cpifp Los Enlaces

PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

2º DESARROLLO DE APLICACIONES MULTIPLATAFORMA

TEMA 8. BASES DE DATOS

T8. BASES DE DATOS

0. ROOM

Las apps que controlan grandes cantidades de datos estructurados pueden beneficiarse con la posibilidad de conservarlos localmente. El caso de uso más común consiste en almacenar en caché datos relevantes para que el dispositivo no pueda acceder a la red, de modo que el usuario pueda explorar ese contenido mientras está sin conexión.

La biblioteca de persistencias **Room** brinda una capa de abstracción para SQLite que permite acceder a la base de datos sin problemas y, al mismo tiempo, aprovechar toda la tecnología de SQLite.

0. ROOM

En particular, **Room** brinda los siguientes beneficios:

- Verificación del tiempo de compilación de las consultas en SQL.
- Anotaciones de conveniencia que minimizan el código estándar repetitivo y propenso a errores.
- Rutas de migración de bases de datos optimizadas.

Debido a estas consideraciones, AndroidDevelopers recomienda usar Room en lugar de usar las APIs de SQLite directamente.

T8. BASES DE DATOS

1. COMPONENTES PRINCIPALES

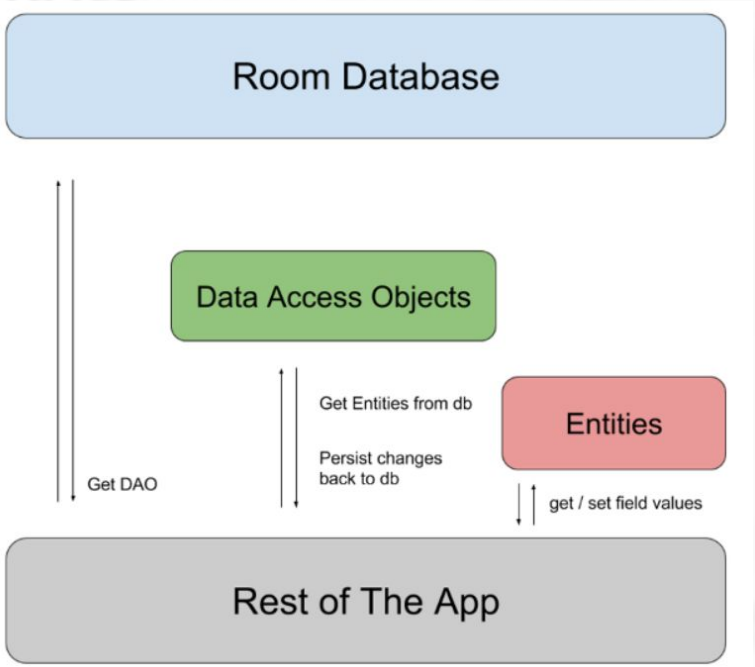
Estos son los tres componentes principales de Room:

- La **clase de la base de datos** que contiene la base de datos y sirve como punto de acceso principal para la conexión subyacente a los datos persistentes de la app.
- Las **entidades** de datos que representan tablas de la base de datos de tu app.
- Los **objetos de acceso a datos** (DAOs) que proporcionan métodos que tu app puede usar para consultar, actualizar, insertar y borrar datos en la base de datos.

T8. BASES DE DATOS

1. COMPONENTES PRINCIPALES

La clase de base de datos proporciona a tu app instancias de los DAOs asociados. A su vez, la app puede usar los DAOs para recuperar datos como instancias de objetos de entidad de datos asociados. La app también puede usar las entidades de datos definidas para actualizar filas de las tablas correspondientes o crear filas nuevas para su inserción.



2. EJEMPLO ANDROID DEVELOPERS

Entidad de datos: El siguiente código define una entidad de datos User. Cada instancia de User representa una fila en una tabla de *user* en la base de datos de la app.

```
@Entity
data class User(
    @PrimaryKey val uid: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

2. EJEMPLO ANDROID DEVELOPERS

Objeto de acceso a datos (DAO): El siguiente código define un DAO llamado UserDao. UserDao proporciona los métodos que el resto de la app usa para interactuar con los datos de la tabla *user*.

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    fun getAll(): List<User>

    @Query("SELECT * FROM user WHERE uid
IN (:userIds)")
    fun loadAllByIds(userIds: IntArray):
List<User>
```

```
    @Query("SELECT * FROM user WHERE
first_name LIKE :first AND " +
"last_name LIKE :last LIMIT 1")
    fun findByName(first: String, last:
String): User
    @Insert
    fun insertAll(vararg users: User)
    @Delete
    fun delete(user: User)
}
```

2. EJEMPLO ANDROID DEVELOPERS

Base de datos: Se define una clase AppDatabase para contener la base de datos. AppDatabase define la configuración de la base de datos y sirve como el punto de acceso principal de la app a los datos persistentes.

```
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
   abstract fun userDao(): UserDao
}
// La versión es para hacer migraciones (como un control de versiones para BD)
```

2. EJEMPLO ANDROID DEVELOPERS

La clase de la base de datos debe cumplir con las siguientes condiciones:

- La clase debe tener una anotación **@Database** que incluya un array entities que enumere todas las entidades de datos asociados con la base de datos.
- Debe ser una clase abstracta que extienda **RoomDatabase**.
- Para cada clase **DAO** que se asoció con la base de datos, esta base de datos debe definir un método abstracto que tenga cero argumentos y muestre una instancia de la clase DAO.

2. EJEMPLO ANDROID DEVELOPERS

Uso: Después de definir la entidad de datos, el DAO y el objeto de base de datos, puedes usar el siguiente código para crear una instancia de la base de datos.

```
val db = Room.databaseBuilder(
    applicationContext,
    AppDatabase::class.java, "database-name"
).build()
```

3. ARQUITECTURA MVVM

Para estudiar el acceso a datos mediante Room vamos a crear una aplicación que consuma una API y almacene localmente aquellos datos que nos interesen, extraídos mediante una consulta a la API.

También vamos a organizar el proyecto siguiendo una arquitectura **Model-View-Viewmodel**, lo que nos permitirá segmentar el código por capas. Esto mejora la escalabilidad del proyecto.

El **modelo** se encarga de manejar todos los datos, la **vista** interactúa directamente con la UI y el **viewmodel** los conecta a ambos.

