JDMobile
jdccmobile@gmail.com
https://www.linkedin.com/in/josedcastro/
https://github.com/JoseD97

POKECOMBAT

Mayo del 2023

Este documento se trata de un resumen del proceso realizado hasta completar la app para que sirva, tanto a mí como a otros estudiantes de Android, a comprender mejor e implementar los recursos utilizados en este proyecto en otros. Cualquier crítica constructiva de mejora de código o de la app siempre será bien recibida 😉

Descripción

POKECOMBAT es una app donde deberás elegir tu Pokémon favorito y realizar combates con el objetivo de derrotar toda la Pokédex de manera seguida sin perder. ¿Podrás vencerlos a todos?

Todo el código en **Github**: https://github.com/jdccMobile/PokeCombat

Tecnologías

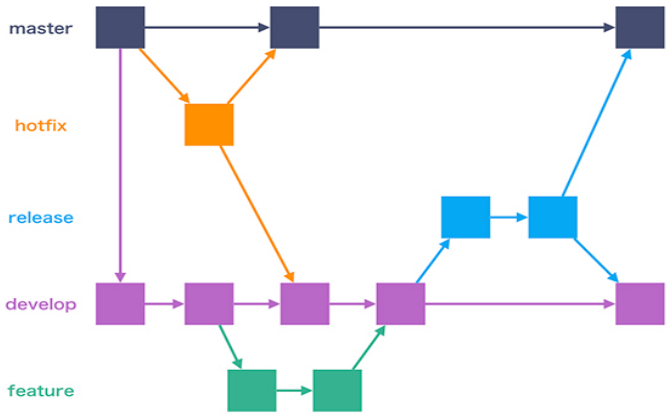
Esta app utiliza:

- Kotlin + Vistas XML
- GitFlow
- Clean Architecture + MVVM
- View binding
- Dagger Hilt
- RecyclerView
- Retrofit + Corrutinas
- DataStore Preferences
- Animaciones

1. GitFlow

GitFlow es un sistema de branching de manejo de ramas en Git.

La idea principal detrás de GitFlow es aislar el trabajo en diferentes tipos de *branches*, lo que le permite adaptarse al proceso colaborativo que necesita un equipo de desarrollo.



GitFlow está basado principalmente en dos *branches* que tienen una vida infinita:

- **Master**: contiene el código de producción.
- **Develop**: contiene el código de desarrollo.

Además de estos *branches* principales, durante el desarrollo se crean otros *branches* de apoyo que tienen una vida finita, es decir, solo existen mientras exista el desarrollo:

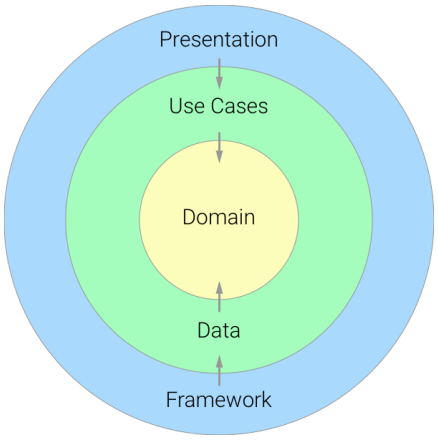
- **Feature**: se crea a partir de Develop para implementar una nueva funcionalidad. Cuando se termina la feature se *mergea* de nuevo en Develop.
- **Release**: se crea a partir de Develop para preparar una nueva versión del código para liberarla en producción. Al finalizar el desarrollo se *mergea* a develop y a master.
- **Hotfix**: se crea a partir de Master cuando es necesario corregir un error detectado en producción de manera urgente. Al finalizar el desarrollo se hace merge a develop y a master.

