Aplicación Android: Arquitectura

Consideraciones al diseñar una aplicación en Android

Como se vio en clases anteriores, una aplicación Android consta de varios componentes: **actividades**, **fragmentos**, **servicios**, **content providers** etc.

Dado que los dispositivos móviles tienen restricciones en cuanto a memoria, capacidad de proceso, etc. el sistema operativo podría cerrar unilateralmente el proceso de la aplicación para liberar recursos.

Esta contingencia está fuera de nuestro control y por ende **no debemos almacenar ni mantener en la memoria ningún estado, ni datos** de la aplicación.

Principios comunes de arquitectura en Android

- **Separación de problemas:** No se debería escribir el código de toda la aplicación en un **activity** o **fragment**, estos sólo deberían contener la lógica de las interacciones con el SO y la interfaz de usuario (**IU**).

- **Controlar la IU a partir de modelos de datos:** Los modelos de datos son independientes de las **views** y por lo tanto, no están vinculados al ciclo de vida de los **activities** o **fragments**. De este modo los usuarios no pierdan los datos si el SO destruye el proceso de la aplicación para liberar recursos.

Principios comunes de arquitectura en Android (cont.)

- **Única fuente de información:** Para cada tipo de dato en la aplicación, se debe asignar **una única fuente de información (Repositorio + Fuente de datos)**.

- **Flujo de datos unidireccional:** El flujo del estado se dirige siempre desde el repositorio a la **view**, mientras que los eventos transitan en el sentido opuesto. Por ejemplo, cuando los datos de la aplicación son mostrados en un activity, los datos provienen de repositorios y al registrarse un evento (presionar un botón), la información en la pantalla es almacenada en el o los repositorios correspondientes.

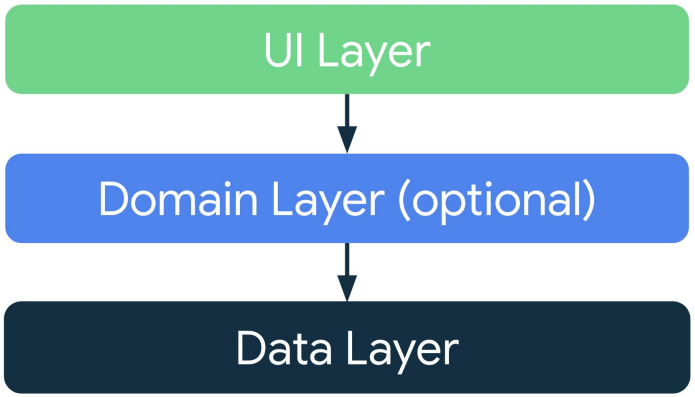
Arquitectura recomendada

¿Cómo diseñamos la arquitectura de nuestra aplicación?

Cada aplicación debe tener mínimamente dos capas:

- La **capa de IU** que muestra los datos en pantalla.
- La **capa de datos** que contiene la lógica de negocio y los datos.

Pudiéndose desagregar la lógica de negocios en una capa adicional (**capa de dominio**) a fin de potenciar la reutilización de las entidades allí definidas.

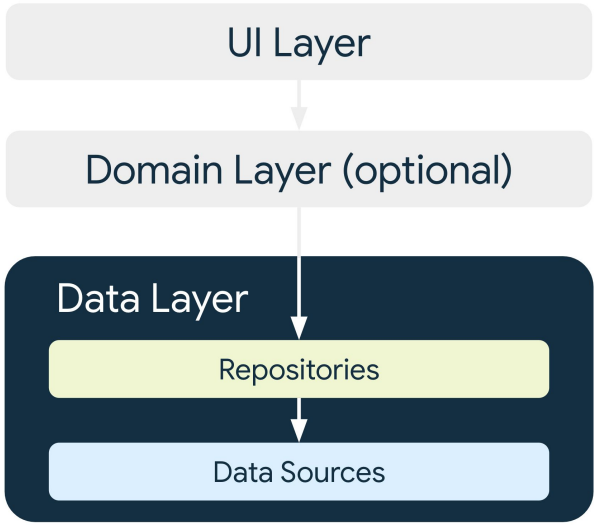


Capa de datos

La **capa de datos** está formada por **repositorios**, uno por cada tipo de dato que se administre en el aplicativo y estos a su vez pueden contener cero o varias **fuentes de datos**.

Los de repositorio son responsables de:

- Proveer la información a la **capa de IU**.
- Centralizar las modificaciones sobre los datos.
- Abstraer y resolver conflictos entre múltiples fuentes de datos.
- Contener a la lógica de negocio (si no se define capa de domino).



