Programación multimedia y dispositivos móviles

10

Arquitectura para Aplicaciones Android. ViewModel. LiveData

IES Nervión
Miguel A. Casado Alías

Introducción

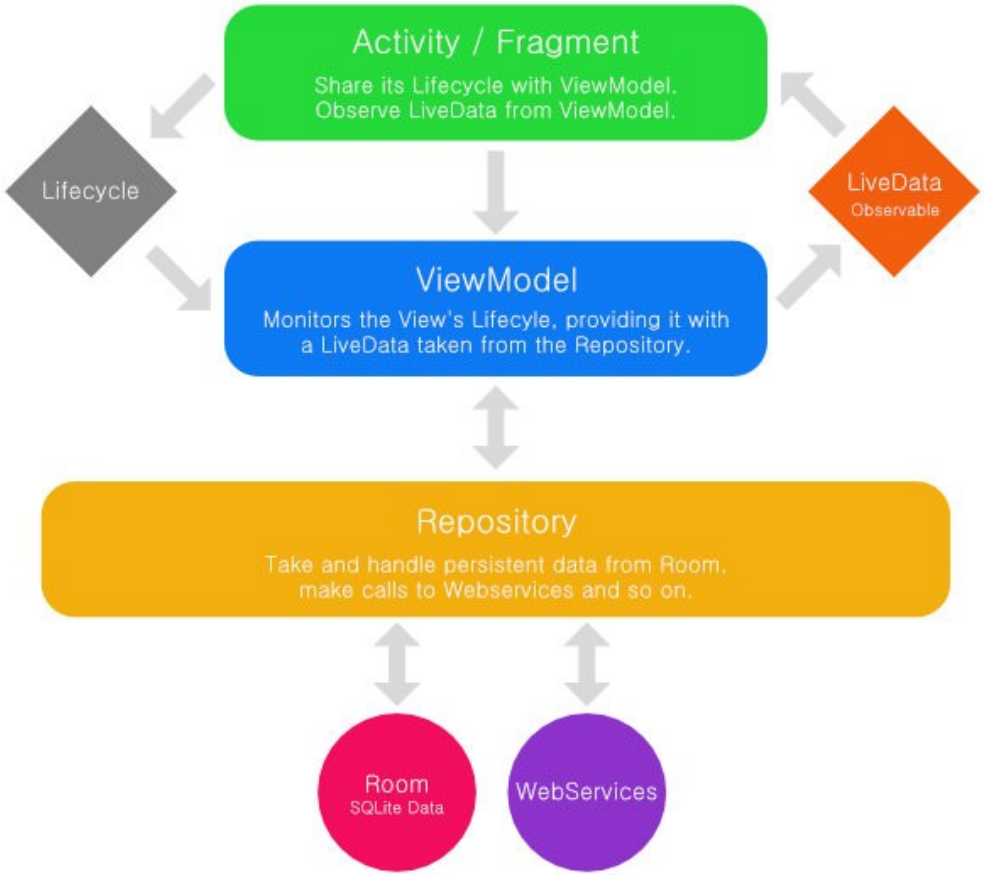
- Hasta el año 2017 no había una recomendación oficial o un estándar sobre una arquitectura para las aplicaciones Android que tuviera en cuenta las peculiaridades del sistema (p.ej. el ciclo de vida de actividades y fragmentos)
- En ese año Google (Android) por fin anuncia una arquitectura oficial y libera componentes para llevarla a cabo:
  - Lifecycle
  - ViewModel
  - LiveData
  - Room
  - Paging (en fase alfa aún; enero 2018)
- Hasta ahora cada programador debía decidir qué arquitectura usar en sus apps, o no usar ninguna arquitectura

Principios de la nueva arquitectura

- Separación de intereses
  - Es un error común escribir todo el código en una actividad o fragmento
  - Todo código que no suponga una **interacción con la interfaz de usuario o con el sistema operativo** no debería estar en las actividades ni en los fragmentos
  - Mantener esas clases lo más simple posible nos ahorrará quebraderos de cabeza relacionados con el ciclo de vida
- Construir la interfaz de usuario a partir de un modelo
  - El modelo es responsable de manejar los datos de la app
  - Es independiente de la vista y de los componentes de la app
  - Esa independencia lo aisla de los problemas derivados del ciclo de vida que tienen dichos componentes

