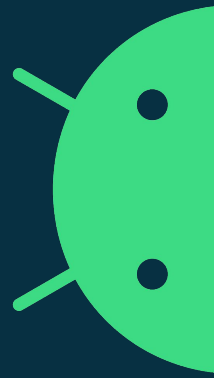
MVVM ARCHITECTURE



android

1

ARCHITETURE COMPONENT
View binding - Databinding

android

View binding

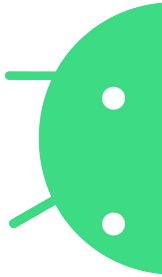
- Enable view binding in a module

```
android {
    ...
    buildFeatures {
        viewBinding = true
    }
}
```

- layout will be ignored while generating binding classes

```
<LinearLayout
        ...
        tools:viewBindingIgnore="true" >
    ...
</LinearLayout>
```

View binding vs FindViewById



- **Null safety:** Since view binding creates direct references to views, there's no risk of a null pointer exception due to an invalid view ID

- **Type safety:** The fields in each binding class have types matching the views they reference in the XML file

View binding vs data binding

- view binding is intended to handle simpler use cases and provides the following benefits over data binding
  - **Faster compilation:** View binding requires no annotation processing, so compile times are faster.
  - **Ease of use:** View binding does not require specially-tagged XML layout files, so it is faster to adopt in your apps.

- view binding has the following limitations compared to data binding
  - View binding doesn't support layout variables or layout expressions
  - View binding doesn't support two-way data binding.

Data binding

- 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

```
findViewById<TextView>(R.id.sample_text).apply {
    text = viewModel.userName
}
```

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

Using data binding

- enable the dataBinding build option in your build.gradle

```
android {
    ...
    buildFeatures {
        dataBinding true
    }
}
```

Data binding - Layouts and binding expressions

- The expression language allows you to write expressions that handle events dispatched by the views

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <variable name="user" type="com.example.User"/>
    </data>
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{user.firstName}"/>
        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{user.lastName}"/>
    </LinearLayout>
</layout>
```

Layouts and binding expressions - binding data

- Activity

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    val binding: ActivityMainBinding = DataBindingUtil.setContentView(
            this, R.layout.activity_main)

    binding.user = User("Test", "User")
}
```

- XML

```
android:text="@{user.lastName}"
```

Layouts and binding expressions - binding data

- If you are using data binding items inside a Fragment, ListView, or RecyclerView adapter, you may prefer to use the inflate() methods of the bindings classes or the DataBindingUtil class

```
val listItemBinding = ListItemBinding.inflate(layoutInflater, viewGroup, false)
// or
val listItemBinding = DataBindingUtil.inflate(layoutInflater, R.layout.list_item, viewGroup, false)
```

Layouts and binding expressions - binding data

- Expression language
  - Common features
  - Null coalescing operator
  - Property references
  - View references
  - Collections
  - Resources ….

```
<data>
    <import type="android.util.SparseArray"/>
    <import type="java.util.Map"/>
    <import type="java.util.List"/>
    <variable name="list" type="List&lt;String>"/>
    <variable name="sparse" type="SparseArray&lt;String>"/>
    <variable name="map" type="Map&lt;String, String>"/>
    <variable name="index" type="int"/>
    <variable name="key" type="String"/>
</data>
…
android:text="@{list[index]}"
…
android:text="@{sparse[index]}"
…
android:text="@{map[key]}"
```

Layouts and binding expressions - Event handing

- Activity

```
class MyHandlers {
    fun onClickFriend(view: View) { ... }
}
```

- XML

```
<TextView android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@{user.firstName}"
    android:onClick="@{handlers::onClickFriend}"/>
```

Imports, variables, and includes

```
<data>
    <import type="com.example.User"/>
    <import type="java.util.List"/>
    <variable name="user" type="User"/>
    <variable name="userList" type="List&lt;User>"/>
</data>
```

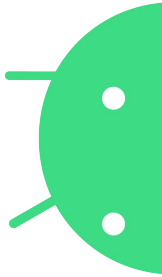
```
<LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <include layout="@layout/name"
        bind:user="@{user}"/>
    <include layout="@layout/contact"
        bind:user="@{user}"/>
</LinearLayout>
```

