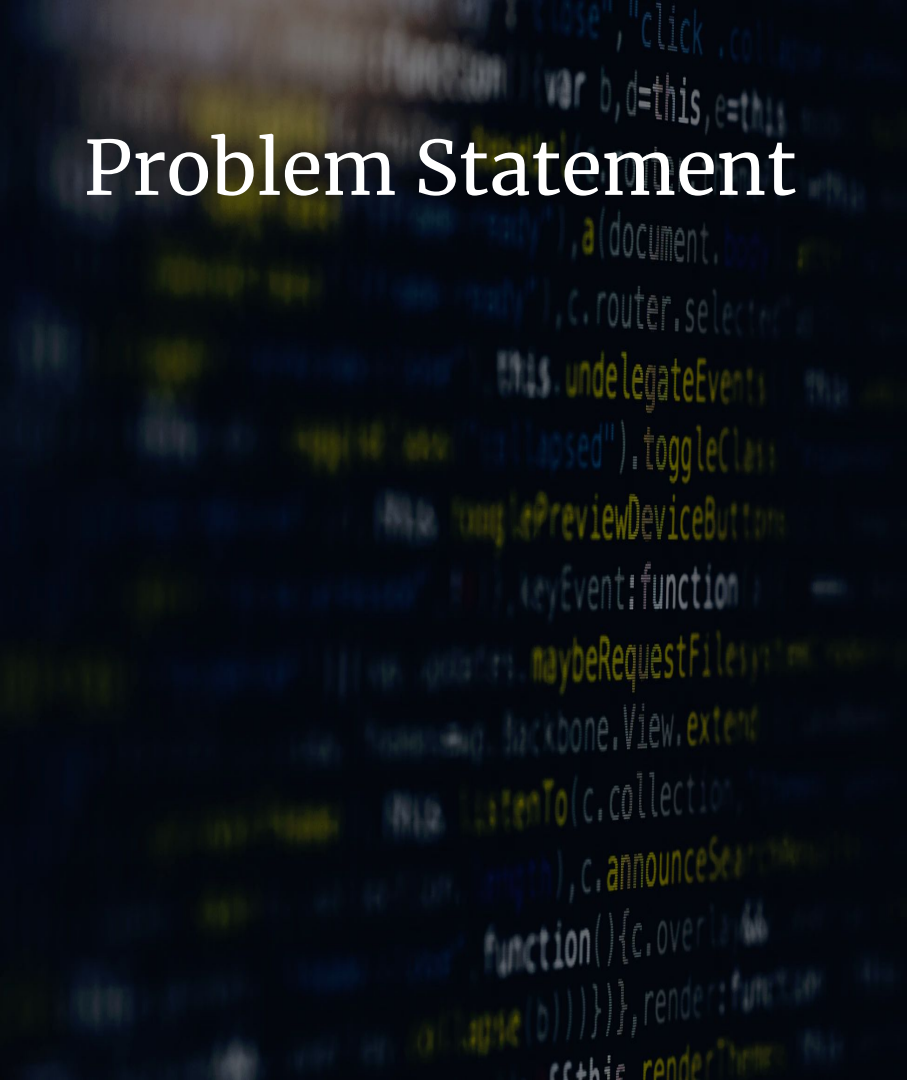
MVVM with Data Binding & LiveData

Jeff Cardillo
Android Summit
October 2019



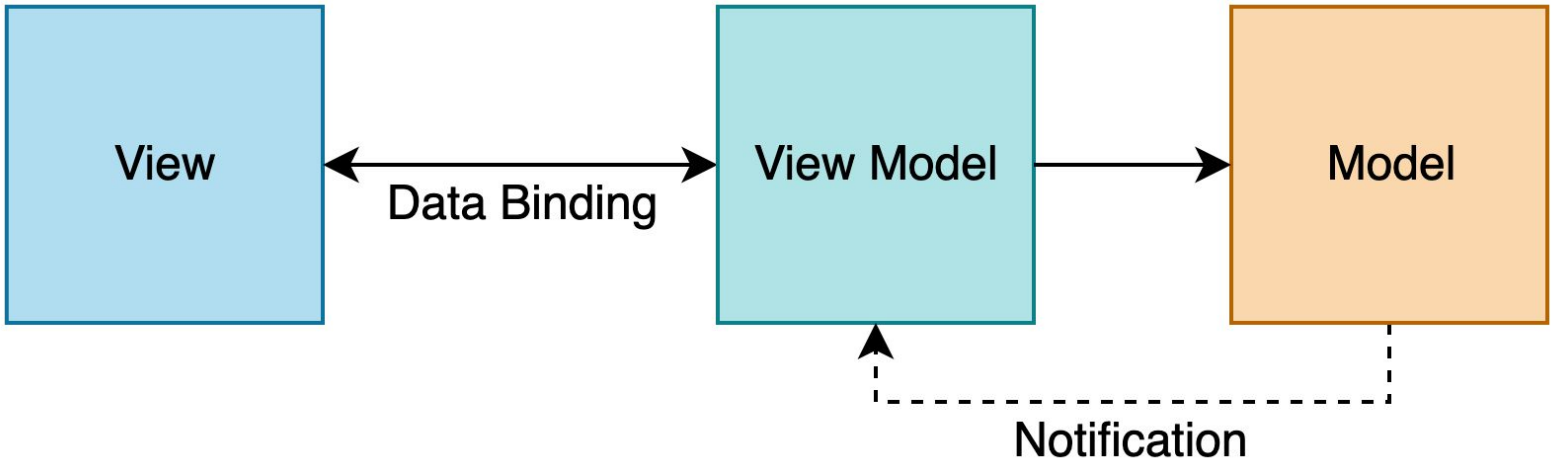
Problem Statement

We want to design our software in such a way that separates responsibility between GUI, business logic, and data layers.

Introducing MVVM

Model-View-Viewmodel is a variation of the Model-View-Presenter pattern to simplify event-driven programming of user interfaces.