2. Clean Architecture + MVVM

Clean Architecture

Clean Architecture es una meta arquitectura que divide la aplicación en capas para desacoplar las diferentes partes del código de manera organizada:

- **Capa de presentación**: Es la capa que interactúa con la interfaz de usuario (*activities, fragments y views*).
- **Casos de uso o *interactors***: Son las acciones que el usuario puede desencadenar y pueden ser tareas activas (pulsar un botón) o acciones implícitas (la app navega a otra pantalla). Normalmente, estos casos de uso se realizan en otro hilo secundario, así evitamos problemas con el hilo de UI.
- **Capa de dominio**: Es la capa de modelo de negocio que son las reglas del negocio. La mayoría de apps Android pinta los resultados obtenidos desde una API por lo que principalmente tendremos en esta capa la solicitud y la persistencia de datos.
- **Capa de datos**: En esta capa se encuentra la definición abstracta de las diferentes fuentes de datos y la forma en que se debe utilizar. Se suele utilizar un patrón repositorio que para una solicitud decide dónde encontrar la información.
- **Capa de *framework***: En esta capa se interactúa con el *framework*.

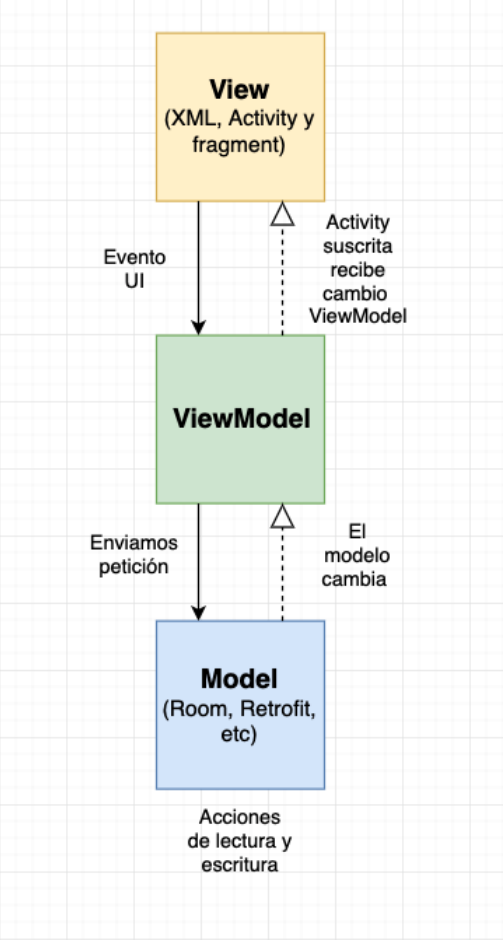


MVVM (Model-View-ViewModel)

Este patrón de arquitectura es la oficial elegida por Google. Organiza la app en tres módulos:

- ***Model***: Representa la parte de datos, tanto su formato como la forma de obtenerlos.
- ***View***: Es la parte de la UI de la app (*Activities* y *Fragments*). Esta capa ejecuta acciones de la UI pero no realizará acciones relacionadas con los datos, ya que se suscribirán al *ViewModel* y este les dirá como y que pintar.
- ***Viewmodel***: Se encarga de la conexión entre la vista y los datos.

El funcionamiento de MVVM es el siguiente:



En la ***View***, haremos que nuestra *Activity* se suscriba a nuestro *ViewModel*, a través de *live data*, que consiste en "conectarse" a través del patrón *observer* para que cuando haya un cambio en los datos la *Activity* se entere. Esto es importante, ya que la actividad solo se pintará cuando dicho *ViewModel* lo notifique. La Activity también tiene que controlar cuando se pulsa la pantalla para avisar al *ViewModel*.

El ***ViewModel*** cuando recibe un evento de la UI, tiene que devolver los datos que le solicita la *View*. Para ello, llamará al ***Model*** que irá al repositorio y realizará una llamada a Retrofit, Room o al servicio que tenga implementado la app y recibirá los datos en la respuesta. Esta respuesta se la pasará al ViewModel que a su vez notificará a la ***Activity*** del cambio en los datos para que lo muestre por pantalla.

