



# LiveData and UI - Parte I

---

## Patrón Observador y Ciclos de vida de una actividad y fragmento

---

### Competencias:

- Conocer qué es el patrón observador.
- Comprender el ciclo de vida de una actividad o fragmento.
- Construir un proyecto que registre los ciclos de vida de una actividad.
- Conocer qué es LifeCycle awareness.

### Introducción

Antes de comenzar a explicar y utilizar LiveData, debemos recordar un patrón de diseño llamado Observador. Recordemos que los patrones de diseño son referencias para la construcción de software y podemos encontrar mucha información y literatura acerca de cómo nos ayudan y cuando utilizarlos, por ejemplo el famoso libro *Patrones de diseño: Elementos reutilizables en programación orientada a objetos*.

Este libro ha sido una gran referencia para el diseño y utilización de patrones de diseño en ingeniería de software, es recomendable tenerlo como referencia de consulta cada vez que estamos utilizando un patrón o definiendo una solución para nuestro problema.

Tener un entendimiento de cómo funcionan las herramientas a este nivel abstracto, nos permitirá que conozcamos más en profundidad el porqué de las cosas. Y más importante aún las bases con las cuales están construidas herramientas que nos facilitaran la vida como desarrolladores.

En el caso de esta unidad, conocer cómo funcionan internamente LiveData o clases y componentes que son Life Cycle awareness, nos permitirán explicar cómo nuestras aplicaciones pueden realizar tareas complejas bajo un nuevo punto de entendimiento.

## El patrón de diseño Observador.

El Patrón observador está calificado como un patrón de comportamiento, este tipo de patrón se encarga principalmente en la comunicación entre clases y objetos en un programa. El patrón observer está recomendado para cuando necesitamos reaccionar a los cambios que ocurren a un objeto, pero sin la necesidad de que estemos constantemente preguntando por el estado de ese objeto. El objeto se encarga de avisar cuando ocurre un cambio en su estado, notificando a todos los que están observando esos cambios.

## Un problema que resuelve el patrón observador: Interfaces gráficas

A medida que avanzó la computación, diferentes cambios llevaron a pensar cómo diseñar nuestros programas de manera más eficiente e inteligente. Cuando aparecen las interfaces gráficas, se nos presenta el problema de que los **programas monolíticos** no escalan de manera correcta para atender este nuevo componente. Debido a que la interfaz podría cambiar por diferentes motivos, no había forma de obtener estos cambios, sin estar constantemente preguntando si los valores habían cambiado. El comportamiento ideal que debería tener el programa en este tipo de casos es:

1. La interfaz reacciona a los cambios, no pregunta por ellos
2. El objeto que cambia debe avisar a los interesados cada vez que cambia
3. Cada vez que los interesados son notificados de los cambios, se procede a actualizar la interfaz que esperaba esos cambios.
4. La relación entre un Sujeto y sus observadores es 1 - N (uno a muchos).

Para obtener este comportamiento se define en el patrón observador los siguientes elementos:

- **Subject:** Es considerado el que mantiene la información, los datos o la lógica de negocios del programa. Es la fuente de la “verdad”.
- **Observer:** Es un interesado en los datos, y quiere ser notificado cuando estos cambien. Es el que reacciona ante el evento gatillado
- **Register/Attach:** Los observers se registran en la lista de observadores del Subject, para ser notificados cuando ocurra un cambio.

- **Event:** Los eventos gatillan cambios en el Subject, todos los observers serán notificados de esos cambios.
- **Notify:** Dependiendo de qué tipo de implementación se haya realizado del patrón el Subject puede empujar la nueva información al observer, o el observer puede rescatar la nueva información desde el Subject, este proceso dependiendo del tipo de implementación será transparente para nosotros.
- **Update:** Los observers actualizan sus estados de manera independiente y sin verse afectado por otros observers.

Podemos simplificar el patrón observador de manera muy abstracta indicando que tendremos un tema de interés (Subject). El cual puede ser un dato en la BBDD, o un cambio de estado de algún elemento. Y luego quien esté interesado en escuchar o saber de los cambios que puedan producirse, deberá suscribirse para convertirse en Observadores(Observers).

Teniendo todo esto en cuenta, podemos observar el siguiente diagrama para ver una representación gráfica abstracta del patrón:

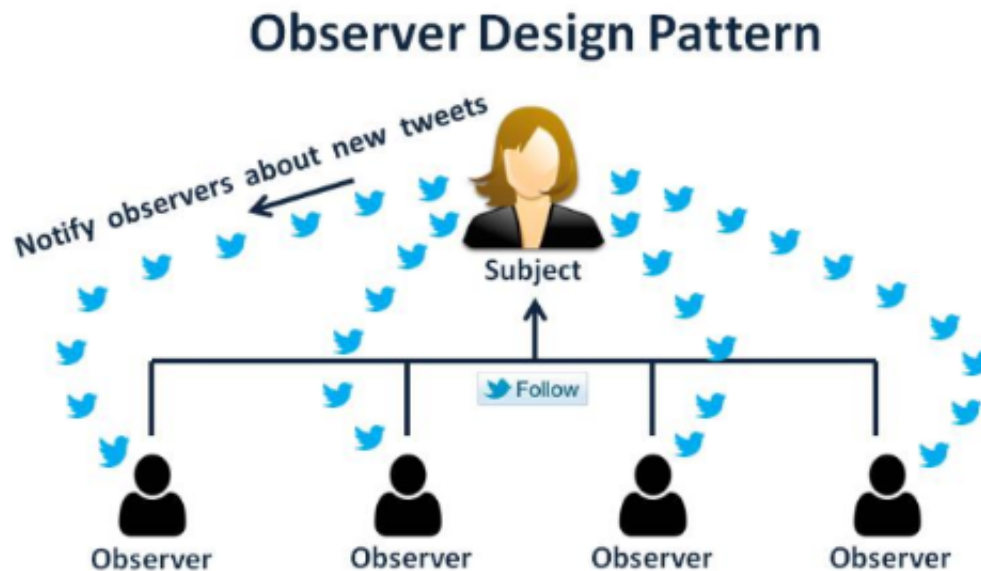


Imagen 1. Patrón observador.

Cómo podemos ver Twitter es un muy buen ejemplo abstracto de cómo funciona el patrón observador. Tenemos un Subject, que twittea su estado, el cual es visto por todos sus Observers, seguidores, los cuales son notificados de ese nuevo twit.

A nivel más técnico la implementación, normalmente, definimos dos interfaces una para el Subject y otra para el Observer, luego los objetos concretos implementan estas interfaces. En el siguiente diagrama podemos ver esta implementación más técnica:

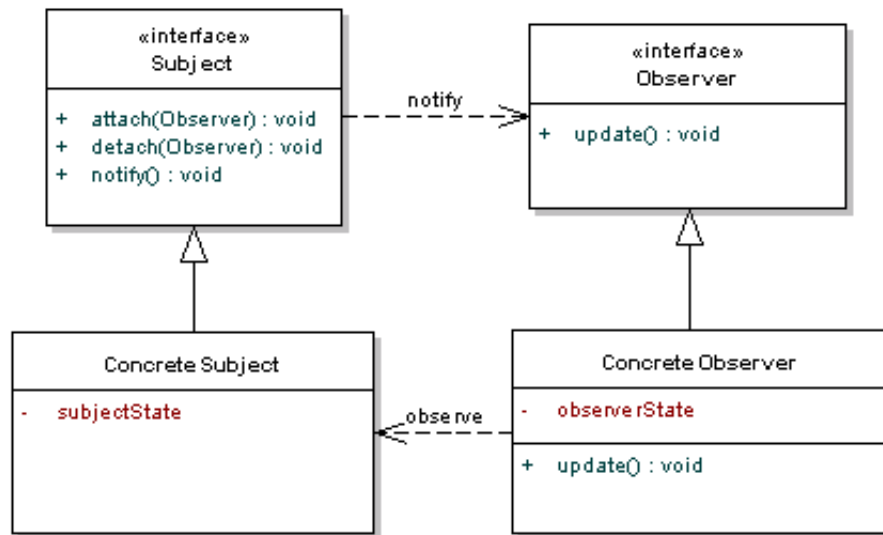


Imagen 2. Diagrama del comportamiento de Twitter.

Ahora que sabemos algunos detalles de cómo funciona este patrón pasaremos a explicar otro concepto muy importante para el desarrollo de aplicaciones móviles Android.

## Ciclo de vida de una Actividad y Fragmento.

Sabemos que una actividad es un punto de partida en una aplicación Android, también es asociada con una representación de una vista cuando se asocia a un Layout. También podemos afirmar que una actividad tiene distintos estados y siempre está asociada a un ciclo de vida el cual llama a diferentes métodos cuando va cambiando de uno a otro, nosotros incluso podemos aprovechar estos métodos para ejecutar código que beneficie las funciones de nuestra aplicación.

Este ciclo de vida está definido en el framework de Android desde sus inicios. Este ciclo cuenta con 6 métodos principales, que son llamados a medida que la Actividad avanza en su ciclo de vida, los métodos son:

- **onCreate:** Este método se llama cuando la actividad es creada por primera vez. En este método debemos unir la interfaz, el layout xml a la actividad. Debemos configurar todas las variables iniciales, valores, y otras cosas que queremos que el usuario vea cuando la aplicación es desplegada. Debemos evitar realizar tareas grandes en este método, ya que entre más tiempo lleve esa tarea, más se demora en aparecer en pantalla la interfaz de la actividad.

- **onStart:** Este método se llama cuando la actividad se encuentra en estado started, ya es visible para el usuario, pero este no puede interactuar con ella. Normalmente en este método podemos gatillar tareas asíncronas, que no interfieren con el ciclo de la aplicación, y pueden cargar información cuando terminen sin bloquear la interfaz de usuario.
- **onResume:** Este método nos señala que la aplicación se encuentra en estado resumed, lo que básicamente significa que la actividad es visible, está ocupando la pantalla y el usuario puede interactuar con ella. En este estado es cuando normalmente se ejecutan las diferentes funciones interactivas de la aplicación, por ejemplo poder hacer like a fotos o escribir un correo.
- **onPaused:** este método se llama cuando la aplicación pierde el foco principal y entra en estado paused. Básicamente el usuario está dejando la actividad actual. Esto ocurre cuando gatillamos otra actividad, aparece un dialog o recibimos una llamada de teléfono.
- **onStop:** este método es llamado cuando la actividad entró en estado stopped. Esto significa que ya no es visible para el usuario, y que podría ser destruida por el sistema. La actividad ya no se encuentra en la pantalla del dispositivo.
- **onDestroy:** este método es el último que se llama en el ciclo de vida de una actividad. Este método se llama justo antes que la actividad sea terminada por el sistema operativo.

Este es a grandes rasgos el ciclo de vida de una Actividad, si bien estos métodos son los principales, debe ser claro que se cuenta con muchos más métodos y otras consideraciones cuando trabajamos con actividades en Android.

