|  |
| --- |
| MAP-06 |
| Complete Android Jetpack Masterclass |
| Research Project |

|  |
| --- |
| Sardaryan, Elina  12-31-2022 |

MAP-06

Related Technologies for Multiplatform Applications

**Research Project**

Student:

Elina Sardaryan

Udemy tutorial “Complete Android Jetpack Masterclass”.

The link for the course: <https://www.udemy.com/course/android-jetpack-masterclass/>

The course covers

• Android Jetpack suite

• ViewBinding and Animation(Splash Screen)

• MVVM (Model View ViewModel)

• Permissions

• Glide

• ROOM Database

• LiveData, Lifecycles and ViewModels

• Navigation Component, Navigation Graph, Safe Args

*Project #1* **1\_FavDish**

1. Open Android Studio and create a new project using **Bottom Navigation Activity.**
2. Name the application as **FavDish.**
3. Select language as **Kotlin** and click the finish button.
4. Now you are done with the project creation and as you can see there are many auto-added files by Android studio already. By default we have one main UI with three fragments(dashboard, home, notification) with their activities already included in the project. We will work with them later. In this current project we are going to set up the **color themes of the application.**
5. Go to res -> values -> themes and you will notice **themes.xml** with day and night combination. Here we will set the same color combination.
6. To choose a color combination you can go to the link <https://material.io/resources/color/#!/?view.left=0&view.right=0> and select the color combination that you want.
7. For this project we will use the primary color **Green 700** with all the combination shades. Add them to our application.
8. For secondary color we will use **Green 300** and add all the combinations to the **colors.xml** file. (Step 1 – for both primary and secondary colors)

<color name="primary\_color">#388E3C</color>  
 <color name="primary\_dark\_color">#00600F</color>  
 <color name="primary\_light\_color">#6ABF69</color>  
   
<color name="secondary\_color">#81C784</color>  
 <color name="secondary\_dark\_color">#519657</color>  
 <color name="secondary\_light\_color">#B2FAB4</color>

1. In **themes.xml** (day) we will change primary and secondary colors, as well as status bar color. (Step 2– for both primary and secondary colors)

<item name="colorPrimary">@color/primary\_color</item>  
 <item name="colorPrimaryVariant">@color/primary\_dark\_color</item>

<item name="colorSecondary">@color/secondary\_color</item>  
 <item name="colorSecondaryVariant">@color/secondary\_dark\_color</item>

<item name="android:statusBarColor" tools:targetApi="l">?attr/colorPrimaryVariant</item>

1. In **themes.xml** (night) we will change primary and secondary colors, as well as status bar color. (Step 3– for both primary and secondary colors)

<item name="colorPrimary">@color/primary\_light\_color</item>  
 <item name="colorPrimaryVariant">@color/primary\_color</item>

<item name="colorSecondary">@color/secondary\_light\_color</item>  
 <item name="colorSecondaryVariant">@color/secondary\_color</item>

<item name="android:statusBarColor" tools:targetApi="l">?attr/colorPrimaryVariant</item>

*Project # 2*  **2\_ActivityLifecycle**

As a user navigates through, out of, and back to your app, the Activity instances in your app transition through different states in their lifecycle. The Activity class provides several callbacks that allow the activity to know that a state has changed: that the system is creating, stopping, or resuming an activity, or destroying the process in which the activity resides.

To navigate transitions between stages of the activity lifecycle, the Activity class provides a core set of six callbacks: onCreate(), onStart(), onResume(), onPause(), onStop(), and onDestroy(). The system invokes each of these callbacks as an activity enters a new state.



1. Open Android Studio and create a new project using **Empty Activity**,

2. Name the application as **ActivityLifecyle**.

3. Select language as **Kotlin** and click the finish button.

4. Override all the lifecycle methods and print the log in it.

override fun onStart() {  
 super.onStart()  
 Log.e("onStart method", "is called...")  
 }

override fun onResume() {  
 super.onResume()  
 Log.e("onResume method", "is called...")  
 }

override fun onPause() {  
 super.onPause()  
 Log.e("onPause method", "is called...")  
 }

override fun onStop() {  
 super.onStop()  
 Log.e("onStop method", "is called...")  
 }

override fun onRestart() {  
 super.onRestart()  
 Log.e("onRestart method", "is called...")  
 }

override fun onDestroy() {  
 super.onDestroy()  
 Log.e("onDestroy method", "is called...")  
 }

5. Run and see which log is printed at what time.

6. To see the Logs, go to Logcat and choose Error as we used Log.e

*Project # 3* **3\_PassingDataToAnotherActivityWithPutExtra**

1. Continue working on **ActivityLifecyle** project.
2. Add a Button in activity\_main.xml.
3. Add id for the TextView and for Button, and make the design adjustments
4. Access the button and add click event to it in Main Activity

val btnSubmit = findViewById<Button>(R.id.*btn\_submit*)  
 btnSubmit.setOnClickListener **{}**

1. Create an Another Activity to launch it via Intent and to pass the data between two activities.
2. Add the TextView to the Another Activity to just see that it is launched.
3. Launch the Another Activity and pass the data using putExtra. Write this code in the button event listener.

val intent = Intent(this@MainActivity, AnotherActivity::class.*java*).*apply* **{**  
putExtra("key1", "Value1")  
 putExtra("key2", "Value2")  
 // You can add as many params as you want.  
 }

startActivity(intent)

1. Get the data in Another Activity from Main Activity and print it in the log.

val keyValue1 = *intent*.getStringExtra("key1")  
 Log.i("value 1", "$keyValue1")  
 val keyValue2 = *intent*.getStringExtra("key2")  
 Log.i("value 2", "$keyValue2")

*Project # 4* **4\_FavDish – SplashScreen**

1. Use the project **FavDish** we created before.
2. Create a new package name as “activities.”
3. Create a new empty activity as Splash Screen with the name SplashActivity.
4. We are going to implement the **ViewBinding** concept.
5. Add the string value “Fav Dish” to the strings.xml file

<string name="splash\_screen\_title">Fav Dish</string>

1. Design the splash screen layout with one TextView and put string value to text.

android:text="@string/splash\_screen\_title"

1. Enable the ViewBinding in build.gradle(:app)

buildFeatures **{**  
viewBinding true  
 }

NOTE\*\*\* - when you create a new project using **Bottom Navigation Activity**, **ViewBinding** is already enabled by default.

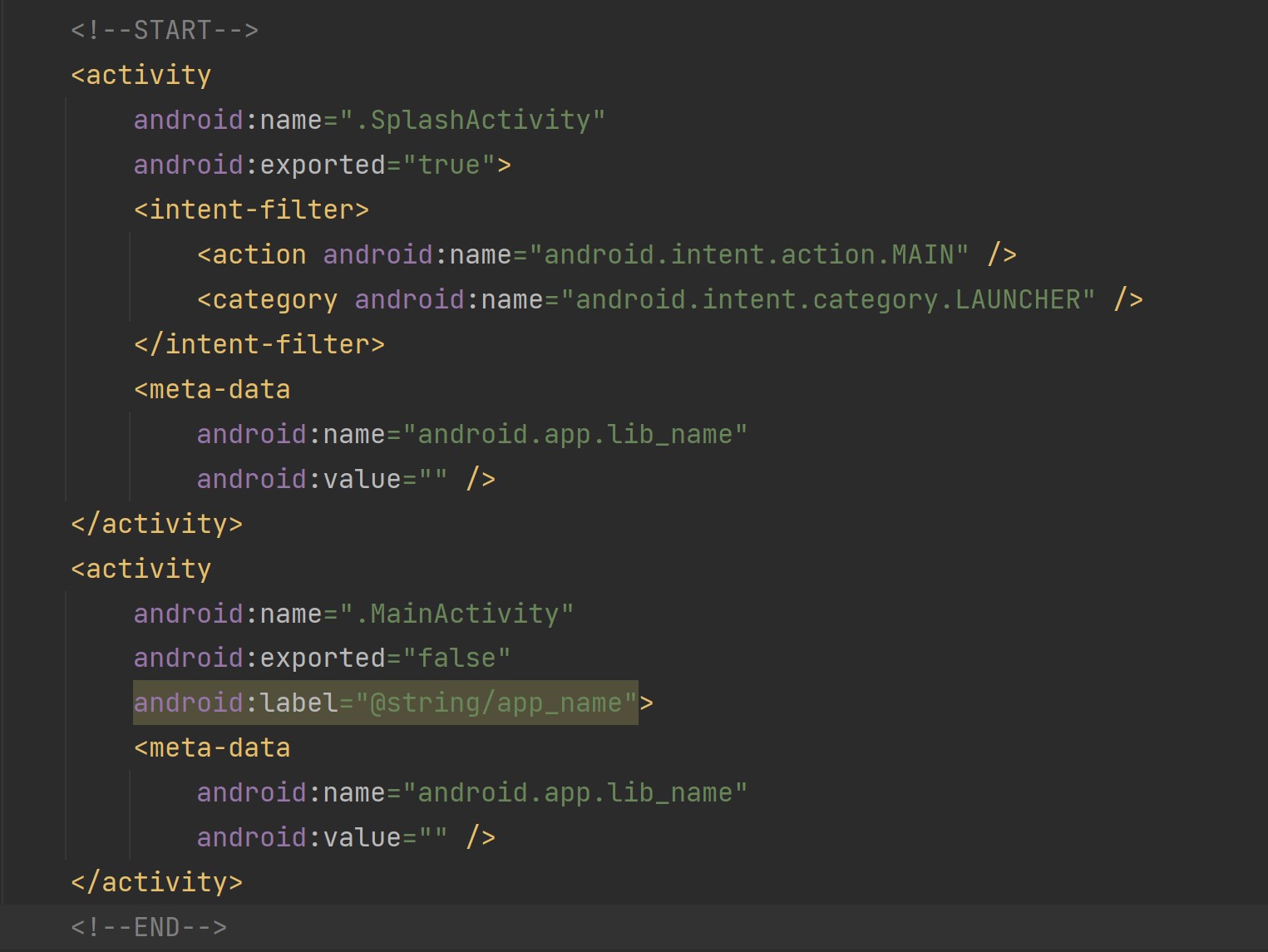
1. Access the XML layout file using the ViewBiding.

val splashBinding: ActivitySplashBinding = ActivitySplashBinding.inflate(*layoutInflater*)

1. Update the content view using the ViewBinding

setContentView(splashBinding.*root*)

1. Create the SplashActivity as the launcher activity instead of MainActivity. So, go to AndroidManifest.xml and take <intent-filter> tag with its content from MainActivity and put it in SplashActivity, and also put exported as true in SplashActivity.



1. Run the application and see the changes.
2. As you can see the launcher screen is changed and it is stuck on the splash screen. We will animate and redirect it to the main screen in the next project.

*Project # 5* **5\_FavDish - AnimatedSplashScreen**

1. Continue with the previous project **FavDish** where we created splash screen for our application.
2. Make the Splash Activity as a full screen view that means hide the Status Bar. So, first in SplashActivity we need to type:

if (Build.VERSION.*SDK\_INT* >= Build.VERSION\_CODES.*R*) {  
 *window*.*insetsController*?.hide( WindowInsets.Type.statusBars())  
 } else {  
 @Suppress("DEPRECATION")  
 *window*.setFlags(  
 WindowManager.LayoutParams.*FLAG\_FULLSCREEN*,  
 WindowManager.LayoutParams.*FLAG\_FULLSCREEN*  
)  
 }

Second, go to themes.xml and type:

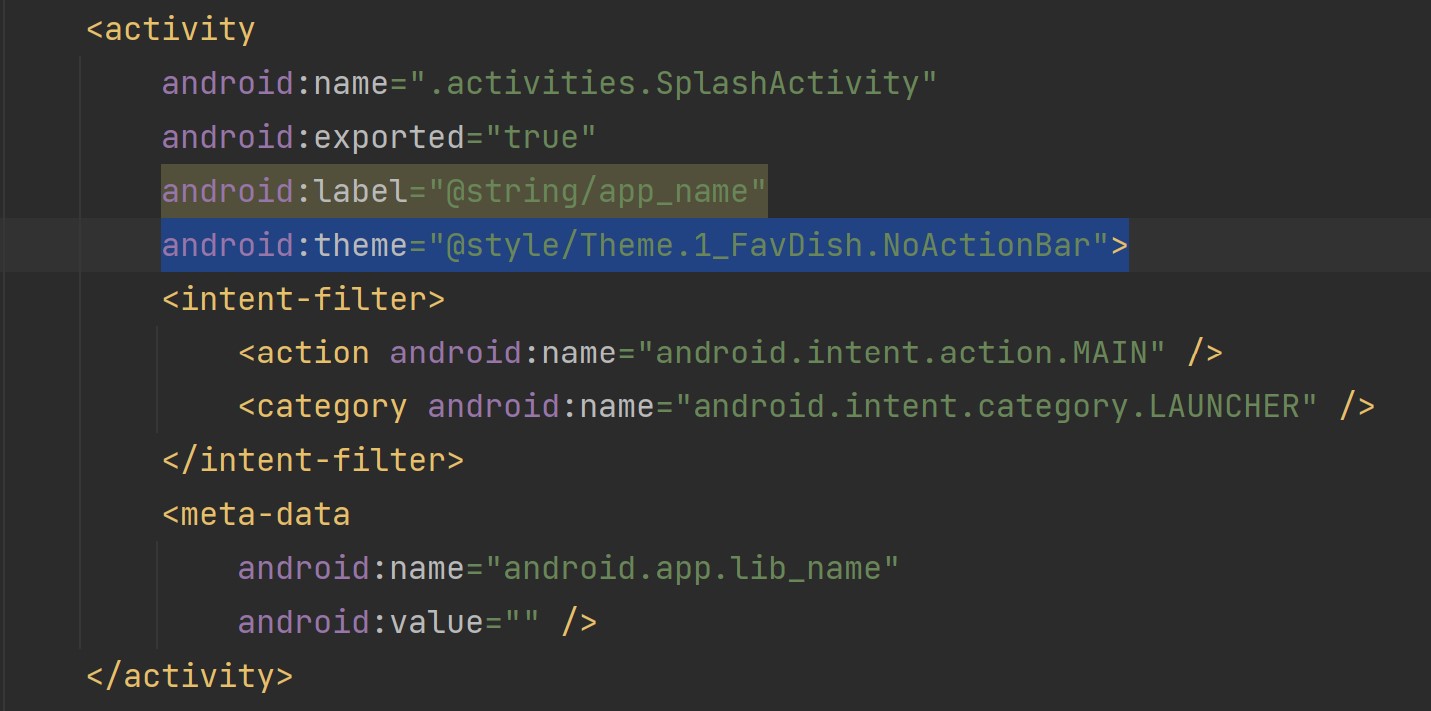
<style name="Theme.FavDish.NoActionBar">  
 <item name="windowActionBar">false</item>  
 <item name="windowNoTitle">true</item>  
 </style>

<style name="Theme.FavDish.AppBarOverlay" parent="ThemeOverlay.AppCompat.Dark.ActionBar" />

<style name="Theme.FavDish.PopupOverlay" parent="ThemeOverlay.AppCompat.Light" />

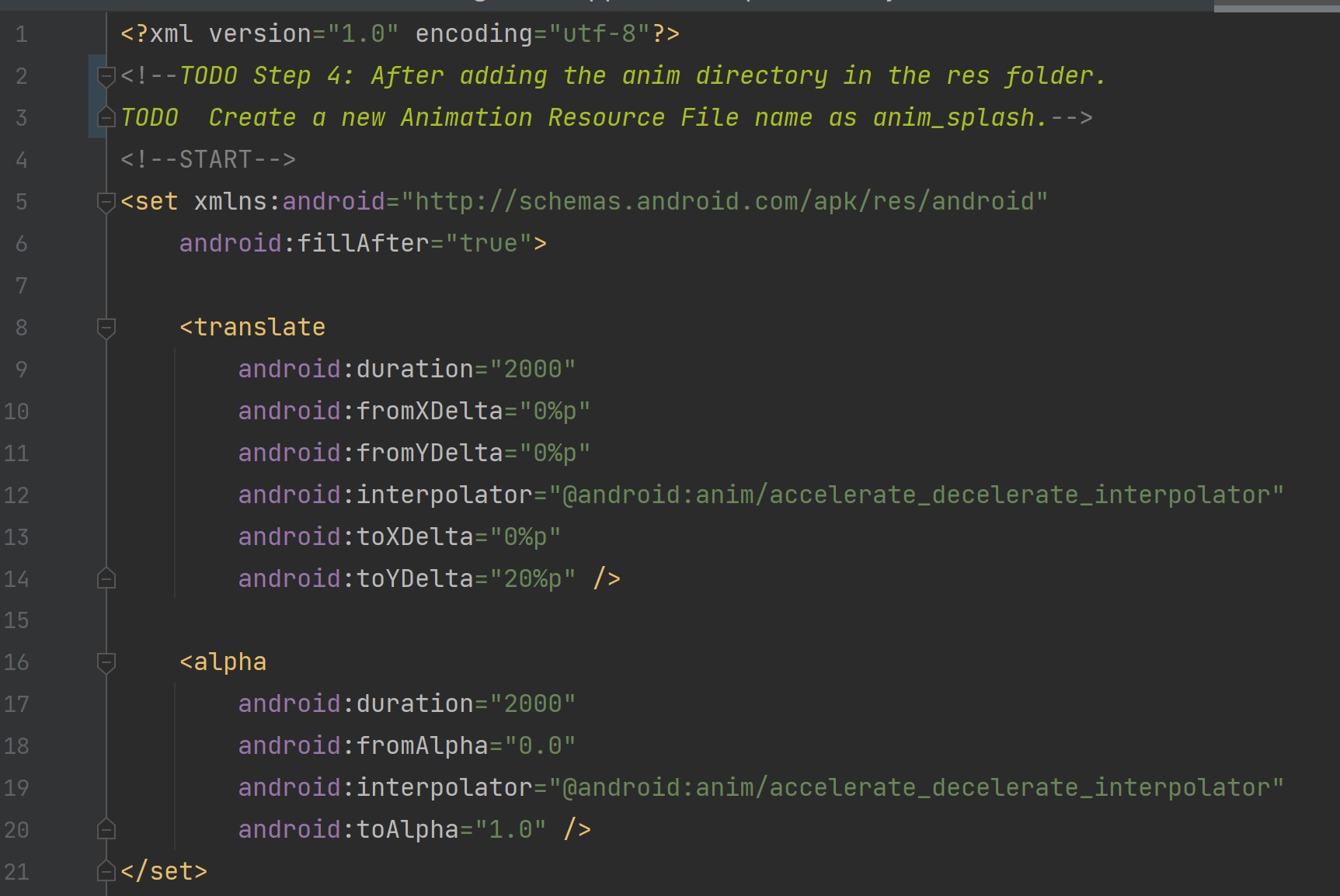
Third, go to AndroidManifest.xml and update the splash activity theme, so type:

android:theme="@style/Theme.FavDish.NoActionBar"



1. After updating the theme of SplashActivity in the manifest.xml file. Add the anim resource directory in the res folder.

1. After adding the “**anim”** directory in the res folder. Create a new Animation Resource File name as “**anim\_splash**”.



1. In Splash Activity Create an access variable for Animation as below.

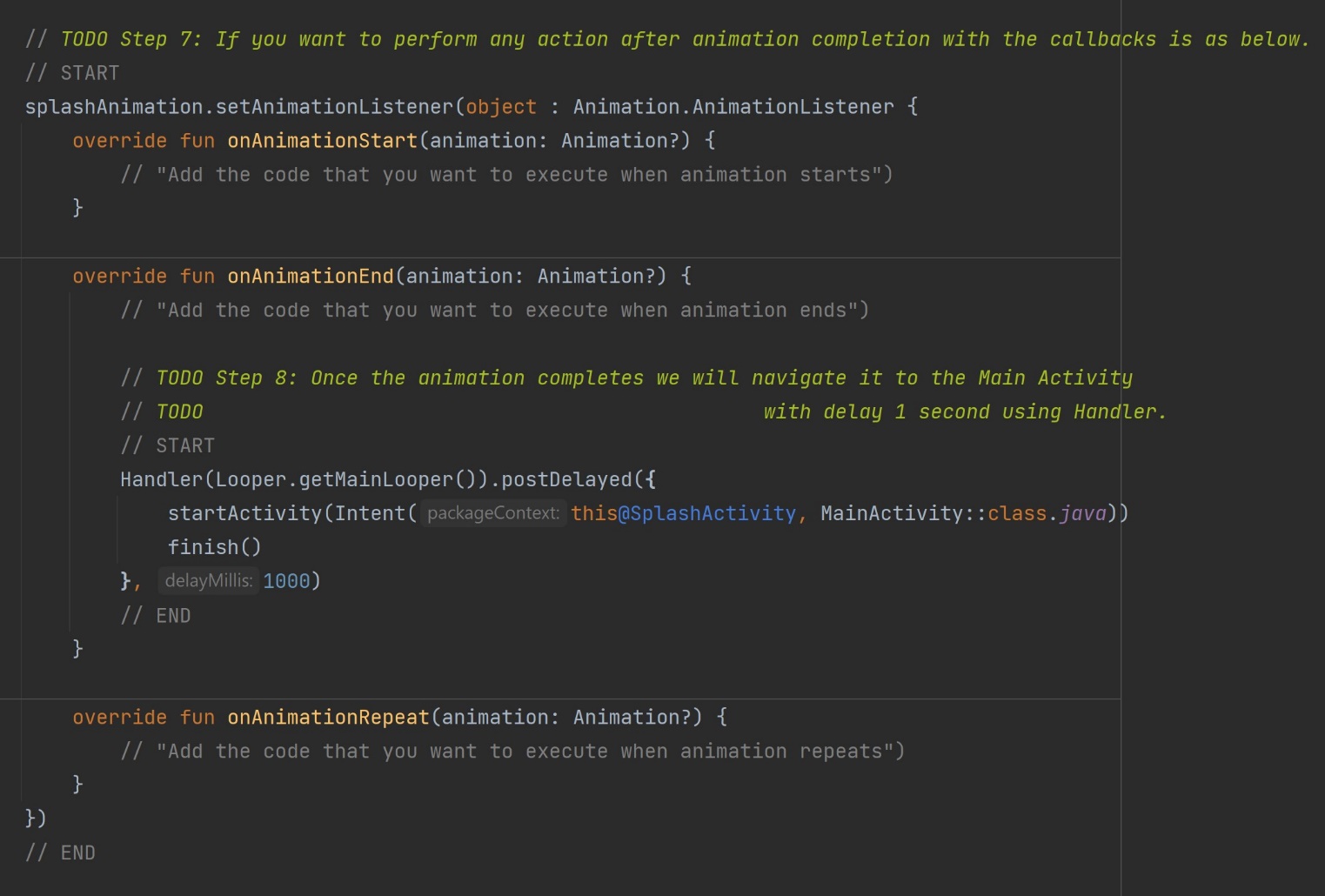
val splashAnimation = AnimationUtils.loadAnimation(this@SplashActivity, R.anim.*anim\_splash*)

1. Apply the animation to TextView

splashBinding.tvAppName.*animation* = splashAnimation

Note\*\*\* - **tvAppName** is id of TextView in **activity\_splash.xml** layout.

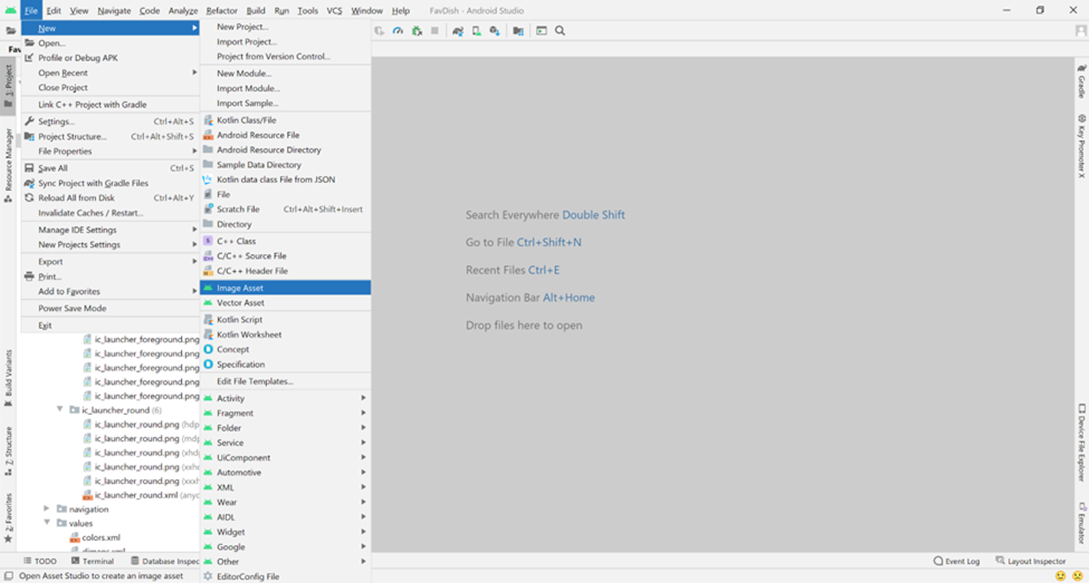
1. Once the animation is completed, we will navigate it to the Main Activity with delay 1 second using Handler. We will use **setAnimationListener** as below:

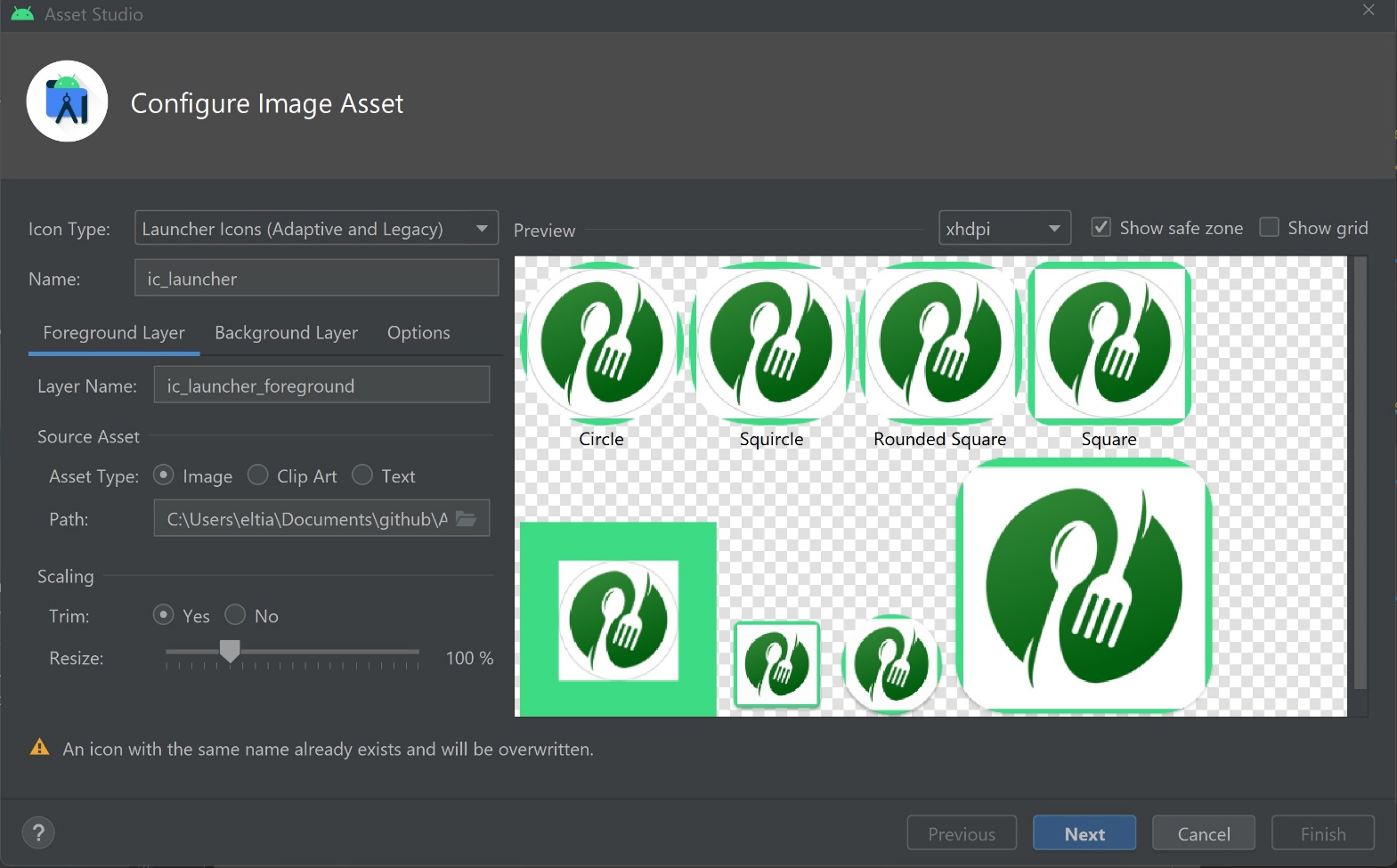


1. Run the application and see the changes.
2. You can get the animations online that you want or create your own as per your requirement. A reference link <https://www.raywenderlich.com/2785491-android-animation-tutorial-with-kotlin#toc-anchor-001>.
3. Next, we are going to create app icon using Android Studio.

First, download an icon you would like to use as your app icon.

Second, click anywhere in the Project and then go to **File -> New -> Image Asset**.



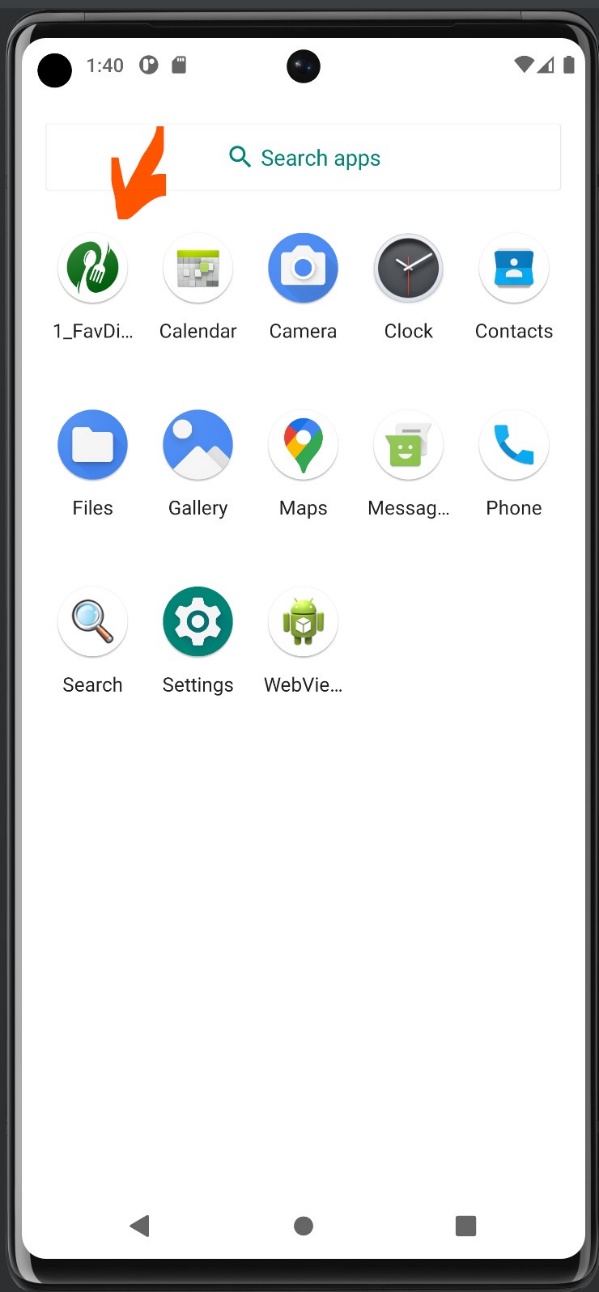
Third, choose the path and select the image you want to use.

Fourth, you can choose to trim it or not. In this project we chose trim.

Fifth, go to Background Layer and choose color white “#FFFFFF”. You can also use another image as background, but in this project, we will choose only the color white.

Sixth, go to Next and then Finish.

1. You will have the below app icon.

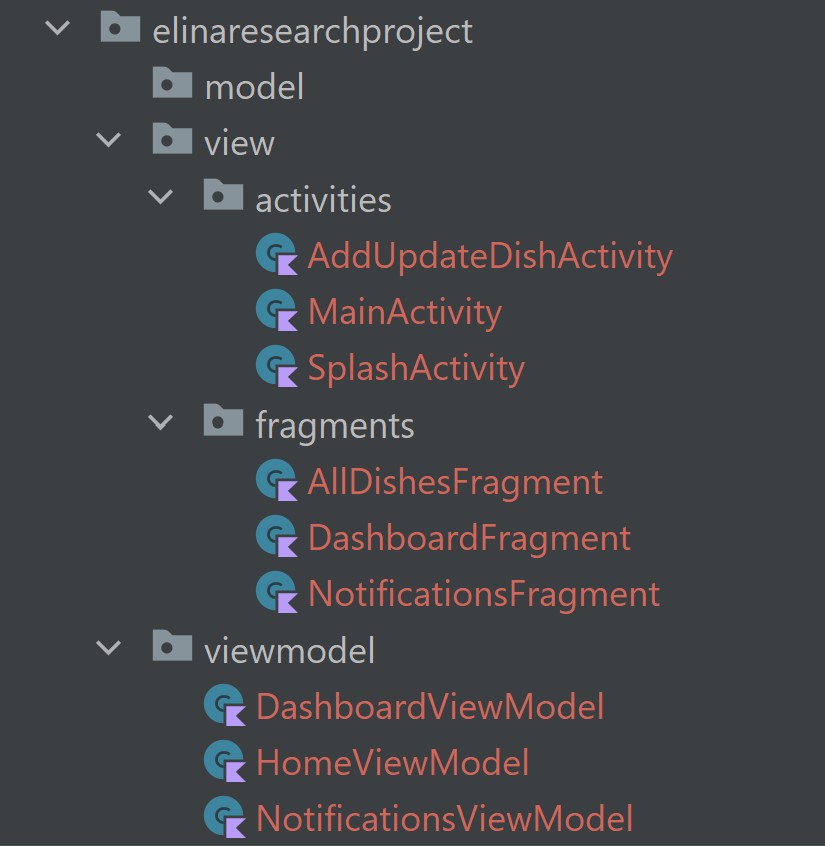


*Project # 6* **6\_ElinaResearchProject**

In this project, we will refactor the folder structure in the java package.

1. Rename the **“ui”** package to **“view”.**
2. Move the **MainActivity.kt** to the “**activities”** package.
3. Create new packages as “**model”** and “**viewmodel”**.
4. Move the “**activities”** package to the **“view”** package.
5. Move all the view models as **DashBoardViewModel.kt**, **HomeViewModel.kt**, and **NotificationViewModel.kt** to the “**viewmodel”** package.
6. Create a new package as “**fragments”** in the “**view”** package.
7. Move all the fragments to it and delete the respective packages.
8. Now, create a new empty activity as **AddUpdateDishesActivity.kt** in the “**activities”** package.

