deploy is back! Join DigitalOcean's virtual conference for global builders. Register today →

We're hiring Blog Docs Get Support Sales

DigitalOcean

Tutorials Questions Tech Talks Learning Paths Product Docs Social Impact

CONTENTS

RELATED

Android External Storage - Read, Write, Save File

View

Exception Handling in Java

View

// Tutorial //

Android MVVM Design Pattern

COOKIE PREFERENCES

Published on August 3, 2022

Android Design Patterns Java

By Anupam Chugh
Developer and author at DigitalOcean.

COOKIE PREFERENCES

In this tutorial, we'll be discussing and implementing the Android MVVM Architectural Pattern in our Android Application. We've previously discussed the Android MVP Pattern.

Why do we need these patterns? Adding everything in a Single Activity or Fragment would lead to problems in testing and refactoring the code. Hence, the use of separation of code and clean architecture is recommended.

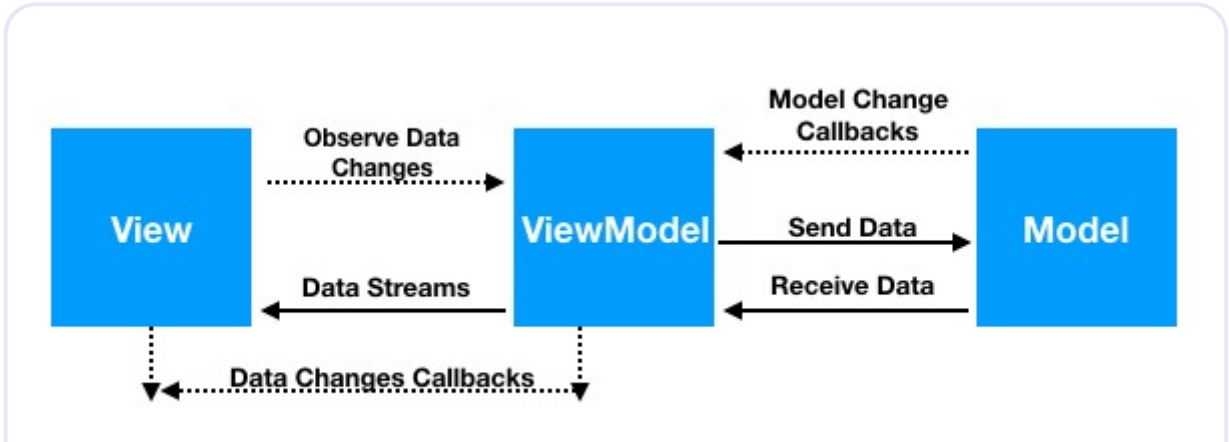
While we believe that this content benefits our community, we have not yet thoroughly reviewed it. If you have any suggestions for improvements, please let us know by clicking the "report an issue" button at the bottom of the tutorial.

Android MVVM

MVVM stands for **Model**, **View**, **ViewModel**.

- **Model**: This holds the data of the application. It cannot directly talk to the View. Generally, it's recommended to expose the data to the ViewModel through Observables.
- **View**: It represents the UI of the application devoid of any Application Logic. It observes the ViewModel.
- **ViewModel**: It acts as a link between the Model and the View. It's responsible for transforming the data from the Model. It provides data streams to the View. It also uses hooks or callbacks to update the View. It'll ask for the data from the Model.

The following flow illustrates the core MVVM Pattern.



COOKIE PREFERENCES

How does this differ from MVP?

- ViewModel replaces the Presenter in the Middle Layer.
- The Presenter holds references to the View. The ViewModel doesn't.
- The Presenter updates the View using the classical way (triggering methods).
- The ViewModel sends data streams.
- The Presenter and View are in a 1 to 1 relationship.
- The View and the ViewModel are in a 1 to many relationship.
- The ViewModel does not know that the View is listening to it.

There are two ways to implement MVVM in Android:

- Data Binding
- RXJava

In this tutorial, we'll be using Data Binding only. Data Binding Library was introduced by Google in order to bind data directly in the xml layout. For more info on Data Binding, refer this tutorial. We'll be creating a simple Login Page Example Application that asks for user inputs. We'll see how the ViewModel notifies the View when to show a Toast Message without keeping a reference of the View.