Esquema de la nueva arquitectura

- Es similar a un Modelo Vista Controlador o Modelo Vista Presentador, pero está muy ligada a la arquitectura del sistema Android

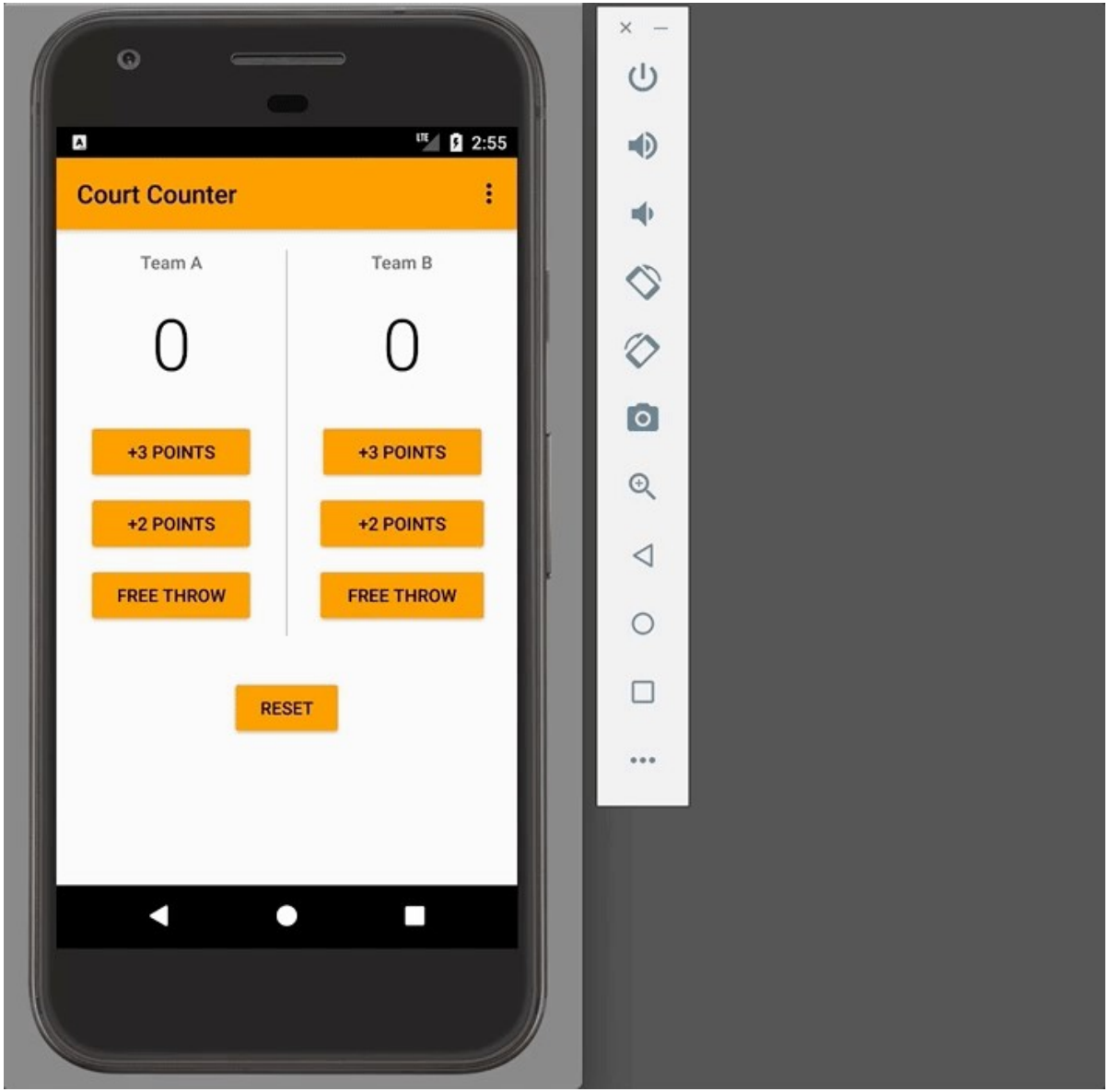


Tareas de los componentes principales

- Actividades / Fragmentos (Vista)
  - Configuran la UI, manejan la interacción con el usuario
  - Observan y muestran elementos LiveData provenientes del ViewModel
- ViewModel
  - Observa automáticamente el estado del ciclo de vida de la vista
  - Mantiene consistencia durante cambios de configuración y otros eventos del ciclo de vida
  - A demanda de la vista obtiene datos provenientes del repositorio en forma de observables LiveData
  - No referenciar la vista directamente (salvo AndroidViewModel)
- Repositorio
  - Obtiene datos de todas las fuentes y los ofrece al ViewModel

Problemas de los cambios de configuración

- No es fácil conservar la información si la guardamos en las actividades
- Si se está haciendo alguna tarea en segunda plano y en ese instante se produce un cambio de configuración, la tarea no podrá devolver el resultado de la misma a la actividad que la encargó, pues se destruyó en el cambio de configuración

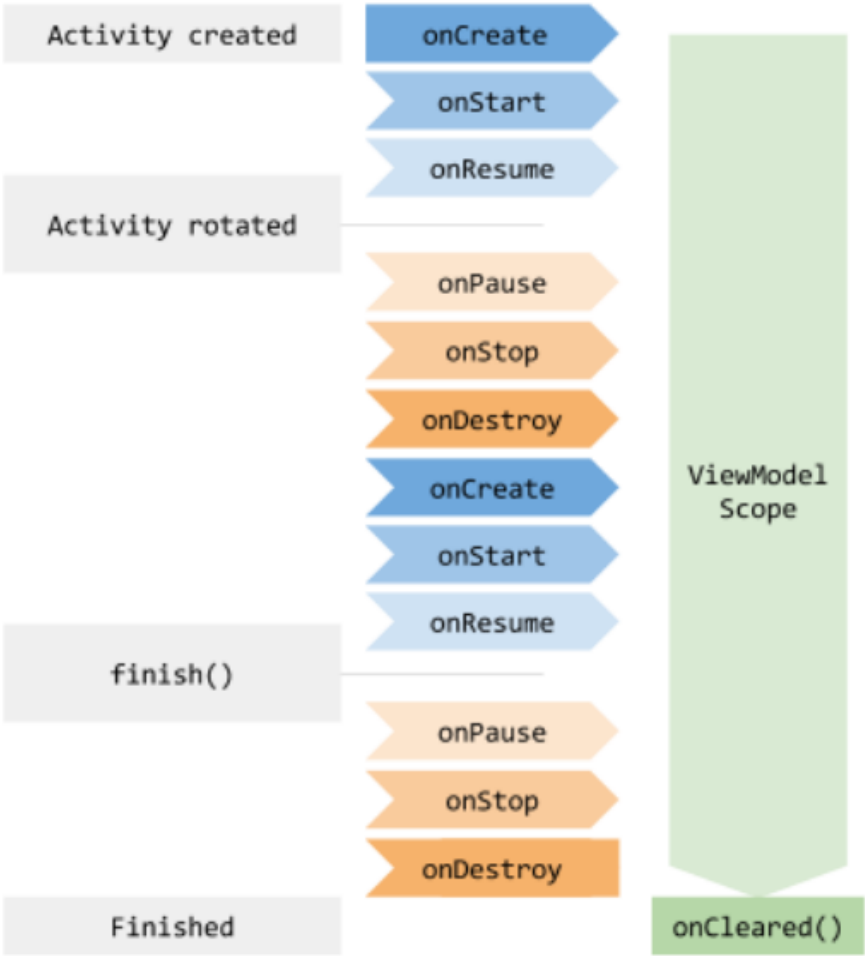


Pero ya teníamos onSaveInstanceState...

- Cierto, podíamos guardar datos en un bundle, y recuperar dicho bundle en el onCreate() de la nueva instancia de la actividad...
- Pero solo sirve para datos pequeños, simples
- No nos sirve para datos de cierto tamaño (de más de un megabyte aproximadamente), como una lista de usuarios o un bitmap grande...
- Para datos grandes podíamos usar un Fragment con setRetainInstance(true), pero era engorroso y complejo
- ViewModel puede sustituir la técnica del Fragment con setRetainInstance(true), y parcialmente a onSaveInstanceState(), pero no totalmente pues dicho método es imprescindible para conservar datos cuando la actividad se destruye por motivos diferentes a un cambio de configuración. Más Información

ViewModel: Introducción

- Diseñado para manejar datos relacionados con la interfaz de usuario teniendo en cuenta el ciclo de vida
- Así los datos “sobreviven” a los cambios de configuración
- El ViewModel se mantiene hasta que la actividad es destruida por motivos distintos a un cambio de configuración



