Android Architecture

This tutorial contains notes about architecture for Android applications which improves testability.

1. Architectures for Android

The Android default templates encourages the creation of large activities or fragments. These components typically contain both business and UI logic. This makes testing and therefore the maintenance of Android applications harder.

Several patterns are popular within the Android community to improve testability.

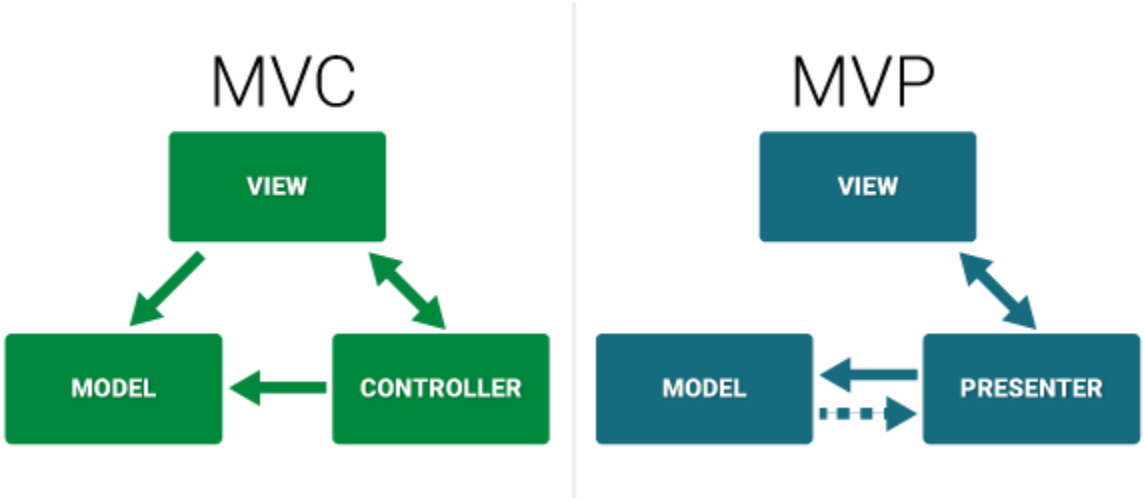
The most popular architecture choices are:

- Model View Presenter (MVP)
- Model View View Model (MVVM) together with Android Data Binding

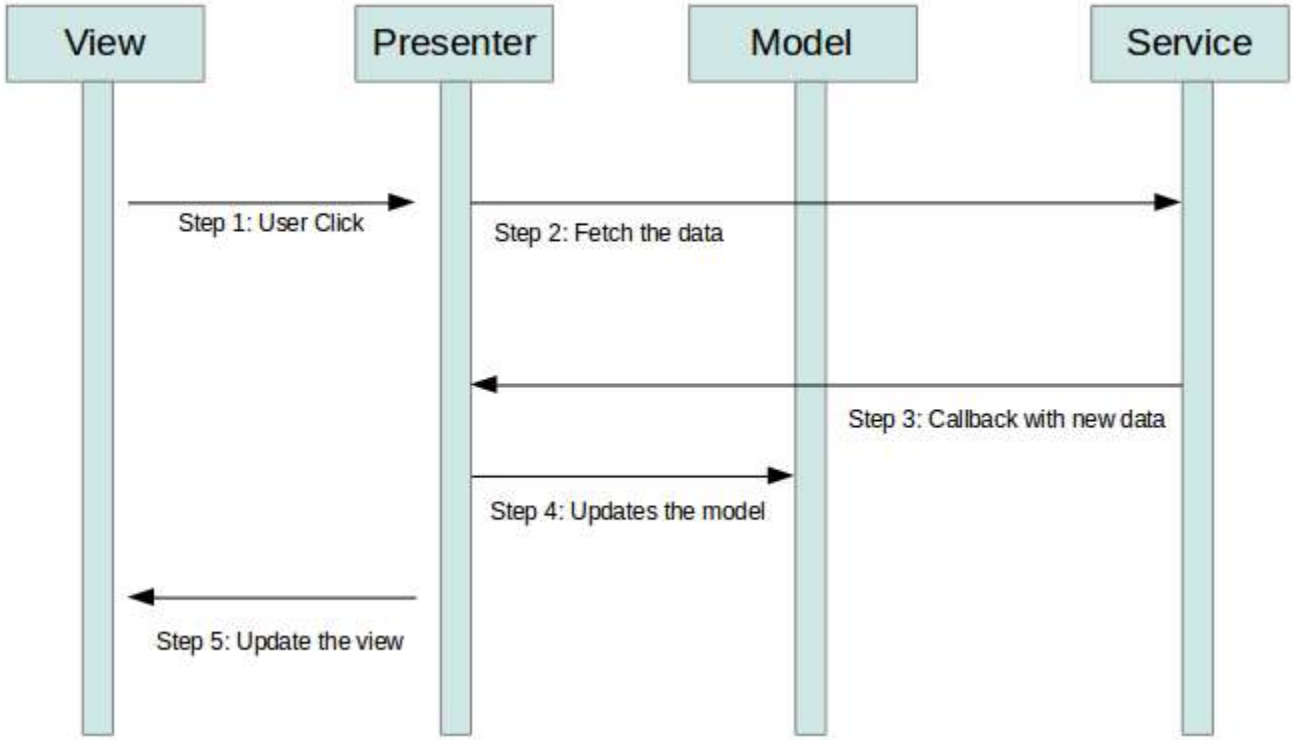
> The view in MVP or MVVM is not the same as the `View` class in Android. A view in MVP it usually implemented via a fragment, activity or a dialog.