También es importante entender que los Fragments también poseen un ciclo de vida que está altamente ligado al ciclo de vida de la actividad que lo soporta.

Los estados y métodos más relevantes en el ciclo de vida de los fragmentos son los siguientes:

- onAttach():
- onCreate():
- onCreateView():
- onPause():

Existen otros, pero por el alcance de esta unidad basta con mencionar los principales.

En el siguiente diagrama podemos ver visualmente cual es la secuencia del ciclo de vida de una actividad y fragmento, también cómo interactúan los diferentes métodos.

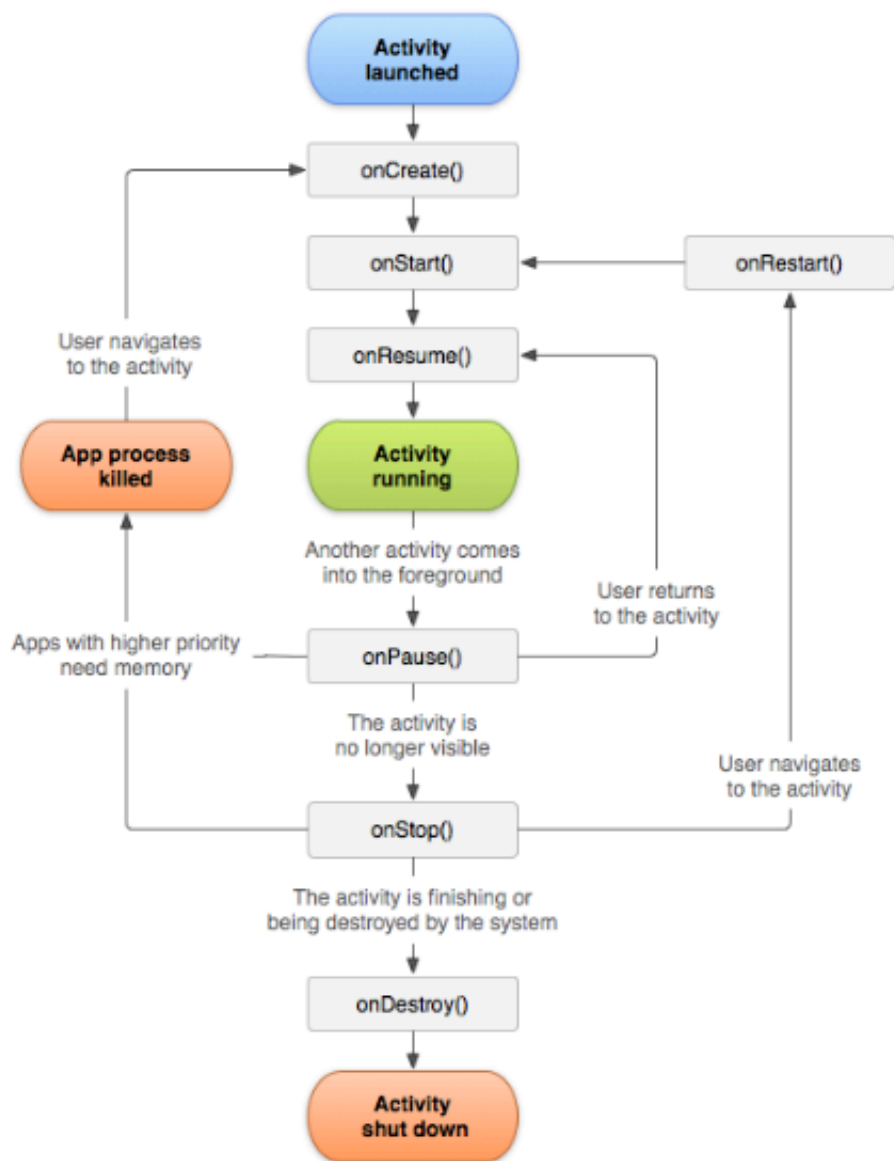


Imagen 3. Ciclo de vida de Actividad

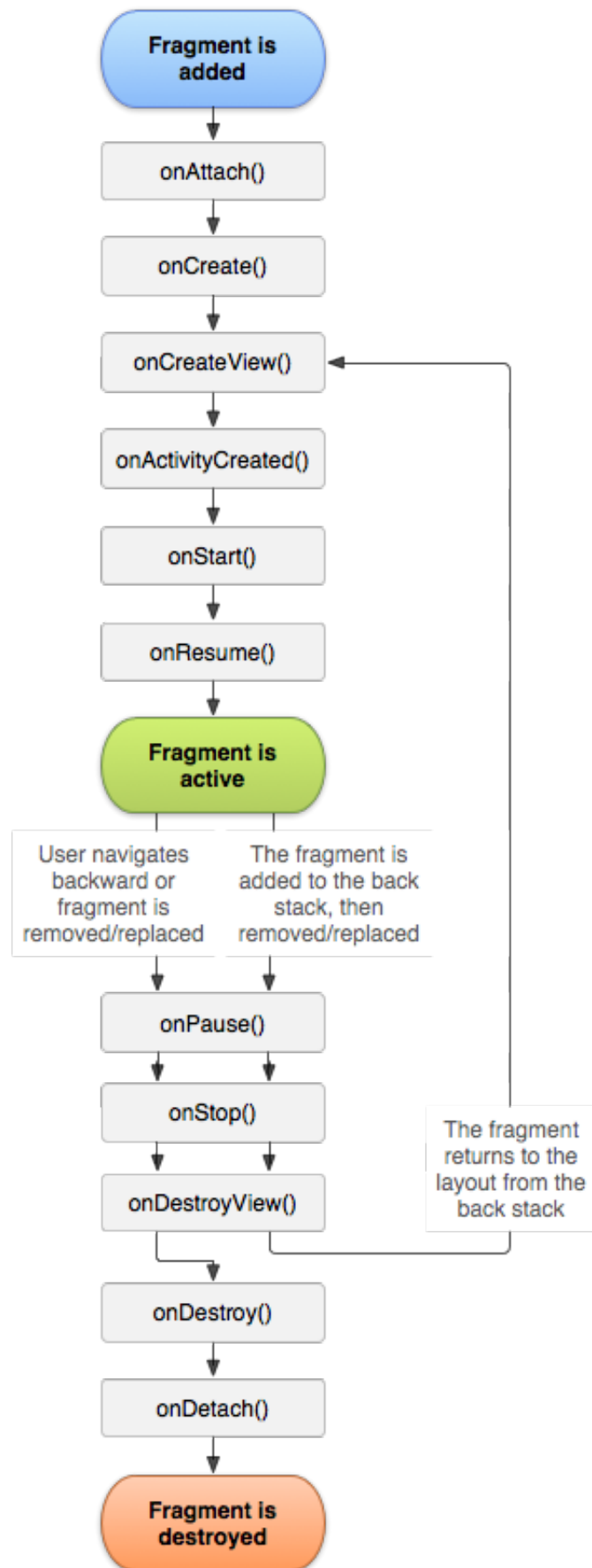


Imagen 4. Ciclo de vida de un Fragmento.

# Manejo Manual de los ciclos de Vida

Como desarrolladores de aplicaciones Android debemos estar preocupados de que nuestras interfaces gráficas o UI mantengan sus datos lo más actualizado posibles o que se adapten a los cambios del ciclo de vida de la actividad o fragmento que se está mostrando, de forma que no pierdan información cuando ocurre un cambio en la configuración, el cual puede ocurrir simplemente rotando el dispositivo.

Existen varias alternativas para solventar estos problemas como por ejemplo:

- Manejar los datos en los componentes de UI (Objeto Dios)
- Usar Listeners
- Usar alguna alternativa de “eventBus”.

Sin embargo muchas de estas alternativas, nos dan un trabajo mayor ya que tenemos que estar preocupados de manejar manualmente el ciclo de vida y los posibles cambios que puedan ocurrir, además de añadir en algunos casos librerías de terceros.

## LifeCycle-Aware components

Gracias a los nuevos componentes de arquitectura ya no tenemos que preocuparnos del manejo de los ciclos de vida como hace algunos años, debido a que elementos como Lifecycle classes se crearon para que los componentes puedan manejar el ciclo de vida en el cual están inmersos y sobrevivir a los cambios de configuración sin tener que realizar tareas extras por nuestra parte.

Estamos mencionando estos componentes conscientes de su ciclo de vida para que quede claro como LiveData y ViewModel manejan estos cambios sin necesidad de que participemos o escribamos código para ello.

## Lifecycle y LifecycleOwner

Como mencionamos, Android cuenta con componentes que son *Lifecycle aware*, lo cual significa que estos son capaces de responder o reaccionar a cambios en el ciclo de vida de otros componentes como Actividades o Fragmentos.

Ahora explicaremos los elementos que están presentes en estos componentes y que permiten que estos tengan esa clase de comportamiento.



## Lifecycle

Es una clase que mantiene información acerca del estado del ciclo de vida de un componente, una actividad o un fragmento. Esta clase ocupa dos enumeraciones principales para mantener un registro de estado del ciclo de vida del componente asociado a la clase.

- Event: Los eventos del ciclo de vida que son despachados desde el framework de Android y la clase Lifecycle. Estos eventos mapean los callbacks del ciclo de vida de las actividades y los fragments.
- State: El estado actual del componente que es trackeado por el objeto Lifecycle.

En el siguiente diagrama podemos ver una representación gráfica del ciclo de vida completo de un componente y los diferentes estados del mismo. Observa como los estados en el lifecycle se producen inmediatamente después de los métodos de la Actividad hasta onResume, pero luego se ejecutan de forma anterior a los métodos desde onPause en adelante hasta el final del ciclo de vida de la actividad.