Capa de datos: Fuente de datos

- Las **fuentes de datos** (**Data Sources**) ofician de puente entre la aplicación y el o los sistemas que administran los datos.

- Las demás capas nunca deben acceder directamente a las **fuentes de datos** y los únicos puntos de entrada a la capa de datos son siempre los **repositorios** beneficiando de este modo a la escalabilidad del aplicativo.

- Cada clase que implementa una fuente de datos debe modelar **una única fuente de datos**, pudiendo ser un **archivo**, un **servicio de red** o una **base de datos local**. Asegurando en todo momento que contiene datos que son coherentes, correctos y actualizados.

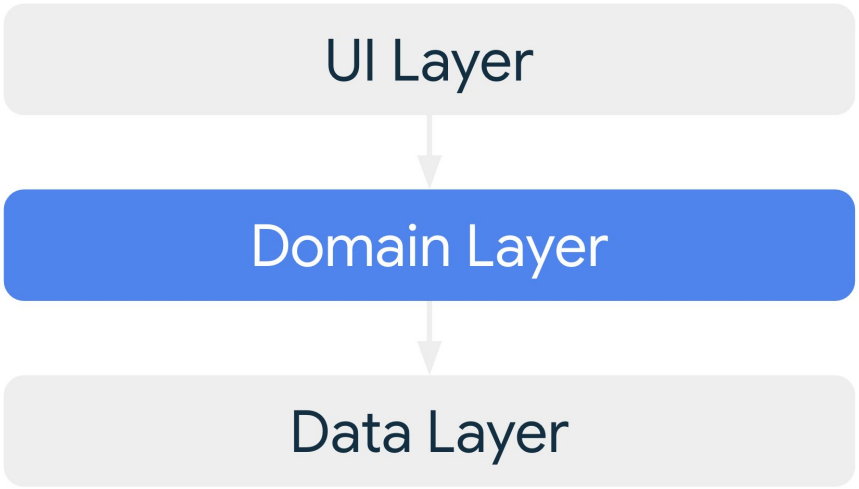
- **Los datos que se exponen deben ser inmutables** para que otras clases no puedan manipularlos pudiendo conllevar a estado inconsistente de los mismos.

Capa de dominio

- La **capa de dominio** es una capa opcional que se encuentra entre la **capa de IU** y la **de datos**.

- La **capa de dominio** es responsable de encapsular la lógica de negocio que varias **views** reutilizan.

- La **capa de dominio** brinda los siguientes beneficios:
  - Evita la duplicación de código.
  - Mejora la legibilidad y la capacidad de prueba.
  - Evita las clases grandes, ya que permite dividir las responsabilidades.

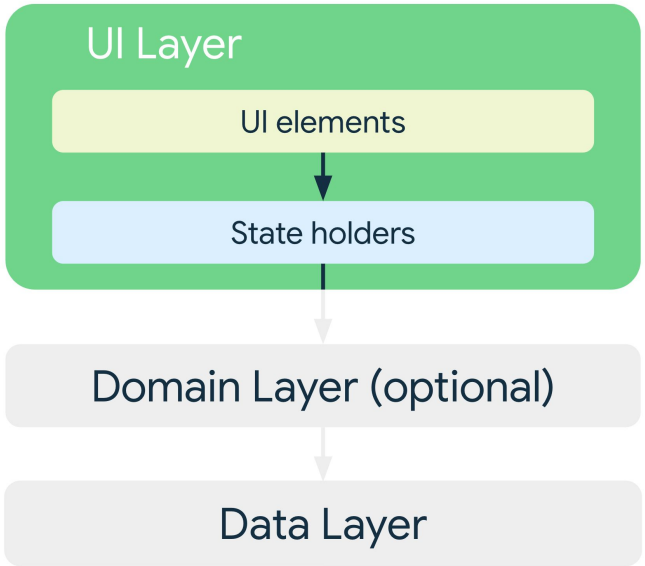


Capa de IU

El término **IU** hace referencia a elementos como **activities** o **fragments** que son los encargados de mostrar los datos.

La **capa de IU** tiene la responsabilidad de:

- Tomar información de la **capa de datos**, formatearla y presentársela al usuario.
- Registrar los eventos de usuario y a partir de estos reflejar sus efectos en los datos según sea necesario.