Imports, variables, and includes

- Data binding doesn’t support include as a direct child of a merge element

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:bind="http://schemas.android.com/apk/res-auto">
    <data>
        <variable name="user" type="com.example.User"/>
    </data>
    <merge><!-- Doesn't work -->
        <include layout="@layout/name"
            bind:user="@{user}"/>
        <include layout="@layout/contact"
            bind:user="@{user}"/>
    </merge>
</layout>
```

Work with observable data objects



- Observability refers to the capability of an object to notify others about changes in its data. The Data Binding Library allows you to make objects, fields, or collections observable
- There are three different types of observable classes: objects, fields, and collections.
- When one of these observable data objects is bound to the UI and a property of the data object changes, the UI is updated automatically
- Android Studio 3.1 and higher allow you to replace observable fields with LiveData objects,

Work with observable data objects - Observable fields

- ObservableBoolean
- ObservableByte
- ObservableChar
- ObservableShort
- ObservableInt
- ObservableLong
- ObservableFloat
- ObservableDouble
- ObservableParcelable

```
class User {
    val firstName = ObservableField<String>()
    val lastName = ObservableField<String>()
    val age = ObservableInt()
}
```

```
user.firstName = "Google"
val age = user.age
```

To access the field value, either use the set() and get() accessor methods or use Kotlin property syntax

Work with observable data objects - Observable object

To make the object observable, extend the class from BaseObservable.

- To make a property observable, use **@Bindable** annotation on getter method.
- Call **notifyPropertyChanged(BR.property)** in setter method to update the UI whenever the data is changed
- The **BR** class will be generated automatically when data binding is enabled.

```
class User : BaseObservable() {

    @get:Bindable
    var firstName: String = ""
        set(value) {
            field = value
            notifyPropertyChanged(BR.firstName)
        }

    @get:Bindable
    var lastName: String = ""
        set(value) {
            field = value
            notifyPropertyChanged(BR.lastName)
        }
}
```

Binding Adapter - BindingMethods

Some attributes have setters that don't match by name. In these situations, an attribute may be associated with the setter using the BindingMethods annotation.

example, the android:tint attribute is associated with the setImageTintList(ColorStateList) method, not with the setTint()

```
@BindingMethods(value = [
    BindingMethod(
        type = android.widget.ImageView::class,
        attribute = "android:tint",
        method = "setImageTintList")])
```

Binding Adapter - BindingAdapter

A **static** binding adapter method with the BindingAdapter annotation allows you to customize how a setter for an attribute is calle

You can also have adapters that receive multiple attributes

```
@BindingAdapter("android:paddingLeft")
fun setPaddingLeft(view: View, padding: Int) {
    view.setPadding(padding,
                    view.getPaddingTop(),
                    view.getPaddingRight(),
                    view.getPaddingBottom())
}
```

```
@BindingAdapter("imageUrl", "error")
fun loadImage(view: ImageView, url: String, error: Drawable) {
    Picasso.get().load(url).error(error).into(view)
}
```

```
<ImageView app:imageUrl="@{venue.imageUrl}" app:error="@{@drawable/venueError}" />
```

Bind layout views to Architecture Components

android

Two way data binding

To make the object observable, extend the class from BaseObservable.

- To make a property observable, use **@Bindable** annotation on getter method.
- Call **notifyPropertyChanged(BR.property)** in setter method to update the UI whenever the data is changed
- The **BR** class will be generated automatically when data binding is enabled.

```
class User : BaseObservable() {

    @get:Bindable
    var firstName: String = ""
        set(value) {
            field = value
            notifyPropertyChanged(BR.firstName)
        }

    @get:Bindable
    var lastName: String = ""
        set(value) {
            field = value
            notifyPropertyChanged(BR.lastName)
        }
}
```

2

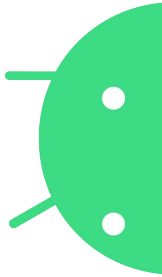
ARCHITETURE COMPONENT
lifecycle-aware componets

android

Handle lifecycle - normal