3. ARQUITECTURA MVVM

Creamos un nuevo proyecto llamado *room_database* y abrimos el archivo *build.gradle.kts (Module: app)* para incluir el método ViewBinding. Además, añadiremos las librerías que vamos a necesitar.

```
    kotlinOptions {
        jvmTarget = "1.8"
    }
    buildFeatures{
        viewBinding = true
    }
}
```

3. ARQUITECTURA MVVM

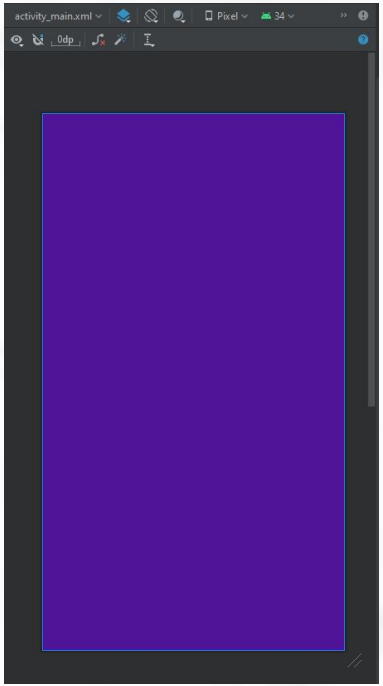
La librería LiveData es la que va a conectar nuestra *activity* con el ViewModel. Cada vez que haya un cambio en nuestro modelo, a través del patrón *observer*, avisará a la *activity* de dicho cambio.

```
dependencies {
    ...
    // Fragment
    implementation ("androidx.fragment:fragment-ktx:1.6.2")
    // Activity
    implementation ("androidx.activity:activity-ktx:1.8.2")
    // ViewModel
    implementation ("androidx.lifecycle:lifecycle-viewmodel-ktx:2.7.0")
    // LiveData
    implementation ("androidx.lifecycle:lifecycle-livedata-ktx:2.7.0")
```

4. LAYOUT PRINCIPAL

Pasamos al diseño y a nuestro ConstraintLayout le añadimos una *id* y un color de fondo. La aplicación mostrará citas famosas tomadas de la API.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
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:id="@+id/viewContainer"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#4F1497"
    tools:context=".MainActivity">
```



4. LAYOUT PRINCIPAL

Insertamos un primer TextView centrado en la pantalla.

```
<TextView
    android:id="@+id/tvQuote"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:gravity="center"
    android:padding="16dp"
    android:textColor="@color/white"
    android:textSize="24sp"
    android:textStyle="italic"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:text='"Cita"'/>
```

4. LAYOUT PRINCIPAL

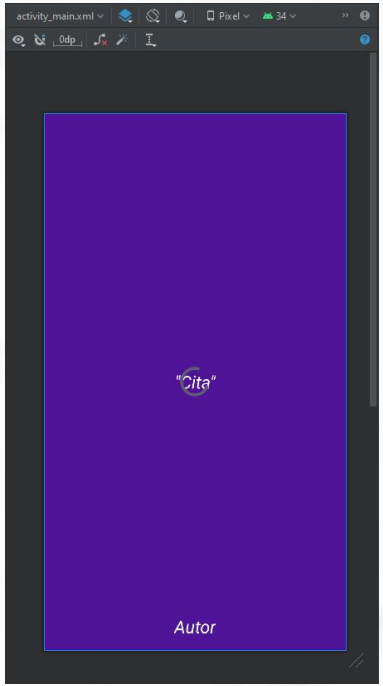
Y un segundo TextView para mostrar el nombre del autor.

```
<TextView
    android:id="@+id/tvAuthor"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:gravity="center"
    android:padding="16dp"
    android:textColor="@color/white"
    android:textSize="24sp"
    android:textStyle="italic"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    tools:text="Autor" />
```

4. LAYOUT PRINCIPAL

Por último una ProgressBar que se mostrará cuando realicemos consultas a la API. En este caso no es necesario ponerle el atributo `android:visibility="gone"`.

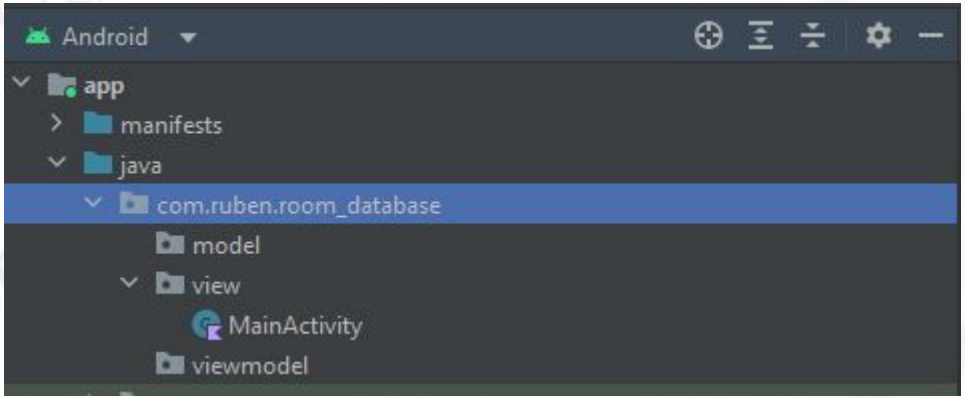
```
<ProgressBar
    android:id="@+id/loading"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:visibility="invisible"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"/>
```



5. LÓGICA

Preparamos el ViewBinding en la actividad principal (intenta hacerlo sin copiar de otras aplicaciones) y preparamos el esqueleto de directorios que vamos a necesitar (*New → Package*): */model*, */view* y */viewmodel*.

Arrastramos la clase *MainActivity* al directorio */view* y aplicamos *Refactor* en la ventana emergente que aparece.



5. LÓGICA

Dentro del directorio */model*, creamos la **clase** *QuoteModel.kt* que representa el modelo de datos con el que vamos a trabajar (objetos que contengan una cita y su autor).

```
data class QuoteModel(val quote: String, val author: String)
```

5. LÓGICA

También vamos a crear una **clase** *QuoteProvider.kt* que va a suministrar una cita aleatoria de un listado definido inicialmente dentro de esta clase.

```
class QuoteProvider {
    private val quote = listOf<QuoteModel>(
        QuoteModel(
            quote = "It's not a bug. It's an undocumented feature!",
            author = "Anonymous"
        ),
        QuoteModel(
            quote = "Software Developer - An organism that turns caffeine into
software.",
            author = "Anonymous"
        ),
```

5. LÓGICA

```
QuoteModel(
    quote = "A user interface is like a joke. If you have to explain it, it´s not that good.",
    author = "Anonymous"
),
QuoteModel(
    quote = "I don´t care if it works on your machine! We are not shipping your machine!",
    author = "Vidiu Platon"
),
QuoteModel(
    quote = "Measuring programming progress by lines of code is like measuring aircraft building progress by weight.",
    author = "Bil Gates"
),
```

5. LÓGICA

```
QuoteModel(
    quote = "My code DOESN'T work, I have no idea why. My code WORKS, I have no idea
why.",
    author = "Anonymous"
),
QuoteModel(quote = "Things aren´t always #000000 or #FFFFFF", author = "Anonymous"),
QuoteModel(quote = "Talk is cheap. Show me the code.", author = "Linus Torvalds"),
QuoteModel(
    quote = "Software and cathedrals are much the same - first we build them, then we
pray",
    author = "Anonymous"
)

)
}
```

5. LÓGICA

Para que pueda devolver una cita aleatoria, vamos a implementar un método dentro de un *companion object*, incluyendo la lista de citas (*private val*), que se pueda invocar desde cualquier parte del código.

```
class QuoteProvider {
    companion object {

        fun random(): QuoteModel {
            val position: Int = (0..8).random()
            return quote[position]
        }

        private val quote = listOf<QuoteModel>(
            QuoteModel(
```

5. LÓGICA

Ahora creamos la **clase** *QuoteViewModel.kt* dentro del directorio */viewmodel*. Esta clase tendrá una propiedad tipo *LiveData* (conecta modelos y vistas) y un método que almacene en ella una cita aleatoria.