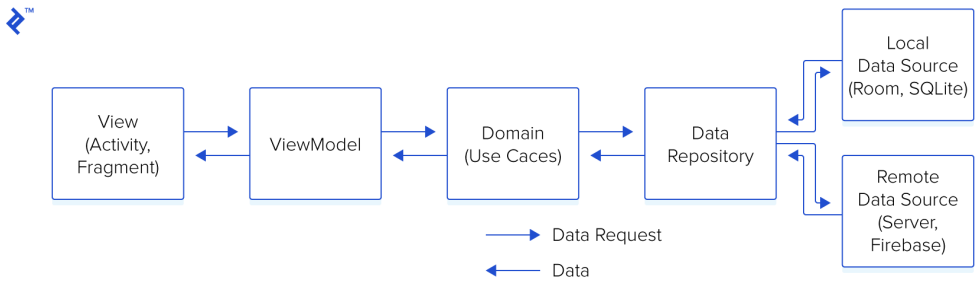
Clean Architecture + MVVM

En esta app se combina Clean Architecture con MVVM para organizar el código. El proyecto se divide en cuatro capas:

- **La capa de Presentación:** La capa de Presentación está formada por los Views y los ViewModels.
- **La capa de Dominio:** Esta capa contiene todos los casos de uso de la aplicación. A la hora de buscar información de la base de datos, no debemos nunca bloquear el hilo principal. Por tanto, este caso de uso estará en otro hilo y la respuesta la recibirá en el hilo principal. Los casos de usos son los mediadores entre la capa de datos y la presentación.
- **La capa de Datos:** Contiene el repositorio al que se conectan los casos de uso. A través de este repositorio se accederá a los servicios para obtener los datos correspondientes.
- **La capa *Util*:** Esta capa es opcional y se usa para almacenar utilidades que podemos usar en todo el proyecto

En resumen, el funcionamiento de la arquitectura utilizada es el siguiente:

La *View* pinta lo que recibe del *ViewModel* y le avisa cuando se produce un evento en la UI. Cuando el *ViewModel* recibe un evento de UI, lanzará un Caso de Uso que acudirá al repositorio del *Model* para obtener los datos requeridos y le devolverá estos datos al *ViewModel*. Una vez recibidos los datos, el *ViewModel* notificará a la *View,* a través del patrón *observer*, para que refresque la UI.



Implementación

Dependencias

En el **Gradle del módulo app**:

```
    implementation "androidx.fragment:fragment-ktx:1.5.7"
    implementation "androidx.activity:activity-ktx:1.7.1"
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.6.1"
    implementation "androidx.lifecycle:lifecycle-livedata-ktx:2.6.1"
```

3. View Binding

El ***binding*** consiste en el proceso de conectar las vistas de una aplicación (UI) con su lógica de programación. Esto permite que los cambios realizados en la interfaz de usuario se reflejen automáticamente en la lógica de la aplicación y viceversa.

Implementación

Dependencias

En el gradle del módulo App:

```
android {
    namespace 'com.jdccmobile.memento'
    compileSdk 33
    //...
    buildFeatures{
        viewBinding = true
    }
}
```

Activity

```
class FavoritesActivity @Inject constructor() : AppCompatActivity() {
    private lateinit var binding: ActivityFavoritesBinding
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityFavoritesBinding.inflate(layoutInflater)
        setContentView(binding.root)
        // ...
    }
```

Fragment

```
class SelectedPokemonFragment @Inject constructor() : Fragment() {

    private var _binding: FragmentSelectedPokemonBinding? = null
    private val binding get() = _binding!!

    // ...
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentSelectedPokemonBinding.inflate(inflater,
container, false)
        return binding.root
    }
    // ...
}
```

4. Dagger Hilt

La **Inyección de Dependencias** (DI, por sus siglas en inglés) es un patrón de diseño utilizado para facilitar la gestión de las dependencias de un sistema. En lugar de que un objeto cree y gestione sus propias dependencias, se le proporcionan a través de la inyección, lo que hace que el sistema sea más flexible, escalable y fácil de mantener.

Dagger Hilt es un framework de inyección de dependencias para aplicaciones Android desarrolladas en Kotlin. Proporciona un conjunto de anotaciones y clases que facilitan la configuración de la inyección de dependencias en una aplicación.