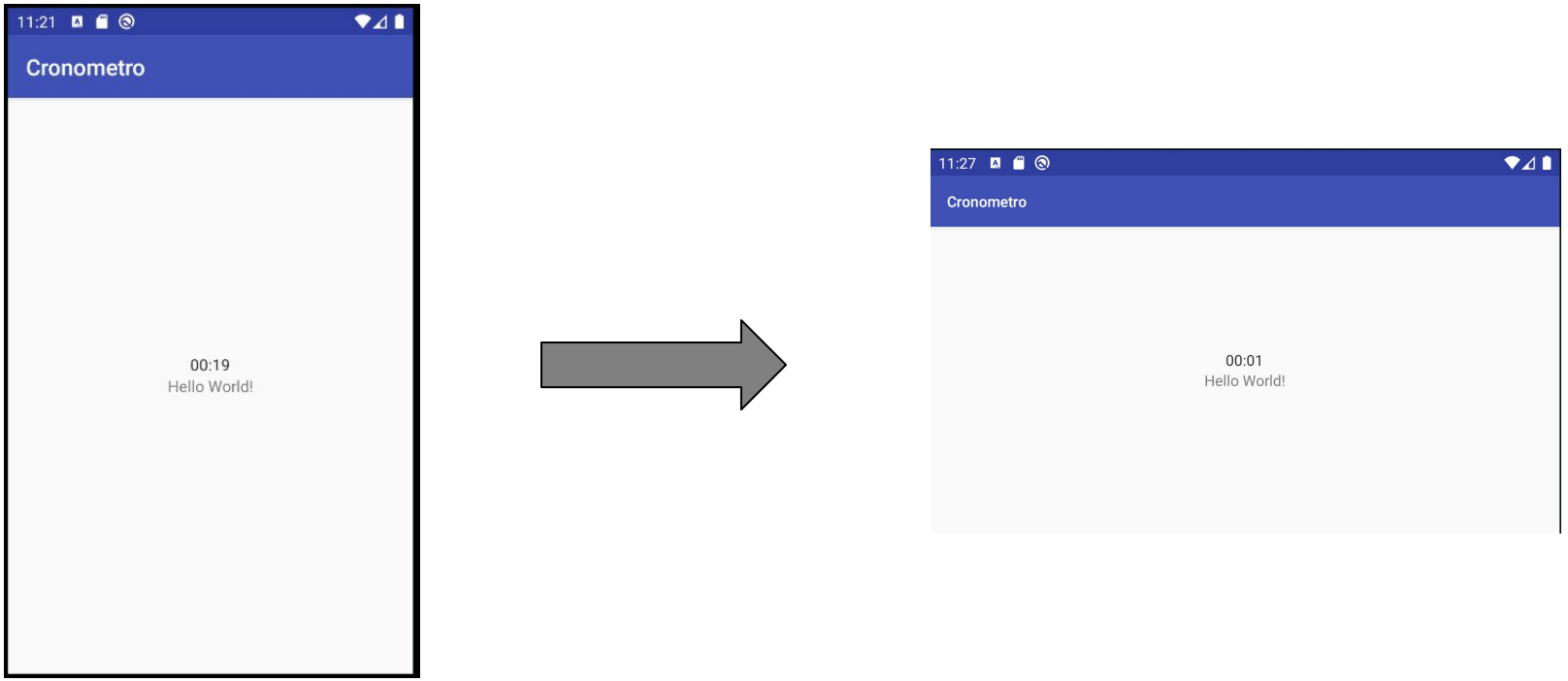
Capa de IU: Contenedores de estado

- Si la **IU** es lo que ve el usuario, el estado de la **IU** es lo que la aplicación dice que debería ver.
- La **IU** es la representación visual del estado y cualquier cambio en este último se refleja inmediatamente en el componente **IU**.

Las clases que son responsables de la producción del estado de la IU se denominan **contenedores de estado**. La implementación típica es **ViewModel**, aunque según sean los requisitos de la aplicación una simple clase podría ser suficiente.