```
class QuoteViewModel : ViewModel() {

    val quoteModel = MutableLiveData<QuoteModel>()

    fun randomQuote() {
        val currentQuote: QuoteModel = QuoteProvider.random()
        quoteModel.postValue(currentQuote)
    }
}
```

5. LÓGICA

En *MainActivity.kt* creamos la variable *quoteViewModel* que implementa la clase creada anteriormente, delegando los métodos para establecer la conexión entre modelo y vista a la función *viewModels()* de LiveData.

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    private val quoteViewModel: QuoteViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
```

5. LÓGICA

Dentro del método *onCreate()* implementamos el patrón *observer*, que está anexionado a LiveData y siempre que se produzca un cambio en el atributo *quoteModel* (de tipo *MutableLiveData*) ejecutará su contenido.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)
    quoteViewModel.quoteModel.observe(this, Observer {
    //Podemos poner nombre a it justo después de "{", por ejemplo, "currentQuote ->"
        binding.tvQuote.text = it.quote
        binding.tvAuthor.text = it.author
    })
}
```

5. LÓGICA

Solo nos queda incluir el método *setOnClickListener* a nuestra pantalla (es decir, al *ConstraintLayout*) para que se actualice con una cita aleatoria a través del método que hemos creado para ello.

```
        quoteViewModel.quoteModel.observe(this, Observer {
    //Podemos poner nombre a it justo después de "{", por ejemplo, "currentQuote ->"
            binding.tvQuote.text = it.quote
            binding.tvAuthor.text = it.author
        })

        binding.viewContainer.setOnClickListener { quoteViewModel.randomQuote() }
    }
}
```

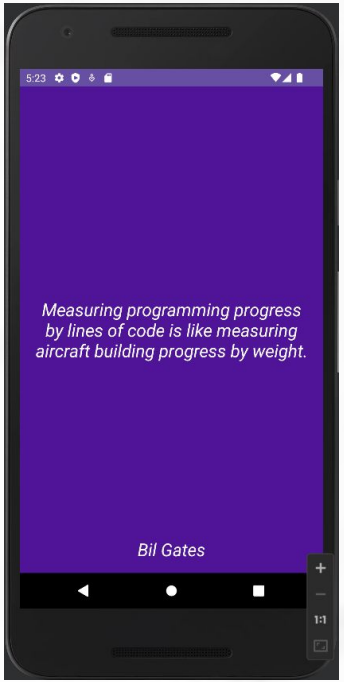
5. LÓGICA

Ahora podemos lanzar la aplicación y comprobar que todo funciona correctamente.

Lo siguiente que vamos a hacer es añadir las citas desde una API. Para eso, en el archivo AndroidManifest.xml incluimos el permiso para internet.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

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



5. LÓGICA

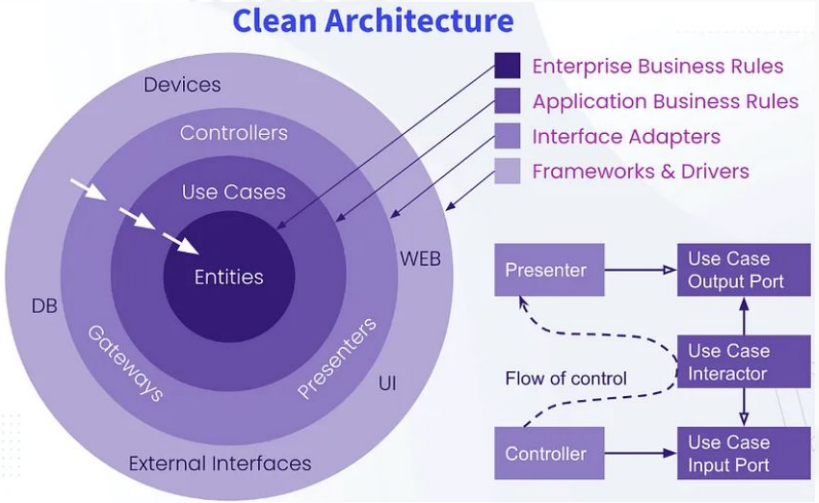
Añadimos las dependencias para Retrofit y corrutinas en el archivo *build.gradle.kts (Module: app)*. Y creamos un nuevo directorio ***/core*** en el directorio de nuestro proyecto.