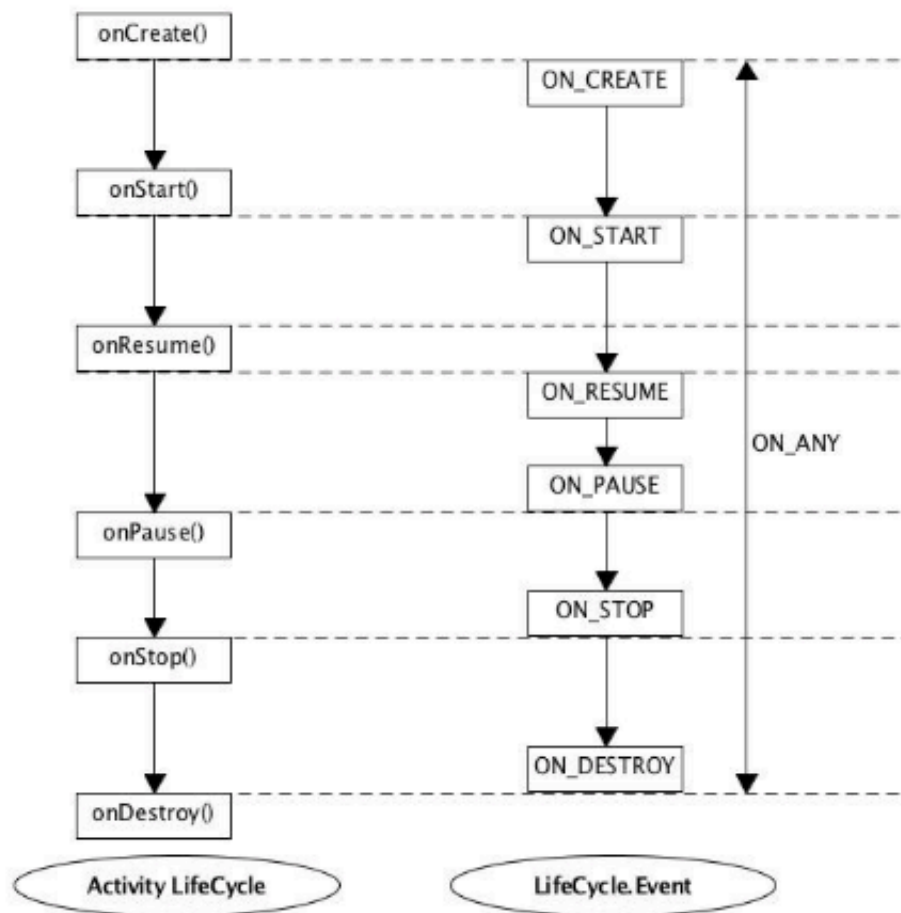


Imagen 5. Diagrama de flujo del ciclo de vida de una actividad.

Las clases que implementen Lifecycle pueden observar los estados de un LifecycleOwner y reaccionar a esos cambios.

## LifecycleOwner y LifecycleObserver

Un Lifecycle owner es cualquier componente que implementa la interfaz LifecycleOwner, esta interfaz indica que tiene un ciclo de vida en Android. Los Fragmentos y las Actividades implementan esta interfaz desde la biblioteca de soporte 26.1.0.

Se pueden crear LifecycleOwners custom, implementando la interfaz LifecycleRegistry que le permite a el componente custom manejar múltiples observers.

Un LifecycleObserver es un componente que observa los diferentes estados asociados a un LifecycleOwner, reaccionando a los diferentes cambios.

## Componentes Lifecycle, LifecycleOwner, LifecycleObserver, LifecycleRegistry

Veremos en acción estas clases en el siguiente capítulo, por ahora veamos sus componentes principales.

- **LifeCycle:**

- **Lifecycle.Event:** Enum que enumera los distintos estados del ciclo de vida ON\_CREATE, ON\_START, ON\_RESUME, ON\_PAUSED, ON\_STOP, ON\_DESTROY y un valor especial que sirve para comparar con cualquier evento ON\_ANY.
  - **Lifecycle.State:** enum que enumera los diferentes estados que tiene el LifecycleOwner asociado. En este caso tenemos lo siguiente
    - **CREATED:** estado después de onCreate y justo antes de onStop
    - **DESTROYED:** Estado final, justo antes de onDestroy, no se despachan más eventos después de este estado.
    - **INITIALIZED:** Es el estado cuando la actividad ha sido construida, pero no se ha llamado a su método onCreate.
    - **RESUMED:** estado después del llamado a onResume
    - **:** estado después de onStart y justo antes de onPause
    - Tiene 3 métodos asociados: isAtLeast, valueOf y values. El primero nos dice si el estado actual es al menos el valor que le pasamos como parámetro.
  - **addObserver:** método que recibe un LifecycleObserver y que será notificado cuando se produzca algún cambio de estado en el ciclo de vida.
  - **getCurrentState:** nos entrega el estado actual del ciclo de vida del componente asociado.
  - **removeObserver:** remueve el observador de la lista de observers. Recibe un LifecycleObserver.
- **LifecycleOwner:** es una interfaz que tiene un sólo método, este método, getLifecycle(), nos entrega el Lifecycle del LifecycleOwner. De esta forma clases lifecycle aware pueden observar los cambios en ese componente.

- **LifecycleObserver**: interfaz que marca una clase como Observador del ciclo de vida de otro componente. Esta interfaz ya no se utiliza, había que implementar un DefaultLifecycleObserver, actualmente se anotan los métodos con la siguiente anotación:
- **OnLifecycleEvent**: esta anotación nos dice qué evento está observando el método anotado. Por ejemplo `@OnLifecycleEvent(Lifecycle.Event.ON_CREATE)` nos señala que el método anotado reaccionará al evento `ON_CREATE`.
- **LifecycleRegistry**: Esta clase es una implementación de Lifecycle que permite manejar múltiples observadores. Se utiliza para implementar los custom LifecycleOwners, cuenta con los siguientes componentes:
  - **Constructor** que recibe un LifecycleOwner que provee el ciclo de vida
  - **addObserver**: para agregar observers, recibe un LifecycleObserver.
  - **getCurrentState**: retorna el estado actual del ciclo de vida
  - **getObserverCount**: retorna el conteo de observadores de este registro.
  - **handleLifecycleEvent**: establece el estado actual del ciclo de vida, recibe un Lifecycle.Event. Notifica a los observers.
  - **markState**: recibe un Lifecycle.State, se mueve a ese estado y despacha todos los eventos necesarios a los observadores.
  - **removeObserver**: recibe un LifecycleObserver, lo remueve de la lista de observadores de este componente.

A continuación realizaremos un pequeño proyecto de ejemplo en el cual registramos estos estados en el LogCat de android studio, esto nos permitirá tener un mayor entendimiento de los diferentes estados del ciclo de vida de una actividad.

## Proyecto Registro de estados

A continuación realizaremos un proyecto de ejemplo el cual será llamado EjemploLifecycle.

### 1. Crea un nuevo proyecto en Android Studio.

- a. Utiliza Kotlin como lenguaje.
- b. Crea una actividad vacía.
- c. utiliza una Api mínima 26.

### 2. Configurando los componentes en el proyecto

Para este caso la configuración es mínima, tendremos que agregar sólo lo siguiente a nuestro archivo gradle de dependencias del módulo app:

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
apply plugin: 'kotlin-kapt'

...
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.0.2'
    implementation 'androidx.core:core-ktx:1.0.2'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    def lifecycle_version = "2.0.0"
    implementation "androidx.lifecycle:lifecycle-livedata:$lifecycle_version"
    kapt "androidx.lifecycle:lifecycle-compiler:$lifecycle_version"
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test:runner:1.2.0'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
}
```

Lo único que debemos agregar nosotros es la variable `lifecycle_version` y la dependencia de LiveData y el procesador de anotaciones con `kapt`, lo demás fue agregado por Android Studio cuando creó el proyecto base. Recordar también agregar el plugin `kapt` cómo se ve en el código adjunto.

### 3. Registrando en un log los estados de un lifecycleOwner

- a. Crea una clase llamada EventLogger
- b. Añade el código que hemos puesto a tu disposición.

Esta clase nos permitirá loguear los distintos eventos de un LifecycleOwner.

La clase EventLogger se verá de la siguiente manera:

```
class EventLogger {
    companion object {
        private const val TAG = "EventLogs"
    }

    fun eventLog(event: Lifecycle.Event) {
        when (event) {
            Lifecycle.Event.ON_CREATE -> Log.d(TAG, "OnCreate")
            Lifecycle.Event.ON_START -> Log.d(TAG, "OnStart")
            Lifecycle.Event.ON_RESUME -> Log.d(TAG, "OnResume")
            Lifecycle.Event.ON_PAUSE -> Log.i(TAG, "onPause")
            Lifecycle.Event.ON_STOP -> Log.i(TAG, "onStop")
            Lifecycle.Event.ON_DESTROY -> Log.i(TAG, "onDestroy")
            /**
             * ON_ANY no es un ciclo de vida, lo dejaremos por si ocurre algún
            evento no cubierto
             * con las opciones anteriores.
             */
            Lifecycle.Event.ON_ANY -> Log.i(TAG, "onAny")
        }
    }
}
```

Cómo vemos la clase es simple, tiene un companion object que es la forma en que declaramos componentes estáticos en las clases de Kotlin, en este caso es el TAG que nos permite mantener un registro del Log.

También cuenta con un constructor vacío, denotado por la ausencia de paréntesis al lado del nombre de la clase.

Dentro de la clase tenemos un método principal llamado eventLog, que recibe como parámetro un Lifecycle.Event e imprime el método que fue llamado con ese evento. Dentro del método tenemos un when, para crear una iteración.

Cuando hacemos que el when maneje el evento, de tipo enum Lifecycle.Event, nos recomendará que sea exhaustivo, es decir, debemos ocupar todos los posibles valores del enum y hacer una acción asociada. No necesitamos un valor default como en un Switch.

#### 4. Creando una clase Lifecycle aware.

A continuación crearemos una clase lifecycle aware que nos permita ir mostrando en la consola los distintos estados de una actividad.

Nuestra clase debe verse así:

```
class LifecycleCustomObservation(  
    private val lifecycle: Lifecycle,  
    private val eventLogger: EventLogger  
) : LifecycleObserver {  
  
    @OnLifecycleEvent(Lifecycle.Event.ON_CREATE)  
    fun logOnCreate() {  
        eventLogger.eventLog(Lifecycle.Event.ON_CREATE)  
    }  
  
    @OnLifecycleEvent(Lifecycle.Event.ON_START)  
    fun logStart() {  
        if (lifecycle.currentState.isAtLeast(Lifecycle.State.STARTED)) {  
            eventLogger.eventLog(Lifecycle.Event.ON_START)  
        }  
    }  
  
    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)  
    fun logResume() {  
        eventLogger.eventLog(Lifecycle.Event.ON_RESUME)  
    }  
  
    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)  
    fun logPause() {  
        eventLogger.eventLog(Lifecycle.Event.ON_PAUSE)  
    }  
  
    @OnLifecycleEvent(Lifecycle.Event.ON_STOP)  
    fun logStop() {  
        eventLogger.eventLog(Lifecycle.Event.ON_STOP)  
    }  
}
```

