DA: Design Patterns (MVVM)

Applications for mobile devices - Theory - Unit 4

Didac Florensa Cazorla

Any: 2021-2022
Curs: 102386
Institut: University of Lleida (Campus Igualada)
Titulació: Bachelor's degree in Digital Interaction and Computing Techniques (GTIDIC)

Agenda

Introduction

ViewModel (VM)

LiveData

Model View ViewModel (MVVM)

Data bindings

Homework

Introduction

Warm-up

- Dr. Jordi Mateo Fornés
- **Office**:
  - Office A.12 (Campus Igualada)
  - Office 3.08 (EPS Lleida)
- **Email**: jordi.mateo@udl.cat
- **Twitter**: https://twitter.com/MatForJordi
- **Github**: https://github/JordiMateoUdL

- Ph.D Dídac Florensa Cazorla
- **Office**:
  - Office A.12 (Campus Igualada)
  - Office 3.08 (EPS Lleida)
- **Email**: didac.florensa@udl.cat

ViewModel (VM)

What happens when configuration changes?

Recap: The Android framework manages the lifecycles of UI controllers, such as activities and fragments. The framework may decide to destroy or re-create a UI controller in response to certain user actions or device events that are completely out of your control.

Challenge

The problem is that an *Activity/Fragment* will go through a different state every time configuration changes. But, what happens to data introduced by the user, generated in the runtime or loaded from a DB...

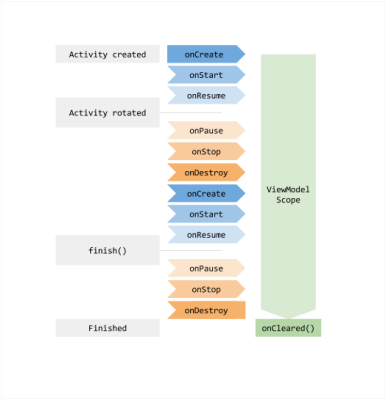
> How can we keep this data safe?

⇒ It would be great if the data did not need to know or manage what lifecycle state the *Activity/Fragment* is in? Instead of having a variable within the *Activity/Fragment*, and therefore tied to all the whims of the *Activity/Fragment* lifecycle, what if that data was stored somewhere outside of the *Activity/Fragment*?

ViewModel Class

The **ViewModel** class is designed to store and manage UI-related data in a lifecycle. So, it allows data to survive configuration changes, for example, in screen rotations.

- **ViewModel** objects are scoped to the Lifecycle passed to the *ViewModelProvider*. The *ViewModel* keeps in memory from (*onCreate*) until the Lifecycle it's scoped to goes away permanently (Activity-Finish, Fragment-Detached).
- **Note**: *onCreate* may be called several times during the life of an **Activity**, such as when the app is rotated, but the ViewModel survives throughout.
- Architecture Components provides the **ViewModel** helper class for the UI controller responsible for preparing data for the UI.



Implementation of the ViewModel

Imagine you need to display a list of users in your app. The responsibility to acquire and keep the list of users will be the ViewModel, instead of an *Activity/Fragment*

```
public class MyViewModel extends ViewModel {
    private MutableLiveData<List<User>> users;
    public LiveData<List<User>> getUsers() {
        if (users == null) {
            users = new MutableLiveData<List<User>>();
            loadUsers();
        }
        return users;
    }
    private void loadUsers() {
     // Do an asynchronous operation to fetch users.
    }
}
```

Using the ViewModel

```
    public void onCreate(Bundle savedInstanceState) {
        MyViewModel model = new ViewModelProvider(this)
            .get(MyViewModel.class);
        model.getUsers().observe(this, users -> {
            // update UI
        });
    }
}
```

If the *activity* is re-created, it receives the same `MyViewModel` instance created by the first activity. When the owner activity is finished, the framework calls the **ViewModel** objects' onCleared() method to clean up resources.

