MVC (Model View Controller) Architecture

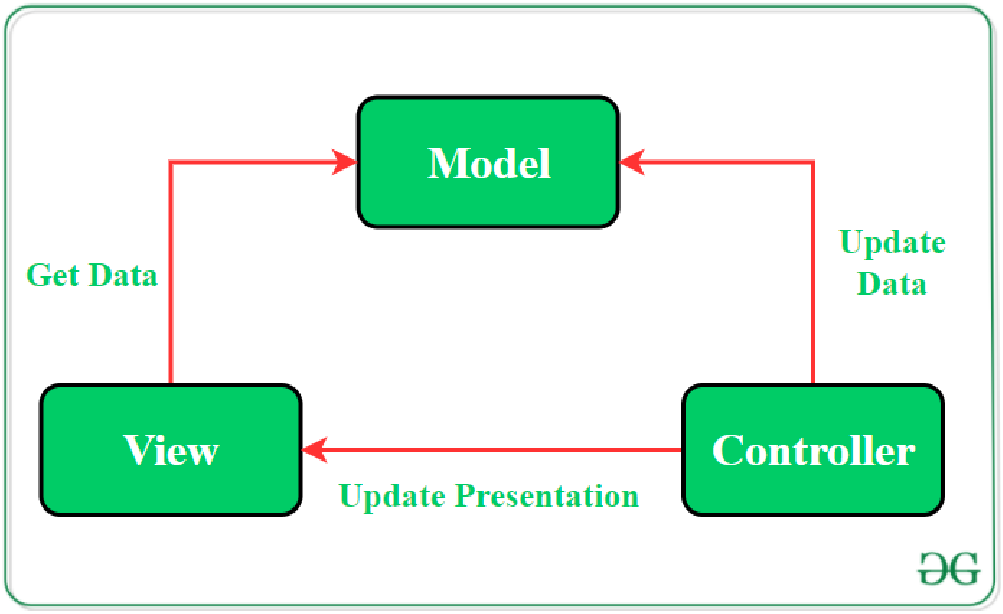
Developing an android application by applying a software architecture pattern is always preferred by the developers. An architecture pattern gives modularity to the project files and assures that all the codes get covered in Unit testing. It makes the task easy for developers to maintain the software and to expand the features of the application in the future. There are some architectures that are very popular among developers and one of them is the **Model—View—Controller(MVC)**

Pattern. The **MVC pattern** suggests splitting the code into 3 components. While creating the class/file of the application, the developer must categorize it into one of the following three layers:

- **Model:** This component stores the application data. It has no knowledge about the interface. The model is responsible for handling the domain logic(real-world business rules) and communication with the database and network layers.

- **View:** It is the UI(User Interface) layer that holds components that are visible on the screen. Moreover, it provides the visualization of the data stored in the Model and offers interaction to the user.

- **Controller:** This component establishes the relationship between the **View** and the **Model.** It contains the core application logic and gets informed of the user's behavior and updates the Model as per the need.



In spite of applying MVC schema to give a modular design to the application, code layers do depend on each other. In this pattern, **View** and **Controller** both depend upon the **Model.** Multiple approaches are possible to apply the MVC pattern in the project:

- **Approach 1:** Activities and fragments can perform the role of Controller and are responsible for updating the View.

- **Approach 2:** Use activity or fragments as views and controller while Model will be a separate class that does not extend any Android class.

In **MVC architecture**, application data is updated by the controller and View gets the data. Since the **Model** component is separated, it could be tested independently of the UI. Further, if the **View** layer respects the **single responsibility principle** then their role is just to update the Controller for every user event and just display data from the Model, without implementing any business logic. In this case, UI tests should be enough to cover the functionalities of the View.

Example of MVC Architecture

To understand the implementation of the MVC architecture pattern more clearly, here is a simple example of an android application. This application will have 3 buttons and each one of them displays the count that how many times the user has clicked that particular button. To develop this application the code has been separated in the following manner:

- **Controller and View** will be handled by the Activity. Whenever the user clicks the buttons, activity directs the Model to handle the further operations. The activity will act as an **observer**.

- The **Model** will be a separate class that contains the data to be displayed. The operations on the data will be performed by functions of this class and after updating the values of the data this **Observable class** notifies the **Observer(Activity)** about the change.

Below is the complete step-by-step implementation of this android application using the MVC architecture pattern:

Note: *Following steps are performed on Android Studio*

Step 1: Create a new project

1. Click on File, then New => New Project.
2. Choose Empty activity

3. Select language as Java/Kotlin
4. Select the minimum SDK as per your need.

Step 2: Modify String.xml file

All the strings which are used in the activity are listed in this file.

```xml
<resources>
    <string name="app_name">GfG | MVC Architecture</string>
    <string name="Heading">MVC Architecture Pattern</string>
    <string name="Text1">Button_1</string>
    <string name="count">Count:0</string>
</resources>
```

Step 3: Working with the activity_main.xml file

Open the activity_main.xml file and add 3 Buttons to it which will display the count values as per the number of times the user clicks it. Below is the code for designing a proper activity layout.

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#168BC34A"
    tools:context=".MainActivity" >

    <!-- Provided Linear layout for the activity. -->
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent">

        <!-- TextView to display heading of the activity. -->
        <TextView
            android:id="@+id/textView"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginTop="40dp"
            android:layout_marginBottom="60dp"
            android:fontFamily="@font/roboto"
            android:text="@string/Heading"
            android:textAlignment="center"
            android:textColor="@android:color/holo_green_dark"
            android:textSize="30sp"
            android:textStyle="bold" />

        <!-- First Button of the activity. -->
        <Button
            android:id="@+id/button"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginStart="20dp"
            android:layout_marginTop="30dp"
            android:layout_marginEnd="20dp"
            android:layout_marginBottom="20dp"
            android:background="#4CAF50"
            android:fontFamily="@font/roboto"
            android:text="@string/count"
            android:textColor="@android:color/background_light"
            android:textSize="24sp"
            android:textStyle="bold" />

        <!-- Second Button of the activity. -->
        <Button
            android:id="@+id/button2"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginStart="20dp"
            android:layout_marginTop="50dp"
            android:layout_marginEnd="20dp"
            android:layout_marginBottom="20dp"
            android:background="#4CAF50"
            android:fontFamily="@font/roboto"
            android:text="@string/count"
            android:textColor="@android:color/background_light"
            android:textSize="24sp"
            android:textStyle="bold" />

        <!-- Third Button of the activity. -->
        <Button
            android:id="@+id/button3"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginStart="20dp"
            android:layout_marginTop="50dp"
            android:layout_marginEnd="20dp"
            android:layout_marginBottom="20dp"
            android:background="#4CAF50"
            android:fontFamily="@font/roboto"
            android:text="@string/count"
            android:textColor="@android:color/background_light"
            android:textSize="24sp"
            android:textStyle="bold" />

        <ImageView
            android:id="@+id/imageView"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginTop="30dp"
            app:srcCompat="@drawable/banner" />

    </LinearLayout>
</androidx.constraintlayout.widget.ConstraintLayout>
```

Step 4: Creating the Model class

Create a new class named Model to separate all data and its operations. This class will not know the existence of View Class.

JavaKotlin

```
import java.util.*;

public class Model extends Observable {

    // declaring a list of integer
    private List<Integer> List;

    // constructor to initialize the list
    public Model(){
        // reserving the space for list elements
        List = new ArrayList<Integer>(3);

        // adding elements into the list
        List.add(0);
        List.add(0);
        List.add(0);
    }

    // defining getter and setter functions

    // function to return appropriate count
    // value at correct index
    public      int      getValueAtIndex(final      int      the_index)      throws
IndexOutOfBoundsException{
        return List.get(the_index);
    }

    // function to make changes in the activity button's
    // count value when user touch it
    public      void      setValueAtIndex(final      int      the_index)      throws
IndexOutOfBoundsException{
       List.set(the_index,List.get(the_index) + 1);
       setChanged();
       notifyObservers();
    }

}
```

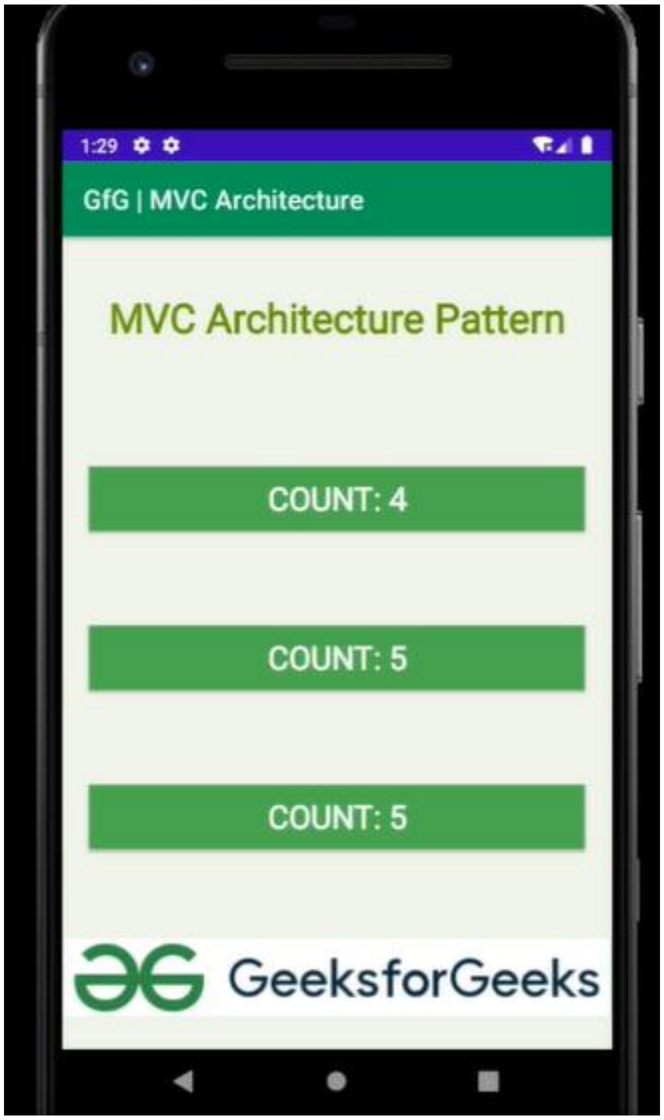
Step 5: Define functionalities of View and Controller in the MainActivity file

This class will establish the relationship between View and Model. The data provided by the Model will be used by View and the appropriate changes will be made in the activity.

JavaKotlin

```
import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import java.util.Observable;
import java.util.Observer;

public   class   MainActivity   extends   AppCompatActivity   implements   Observer,
View.OnClickListener {

    // creating object of Model class
    private Model myModel;

    // creating object of Button class
    private Button Button1;
    private Button Button2;
    private Button Button3;


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // creating relationship between the
        // observable Model and the
        // observer Activity
        myModel = new Model();
        myModel.addObserver(this);

        // assigning button IDs to the objects
        Button1 = findViewById(R.id.button);
        Button2 = findViewById(R.id.button2);
        Button3 = findViewById(R.id.button3);


        // transfer the control to Onclick() method
        // when a button is clicked by passing
        // argument "this"
        Button1.setOnClickListener(this);
        Button2.setOnClickListener(this);
        Button3.setOnClickListener(this);

    }

    @Override
     // calling setValueAtIndex() method
     // by passing appropriate arguments
     // for different buttons
    public void onClick(View v) {
        switch(v.getId()){

            case R.id.button:
                myModel.setValueAtIndex(0);
                break;

            case R.id.button2:
                myModel.setValueAtIndex(1);
                break;

            case R.id.button3:
                myModel.setValueAtIndex(2);
                break;
        }
    }

    @Override
     // function to update the view after
     // the values are modified by the model
    public void update(Observable arg0, Object arg1) {


        // changing text of the buttons
        // according to updated values
        Button1.setText("Count: "+myModel.getValueAtIndex(0));
        Button2.setText("Count: "+myModel.getValueAtIndex(1));
        Button3.setText("Count: "+myModel.getValueAtIndex(2));

    }
}
```

Output



Advantages of MVC architecture pattern

- MVC pattern increases the code testability and makes it easier to implement new features as it highly supports the separation of concerns.

- Unit testing of Model and Controller is possible as they do not extend or use any Android class.
- Functionalities of the View can be checked through UI tests if the View respect the single responsibility principle(update controller and display data from the model without implementing domain logic)

Disadvantages of MVC architecture pattern

- Code layers depend on each other even if MVC is applied correctly.
- No parameter to handle UI logic i.e., how to display the data.