```

@OnLifecycleEvent(Lifecycle.Event.ON_DESTROY)
fun logDestroy() {
    if (lifeCycle.currentState.isAtLeast(Lifecycle.State.DESTROYED)) {
        eventLogger.eventLog(Lifecycle.Event.ON_DESTROY)
    }
}
}

```

En esta clase podemos ver varias cosas importantes, partamos por su constructor y la interfaz que implementa.

Vemos que su constructor recibe 2 parámetros, un lifecycle que nos servirá para entender el estado en que se encuentra el LifecycleOwner, y un logger que nos permite escribir en la consola los eventos a medida que van ocurriendo.

Además esta clase implementa una interfaz LifecycleObserver, que le dice a nuestra aplicación, que esta clase puede observar cambios en el ciclo de vida de un componente.

Todos los métodos de la clase se encuentran anotados con @OnLifecycleEvent, esta anotación nos permite ser notificados de un evento en el ciclo de vida del owner, y según lo que definamos en los paréntesis de la anotación, se activará el método correspondiente.

Debemos notar que los eventos onStart y onDestroy, requieren el chequeo del estado del ciclo de vida, lifeCycle, que recibimos en el constructor. ¿Por qué ocurre esto? recordemos que los estados de un ciclo de vida no son lo mismo que los eventos de el lifeCycle observer, los estados son valores que nos indican en qué estado se encuentra el ciclo de vida, los eventos nos avisan cuando se ha llamado un cierto método del ciclo de vida.

Entonces para el evento ON\_START el ciclo de vida debe estar en por lo menos el estado STARTED, esto lo sabemos usando el método isAtLeast del Enum State de Lifecycle. Lo mismo ocurre para el evento ON\_DESTROY.

## 5. Añadiendo el observador a la Actividad

Finalmente añadiremos en la actividad el observador para loguear sus diferentes estados en la consola. La actividad principal queda de la siguiente manera:

```
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        setSupportActionBar(toolbar)  
  
        val observer = LifecycleCustomObservation(lifecycle, EventLogger())  
        lifecycle.addObserver(observer)  
    }  
}
```

La actividad crea una variable del tipo observer, entregando dos parámetros: el lifecycle de sí misma y un EventLogger().

Finalmente agregamos ese observador a los observadores del ciclo de vida de la actividad.

## 6. Ejecutando la Aplicación.

Ahora ejecutamos la aplicación, si interactuamos con ella, presionando el botón para volver al escritorio, o la cerramos con el task manager. Veremos que diferentes Logs se registraran en el LogCat de Android Studio.

Veamos las siguientes imágenes para tener idea de cómo va esta secuencia:

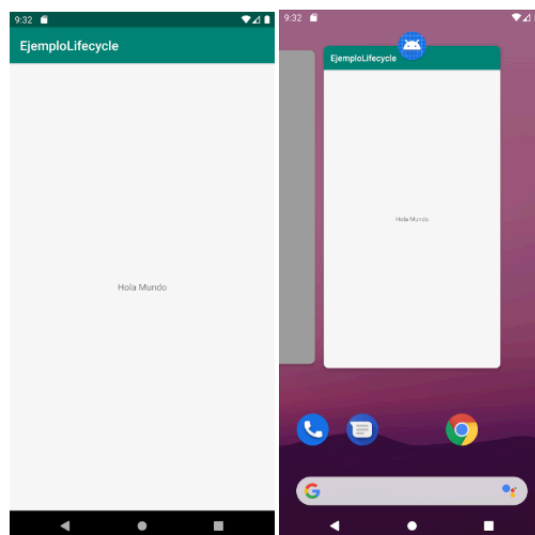


Imagen 6. Cerrando la aplicación.



Finalmente en Logcat podemos ver el los logs generados al ejecutar y cerrar la aplicación, usando el TAG “EventLogs” para encontrar más rápido y desplegar sólo los logs que queremos ver.

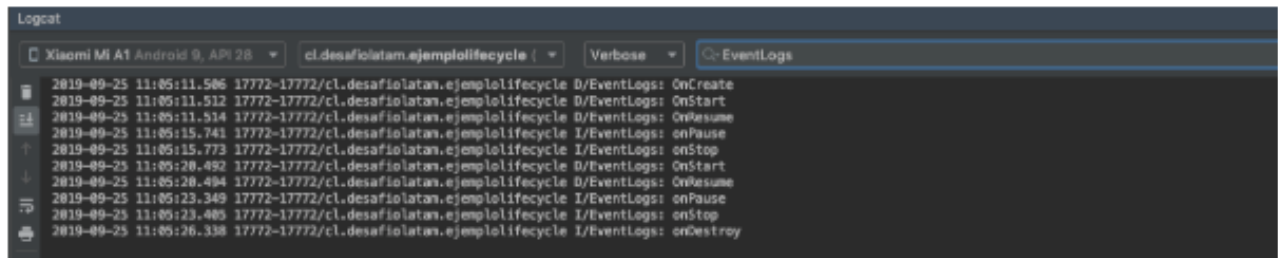


Imagen 7. Logcat.

Como puedes observar, a medida que ocurren los distintos cambios de configuración o simples pasos en los ciclos de vida, nuestra aplicación registrará estos cambios dejando un registro en el LogCat de Android Studio.

# LiveData y ViewModel

---

## Competencias:

- Entender que es LiveData
- Entender que es ViewModel
- Conocer los beneficios de usar Live Data
- Observar objetos LiveData
- Conocer MutableLiveData
- Utilizar setValue y postValue

## Introducción

Ahora que sabemos lo que es un patrón observador y también sabemos que existen diferentes ciclos de vida en los elementos de nuestra UI, elementos que hace algún tiempo deberíamos manejarlos manualmente, lo cual se vio facilitado gracias a componentes de arquitectura que son lifecycle awareness (conscientes del ciclo de vida donde están funcionando), podemos comenzar a conocer LiveData y ViewModel.

La principal motivación para comprender qué es y cómo podemos utilizar estos elementos debe ser que nos permitirán mantener nuestras interfaces de usuarios actualizadas con los datos más frescos y siempre estarán escuchando cualquier cambio que se produzca en estos, sin tener que manejar los ciclos de vida de forma manual.

Para esta unidad utilizaremos conocimientos adquiridos en unidades anteriores como el uso de Room para proporcionar una persistencia de datos y a través de LiveData y el uso de ViewModels alimentaremos nuestra UI.

## LiveData

Según la documentación oficial LiveData es una clase contenedora de datos observables. Además indica que LiveData al contrario de otros observables es consciente de los ciclos de vida, lo que quiere decir que respeta los ciclos de vida de otros componentes como por ejemplo Actividades, Fragmentos o servicios.

Si bien LiveData no es la única implementación del patrón observador en cuanto a contenedores de datos en Android, es la única implementación que es sensible a estos cambios en el ciclo de vida. Esta capacidad de entender el ciclo de vida de los observadores nos entrega una tremenda ventaja, LiveData **sólo notifica a los componentes que se encuentran en estado activo**.

LiveData considera que un observador está activo si su ciclo de vida se encuentra en STARTED o RESUMED. Sólo los objetos que cumplan con ese requisito serán notificados de los cambios. Los observadores inactivos no son notificados de dichos cambios. Esto es de mucha importancia ya que este comportamiento predeterminado ayuda mucho a que podamos en actividades o fragmentos observar datos sin tener que preocuparnos por leaks o pérdida de datos, debido a que automáticamente estos serán descartados de la suscripción de observación cuando su ciclo de vida sea destruido.

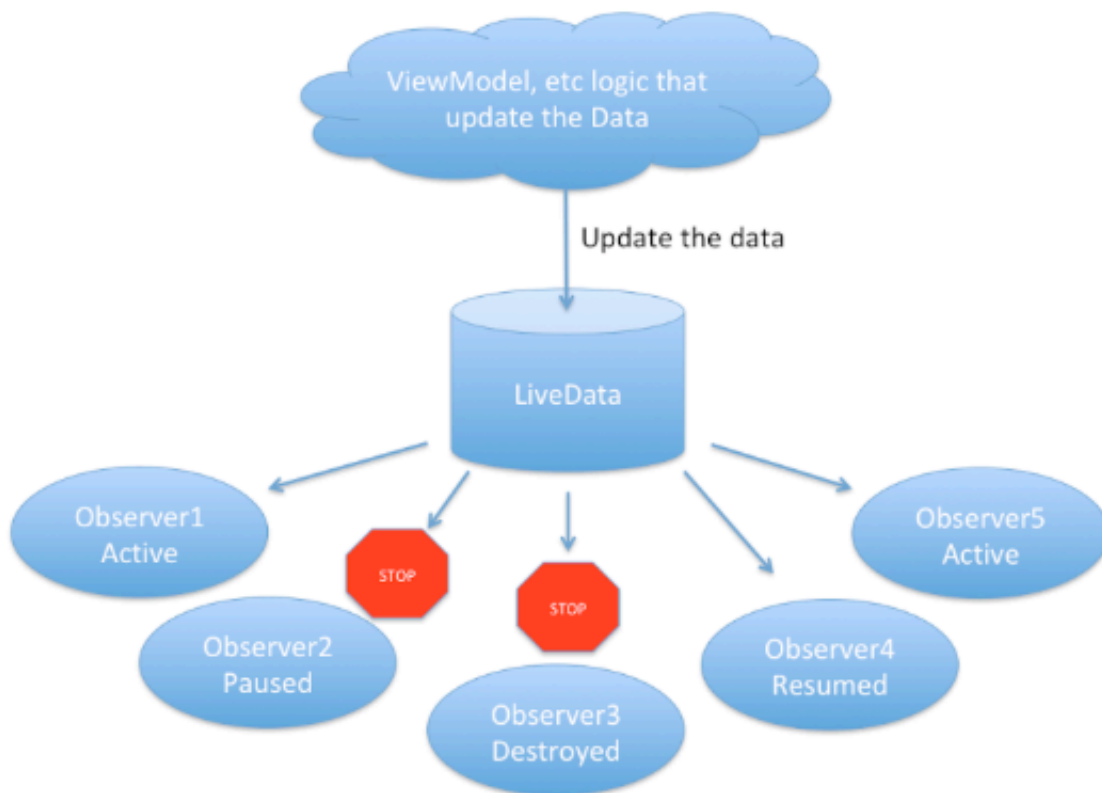


Imagen 8. Diagrama de LiveData.

Las ventajas de usar LiveData son las siguientes:

- **Asegura que la Interfaz de Usuario, UI, esté siempre actualizada:** Al seguir el patrón observador LiveData notifica los cambios a sus observadores, en este caso la interfaz gráfica. El observador será el encargado de actualizar la interfaz gráfica cuando ocurran cambios.
- **No más memory leaks:** Esto es una de las mejores ventajas, ya no debemos preocuparnos de limpiar los observadores para evitar memory leaks cuando están referenciados, pero no se usan más. Recordando que un objeto referenciado, pero que no se usa, no puede ser reciclado por el garbage collector.
- **No más caídas debido a actividades que son detenidas:** Si el ciclo de vida del observador está inactivo, los eventos no son notificados a ese observador, el cual puede ser null y generar un NullPointerException, muy común en otras implementaciones del patrón observer, cómo RxJava, si no se limpian los observadores.
- **No hay necesidad de manejar los ciclos de vida a mano:** Los componentes gráficos sólo deben observar los datos relevantes, no es necesario inscribir y reinscribir los observadores al pausar y resumir los ciclos de vida.
- **Datos siempre actualizados:** si un ciclo de vida pasa a estar inactivo, recibirá el último valor de los datos cuando vuelva a estar activo. Por ejemplo si tenemos una actividad en segundo plano, cuando vuelva a ser mostrada en primer plano, será actualizada con el último valor de los datos disponibles.
- **Manejo de cambios de configuración:** si ocurre algún cambio de configuración en el dispositivo, se rota la pantalla por ejemplo, la actividad o fragment afectado por ese cambio, recibirá inmediatamente el valor más actual de los datos.

## Utilizar LiveData objects

Existen tres pasos para utilizar objetos de LiveData:

1. Crea una instancia de LiveData para mantener algún tipo de dato, regularmente esto se realizará al interior de una clase ViewModel.
2. Crea un objeto Observador que defina cuando se ejecuta un método onChange(), el cual controla que ocurre con el objeto LiveData cuando se produce un cambio. Regularmente el objeto observador lo crearemos en algún elemento de UI, ya sea actividad o fragmento.
3. Luego debes unir al objeto observador con el de LiveData usando el método **observe()**, Esto realiza la subscripción del objeto observador al objeto Livedata para que los cambios puedan ser notificados. Regularmente el objeto observador será atado a una actividad o fragmento.

En otras palabras, cuando se actualizan los datos almacenados en un objeto **LiveData**, esto gatilla a todos los observadores registrados durante el tiempo que el **LifeCycleOwner** esté en estado activo.

De esta forma **LiveData** mantiene a los elementos de UI automáticamente recibiendo actualizaciones.

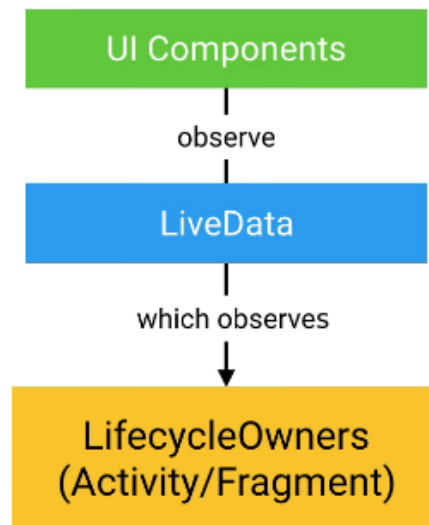


Imagen 9. LiveData mantiene los elementos de UI actualizados.

## Crear objetos LiveData

Como LiveData es un envoltorio de datos, lo podemos utilizar con cualquier tipo de objeto por ejemplo colecciones, listas etc. Los objetos LiveData regularmente son almacenados al interior de un objeto ViewModel y se puede acceder a ellos a través de métodos de getter.