```
internal class MyLocationListener(
        private val context: Context,
        private val callback: (Location) -> Unit
) {

    fun start() {
        // connect to system location service
    }

    fun stop() {
        // disconnect from system location service
    }
}

class MyActivity : AppCompatActivity() {
    private lateinit var myLocationListener: MyLocationListener

    override fun onCreate(...) {
        myLocationListener = MyLocationListener(this) { location ->
            // update UI
        }
    }

    public override fun onStart() {
        super.onStart()
        myLocationListener.start()
        // manage other components that need to respond
        // to the activity lifecycle
    }

    public override fun onStop() {
        super.onStop()
        myLocationListener.stop()
        // manage other components that need to respond
        // to the activity lifecycle
    }
}
```

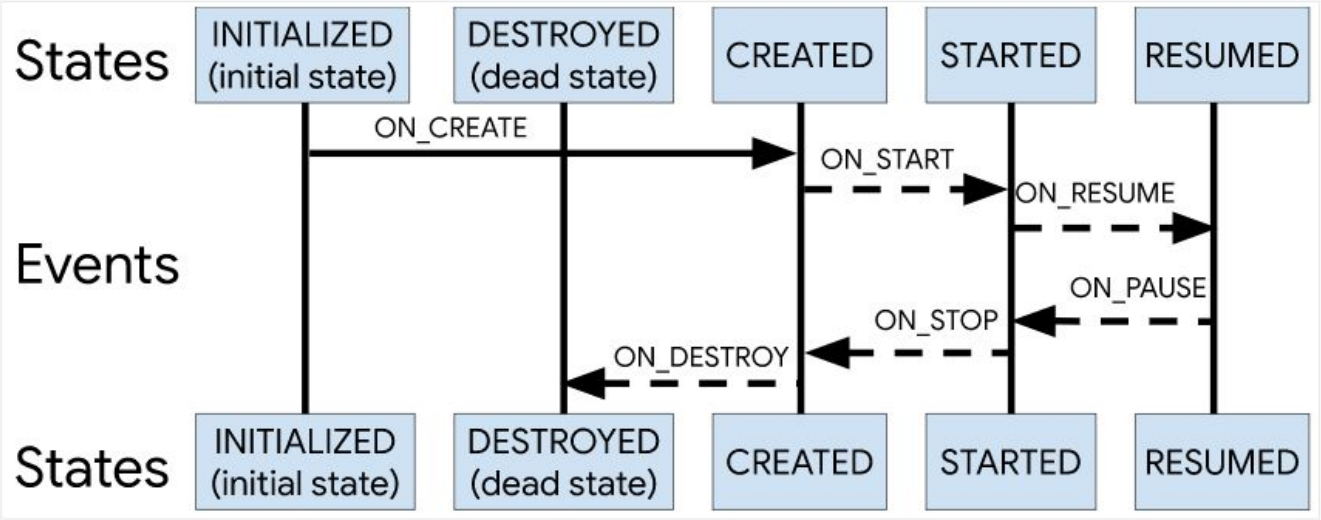
asynchronous



This

Handle lifecycle - androidx.lifecycle

- Lifecycle is a class that holds the information about the lifecycle state of a component (like an activity or a fragment) and allows other objects to observe this state.
- Lifecycle uses two main enumerations to track the lifecycle status for its associated component:



Handle lifecycle - lifecycleObservable

```
class MyLifeCycleObserver(private val lifeCycle: Lifecycle, private val logger: MyLogger) : LifecycleObserver {

    @OnLifecycleEvent(Lifecycle.Event.ON_CREATE)
    fun logCreate() {
        logger.logCreate()
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_START)
    fun logStart() {
        if (lifeCycle.currentState.isAtLeast(Lifecycle.State.STARTED)) {
            logger.logStart()
        }
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
    fun logResume() {
        logger.logResume()
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
    fun logPause() {
        logger.logPause()
    }
```

Handle lifecycle - lifecycleObservable

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    val lifeCycleObserver = MyLifeCycleObserver(lifecycle, MyLogger())
    lifecycle.addObserver(lifeCycleObserver)
```

Best practices for lifecycle-aware components

- Keep UI controllers as lean as possible. Use ViewModel to manage logic and data, and observable a object to reflect the changes back to the views
- Try to write data-driven UIs where your UI controller's responsibility is to update the views as data changes, or notify user actions back to the ViewModel
- Use Data Binding to maintain a clean interface between your views and the UI controller.
- Avoid referencing a View or Activity context in your ViewModel. If the ViewModel outlives the activity (in case of configuration changes), your activity leaks and isn't properly disposed by the garbage collector.
- Use Kotlin coroutines to manage long-running tasks and other operations that can run asynchronously.

