Login API with retrofit and MVVM with auto-login in android kotlin.

Today we are going to learn how to do login using android kotlin with retrofit MVVM.

For API implementation I am using this website.

https://www.appsloveworld.com/sample-rest-api-url-for-testing-with-authentication

First, we need to add internet permission to the manifest file.

```
<uses-permission android:name="android.permission.INTERNET" />
```

Now we need to add dependency in the build.Gradle(:app)

```
//RetroFit Dependencies

implementation 'com.google.code.gson:gson:2.8.9'

implementation 'com.squareup.retrofit2:retrofit:2.9.0'
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'

implementation

'com.squareup.okhttp3:logging-interceptor:5.0.0-alpha.1'

//Coroutains"

implementation

"org.jetbrains.kotlinx:kotlinx-coroutines-android:1.5.2"

implementation

'androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.1' //viewModel

scope

implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.4.1'

//lifecycle scope

implementation 'androidx.fragment:fragment-ktx:1.4.1'



//Lifecycle

implementation 'androidx.lifecycle:lifecycle-common:2.4.1'

implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.4.1'

implementation

'androidx.lifecycle:lifecycle-livedata-ktx:2.4.1'

//size dp/sp

implementation 'com.intuit.sdp:sdp-android:1.0.6'

implementation 'com.intuit.ssp:ssp-android:1.0.6'
implementation "androidx.preference:preference-ktx:1.2.0"
```

after that, we create one object file for adding the base URL.

Constant.kt

```
object Constant {
    const val BASE_URL = "http://restapi.adequateshop.com"
}
```

here I am using HTTP instead of HTTPS that's why I am adding a network

security config file in XML/network_security_config.

network_security_config.xml

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
<domain-config cleartextTrafficPermitted="true">
<domain
includeSubdomains="true">http://restapi.adequateshop.com<
/domain>
</domain-config>
</network-security-config>
```

Now we need to create API requests and response classes.

LoginRequest.kt

```
import com.google.gson.annotations.SerializedName

data class LoginRequest(
   @SerializedName("email")
   var email: String,
   @SerializedName("password")
   var password: String
)
```

```
import com.google.gson.annotations.SerializedName

data class LoginResponse(
   @SerializedName("code")
   var code: Int,
   @SerializedName("data")
   var `data`: Data,
   @SerializedName("id")
   var id: String,
   @SerializedName("message")
   var message: String
) {
   data class Data(
      @SerializedName("Email")
      var email: String,
      @SerializedName("id")
      var id: String,
      @SerializedName("Id")
      var id2: Int,
      @SerializedName("Name")
      var name: String,
      @SerializedName("Token")
      var token: String
   )
}
```

BaseResponse.kt

```
sealed class BaseResponse<out T> {
   data class Success<out T>(val data: T? = null) :
BaseResponse<T>()
   data class Loading(val nothing: Nothing?=null) :
BaseResponse<Nothing>()
   data class Error(val msg: String?) : BaseResponse<Nothing>()
}
```

Here we add our retrofit client class.

ApiClient.kt

```
object ApiClient {
   var mHttpLoggingInterceptor = HttpLoggingInterceptor()
      .setLevel(HttpLoggingInterceptor.Level.BODY)

   var mOkHttpClient = OkHttpClient
      .Builder()
      .addInterceptor(mHttpLoggingInterceptor)
      .build()

   var mRetrofit: Retrofit? = null


   val client: Retrofit?
      get() {
         if(mRetrofit == null){
            mRetrofit = Retrofit.Builder()
               .baseUrl(Constant.BASE_URL)
               .client(mOkHttpClient)
               .addConverterFactory(GsonConverterFactory.crea
te())
               .build()
            }
            return mRetrofit
        }
}
```

After adding the retrofit client class now we need to add an API interface.

UserApi.kt

```
interface UserApi {

    @POST("/api/authaccount/login")
    suspend fun loginUser(@Body loginRequest: LoginRequest):
Response<LoginResponse>
    companion object {
        fun getApi(): UserApi? {
            return ApiClient.client?.create(UserApi::class.java)
        }
    }
}
```

After creating the API interface now we need to add a repository class.

UserRepository.kt

```
class UserRepository {

   suspend fun loginUser(loginRequest:LoginRequest):
Response<LoginResponse>? {
     return  UserApi.getApi()?.loginUser(loginRequest =
loginRequest)
    }
}
```

Now we will crate the view model file.

LoginViewModel.kt