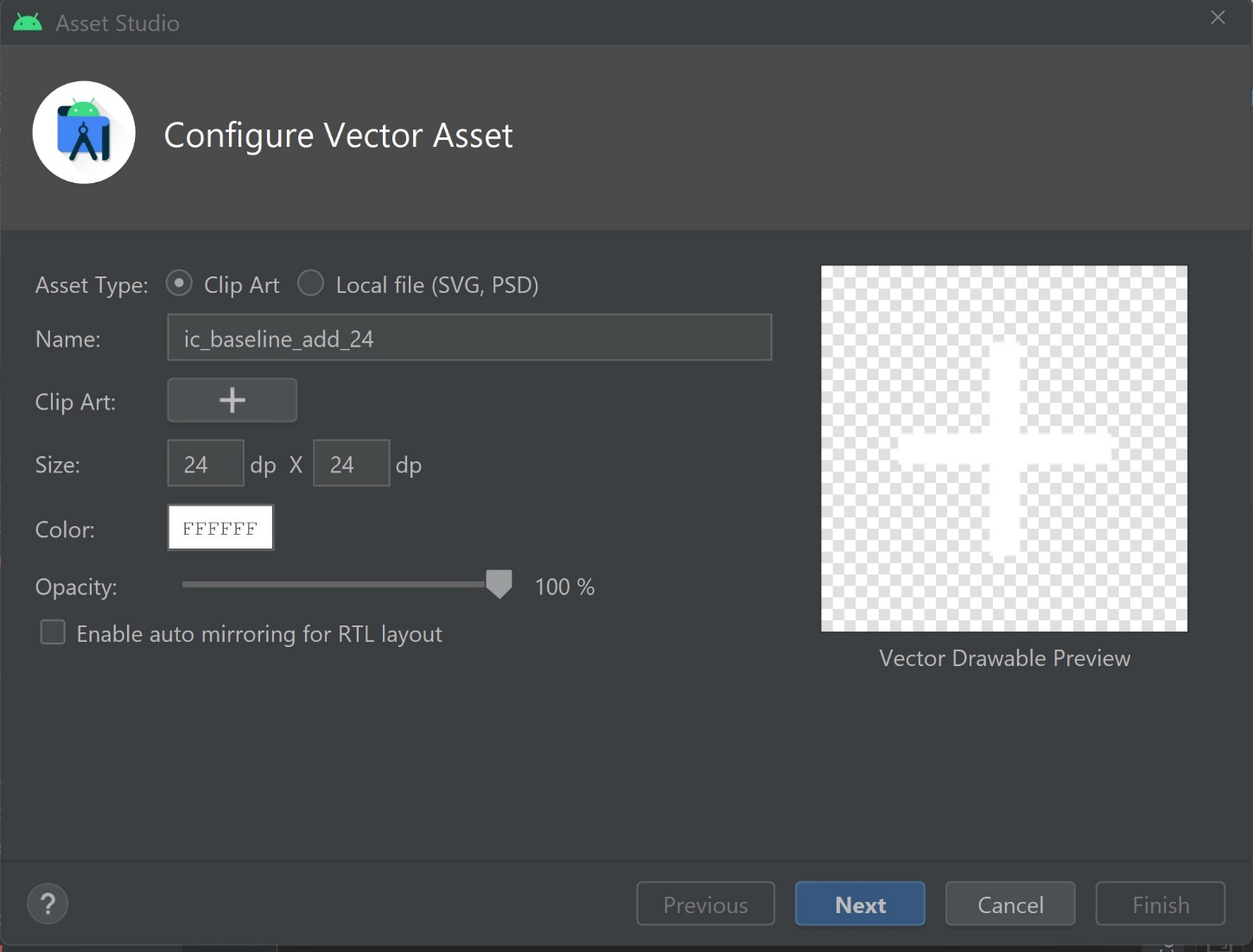
The structure of the project now will be as below (MVVM):



1. Make the style of **AddUpdateDishesActivity** activity with no action bar and also add alabel “Add Dish” in **AndroidManifest.xml**. Update the activity tag as below:

android:label="Add Dish"  
android:theme="@style/Theme.ElinaResearchProject.NoActionBar"

1. Rename **HomeFragment.kt** as **AllDishesFragment.kt**.
2. Rename the **fragment\_home.xml** file as **fragment\_all\_dishes.xml**.
3. Create a new icon and add it in drawable. **New -> Vector Asset...**



1. Name it as “**ic\_add**”.
2. Change the color to white “**FFFFFF**”.
3. Click **Next** and then **Finish**.
4. When add icon is clicked, a new activity is launched. For now, it will be a blank activity, which we will modify in the next steps.
5. Create a menu file as **menu\_all\_dishes.xml** and add the item as below.

<item android:id="@+id/action\_add\_dish"  
 android:icon="@drawable/ic\_add"  
 android:title="@string/action\_add\_dish"  
 app:showAsAction="always"/>

1. Add the code below in **strings.xml** file.

<string name="action\_add\_dish">Add Dish</string>

1. Override the **onCreate** function and enable **setHasOptionMenu** to add the action menu to Fragment.

override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 setHasOptionsMenu(true)  
}

1. Override the **onCreateOptionMenu** and **onOptionsItemSelected** methods and launch the **AddUpdateDishActivity** on selection.

override fun onOptionsItemSelected(item: MenuItem): Boolean {  
  
 when (item.*itemId*) {  
 R.id.*action\_add\_dish* -> {  
 // requireActivity() will give us the fragment's activity, similar to "this"  
 startActivity(Intent(requireActivity(), AddUpdateDishActivity::class.*java*))  
 return true  
 }  
 }  
 return super.onOptionsItemSelected(item)  
}

*Project # 7*  **7\_ElinaResearchProject**

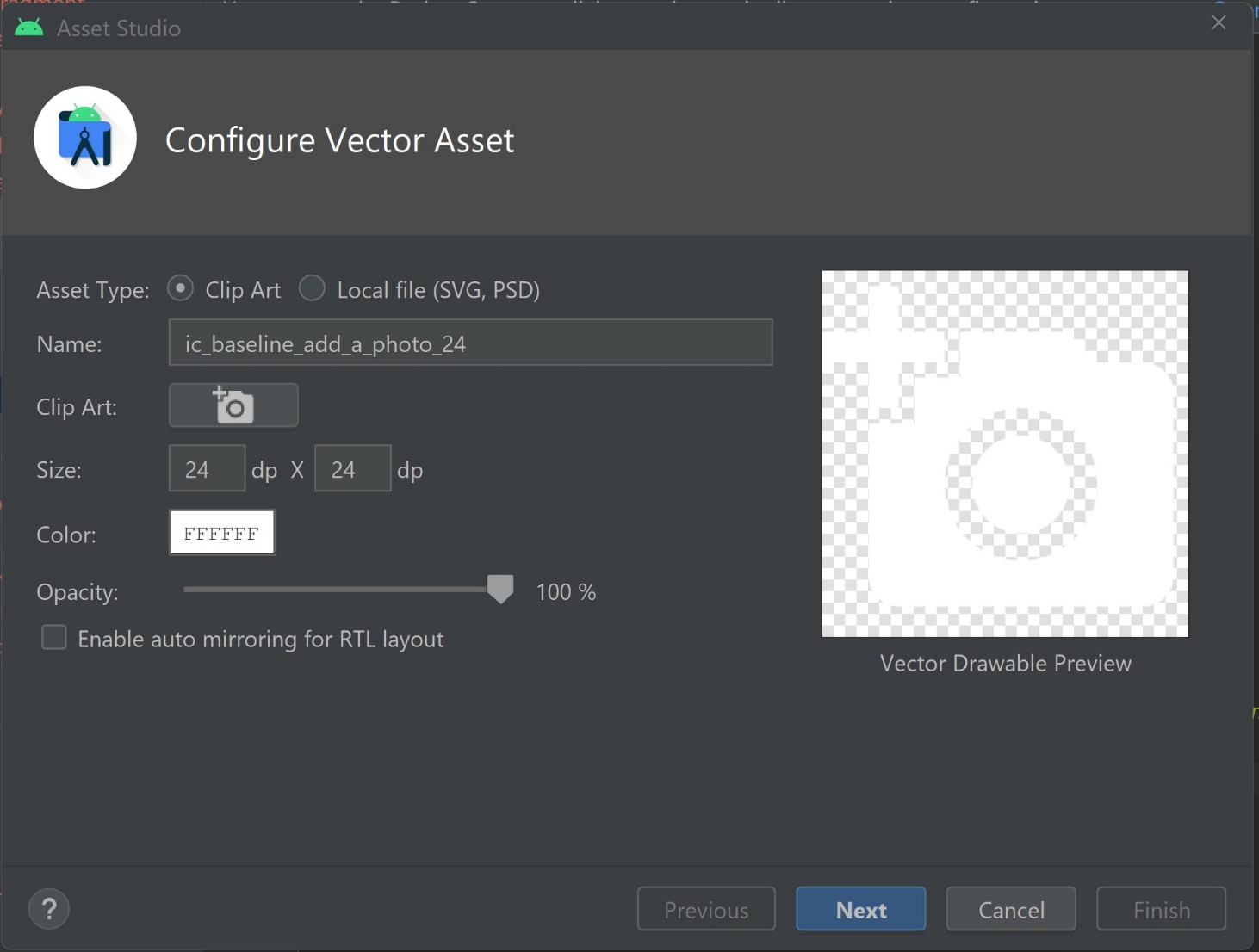
In this project we will design the **activity\_add\_update\_dish.xml** layout file using **ConstraintLayou**t, **ScrollView**, and **FrameLayout**.