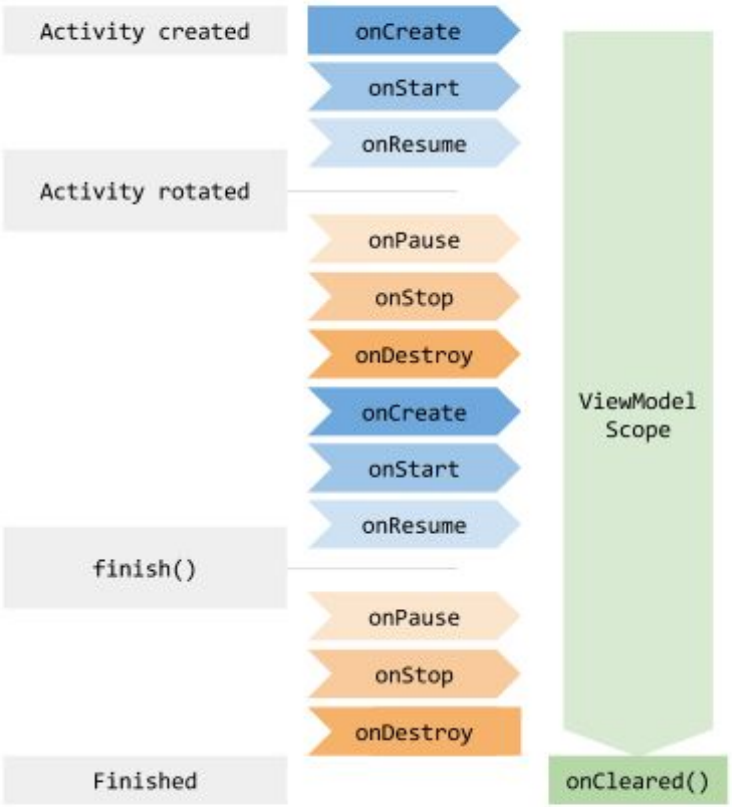
Aplicacion Cronometro

- La aplicación muestra los segundos transcurridos desde que comenzó a ejecutar.
- Si el usuario rota el dispositivo cambiando la configuración de la aplicación, ¿Qué sucede con el registro del tiempo transcurrido hasta el momento?



La clase ViewModel

- La clase **ViewModel** permite que se **conserven los datos de la vista** luego de cambios de configuración, como por ejemplo las rotaciones de pantallas.
- Cuando una **actividad es recreada** Android reconecta automáticamente el **ViewModel** al activity.

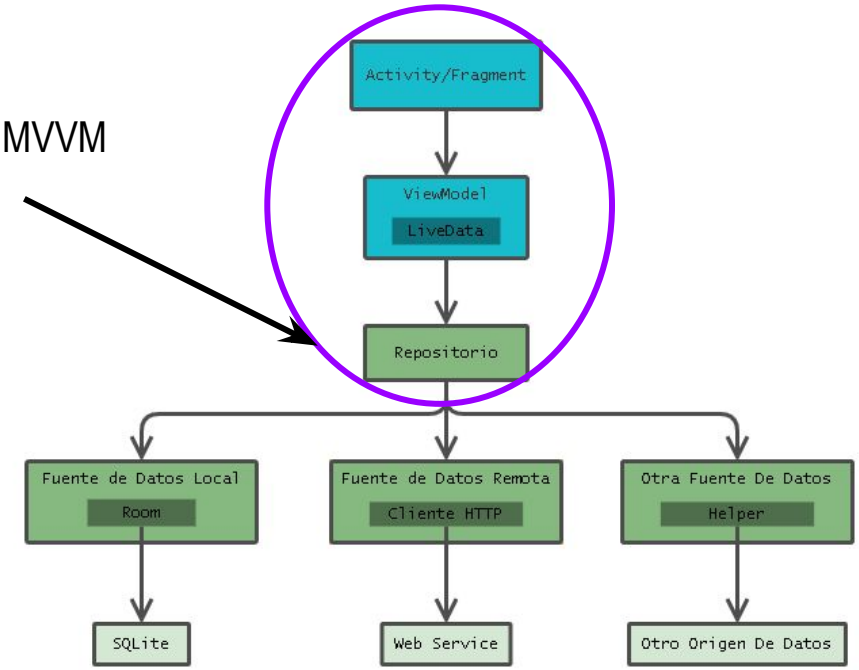


- Estados del ciclo de vida de la actividad a medida que atraviesa una rotación.
- El ciclo de vida del **ViewModel** junto al de la actividad asociada.
- Los mismos estados básicos se aplican al ciclo de vida de un fragmento.

La clase ViewModel

- Provee **separación de responsabilidades** entre la **visualización** y la **lógica** de los activities y fragments (por ejemplo la carga de datos de alguna fuente).
- La clase **ViewModel** se utiliza para **almacenar** y **administrar datos de la IU** de manera optimizada durante el ciclo de vida de los activities.
- En síntesis, la componente **ViewModel** de Android cumple **dos funciones**:
  - **Prepara y administra** los datos de la **vista**.
  - **Maneja la comunicación de la vista** con el resto de la aplicación (datos).

Implementación del patrón MVVM
(Model-View - ViewModel)



La clase ViewModel: ¿cómo usarla? - LiveData

- Para usar el componente **ViewModel** para el ejemplo anterior es necesario definir:

```
class MyViewModel : ViewModel() {
  val ONE_SECOND:Long = 1000
  val mElapsedTime = MutableLiveData<Long>()
  val mInitialTime: Long = SystemClock.elapsedRealtime()
  val timer: Timer = Timer()
  init {
    this.timer.scheduleAtFixedRate(
      object : TimerTask() {
        override fun run() {
          val newValue = (SystemClock.elapsedRealtime() - mInitialTime) / ONE_SECOND
          mElapsedTime.postValue(newValue)
        }
      }, ONE_SECOND,ONE_SECOND )
    }
    fun getElapsedTime(): LiveData<Long> {
        return mElapsedTime
    }
    . . .
```

LiveData es una clase de contenedor de datos observables.

Actualiza cada segundo.

```
override fun onCleared() {
    super.onCleared()
    timer.cancel()
}
```

La clase ViewModel: ¿cómo usarla? - StateFlow

- Para usar el componente **ViewModel** con **StateFlow** para el ejemplo anterior es necesario definir:

StateFlow es un flujo observable de datos.

Actualiza cada segundo.

```
class MyViewModel: ViewModel() {

    val ONE_SECOND:Long = 1000

    var time: Long = 0

    val mElapsedTime = MutableStateFlow<Long>(time)

    var fin = false

    init {

        viewModelScope.launch {

            while (!fin) {

                delay(ONE_SECOND)

                mElapsedTime.emit(++time)

            }

        }

    }

    fun getElapsedTime(): StateFlow<Long> {

        return mElapsedTime

    }
```

```
override fun onCleared() {

    super.onCleared()

    fin = true

}
```

La clase ViewModel: ¿cómo usarla? - LiveData

Se crea el **ViewModel** la primera vez que el sistema llama al método **onCreate()** de un activity.

Las **actividades recreadas** reciben la misma instancia del **ViewModel** creada por el primer activity.

En el ejemplo:

```
class MyActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
      val model: MyViewModel by viewModels()
      model.getElapsedTime().observe(this, Observer<Long>{ value ->
          val newText: String = getString(R.string.seconds, value)
          (findViewById<View>(R.id.timer_textview) as TextView).text = newText
          Log.d("CronometroActivity", "Actualizando timer")
        })
    }
}
```

Registro del observer.

La clase ViewModel: ¿cómo usarla? - StateFlow

En el ejemplo:

```
class MyActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
      val model: MyViewModel by viewModels()
      lifecycleScope.launch {
          repeatOnLifecycle(Lifecycle.State.STARTED) {
              myViewModel.mElapsedTime.collect{ value ->
                  val newText: String = getString(R.string.seconds, value)
                  (findViewById<View>(R.id.timer_textview) as TextView).text = newText
                  Log.d("CronometroActivity", "Actualizando timer")
              }
          }
      }
  }
}
```

Inicia la corrutina en scope del ciclo de vida del activity

Ejecuta en una nueva corrutina cada vez que el activity es STARTED en su ciclo de vida y es cancelada cuando es STOPPED.

LiveData vs StateFlow

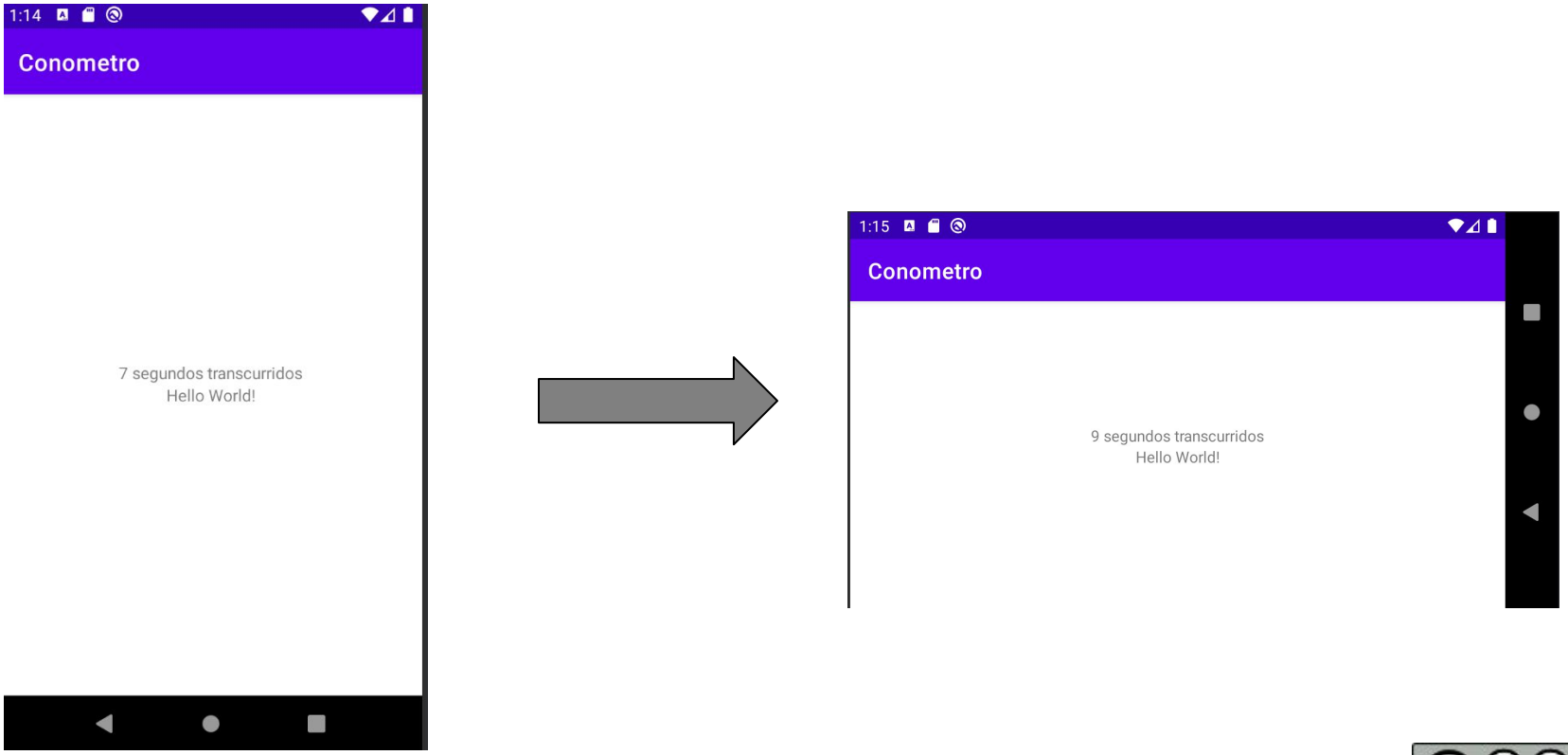
StateFlow y **LiveData** tienen similitudes. Ambas son clases contenedoras de datos observables y siguen un patrón similar cuando se usan en la arquitectura de la aplicación.

Sin embargo, **StateFlow** y **LiveData** se diferencian en:

- **StateFlow** requiere que se pase un estado inicial al constructor, mientras que **LiveData**, no.
- **`LiveData.observe()`** cancela automáticamente el registro del observador cuando se pasa al estado **STOPPED**, mientras que **StateFlow** no suspende el colectado de datos. Para obtener el mismo comportamiento, debes recolectar el flujo desde un bloque **`Lifecycle.repeatOnLifecycle`**.

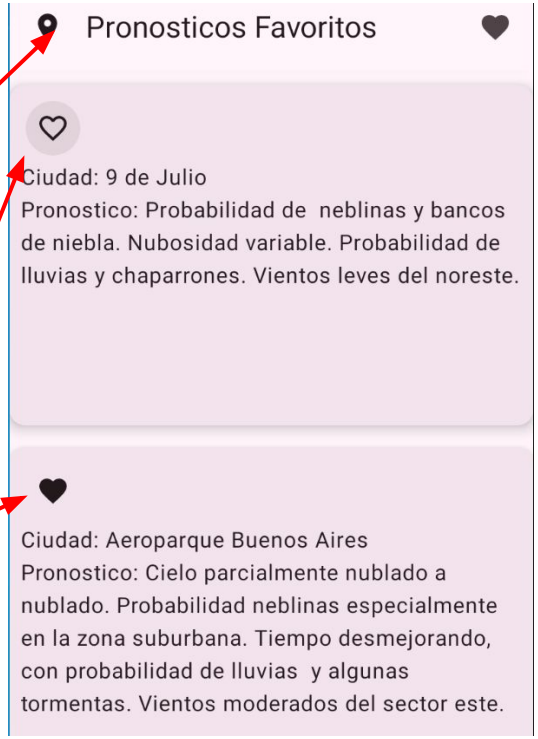
La clase ViewModel: ¿cómo usarla?

- Ahora si el usuario rota el dispositivo cambiando la configuración de la aplicación, ¿Qué sucede con el registro del tiempo transcurrido hasta el momento?



Aplicación Pronósticos Favoritos

- Se debe desarrollar una aplicación que recupere los pronósticos meteorológicos para todo el país desde el sitio del Servicio Meteorológico Nacional, permitiendo al usuario seleccionar y guardar solo aquellos pronósticos que le resulten de interés.



Obtiene y muestra todos los pronósticos meteorológico de todas las regiones del país.

Marca / desmarca como favorito al pronóstico indicado.

Filtra y muestra solo los pronósticos favoritos.

Aplicación Pronósticos Favoritos

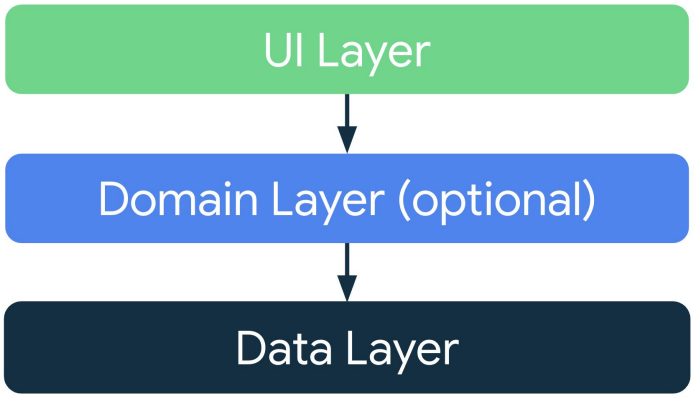
- ¿Cómo comenzamos a diseñar la aplicación?
- ¿Qué vamos a necesitar?
- ¿Dónde se ubica cada componente de software?

Aplicación Pronósticos Favoritos

- ¿Cómo comenzamos a diseñar la aplicación?

Como vimos anteriormente la aplicación debe estar diseñada en capas y en particular definiremos tres capas:

- La **capa de IU** que muestra los datos en pantalla.
- La **capa de datos** que contiene la lógica de negocio y los datos.
- La **capa de dominio** contendrá las entidades utilizadas en las dos capas anteriores.



Aplicación Pronósticos Favoritos

- ¿Qué vamos a necesitar?

1. Una biblioteca para trabajar con APIs REST que nos permita obtener los pronósticos del Servicio Meteorológico Nacional de manera eficiente.
2. Una biblioteca de persistencia de datos para aplicaciones Android, que nos permita almacenar los pronósticos favoritos.

Capa de dominio

- La **capa de dominio** contendrá las siguientes entidades que serán utilizadas por todo el aplicativo:

```
@Entity(tableName = "favorite")
data class Favorite (
    @PrimaryKey(autoGenerate = true) val id: Long = 0,
    var marker : Boolean,
    @Embedded val forecast: Forecast?
)

data class Forecast (val _id: String, val dist: String, val lid: String,
val name: String, val province:String, val lat: String,val lon: String,
val zoom: String,@Embedded val weather: Weather?)

data class Weather (val day: String, val morning_temp: String,
val morning_id: String, val morning_desc: String,val afternoon_temp:
String, val afternoon_id: String, val afternoon_desc: String)
```

Aplicación Pronósticos Favoritos - Capa de datos

La **capa de datos** estará formada por dos **repositorios**, uno por cada tipo de dato, los pronósticos (`Forecast`) y los favoritos (`Favorite`). Cada **repositorios** contendrán una de dos **fuentes de datos**, una que utiliza **Retrofit** y la otra **Room**.

Fuente de datos y repositorio de pronósticos:

Repositorio *Forecast*

```
interface ForecastRepository {
    @GET("map items/forecast/1")
    suspend fun getForecast(): List<Forecast>
}
```

Fuentes de Datos

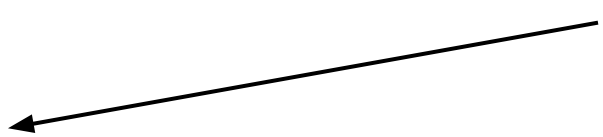
Retrofit

```
object ForecastRepositoryInstance {
    private const val BASE_URL = "https://ws.smn.gob.ar/"

    val api: ForecastRepository by lazy {
        Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create(ForecastRepository:: class.java)
    }
}
```

Aplicación Pronósticos Favoritos - Capa de datos

Fuente de datos de favoritos:




```
@Database(entities = [Favorite::class], version = 1, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {
    abstract fun favoriteDao(): FavoriteDao

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

        fun getDatabase(context: Context): AppDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                    "favorites_database"
                ).build()
                INSTANCE = instance
                instance
            }
        }
    }
}
```

Aplicación Pronósticos Favoritos - Capa de datos

Repositorio de favoritos:

```
@Dao
interface FavoriteDao {

    @Insert
    suspend fun insert(favorite: Favorite)

    @Update
    suspend fun update(favorite: Favorite)

    @Query("SELECT * FROM favorite ORDER BY name ASC")
    fun getAllFavorites(): Flow<List<Favorite>>
. . .
}

class FavoriteRepository(private val favoriteDao: FavoriteDao) {

    val allfavorites: Flow<List<Favorite>> =  favoriteDao.getAllFavorites()

    suspend fun insert(favorite: Favorite) {
        favoriteDao.insert(favorite)
    }

    suspend fun update(favorite: Favorite) {
        favoriteDao.update(favorite)
    }

    suspend fun getFavForecasts(): Flow<List<Favorite>> {
        return favoriteDao.getFavForecast()
    }
. . .
}
```

Repositorio
Favorite

Aplicación Pronósticos Favoritos - Capa UI

En la **capa de IU** se ubicará el **MainActivity** que desplegará al usuario los pronósticos y el **ViewModel** que se gestionará el estado de la vista.

ViewModel favoritos:

```
class FavoriteViewModel(private val repository: FavoriteRepository) : ViewModel() {
   private val _allfavorites = MutableStateFlow<List<Favorite>>(emptyList())
   val allfavorites: StateFlow<List<Favorite>> =_allfavorites

   init {
       viewModelScope.launch {
           repository.allfavorites.collect { response -> _allfavorites.value = response }
       }
   }
   fun fetchForecasts() = viewModelScope.launch {
        repository.deleteAll()
        val forecasts = ForecastRepositoryInstanceapi.getForecast()
        forecasts.forEach { forecast ->  repository.insert(Favorite(0, false, forecast)) }
   }
   fun onToggleFavorite(favorite:Favorite) =viewModelScope.launch {
       favorite.marker=!favorite.marker
       repository.update(favorite)
       _allfavorites.update { state -> emptyList() }
   }
   fun getFavBookmarks() = viewModelScope.launch {
           repository.getFavForecasts().collect{ response -> _allfavorites.value = response }
   }
}
class FavoriteViewModelFactory(private val repository: FavoriteRepository) : ViewModelProvider.Factory {
   override fun <T : ViewModel> create(modelClass: Class<T>): T {
       if (modelClass.isAssignableFrom(FavoriteViewModel::class.java)) {
           @Suppress("UNCHECKED_CAST")
           return FavoriteViewModel(repository) as T
       }
       throw IllegalArgumentException("Unknown ViewModel class")
   }
}
```

Flujo de datos observables.

Aplicación Pronósticos Favoritos - Capa UI

MainActivity:

Contenedor del estado de la vista

```
class MainActivity : ComponentActivity() {
    private val favoriteViewModel: FavoriteViewModel by viewModels {
        FavoriteViewModelFactory(( application as FavoriteApplication). repository)
    }
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            FavoriteForecastsTheme {
                topAppBar(favoriteViewModel)
            }
        }
    }
}

. . . .
```

Aplicación Pronósticos Favoritos - Capa UI

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun topAppBar(viewModel: FavoriteViewModel =  viewModel()) {
  val allfavorites = viewModel.allfavorites.collectAsState(initial = emptyList())
   Scaffold(
    topBar = {
      TopAppBar(
          title = { Text("Pronosticos Favoritos", maxLines = 1, overflow = TextOverflow.Ellipsis)
},
         navigationIcon = {
           IconButton(onClick = { viewModel.fetchForecasts() }) {
              Icon(imageVector = Icons.Filled.LocationOn, contentDescription = "ver pronosticos")
            }
         },
         actions = {
            IconButton(onClick = { viewModel.getFavBookmarks() }) {
               Icon(imageVector = Icons.Filled.Favorite, contentDescription = "ver favoritos" )
             }
         }
        )
       },
       content = { innerPadding ->
              LazyColumn(
                   contentPadding = innerPadding,
                   verticalArrangement = Arrangement.spacedBy(8.dp)
              ) {
                   items(allfavorites.value) { favorite ->
                          favElevatedCard(favorite = favorite, viewModel)
                   }
              }
          }
       }
```

Aplicación Pronósticos Favoritos - Capa UI

```
@Composable
fun favElevatedCard(favorite: Favorite, viewModel: FavoriteViewModel =  viewModel()) {
   ElevatedCard(
       elevation = CardDefaults.cardElevation(
           defaultElevation = 6.dp
       ),
       modifier = Modifier.size(width = 500.dp, height = 250.dp)
   ) {
       favItem(favorite = favorite, viewModel)
   }
   Spacer(modifier = Modifier.height(8.dp))
}


@Composable
fun favItem(favorite: Favorite, viewModel: FavoriteViewModel =  viewModel()) {
   Column(modifier = Modifier.padding(8.dp)) {
     IconButton(onClick = {viewModel.onToggleFavorite(favorite)}, enabled = true) {
                          Icon(imageVector    =    if(favorite.marker){Icons.Filled.Favorite}    else
{Icons.Sharp.FavoriteBorder},                              contentDescription = "Favorite")
       }
       Text(text = "Ciudad: ${favorite.forecast?.name}")
       Text(text = "Pronostico: ${favorite.forecast?.weather?.morning_desc}")
       Spacer(modifier = Modifier.height(8.dp))
   }
}
```