* **LiveData**
* **Data Binding**
* **ViewModel**
* **DataBinding**
* **MVVM**
* **MVP**

[LiveData](https://developer.android.com/reference/androidx/lifecycle/LiveData):

LiveData is lifecycle-aware, meaning it respects the lifecycle of other app components, such as activities, fragments, or services. LiveData is an observable data holder class and it’s keep/ensure only updated data. LiveData only updates app component observers that are in an active lifecycle state.

Two ways to set value in MutableLiveData

1. setValue() : setValue() method must be called from the main thread

2. postValue(): postValue() method must be called from the backgrpund thread

* **The advantages of using LiveData**
* Ensures your UI matches your data state
* No memory leaks
* No crashes due to stopped activities
* Always up to data
* **Steps to work with [LiveData](https://developer.android.com/reference/androidx/lifecycle/LiveData) objects:**
* Create an instance of LiveData to hold a certain type of data. This is usually done within your [ViewModel](https://developer.android.com/reference/androidx/lifecycle/ViewModel) class.
* Attach the Observer object to the LiveData object using the [observe()](https://developer.android.com/reference/androidx/lifecycle/LiveData%23observe(android.arch.lifecycle.LifecycleOwner,%250Aandroid.arch.lifecycle.Observer%253CT%253E)) method. The observe() method takes a [LifecycleOwner](https://developer.android.com/reference/androidx/lifecycle/LifecycleOwner) object.

class NameViewModel : ViewModel() {

    // Create a LiveData with a String

    val currentName: MutableLiveData<String> by lazy {

        MutableLiveData<String>()

    }

    // currentName.setValue(anotherName)

}

class NameActivity : AppCompatActivity() {

// Use the 'by viewModels()' Kotlin property delegate

    // from the activity-ktx artifact

    private val model: NameViewModel by viewModels()

override fun onCreate(savedInstanceState: Bundle?) {

        super.onCreate(savedInstanceState)

// Create the observer which updates the UI.

        val nameObserver = Observer<String> { newName ->

// Update the UI, in this case, a TextView.

            nameTextView.text = newName

}

        // Observe the LiveData, passing in this activity as the LifecycleOwner and the observer.

        model.currentName.observe(this, nameObserver)

**}**

**}**

Data Binding:

enable the dataBinding build option in your build.gradle

android {

    ...

    buildFeatures {

        dataBinding true

    }

}

The Data Binding Library is a support library that allows you to bind UI components in your layouts to data sources in your app using a declarative format rather than programmatically. The Data Binding Library provides classes and methods to easily observe data for changes and  we don't need to refreshing the UI when the underlying data source changes. Data binding layout files are slightly different and start with a root tag of layout followed by a data element and a view root element.

* **One Way data-binding**: One-way-data-binding means data flows in only one direction like updating UI from the data source.
* **Two Way data-binding**: Two–way-data-binding means data flow in both direction like Updating data source from UI and getting updated from a data source.

**Data Class(with simple which data doesn**’**t change in future):**

data class User(val firstName: String, val lastName: String)

**Data Class(with simple which data change in future):**

**Observable objects:** A class that implements the [Observable](https://developer.android.com/reference/android/databinding/Observable) interface allows the registration of listeners that want to be notified of property changes of on the observable object.The data class that implements BaseObservable is responsible for notifying when the properties change. This is done by assigning a [Bindable](https://developer.android.com/reference/android/databinding/Bindable) annotation to the getter and calling the [notifyPropertyChanged()](https://developer.android.com/reference/android/databinding/BaseObservable%23notifypropertychanged) method in the setter. Data binding generates a class named BR in the module package which contains the IDs of the resources used for data binding.

class User : BaseObservable() {

    @get:Bindable

    var firstName: String = “"

        set(value) {

            field = value

            notifyPropertyChanged(BR.firstName)

        }

    @get:Bindable

    var lastName: String = “"

        set(value) {

            field = value

            notifyPropertyChanged(BR.lastName)

        }

}

**Bind DataClass & View(Activity):**

val binding: ActivityMainBinding = DataBindingUtil.setContentView(this, R.layout.activity\_main)

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

**Bind DataClass & View(Fragment & RecycleView Adapter):**

val listItemBinding = ListItemBinding.inflate(layoutInflater, viewGroup, false)