```
class LoginViewModel(application: Application) :
AndroidViewModel(application) {
val userRepo = UserRepository()
val loginResult: MutableLiveData<BaseResponse<LoginResponse>> =
MutableLiveData()
fun loginUser(email: String, pwd: String) {
loginResult.value = BaseResponse.Loading()
viewModelScope.launch {
try {

val loginRequest = LoginRequest(
password = pwd,
email = email
)
val response = userRepo.loginUser(loginRequest = loginRequest)
if (response?.code() == 200) {
loginResult.value = BaseResponse.Success(response.body())
} else {
loginResult.value = BaseResponse.Error(response?.message())
}

} catch (ex: Exception) {
loginResult.value = BaseResponse.Error(ex.message)
} }
}
```

we will create our design part first

Kotlin Basics with MVVM

Prepared by TommyTV

activity_main.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"
tools:context=".ui.MainActivity">

<ProgressBar
android:id="@+id/prgbar"
android:layout_width="48dp"
android:layout_height="48dp"
android:visibility="gone"
app:layout_constraintBottom_toBottomOf="parent"
app:layout_constraintEnd_toEndOf="parent"
app:layout_constraintStart_toStartOf="parent"
app:layout_constraintTop_toTopOf="parent" />

<com.google.android.material.textfield.TextInputLayout
android:id="@+id/txtLay_emailAdd"
style="@style/Widget.MaterialComponents.TextInputLayout.OutlinedB
ox"
android:layout_width="0dp"
android:layout_height="0dp"
android:hint="Email"
app:endIconMode="clear_text"
app:layout_constraintBottom_toBottomOf="parent"
app:layout_constraintEnd_toEndOf="parent"
app:layout_constraintHeight_percent="0.1"
app:layout_constraintStart_toStartOf="parent"
app:layout_constraintTop_toTopOf="parent"
app:layout_constraintVertical_bias="0.4"
app:layout_constraintWidth_percent="0.9">

<com.google.android.material.textfield.TextInputEditText
android:id="@+id/txtInput_email"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:maxLength="20" />
</com.google.android.material.textfield.TextInputLayout>
```

```
<com.google.android.material.textfield.TextInputLayout
android:id="@+id/txtLay_pass_signup"
style="@style/Widget.MaterialComponents.TextInputLayout.OutlinedB
ox"
android:layout_width="0dp"
android:layout_height="0dp"
android:hint="PASSWORD"
app:endIconDrawable="@drawable/ic_lock"
app:endIconMode="password_toggle"
app:layout_constraintEnd_toEndOf="parent"
app:layout_constraintHeight_percent="0.1"
app:layout_constraintStart_toStartOf="parent"
app:layout_constraintTop_toBottomOf="@+id/txtLay_emailAdd"
app:layout_constraintWidth_percent="0.9">

<com.google.android.material.textfield.TextInputEditText
android:id="@+id/txt_pass"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:inputType="textPassword"
android:maxLength="15" />
</com.google.android.material.textfield.TextInputLayout>

<Button
android:id="@+id/btn_login"
android:layout_width="0dp"
android:layout_height="48dp"
android:layout_marginEnd="@dimen/_2sdp"
android:text="@string/login"
app:layout_constraintBottom_toBottomOf="parent"
app:layout_constraintEnd_toStartOf="@+id/btn_register"
app:layout_constraintHorizontal_bias="0.5"
app:layout_constraintStart_toStartOf="parent" />


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

SessionManager.kt

```
object SessionManager {

    const val USER_TOKEN = "user_token"

    /**
     * Function to save auth token
     */
    fun saveAuthToken(context: Context, token: String) {
        saveString(context, USER_TOKEN, token)
    }

    /**
     * Function to fetch auth token
     */
    fun getToken(context: Context): String? {
        return getString(context, USER_TOKEN)
    }

    fun saveString(context: Context, key: String, value: String)
{
        val prefs: SharedPreferences =

context.getSharedPreferences(context.getString(R.string.app_na
me), Context.MODE_PRIVATE)
        val editor = prefs.edit()
        editor.putString(key, value)
        editor.apply()

    }

    fun getString(context: Context, key: String): String? {
        val prefs: SharedPreferences =

context.getSharedPreferences(context.getString(R.string.app_na
me), Context.MODE_PRIVATE)
        return prefs.getString(this.USER_TOKEN, null)
    }
    fun clearData(context: Context){
     val editor =
context.getSharedPreferences(context.getString(R.string.app_na
me), Context.MODE_PRIVATE).edit()
        editor.clear()
        editor.apply()
    }
}
```

now we will code for the main activity

MainActivity.kt

```
class MainActivity : AppCompatActivity() {

private lateinit var binding: ActivityMainBinding
private val viewModel by viewModels<LoginViewModel>()

override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
binding = ActivityMainBinding.inflate(layoutInflater)
val view = binding.root
setContentView(view)
val token = SessionManager.getToken(this)
if (!token.isNullOrBlank()) {
navigateToHome()
}

viewModel.loginResult.observe(this) {
when (it) {
is BaseResponse.Loading -> {
showLoading()
}

is BaseResponse.Success -> {
stopLoading()
processLogin(it.data)
}

is BaseResponse.Error -> {
processError(it.msg)
}
else -> {
stopLoading()
}
}
}

binding.btnLogin.setOnClickListener {
doLogin()