How is it possible to notify some class without having a reference of it? It can be done in three different ways:

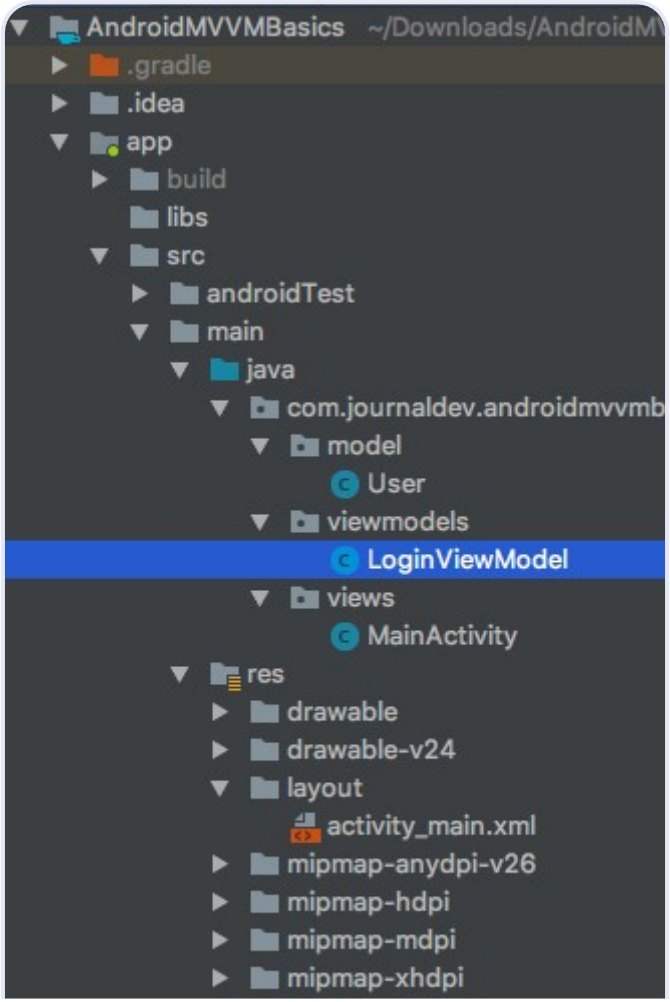
- Using Two Way Data Binding
- Using Live Data
- Using RxJava

Two Way Data Binding

COOKIE PREFERENCES

Two-way Data Binding is a technique of binding your objects to your XML layouts such that the Object and the layout can both send data to each other. In our case, the ViewModel can send data to the layout and also observe changes. For this, we need a `BindingAdapter` and custom attribute defined in the XML. The Binding Adapter would listen to changes in the attribute property. We'll learn more about Two-way Data Binding through our example below.

Android MVVM Example Project Structure



COOKIE PREFERENCES

Adding the Data Binding Library

Add the following code to your app's build.gradle file:

```
android {

    dataBinding {
        enabled = true
    }
}
```

This enables Data Binding in your Application.

Adding the Dependencies

Add the following dependencies in your `build.gradle` file :

```
implementation 'android.arch.lifecycle:extensions:1.1.0'
```

Model

The Model would hold the user's email and password. The following User.java class does it:

```
package com.journaldev.androidmvvmbasics.model;
```

COOKIE PREFERENCES

```
public class User {
    private String email;
    private String password;

    public User(String email, String password) {
        this.email = email;
        this.password = password;
    }

    public void setEmail(String email) {
        this.email = email;
    }


    public String getEmail() {
        return email;
    }

    public void setPassword(String password) {
        this.password = password;
    }


    public String getPassword() {
        return password;
    }


}
```

Two-way Data Binding allows us to bind objects in the XML layouts such that the object can send data to the layout, and vice versa. The Syntax for two way data binding is `@={variable}`

Layout

COOKIE PREFERENCES

The code for the activity_main.xml is given below:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="https://schemas.android.com/apk/res/android"
    xmlns:bind="https://schemas.android.com/tools">

    <data>

        <variable
            name="viewModel"
            type="com.journaldev.androidmvvmbasics.viewmodels.LoginViewModel" />
    </data>


    <ScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:layout_margin="8dp"
            android:orientation="vertical">

            <EditText
                android:id="@+id/inEmail"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:hint="Email"
                android:inputType="textEmailAddress"
                android:padding="8dp"
                android:text="@={viewModel.userEmail}" />
```

COOKIE PREFERENCES

```
            <EditText
                android:id="@+id/inPassword"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:hint="Password"
                android:inputType="textPassword"
                android:padding="8dp"
                android:text="@={viewModel.userPassword}" />


            <Button
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:layout_marginTop="8dp"
                android:onClick="@{()-> viewModel.onLoginClicked()}"
                android:text="LOGIN"
                bind:toastMessage="@{viewModel.toastMessage}" />


        </LinearLayout>

    </ScrollView>

</layout>
```

Data Binding requires us to set the layout tag at the top. Here our ViewModel binds the data to the View. `()-> viewModel.onLoginClicked()` invokes the Button click listener lambda defined in our ViewModel. The EditText updates the values in the Model (via View Model). `bind:toastMessage="@{viewModel.toastMessage}"` is a custom attribute we've created for two-way data binding. Based on changes in the toastMessage in the ViewModel the BindingAdapter would get triggered in the View.

ViewModel

COOKIE PREFERENCES

The code for the LoginViewModel.java is given below:

```
package com.journaldev.androidmvvmbasics.viewmodels;

import android.databinding.BaseObservable;
import android.databinding.Bindable;
import android.text.TextUtils;
import android.util.Patterns;

import com.android.databinding.library.baseAdapters.BR;
import com.journaldev.androidmvvmbasics.model.User;

public class LoginViewModel extends BaseObservable {
    private User user;


    private String successMessage = "Login was successful";
    private String errorMessage = "Email or Password not valid";

    @Bindable
    private String toastMessage = null;


    public String getToastMessage() {
        return toastMessage;
    }


    private void setToastMessage(String toastMessage) {

        this.toastMessage = toastMessage;
        notifyPropertyChanged(BR.toastMessage);
    }
```

COOKIE PREFERENCES

```
public void setUserEmail(String email) {
    user.setEmail(email);
    notifyPropertyChanged(BR.userEmail);
}

@Bindable
public String getUserEmail() {
    return user.getEmail();
}

@Bindable
public String getUserPassword() {
    return user.getPassword();
}

public void setUserPassword(String password) {
    user.setPassword(password);
    notifyPropertyChanged(BR.userPassword);
}

public LoginViewModel() {
    user = new User("","");
}

public void onLoginClicked() {
    if (isInputDataValid())
        setToastMessage(successMessage);
    else
        setToastMessage(errorMessage);
}

public boolean isInputDataValid() {
    return !TextUtils.isEmpty(getUserEmail()) && Patterns.EMAIL_ADDRESS.matcher(getUserEmail
}
```

COOKIE PREFERENCES

```
}
```

The methods were called in the layout are implemented in the above code with the same signature. If the XML counterpart of the method doesn't exist, we need to change the attribute to `app:` . The above class can also extend ViewModel. But we need BaseObservable since it converts the data into streams and notifies when the `toastMessage` property is changed. We need to define the getter and setter for the toastMessage custom attribute defined in the XML. Inside the setter, we notify the observer (which will be the View in our application) that the data has changed. The View(Our activity) can define the appropriate action.

BR class is auto-generated from data binding when you rebuild the project

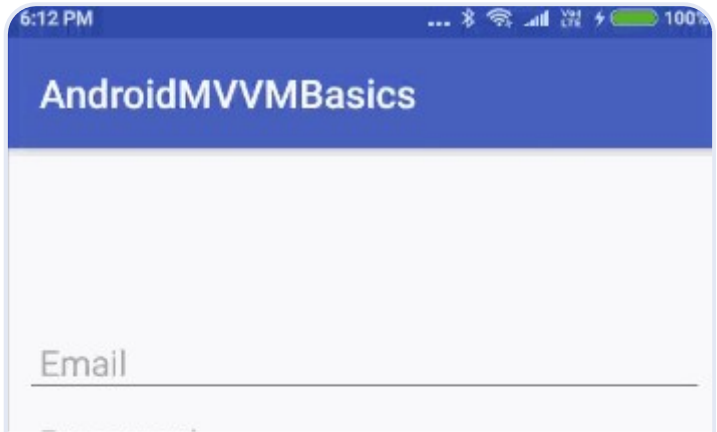
The code for the `MainActivity.java` class is given below:

```
package com.journaldev.androidmvvmbasics.views;


import android.databinding.BindingAdapter;
import android.databinding.DataBindingUtil;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;


import com.journaldev.androidmvvmbasics.R;
import com.journaldev.androidmvvmbasics.databinding.ActivityMainBinding;
import com.journaldev.androidmvvmbasics.viewmodels.LoginViewModel;


public class MainActivity extends AppCompatActivity {
```

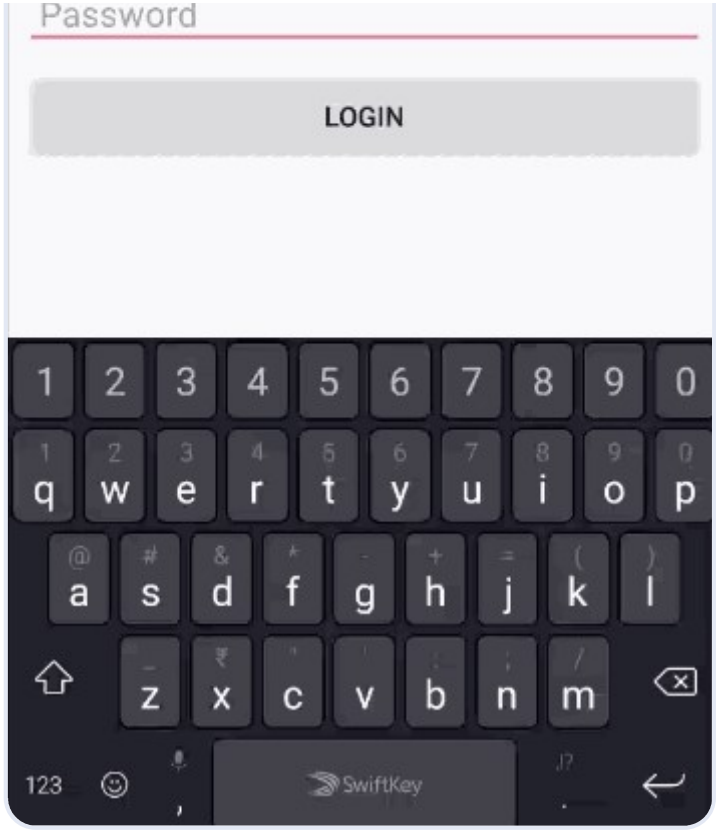
COOKIE PREFERENCES

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ActivityMainBinding activityMainBinding = DataBindingUtil.setContentView(this, R.layout.activi
        activityMainBinding.setViewModel(new LoginViewModel());
        activityMainBinding.executePendingBindings();

    }

    @BindingAdapter({"toastMessage"})
    public static void runMe(View view, String message) {
        if (message != null)
            Toast.makeText(view.getContext(), message, Toast.LENGTH_SHORT).show();
    }
}
```

Thanks to DataBinding, the `ActivityMainBinding` class is auto-generated from the layout. The `@BindingAdapter` method gets triggered whenever toastMessage attribute defined on the Button is changed. It must use the same attribute as defined in the XML and in the ViewModel. So in the above application, the ViewModel updates the Model by listening to the changes in the View. Also, the Model can update the view via the ViewModel using the `notifyPropertyChanged` The output of the above application in action is given below:



COOKIE PREFERENCES



This brings an end to this tutorial on Android MVVM Using DataBinding. You can download the project from the link given below.

AndroidMVVMBasics

Github Project Link

Want to learn more? Join the DigitalOcean Community!

Join our DigitalOcean community of over a million developers for free! Get help and share

COOKIE PREFERENCES