```
class NameViewModel : ViewModel(){
    //Create a LiveData with a String
    val currentName: MutableLiveData<String> by lazy{
        MutableLiveData<String>()
    }
    // Rest of the ViewModel...
}
```

Más adelante realizaremos este procedimiento.

## Observar objetos LiveData

Según la documentación oficial el método más idóneo para comenzar a observar un objeto es en el método **onCreate()**. Los motivos son los siguientes.

1. Para asegurarse que el sistema no realice llamadas redundantes desde un actividad o fragmento al momento de ejecutarse el método **onResume()**
2. Para asegurar que la actividad fragmento obtenga la data y la pueda mostrar tan pronto como esté activa. Es decir tan pronto como los componentes de la app están en STARTED, puedan recibir la actualización de los datos.

Generalmente, LiveData envía actualizaciones cuando los datos cambian y solo a los observadores activos. El siguiente código muestra cómo comenzar a observar un objeto **LiveData**.

```
class NameActivity : AppCompatActivity() {  
  
    private lateinit var model: NameViewModel  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
  
        // Other code to setup the activity...  
  
        // Get the ViewModel.  
        model = ViewModelProviders.of(this).get(NameViewModel::class.java)  
  
        // Create the observer which updates the UI.  
        val nameObserver = Observer<String> { newName ->  
            // Update the UI, in this case, a TextView.  
            nameTextView.text = newName  
        }  
  
        // Observe the LiveData, passing in this activity as the LifecycleOwner and the observer.  
        model.currentName.observe(this, nameObserver)  
    }  
}
```

Imagen 10. Comenzado a observar un objeto LiveData.

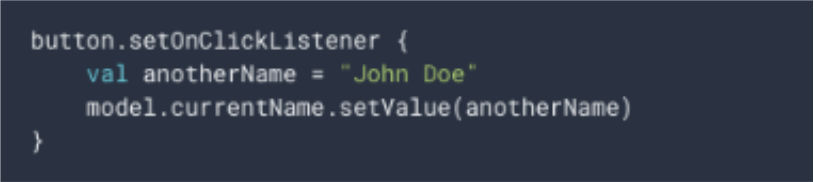
## Actualizando objetos LiveData (Update)

LiveData no tiene métodos públicos para actualizar los datos almacenados. Para esto, la clase MutableLiveData tiene a nuestra disposición los métodos setValue(T) y postValue(T), por lo tanto debemos usarla si necesitamos cambiar los datos almacenados en un objeto LiveData.

Usualmente MutableLiveData es usado en el ViewModel y el ViewModel solo expone objetos inmutables LiveData a los observadores. Para esto se utiliza la buena práctica de exponer estos objetos con un getter hacia la vista, esto ayudará a limitar la modificación de ese objeto. Esto ocurre de esta forma porque LiveData no tiene implementado los métodos setValue y postValue. Entonces si necesitáramos que se modifiquen, es mejor añadir una función que lo actualice. por ejemplo:

```
public LiveData<User> getUser(){
    return mUserInfoLiveData;
}
public void updateUser(User userUpdated){
    mUserInfoLiveData.postValue(userUpdated);
}
```

La documentación oficial indica que después que tengas establecida la relación con el observador, puedes actualizar los valores de un objeto LiveData.