// or

val listItemBinding = DataBindingUtil.inflate(layoutInflater, R.layout.list\_item, viewGroup, false)

**Layout:**

// android:text="@{user.lastName}"

// android:text="@{user.displayName ?? user.lastName}"

// android:text="@{user.displayName != null ? user.displayName : user.lastName}"

// android:onClick="@{handlers::onClickFriend}"

// android:visibility="@{user.isAdult ? View.VISIBLE : View.GONE}"/>

<?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"/>

<variable

name="handlers"

type="com.example.MyHandlers"/>

   </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.firstName}"

            android:onClick="@{handlers::onClickFriend}"/>

<Button android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

        android:onClick="@{() -> handlers.onSaveClick(task)}" />

<Button android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

        android:onClick="@{(view) -> handlers.onSaveClick(task)}"

<include layout="@layout/contact"

            bind:user="@{user}"/>

    </LinearLayout>

</layout>

**Handler Class:**

class MyHandlers {

    fun onClickFriend(view: View) { ... }

fun onSaveClick(task: Task){}

fun onSaveClick(view: View, task: Task){}

}

MVVM:

There are two ways to implement MVVM in Android:

1. Data Binding
2. RxJava

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 too many relationship.
* The ViewModel does not know that the View is listening to it.
* Model doesn’t have any reference of ViewModel & View
* ViewModel doesn’t have reference of View of have Model reference
* View doesn’t have reference of Model but reference of 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](https://developer.android.com/reference/androidx/lifecycle/ViewModel): The [ViewModel](https://developer.android.com/reference/androidx/lifecycle/ViewModel) class is designed to store and manage UI-related data in a lifecycle conscious way. The` [ViewModel](https://developer.android.com/reference/androidx/lifecycle/ViewModel) class allows data to survive configuration changes such as screen rotations. If the activity is re-created, it receives the same ViewModel instance that was created by the first activity. When the owner activity is finished, the framework calls the [ViewModel](https://developer.android.com/reference/androidx/lifecycle/ViewModel) objects's [onCleared()](https://developer.android.com/reference/androidx/lifecycle/ViewModel%23onCleared()) method so that it can clean up resources.

It acts as a link between the Model and the View. It’s responsible for transforming the data from the Model to view. It provides data streams to the View.

The ViewModel doesn’t holds references to the View, The ViewModel sends data streams. The View and the ViewModel are in a 1 to many relationship and The ViewModel does not know that the View is listening to it.

Caution: A ViewModel must never reference a view, Lifecycle, or any class that may hold a reference to the activity context.

//Kotlin

class MyViewModel : ViewModel() {

    private val users: MutableLiveData<List<User>> by lazy {

        MutableLiveData<List<User>>().also {

            loadUsers()

        }

    }

    fun getUsers(): LiveData<List<User>> {

        return users

    }

    private fun loadUsers() {

        // Do an asynchronous operation to fetch users.

    }

}

class MyActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {

        // Create a ViewModel the first time the system calls an activity's onCreate() method.

        // Re-created activities receive the same MyViewModel instance created by the first activity.

        // Use the 'by viewModels()' Kotlin property delegate

        // from the activity-ktx artifact

        val model: MyViewModel by viewModels()

        model.getUsers().observe(this, Observer<List<User>>{ users ->

            // update UI

        })

    }

}

//Java

public class MyViewModel extends ViewModel {

    private MutableLiveData<List<User>> users;

    public LiveData<List<User>> getUsers() {

        if (users == null) {

            users = new MutableLiveData<List<User>>();

            loadUsers();

        }

        return users;

    }

    private void loadUsers() {

        // Do an asynchronous operation to fetch users.

    }

}

public class MyActivity extends AppCompatActivity {

    public void onCreate(Bundle savedInstanceState) {

        // Create a ViewModel the first time the system calls an activity's onCreate() method.

        // Re-created activities receive the same MyViewModel instance created by the first activity.