}
```

```
binding.btnRegister.setOnClickListener {
doSignup()
}

}

private fun navigateToHome() {
val intent = Intent(this, LogoutActivity::class.java)
intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP or
Intent.FLAG_ACTIVITY_NEW_TASK)
intent.addFlags(FLAG_ACTIVITY_NO_HISTORY)
startActivity(intent)
}

fun doLogin() {
val email = binding.txtInputEmail.text.toString()
val pwd = binding.txtPass.text.toString()
viewModel.loginUser(email = email, pwd = pwd)

}

fun doSignup() {

}

fun showLoading() {
binding.prgbar.visibility = View.VISIBLE
}

fun stopLoading() {
binding.prgbar.visibility = View.GONE
}

fun processLogin(data: LoginResponse?) {
showToast("Success:" + data?.message)
if (!data?.data?.token.isNullOrEmpty()) {
data?.data?.token?.let { SessionManager.saveAuthToken(this,
it) }
navigateToHome()
}
}

fun processError(msg: String?) {
showToast("Error:" + msg)
```

```
}

fun showToast(msg: String) {
Toast.makeText(this, msg, Toast.LENGTH_SHORT).show()
}
}
```

now We design our logout screen.

activity_logout.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"
   tools:context=".ui.LogoutActivity">

   <TextView
      android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:text="Welcome user"
        android:textSize="@dimen/_16sdp"
        android:textStyle="bold|italic"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        android:layout_marginBottom="@dimen/_70sdp"
        />
<Button
    android:id="@+id/btn_logout"
    android:text="Logout"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toBottomOf="parent"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

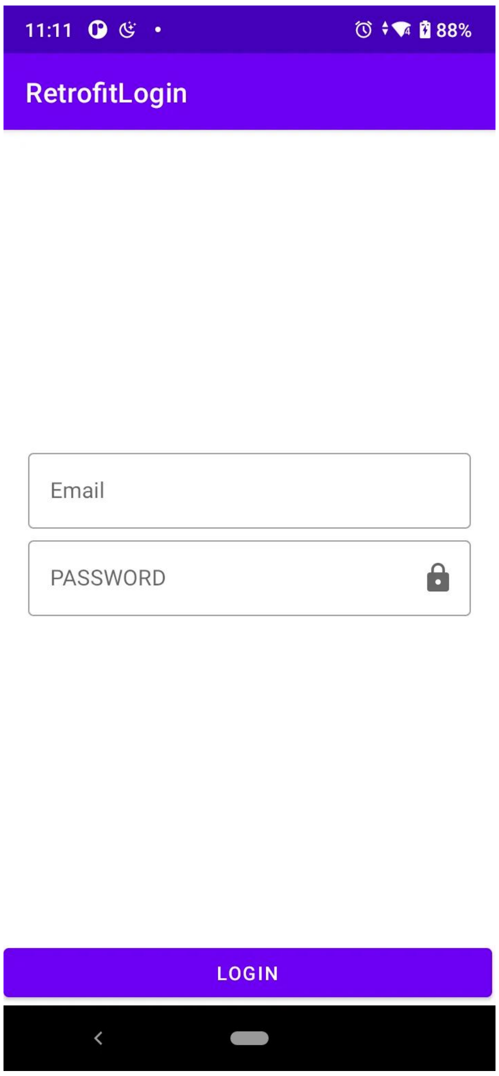
We need to code for the logout screen.

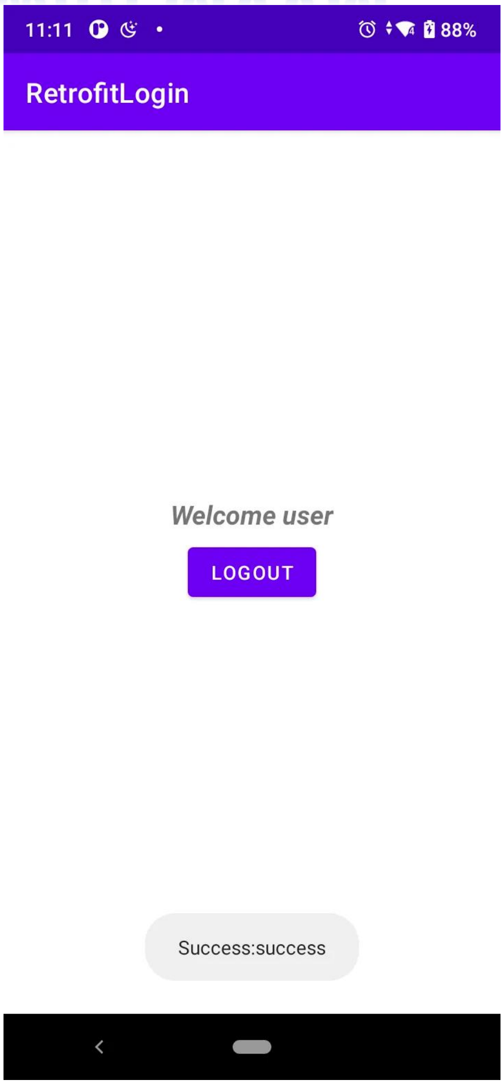
LogoutActivity.kt