1. Add the library for designing the better layout using dimensions in **build.gradle**(:app)

implementation 'com.intuit.sdp:sdp-android:1.1.0'

For more information, visit *https://github.com/intuit/sdp*

1. Create a new vector asset in drawable, which will be an icon to add a photo. So, right click on **drawable -> New -> Vector Asset**



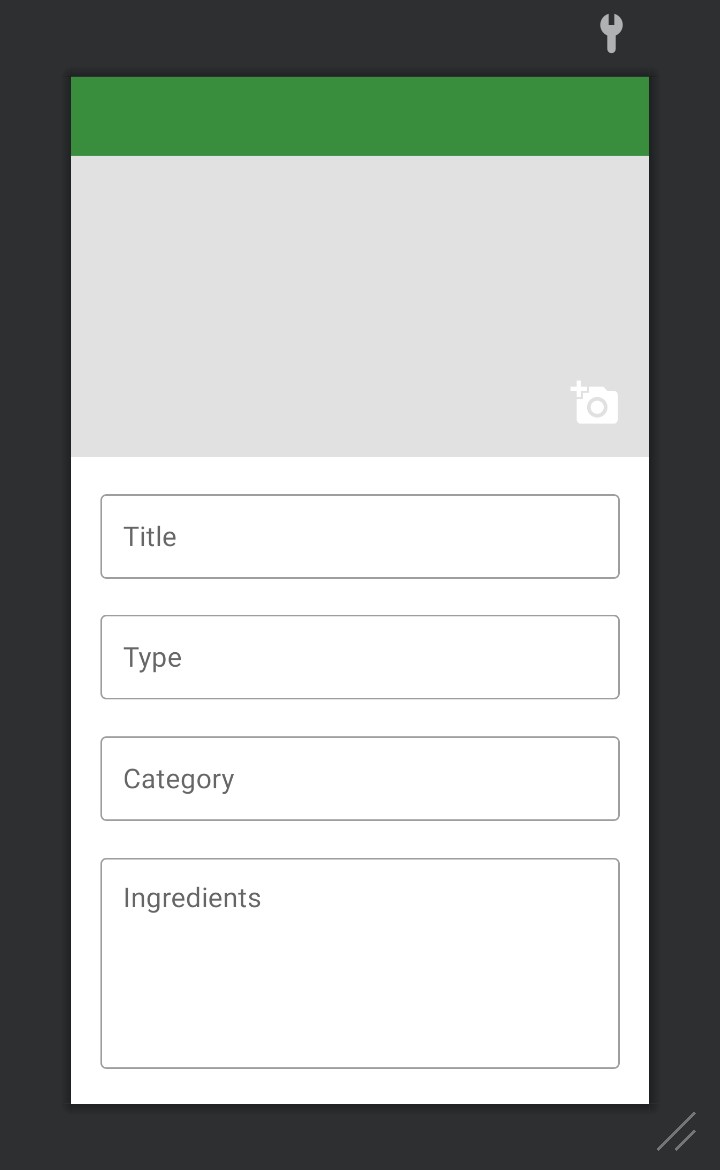
1. Find from **Clip Art** “add\_a\_photo” vector
2. Change the name to “**ic\_add\_a\_photo**”
3. Change the color to white “**FFFFFF**”
4. Click on **Next** and then **Finish**
5. Add new string values in **strings.xml** as below:

<string name="image\_contentDescription">image</string>  
<string name="lbl\_title">Title</string>  
<string name="lbl\_type">Type</string>  
<string name="lbl\_category">Category</string>  
<string name="lbl\_ingredients">Ingredients</string>  
<string name="lbl\_cooking\_time\_in\_minutes">Cooking Time In Minutes</string>  
<string name="lbl\_direction\_to\_cook">Direction To Cook</string>  
<string name="lbl\_add\_dish">ADD DISH</string>

1. Add new color values in **colors.xml** as below:

<color name="dish\_image\_background">#E1E1E1</color>  
<color name="blue\_grey\_700">#37474f</color>  
<color name="grey\_900">#212121</color>

1. Create new layout using constraints, views (widgets) and view groups(layouts) as below:



Note \*\*\* - use as a guide activity\_add\_update\_dish.xml

1. Add the “**configChanges”** attribute in **AndroidManifest.xml** to allow layout to rotate it horizontally in the application if user rotates it. This will prevent the app from restarting when the screen orientation changes.

android:configChanges="orientation"

1. In **AddUpdateDishActivity.kt** create a global variable for layout ViewBinding

private lateinit var mBinding: ActivityAddUpdateDishBinding

1. Initialize the layout **ViewBinding** variable and set the **contentView**.

mBinding = ActivityAddUpdateDishBinding.inflate(*layoutInflater*)  
setContentView(mBinding.*root*)

1. Create a function to setup the ActionBar in **AddUpdateDishActivity.kt**

private fun setupActionBar() {  
 // action bar in xml -> id is toolbar\_add\_dish\_activity  
 // we are assigning this bar using default method setSupportActionBar  
 setSupportActionBar(mBinding.toolbarAddDishActivity)  
  
 // this will allow us to have back button  
 *supportActionBar*?.setDisplayHomeAsUpEnabled(true)  
  
 // add click listener to back button  
 mBinding.toolbarAddDishActivity.setNavigationOnClickListener**{**

onBackPressed()

**}**  
**}**

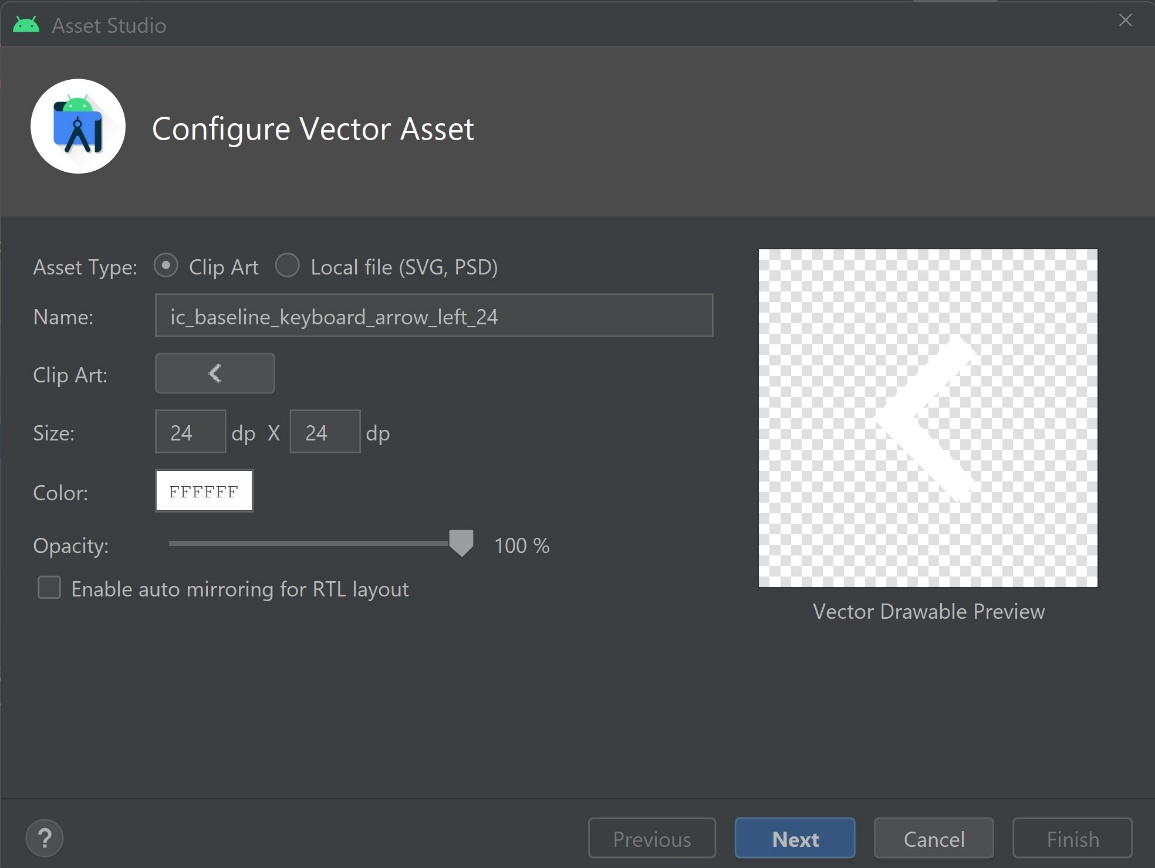
1. Call the method of **setupActionBar** in *onCreate* in **AddUpdateDishActivity.kt**

setupActionBar()

*Project # 8*  **8\_ElinaResearchProject**

In this project we will generate and add the back arrow icon for the action bar for the **AddUpdateDishesActivity** and assign the click event for ImageView.

1. Generate and add the back arrow icon for action bar home back icon. Right click on **drawable -> New -> Vector Asset**



1. Find from **Clip Art** “ic\_arrow\_back” vector
2. Change the name to “**ic\_arrow\_back**”
3. Change the color to white “**FFFFFF**”
4. Click on **Next** and then **Finish**
5. Replace the back arrow icon of the action bar. In **AddUpdateDishActivity.kt** in replace back button in **setupActionBar** method as below:

*supportActionBar*?.setHomeAsUpIndicator(R.drawable.*ic\_arrow\_back*)

1. Implement the View.OnClickListener for **AddUpdateDishActivity** .
2. Override the onClick listener method.

override fun onClick(v: View) {}

1. Perform the action when user clicks on the *iv\_add\_dish\_image* and show a Toast message.

override fun onClick(v: View) {  
  
 when (v.*id*) {  
 R.id.*iv\_add\_dish\_image* -> {  
 Toast.makeText(  
 this,  
 "You have clicked on the ImageView.",  
 Toast.*LENGTH\_SHORT*  
).show()  
 return  
 }  
 }  
}