How Does it Work?



View

Layout and appearance of what the user sees on the screen.

Displays representation of View Model

Receives user input and forwards the handling of these to view model (via data binding)

Viewmodel

Decorates business logic for the purpose of displaying and interacting with it.

Should be decoupled from a view. The viewmodel should not know who is interacting with it.

Model

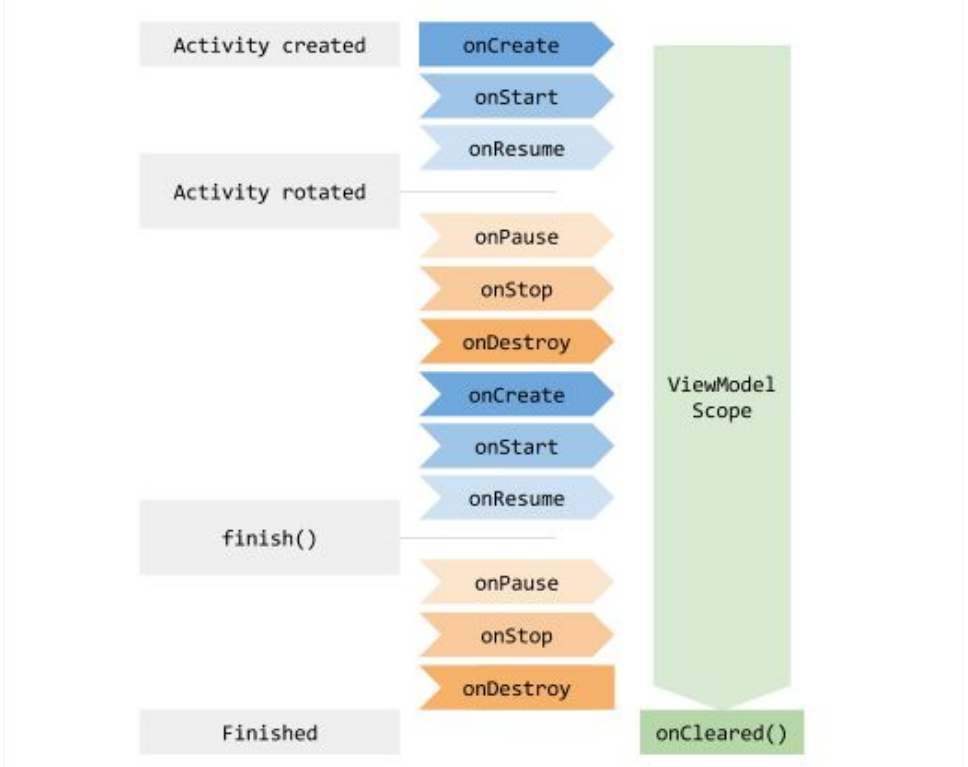
Represents the data.

Holds information, not behaviors or services that manipulate the data.

Keep in mind

- ViewModel is a helper class in Arch. Components that is responsible for preparing data for the UI.
- ViewModel objects are retained across config changes so their data persists and can be reused.
- ViewModels should never reference a View or any class that holds a reference to the activity context.

ViewModel with Activity Lifecycle



How to use ViewModel

Example ViewModel:

```
class MyViewModel : ViewModel() {
    private val users: MutableLiveData<List<User>> by lazy {
        MutableLiveData().also {
            loadUsers()
        }
    }

    fun getUsers(): LiveData<List<User>> {
        return users
    }

    private fun loadUsers() {
        // Do an asynchronous operation to fetch users.
    }
}
```

Using the ViewModel:

```
class MyActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        // Create a ViewModel the first time the system calls an activity's o
        // Re-created activities receive the same MyViewModel instance create

        val model = ViewModelProviders.of(this)[MyViewModel::class.java]
        model.getUsers().observe(this, Observer<List<User>>{ users ->
            // update UI
        })
    }
}
```

MVVM is Not a Silver Bullet

As with all patterns, there are problems that they solve. There are also times when they should not be used:

- Overkill for very simple UI
- Memory footprint in very large applications considerable.

Data Binding

The **Data Binding Library** allows you to bind UI components in your layouts to data sources using a declarative format (rather than programmatically).

Data Binding Simple Example

```xml
<?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>
```

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

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

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

Data Binding

Classes are generated to bind layout properties to corresponding views

Data Binding

If data will be changing often and the UI needs to update dynamically, then the object needs to notify of changes.

1. Extend **`BaseObservable`**
2. Assign **`@Bindable`**
3. Annotate the getter
4. **Notify** in the setter.

Data Binding

BR is auto-generated class for `SortController`

UI bound to notify event

```
class SortController : BaseObservable() {
    var sortAscending = true
        private set

    @get:Bindable
    var nextSortText : String  = SORT_ASC
        private set

    fun switchNextSortDirection() {
        sortAscending = !sortAscending

        this.nextSortText = if (sortAscending) SORT_ASC else SORT_DESC

        notifyPropertyChanged(BR.nextSortText)
    }

    companion object {
        private const val SORT_ASC = "Sort A...Z"
        private const val SORT_DESC = "Sort Z...A"
    }
}
```

```
<Button
    android:id="@+id/sort_button"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    layout_constraintTop_toTopOf="parent"
    android:text="@{sortController.nextSortText}"
    android:onClick="@{() -> viewModel.sortMovies()}" />
```

A note about Custom BindingAdapters

Custom `BindingAdapters` allow complex transformations of data

```
<ImageView
          android:id="@+id/icon"
          android:layout_width="40dp"
          android:layout_height="fill_parent"
          android:layout_alignParentBottom="true"
          android:layout_alignParentTop="true"
          android:layout_marginRight="6dip"
          android:contentDescription="TODO"
          android:src="@{obj.url}"
          />
```

Register for this property on ImageView and use Glide to download images:

```
@BindingAdapter("android:src")
    public static void setImageUrl(ImageView view, String url) {
        Glide.with(view.getContext()).load(url).into(view);
    }
```

ViewModel Communication

Can be accomplished with a number of techniques, including:

- EventBus
- RxAndroid
- Observer pattern,
- Google’s ViewModel architecture component with **LiveData**

LiveData

LiveData is a wrapper for data that is susceptible to change.