```
class LogoutActivity : AppCompatActivity() {

    private lateinit var binding: ActivityLogoutBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding =
ActivityLogoutBinding.inflate(layoutInflater)
        val view = binding.root
        setContentView(view)

        binding.btnLogout.setOnClickListener {
            SessionManager.clearData(this)
            val intent = Intent(this,
MainActivity::class.java)
            intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP
or Intent.FLAG_ACTIVITY_NEW_TASK)
            intent.addFlags(Intent.FLAG_ACTIVITY_NO_HISTORY)
            startActivity(intent)
        }
    }
}
```

That's it. Happy Coding

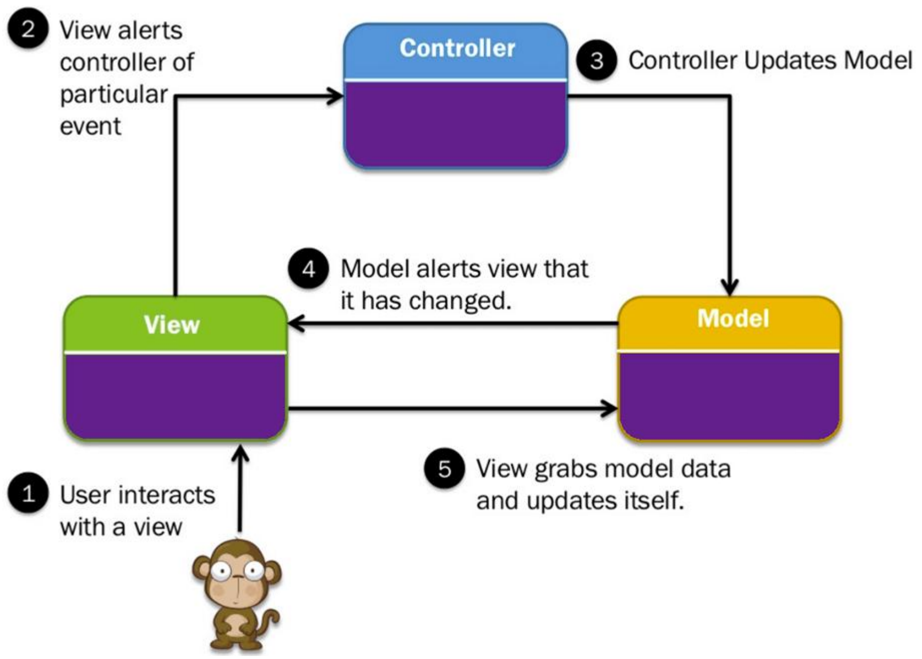
Kotlin Basics with MVVM





Prepared by TommyTV

Part 2 : Model View View-Model


2 View alerts controller of particular event
Controller
3 Controller Updates Model
4 Model alerts view that it has changed.
View
Model
1 User interacts with a view
5 View grabs model data and updates itself.

three important MVC the components are:

- Model: It includes all the data and its related logic.
- View: Present data to the user or handles user interaction.
- Controller: An interface between Model and View components.

Let’s see each of this component in detail:

Model

The model component stores data and related logic. It represents data that is being transferred between controller components or any other related business logic.

For example, a Controller object helps you to retrieve the customer info from the database. It manipulates data and sends it back to the database or use it to render the same data.

A View is that part of the Application that represents the presentation of data. Views are created by the data gathered from the model data. A view requests the Model to give information so that it resents the output to the user.

The View also represents the data from charts, diagrams, and table. For example, any customer view will include all the UI components like text boxes, dropdowns, etc.

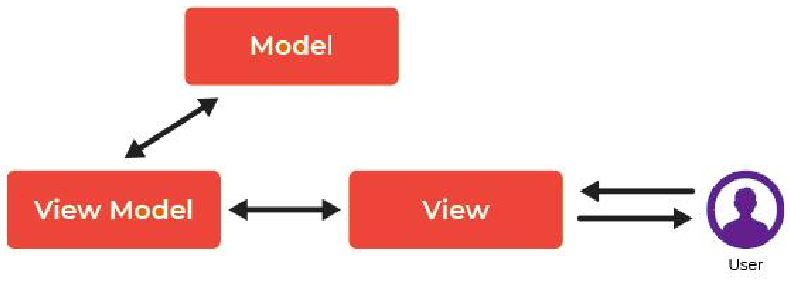
Controller

The Controller is that part of the Application that handles the user interaction. The Controller interprets the mouse and keyboard inputs from the user, informing the Model and the View to change as appropriate.

A Controller sends commands to the Model to update its state(E.g., Saving a specific document). The Controller also sends commands to its associated view to change the View's presentation (For example, scrolling a particular document).