```
// ViewModel
implementation ("androidx.lifecycle:lifecycle-viewmodel-ktx:2.7.0")
// LiveData
implementation ("androidx.lifecycle:lifecycle-livedata-ktx:2.7.0")
// Retrofit
implementation ("com.squareup.retrofit2:retrofit:2.9.0")
implementation ("com.squareup.retrofit2:converter-gson:2.9.0")
//Corrutinas
implementation ("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.7.1")
```

6. CLEAN ARQUITECTURE

Este nuevo directorio */core* contendrá las funciones principales de nuestra aplicación. Nos servirá para implementar un patrón de diseño ***Clean Arquitecture***.

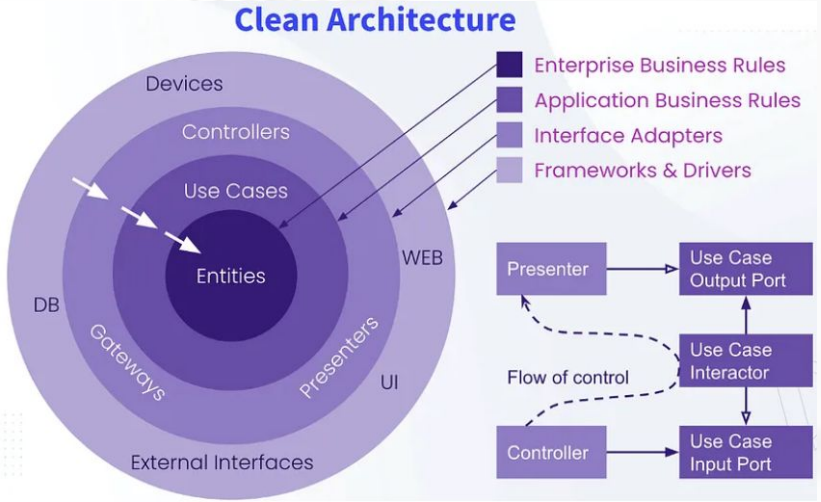
Las arquitecturas limpias son un concepto general que proporciona una guía para diseñar sistemas de software con una estructura modular y una clara separación de responsabilidades.



6. CLEAN ARQUITECTURE

La idea principal detrás de las arquitecturas limpias es separar las preocupaciones en diferentes capas bien definidas, con reglas estrictas sobre cómo deben interactuar entre sí.

Capa de Entidades (Entities):
Contiene las entidades centrales y los objetos de negocio del dominio. Son clases que encapsulan los datos principales sin estar acopladas a ninguna otra capa del sistema.



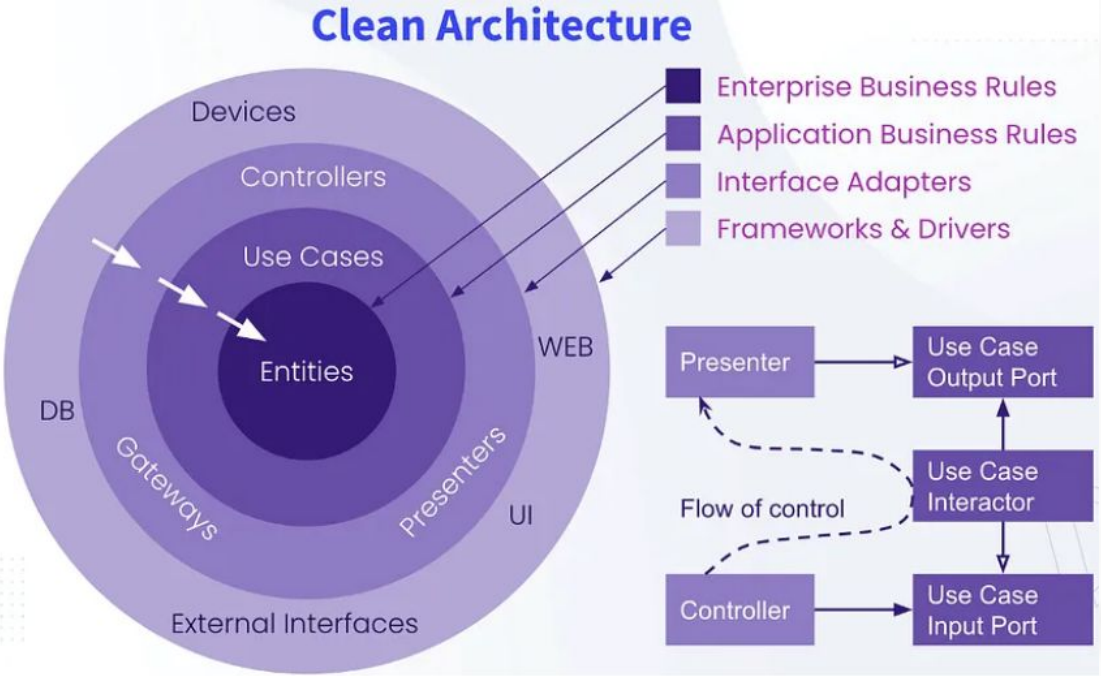
6. CLEAN ARQUITECTURE

Capa de Casos de Uso (Use Cases): Esta capa contiene la lógica de la aplicación y coordina la interacción entre las entidades del dominio. Los casos de uso encapsulan la lógica de negocio específica para cada función o acción que debe realizar la aplicación.

Capa de Adaptadores (Adapters): Los adaptadores son las interfaces con el mundo exterior, como la interfaz de usuario (UI), la base de datos y otros servicios externos. Estos adaptadores convierten los datos del exterior en un formato que las capas internas puedan entender y viceversa.

6. CLEAN ARQUITECTURE

Capa de Frameworks y Herramientas (Frameworks and Tools): Esta capa contiene el código que interactúa con bibliotecas, frameworks y herramientas externas. Es la capa más externa de la arquitectura.



6. CLEAN ARQUITECTURE

Dentro de este directorio (*/core*), vamos a crear un **objeto** *RetrofitHelper* que contenga la función `getRetrofit()` de proyectos anteriores. Con la URL base de la API que vamos a consumir.

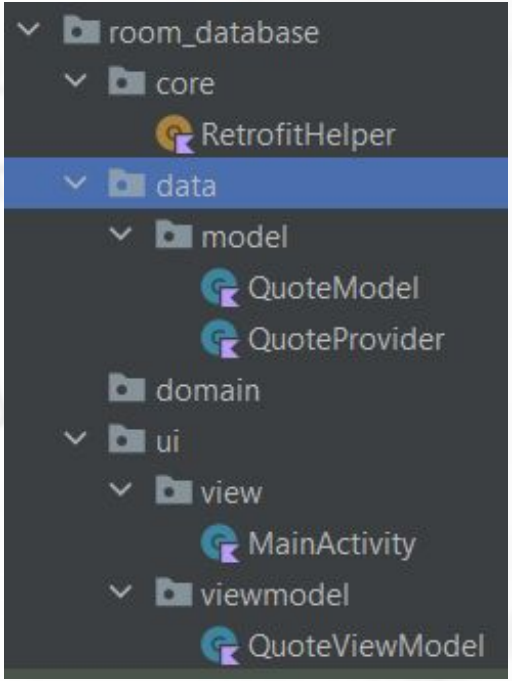
```
object RetrofitHelper {
  fun getRetrofit():Retrofit{
      return Retrofit.Builder()
   .baseUrl("https://drawsomething-59328-default-rtdb.europe-west1.firebasedatabase.app/")
   .addConverterFactory(GsonConverterFactory.create())
   .build()
  }
}
```

6. CLEAN ARQUITECTURE

Ahora añadimos el directorio */ui*, que es la capa exterior de nuestra *Clean Arquitecture*, y arrastramos los directorios */view* y */viewmodel* dentro de éste.

La siguiente capa será */domain*, que contendrá toda la lógica de negocio.

Y la última capa será */data*, que contendrá el modelo creado anteriormente.



7. LÓGICA

La clase que va a recoger los datos suministrados por la API es la que ya hemos creado anteriormente (*QuoteModel.kt*). Aunque los campos definidos en la API se llaman igual que los atributos, implementamos *@SerializedName* a los mismos para identificar claramente que ésta es la clase que recoge los datos.

```
import com.google.gson.annotations.SerializedName

data class QuoteModel(
    @SerializedName("quote") val quote: String,
    @SerializedName("author") val author: String
)
```

7. LÓGICA

Para recuperar la información de backend (a través de internet) creamos un nuevo directorio */network*, dentro de */data*, y creamos la **interfaz** de acceso a la API en él. La llamaremos *QuoteApiClient.kt* y contendré el método GET junto a una función (`suspend fun` para las corrutinas) que se encargará de recoger todas las citas del archivo JSON recibido.

```
interface QuoteApiClient {
    @GET("/.json")
    suspend fun getAllQuotes(): Response<List<QuoteModel>>
}
```

7. LÓGICA

Dentro de */network* también, vamos a crear la **clase** *QuoteService.kt*. La finalidad de esta implementación es que, si en el futuro cambiamos de Retrofit a Firebase o cualquier otra librería, solo tendremos que modificar este código y el resto del proyecto no se verá afectado.

```
class QuoteService {
    private val retrofit = RetrofitHelper.getRetrofit()
    suspend fun getQuotes(): List<QuoteModel> {
        return withContext(Dispatchers.IO) {
            val response = retrofit.create(QuoteApiClient::class.java).getAllQuotes()
            response.body() ?: emptyList()
        }
    }}
```

7. LÓGICA

Ahora, en el directorio */data* (un nivel por encima de */network*) vamos a crear el repositorio (**clase** *QuoteRepository.kt*). Éste se encargará de gestionar los datos a través de internet o de base de datos local. La primera vez solo insertará los datos en la clase *QuoteProvider.kt*.

```
class QuoteRepository {
    private val api = QuoteService()

    suspend fun getAllQuotes():List<QuoteModel>{
        val response:List<QuoteModel> = api.getQuotes()
        return response
    }
}
```

7. LÓGICA

Por lo tanto, debemos ir a la clase *QuoteProvider.kt* y borrar todos los datos que hemos insertado anteriormente. Creamos entonces una variable que almacene la lista de objetos *QuoteModel* inicializada con una lista vacía.

```
class QuoteProvider {
    companion object {
        var quotes:List<QuoteModel> = emptyList()
    }
}
```

7. LÓGICA

Lo que debemos hacer ahora es suministrarle la lista de citas al atributo *quotes* desde el repositorio. Y con esto ya tenemos la parte de los datos configurada por el momento.

