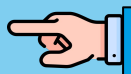
Arquitectura en Android

Componentes de arquitectura (Parte II)

{desafío} latam_

Utilizar patrones de arquitectura escalables para la construcción de una aplicación Android de acuerdo a los requerimientos

- Unidad 1: Kotlin para el desarrollo de aplicaciones.
- Unidad 2: Ciclo de vida de componentes.
- Unidad 3: Arquitectura en Android.
- Unidad 4: Programación asíncrona en Android.



Te encuentras aquí

{desafío} latam_

¿Qué aprenderás en esta sesión?

- *Implementar un aplicativo utilizando el patrón MVVM de acuerdo a las buenas prácticas recomendadas*

Como usuari@ ¿Has notado alguna vez un software con programación reactiva? ¿Podrías dar un ejemplo?

/* Patrón Observador */

Observer Pattern o Patrón Observador

- Observer Pattern es un patrón de diseño de comportamiento. Especifica la comunicación entre objetos: **observables** y **observadores**.

- *Un observable es un objeto que notifica a los observadores sobre los cambios en su estado.*

- Por ejemplo, una agencia de noticias puede notificar a los canales cuando reciben noticias. Recibir noticias es lo que cambia el estado de la agencia de noticias, y hace que los canales sean notificados.

Ejemplo de Observer Pattern

```
open class Subject {
    private var observers = mutableListOf<Observer>()

    fun callObservers() {
        for(obs in observers)
            obs.update()
    }

    fun attach(obs : Observer) {
        observers.add(obs)
    }

    fun detach(obs : Observer) {
        observers.remove(obs)
    }
}
```

```
interface Observer {
   fun update()
}

class Sensor : Subject() {
    var temperature: Int = 0
        set(value) {
        field = value
        callObservers()
    }
}
```

Ejemplo de Observer Pattern

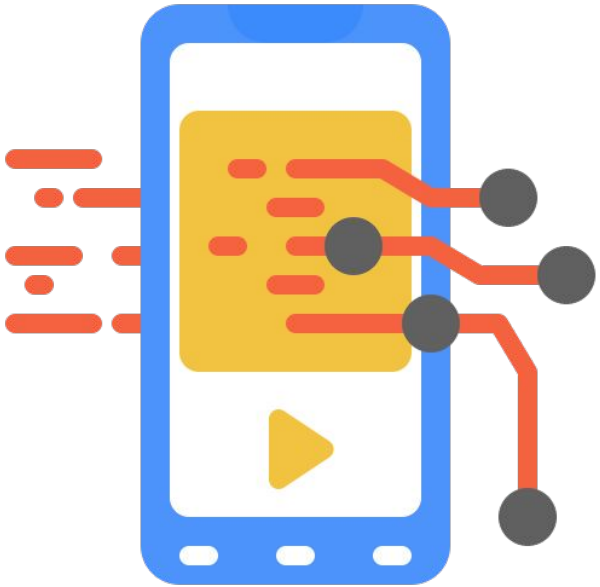
```
class Monitor(val sensor: Sensor) : Observer {
    init {
        sensor.attach(this)
    }

    override fun update() {
        val newTemperature = sensor.temperature
        println("update Monitor")
    }

}

fun main() {

    val sensor = Sensor()
    val monitor = Monitor(sensor)

    sensor.temperature = 5
}
```

/* Live Data */

LiveData

LiveData es una clase de titular de datos observable.

A diferencia de un observable regular, LiveData es consciente del ciclo de vida, lo que significa que respeta el ciclo de vida de otros componentes de la aplicación, como actividades, fragmentos o servicios. Esta conciencia garantiza que LiveData solo actualice los observadores de los componentes de la aplicación que se encuentran en un estado de ciclo de vida activo.



Las ventajas de usar LiveData

Garantiza que su interfaz de usuario coincida con su estado de datos

LiveData sigue el patrón del observador y notifica a los objetos Observer cuando cambian los datos subyacentes. Puede consolidar su código para actualizar la interfaz de usuario en estos objetos Observer. De esa forma, no necesita actualizar la interfaz de usuario cada vez que cambien los datos de la aplicación porque el observador lo hace por usted.

Sin fugas de memoria

Los observadores están vinculados a los objetos del ciclo de vida y se limpian cuando se destruye su ciclo de vida asociado.

Sin bloqueos debido a actividades detenidas

Si el ciclo de vida del observador está inactivo, como en el caso de una actividad en la pila de actividades, entonces no recibe ningún evento LiveData

No más manejo manual del ciclo de vida

Los componentes de la interfaz de usuario solo observan los datos relevantes y no detienen ni reanudan la observación. LiveData administra todo esto automáticamente, ya que está al tanto de los cambios de estado relevantes del ciclo de vida mientras observa.

Las ventajas de usar LiveData

Datos siempre actualizados

Si un ciclo de vida se vuelve inactivo, recibe los datos más recientes al volver a estar activo.

Por ejemplo, una actividad que estaba en segundo plano recibe los datos más recientes justo después de volver al primer plano.

Cambios de configuración adecuados

Si se vuelve a crear una actividad o un fragmento debido a un cambio de configuración, como la rotación del dispositivo, recibe inmediatamente los últimos datos disponibles.