Usando ViewModel

- Añadimos dependencias de gradle:

```
// ViewModel and LiveData
implementation "android.arch.lifecycle:extensions:1.0.0"
annotationProcessor "android.arch.lifecycle:compiler:1.0.0"
```

- Creamos una clase que extienda de ViewModel, con atributos para conservar información relativa a la vista (e idealmente con sus getters y setters)

```
public class ScoreViewModel extends ViewModel {
    // Tracks the score for Team A
    public int scoreTeamA = 0;

    // Tracks the score for Team B
    public int scoreTeamB = 0;
}
```

Usando ViewModel (II)

- Asociamos la actividad o fragmento al ViewModel creado
  - Normalmente tendremos un ViewModel por pantalla
  - Los ViewModels no deben tener referencias a Actividades, ni a Fragmentos, ni a Context, ni a ningún elemento de la vista (TextViews, EditTexts, etc.)
  - Una excepción a lo anterior es que a veces es necesario un “Application Context” (¡¡no “Activity Context”!!) para poder acceder a cosas como servicios del sistema por ejemplo
  - Almacenar un “Application Context” está permitido porque no está ligado a la actividad, sino a la aplicación. En esos casos deberíamos extender de **AndroidViewModel**

```
mViewModel = ViewModelProviders.of(this).get(ScoreViewModel.class);
```

“this” hace referencia a un “UI controller” (p.ej: una Actividad)

Usando ViewModel (III)

- Usamos los datos del ViewModel en nuestra actividad
  - En este ejemplo se accede a las propiedades directamente, sin getters ni setters, pero sería mejor hacer uso de ellos

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    mViewModel = ViewModelProviders.of(this).get(ScoreViewModel.class);
    displayForTeamA(mViewModel.scoreTeamA);
    displayForTeamB(mViewModel.scoreTeamB);
}

// An example of both reading and writing to the ViewModel
public void addOneForTeamA(View v) {
    mViewModel.scoreTeamA = mViewModel.scoreTeamA + 1;
    displayForTeamA(mViewModel.scoreTeamA);
}
```

LiveData

- Es una clase que almacena un **dato observable**
- Tiene en cuenta el ciclo de vida de otros componentes (actividades, fragmentos, servicios…)
- Solo notifica sobre cambios en el dato a los observadores de los componentes que se encuentren en un estado de ciclo de vida activo (STARTED o RESUMED)
- Las actividades y fragmentos que observan un LiveData son automáticamente desuscritos cuando son destruidos por razones distintas a un cambio de configuración
- Las actividades y fragmentos inactivos recibirán la última actualización del dato cuando estén activos de nuevo
- Después de una destrucción por cambio de configuración, las actividades o fragmentos reciben la última actualización del dato cuando se vuelven a crear

Crear LiveData

- LiveData es una clase abstracta
- MutableLiveData extiende LiveData y sí permite instancias
  - postValue(value): actualiza desde un hilo en segundo plano
  - setValue(value): actualiza desde el hilo principal

```
public class NameViewModel extends ViewModel {

// Create a LiveData with a String
private MutableLiveData<String> mCurrentName;

    public MutableLiveData<String> getCurrentName() {
        if (mCurrentName == null) {
            mCurrentName = new MutableLiveData<String>();
        }
        return mCurrentName;
    }

// Rest of the ViewModel...
}
```

Observar LiveData

- Desde nuestra actividad o fragment observamos los cambios que se produzcan en el dato observado
- Cuando el dato cambie, el método onChanged del observador será llamado

```
// Other code to setup the activity...

// Get the ViewModel.
mModel = ViewModelProviders.of(this).get(NameViewModel.class);

// Create the observer which updates the UI.
final Observer<String> nameObserver = new Observer<String>() {
    @Override
    public void onChanged(@Nullable final String newName) {
        // Update the UI, in this case, a TextView.
        mNameTextView.setText(newName);
    }
};

// Observe the LiveData, passing in this activity as the LifecycleOwner and the observer.
mModel.getCurrentName().observe(this, nameObserver);
```

Actualizar LiveData

- A veces nos interesa que en función de una interacción de usuario, se actualice un LiveData
- Si se trata de un MutableLiveData, haremos uso del método setValue (o postValue si no es desde el hilo principal)

```
mButton.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        String anotherName = "John Doe";
        mModel.getCurrentName().setValue(anotherName);
    }
});
```