```
class QuoteRepository {
    private val api = QuoteService()

    suspend fun getAllQuotes():List<QuoteModel>{
        val response:List<QuoteModel> = api.getQuotes()
        QuoteProvider.quotes = response
        return response
    }
}
```

7. LÓGICA

Pasamos a la capa de dominio, donde se implementan los *casos de uso* o *interactors*. Se trata principalmente de las acciones que el usuario puede desencadenar. Creamos la **clase** *GetQuotesUseCase.kt*.

La sintaxis `invoke()` sirve para poder implementar un método sin necesidad de instanciar un objeto (es un método estático).

```
class GetQuotesUseCase {
    private val repository = QuoteRepository()
    suspend operator fun invoke():List<QuoteModel>? = repository.getAllQuotes()
}
```

7. LÓGICA

Lo siguiente es pasar a la interfaz de usuario y en *MainActivity.kt* invocamos a un método `onCreate()` de la clase *ViewModel.kt*, que crearemos posteriormente para que, al iniciar la aplicación, se cargue en pantalla una primera cita.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    quoteViewModel.onCreate()

    quoteViewModel.quoteModel.observe(this, Observer {
```

7. LÓGICA

En la clase *QuoteViewModel.kt*, comentamos las líneas de código dentro del método `randomQuote()` y añadimos una variable que almacena el método *invoke* de la clase *GetQuotesUseCase.kt*.

```
class QuoteViewModel : ViewModel() {
    val quoteModel = MutableLiveData<QuoteModel>()
    val getQuotesUseCase = GetQuotesUseCase()

    fun randomQuote() {
//          val currentQuote: QuoteModel = QuoteProvider.random()
//          quoteModel.postValue(currentQuote)
    }
}
```

7. LÓGICA

La propia librería de ViewModel trae acceso a un tipo de corrutinas automáticas que no es necesario cerrar cuando muere la actividad que las está utilizando. El método `onCreate()` que vamos a definir añade la primera cita del JSON a nuestra variable LiveData.

```
fun onCreate() {
    viewModelScope.launch {
        val result = getQuotesUseCase()
        if (!result.isNullOrEmpty()) {
            quoteModel.postValue(result[0])
        }
    }
}
```

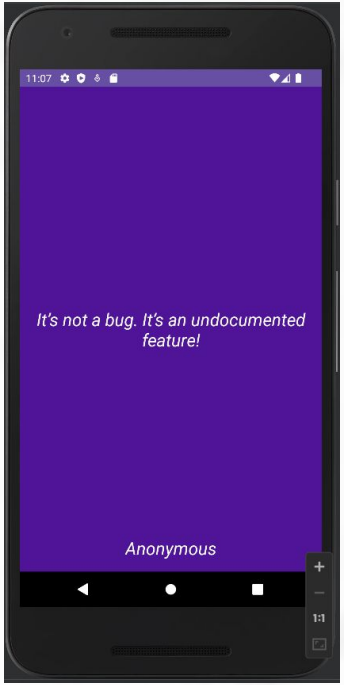
7. LÓGICA

Ejecutamos la aplicación y comprobamos que se muestra la cita en la pantalla inicial.

Vamos a configurar la visibilidad de la ProgressBar del layout principal al cargar esta primera cita.

```
class QuoteViewModel : ViewModel() {

    val quoteModel = MutableLiveData<QuoteModel>()
    val getQuotesUseCase = GetQuotesUseCase()
    val isLoading = MutableLiveData<Boolean>()
```



7. LÓGICA

La variable *isLoading*, tipo LiveData, se encargará de actualizar la vista mediante su patrón *observer*. Solo tenemos que indicarle cuando queremos que se muestre o se oculte.

```
fun onCreate() {
    isLoading.postValue(true)
    viewModelScope.launch {
        val result = getQuotesUseCase()
        if (!result.isNullOrEmpty()) {
            quoteModel.postValue(result[0])
            isLoading.postValue(false)
        }
    }
}
```

7. LÓGICA

Y en la actividad principal debemos crear el *binding* con la ProgressBar (ID *loading* definido en el layout) dentro del patrón *observer* del atributo *isLoading* (tipo LiveData) de la clase QuoteViewModel.

```
quoteViewModel.quoteModel.observe(this, Observer {
    //Podemos poner nombre a it justo después de "{", por ejemplo, "currentQuote ->"
    binding.tvQuote.text = it.quote
    binding.tvAuthor.text = it.author
})
quoteViewModel.isLoading.observe(this, Observer {
    binding.loading.isVisible = it
})

binding.viewContainer.setOnClickListener { quoteViewModel.randomQuote() }
```

7. LÓGICA

Ahora debemos crear un nuevo *caso de uso* para tomar una cita aleatoria cuando el usuario pulse la pantalla. En */domain* creamos la **clase** *GetRandomQuoteUseCase.kt*.

```
class GetRandomQuoteUseCase {
   private val repository = QuoteRepository()
//Ahora no necesita una corrutina porque lo tenemos almacenado en QuoteProvider
   operator fun invoke(): QuoteModel? {
       val quotes = QuoteProvider.quotes
       if (!quotes.isNullOrEmpty()) {
           val randomNumber = (quotes.indices).random()
           return quotes[randomNumber]
       }
       return null  /*No hace falta incluir else*/   }    }
```

7. LÓGICA

Y lo instanciamos desde la clase *QuoteViewModel*. Para ello, creamos la variable *getRandomQuoteUseCase*, que almacena dicho método, y cambiamos la función `randomQuote()` de la siguiente manera:

```
val getRandomQuoteUseCase = GetRandomQuoteUseCase()

fun randomQuote() {
    isLoading.postValue(true)
    val quote = getRandomQuoteUseCase()
    if(quote!=null) {
        quoteModel.postValue(quote)
    }
    isLoading.postValue(false)
}
```

7. LÓGICA

Si ejecutamos la aplicación, podremos comprobar como se muestran aleatoriamente las citas importadas desde la API, ahora almacenadas en nuestra clase *QuoteProvider*.

Lo siguiente que vamos a estudiar es la *inyección de dependencias*. Puedes leer la documentación oficial sobre inyección de dependencias en el siguiente enlace:

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

8. DAGGER HILT

Hilt es una biblioteca de inserción de dependencias para Android que permite reducir el trabajo repetitivo de insertar dependencias de forma manual en tu proyecto.

Para la inserción manual de dependencias, debes construir cada clase y sus dependencias de forma manual, y usar contenedores para reutilizar y administrar las dependencias.

8. DAGGER HILT

Hilt proporciona una forma estándar de usar la inserción de dependencias en tu aplicación, ya que proporciona contenedores para cada clase de Android en tu proyecto y administra automáticamente sus ciclos de vida.

Hilt se basa en la popular biblioteca de inserción de dependencias Dagger y se beneficia de la corrección en tiempo de compilación, el rendimiento del entorno de ejecución, la escalabilidad y la compatibilidad con Android Studio que proporciona.

Para obtener más información, consulta Hilt y Dagger.