2. The Model View Presenter architecture for Android

The Model View Presenter (MVP) architecture pattern improve the application architecture to increase testability. The MVP pattern separates the *data model*, from a *view* through a *presenter*.



The following demonstrates an example data flow throw the MVP.



2.1. The view

A view component in MVP contains a visual part of the application.

It contains only the UI and it does not contain any logic or knowledge of the data displayed. In typical implementations the view components in MVP exports an interface that is used by the Presenter. The presenter uses these interface methods to manipulate the view. Example method names would be: `showProgressBar`, `updateData`.

2.2. The presenter

The presenter triggers the business logic and tells the view when to update. It therefore interacts with the model and fetches and transforms data from the model to update the view. The presenter should not have, if possible, a dependency to the Android SDK.

2.3. The model

Contains a data provider and the code to fetch and update the data. This part of MVP updates the database or communicate with a webserver.

```
public class MainActivityPresenter {
                                                    private User user;
                                                    private View view;
                                                    public MainActivityPresenter(View view) {
                                                        this.user = new User();
                                                        this.view = view;
                                                    }
                                                    public void updateFullName(String fullName){
                                                        user.setFullName(fullName);

                                                view.updateUserInfoTextView(user.toString());
                                                    }
                                                    public void updateEmail(String email){
                                                        user.setEmail(email);

                                                view.updateUserInfoTextView(user.toString());
                                                    }
                                                    public interface View{
                                                        void updateUserInfoTextView(String info);
                                                        void showProgressBar();
                                                        void hideProgressBar();
                                                    }
                                                }
public
class User
{
                private String fullName = "", email = "";
                public User() {
                }
                public User(String fullName, String email) {
                    this.fullName = fullName;
                    this.email = email;
                }
                public String getFullName() {
                    return fullName;
                }
                public void setFullName(String fullName) {
                    this.fullName = fullName;
                }
                public String getEmail() {
```

```
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    @Override
    public String toString() {
        return "Email : " + email + "\nFullName : " +
fullName;
    }
}
public class
MainActivity extends
AppCompatActivity
implements
MainActivityPresenter.
View {
                    private MainActivityPresenter presenter;
                    private TextView myTextView;
                    private ProgressBar progressBar;
                    @Override
                    protected void onCreate(Bundle savedInstanceState) {
                        super.onCreate(savedInstanceState);
                        setContentView(R.layout.activity_main);
                        Toolbar toolbar = findViewById(R.id.toolbar);
                        setSupportActionBar(toolbar);
                        presenter = new MainActivityPresenter(this);
                        myTextView = findViewById(R.id.myTextView);
                        EditText userName = findViewById(R.id.username);
                        EditText email = findViewById(R.id.email);
                        initProgressBar();
                        userName.addTextChangedListener(new TextWatcher() {
                            @Override
                            public void beforeTextChanged(CharSequence s, int start,
                    int count, int after) {
                            }
                            @Override
                            public void onTextChanged(CharSequence s, int start, int
                    before, int count) {
                                presenter.updateFullName(s.toString());
                            }
                            @Override
                            public void afterTextChanged(Editable s) {
```

```
                hideProgressBar();
            }
        });
        email.addTextChangedListener(new TextWatcher() {
            @Override
            public void beforeTextChanged(CharSequence s, int start,
int count, int after) {
            }
            @Override
            public void onTextChanged(CharSequence s, int start, int
before, int count) {
                presenter.updateEmail(s.toString());
            }
            @Override
            public void afterTextChanged(Editable s) {
                hideProgressBar();
            }
        });
    }
    private void initProgressBar() {
        progressBar = new ProgressBar(this, null,
android.R.attr.progressBarStyleSmall);
        progressBar.setIndeterminate(true);
        RelativeLayout.LayoutParams params = new
RelativeLayout.LayoutParams(Resources.getSystem().getDisplayMetrics().w
idthPixels,
                250);
        params.addRule(RelativeLayout.CENTER_IN_PARENT);
        this.addContentView(progressBar, params);
        showProgressBar();
    }
    @Override
    public void updateUserInfoTextView(String info) {
        myTextView.setText(info);
    }
    @Override
    public void showProgressBar() {
        progressBar.setVisibility(View.VISIBLE);
    }
    @Override
    public void hideProgressBar() {
        progressBar.setVisibility(View.INVISIBLE);
    }
```

```
}
```

2.4. Considerations for using the MVP design pattern

MVP makes it easier to test your presenter logic and to replace dependencies. But using MVP also comes with a costs, it makes your application code longer. Also as the standard Android templates at the moment do not use this approach, not every Android developer will find this code structure easy to understand.

