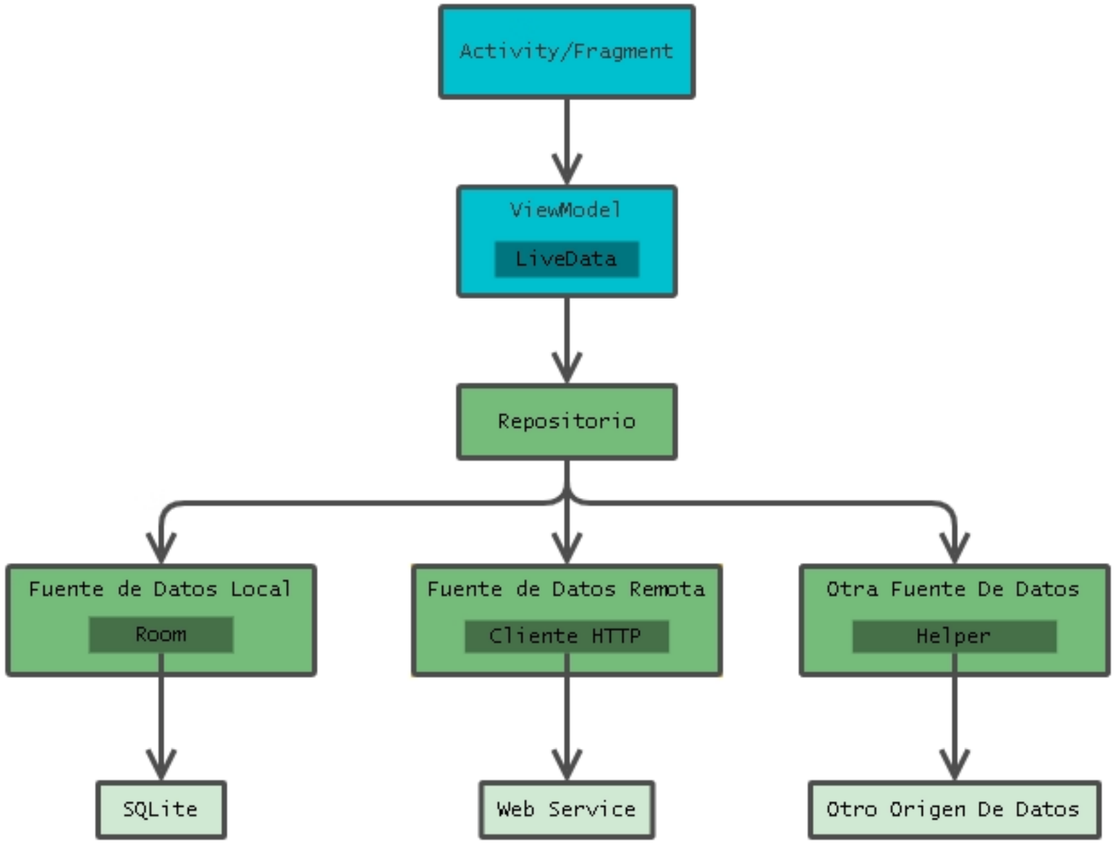
ViewModel

El componente **ViewModel** de Android cumple dos funciones: se encarga de la preparación y administración de los datos relacionados con la vista y maneja la comunicación de la vista con el resto de la aplicación.

Desde el punto de vista de arquitectura la clase ViewModel está diseñada como una plantilla para crear modelos de vista en el patrón MVVM.