Todo el código de inyección ocurre en el *onCreate()* de la clase correspondiente, así que no hay que hacer nunca nada con las dependencias antes de llamar al *super.onCreate()*.

Dagger Hilt da los errores en tiempo de compilación.

Kapt = *Kotlin Annotation Processing Tool*

Implementación

Dependencias

En el **Gradle del módulo app** tenemos que introducir lo siguiente:

```
dependencies {
  // Dagger hilt
   implementation("com.google.dagger:hilt-android:2.44")
   kapt("com.google.dagger:hilt-android-compiler:2.44")
}
plugins {
    id 'kotlin-kapt'
    id 'com.google.dagger.hilt.android'
}
```

En el **Gradle del proyecto**:

```
plugins {
    id 'com.google.dagger.hilt.android' version "2.44" apply false
}
```

Configurar el Application

La clase Application es la primera que se ejecuta al abrir una app. Esta clase tiene que **extender de Application,** normalmente se crea con el nombre de la app terminado en "App" y utilizar la siguiente etiqueta **@HiltAndroidApp**. Luego, hay que ir al *Manifest* y poner el nombre de esta clase en la etiqueta **android:name**.

Configurar Activity y Fragment

Hay que poner la etiqueta **@AndroidEntryPoint** para configurar la Activity y esté preparada para poder inyectar algo.

Con **@Inject constructor()** podemos inyectar lo que se necesite.

Configurar un ViewModel

Hay que poner la etiqueta **@HiltViewModel** para configurar la Activity y esté preparada para poder inyectar algo.

Con **@Inject constructor()** podemos inyectar lo que se necesite.

Configurar una clase

Para preparar y poder inyectar una clase solo es necesario utilizar **@Inject constructor().** De esta forma pueden inyectar y ser inyectadas.

Inyectar interfaces y librerías externas

Para poder inyectar interfaces o librerías externas tenemos que crear un módulo. Para ello, creamos una nueva clase que será nuestro módulo. A esta clase se le asigna la etiqueta ***@module*** y ***@InstallIn()*** que define el alcance de las dependencias de este módulo. Es importante definir bien el alcance de la dependencia porque cuando el módulo provea alguna dependencia, *dagger* creará una instancia que no morirá hasta que se salga del alcance establecido. En este caso utilizaremos un alcance *singleton* que se trata de un alcance global que solo morirá cuando se cierre la app.

Dentro de la clase se creará una función que será la encargada de definir lo que se va a proveer. Es necesario asignarle la etiqueta ***@provide*** y ***@singleton*** para que solo cree una única instancia.

```
@Module
@InstallIn(SingletonComponent::class)
class AppModule {

    // Preferences
    @Singleton
    @Provides
    fun provideDataStore(
        @ApplicationContext app: Context
    ): PreferencesDataStore = PreferencesImp(app)

    // Retrofit
    @Singleton
    @Provides
    fun provideRetrofit() : Retrofit {
        return Retrofit.Builder()
            .baseUrl("https://pokeapi.co/api/v2/")
            .addConverterFactory(GsonConverterFactory.create())
            .build()
    }

    // Interfaz
    @Singleton
```

12

```
    @Provides
    fun providePokemonApiClient(retrofit: Retrofit) : PokemonApiClient
{
        return retrofit.create(PokemonApiClient::class.java)
    }

}
```

5. RecyclerView

RecyclerView facilita que se muestren de manera eficiente grandes conjuntos de datos. Le pasamos los datos y definimos el aspecto de cada elemento, y la biblioteca de *RecyclerView* creará los elementos de forma dinámica cuando los necesite. Como su nombre indica, el *RecyclerView* recicla esos elementos individuales. Cuando un elemento se desplaza fuera de la pantalla, el *RecyclerView* no destruye su vista, sino que reutiliza la vista para los nuevos elementos que se muestran ahora en pantalla. Esto mejora en gran medida el rendimiento y la capacidad de respuesta de la app y reduce el consumo de energía.

Implementación

Vista XML

Primero, tenemos que crear el **RecyclerView** en el layout XML:

```
<androidx.recyclerview.widget.RecyclerView/>
```

Para crear el **Item** de cada celda del RecyclerView tenemos que generar un nuevo layout XML.