        MyViewModel model = new ViewModelProvider(this).get(MyViewModel.class);

        model.getUsers().observe(this, users -> {

            // update UI

        });

    }

}

MVVM With Retrofit

<https://howtodoandroid.com/mvvm-retrofit-recyclerview-kotlin/>

* 1. **Data Class: for hold the data**

data class Movie(val name: String, val imageUrl: String, val category: String, val desc: String)

* 1. **Retrofit Interface & Object: For Networking call**

interface RetrofitService {

@GET("movielist.json")

fun getAllMovies() : Call<List<Movie>>

companion object {

var retrofitService: RetrofitService? = null

fun getInstance() : RetrofitService {

if (retrofitService == null) {

val retrofit = Retrofit.Builder()

.baseUrl("https://howtodoandroid.com/")

.addConverterFactory(GsonConverterFactory.create())

.build()

retrofitService = retrofit.create(RetrofitService::class.java)

}

return retrofitService!!

}

}

}

* 1. **Data Repository:** I am using a repository pattern to handle the data from API. In the repository class, we need to pass the retrofit service instance to perform the network call. We don’t need to handle the response here in the repository. That will be part of the ViewModel.

class MainRepository constructor(private val retrofitService: RetrofitService) {

fun getAllMovies() = retrofitService.getAllMovies()

}

* 1. **View-Model:**  In the ViewModel setup, We need to create a class and **extend** the ViewModel. ViewModel class has the business logic and API call implementations. In the ViewModel constructor, we need to pass the data repository to handle the data.

**Note**: We are using Live data to update the data to UI.

class MainViewModel constructor(private val repository: MainRepository) : ViewModel() {

val movieList = MutableLiveData<List<Movie>>()

val errorMessage = MutableLiveData<String>()

fun getAllMovies() {

val response = repository.getAllMovies()

response.enqueue(object : Callback<List<Movie>> {

override fun onResponse(call: Call<List<Movie>>, response: Response<List<Movie>>) {

movieList.postValue(response.body())

}

override fun onFailure(call: Call<List<Movie>>, t: Throwable) {

errorMessage.postValue(t.message)

}

})

}

}

* 1. **View Model Factory:** 
     1. We can not create ViewModel on our own. We need the ViewModelProviders utility provided by Android to create ViewModels.
     2. But ViewModelProviders can only instantiate ViewModels with the no-arg constructor.
     3. So if I have a ViewModel with multiple arguments, then I need to use a Factory that I can pass to ViewModelProviders to use when an instance of MyViewModel is required.

class MyViewModelFactory constructor(private val repository: MainRepository): ViewModelProvider.Factory {

override fun <T : ViewModel> create(modelClass: Class<T>): T {

return if (modelClass.isAssignableFrom(MainViewModel::class.java)) {

MainViewModel(this.repository) as T

} else {

throw IllegalArgumentException("ViewModel Not Found")

}

}

}

* 1. **View: For show/Bind the data**

class MainActivity : AppCompatActivity() {

private lateinit var viewModel: MainViewModel

private val retrofitService = RetrofitService.getInstance()

override fun onCreate(savedInstanceState: Bundle?) {

viewModel = ViewModelProvider(this, MyViewModelFactory(MainRepository(retrofitService))).get(MainViewModel::class.java)

viewModel.movieList.observe(this, Observer {

Log.d(TAG, "onCreate: $it")

})

viewModel.errorMessage.observe(this, Observer {

Log.d(TAG, "onCreate: $it")

})

viewModel.getAllMovies()

}

}

MVC(Model View Controller):

Developing an [android](https://www.geeksforgeeks.org/kotlin-android-tutorial/)application by applying a software architecture pattern is always preferred by the developers. An [architecture pattern](https://www.geeksforgeeks.org/android-architecture-patterns/) gives modularity to the project files and assures that all the codes get covered in Unit testing.

-MVC is **an architectural pattern consisting of three parts: Model, View, Controller**. Model: Handles data logic. View: It displays the information from the model to the user. Controller: It controls the data flow into a model object and updates the view whenever data changes.  
MVC Architecture Components

**It has 3 Components i.e. MODEL-VIEW-CONTROLLER**.

* **MODEL**  
  \* Here Model is nothing but a data, it directly manages the data, logic and rules of the application.  
  \* A Model is responsible for managing data of an app.  
  \* 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**  
  \* A View in MVC is nothing but a UI design, How you can display the data to the USER screen.  
  \* A view means presentation of the data in a particular format.  
  \* Itis theUI(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**  
  \* A Controller is typically a piece, which control all the task/event that a user perform, Such as event handling, navigation, Communication between model & view happens in controller in MVC.  
  \* A Controller recieve the input, validate it, & pass the validated input to Model.  
  \* 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 **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.