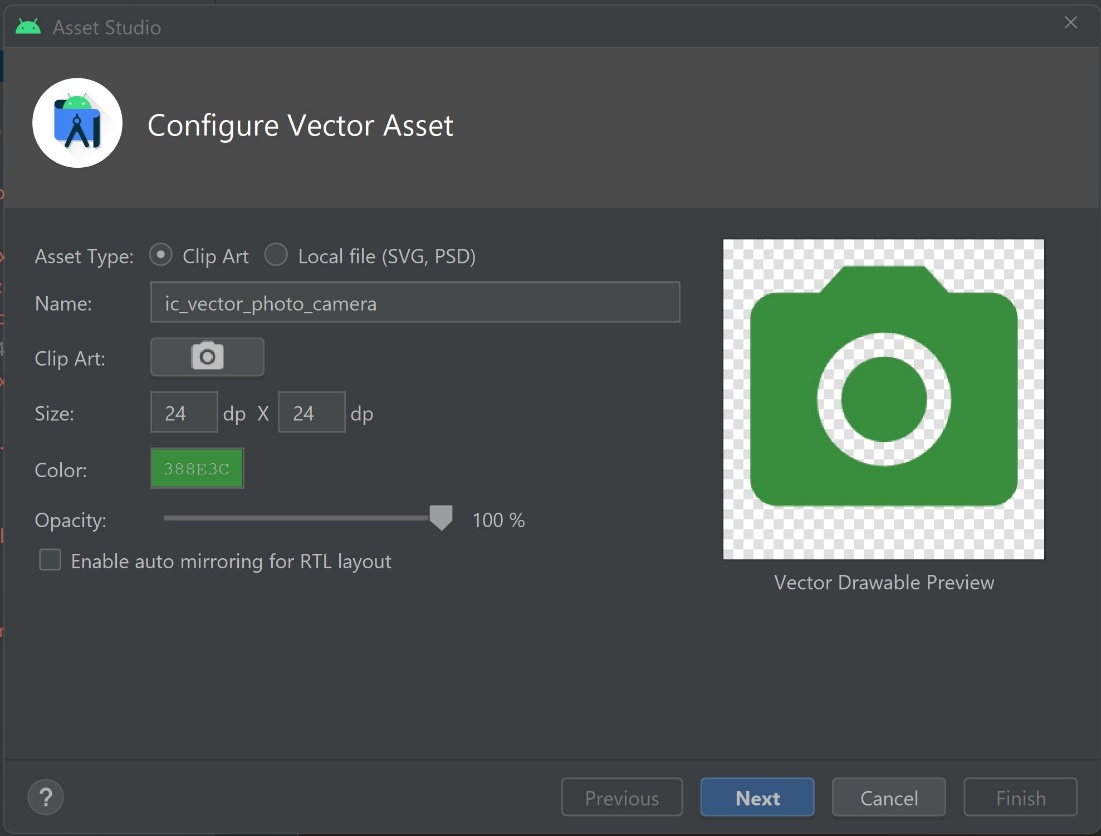
1. Assign the click event to the image button in onCreate method of **AddUpdateDishActivity**.

mBinding.ivAddDishImage.setOnClickListener(this)

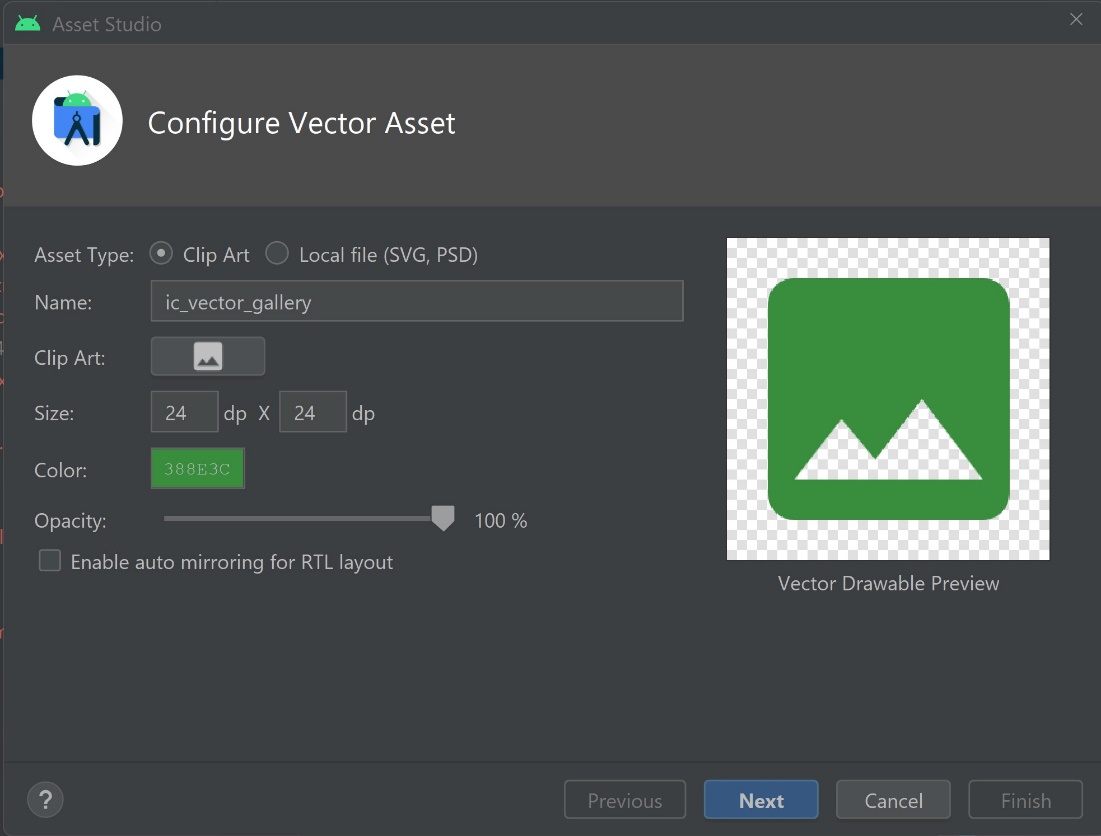
*Project # 9*  **9\_ElinaResearchProject**

In this project we will implement *custom dialog* for image selection i.e either from **Camera** or **Gallery**.

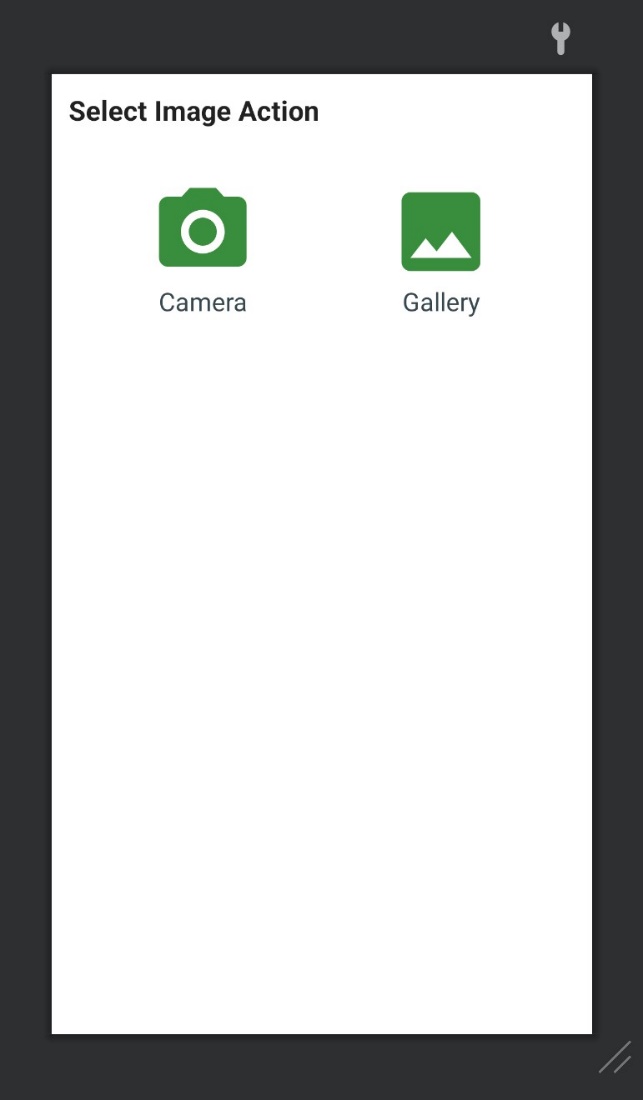
1. Generate and add vector icon in drawable. Right click on **drawable -> New -> Vector Asset**



1. Find from **Clip Art** “camera alt” vector
2. Change the name to “**ic\_vector\_photo\_camera**”
3. Change the color into “#388E3C”
4. Click on **Next** and then **Finish**
5. Generate and add another vector icon in drawable. Right click on **drawable -> New -> Vector Asset**



1. Find from **Clip Art** “image” vector
2. Change the name to “**ic\_vector\_gallery**”
3. Change the color into “#388E3C”
4. Click on **Next** and then **Finish**
5. Design a custom dialog using the constraint layout. (See dialog\_custom\_image\_selection.xml)



1. Add the string values to the strings.xml file.

<string name="title\_select\_image\_action">Select Image Action</string>  
<string name="lbl\_camera">Camera</string>  
<string name="lbl\_gallery">Gallery</string>

1. Create a function customImageSelectionDialog() to launch a custom dialog.

private fun customImageSelectionDialog() {  
 val dialog = Dialog(this)  
  
 val binding: DialogCustomImageSelectionBinding = DialogCustomImageSelectionBinding.inflate(*layoutInflater*)  
  
 /\*Set the screen content from a layout resource.  
 The resource will be inflated, adding all top-level views to the screen.\*/  
 dialog.setContentView(binding.*root*)  
  
 // *Assign the click for Camera and Gallery. Show the Toast message for now.*  
 binding.tvCamera.setOnClickListener **{**  
Toast.makeText(this, "You have clicked on the Camera.", Toast.*LENGTH\_SHORT*).show()  
 dialog.dismiss()  
 **}**  
  
binding.tvGallery.setOnClickListener **{**  
Toast.makeText(this, "You have clicked on the Gallery.", Toast.*LENGTH\_SHORT*).show()  
 dialog.dismiss()  
 **}**  
  
 //Start the dialog and display it on screen.  
 dialog.show()  
}

1. Replace the Toast Message in onClick() with the customImageSelectionDialog() function.

*Project # 10*  **10\_ElinaResearchProject**

In this project we will implement the functionality of runtime permissions for **Camera** and **Storage** using the third-party library **Dexter** <https://github.com/Karumi/Dexter>.