Compartir recursos

Puede extender un objeto LiveData usando el patrón singleton para envolver los servicios del sistema para que puedan compartirse en su aplicación. El objeto LiveData se conecta al servicio del sistema una vez, y luego cualquier observador que necesite el recurso puede ver el objeto LiveData.

Para obtener más información, consulte Extender LiveData

/* StateFlow y SharedFlow */

StateFlow y SharedFlow

StateFlow y SharedFlow son API de flujo que permiten que los flujos emitan actualizaciones de estado y valores a múltiples consumidores de manera óptima.

StateFlow es un flujo observable de titular de estado que emite las actualizaciones de estado actuales y nuevas a sus recopiladores. El valor del estado actual también se puede leer a través de su propiedad de valor. Para actualizar el estado y enviarlo al flujo, asigne un nuevo valor a la propiedad value de la clase MutableStateFlow.

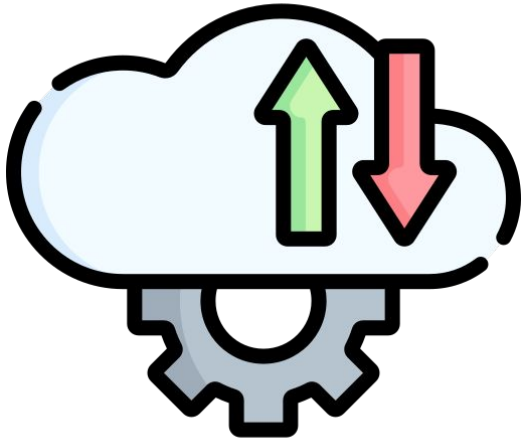
StateFlow y SharedFlow

En Android, StateFlow es ideal para las clases que necesitan mantener un estado mutable observable.

Siguiendo los ejemplos de los flujos de Kotlin, se puede exponer un StateFlow desde LatestNewsViewModel para que View pueda escuchar las actualizaciones de estado de la interfaz de usuario e inherentemente hacer que el estado de la pantalla sobreviva a los cambios de configuración.

SharedFlow

- La función **shareIn** devuelve un **SharedFlow**, un flujo activo que emite valores a todos los consumidores que recopilan de él.
- Un SharedFlow es una generalización altamente configurable de StateFlow.
- Puede crear un SharedFlow sin usar shareIn. Como ejemplo, podría emplear SharedFlow para enviar marcas al resto de la aplicación para que todo el contenido se actualice periódicamente al mismo tiempo.



/* ¿Entonces, StateFlow o LiveData? */

¿Entonces, StateFlow o LiveData?

StateFlow y LiveData tienen similitudes. Ambas son clases de contenedores de datos observables y ambas siguen un patrón similar cuando se usan en la arquitectura de su aplicación.

Ten en cuenta, sin embargo, que StateFlow y LiveData se comportan de manera diferente:

- StateFlow requiere que se pase un estado inicial al constructor, mientras que LiveData no.
- LiveData.observe() anula automáticamente el registro del consumidor cuando la vista pasa al estado DETENIDO, mientras que la recopilación de un StateFlow o cualquier otro flujo no deja de recopilar automáticamente. Para lograr el mismo comportamiento, debe recopilar el flujo de un bloque Lifecycle.repeatOnLifecycle.

Ejemplo LiveData:

El siguiente ejemplo es una función la cual, cada vez que es llamada, incrementa el valor de una variable en una unidad:

```
private val _counterMutableLiveData:
MutableLiveData<Int> = MutableLiveData()
val counterLiveData: LiveData<Int> =
_counterMutableLiveData

fun increaseCounter() {
    _counterMutableLiveData.value?.plus(1)
}
```

- Para usar la función desde un Activity o Fragment, basta con llamar la función a través del ViewModel

```
viewModel.increaseCounter()
```

- Para leer el valor de la variable, no se recomienda leer directamente un objeto mutable, es por eso que se creó la variable xxx

```
viewModel.counterLiveData.observe(viewLifecyc
leOwner) { counter->
        Log.d(TAG, "counter: $counter")
}
```

Ejemplo StateFlow:

Ahora veremos el mismo caso del ejemplo anterior pero usando StateFlow

```
private val _counterMutableStateFlow:
MutableStateFlow<Int> = MutableStateFlow(0)
val counterStateFlow: StateFlow<Int> =
_counterMutableStateFlow.asStateFlow()

fun increase() {
    _counterMutableStateFlow.value += 1
}
```

TIP: Al igual que en el caso del LiveData, no se recomienda leer un objeto mutable

- Una de las diferencias respecto a LiveData es que StateFlow requiere un valor por defecto, en este caso estamos pasando el valor 0, para usar la función desde un Activity o Fragment, es igual al caso del LiveData:

```
viewModel.increase()
```

- Pero para leer el valor de la variable **counterStateFlow** en la cual estamos guardando el contador, es un poco diferente

```
lifecycleScope.launchWhenCreated {
    viewModel.counterStateFlow.collect {
counter ->
        Log.d(TAG, "counter: $counter")
    }
}
```

Resumen

- LiveData y StateFlow son los primeros vistazos de lo que se conoce como Programación Reactiva.
- Con LiveData y StateFlow puedes hacer que una app parezca más dinámica.
- En Android, StateFlow es ideal para las clases que necesitan mantener un estado mutable observable.

Próxima sesión...

- ***Desafío evaluado***.

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales