Android Development with Kotlin

ViewModel and Livedata

Separation of Concern - A Programming Principle:

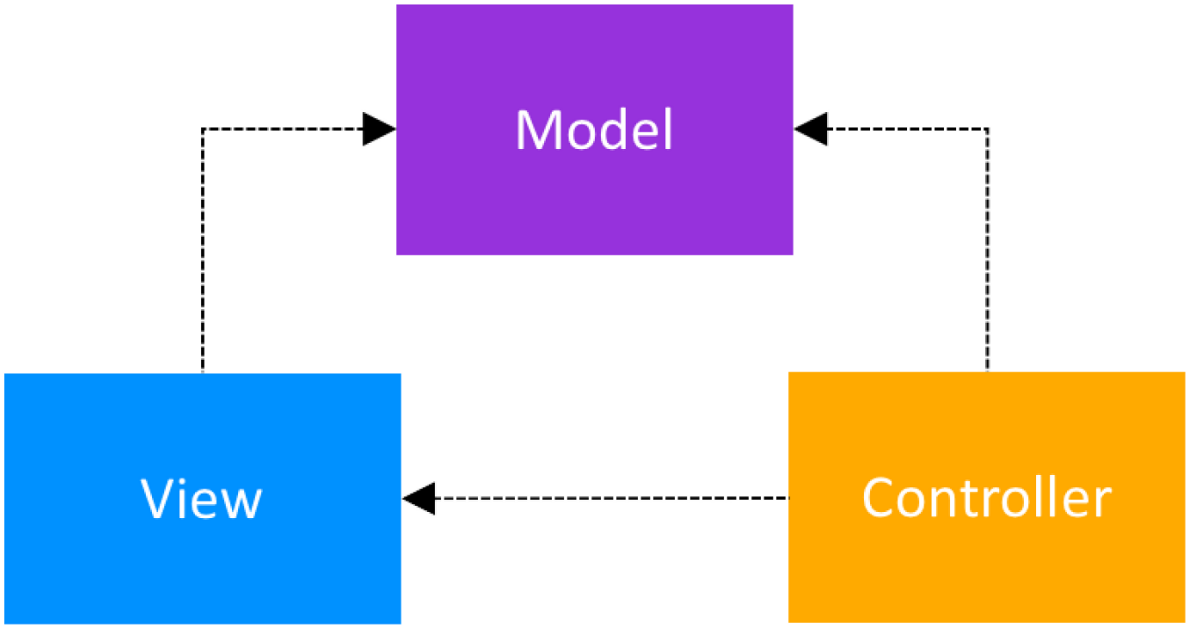
A key principle of software development and architecture is the notion of separation of concerns. At a low level, this principle is closely related to the **Single Responsibility Principle (feel free to explore more)** of object-oriented programming. **The general idea is that one should avoid co-locating different concerns within the design or code.** For instance, if your application includes **business logic** for identifying certain noteworthy items to display to the user, and your application formats such items in a certain way to make them more noticeable, **it would violate separation of concerns** if both the logic for determining which items were noteworthy and the formatting of these items were in the same place. The design would be more maintainable, less tightly coupled, and less likely to violate the Don't Repeat Yourself principle if the logic for determining which items needed formatted were located in a single location (with other business logic), and were exposed to the user interface code responsible for formatting simply as a property.

Various Architecture Patterns:

1) MVC (Model View Controller):

In a world where the **user interface logic** tends to change more often than the business logic, the desktop and Web developers needed a way of separating user interface functionality. The MVC pattern was their solution.

Model: The data layer, responsible for managing the business logic and handling network or database API.
View: The UI layer — a visualization of the data from the Model.
Controller: The logic layer, gets notified of the user's behavior and updates the Model as needed.



So, we will be understanding the MVC architecture principle using a Tic Tac Toe app example and will explore all the principles using this example.

Let's examine the controller in more detail:

```
public class TicTacToeActivity extends AppCompatActivity {

    private Board model;

    /* View Components referenced by the controller */

    private ViewGroup buttonGrid;

    private View winnerPlayerViewGroup;

    private TextView winnerPlayerLabel;

    /**

     * In onCreate of the Activity we lookup & retain references to view components

     * and instantiate the model.

     */

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.tictactoe);
```

```
    winnerPlayerLabel = (TextView) findViewById(R.id.winnerPlayerLabel);

    winnerPlayerViewGroup = findViewById(R.id.winnerPlayerViewGroup);

    buttonGrid = (ViewGroup) findViewById(R.id.buttonGrid);

    model = new Board();

}

/**

 * Here we inflate and attach our reset button in the menu.

 */

@Override

public boolean onCreateOptionsMenu(Menu menu) {

    MenuInflater inflater = getMenuInflater();

    inflater.inflate(R.menu.menu_tictactoe, menu);

    return true;

}

/**

 *  We tie the reset() action to the reset tap event.

 */

@Override

public boolean onOptionsItemSelected(MenuItem item) {

    switch (item.getItemId()) {

        case R.id.action_reset:
```

```
            reset();

            return true;

        default:

            return super.onOptionsItemSelected(item);

    }

}

/**

 *  When the view tells us a cell is clicked in the tic tac toe board,

 *  this method will fire. We update the model and then interrogate it's state

 *  to decide how to proceed.  If X or O won with this move, update the view

 *  to display this and otherwise mark the cell that was clicked.

 */

public void onCellClicked(View v) {



    Button button = (Button) v;

    int row = Integer.valueOf(tag.substring(0,1));

    int col = Integer.valueOf(tag.substring(1,2));

    Player playerThatMoved = model.mark(row, col);

    if(playerThatMoved != null) {

        button.setText(playerThatMoved.toString());

        if (model.getWinner() != null) {
```

```
                winnerPlayerLabel.setText(playerThatMoved.toString());

                winnerPlayerViewGroup.setVisibility(View.VISIBLE);

            }

        }

    }

    /**

     * On reset, we clear the winner label and hide it, then clear out each button.

     * We also tell the model to reset (restart) it's state.

     */

    private void reset() {

        winnerPlayerViewGroup.setVisibility(View.GONE);

        winnerPlayerLabel.setText("");

        model.restart();

        for( int i = 0; i < buttonGrid.getChildCount(); i++ ) {

            ((Button) buttonGrid.getChildAt(i)).setText("");

        }

    }

}
```

2) **MVP (Model View Presenter):**

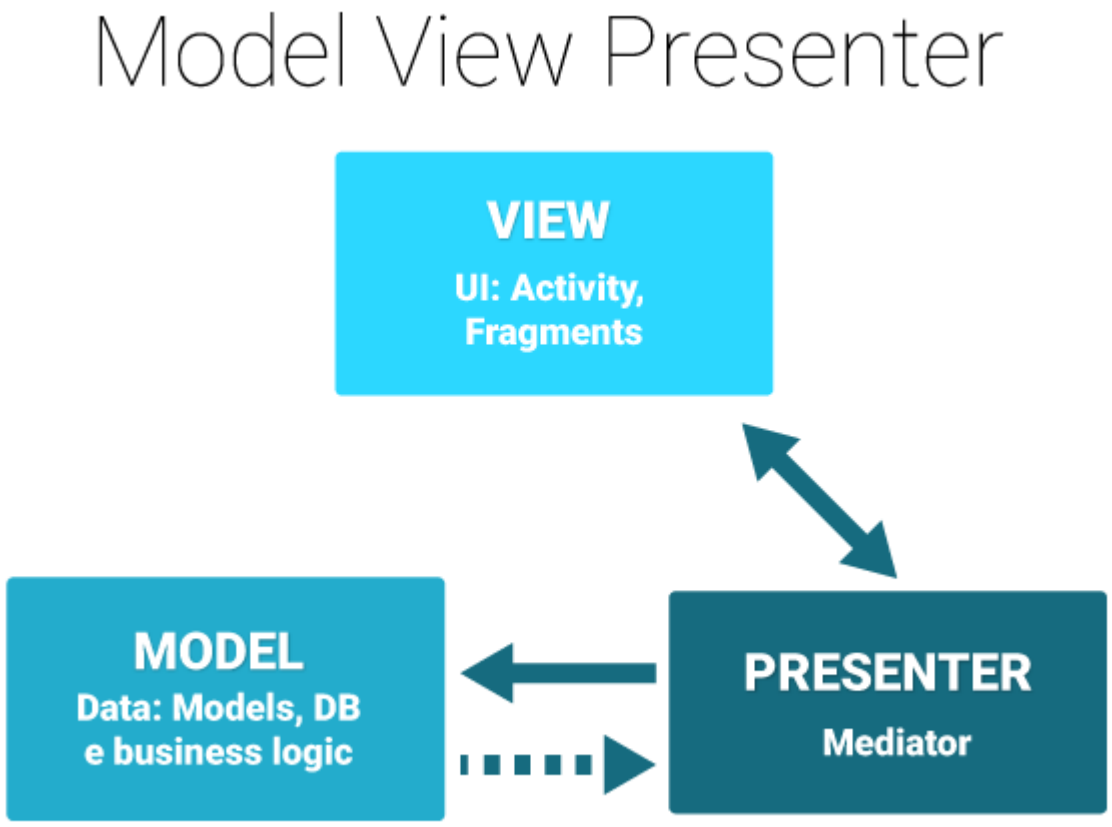


Image Credit: Tin Megali

View : A passive interface that displays data and routes user actions to the Presenter. In Android, it is represented by Activity, Fragment, or View.

Model : a layer that holds business logic controls how data is created, stored, and modified. In Android, it is a data access layer such as database API or remote server API.

Presenter : A middle man which retrieves data from Model and shows it in the View. It also processes user action forwarded to it by the View.

Important points of **MVP** are:

- View can not access Model.
- Presenter is tied to a single View. (One-to-One relationship)
- View is completely **dumb and passive** (only retrieve user action and leave all other things for Presenter to handle).

So let's continue with our tic tac toe example from above:


```
public class TicTacToePresenter implements Presenter {

    private TicTacToeView view;

    private Board model;

    public TicTacToePresenter(TicTacToeView view) {

        this.view = view;

        this.model = new Board();

    }

    // Here we implement delegate methods for the standard Android Activity Lifecycle.

    // These methods are defined in the Presenter interface that we are implementing.

    public void onCreate() { model = new Board(); }

    public void onPause() { }

    public void onResume() { }

    public void onDestroy() { }

    /**

     * When the user selects a cell, our presenter only hears about

     * what was (row, col) pressed, it's up to the view now to determine that from

     * the Button that was pressed.
```

```
     */

    public void onButtonSelected(int row, int col) {

        Player playerThatMoved = model.mark(row, col);

        if(playerThatMoved != null) {

            view.setButtonText(row, col, playerThatMoved.toString());



            if (model.getWinner() != null) {

                view.showWinner(playerThatMoved.toString());

            }

        }

    }

    /**

     *  When we need to reset, we just dictate what to do.

     */

      public void onResetSelected() {

        view.clearWinnerDisplay();

        view.clearButtons();

        model.restart();

}

}
```

To make this work without tying the activity to the presenter we create an interface that the Activity implements. In a test, we'll create a mock based on this interface to test interactions with the view from the presenter.

```
public interface TicTacToeView {

    void showWinner(String winningPlayerDisplayLabel);

    void clearWinnerDisplay();

    void clearButtons();

    void setButtonText(int row, int col, String text);

}
```

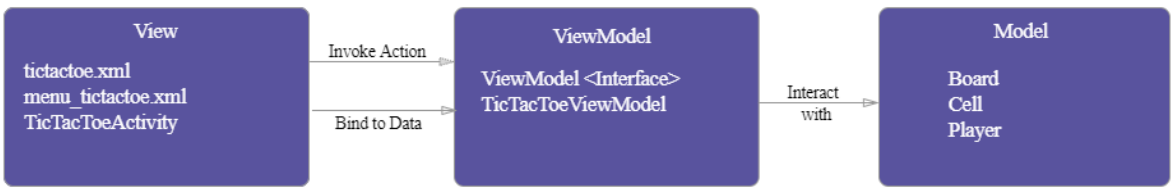
3) MVVM (Model View View Model):

The main players in the MVVM pattern are:

- The View — that informs the ViewModel about the user's actions
- The ViewModel — exposes streams of data relevant to the View
- The DataModel — abstracts the data source. The ViewModel works with the DataModel to get and save the data.

Image src: https://miro.medium.com/max/1400/0*5mD214cjNXU-V6lf.png

So lets continue with our tic tac toe example from above:



```
public class TicTacToeViewModel implements ViewModel {

    private Board model;

    /*

     * These are observable variables that the viewModel will update as appropriate

     * The view components are bound directly to these objects and react to changes

     * immediately, without the ViewModel needing to tell it to do so. They don't

     * have to be public, they could be private with a public getter method too.

     */

    public final ObservableArrayMap<String, String> cells = new ObservableArrayMap<>();

    public final ObservableField<String> winner = new ObservableField<>();

    public TicTacToeViewModel() {

        model = new Board();

    }

    // As with presenter, we implement standard lifecycle methods from the view

    // in case we need to do anything with our model during those events.

    public void onCreate() { }

    public void onPause() { }

    public void onResume() { }

    public void onDestroy() { }
```

```
    /**
     * An Action, callable by the view.  This action will pass a message to the model
     * for the cell clicked and then update the observable fields with the current
     * model state.
     */
    public void onClickedCellAt(int row, int col) {
        Player playerThatMoved = model.mark(row, col);
        cells.put("" + row + col, playerThatMoved == null ?
                                                  null : playerThatMoved.toString());
        winner.set(model.getWinner() == null ? null : model.getWinner().toString());
    } /**
     * An Action, callable by the view.  This action will pass a message to the model
     * to restart and then clear the observable data in this ViewModel.
     */
    public void onResetSelected() {
        model.restart();
        winner.set(null);
        cells.clear();
    }
}
```

A couple snippets from the view to see how these variables and actions are bound.

```
<!--

    With Data Binding, the root element is <layout>.  It contains 2 things.

    1. <data> - We define variables to which we wish to use in our binding expressions and

                import any other classes we may need for reference, like android.view.View.

    2. <root layout> - This is the visual root layout of our view.  This is the root xml tag
in the MVC and MVP view examples.

-->

<layout xmlns:android="http://schemas.android.com/apk/res/android"

    xmlns:tools="http://schemas.android.com/tools"

    xmlns:app="http://schemas.android.com/apk/res-auto">



    <!-- We will reference the TicTacToeViewModel by the name viewModel as we have defined it
here. -->

    <data>

        <import type="android.view.View" />

        <variable name="viewModel" type="com.acme.tictactoe.viewmodel.TicTacToeViewModel" />

    </data>

    <LinearLayout...>

        <GridLayout...>

            <!-- onClick of any cell in the board, the button clicked will invoke the
onClickedCellAt method with its row,col -->
```

```
            <!-- The display value comes from the ObservableArrayMap defined in the ViewModel
-->

            <Button

                style="@style/tictactoebutton"

                android:onClick="@{() -> viewModel.onClickedCellAt(0,0)}"

                android:text='@{viewModel.cells["00"]}' />

            ...

            <Button

                style="@style/tictactoebutton"

                android:onClick="@{() -> viewModel.onClickedCellAt(2,2)}"

                android:text='@{viewModel.cells["22"]}' />

        </GridLayout>



        <!-- The visibility of the winner view group is based on whether or not the winner
value is null.

             Caution should be used not to add presentation logic into the view.  However, for
this case

             it makes sense to just set visibility accordingly.  It would be odd for the view
to render

             this section if the value for winner were empty.  -->

        <LinearLayout...

            android:visibility="@{viewModel.winner != null ? View.VISIBLE : View.GONE}"

            tools:visibility="visible">
```

```
            <!-- The value of the winner label is bound to the viewModel.winner and reacts if
that value changes -->

            <TextView

                ...

                android:text="@{viewModel.winner}"

                tools:text="X" />

            ...

        </LinearLayout>

    </LinearLayout>

</layout>
```

So now as you have studied all architecture patterns in detail, its time to explore the differences between the three of them. **Visit these links for reading the difference between MVC,MVP,MVVM:**
https://stackoverflow.com/questions/19444431/what-is-difference-between-mvc-mvp-mvvm-design-pattern-in-terms-of-coding-c-s

https://www.oreilly.com/library/view/learning-javascript-design/9781449334840/ch10s09.html#:~:text=MVC%20Versus%20MVP%20Versus%20MVVM,they%20are%20to%20each%20other.&text=In%20MVP%2C%20the%20role%20of,is%20replaced%20with%20a%20Presenter

ViewModel in Android:

The ViewModel class is designed to store and manage UI-related data in a lifecycle conscious way. The ViewModel class allows data to survive configuration changes such as screen rotations.

The lifecycle of a ViewModel :

ViewModel objects are scoped to the Lifecycle passed to the ViewModelProvider when getting the ViewModel. The ViewModel remains in memory until the Lifecycle it's scoped to

goes away permanently: in the case of an activity, when it finishes, while in the case of a fragment, when it's detached.

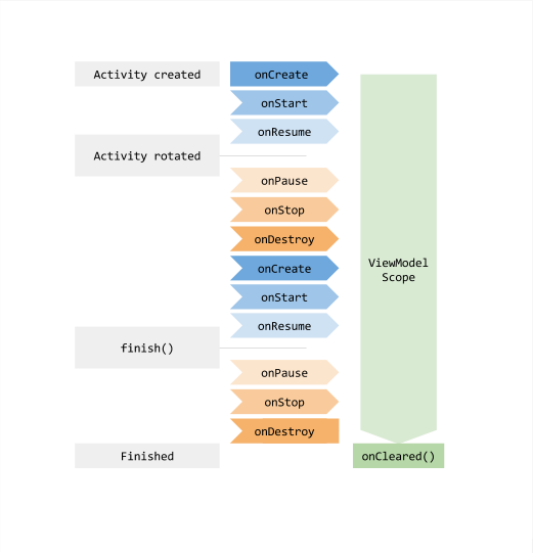


Image src: https://developer.android.com/images/topic/libraries/architecture/viewmodel-lifecycle.png

You usually request a ViewModel the first time the system calls an activity object's onCreate() method. The system may call onCreate() several times throughout the life of an activity, such as when a device screen is rotated. The ViewModel exists from when you first request a ViewModel until the activity is finished and destroyed.

Livedata in Android:

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. **LiveData considers an observer, which is represented by the Observer class, to be in an active state if its lifecycle is in the STARTED or RESUMED state. LiveData only notifies active observers about updates. Inactive observers registered to watch LiveData objects aren't notified about changes.**

You can register an observer paired with an object that implements the Lifecycle Owner interface. This relationship allows the observer to be removed when the state of the corresponding Lifecycle object changes to DESTROYED. This is especially useful for activities and fragments because they can safely observe LiveData objects and not worry about leaks—activities and fragments are instantly unsubscribed when their lifecycles are destroyed.

Advantages:

Ensures your UI matches your data state:

LiveData follows the observer pattern. LiveData notifies Observer objects when the lifecycle state changes. You can consolidate your code to update the UI in these Observer objects. Instead of updating the UI every time the app data changes, your observer can update the UI every time there's a change.

No memory leaks:
Observers are bound to Lifecycle objects and clean up after themselves when their associated lifecycle is destroyed.

No crashes due to stopped activities:
If the observer's lifecycle is inactive, such as in the case of an activity in the back stack, then it doesn't receive any LiveData events.

No more manual lifecycle handling:
UI components just observe relevant data and don't stop or resume observation. LiveData automatically manages all of this since it's aware of the relevant lifecycle status changes while observing.

Always up to date data:
If a lifecycle becomes inactive, it receives the latest data upon becoming active again. For example, an activity that was in the background receives the latest data right after it returns 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 just watch the LiveData object.

To explore and read more about these topics use these links :

Livedata: https://developer.android.com/topic/libraries/architecture/livedata

ViewModel: https://developer.android.com/reference/android/arch/lifecycle/ViewModel

Viewmodel and livedata example: https://medium.com/@taman.neupane/basic-example-of-livedata-and-viewmodel-14d5af922d0