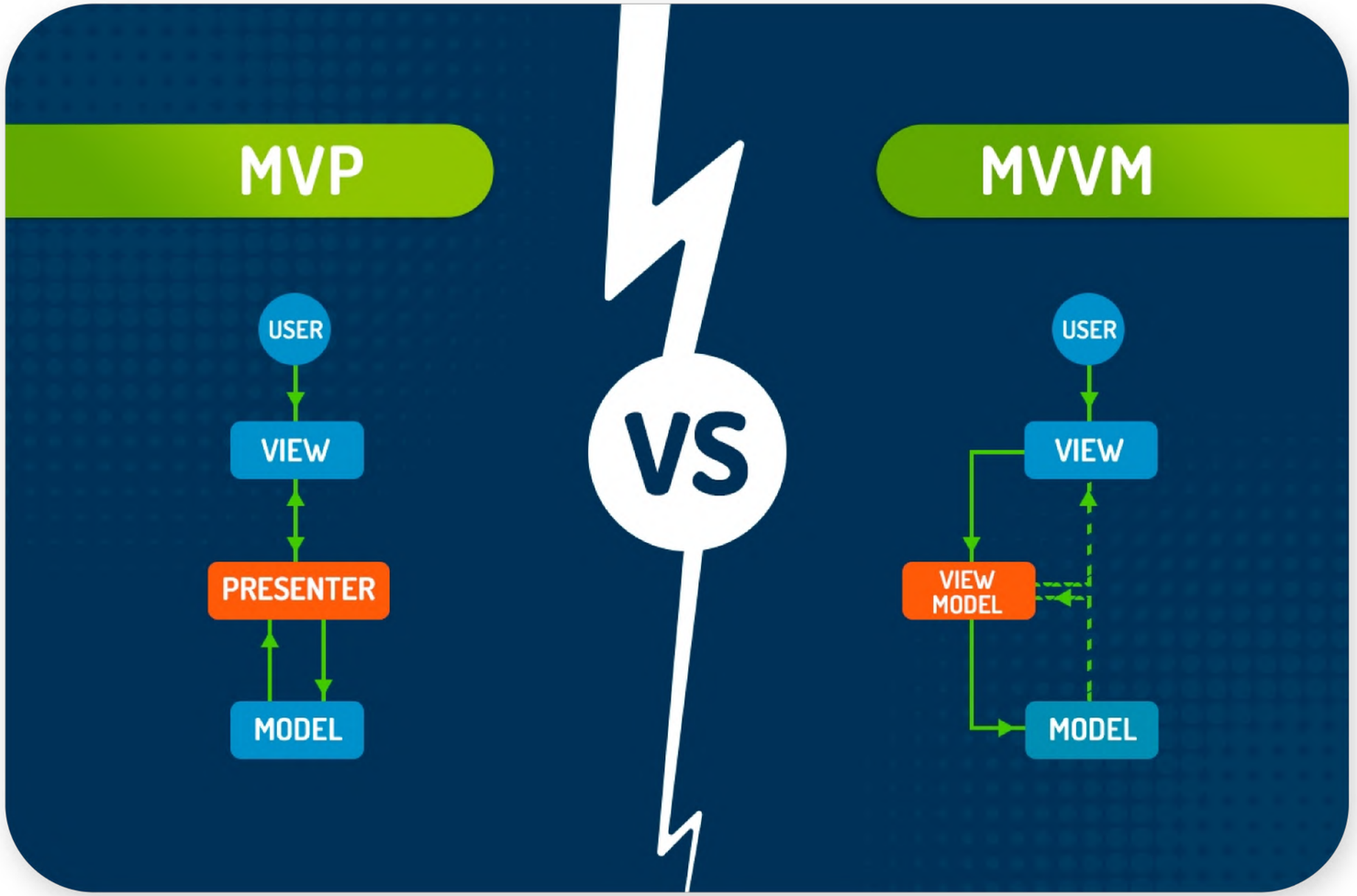
¿Por qué usarlo?
Porque mejora el testing y la eficiencia de mantenimiento de tus casos de uso al aislar responsabilidades de tus controladores de UI (actividades y fragmentos).