Image

Kotlin MVC Example

In **Model** Package create 2 files and add the code as below

* 1. **IUser** As **Interface**
  2. **User** As Model **Class**

In **Controller** Package Create 2 files and add the code as below

* 1. **ILoginController** Ass **Interface**
  2. **LoginController** As **Class**

In **View** Package Create 2 files and add the code as below

* 1. **ILoginView** As **Interafce**
  2. **MainActivity** As **Class**

**IUser.kt (Interface)**

interface IUser {

fun getEmail(): String?

fun getPassword(): String?

fun isValid(): Int

}

**User.kt (Class)**

class User(

private val email: String?,

private val password: String?

) : IUser {

override fun getEmail(): String? {

return email

}

override fun getPassword(): String? {

return password

}

override fun isValid(): Int {

if(TextUtils.isEmpty(getEmail()))

return 0

else if(!Patterns.EMAIL\_ADDRESS.matcher(getEmail()).matches())

return 1

else if(TextUtils.isEmpty(getPassword()))

return 2

else if(getPassword()?.length!! <=6)

return 3

else

return -1;

}

}

**ILoginController(Interface)**

interface ILoginController {

fun OnLogin(email: String?, password: String?)

}

LoginController(Class)

import com.team42.mvcdemokotlinloginexample.Model.User

import com.team42.mvcdemokotlinloginexample.View.ILoginView

class LoginController(

private val loginView: ILoginView

):ILoginController {

override fun OnLogin(email: String?, password: String?) {

val user = User(email, password)

val loginCode = user.isValid()

when (loginCode) {

0 -> {

loginView.OnLoginError("Please enter Email");

}

1 -> {

loginView.OnLoginError("Please enter A valid Email");

}

2 -> {

loginView.OnLoginError("Please enter Password");

}

3 -> {

loginView.OnLoginError("Please enter Password greater the 6 char");

}

else -> {

loginView.OnLoginSuccess("login Successful");

}

}

}

}

**ILoginView(Interface)**

interface ILoginView {

fun OnLoginSuccess(message: String?)

fun OnLoginError(message: String?)

}

**MainActivity.kt**

import android.os.Bundle

import android.widget.Button

import android.widget.EditText

import android.widget.Toast

import androidx.appcompat.app.AppCompatActivity

import com.team42.mvcdemokotlinloginexample.Controller.ILoginController

import com.team42.mvcdemokotlinloginexample.Controller.LoginController

import com.team42.mvcdemokotlinloginexample.View.ILoginView

class MainActivity : AppCompatActivity(), ILoginView {

var email: EditText? = null

var password: EditText? = null

var loginb: Button? = null

var loginPresenter: ILoginController? = null

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_main)

email = findViewById(R.id.edtUserName)

password = findViewById(R.id.edtPassword)

loginb = findViewById(R.id.mButtonLogin)

loginPresenter = LoginController(this)

loginb?.setOnClickListener {

(loginPresenter as LoginController).OnLogin(

email?.text.toString(),

password?.text.toString().trim()

)

}

}

override fun OnLoginSuccess(message: String?) {

Toast.makeText(this,message, Toast.LENGTH\_LONG).show()

}

override fun OnLoginError(message: String?) {

Toast.makeText(this,message,Toast.LENGTH\_LONG).show()

}

}

Image

Java MVC Example

In **Model** Package create 2 files and add the code as below

* 1. **IUser** As **Interface**
  2. **User** As Model **Class**

In **Controller** Package Create 2 files and add the code as below

* 1. **ILoginController** Ass **Interface**
  2. **LoginController** As **Class**

In **View** Package Create 2 files and add the code as below

* 1. **ILoginView** As **Interafce**
  2. **MainActivity** As **Class**

**IUser.java( Interface)**

|  |
| --- |
| public interface IUser **{** |
| String getEmail**()**; |
| String getPassword**()**; |
| int isValid**()**; |
| **}** |

**User.java (class)**

|  |
| --- |
| public **class** User **implements** IUser**{** |
| private String email,password; |
| public User**(**String email, String password**)** **{** |
| **this**.email = email; |
| **this**.password = password; |
| **}** |
| @Override |
| public String getEmail**()** **{** |
| **return** email; |
| **}** |
| @Override |
| public String getPassword**()** **{** |
| **return** password; |
| **}** |
| @Override |
| public int isValid**()** **{** |
| // 0. Check for Email Empty |
| // 1. Check for Email Match pattern |
| // 2. Check for Password > 6 |
|  |
| **if(**TextUtils.isEmpty**(**getEmail**()))** |
| **return** 0; |
| **else** **if(**!Patterns.EMAIL\_ADDRESS.matcher**(**getEmail**())**.matches**())** |
| **return** 1; |
| **else** **if(**TextUtils.isEmpty**(**getPassword**()))** |
| **return** 2; |
| **else** **if(**getPassword**()**.length**()<**=6**)** |
| **return** 3; |
| **else** |
| **return** -1; |
| **}** |
| **}** |

**ILoginController(Interface)**

|  |
| --- |
| public interface ILoginController **{** |
| **void** OnLogin**(**String email,String Password**)**; |
| **}** |

**LoginController(Class)**

|  |
| --- |
| public **class** LoginController **implements** ILoginController **{** |
|  |
| ILoginView loginView; |
|  |
| public LoginController**(**ILoginView loginView**)** **{** |
| **this**.loginView = loginView; |
| **}** |
|  |
| @Override |
| public **void** OnLogin**(**String email, String password**)** **{** |
| User user = **new** User**(**email,password**)**; |
| int loginCode = user.isValid**()**; |
| **if(**loginCode == 0**)** |
| **{** |
| loginView.OnLoginError**(**"Please enter Email"**)**; |
| **}else** **if** **(**loginCode == 1**){** |
| loginView.OnLoginError**(**"Please enter A valid Email"**)**; |
| **}** **else** **if** **(**loginCode == 2**)** |
| **{** |
| loginView.OnLoginError**(**"Please enter Password"**)**; |
| **}else** **if(**loginCode == 3**){** |
| loginView.OnLoginError**(**"Please enter Password greater the 6 char"**)**; |
| **}** |
| **else** **{** |
| loginView.OnLoginSuccess**(**"login Successful"**)**; |
| **}** |
| **}** |
| **}** |

**ILoginView (Interface)**

|  |
| --- |
| public interface ILoginView **{** |
|  |
| **void** OnLoginSuccess**(**String message**)**; |
| **void** OnLoginError**(**String message**)**; |
| **}** |

**MainActivity.java**

|  |
| --- |
| import com.example.mvcexample.Controller.ILoginController; |
| import com.example.mvcexample.Controller.LoginController; |
| import com.example.mvcexample.View.ILoginView; |
|  |
| public **class** MainActivity **extends** AppCompatActivity **implements** ILoginView **{** |
| EditText email,password; |
| Button loginb; |
| ILoginController loginPresenter; |
|  |
| @Override |
| protected **void** onCreate**(**Bundle savedInstanceState**)** **{** |
| super.onCreate**(**savedInstanceState**)**; |
| setContentView**(**R.layout.activity\_main**)**; |
|  |
| email = **(**EditText**)** findViewById**(**R.id.email**)**; |
| password = **(**EditText**)**findViewById**(**R.id.password**)**; |
|  |
| loginb = **(**Button**)** findViewById**(**R.id.loginb**)**; |
| loginPresenter = **new** LoginController**(this)**; |
|  |
| loginb.setOnClickListener**(new** View.OnClickListener**()** **{** |
| @Override |
| public **void** onClick**(**View v**)** **{** |
| loginPresenter.OnLogin**(**email.getText**()**.toString**()**.trim**()**,password.getText**()**.toString**()**.trim**())**; |
| **}** |
| **})**; |
| **}** |
|  |
| @Override |
| public **void** OnLoginSuccess**(**String message**)** **{** |
| Toast.makeText**(this**,message,Toast.LENGTH\_SHORT**)**.show**()**; |
| **}** |
|  |
| @Override |
| public **void** OnLoginError**(**String message**)** **{** |
| Toast.makeText**(this**,message,Toast.LENGTH\_SHORT**)**.show**()**; |
| **}** |
| **}** |