MVVM Pattern

Here, is a pattern for MVVM:



MVVM architecture offers two-way data binding between view and view-model. It also helps you to automate the propagation of modifications inside View-Model to the view. The view-model makes use of observer pattern to make changes in the view-model.

Let's see each other this component in detail:

Model

The model stores data and related logic. It represents data that is being transferred between controller components or any other related business logic.

For example, a Controller object will retrieve the student info from the school database. It manipulates data and sends it back to the database or use it to render the same data.

View:

The View stands for UI components like HTML, CSS, jQuery, etc. In MVVC pattern view is held responsible for displaying the data which is received from the Controller as an outcome. This View is also transformed Model (s) into the User Interface (UI).

View Model:

The view model is responsible for presenting functions, commands, methods, to support the state of the View. It is also accountable to operate the model and activate the events in the View.

Difference between MVC and MVVM Architecture

MVC (Model View Controller)	MVVM (Model View ViewModel)
Controller is the entry point to the Application.	The view is the entry point to the Application.
One to many relationships between Controller & View.	One to many relationships between View & View Model.
View Does not have reference to the Controller	View have references to the View-Model.
MVC is Old Model	MVVM is a relatively New Model.
Difficult to read, change, to unit test, and reuse this Model	The debugging process will be complicated when we have complex data bindings.
MVC Model component can be tested separately from the	Easy for separate unit testing and code is event-driven.

Here, are the important difference between MVVM and MVC

Implementing MVVM architecture in Android using Kotlin

Prepared by TommyTV

Kotlin Basics with MVVM

This tutorial is suitable for beginners. Especially those who have just started learning Android programming in Kotlin. Every application needs to follow certain architectural principles.

Failure to adhere to this requirement results in applications difficult to scale and maintain. As a result, more time and resources will be needed to push even simple updates. Therefore, the developer may end up missing crucial opportunities.