2.5. Comparison to Model View Controller

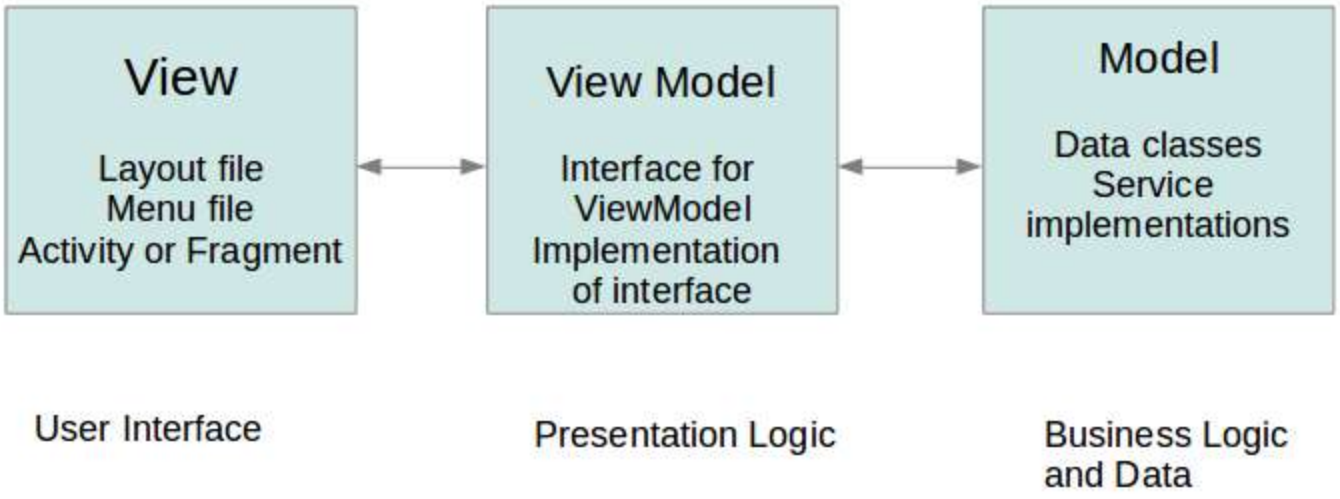
In the *Model View Presenter* pattern, the views more separated from the model. The presenter communicates between model and view. This makes it easier to create unit tests Generally there is a one to one mapping between view and Presenter, but it is also possible to use multiple presenters for complex views.

In the *Model View Controller* pattern the controllers are behavior based and can share multiple views. View can communicate directly with the model.

MVP is currently on of the patterns that the Android community prefers.

3. The Model View View Model architecture for Android

The Model View View Model design pattern is also known as Model View Binder.



3.1. The view

A view component in MVP contains a visual part of the application.

The view binds to observable variables and actions exposed by the view model typically using the data binding framework.

The view is responsible for handling for example:

- Menus
- Permissions
- Event listeners
- Showing dialogs, Toasts, Snackbars
- Working with Android `View` and `Widget`
- Starting activities
- All functionality which is related to the Android `Context`

3.2. The view model

The view model contains the data required for the view. It is an abstraction of the view and exposes public properties and commands. It uses observable data to notify the view about changes. It also allows to pass events to the model. It is also a value converter from the raw model data to presentation-friendly properties)

The view model has the following responsibilities:

- Exposing data

- Exposing state (progress, offline, empty, error, etc)
- Handling visibility
- Input validation
- Executing calls to the model
- Executing methods in the view

The view model should only know about the application context. the application context can:

- Start a service
- Bind to a service
- Send a broadcast
- Register a broadcast receiver
- Load resource values

It cannot:

- Show a dialog
- Start an activity
- Inflate a layout

3.3. The model

Contains a data provider and the code to fetch and update the data. The data can be retrieved from different sources, for example:

- REST API
- Realm db
- SQLite db
- Handles broadcast
- Shared Preferences
- Firebase
- etc.

Basically the same as the model in the MVP.

3.4. Differences to MVP

MVVM uses data binding and is therefore a more event driven architecture. MVP typically has a one to one mapping between the presenter and the view, while MVVM can map many views to one view model In MVVM the view model has no reference to the view, while in MVP the view knows the presenter.