LiveData is lifecycle aware and so will only update observers that are in active state.

Using LiveData

Declare something as LiveData

```
private var movieList: MutableLiveData<List<TheMovieDbMovie>> = MutableLiveData()
```

Set the value

```
movieList?.setValue(movieResponse?.results)
```

Observe changes on the data

```
viewModel.getPopularMovies().observe( owner: this,
    Observer<List<TheMovieDbMovie>> { movieList :List<TheMovieDbMovie>! ->
        activity?.let { it: FragmentActivity
            adapter = MovieAdapter(it, movieList)
            recyclerView?.setAdapter(adapter) ^let
        }
    })
```

Introduction to the Workshop Project

We are going to build a project that uses all of these concepts!

- Exemplifies use of **MVVM**, **Data Binding**, and **LiveData** patterns
- Integrates with The Movie DB API
- Project available at this GitHub Repository

Workshop Overview

- The repository has a “starter” set of code and a “completed” set of code.
- Starter - project with most of the code available, but missing some key pattern code that we will fill in together.
- Completed - this is the completed working project with all of the pattern code filled in.

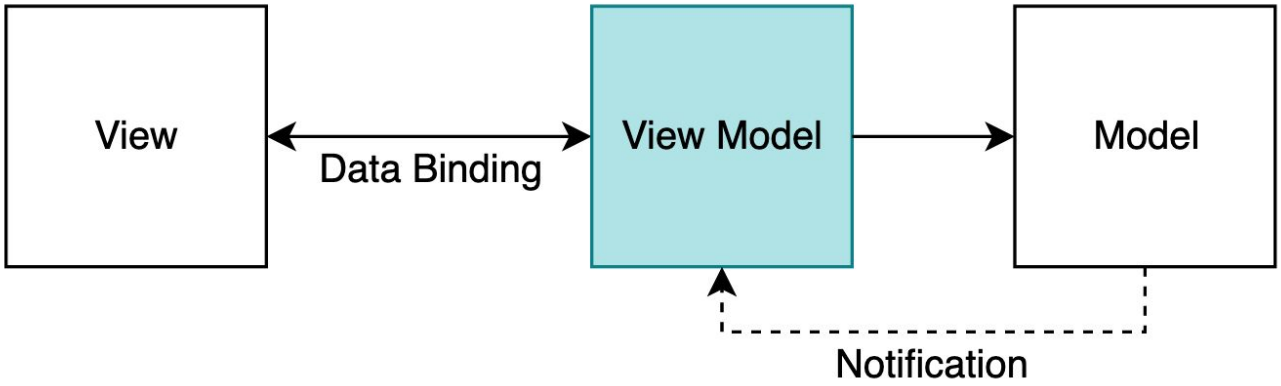
Set up your own TheMovieDB API Key

- Create an account at TheMovieDB and register a new app. Approval is instant!
- In your IDE, add the following line to your **`local.properties`** file:
  `THE_MOVIE_DB_API_KEY = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"`
- The gradle file will use this at build time and add it to the `BuildConfig` object, which is then used in the code.

First, let's build the ViewModel

Remember:

- Coordinates the data fetch when needed
- Notifies listeners of data updates through Data Bindings / Live Data



The ViewModel - Define Variables

- Variables needed for our ViewModel
- **`popularMovieService`** is used to fetch a list of popular movies
- **`movieList`** is an instance of **`MutableLiveData`**
- **`sortController`** extends **`BaseObservable`**

```
class MainViewModel : ViewModel() {

    private val popularMovieService :TheMovieDbApi  = ApiFactory.theMovieDbApi

    private var movieList: MutableLiveData<List<TheMovieDbMovie>>? = null

    var sortController = SortController()
```

The ViewModel - Populating Data

- Loading from the API happens in a coroutine that is launched with the **`viewModelScope`**
- The **`liveData.setValue`** method is what will trigger observers that the data has changed

```
fun getPopularMovies() : LiveData<List<TheMovieDbMovie>> {
    if (movieList == null) {
        movieList = MutableLiveData()
        loadPopularMovies()
    }

    return movieList as LiveData<List<TheMovieDbMovie>>
}

private fun loadPopularMovies() {
    viewModelScope.launch(Dispatchers.Main) { this: CoroutineScope
        val popularMovieRequest = popularMovieService.getPopularMovieAsync()
        try {
            val response = popularMovieRequest.await()
            if (response.isSuccessful) {
                val movieResponse = response.body()
                movieList?.setValue(movieResponse?.results)
            } else {
                Log.d( tag: "MainViewModel ", response.errorBody().toString())
            }
        } catch (e: Exception) { }
    }
}
```

The ViewModel - The Sort Method

- The **`sortMovies()`** method will be bound to the sort button in XML

```
fun sortMovies() {
    movieList?.let { it: MutableLiveData<List<TheMovieDbMovie>>
        if (sortController.sortAscending) {
            movieList?.setValue(movieList?.value?.sortedBy { it.title }) ^let
        } else {
            movieList?.setValue(movieList?.value?.sortedByDescending { it.title }) ^let
        }
    }

    sortController.switchNextSortDirection()
}
```

The SortController Class

- This class will keep track of the next sort direction (ascending / descending) and automatically update the text on the sort button through Data Binding
- Included to show dynamic updating of properties declared in xml (sort button name)

```
class SortController : BaseObservable() {
    var sortAscending = true
        private set

    @get:Bindable
    var nextSortText : String  = SORT_ASC
        private set

    fun switchNextSortDirection() {
        sortAscending = !sortAscending

        this.nextSortText = if (sortAscending) SORT_ASC else SORT_DESC

        notifyPropertyChanged(BR.nextSortText)
    }

    companion object {
        private const val SORT_ASC = "Sort A...Z"
        private const val SORT_DESC = "Sort Z...A"
    }
}
```

Setting up the Data Bindings

- Note that layout files that use Data Bindings are wrapped in layout tags
- In **`main_fragment.xml`** there are two variables declared

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <variable
            name="viewModel"
            type="com.jeffcardillo.androidsummit.themoviedb.ui.MainViewModel"/>