ViewModel - problem

- If the system destroys or re-creates a UI controller, any transient UI-related data you store in them is lost. For simple data, the activity can use the onSaveInstanceState() method and restore its data from the bundle in onCreate(), but this approach is only suitable for small amounts of data that can be serialized then deserialized, not for potentially large amounts of data like a list of users or bitmaps.
- UI controllers frequently need to make asynchronous calls that may take some time to return. The UI controller needs to manage these calls and ensure the system cleans them up after it's destroyed to avoid potential memory leaks. This management requires a lot of maintenance, and in the case where the object is re-created for a configuration change, it's a waste of resources since the object may have to reissue calls it has already made.
- Assigning excessive responsibility to UI controllers can result in a single class that tries to handle all of an app's work by itself, instead of delegating work to other classes. Assigning excessive responsibility to the UI controllers in this way also makes testing a lot harder.

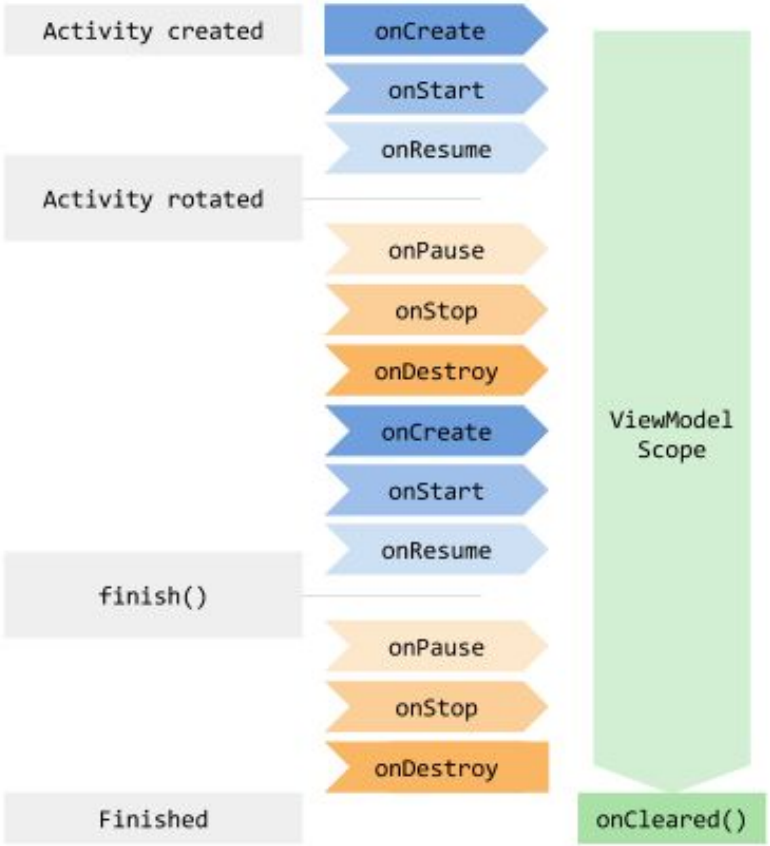
ViewModel - implement

- Architecture Components provides ViewModel helper class for the UI controller that is responsible for preparing data for the UI. ViewModel objects are automatically retained during configuration changes so that data they hold is immediately available to the next activity or fragment instance

```
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.
    }
}
```

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

ViewModel - Lifecycle



ViewModel - Share data between fragment

Benefit

- The activity does not need to do anything, or know anything about this communication.
- Fragments don't need to know about each other besides the SharedViewModel contract. If one of the fragments disappears, the other one keeps working as usual.
- Each fragment has its own lifecycle, and is not affected by the lifecycle of the other one. If one fragment replaces the other one, the UI continues to work without any problems.

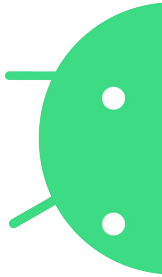
```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    model = ViewModelProvider(requireActivity()).get(SharedViewModel::class.java)
    model.select( item: "Sanh test nhe cai share viewmodel")
}
```

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    model = ViewModelProvider(requireActivity()).get(SharedViewModel::class.java)
    model.selected.observe(viewLifecycleOwner, Observer<String> { item ->
        Toast.makeText(activity, text: "ListFragment " + item, Toast.LENGTH_LONG).show()
    })
}
```

LiveData - What ?

- 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 that are in an active lifecycle state
- Benefit
  - **Ensures your UI matches your data state**
  - **No memory leaks**
  - **No crashes due to stopped activities**
  - **No more manual lifecycle handling**
  - **Always up to date data**
  - **Proper configuration changes**
  - **Sharing resources**

LiveData - Work with LiveData objects



- Create an instance of LiveData to hold a certain type of data. This is usually done within your ViewModel class.
- Create an Observer object that defines the onChanged() method, which controls what happens when the LiveData object's held data changes. You usually create an Observer object in a UI controller, such as an activity or fragment.
- Attach the Observer object to the LiveData object using the observe() method. The observe() method takes a LifecycleOwner object. This subscribes the Observer object to the LiveData object so that it is notified of changes. You usually attach the Observer object in a UI controller, such as an activity or fragment.

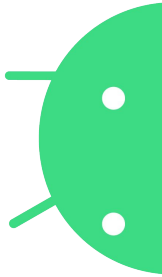
LiveData - create livedata object