8. DAGGER HILT

Pero lo mejor para entenderlo es aplicarlo a nuestro proyecto. Abrimos el archivo *build.gradle.kts (Project: room_database)* y agregamos el siguiente complemento:

```
// Top-level build file where you can add configuration options common to all
sub-projects/modules.
plugins {
    id("com.android.application") version "8.1.1" apply false
    id("org.jetbrains.kotlin.android") version "1.9.0" apply false
    id("com.google.dagger.hilt.android") version "2.50" apply false
}
```

8. DAGGER HILT

En el archivo *build.gradle.kts (Module: app)* añadimos las siguientes líneas en los objetos correspondientes:

```
plugins {
    id("kotlin-kapt")
    id("com.google.dagger.hilt.android")
}
dependencies {
    // Dagger Hilt
    implementation("com.google.dagger:hilt-android:2.50")
    kapt("com.google.dagger:hilt-android-compiler:2.50")
}
kapt {    // Nuevo objeto creado bajo "dependencies"
    correctErrorTypes = true  // Allow references to generated code
}
```

8. DAGGER HILT

Comprobamos además que las opciones de compilación estén configuradas con la versión de Java 1.8. Tras todos estos pasos le damos a sincronizar (*Sync Now* arriba a la derecha).

```
android {
    ...
    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_1_8
        targetCompatibility = JavaVersion.VERSION_1_8
    }
}
```

8. DAGGER HILT

Creamos en el directorio principal una **clase** *room_databaseApp.kt*. Ésta la definiremos como tipo *Application* que es un tipo de clases que heredan de la propia aplicación y se ejecutan al iniciarla.

Con esto le estamos indicando a la aplicación que vamos a usar la inyección de dependencias a través de Dagger Hilt y la propia librería de Hilt nos proporciona todos los métodos necesarios para esta clase.

```
@HiltAndroidApp
class room_databaseApp:Application()
```

8. DAGGER HILT

[MANIFEST]

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools" >

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

    <application
        android:name=".room databaseApp"
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
```

8. DAGGER HILT

Lo primero que tenemos que preparar son las *activities* y el *viewmodel*. Vamos a *MainActivity.kt* e insertamos la siguiente etiqueta:

```
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    private val quoteViewModel: QuoteViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
```

8. DAGGER HILT

Y en *QuoteViewModel.kt* añadimos las siguientes líneas. En este caso, vamos a especificar también la inyección de dependencias en la clase.

```
import dagger.hilt.android.lifecycle.HiltViewModel
import javax.inject.Inject

@HiltViewModel
class QuoteViewModel @Inject constructor() : ViewModel() {

    val quoteModel = MutableLiveData<QuoteModel>()
    val isLoading = MutableLiveData<Boolean>()

    val getQuotesUseCase = GetQuotesUseCase()
    val getRandomQuoteUseCase = GetRandomQuoteUseCase()
```

8. DAGGER HILT

Lo que permite eliminar las instancias de los *casos de uso* gestionados por el *viewmodel* y pasárselas al constructor como parámetros.

```
@HiltViewModel
class QuoteViewModel @Inject constructor(
    private val getQuotesUseCase:GetQuotesUseCase,
    private val getRandomQuoteUseCase:GetRandomQuoteUseCase
) : ViewModel() {

    val quoteModel = MutableLiveData<QuoteModel>()
    val isLoading = MutableLiveData<Boolean>()

    val getQuotesUseCase = GetQuotesUseCase()
    val getRandomQuoteUseCase = GetRandomQuoteUseCase()
```

8. DAGGER HILT

Pero las clases que se van a inyectar también tiene que ser preparadas. Abrimos *GetQuotesUseCase.kt* y *GetRandomQuoteUseCase.kt* y les añadimos la etiqueta *@Inject*.

```
class GetQuotesUseCase @Inject constructor(){
   private val repository = QuoteRepository()
   suspend operator fun invoke(): List<QuoteModel>? = repository.getAllQuotes()
}
```

```
class GetRandomQuoteUseCase @Inject constructor(){
   private val repository = QuoteRepository()
   operator fun invoke(): QuoteModel? {
      val quotes = QuoteProvider.quotes
```

8. DAGGER HILT

A su vez, estas clases tiene instancias de otras clases y, para administrar bien el proyecto, debemos seguir el mismo patrón en todas. Pasamos entonces las instancias por parámetro a estas clases.

```
class GetQuotesUseCase @Inject constructor(private val repository:QuoteRepository){
    private val repository = QuoteRepository()
    suspend operator fun invoke(): List<QuoteModel>? = repository.getAllQuotes()
}
```

```
class GetRandomQuoteUseCase @Inject constructor(private val repository:QuoteRepository){
    private val repository = QuoteRepository()
    operator fun invoke(): QuoteModel? {
        val quotes = QuoteProvider.quotes
```

8. DAGGER HILT

De aquí pasamos a *QuoteRepository.kt* (CTRL + click en la declaración del tipo de parámetro) y volvemos a implementar la etiqueta *@Inject constructor()*.

```
class QuoteRepository @Inject constructor(private val api : QuoteService) {
    private val api = QuoteService()

    suspend fun getAllQuotes():List<QuoteModel>{
        val response:List<QuoteModel> = api.getQuotes()
        QuoteProvider.quotes = response
        return response
    }
}
```

8. DAGGER HILT

Hacemos lo mismo para *QuoteService.kt*, pero el problema que nos encontramos es que la clase Retrofit no se puede inyectar directamente porque pertenece una librería y no podemos modificarla.

```
class QuoteService @Inject constructor(private val api:QuoteApiClient){
  private val retrofit = RetrofitHelper.getRetrofit()
  suspend fun getQuotes(): List<QuoteModel> {
    return withContext(Dispatchers.IO) {
      val response = retrofit.create(QuoteApiClient::class.java) api.getAllQuotes()
      response.body() ?: emptyList()
    }
  }
}
```

8. DAGGER HILT

Para poder inyectar un objeto *retrofit* en cualquier parte del código podemos crear módulos. Creamos el directorio */di* (*dependency_injection*) dentro de */core* y creamos el **objeto** *NetworkModule.kt* en él.

```
@Module
@InstallIn(SingletonComponent::class) //Alcance de aplicación
object NetworkModule {

    @Singleton //Solo permite una instancia de Retrofit para no consumir memoria
    @Provides
    fun provideRetrofit(): Retrofit {

    }
}
```

8. DAGGER HILT

Dentro de la función `provideRetrofit()` copiamos el constructor de Retrofit que teníamos en el método `getRetrofit()` del objeto *RetrofitHelper.kt*.

```
@Singleton //Solo permite una instancia de Retrofit para no consumir memoria
@Provides
fun provideRetrofit(): Retrofit {
    return Retrofit.Builder()

.baseUrl("https://acceso-a-bd-androidstudio-default-rtdb.europe-west1.firebasedatabase.app/"  )
        .addConverterFactory(GsonConverterFactory.create())
        .build()
}
```

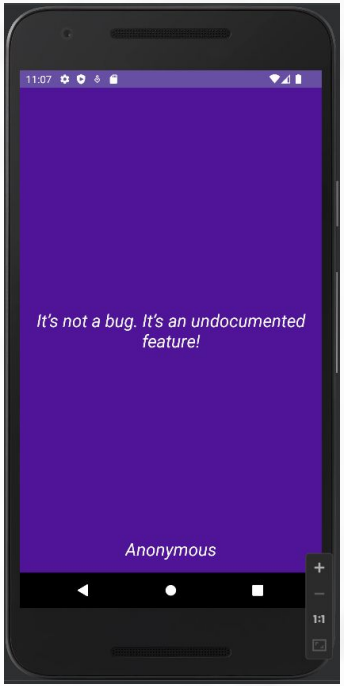
8. DAGGER HILT

Además, dentro de este módulo, creamos también el método que implemente la interfaz de acceso a la API, haciendo uso del objeto Retrofit que ahora podemos instanciar desde cualquier parte del código.

```
@Singleton
@Provides
fun provideQuoteApiClient(retrofit: Retrofit): QuoteApiClient {
    return retrofit.create(QuoteApiClient::class.java)
}
```

8. DAGGER HILT

Si ahora ejecutamos la aplicación, veremos que funciona perfectamente porque *Dagger* permite que haya inyección de dependencias solo en las clases que necesitemos. Pero lo ideal es que preparemos todo el proyecto para que todas las clases puedan ser inyectadas.



Por eso, vamos a modificar nuestro *QuoteProvider.kt*:

```
class QuoteProvider @Inject constructor() {
    var quotes:List<QuoteModel> = emptyList()
}    //Hemos eliminado el companion object
```

8. DAGGER HILT

Pero ahora no podemos acceder al atributo *quotes* desde la clase *QuoteRepository.kt* y lo que tenemos que hacer es inyectarle la clase *QuoteProvider* por parámetro.

```
class QuoteRepository @Inject constructor(
    private val api: QuoteService,
    private val quoteProvider: QuoteProvider
) {
    suspend fun getAllQuotes(): List<QuoteModel> {
        val response: List<QuoteModel> = api.getQuotes()
        quoteProvider.quotes = response
        return response
    }
}
```

8. DAGGER HILT

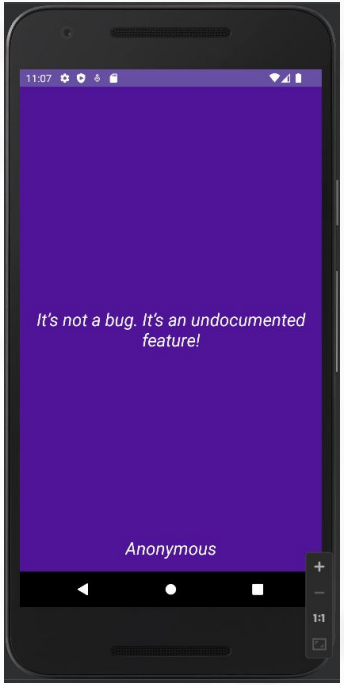
Y lo mismo para el caso de uso *GetRandomQuoteUseCase.kt*.

```
class GetRandomQuoteUseCase @Inject constructor(
    private val repository: QuoteRepository,
    private val quoteProvider: QuoteProvider
) {
    operator fun invoke(): QuoteModel? {
        val quotes = quoteProvider.quotes
        if (!quotes.isNullOrEmpty()) {
            val randomNumber = (quotes.indices).random()
            return quotes[randomNumber]
        }
        return null
    }
}
```

8. DAGGER HILT

Volvemos a ejecutar pero ahora, si el usuario pulsa en la pantalla, no se muestra una cita aleatoria. Esto se debe a que estamos instanciando *QuoteProvider* dos veces.

Desde *QuoteRepository* se creará un objeto *QuoteProvider* y se llenará su lista *quotes* con las citas obtenidas de la API. Pero desde *GetRandomQuoteUseCase*, que contiene el método de actualización de citas, se creará otro objeto con la lista vacía que tiene inicialmente *QuoteProvider*.

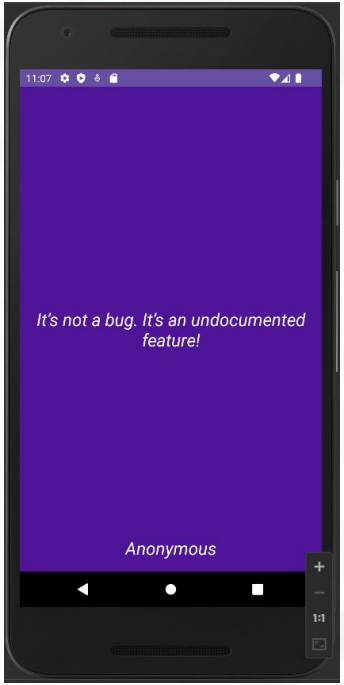


8. DAGGER HILT

La solución es convertir esta clase *QuoteProvider* en una clase *singleton*.

```
@Singleton
class QuoteProvider @Inject constructor() {
    var quotes:List<QuoteModel> = emptyList()
}
```

Ahora que tenemos implementada la inyección de dependencias, vamos a pasar al uso de base de datos con Room.



9. ROOM

Incluimos en *build.gradle.kts (Module: app)* las dependencias propias de **Room**. Además, en el directorio */data* creamos el directorio */database* y, dentro de este último, el directorio */entities*.

```
dependencies {
    ...
    // Dagger Hilt
    implementation ("com.google.dagger:hilt-android:2.50")
    kapt("com.google.dagger:hilt-compiler:2.50")
    //Room
    implementation ("androidx.room:room-ktx:2.6.1")
    kapt ("androidx.room:room-compiler:2.6.1")
    ...
}
```

9. ROOM

Dentro de */entities* vamos a crear la **clase** *QuoteEntity.kt* que generará la tabla donde vamos a almacenar las citas recogidas de la API. Para cada registro de esta tabla solo será necesario pasarle los campos *quote* y *author*, ya que su *id* será autogenerada y la inicializamos a 0.

```
@Entity(tableName = "quote_table")
data class QuoteEntity(
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "id") val id: Int = 0,
    @ColumnInfo(name = "quote") val quote: String,
    @ColumnInfo(name = "author") val author: String
)
```

9. ROOM

Lo siguiente es crear el *DAO* (*Data Access Object*), dentro del directorio */dao* en */database*. Éste lo definiremos en una **interfaz** *QuoteDao.kt*.

```
@Dao
interface QuoteDao {
    @Query("SELECT * FROM quote_table ORDER BY author DESC")
    suspend fun getAllQuotes():List<QuoteEntity>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertAll(quotes:List<QuoteEntity>)

    @Query("DELETE FROM quote_table")
    suspend fun deleteAllQuotes()
}
```

9. ROOM

Y por último, debemos crear la **clase** *QuoteDatabase.kt* para la base de datos. Ésta la dejaremos en el directorio */database* directamente.

```
@Database(entities = [QuoteEntity::class], version = 1)
abstract class QuoteDatabase: RoomDatabase() {

    abstract fun getQuoteDao(): QuoteDao
}
```

Así tenemos todos los componentes, que explicábamos al principio, necesarios para implementar una base de datos local con acceso a la tecnología SQLite.

9. ROOM

Estas clases las vamos a inyectar con Hilt, así que nos vamos a nuestro directorio */data/di* y creamos el **objeto** *RoomModule.kt*.