        <variable
            name="sortController"
            type="com.jeffcardillo.androidsummit.themoviedb.ui.SortController"/>
    </data>
```

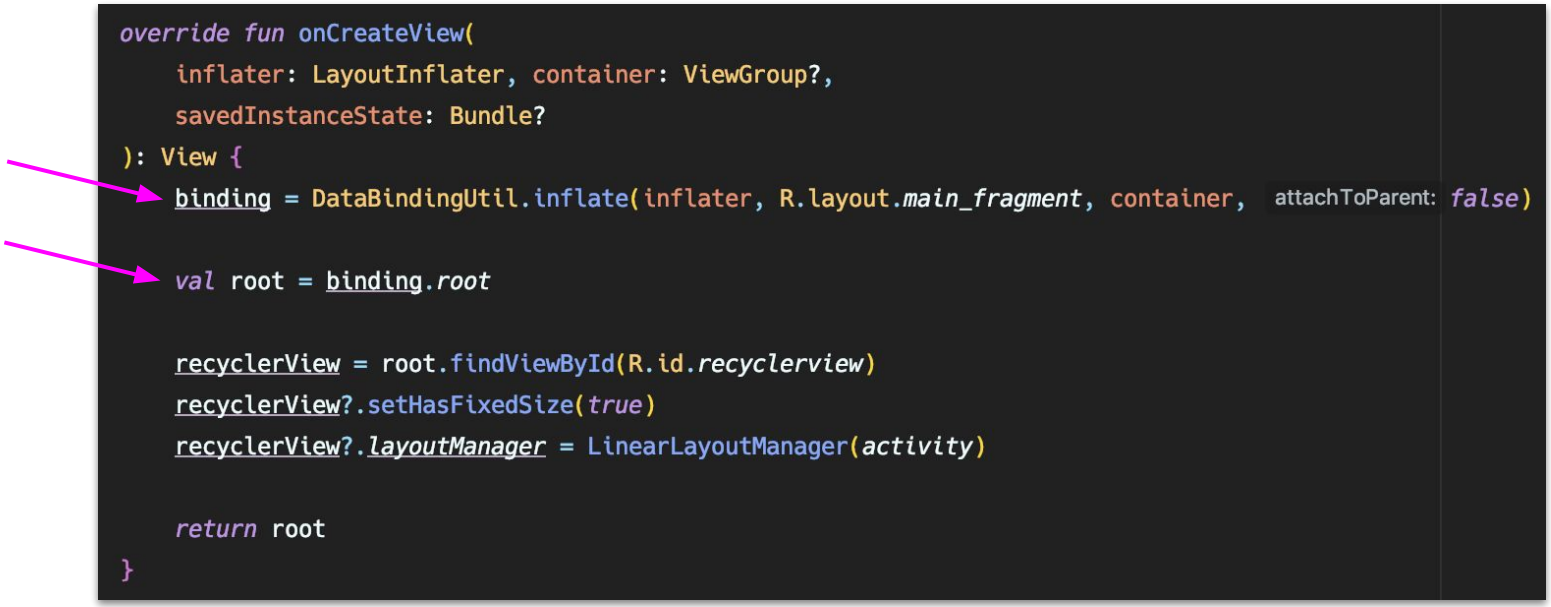
Setting up the Data Bindings

- Hook up the `sort_button` to use the **`sortController`** variable to set the button text and to call **`viewModel.sortMovies()`** during an onClick event

```
<Button
    android:id="@+id/sort_button"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    layout_constraintTop_toTopOf="parent"
    android:text="@{sortController.nextSortText}"
    android:onClick="@{() -> viewModel.sortMovies()}" />
```

Wiring in the View

- Inflate the binding layout in **`onCreateView`** in the **`MainFragment.java`** class



```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View {
    binding = DataBindingUtil.inflate(inflater, R.layout.main_fragment, container,  attachToParent: false)

    val root = binding.root

    recyclerView = root.findViewById(R.id.recyclerview)
    recyclerView?.setHasFixedSize(true)
    recyclerView?.layoutManager = LinearLayoutManager(activity)

    return root
}
```

Wiring the ViewModel to the View

- Create the ViewModel, set the XML bindings using the binding object.
- Set up the observer for for the ViewModel LiveData.

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

    viewModel = ViewModelProvider( owner: this).get(MainViewModel::class.java)

    binding.viewModel = viewModel
    binding.sortController = viewModel.sortController

    viewModel.getPopularMovies().observe(viewLifecycleOwner,
        Observer<List<TheMovieDbMovie>> { movieList :List<TheMovieDbMovie>! ->
            activity?.let { it: FragmentActivity
                adapter = MovieAdapter(it, movieList)
                recyclerView?.setAdapter(adapter) ^let
            }
        })
}
```

References

- **MVVM**
  https://developer.android.com/topic/libraries/architecture/viewmodel
- **Data Binding**
  https://www.vogella.com/tutorials/AndroidDatabinding/article.html
- **Live Data**
  https://medium.com/@elye.project/understanding-live-data-made-simple-a820fcd7b4d0