Considerations to take in mind

ViewModel objects are designed to outlive specific instantiations of views or LifecycleOwners. This design also means you can write tests to cover a ViewModel more quickly as it doesn't know about the view and Lifecycle objects.

Exemple

Suppose the **ViewModel** needs the Application context to find a system service. In that case, it can extend the *AndroidViewModel* class and have a constructor that receives the *Application* in the constructor since the *Application* class extends Context.

Caution

1. **ViewModels** should not, though, hold a reference to Activities, Fragments, or Contexts.
2. **ViewModels** should not contain elements that contain references to UI controllers, such as Views, since this will create an indirect reference to a Context.

LiveData

What is the LiveData?

Definition

LiveData is an *observable* data holder class. Unlike a regular observable, *LiveData* is lifecycle-aware, meaning it respects the lifecycle of other app components, such as activities, fragments, or services. This awareness ensures **LiveData** only updates app component observers in an active lifecycle state.

```
// LiveData
implementation 'androidx.lifecycle:lifecycle-livedata:X.Y.Z'
```

Which are the advantages of using LiveData?

Ensures your UI matches your data state

LiveData follows the observer pattern. ⇒ It notifies *Observer* when underlying data changes. You don't need to update the UI every time the app data changes because the *observer* does it for you.

No memory leaks

Observers are bound to Lifecycle objects and clean up after themselves when their associated lifecycle is destroyed.

Avoid crashes due to stopped activities

Nothing happens if the *observer's* lifecycle is inactive, such as in the case of an activity in the back stack, it doesn't receive any *LiveData* events.

No more manual lifecycle handling

UI components observe relevant data and don't stop or resume observation. **LiveData** automatically manages everything since it knows the appropriate lifecycle status changes while watching.

Which are the advantages of using LiveData?

Always up to date data

If a lifecycle becomes inactive, it receives the latest data upon becoming active again. For example, an activity in the background gets the latest data right after returning to the foreground.

Proper configuration changes

If an activity or fragment is recreated due to a configuration change, like device rotation, it immediately receives the latest available data.

Sharing resources

You can extend a **LiveData** object using the singleton pattern to wrap system services so that they can be shared in your app. The **LiveData** object connects to the system service once, and then any observer that needs the resource can watch the **LiveData** object.

What is the Singleton Pattern?

The Singleton pattern involves a single class responsible for creating an object while ensuring that only a single object gets created. This class provides a way to access its only object, which can be accessed directly without needing to instantiate the class object.

```
public class SingleObject {
   private static SingleObject instance = new SingleObject();
   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}
   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

}
```

Creating LiveData Objects

LiveData is a wrapper that can be used with any data, including objects that implement Collections, custom classes,...

```
public class NameViewModel extends ViewModel {
// Create a LiveData with a String
private MutableLiveData<String> currentName;

    public MutableLiveData<String> getCurrentName() {
        if (currentName == null) {
            currentName = new MutableLiveData<String>();
        }
        return currentName;
    }
// Rest of the ViewModel...
}
```

Observing LiveData

```
public class MainActivity extends AppCompatActivity {
    private MainActivityViewModel model;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        model = new ViewModelProvider(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.
                nameTextView.setText(newName);
            }
        };
        // Observe the LiveData
        model.getCurrentName().observe(this, nameObserver);
    }}
```

Update LiveData objects

LiveData has no publicly available methods to update the stored data. The **MutableLiveData** class exposes the *setValue(T)* and *postValue(T)* methods publicly, and you must use these if you need to edit the value stored in a LiveData object.

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

TIP: Usually, MutableLiveData is used in the ViewModel, and then the ViewModel only exposes immutable LiveData objects to the observers.

Note: Async or Sync? You must call the *setValue(T)* method to update the LiveData object from the main thread. If the code is executed in a worker thread, you can use the *postValue(T)*.

Model View ViewModel (MVVM)

What is the MVVM?

Definition

A design pattern overcomes all drawbacks of *MVP* and *MVC* in the Android environment.