Introduction

Let us start by evaluating what android architectures existed before MVVM. The first component is Model View Presenter denoted by MVP. Though this architecture separates the business logic from the app's UI, it is difficult to implement.

In the long-run, this can translate into high development costs. The second android architecture is MVC.

Just like MVP, it is also quite complex and not suitable for minor projects. Google introduced MVVM

(Model-View-ViewModel) to resolve these challenges. By separating code into smaller chunks, MVVM simplifies the debugging process.

Through this article, you'll understand MVVM architecture and implement it in an application. This article shows how to debug common errors resulting from this architecture.

Learn more about MVVM here.

Let's dive in!

Prerequisites

1. Have Android studio installed.
2. You must be familiar with Kotlin.
3. Install lifecycle dependencies.
4. Download the start code from here.

The goal of the tutorial

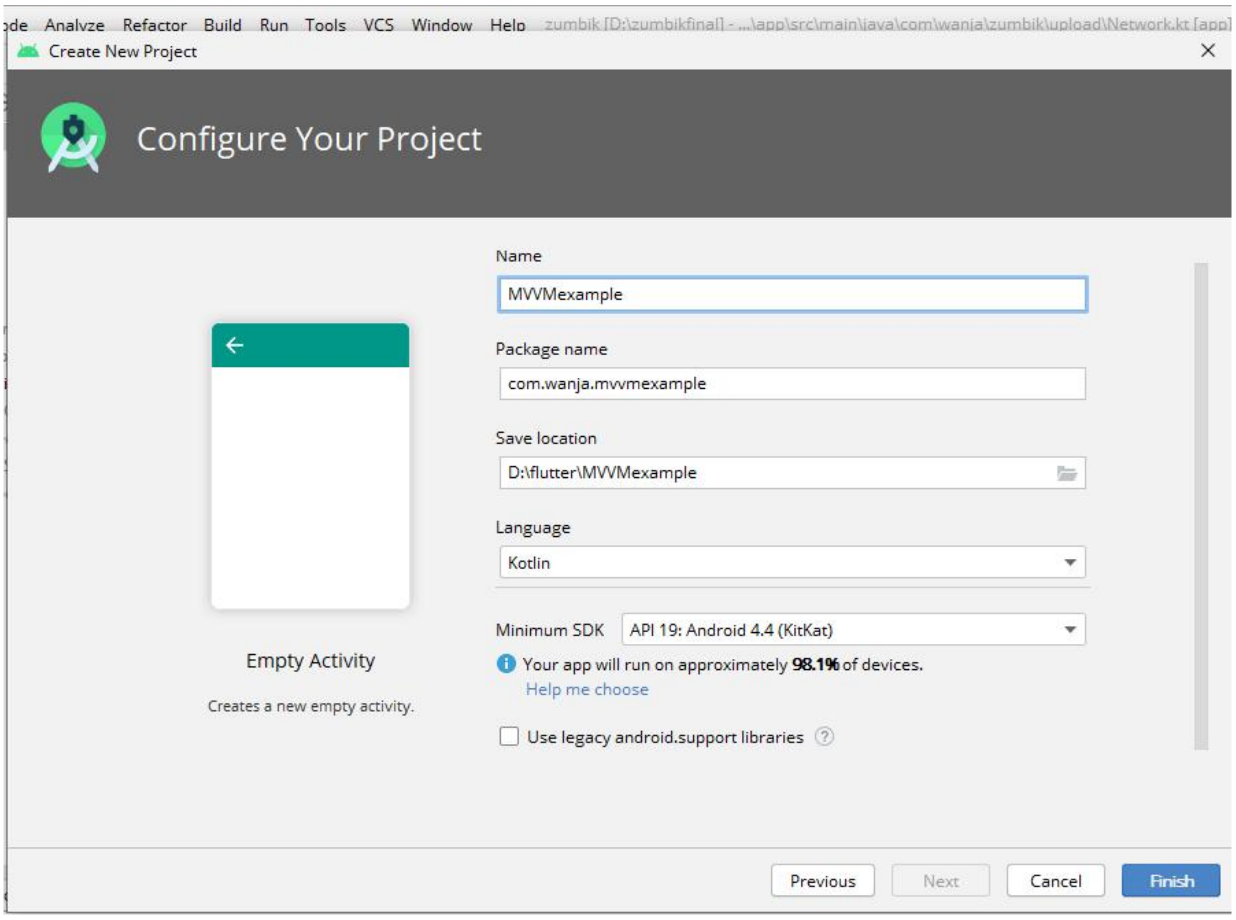
Kotlin Basics with MVVM

By the end of this tutorial, you will create an app that takes input and displays it on a recycler view. Below is the screenshot of the app.

Step 1 – Launching Android Studio

Launch Android Studio and create a new project, as shown below. Make sure that you select Kotlin as your preferred programming language.

If you don't have Android Studio, you can install it from here.


ode Analyze Refactor Build Run Tools VCS Window Help zumbik [D:\zumbikfinal] - ...\app\src\main\java\com\wanja\zumbik\upload\Network.kt [app]
Create New Project
Configure Your Project
Empty Activity
Creates a new empty activity.
Previous
Next
Cancel
Finish

Name: MVVMexample

Package name: com.wanja.mvvmexample

Save location: D:\flutter\MVVMexample

Language: Kotlin

Minimum SDK: API 19: Android 4.4 (KitKat)

Your app will run on approximately **98.1%** of devices.
Help me choose

☐ Use legacy android.support libraries

Step 2 – Creating the model

Create the app model. Also referred to as the data class. To avoid confusion, create a package named model inside the java folder. Then, create a data class named Blog in the model package, as shown below.

For simplicity, the data class will only have one variable (title). There is no need to add getters and setters; Kotlin adds them to the class automatically.

Here's the code for the class.