```
class NameViewModel : ViewModel() {

    // Create a Live
    val currentName
        MutableLiveD
    }

    // Rest of the V
}
```

```
class NameViewModel : ViewModel() {

    // Create a LiveData with a String
    val currentName: MutableLiveData<String> by lazy {
        MutableLiveData<String>()
    }

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


android

LiveData - Observe LiveData objects



```
private val model: NameViewModel by viewModels()

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    // Other code to setup the activity...

    // 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)
}
```

LiveData - Update LiveData Object

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

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

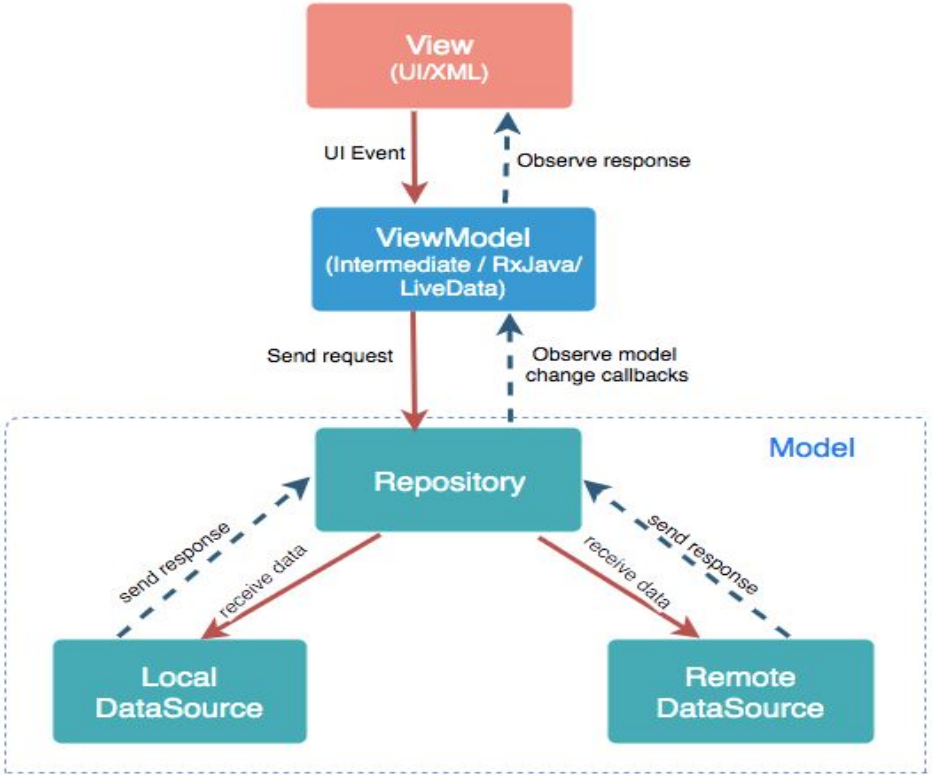
3

MVVM

MVVM - What?

- **Model:** It represents the data and the business logic of the Android Application. It consists of the business logic - local and remote data source, model classes, repository.
- **View:** It consists of the UI Code(Activity, Fragment), XML. It sends the user action to the ViewModel but does **not** get the response back directly. To get the response, it has to subscribe to the observables which ViewModel exposes to it.
- **ViewModel:** It is a bridge between the View and Model(business logic). It does not have any clue which View has to use it as it does not have a direct reference to the View. So basically, the ViewModel should not be aware of the view who is interacting with. It interacts with the Model and exposes the observable that can be observed by the View.

MVVM - What?


View
(UI/XML)
UI Event
Observe response
ViewModel
(Intermediate / RxJava/
LiveData)
Send request
Observe model
change callbacks
Model
Repository
send response
receive data
receive data
send response
Local
DataSource
Remote
DataSource

- Live data is not public method setValue and postValue (# MutableLiveData)