Configuración en la Activity

A continuación, en la **activity** donde queremos utilizar el RecyclerView tenemos que inicializarlo llamando al adapter que crearemos a continuación:

```
private fun initRecyclerView() {
        //binding.rvPokemon.layoutManager = LinearLayoutManager(this)
        // Para mostrar los items en formato de tabla
        binding.rvPokemon.layoutManager = GridLayoutManager(this, 2)
        binding.rvPokemon.adapter = PokemonAdapter(pokemons)
    }
```

Configuración del RecyclerView

Ahora creamos el Adapter y el View Holder.
El **Adapter** es la clase que se encarga de recibir la información y meterla en el RecyclerView.
El **View Holder** es la clase que se encarga de pintar las celdas de cada una de las celdas del listado. En este caso lo utilizamos como *inner class* para mejorar rendimiento, ya que cuando el adapter se muera automáticamente la inner class. Si lo creamos en una *class* puede darse que cuando se muera el *Adapter* quede la instancia del *ViewHolder* viva.
Así, quedaría nuestra clase:

```
class PokedexAdapter(private val pokemon: List<PokemonList>, private
val onClickListener: (Int) -> Unit) :
RecyclerView.Adapter<PokedexAdapter.PokemonViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType:
Int): PokemonViewHolder {
// Le pasa al view holder el layout de cada item que va a pintar
        val layoutInflater = LayoutInflater.from(parent.context)
        return
PokemonViewHolder(layoutInflater.inflate(R.layout.item_pokemon,
parent, false))
    }

// Tamaño del listado que se va a pintar
    override fun getItemCount(): Int = pokemon.size

    override fun onBindViewHolder(holder: PokemonViewHolder,
position: Int) {
// Pasa por cada uno de los items y llama al render del
PokemonViewHolder
        val item = pokemon[position]
        holder.render(position, item, onClickListener)
    }
```

```
    // ViewHolder
    inner class PokemonViewHolder(view: View) :
RecyclerView.ViewHolder(view) {
        // Es llamado con cada uno de los items de listado de entrada
y coge los atributos y los pinta
        private val binding = ItemPokemonBinding.bind(view)

        fun render(position: Int, pokemon: PokemonList,
onClickListener: (Int) -> Unit){ // se llama automaticamente por cada
item
            binding.tvPkmName.text = pokemon.name.replaceFirstChar {
it.uppercase() }
            // Al pulsar en el item
            itemView.setOnClickListener{
                Log.i("TAG", "pulsado")
                onClickListener(position + 1)
            }
        }
    }
}
```

RecyclerView infinito

En este caso cada vez que lleguemos al final del *RecyclerView* necesitamos hacer una nueva petición a la API para mostrar más pokémons. Para saber en que momento se hace scroll hasta el último ítem utilizamos la interfaz *addOnScrollListener():*

```
binding.rvPokemon.addOnScrollListener(object :
RecyclerView.OnScrollListener() {
   override fun onScrolled(recyclerView: RecyclerView, dx: Int, dy:
Int) {
       super.onScrolled(recyclerView, dx, dy)

       val layoutManager = recyclerView.layoutManager as
GridLayoutManager
       val lastVisibleItemPosition =
layoutManager.findLastVisibleItemPosition()
       val totalItemCount = layoutManager.itemCount

       if (!isLoading && lastVisibleItemPosition == totalItemCount -
1) {
           // Llegaste al final de la lista y no se está cargando más
datos
           loadMoreData()
       }
   }
})
```

La función *loadMoreData()* aumenta el offset y hace una nueva petición a la base de datos y cuando los reciba el *ViewModel* notifica a la *Activity* que, a su vez, notifica al *Adapter* para que añada los nuevos pokémons al *RecyclerView* y notifique de los cambios, actualizando la interfaz de usuario.

6. Retrofit + Corrutinas

Retrofit

Retrofit es una biblioteca de Android desarrollada por Square que facilita la implementación de **peticiones HTTP y el consumo de APIs** en nuestras aplicaciones.