```
data class Blog(        var title:String
)
```

Step 3 – Creating the view

The view is what the user sees on the screen. The UI, therefore, needs to be well structured to minimize any confusion and dissatisfaction.

Open activity_main.xml file and change the Layout from constraint to linear Layout. Set the orientation to vertical; this arranges the UI components vertically in the Layout. Our app's primary widgets are Edittext, Button, and a RecyclerView.

Make sure all these widgets have IDs since they will be vital at a later stage. This is how our activity_main.xml file should look like.

```
<?xml version="1.0" encoding="utf-8"?><LinearLayout
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:orientation="vertical"
android:layout_height="match_parent"
tools:context=".MainActivity">       <EditText
android:id="@+id/titletxt"
```

```
android:layout_width="match_parent"

android:layout_height="wrap_content"

android:layout_marginTop="24dp"             android:ems="10"

android:textColor="#000"

android:inputType="textPersonName"

android:hint="Enter Information"

android:textAlignment="center"

app:layout_constraintBottom_toTopOf="@+id/button"

app:layout_constraintEnd_toEndOf="parent"

app:layout_constraintHorizontal_bias="0.497"

app:layout_constraintStart_toStartOf="parent"

app:layout_constraintTop_toTopOf="parent" />       <Button

android:id="@+id/button"

android:layout_width="wrap_content"

android:layout_height="wrap_content"

android:layout_marginBottom="40dp"

android:backgroundTint="@color/colorAccent"

android:layout_marginTop="20dp"

android:text="Submit"             android:layout_gravity="center"

app:layout_constraintBottom_toTopOf="@+id/recycler"

app:layout_constraintEnd_toEndOf="parent"

app:layout_constraintHorizontal_bias="0.498"
```

```
app:layout_constraintStart_toStartOf="parent" />

<androidx.recyclerview.widget.RecyclerView

android:id="@+id/recycler"

android:background="#E8DDDD"

android:layout_width="match_parent"

android:layout_height="match_parent"

app:layout_constraintBottom_toBottomOf="parent"

app:layout_constraintEnd_toEndOf="parent"

app:layout_constraintStart_toStartOf="parent"

/></LinearLayout>
```

Step 4 – Creating the item_view

Still on the Layout, we need to create the design of the element shown in the RecyclerView. Therefore, create a file named item.xml and add the code shown in the image below. The design is simple since the user can also access one attribute from the data class.

```
<?xml version="1.0" encoding="utf–8"?>

<androidx.cardview.widget.CardView

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

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

android:layout_width="match_parent"

android:layout_margin="10dp"

android:layout_height="wrap_content">             <RelativeLayout

android:layout_width="match_parent"

android:layout_height="100dp"

android:orientation="horizontal">                 <TextView

android:id="@+id/title"

android:layout_width="wrap_content"

android:layout_height="wrap_content"

android:layout_alignParentTop="true"

android:layout_alignParentBottom="true"

android:layout_marginTop="31dp"

android:layout_marginEnd="276dp"

android:layout_marginBottom="30dp"

android:padding="10dp"                         android:text="Hallo"

android:textColor="#000" />                    <ImageButton

android:id="@+id/delete"

android:layout_width="wrap_content"

android:layout_height="wrap_content"

android:layout_alignParentLeft="true"

android:layout_alignParentTop="true"
```

```
android:layout_alignParentEnd="true"

android:layout_alignParentBottom="true"

android:layout_marginLeft="328dp"

android:backgroundTint="#ffff"

android:layout_marginTop="22dp"

android:layout_marginEnd="16dp"

android:layout_marginBottom="35dp"

android:src="@drawable/ic_baseline_delete_24" />

</RelativeLayout>      </androidx.cardview.widget.CardView>
```

Step 5 – Create a RecyclerView Adapter

A RecyclerView adapter handles the binding of the item.xml layout to the RecyclerView. It also takes in a list of items and displays them to the user. The code for the RecyclerView adapter is shown below.

```
class NoteRecyclerAdapter(val viewModel: MainViewModel, val arrayList:

ArrayList<Blog>, val context: Context):

RecyclerView.Adapter<NoteRecyclerAdapter.NotesViewHolder>() {

override fun onCreateViewHolder(

        parent: ViewGroup,
        viewType: Int,      ):
NoteRecyclerAdapter.NotesViewHolder {
       var root =
LayoutInflater.from(parent.context).inflate(R.layout.item,parent,false)
return NotesViewHolder(root)
  }
 override fun onBindViewHolder(
holder: NoteRecyclerAdapter.NotesViewHolder, position: Int)
{           holder.bind(arrayList.get(position))      }
   override fun getItemCount(): Int {
   if(arrayList.size==0){
             Toast.makeText(context,"List is
empty",Toast.LENGTH_LONG).show()
        }else{
   }
   return arrayList.size
  }
 inner   class NotesViewHolder(private val binding: View) :
RecyclerView.ViewHolder(binding) {
     fun bind(blog: Blog){
           binding.title.text = blog.title
binding.delete.setOnClickListener {
```

```
                    viewModel.remove(blog)
notifyItemRemoved(arrayList.indexOf(blog))
             }
       }
 }
}
```

Step 6 – Creating the ViewModel

Create a package named ViewModel. Inside this folder, create a Kotlin class and name it MainViewModel. The class should extend the Android ViewModel. You might face an error if you failed to add lifecycle dependencies from Jetpack.

The MainViewModel will have a mutable livedata item that holds the array list. It's vital to use LiveData since it notifies the UI in case of any data change.
The MainViewModel code is shown below.