Cómo funciona en cambios de configuración?
Asegura el guardado del estado en tu UI (Scope) cuando hay cambios de configuración como una rotación de pantalla. Ya que cuando una actividad es recreada el framework reconecta automáticamente al ViewModel a la instancia.



https://developer.android.com/topic/libraries/architecture/viewmodel

Arquitectura de una app

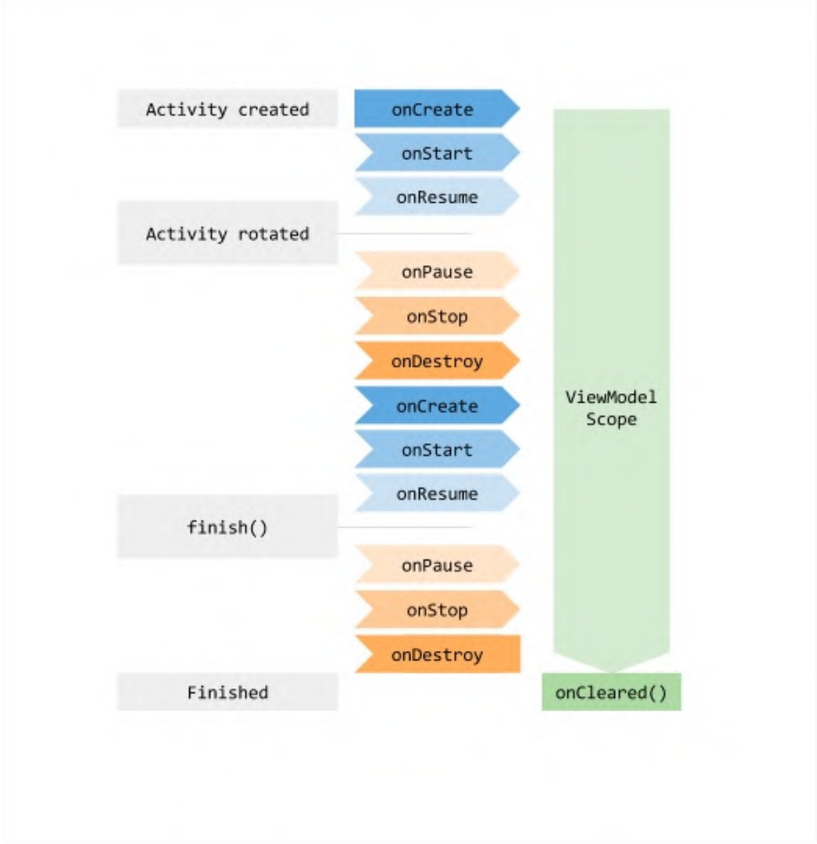


Arquitectura de una app

La arquitectura te orienta para asignar responsabilidades en tu app, entre las clases.
Una arquitectura de app bien diseñada te ayuda a escalar tu app y a extenderla con funciones adicionales en el futuro.
También facilita la colaboración.

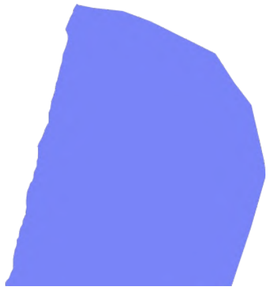
Los principios arquitectura más comunes son la separación de problemas y el control de la UI a partir de un modelo.

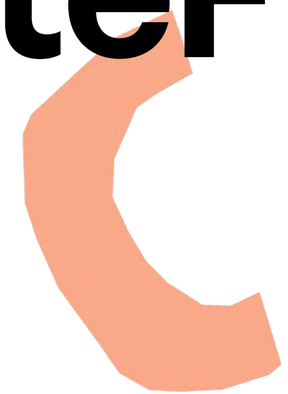
Responsabilidades del **fragment/activity**	Responsabilidades del **ViewModel**
Las actividades y los fragmentos son responsables de dibujar vistas y datos en la pantalla y responder a los eventos del usuario.	El ViewModel es responsable de retener y procesar todos los datos necesarios para la IU. Nunca debes acceder a tu jerarquía de vistas (como un objeto de vinculación de vista) ni retener una referencia a la actividad o al fragmento.



https://developer.android.com/topic/libraries/architecture/viewmodel

LiveData vs StateFlow





LiveData y StateFlow

Cuando trabajamos con un ViewModel, podemos usar dos tipos de objetos para transmitir los datos a la vista.

LiveData

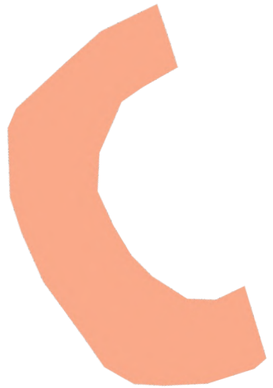
- que está integrado en Android y eso ayuda a evitar Leaks de memoria
- está programado con los ciclos de vida de las Activity/Fragment
- y maneja las rotaciones de pantalla para no tener que recargar toda la app

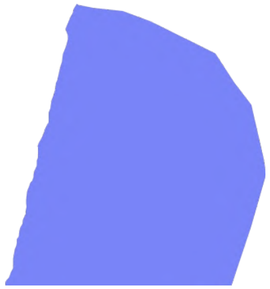
StateFlow

- que está integrado en Kotlin y permite emitir multiples valores secuencialmente
- es independiente del ciclo de vida de cada Activity/Fragment (en lo bueno y lo malo)

De un vistazo, en realidad son muy similares y puedes hacer prácticamente LO MISMO.
Sin embargo, bajo el capó, los Flow son una API más rica y flexible que amplía las capacidades de LiveData, sin algunos de los inconvenientes.

Retrofit



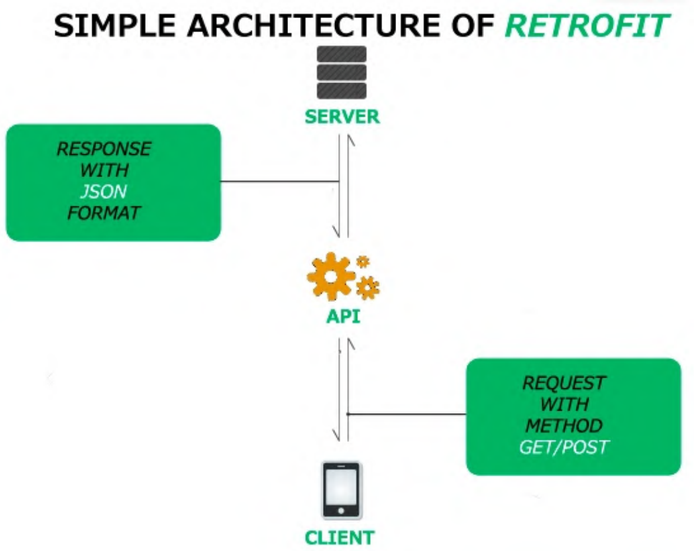


Retrofit

Para hacer peticiones a una API, podemos usar la Librería de Retrofit que nos ayuda a trabajar con peticiones de manera sencilla y rápida.

Una **API** es un conjunto de reglas que permite que diferentes programas se comuniquen entre sí. Utilizaremos JSON como "lenguaje" de comunicación entre servidores y la app.

Retrofit es un cliente de servidores REST para Android y Java desarrollado por Square, muy simple y muy fácil de aprender. Permite hacer peticiones al servidor tipo: GET, POST, PUT, PATCH, DELETE y HEAD, y gestionar diferentes tipos de parámetros, paseando automáticamente la respuesta a un tipo de datos.

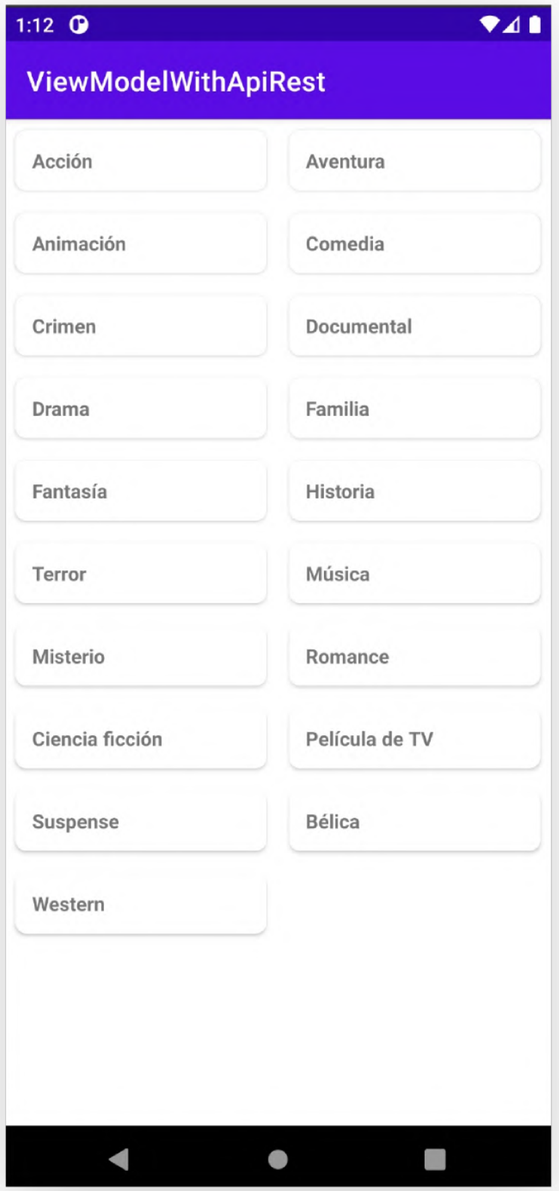


Práctica. 1/10

Vamos a utilizar la API de TheMovieDB para obtener generos de peliculas en una app. El resultado de la app será el siguiente.

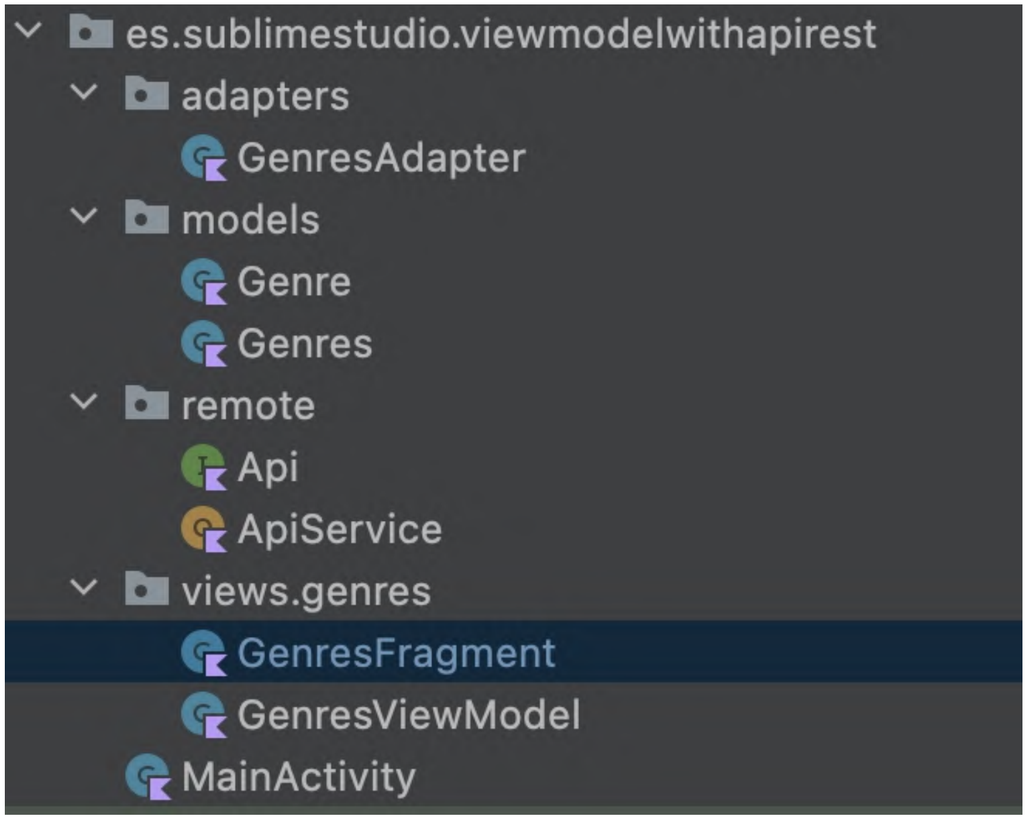
Empieza creando todos los elementos visuales, es decir:

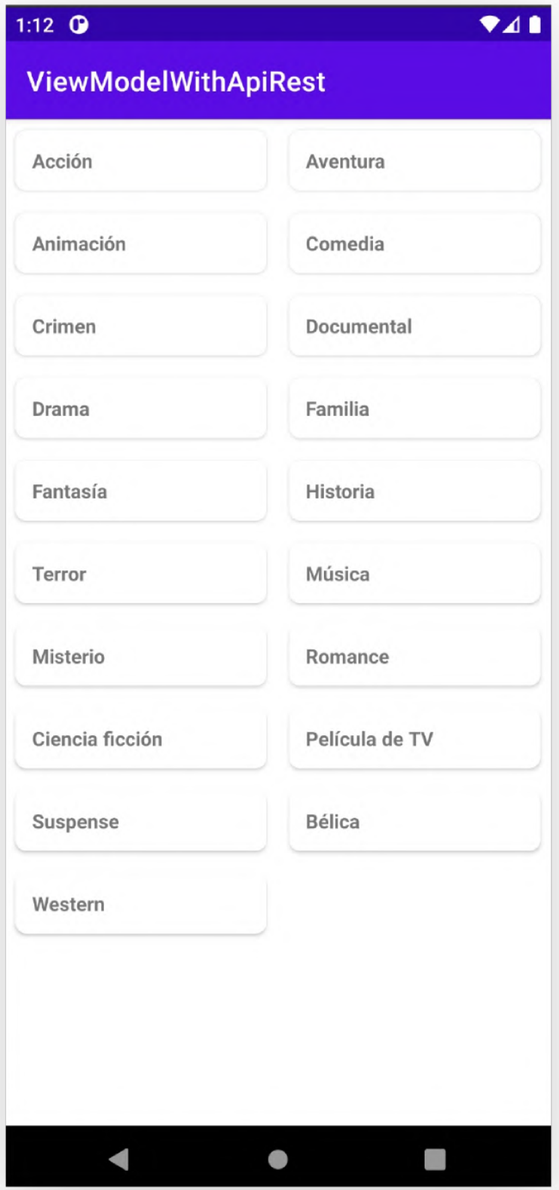
- Crea el elemento a repetir y llámalo item_genres.xml
- Crea un fragment que llevará el recyclerView y llámalo FragmentGenres (de momento no crees datos para el recycler ni el adapter, en un rato lo haremos)
- Crea el nav_graph y relacionalo con el activity
- Y ya que estás mete progressBar en el fragment



Práctica. 2/10

El siguiente paso es crear las carpetas donde irá todo nuestro código, para organizarlo mejor os propongo el siguiente formato.





Práctica. 3/10

Vamos a registrarnos en https://www.themoviedb.org/documentation/api y obtengamos una API Key para poder utilizar su API.