La API más común es la API REST. Una API REST es un servicio que nos provee de las funciones que necesitamos para poder obtener información de un cliente externo, como por ejemplo, una base de datos alojada en cualquier parte del mundo desde dentro de nuestra propia aplicación.

Disponemos de cuatro tipos distintos de peticiones como norma general:

- **Get:** Recibe la información del servidor y se pueden pasar parámetros a la petición a través de la url.
- **Post:** Envía datos al servidor. El tipo del cuerpo de la solicitud es indicada por la cabecera Content-Type.
- **Put**: Se suele usar para crear la entidad, es decir, si pensamos en un servicio, como el acceso a una base de datos, este crearía el usuario por ejemplo.
- **Delete:** Permite borrar los registros de la base de datos.

Retrofit tiene varios tipos de notación para configurar la URL que se envía:

- **@Path:** Reemplaza con nombre un segmento de la ruta de la URL.
- **@Query:** Añade parámetros de consulta a la URL.
- **@Body**: Controla el cuerpo de una solicitud POST/PUT, en vez de enviarlos como parámetros.

Ejemplo de una URL con **@Path** y **@Query** :

www.app.net/api/searchtypes/**{Path}**/filters**?**Type=**{Query}**&SearchText=**{Query}**

La información se suele enviar con el formato JSON.

JSON es un formato de texto simple, es el acrónimo de JavaScript Object Notation. Se trata de uno de los estándar para el traspaso de información entre plataformas, tiene una forma muy legible que nos permite entender su contenido sin problema.

Corrutinas

Hay que tener en cuenta que si hacen peticiones muy largas pueden llegar a bloquear el hilo principal y arruinar la experiencia de usuario. Por ello, hay que hacer la llamada a la API en un hilo secundario, utilizando las corrutinas.

En resumen, para modificar datos de la UI hay que utilizar el hilo principal, sino la app se cerrará. Para obtener datos pesados, hacer peticiones a una API u otras acciones acciones que puedan bloquear el hilo principal, hay que usar otro hilo para no perjudicar la experiencia de usuario.

Implementación

Manifest

Añadimos los permisos de internet para que la app pueda acceder

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Dependencias

Añadimos las dependencias en el **Gradle del módulo App**. Hay dos dependencias principales: la de **Retrofit** y la del **Gson Converte**r que simplifica el proceso de pasar de un JSON a una Data Class. También, son necesarias las **corrutinas** para hacer las peticiones en segundo plano y no bloquear el hilo principal y **Picasso** que permite transformar las URLs en imágenes.

```
implementation 'com.squareup.retrofit2:retrofit:2.7.2'
implementation 'com.squareup.retrofit2:converter-gson:2.4.0'
implementation
'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.6'
implementation "com.squareup.picasso:picasso:2.71828"
```

De JSON a *DataClass*

El plugin “JSON To Kotlin Class” devuelve la estructura de la respuesta JSON en un Data Class con la podremos trabajar en el proyecto.

Para poder recuperar la información el nombre de las variables tiene que ser el mismo que el del JSON. Para poder cambiar el nombre de la variable a nuestro gusto podemos utilizar la etiqueta *SerializedName* que hace de puente entre el nombre obligatorio y el que queramos ponerle. Además, esta etiqueta también nos evitará errores cuando se lance la aplicación a producción y se haga la ofuscación.

Por tanto, en ***SerializedName*** ponemos el nombre del JSON y en la variable el que queramos.

```
data class PokemonList(
    @SerializedName("name") val name: String,
    @SerializedName("url") val url: String
)
```

Creación del servicio

Comenzamos creando una **interfaz** que define los métodos que utilizaremos para realizar las operaciones CRUD.

```
 interface PokemonApiClient {
    @GET("pokemon")
    suspend fun getAllPokemons(): Response<PokedexResult>
    @GET("pokemon/{pokemonId}")
    suspend fun getPokemonInfo(@Path("pokemonId") pokemonId : Int):
Response<PokemonInfoResult>
}
```