```
button.setOnClickListener {
    val anotherName = "John Doe"
    model.currentName.setValue(anotherName)
}
```

Imagen 11. Actualizar valores de un objeto LiveData.

Llamar al método setValue(T) del ejemplo resultará en que los observadores llamen al método onChange() pasando el nuevo nombre("jhon Doe").

La documentación también indica que si necesitamos realizar la actualización desde el Main Thread se utiliza setValue(T) y si lo haces desde un background thread, lo realizaremos con postValue(). Ambos métodos provocarán que los observadores actualicen la UI.

## LiveData y Room

Afortunadamente la librería de persistencia de datos Room soporta consultas observables, las cuales pueden retornar objetos de LiveData. Estas consultas se escriben directamente en los DAO(Database access object).

Room generará todo el código necesario para actualizar el objeto LiveData cuando la base de datos sea actualizada. Estas consultas funcionan de forma asíncrona en un background thread cuando sean necesarias, esto permite que nuestra UI se mantenga sincronizado con los cambios en la base de datos.

En el siguiente capítulo realizaremos un ejemplo de esto.

## ViewModel

ViewModel también es un nuevo elemento presente en los componentes de arquitectura, se define como una clase llamada ViewModel, la cual es la responsable de preparar la data para ser mostrada en la UI o vista, esta clase está optimizada para respetar los ciclos de vida. Esto quiere decir que no importa que se produzca alguna rotación del dispositivo o cambio de configuración, los datos estarán disponibles para la actividad o fragmento objetivo.

En la práctica utilizaremos ViewModel en conjunto con LiveData, será en este elemento donde se almacenan esos datos para ser mostrados en la actividad o fragmento que utilizaremos como UI.

ViewModel también cuenta con un ciclo de vida, lo podemos ver en la siguiente Imagen.

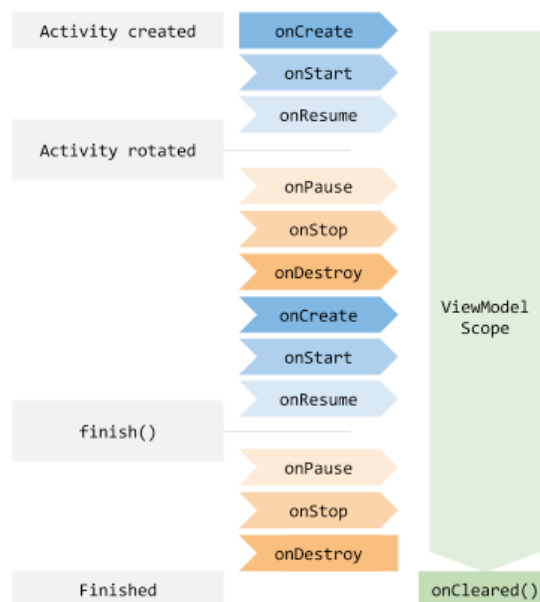


Imagen 12. Ciclo de vida.



Usualmente llamaremos al ViewModel la primera vez que el sistema llama al objeto de la actividad en el método onCreate(); El sistema podrá llamar a este método muchas veces si se produce algún cambio de configuración. Pero nuestro ViewModel ya existe hasta que la actividad sea finalizada y destruida.

En los siguientes capítulos veremos cómo implementar estos elementos en algún proyecto, mezclando diferentes componentes de arquitectura y aprovechando las características que hemos visto en este capítulo.

## **Pasando Datos entre Fragmentos con ViewModel**

En el desarrollo de aplicaciones, existen varias formas de pasar datos entre Fragmentos la más usual es a través de una Interface. Gracias a la creación de **ViewModel** existe una nueva forma de hacerlo.

Esta forma nos trae todas las ventajas como permitirnos no tener que traer data múltiples veces si existe un cambio de configuración o rotación del dispositivo. Y esto ocurre gracias a que el viewModel está asociado al ciclo de vida de la actividad.

Los pasos son simples, necesitamos crear un ViewModel con el scope de la actividad de los dos fragmentos, luego inicializar el objeto ViewModel y pasar los valores a través de un objeto LiveData.

Después de esto el otro fragmento puede observar el objeto LiveData y obtener los valores para ser mostrados en la UI

# LiveData y ViewModel en un proyecto.

---

## Competencias

- Crear un proyecto usando el patrón de diseño MVVM.
- Utilizar LiveData en un Proyecto.
- Utilizar ViewModel para el manejo de los datos.
- Utilizar Patrón repositorio en un proyecto.
- Utilizar LiveData con Room.

## Introducción

En este capítulo comenzaremos a converger nuestros conocimientos, para ello crearemos un proyecto en el cual añadiremos los componentes de arquitectura que venimos repasando hasta esta unidad.

Nos centraremos en crear la estructura para seguir un patrón de arquitectura recomendado, utilizando Room para persistencia de datos en conjunto con LiveData para alimentar nuestra UI.

El ejemplo será simple, pero demostrará todo lo que necesitamos saber de estos componentes. La idea es que todos los elementos que hemos estado estudiando trabajen juntos y tengamos una idea de cómo las aplicaciones modernas son construidas.

Durante esta unidad nos centraremos en los componente base y en la siguiente la conectaremos con la UI.

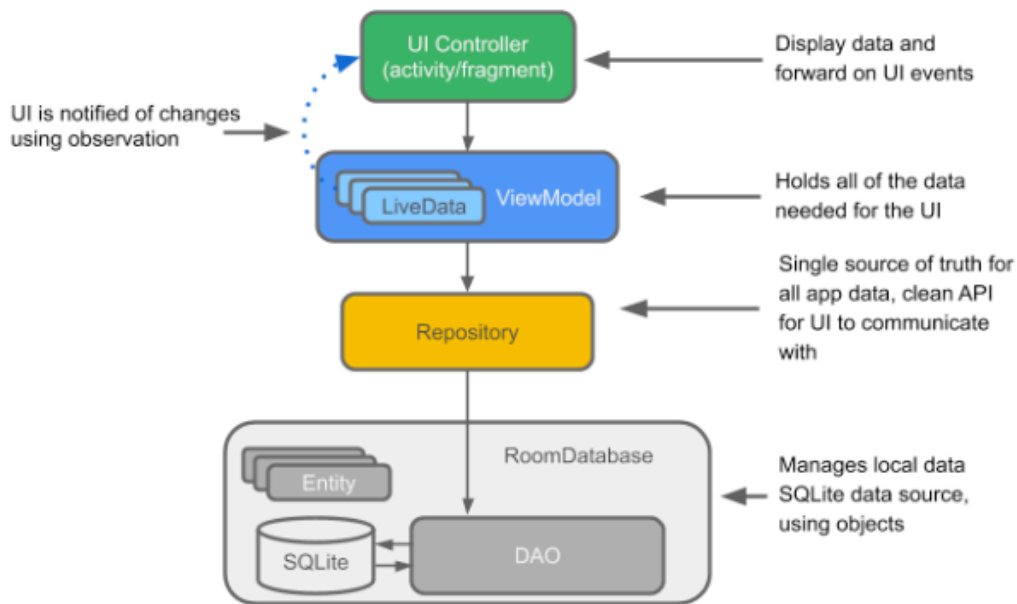


Imagen 13. Esquema del proyecto.

Este es el esquema que nuestra aplicación tendrá, como podemos observar es el mismo que propone google como alternativa a otros patrones de diseño.

A modo de resumen los participantes serán:

Nuestras Entidades describirán las tablas de la aplicación manejadas por Room.

- **SQLite database** se refiere a la persistencia de datos en el dispositivo y para facilitarnos la vida será también manejada con Room.
- **DAO** Data access object serán las encargadas de ser el puente entre nuestra aplicación y las consultas a esta base de datos, afortunadamente Room se encargará de esto.
- **Repository** Esta clase se encargará de manejar las diferentes fuentes de datos que puede tener nuestra aplicación, en el caso particular de nuestra aplicación solo maneja los datos desde la persistencia local. Pero más adelante se pueden añadir otras fuentes.
- **ViewModel** Actuará como el centro de comunicación entre el repositorio es decir los datos y nuestra UI. Nuestra UI ya no tienen que preocuparse por el origen de los datos, además la instancia del ViewModel sobrevive a los cambios de configuración.
- **LiveData** Será la clase que contenga los datos que podemos observar, Siempre tendrá la última versión de los datos y va a notificar a los observadores cuando se produzca algún cambio, además LiveData también es consciente de los ciclos de vida de la aplicación.

Los componentes de nuestra UI solo observan estos datos y no tendrán que preocuparse de otros detalles.

Si quieres saber más acerca de guías para la arquitectura de aplicaciones puedes verlo en el siguiente enlace. <https://developer.android.com/jetpack/docs/guide>

## ¿Que vamos a construir?

Respecto al alcance de esta unidad, vamos a construir una aplicación que tenga una interfaz grafica que pueda ser actualizada con datos que nosotros generemos.

La aplicación será un contador de tragos y pedidos, con ella podremos mantener un control de cuanto hemos consumido en un pub o restaurante.

- Los productos o Ítems serán almacenados en la base de datos del dispositivo.
- Mostraremos nuestro consumo a través de un RecyclerView en la MainActivity.
- Para crear un nuevo consumo deberemos presionar el botón flotante para se abra una segunda actividad para la creación.
- En esta Pantalla modificaremos los precios y la cantidad, veremos el total en tiempo real gracias a LiveData.
- Podremos borrar todos los consumos desde un botón en la actividad principal

## Imágenes de referencia

Estas son algunas pantallas de la aplicación, puedes mejorar la UI como tu quieras, solo ten cuidado de utilizar los mismos campos que mencionaremos en el código.

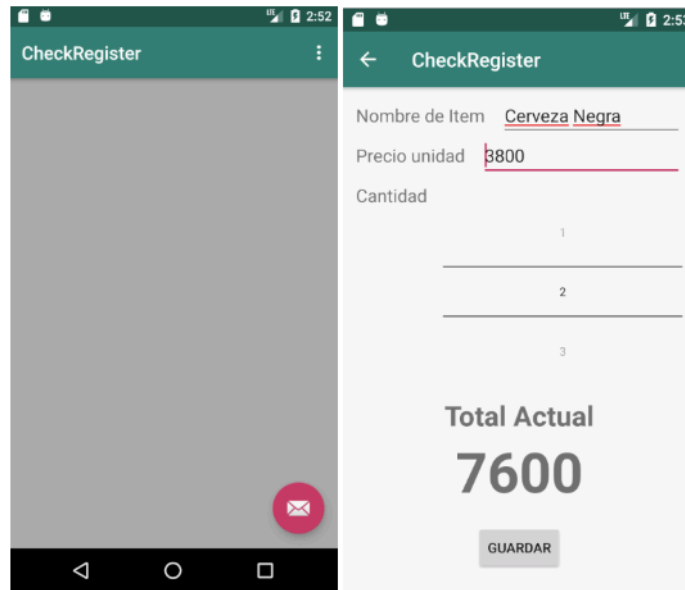


Imagen 14. Imágenes de referencia para la creación de la aplicación.

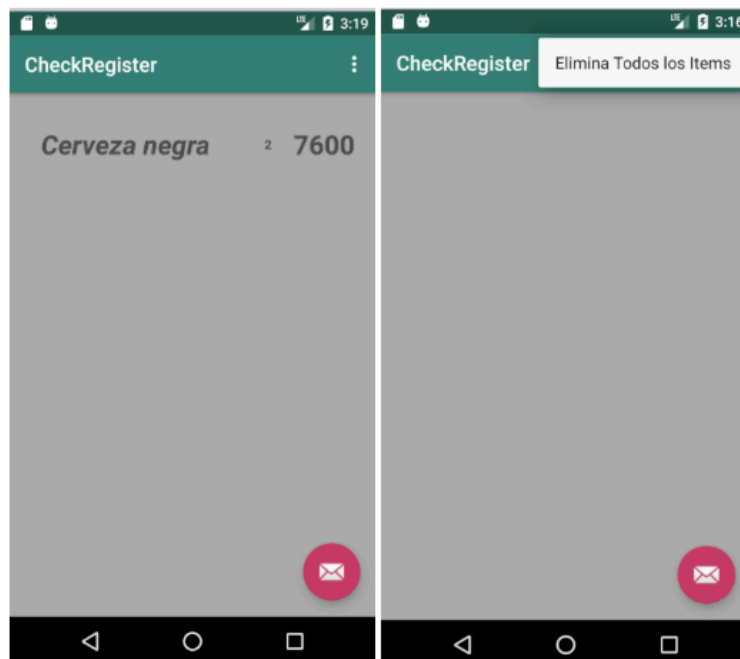


Imagen 16. Imágenes de referencia para la creación de la aplicación.

La aplicación utilizará LiveData para mostrar el listado de Items creados y también utilizará LiveData para guardar cambios en Room cambiar las vistas en base a estos.

## Diagrama de la aplicación.

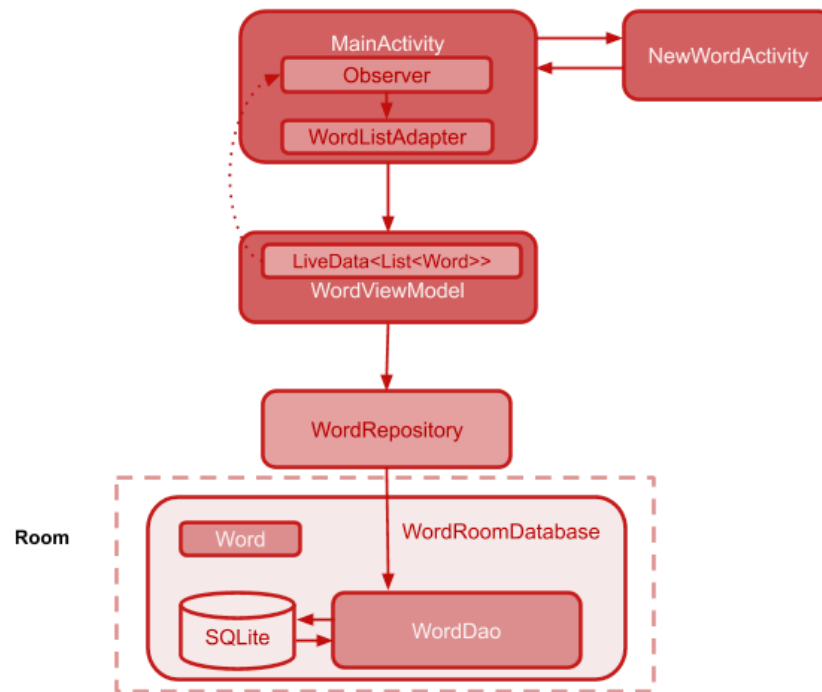


Imagen 17. Diagrama de la aplicación.

La imagen 17 obtenida de los CodeLabs oficiales de Google, resume un poco el funcionamiento de nuestra aplicación.

Como el alcance de esta unidad está centrado en el aprendizaje de LiveData, MVVM y pasar datos a la UI. Algunos componentes de la app no serán explicados con detalles porque se vieron en unidades anteriores, de igual forma todo el código estará a tu disposición.

# Creando el proyecto CheckRegister

## 1. Creando El proyecto

- Crea un proyecto en Android Studio.
- Escoge una Basic Activity.
- El lenguaje debe ser Kotlin.
- Como Api mínima usaremos 23.

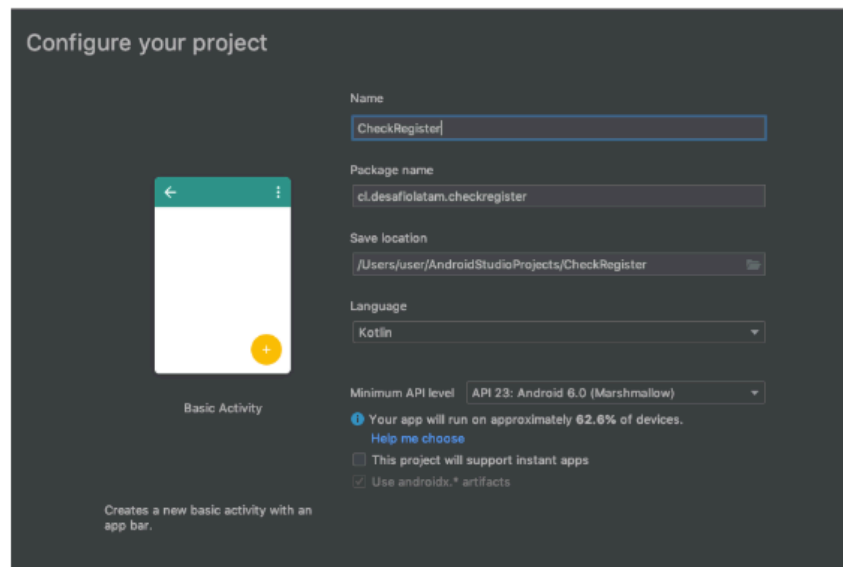


Imagen 18. Creando un nuevo proyecto.

## 2. Archivos Gradle y dependencias

- Primero vamos a añadir el siguiente plugin en nuestro archivo build.gradle (Module: app)

```
apply plugin: 'kotlin-kapt'
```

b. A continuación añadiremos al interior del bloque de Android lo siguiente, esto disminuirá los warning de la librería y también utilizaremos DataBinding.