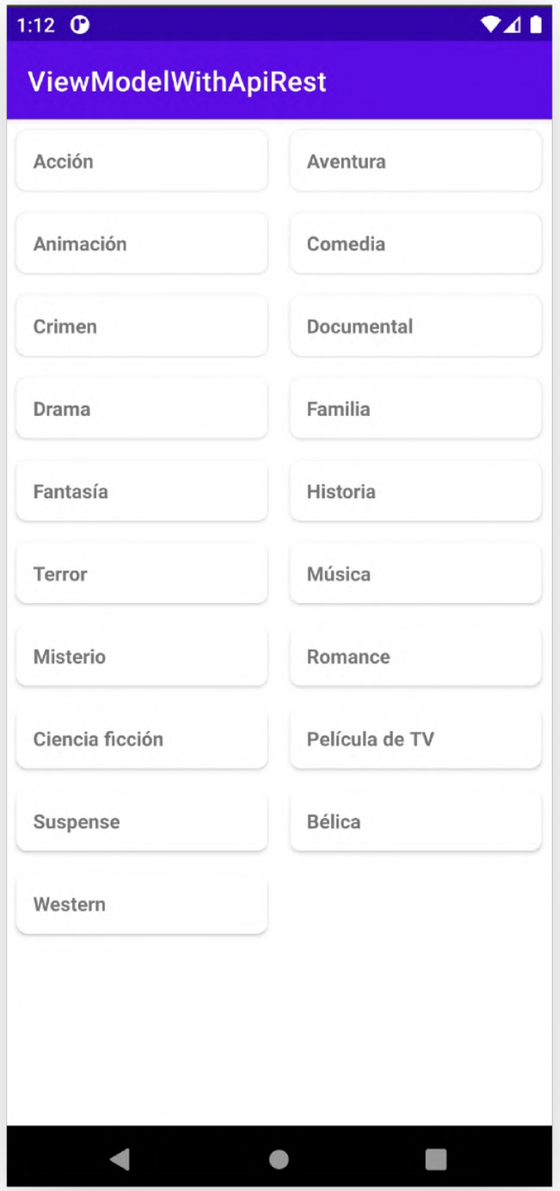
El endpoint que usaremos será este https://developers.themoviedb.org/3/genres/get-movie-list y la llamada desde nuestra app será algo así

```
https://api.themoviedb.org/3/genre/movie/list?api_key=
<<api_key>>&language=es-ES
```

Una vez sepamos lo que responde la API, vamos a crear el modelo Genre y Genres.

```
data class Genre(
    val id: Int,
    val name: String
)

data class Genres(
    val genres: List<Genre> = listOf()
)
```



Práctica. 4/10

Vamos a empezar a trabajar con Retrofit. Implementémosle en el proyecto.

```
// Retrofit
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
implementation "com.squareup.okhttp3:okhttp:5.0.0-alpha.2"
```

En la carpeta "remote" crea la siguiente clase llamada ApiService con el siguiente código:

```
object ApiService {
    lateinit var api: Api

    val URL = "https://api.themoviedb.org/3/"
    val api_key = "TUAPIQUEKEYPAJARO"
    val language = "es-ES"

    init {
        initService()
    }

    private fun initService() {
        val retrofit = Retrofit.Builder()
            .baseUrl(URL)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
        api = retrofit.create(Api::class.java)
    }

}
```

Práctica. 5/10

En la carpeta "remote" crea la siguiente clase llamada Api con el siguiente código:

```
interface Api {

    @GET("genre/movie/list")
    suspend fun getGenres(
        @Query("api_key") apikey: String = ApiService.api_key,
        @Query("language") language: String = ApiService.language
    ): Response<Genres>

}
```

Práctica. 6/10

Vamos a crear nuestro esperado ViewModel e intentamos entenderlo.

```
class GenresViewModel: ViewModel() {
    val genresList = MutableStateFlow(Genres())
    val loading = MutableStateFlow(false)

    fun getGenres() {
        loading.value = true
        viewModelScope.launch {
            val response = ApiService.api.getGenres()
            if (response.isSuccessful) {
                genresList.value = response.body() ?: Genres()
                Log.v("GenresVM", "Todo fenomenal en la petición de generos")
            } else {
                Log.v("GenresVM", "Error en la petición de generos ${response.toString()}")
            }

            loading.value = false
        }
    }

}
```

Práctica. 7/10

Ahora, en el Fragment llega lo bueno.
Para usar el viewModel en el fragment, tenemos que llamarlo así: `private val viewModel: GenresViewModel by activityViewModels()`