Una vez creada la interfaz, necesitamos la **clase que implemente dicha interfaz y utiliza Retrofit** gracias a la inyección de dependencias:

```
class PokemonService @Inject constructor(
    private val apiClient: PokemonApiClient
) {
    suspend fun getAllPokemons(): List<PokemonList> {
        return withContext(Dispatchers.IO) { // devuelve la
informacion cuando este creado sin afectar la hilo principal
            val response = apiClient.getAllPokemons()
```

```
            val body = response.body()
            body?.results ?: emptyList()
        }
    }
    suspend fun getPokemonInfo(pokemonId : Int): PokemonInfoResult {
        return withContext(Dispatchers.IO) {
            val response = apiClient.getPokemonInfo(pokemonId)
            val body = response.body()
            body ?: PokemonInfoResult("Error de conexion", 0,
emptyList(), emptyList())
        }
    }
}
```

Inyección de dependencias

En este caso, como utilizamos una interfaz y una librería externa no podemos utilizar *@Inject constructor* como hicimos anteriormente. Por tanto, necesitamos crear un módulo para poder inyectar este tipo de dependencias.

Para ello, creamos un módulo y le añadimos las etiquetas *@Module*, obligatoria e informa a Hilt como proporcionar instancias de determinados tipos, y *@InstallIn(SingletonComponent::class)*, que define el alcance de nuestras dependencias, es decir, cuando el módulo provea alguna dependencia, dagger creará una instancia y durará hasta que se salga del alcance definido. Entonces, con este alcance se creará una única instancia que durará hasta que se termine la app.

Una vez definido el módulo, dentro de él indicamos las dependencias que vamos a inyectar a través de él. En este caso necesitaremos las etiquetas *@Provides y @Singleton*.

Nota: La URL base debe terminar siempre en “ / ”

```
@Module
@InstallIn(SingletonComponent::class)
class AppModule {
```

```
    // Preferences
    @Singleton
    @Provides
    fun provideDataStore(
        @ApplicationContext app: Context
    ): PreferencesDataStore = PreferencesImp(app)
    // Retrofit
    @Singleton
    @Provides
    fun provideRetrofit() : Retrofit {
        return Retrofit.Builder()
            .baseUrl("https://pokeapi.co/api/v2/")
            .addConverterFactory(GsonConverterFactory.create())
            .build()
    }
    @Singleton
    @Provides
    fun providePokemonApiClient(retrofit: Retrofit) :
PokemonApiClient {
        return retrofit.create(PokemonApiClient::class.java)
    }
}
```

7. DataStore Preferences

DataStore es una solución nueva y mejorada de almacenamiento de datos que apunta a reemplazar SharedPreferences. Se usa para almacenar conjuntos de datos pequeños o simples con frecuencia, se basa en corrutinas de Kotlin y Flow y proporciona dos implementaciones diferentes: Proto DataStore, que almacena objetos escritos (con el respaldo de búferes de protocolo) y **Preferences DataStore,** que almacena pares clave-valor. Los datos se almacenan de forma asíncrona, simple y segura.

En este proyecto se utiliza Preferences DataStore que tiene las siguientes características:

- API asíncrona (Flow y RxJAva 2 y 3 Flowable)
- Se ejecuta en el Dispatchers.IO por lo que no bloquea el hilo principal
- Puede indicar errores
- Seguro contra excepciones de tiempo de ejecución
- Controla la migración de datos

La principal diferencia con Proto DataStore es que este ofrece seguridad de tipos gracias a los búferes de protocolos.