```
class MainViewModel: ViewModel() {

  var lst = MutableLiveData<ArrayList<Blog>>()

  var newlist = arrayListOf<Blog>()

 fun add(blog: Blog){

      newlist.add(blog)

          lst.value=newlist

   }

 fun remove(blog: Blog){

        newlist.remove(blog)

      lst.value=newlist

  }}
```

Step 7 – Create the ViewModel Factory

The purpose of a ViewModel factory is to instantiate the ViewModel. This prevents the app from crashing in case an activity is not found.

The code for our MainViewModelFactory is shown below.

```
class MainViewModelFactory(): ViewModelProvider.Factory{

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

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

```
        {

            return MainViewModel() as T

    }             throw IllegalArgumentException ("UnknownViewModel")

       }

    }
```

Step 8 – MainActivity (connecting the code)

We have created the model, ViewModel, ViewModelfactory, and RecyclerView. These components need to be instantiated in the MainActivity class for the application to work.

Start by declaring the RecyclerView and instantiating it. Set the layout manager for the RecyclerView to LinearLayoutManager. The MainActivity file contains three major methods; initialiseAdapter,observeData, and addData. the initialiseAdapter method assigns a ViewManager to the RecyclerView.

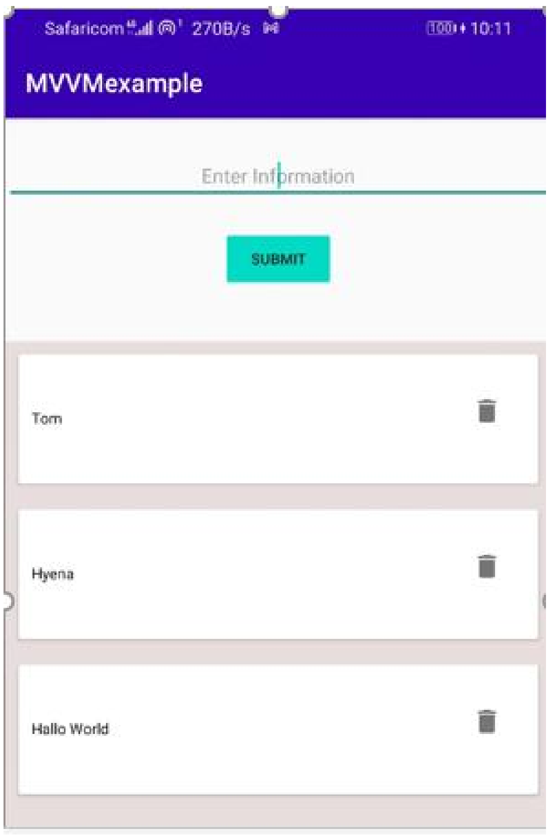
The observeData function looks for changes in the viewmodel and forwards them to the RecyclerAdapter.

The addData method takes in the user's input and updates the list in the ViewModel.

```
class MainActivity : AppCompatActivity() {

  private var viewManager = LinearLayoutManager(this)      private lateinit

   var viewModel: MainViewModel

   private lateinit var mainrecycler: RecyclerView

   private lateinit var but: Button

   override fun onCreate(savedInstanceState: Bundle?) {

  super.onCreate(savedInstanceState)

    setContentView(R.layout.activity_main)           mainrecycler =

    findViewById(R.id.recycler)

     val application = requireNotNull(this).application            val factory =

    MainViewModelFactory()

      viewModel =

    ViewModelProviders.of(this,factory).get(MainViewModel::class.java)

      but = findViewById(R.id.button)           but.setOnClickListener {

          addData()

     }           initialiseAdapter()

  }    private fun initialiseAdapter(){

      mainrecycler.layoutManager = viewManager           observeData()

  }
  fun observeData(){

          viewModel.lst.observe(this, Observer{

                    Log.i("data",it.toString())

              mainrecycler.adapter= NoteRecyclerAdapter(viewModel, it, this)

        })

  }

    fun addData(){

        var txtplce = findViewById<EditText>(R.id.titletxt)

          var title=txtplce.text.toString()

        if(title.isNullOrBlank()){

                Toast.makeText(this,"Enter

     value!",Toast.LENGTH_LONG).show()

          }else{

    var blog= Blog(title)

                viewModel.add(blog)

                  txtplce.text.clear()

     mainrecycler.adapter?.notifyDataSetChanged()

          }

 }

}
```

Step 9 – Results

If there were no errors in your code, it should compile and show the UI in the image below. Whatever you type in the EditText field should display in the recyclerview once you click the submit button.



Conclusion

MVVM architecture has made it easier to build complex applications. As shown, it’s easier to identify bugs due to the separation of business logic from the UI code. The architecture also prevents data loss during configuration changes. Ensure that all dependencies are present before using MVVM. This measure helps prevent runtime errors.

References

JetPack

MVC

Kotlin