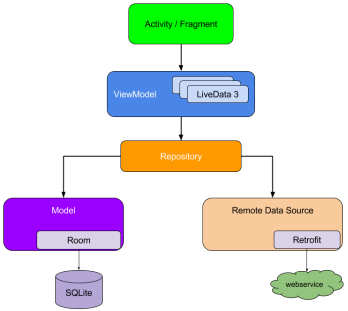


Figure 1: Schema of the MVVM architecture

How to implement the MVVM?

It proposes to separate code layers in:

- **Model**: This layer is responsible for the abstraction of the data sources. Model and ViewModel work together to get and save the data.
- **View**: The purpose of this layer is to inform the ViewModel about the user's action. This layer observes the ViewModel and does not contain any application logic.
- **ViewModel**: It exposes those data streams relevant to the View. Moreover, it serves as a link between the Model and the View.

There are two ways to implement MVVM design patterns in Android projects:

- Using the **DataBinding** library released by Google. **RECOMMENDED**
- Using any tool like *RxJava* for **DataBinding**. (NOT RECOMENDED)

Data bindings

What are the databindings?

Definition

The **Data Binding** Library is a support library that allows you to bind UI components in your layouts to data sources in your app using a declarative format rather than programmatically.

Activity/Fragment

```
TextView textView =
findViewById(R.id.sample_text);
textView.setText(
    viewModel.getUserName());
```

Layout (xml)

```
<TextView
    android:text="
    @{viewmodel.userName}" />
```

⇒ Binding components in the layout file lets you remove many UI framework calls in your activities, making them simpler and easier to maintain. **Note**: In many cases, view binding can provide the same benefits as data binding with simpler implementation and better performance. If you are using data binding primarily to replace findViewById() calls, consider using *view binding* instead.

Before using it?

Edit the build.gradle and add:

```
android {
...
    dataBinding {
       enabled true
    }
}
```

How to use it?

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto">
    <data>
        <variable
            name="viewmodel"
            type="com.myapp.data.ViewModel" />
    </data>
    <ConstraintLayout... /> <!-- UI layout's root element -->
</layout>
```

Expressions

One-way binding

```
// Bind the name property of the viewmodel to the text attribute
<android:text="@{viewmodel.name}">
// Bind the nameVisible property of the viewmodel to the visibility attribute
<android:visibility="@{viewmodel.nameVisible}">
// Call the onLike() method on the viewmodel when the View is clicked.
<android:onClick="@{() -> viewmodel.onLike()}">
```

Two ways binding

The `@={}` receives data changes to the property and listen to user updates at the same time.

```
<CheckBox
    android:id="@+id/rememberMeCheckBox"  android:checked="@={viewmodel.rememberMe}"/>
```

Binding the Activity

```
class ViewModelActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
    // Inflate view and obtain an instance of the binding class.
    UserBinding binding =
        DataBindingUtil.setContentView(this, R.layout.user);
    // Specify the current activity as the lifecycle owner.
    binding.setLifecycleOwner(this);
    }
}
```

Binding the Activity and the ViewModel

```
class ViewModelActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
    // Obtain the ViewModel component.
    UserModel userModel = new ViewModelProvider(this).get(UserModel.class);
    UserBinding binding = DataBindingUtil.setContentView(this, R.layout.user);
    // Assign the component to a property in the binding class.
    binding.viewmodel = userModel;
    }
}
```

Homework

Task A: Review concepts

- Review this Google CodeLab. It is in Kotlin, but the data binding part in XML is the same in JAVA.
- Review this repo MVVM.
- Review the live-code done in class. DAM-LiveCoding-Paper-Rock-Scissor-Lizard-Spock.

Task B: Project

Time to code and make progress for the Sprint II.

That's all

QUESTIONS?

About me

www — jordimateofornes.com
github — github.com/JordiMateo
twitter — @MatForJordi
gdc — Distributed computation group