Para escuchar los cambios de estado de nuestras dos variables, ***loading*** y ***genresList*** tenemos que hacerlo con **collect** dentro de una **corrutina** en el ***onViewCreated***, así:

```
viewLifecycleOwner.lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.STARTED) {

        launch {
            viewModel.loading.collect { loading ->
                if (loading) {
                    progressBar.visibility = View.VISIBLE
                } else {
                    progressBar.visibility = View.GONE
                }
            }
        }

        launch {
            viewModel.genresList.collect {
                adapter.updateData(it.genres)
            }
        }

    }
}
```

Práctica. 8/10

Ahora, por último tocaría llamar a la función del viewModel que pide los datos: `viewModel.getGenres()`

Este será el aspecto de nuestro Fragment (más o menos) Nos falta una única cosa, crear nuestro Adapter.

```
class GenresFragment : Fragment() {
    private val viewModel: GenresViewModel by activityViewModels()
    private lateinit var progressBar: ProgressBar
    private lateinit var adapter: GenresAdapter

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_genres, container, false)
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        progressBar = view.findViewById(R.id.progressDialog)

        viewLifecycleOwner.lifecycleScope.launch {
            repeatOnLifecycle(Lifecycle.State.STARTED) {

                // podemos usar varios "launch"
                launch {
                    viewModel.loading.collect { loading ->
                        if (loading) {
                            progressBar.visibility = View.VISIBLE
                        } else {
                            progressBar.visibility = View.GONE
                        }
                    }
                }

                launch {
                    viewModel.genresList.collect {
                        adapter.updateData(it.genres)
                    }
                }

            }
        }

        adapter = GenresAdapter {
            // TODO navigate
        }

        val recyclerView = view.findViewById<RecyclerView>(R.id.recyclerview)
        recyclerView.adapter = adapter

        viewModel.getGenres()
    }

}
```

Práctica. 9/10

Nuestro es como cualquier adapter de un recyclerview, solo que tiene la peculiaridad de una función extra, que servirá para actualizar los datos en cuanto lleguen de la API.

```
class GenresAdapter(val onClick: (Genre) -> Unit): RecyclerView.Adapter<GenresAdapter.ViewHolder>() {

    var data = mutableListOf<Genre>()

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val view = LayoutInflater.from(parent.context).inflate(R.layout.item_genres, parent, false)
        return ViewHolder(view)
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        holder.bind(data[position])
    }

    override fun getItemCount(): Int {
        return data.size
    }

    fun updateData(genresData: List<Genre>) {
        data = genresData.toMutableList()
        notifyDataSetChanged()
    }

    inner class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        val genero = itemView.findViewById<TextView>(R.id.name)
        val card = itemView.findViewById<CardView>(R.id.card)

        fun bind(item: Genre) {
            genero.text = item.name
            card.setOnClickListener {
                Log.v("Pulso sobre", item.id.toString())
                onClick(item)
            }
        }
    }
}
```

Práctica. 10/10

Si quieres que te salga como a mi, el item_genres lo tengo así.

¡A disfrutar!

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.widget.CardView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="8dp"
    android:elevation="8dp"
    android:id="@+id/card"
    app:cardCornerRadius="8dp">

    <LinearLayout
        android:layout_width="match_parent"
        android:orientation="vertical"
        android:layout_height="wrap_content">
        <TextView
            android:id="@+id/name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_gravity="bottom"
            android:background="#D3FFFFFF"
            android:gravity="center_vertical"
            android:lines="1"
            android:maxLines="1"
            android:padding="12dp"
            android:textAppearance="@style/TextAppearance.AppCompat.Small"
            android:textStyle="bold"
            tools:text="@tools:sample/full_names" />
    </LinearLayout>

</androidx.cardview.widget.CardView>
```

¡Reto!

Si te has sentido impresionado por todo lo que hemos hecho y te atreves a hacer más, te invito a que hagas la pantalla donde se muestra el listado de películas según el género.


gracias
Phone
Contacts
Messages
Gmail
Sparkasse
Music
PayPal
YouTube
Translate
Pitch