```
@Module
@InstallIn(SingletonComponent::class)
object RoomModule {
    private const val QUOTE_DATABASE_NAME = "quote_database"

    @Singleton    @Provides
    fun provideRoom(@ApplicationContext context: Context) =
        Room.databaseBuilder(context , QuoteDatabase::class.java, QUOTE_DATABASE_NAME ).build()
    @Singleton    @Provides
    fun provideQuoteDao(db: QuoteDatabase) = db.getQuoteDao()
}
```

9. ROOM

Ahora, en *QuoteRepository.kt* vamos a cambiar la inyección de la clase *QuoteProvider* por el DAO que acabamos de crear.

```
class QuoteRepository @Inject constructor(
    private val api: QuoteService,
    private val quoteDao: QuoteDao
) {

    suspend fun getAllQuotes(): List<QuoteModel> {
        val response: List<QuoteModel> = api.getQuotes()
        quoteProvider.quotes = response
        return response
    }
}
```

9. ROOM

Y en vez de un solo método, va a tener dos: uno que importe las citas de la API y otro que las importe de la base de datos.

```
class QuoteRepository @Inject constructor(
    private val api: QuoteService,
    private val quoteDao: QuoteDao
) {
    suspend fun getAllQuotesFromApi(): List<QuoteModel> {
        val response: List<QuoteModel> = api.getQuotes()
        quoteProvider.quotes = response
        return response
    }
    suspend fun getAllQuotesFromDatabase(): List<Quote> {}
}
```

9. ROOM

Dentro del directorio */domain* vamos a crear un directorio */model* con una **clase** *Quote.kt* similar a *QuoteModel*. La razón de crear clases semejantes en las distintas capas del código es para evitar tener que hacer múltiples modificaciones si en la aplicación se realiza algún cambio a nivel de datos (Retrofit, Room,...).

Esta clase solo interactuará con la capa de dominio y la capa de interfaz de usuario. Será el modelo de datos final, mientras que *QuoteModel* será el modelo de datos a nivel de capa de entidades.

9. ROOM

La respuesta que obtenemos de Retrofit es de tipo *QuoteModel*, mientras que el modelo final con el que vamos a trabajar es tipo *Quote*. Esto es habitual entre los modelos que recibimos del servidor y los que suministramos a la UI. Por eso existen los *mappers*, que transforman un tipo de objeto en otro mediante su método `toDomain()`. Implementamos este método tanto para *QuoteModel* como para *QuoteEntity*.

```
data class Quote (val quote:String, val author:String)
// Los mapper se declaran en la clase final donde queremos heredar
fun QuoteModel.toDomain() = Quote(quote, author)    //Quote hereda los atributos
quote y author de QuoteModel
fun QuoteEntity.toDomain() = Quote(quote, author)
```

9. ROOM

Si volvemos a *QuoteRepository*, hacemos un mapeo de cada elemento tipo *QuoteModel* obtenido con Retrofit y lo transformamos a tipo *Quote*.

```
class QuoteRepository @Inject constructor(
    private val api: QuoteService,
    private val quoteDao: QuoteDao
) {
    suspend fun getAllQuotesFromApi(): List<Quote> {
        val response: List<QuoteModel> = api.getQuotes()
        return response.map { it.toDomain() }
    }
    suspend fun getAllQuotesFromDatabase(): List<Quote> {}
}
```

9. ROOM

Y para los elementos obtenidos desde la base de datos hacemos lo mismo, pero a través del DAO.

```
    suspend fun getAllQuotesFromApi(): List<Quote> {
        val response: List<QuoteModel> = api.getQuotes()
        return response.map { it.toDomain() }
    }
    suspend fun getAllQuotesFromDatabase(): List<Quote> {
        val response: List<QuoteEntity> = quoteDao.getAllQuotes()
        return response.map { it.toDomain() }
    }
}
```

9. ROOM

Pero ahora nos toca corregir el caso de uso *GetQuotesUseCase*, que toma las citas de la API solo una vez cuando arranca la aplicación.

```
    suspend operator fun invoke():List<Quote>{
        val quotes = repository.getAllQuotesFromApi()

        return if(quotes.isNotEmpty()){
            repository.clearQuotes()
            repository.insertQuotes(quotes.map { it.toDatabase() })
            quotes
        }else{ //Si falla la llamada a la API, se cargan las citas desde la base de datos
            repository.getAllQuotesFromDatabase()
        }
    }
```

9. ROOM

Al invocar este caso de uso en el método `onCreate()` de *QuoteViewModel* estamos devolviendo el listado de citas desde la API y, a su vez, almacenándolo en la base de datos mediante los métodos `clearQuotes()` e `insertQuotes()`, que deberemos definir en la clase *QuoteRepository*:

```
suspend fun insertQuotes(quotes:List<QuoteEntity>){
    quoteDao.insertAll(quotes)
}
suspend fun clearQuotes(){
    quoteDao.deleteAllQuotes()
}
```

9. ROOM

Pero además, el modelo obtenido desde la API debemos transformarlo al modelo *QuoteEntity* que almacenar en la base de datos y, para ello, lo que hacemos es invocar al método `toDatabase()` en dicha clase.

```
@Entity(tableName = "quote_table")
data class QuoteEntity(
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "id") val id: Int = 0,
    @ColumnInfo(name = "quote") val quote: String,
    @ColumnInfo(name = "author") val author: String
)
// Hay que especificarle los campos porque también existe el campo "id"
fun Quote.toDatabase() = QuoteEntity(quote = quote, author =  author)
```

9. ROOM

Si estudiamos el resto de clases de nuestro de proyecto, veremos que en *QuoteViewModel* creábamos un objeto LiveData tipo *QuoteModel*. Este objeto ahora debe ser tipo *Quote*, ya que pertenece a la capa de UI.

```
@HiltViewModel
class QuoteViewModel @Inject constructor(
    private val getQuotesUseCase:GetQuotesUseCase,
    private val getRandomQuoteUseCase:GetRandomQuoteUseCase
) : ViewModel() {

    val quoteModel = MutableLiveData<QuoteModel>()
    val isLoading = MutableLiveData<Boolean>()
```

9. ROOM

La última clase que debemos modificar es *GetRandomQuoteUseCase* para que al invocarla tome una cita aleatoria de la base de datos.

```
class GetRandomQuoteUseCase @Inject constructor(private val repository: QuoteRepository) {

    suspend operator fun invoke(): Quote? {
        val quotes = repository.getAllQuotesFromDatabase()
        if (!quotes.isNullOrEmpty()) {
            val randomNumber = (quotes.indices).random()
            return quotes[randomNumber]
        }
        return null
    }
}
```

9. ROOM

Esta última modificación implica que el método `invoke()` del caso de uso se ha de ejecutar en una corrutina y, por lo tanto, debemos implementarla al utilizarlo en nuestro *QuoteViewModel*.

```
fun randomQuote() {
    viewModelScope.launch {
        isLoading.postValue(true)
        val quote = getRandomQuoteUseCase()
        if (quote != null) {
            quoteModel.postValue(quote)
        }
        isLoading.postValue(false)
    }
}
```

9. ROOM

Y ahora reza todas las oraciones que te sepas y ejecuta tu aplicación. Si todo ha ido bien, funcionará perfectamente. Si no inicia,es que Android te odia y debes rezar más fuerte.

Puedes intentar migrar de *kapt* a *ksp* como se indica en este enlace y, además, actualizarlo a su última versión antes de sincronizar. También puedes limpiar el proyecto desde la opción *Build → Clean Project*. Por último, puedes reiniciar AndroidStudio o tirar el ordenador por la ventana.