```
android {  
  
    // other configuration  
    packagingOptions {  
        exclude 'META-INF/atomicfu.kotlin_module'  
    }  
  
    dataBinding {  
        enabled = true  
    }  
  
}
```

c. Ahora vamos añadir las dependencias.

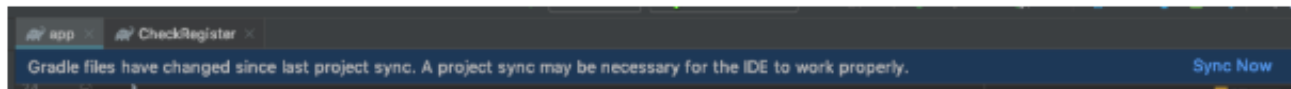
```
// Room components  
implementation "androidx.room:room-runtime:$rootProject.roomVersion"  
implementation "androidx.room:room-ktx:$rootProject.roomVersion"  
kapt "androidx.room:room-compiler:$rootProject.roomVersion"  
androidTestImplementation "androidx.room:room-testing:$rootProject.roomVersion"  
  
// Lifecycle components  
implementation "androidx.lifecycle:lifecycle-  
extensions:$rootProject.archLifecycleVersion"  
kapt "androidx.lifecycle:lifecycle-compiler:$rootProject.archLifecycleVersion"  
androidTestImplementation "androidx.arch.core:core-  
testing:$rootProject.androidxArchVersion"  
  
// ViewModel Kotlin support  
implementation "androidx.lifecycle:lifecycle-viewmodel-  
ktx:$rootProject.archLifecycleVersion"
```



Como puedes observar no estamos indicando un número específico en cada versión de los componentes, solo una referencia. por ejemplo “rootProject.roomVersion”, Lo dejamos de esta forma porque en nuestro archivo build.gradle (Project: CheckRegister) indicaremos las versiones de cada componente.

```
ext {  
    roomVersion = '2.2.0-rc01'  
    archLifecycleVersion = '2.2.0-alpha05'  
    androidxArchVersion = '2.0.0'  
    coroutines = '1.2.0'  
}
```

Estas son las versiones de las librerías actualizadas hasta la publicación de esta lectura. No Olvides realizar la sincronización de Gradle.



### 3. Creación de entidades y DAO

a. Ahora vamos a crear nuestra DataClass Check la cual será la entidad que represente los datos en la base de datos.

```
@Entity(tableName = "check_items")  
data class Check(  
    @PrimaryKey @NonNull val name: String,  
    val singlePrice: String = "",  
    val quantity: Int = 0,  
    val totalItem: String = "0"  
)
```

Como puedes observar hemos añadido los atributos o variables name para asignar un nombre y como clave primaria, singlePrice que será el precio unitario del ítem registrado, quantity para la cantidad del ítem y totalItem que almacenará el total del ítem que estemos guardando.

b. A continuación añadiremos el DAO a nuestro proyecto, Crea un nuevo Kotlin File de nombre CheckDao y añade este código.

```
@Dao
interface CheckDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertCheckItem(check: Check)

    @Insert
    suspend fun updateCheckItem(check: Check)

    @Delete
    fun clearAllCheckItems(vararg creature: Check)

    @Query("SELECT * FROM check_items ORDER BY name ASC")
    fun getAllCheckItems(): LiveData<List<Check>>
}
```

Como puedes observar este Dao es igual a los que has construido anteriormente, solo tiene funciones o métodos que nos permiten insertar un nuevo elemento, hacer update, borrar todos los elementos y obtener todos los elementos.

Debes haber notado que el método `getAllCheckItems()` incorpora `LiveData` antes del listado de `check`, añadir `LiveData` en este método hará que Room se encarga internamente de actualizar la información desde la base de datos al objeto `LiveData`, cuando la base de datos cambie.

Recordemos que, cuando los datos sean cambiados, necesitamos que se actualice nuestra UI, esto quiere decir que necesitamos observar estos datos, y cuando cambien podamos reaccionar.

Más adelante en este ejemplo veremos los cambios que se producen a través de un observador en nuestro `MainActivity`.

#### 4. Añadiendo Room al proyecto.

a. Debemos añadir la clase de configuración de Room. Crea un nuevo Kotlin File llamado CheckRoomDataBase y añade el siguiente código:

```
// Annotates class to be a Room Database with a table (entity) of the Check
class
@Database(entities = [Check::class], version = 1)
abstract class CheckRoomDataBase : RoomDatabase() {

    abstract fun checkDao(): CheckDao

    companion object {
        // Singleton prevents multiple instances of database opening at the
        // same time.
        @Volatile
        private var INSTANCE: CheckRoomDataBase? = null

        fun getDatabase(context: Context): CheckRoomDataBase {
            val tempInstance = INSTANCE
            if (tempInstance != null) {
                return tempInstance
            }
            synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    CheckRoomDataBase::class.java,
                    "check_database"
                ).build()
                INSTANCE = instance
                return instance
            }
        }
    }
}
```

No entraremos en detalle en este código en particular, es solo la implementación de Room como una clase abstracta heredando desde RoomDatabase y los métodos correspondientes para inicializar un Singleton o instancia única e la base de datos.

También crearemos un archivo de Kotlin que será el punto de partida de nuestra aplicación, en el se inicializará la BBDD.

```
class CheckApplication : Application() {

    companion object {
        lateinit var database: CheckRoomDataBase
    }

    override fun onCreate() {
        super.onCreate()
        database = Room.databaseBuilder(this, CheckRoomDataBase::class.java,
            "check_database")
            .build()
    }
}
```

Se llamará CheckApplication, también debemos indicarlo en nuestro Manifest.

```
<application
    android:name=".app.CheckApplication"
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity
        android:name=".ui.MainActivity"
        android:label="@string/app_name"
        android:theme="@style/AppTheme.NoActionBar">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
```

## 5. Creando el Repositorio

El repositorio será una clase abstracta que tendrá acceso a diferentes fuentes de datos. Esta clase repositorio no forma parte de los componentes de arquitectura, es solo sugerencia de buena práctica para mejorar la separación de código y la arquitectura de la aplicación.

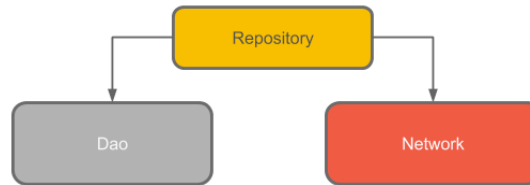


Imagen 20. Diagrama de implementación simple de un repositorio.

La imagen 2 representa una implementación simple de un repositorio, en nuestro caso solo nos conectaremos al DAO, ya que nuestro ejemplo no tiene acceso a la red o a un servidor para obtener datos, si ese fuera el caso, se utilizará el mismo repositorio.

El uso de un repositorio permite manejar las consultas a múltiples Backends o servicios, El ejemplo más recurrente es que el repositorio maneja la lógica de actualizar los datos desde un servidor o usar los datos en la base de datos local.

Integremos una clase pública CheckRepository y añada el siguiente código:

```
// Declares the DAO as a private property in the constructor. Pass in the DAO
// instead of the whole database, because you only need access to the DAO
class CheckRepository {

    private val checkDao: CheckDao = CheckApplication.database.checkDao()
    // Room executes all queries on a separate thread.
    // Observed LiveData will notify the observer when the data has changed.
    val allCheckItems: LiveData<List<Check>> = checkDao.getAllCheckItems()

    // this function are using AsyncTask for not block the UI
    fun insertCheckItem(checkItem: Check) {
        InsertAsyncTask(checkDao).execute(checkItem)
    }

    fun deleteAllCheckItems() {
        val creatureArray = allCheckItems.value?.toTypedArray()
        if (creatureArray != null) {
            DeleteAsyncTask(checkDao).execute(*creatureArray)
        }
    }
}
```

```

        private class InsertAsyncTask internal constructor(private val dao:
CheckDao) :
    AsyncTask<Check, Void, Void>() {
        override fun doInBackground(vararg params: Check): Void? {
            dao.insertCheckItem(params[0])
            return null
        }
    }

        private class DeleteAsyncTask internal constructor(private val dao:
CheckDao) :
    AsyncTask<Check, Void, Void>() {
        override fun doInBackground(vararg params: Check): Void? {
            dao.clearAllCheckItems(*params)
            return null
        }
    }
}

```

A grandes rasgos estamos pasando CheckDao a nuestra clase repositorio. De esta forma solo accederemos a los métodos que escriben o leen la base de datos, sin tener que exponer la base de datos completa al repositorio.

El listado de elementos viene inicializado al obtener el objeto LiveData que proviene desde Room, más específicamente del método getAllCheckItems. Room ejecutará esta petición en un hilo diferente al de UI y nos traerá los datos que han cambiado al main Thread.

Las operaciones de Insertar un nuevo elemento y Borrar se realizarán a través de AsyncTask, de esa forma no bloquearemos la interfaz de usuario cuando se ejecuten.

## 6. Creando el ViewModel

A continuación vamos a crear nuestro primer ViewModel, este se encargará de pasar la data a nuestra vista y sobrevivir a cualquier cambio de configuración. En otras palabras el ViewModel funcionara como un centro de comunicación o puente entre el repositorio y la UI, además incluso podemos utilizarlo para compartir datos entre fragmentos.

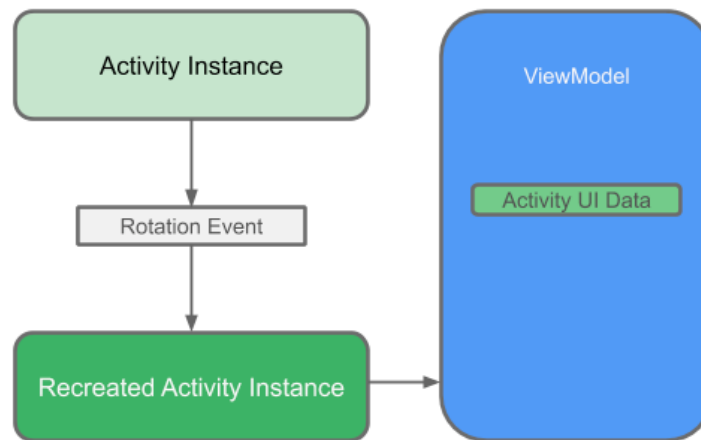


Imagen 21. Diagrama relacionando el ViewModel.

Nuestro Viewmodel contendrá los datos de la aplicación que se mostrarán en la UI, separando estos datos de nuestras actividades o fragmentos nos permitirá seguir el principio de “single responsibility” es decir de una sola responsabilidad. Ya que las vistas solo se encargan de dibujar estos datos en la pantalla mientras el ViewModel se tomará el control de procesar estos datos.

### a. Implementando el ViewModel

Crea un Kotlin file llamado CheckViewModel y añade el siguiente código:

```
// Class extends AndroidViewModel and requires application as a parameter.
class CheckViewModel(application: Application) : AndroidViewModel(application)
{

    // The ViewModel maintains a reference to the repository to get data.
    private val repository: CheckRepository

    // LiveData gives us updated checkItem when they change.
    val AllCheckItems: LiveData<List<Check>>

    // Gets reference to CheckDao from CheckRoomDatabase to construct
    // the correct CheckRepository.
    init {
        val checkDao = CheckRoomDataBase.getDatabase(application).checkDao()
        repository = CheckRepository()
        AllCheckItems = repository.allCheckItems
    }
}
```

```

fun insertCheckItem(checkItem: Check) {
    repository.insertCheckItem(checkItem)
}

fun deleteAllCheckItem() {
    repository.deleteAllCheckItems()
}
}

```

En este código creamos la clase llamada CheckViewModel la cual tiene como parámetro Application y hereda AndroidViewModel.

Tenemos una variable privada que referencia nuestro repositorio, también añadimos una variable de LiveData para mantener un caché de el listado de Items de la cuenta.

También creamos un bloque init el cual hace referencia a nuestro CheckDao desde CheckRoomDatabase. En este mismo bloque inicializamos AllCheckItems LiveData usando el repositorio.

Luego tenemos las funciones que realizarán las tareas de insertar un nuevo item o borrarlos todos. Si necesitaramos realizar otra tarea que necesito acceso a la BBDD, este es el archivo.

## 7. Lógica de negocio

Ahora vamos a crear la clase que se encargará de manejar los cálculos que se realizarán a los ítem que vamos a ir añadiendo. El ejemplo es bastante básico pero nos permitirá tener una visión de cómo realizar gestiones más completas siguiente una estructura.

```

class CheckGeneratorTotal {

    fun generateCheckTotal(
        name: String = "",
        singlePrice: String = "",
        quantity: Int = 0) : Check {
        val total = quantity * singlePrice.toInt()
        return Check(name, singlePrice, quantity, total.toString())
    }
}

```

La lógica es ir multiplicando por la cantidad el valor unitario para obtener el total del ítem añadido.



## 8. Organización de package del proyecto

a. Organizaremos un poco el proyecto

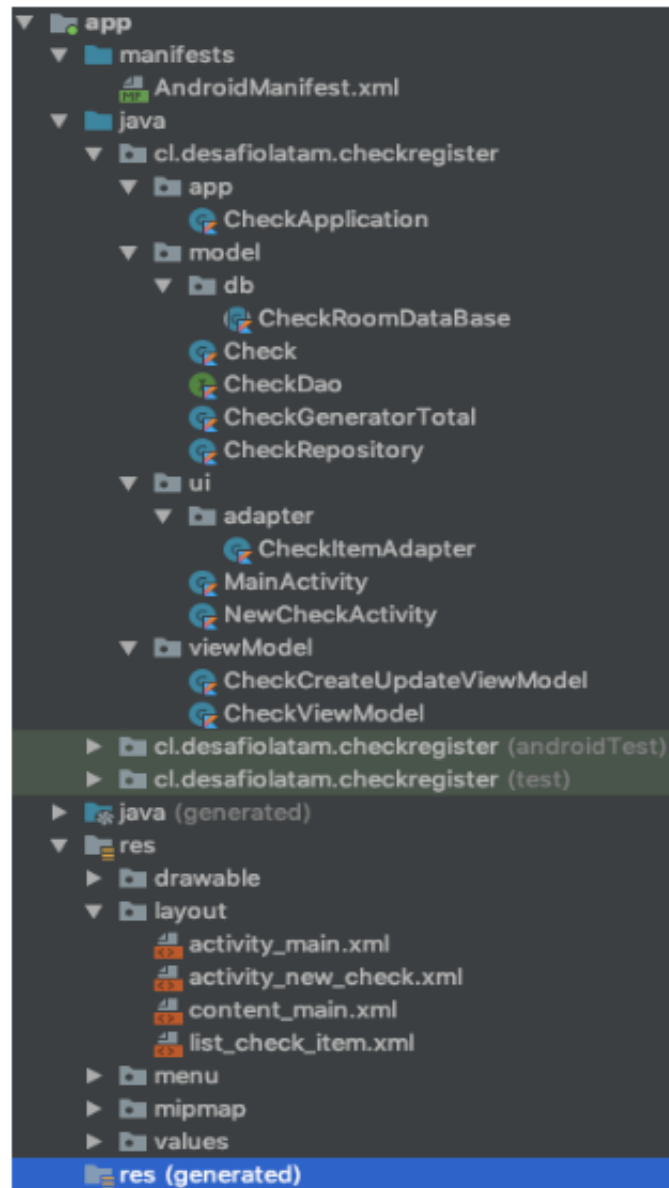


Imagen 22. Estructura del proyecto.

Creamos cuatro package principales al interior de nuestro proyecto, “app”, “model”, “ui” y “viewModel”. Refactorizamos moviendo las clases que hemos creado a sus respectivas carpetas.

En la imagen hay archivos que aún no has creado, tranquilo en los siguientes pasos se crearán en su totalidad.

## 9. Creando la Interfaz Gráfica.

A continuación crearemos los elementos que ya conocemos, Parte de la UI , layouts de ítems para el recyclerView y también el adapter que mostrará todos los items.list\_check\_item.xml

```
?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"
    android:layout_width="match_parent"
    android:layout_height="100dp"
    android:orientation="vertical">

    <TextView
        android:id="@+id/title_item"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="32dp"
        android:text="title"
        android:textSize="26sp"
        android:textStyle="italic|bold"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/quantity_item"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="48dp"
        android:text="Cantidad"
        android:textSize="12sp"
        android:textStyle="bold"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toStartOf="@+id/total_item"
        app:layout_constraintStart_toEndOf="@+id/title_item"
        app:layout_constraintTop_toTopOf="parent" />
```

```

<TextView
    android:id="@+id/total_item"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="16dp"
    android:layout_marginEnd="16dp"
    android:text="Total"
    android:textSize="26sp"
    android:textStyle="bold"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/quantity_item"
    app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>

```

Tambien vamos a añadir el recyclerView en activity\_main.xml

```

<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:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:context=".ui.MainActivity"
    tools:showIn="@layout/activity_main">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/check_item_recyclerview"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@android:color/darker_gray"
        tools:listitem="@layout/list_check_item" />

</androidx.constraintlayout.widget.ConstraintLayout>

```

Este será nuestro Adaptador para mostrar los elementos. CheckItemAdapter.kt

```
class CheckItemAdapter(  
    private val checkItems: MutableList<Check>  
) : RecyclerView.Adapter<CheckItemAdapter.CheckItemViewHolder>() {  
  
    private var checkItem = emptyList<Check>() // Cached copy of checkItem  
    inner class CheckItemViewHolder(itemView: View) :  
RecyclerView.ViewHolder(itemView) {  
        val checkItemView: TextView = itemView.findViewById(R.id.title_item)  
        val checkQuantityItem: TextView =  
itemView.findViewById(R.id.quantity_item)  
        val checkTotalItem: TextView = itemView.findViewById(R.id.total_item)  
    }  
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):  
CheckItemViewHolder {  
        val view =  
LayoutInflater.from(parent.context).inflate(R.layout.list_check_item, parent,  
false)  
        return CheckItemViewHolder(view)  
    }  
    override fun onBindViewHolder(holder: CheckItemViewHolder, position: Int) {  
        val current = checkItems[position]  
        holder.checkItemView.text = current.name  
        holder.checkQuantityItem.text = current.quantity.toString()  
        holder.checkTotalItem.text = current.totalItem  
    }  
    internal fun updateCheckItems(checkItems: List<Check>) {  
        this.checkItems.clear()  
        this.checkItems.addAll(checkItems)  
        notifyDataSetChanged()  
    }  
    override fun getItemCount() = checkItems.size  
}
```

## 10. Añadiendo los observadores en Main Activity

Ahora vamos a terminar la primera parte de esta aplicación añadiendo los observadores de LiveData en la MainActivity.

```
private lateinit var viewModel: CheckViewModel
private val adapter = CheckItemAdapter(mutableListOf())

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    setSupportActionBar(toolbar)
    viewModel = ViewModelProviders.of(this).get(CheckViewModel::class.java)

    val recyclerView = findViewById<RecyclerView>
(R.id.check_item_recyclerview)
    recyclerView.adapter = adapter
    recyclerView.layoutManager = LinearLayoutManager(this)

    fab.setOnClickListener { view ->
        startActivity(Intent(this, NewCheckActivity::class.java))
    }

    viewModel.AllCheckItems.observe(this, Observer { checkItem ->
        checkItem?.let {
            adapter.updateCheckItems(it)
        }
    })
}

override fun onCreateOptionsMenu(menu: Menu): Boolean {
    menuInflater.inflate(R.menu.menu_main, menu)
    return true
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {

    return when (item.itemId) {
        R.id.action_settings -> {
            viewModel.deleteAllCheckItem()
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

Como puedes observar, estamos observando los datos que están presentes en el ViewModel, que a su vez son obtenidos desde el repositorio.

Si ejecutamos nuestra aplicación aún no podremos ver los cambios, ya que nuestra base de datos está vacía, En el próximo capítulo, vamos añadir una nueva actividad y también a través de una combinación entre LiveData, ViewModel y databinding, veremos cambios en tiempo real en nuestra Interfaz de usuario, además cuando hayamos creado nuestros items, de forma automática se actualiza el recyclerView con los datos más nuevos.