Implementación

Dependencias

Primero, tenemos que añadir las dependencias en el **Gradle del módulo App**:

```
implementation "androidx.datastore:datastore-preferences:1.0.0"
```

Configuración del servicio

Comenzamos creando la **interfaz del repositorio**, preparando las funciones de leer/guardar un entero o un booleano. Utilizamos una interfaz porque nos ayuda al desacoplamiento y a la hora de hacer los test Unitarios, ya que durante las pruebas no queremos guardar datos en data store.

```
interface PreferencesDataStore {
    suspend fun putInt(key: Preferences.Key<Int>, value: Int)
    suspend fun getInt(key: Preferences.Key<Int>) : Int?
    suspend fun putBoolean(key: Preferences.Key<Boolean>, value:
Boolean)
    suspend fun getBoolean(key: Preferences.Key<Boolean>) : Boolean?
}
```

Para implementar la interfaz, creamos la clase PreferencesImp donde estará la lógica para leer o guardar datos en DataStore.
Primero, **creamos la instancia de DataStore**:

```
// Creamos instancia de DataStore
private const val PREFERENCES_NAME = "preferences"
private val Context.dataStore by preferencesDataStore(name =
PREFERENCES_NAME)
```

Segundo, como DataStore Preferences utiliza un sistema de pares clave-valor. Esta clave se trata de un valor {boolean/int/long…}PreferencesKey. Para tener las claves organizadas y poder obtenerlas desde cualquier parte del código, las creamos en un **objeto PreferencesKeys**.

```
object PreferencesKeys {
    val VICTORIES = intPreferencesKey("victories")
    val SHOW_COMBAT_INFO = booleanPreferencesKey("show_combat_info")
}
```

Grabación de datos

Para **grabar datos**, utilizamos la función de extensión de .edit:

```
        context.dataStore.edit { preferences ->
            preferences[key] = value
        }
```

Lectura de datos

Para **leer datos**, utilizamos la función de la corrutina flow .first() para obtener el primer valor que encuentre para la clave indicada. Se utiliza una bloque try-catch para recoger las posibles excepciones que puede devolver:

```
        return try {
            val preferences = context.dataStore.data.first()
            preferences[key]
        } catch (e: Exception){
            e.printStackTrace()
            null
        }
```

Inyección de dependencias

Como en este proyecto utilizamos inyección de dependencias, necesitamos inyectar la interfaz de nuestro DataStorePreferences. Como Hilt no tiene información sobre las interfaces, **creamos una función Singleton en el AppModule** de la carpeta DI (Revisar como lo explica Aris):

```
    @Singleton
    @Provides
    fun provideDataStore(
        @ApplicationContext app: Context
    ): PreferencesDataStore = PreferencesImp(app)
```

Ejemplo completo

Finalmente, nuestra clase completa quedaría así:

```
// Creamos instancia de DataStore
private const val PREFERENCES_NAME = "preferences"
private val Context.dataStore by preferencesDataStore(name =
PREFERENCES_NAME)

 class PreferencesImp @Inject constructor(
    private val context: Context
) : PreferencesDataStore {

    override suspend fun putInt(key:  Preferences.Key<Int>, value:
Int){
        context.dataStore.edit { preferences ->
            preferences[key] = value
        }
    }

    override suspend fun getInt(key: Preferences.Key<Int>) : Int? {
        return try {
            val preferences = context.dataStore.data.first()
            preferences[key]
        } catch (e: Exception){
```

```
            e.printStackTrace()
            null
        }
    }
}
```

8. Enlaces

View Binding

https://developer.android.com/topic/libraries/view-binding?hl=es-419

https://cursokotlin.com/capitulo-29-view-binding-en-kotlin/

Dagger Hilt

https://developer.android.com/training/dependency-injection?hl=es-419

https://cursokotlin.com/dagger-hilt-inyeccion-de-dependencias-mvvm/

RecyclerView

https://developer.android.com/guide/topics/ui/layout/recyclerview?hl=es-419

Retrofit

https://devexperto.com/retrofit-android-kotlin/

https://cursokotlin.com/tutorial-retrofit-2-en-kotlin-con-corrutinas-consumiendo-api-capitulo-20-v2/

https://square.github.io/retrofit/2.x/retrofit/index.html?retrofit2/http/Query.html

Data Store Preferences

https://developer.android.com/codelabs/android-preferences-datastore?hl=es-419#0

https://medium.com/@vgoyal_1/datastore-android-how-to-use-it-like-a-pro-using-kotlin-2c2440683d78

Animaciones

https://developer.android.com/training/animation/overview?hl=es-419