Dexter is a third party library, which is amazing when it comes to requesting permissions at runtime, as it simplifies the process significantly.

We are going to implement the runtime request for permission, because if you want to

use any data from your phone or want to use the camera, you need to ask the user for permission as a developer.

We will be using a third-party library. This reduces our lines of code and we will learn to implement the third-party library.

In the permissions demo, we will show how to ask the runtime permission manually without using a third-party library.

A reference link: <https://www.raywenderlich.com/9577211-scoped-storage-in-android-10-getting-started>

1. Go to **AndroidManifest.xml** and add a couple of lines of code, because whenever you ask for permissions, you need to add users permission here. We want to be able to write to the external storage and also read from it. We write the maximum SDK version of 28, because afterwards, it's not really necessary for the permissions. It depends on the version of the device that the user is using, but with this, we're making sure that it's going to work perfectly on all the devices.<uses-permission android:name="android.permission.READ\_EXTERNAL\_STORAGE" />  
   <!--For WRITE EXTERNAL STORAGE warning you can have a look at this article I hope it will clear your doubt.  
    https://www.raywenderlich.com/9577211-scoped-storage-in-android-10-getting-started -->  
   <uses-permission  
    android:name="android.permission.WRITE\_EXTERNAL\_STORAGE"  
    android:maxSdkVersion="28" />

After external storage, we will also ask for the permission for the camera as well.

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

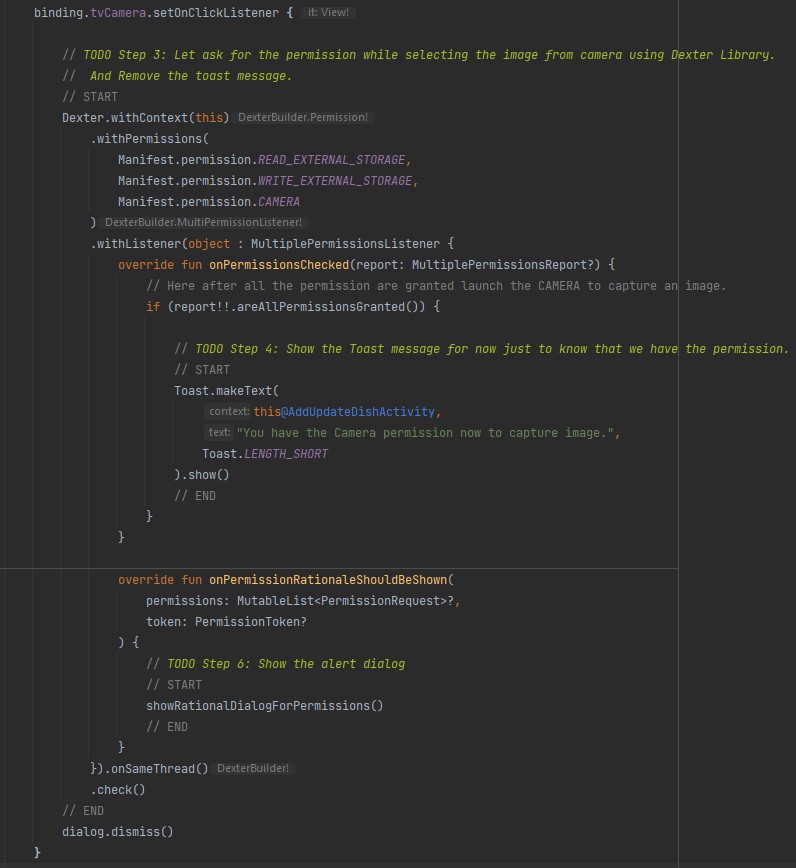
**Note \*\*\*\*** - this is not the actual permission request that we write in **AndroidManifest.xml**. Here in the Android manifest file, we are saying that we want to request for those permissions at one point in our application and that we will do it later on in our application.

1. Add the **Dexter** runtime permissions library in **build.gradle** (:app) and synchronize it.

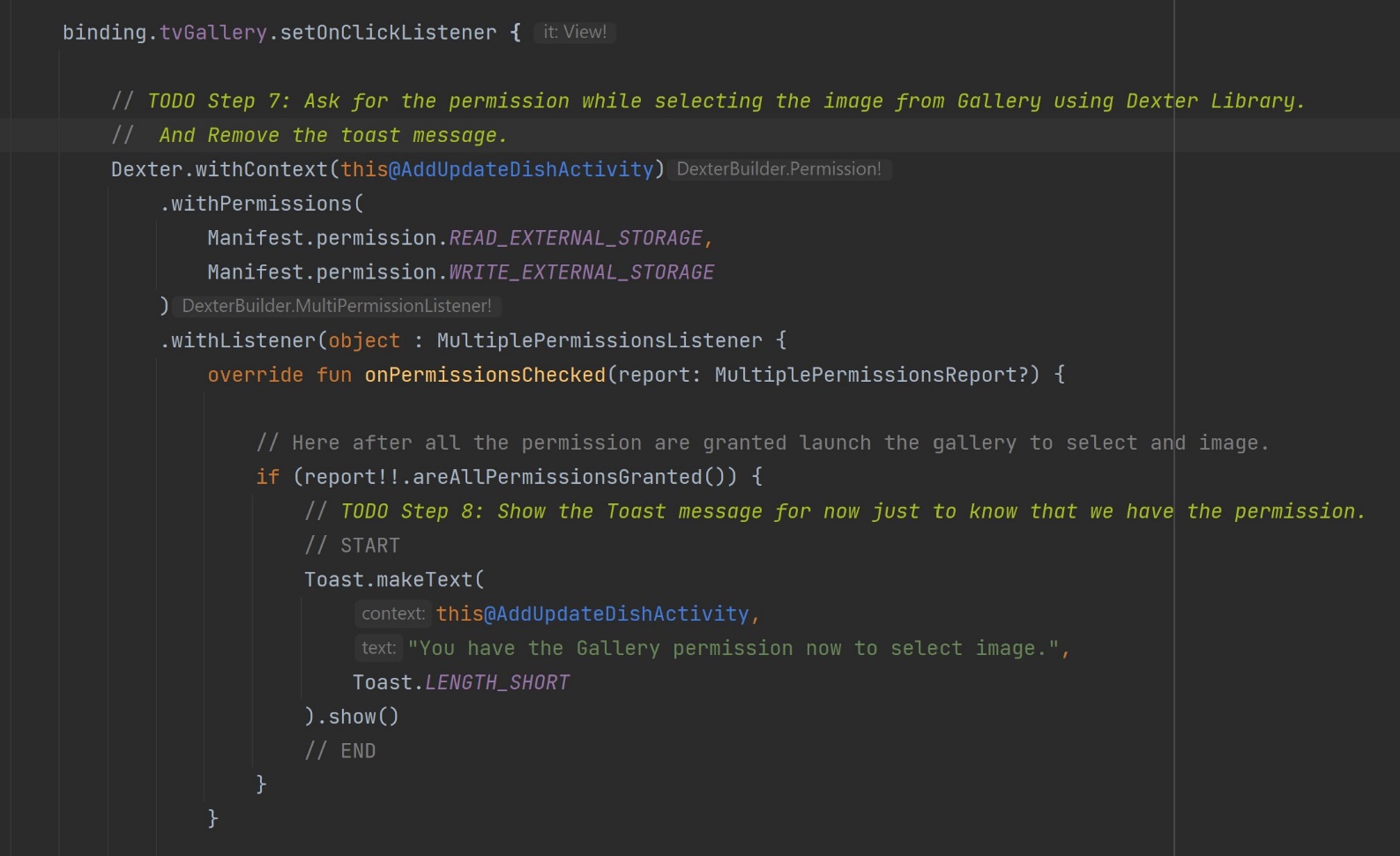
implementation 'com.karumi:dexter:6.2.2'

1. Go to **AddUpdateDishActivity** and ask for permissions while selecting the image from camera using **Dexter** Library. Remove the toast message.
2. Show the Toast message for now just to know that we have the permissions.
3. Create a function to show the alert message that the permission is necessary to proceed further if the user denies it. And ask him to allow it from setting.
4. Show the alert dialog.
5. Ask for the permission while selecting the image from Gallery using Dexter Library. And remove the toast message.
6. Show the Toast message for now just to know that we have the permission.

*See the images in the next pages for steps 3 to 8.*



If you want to receive permission listener callbacks on the same thread that fired the permission request, you just need to call **onSameThread** before checking for permissions. *https://github.com/Karumi/Dexter*



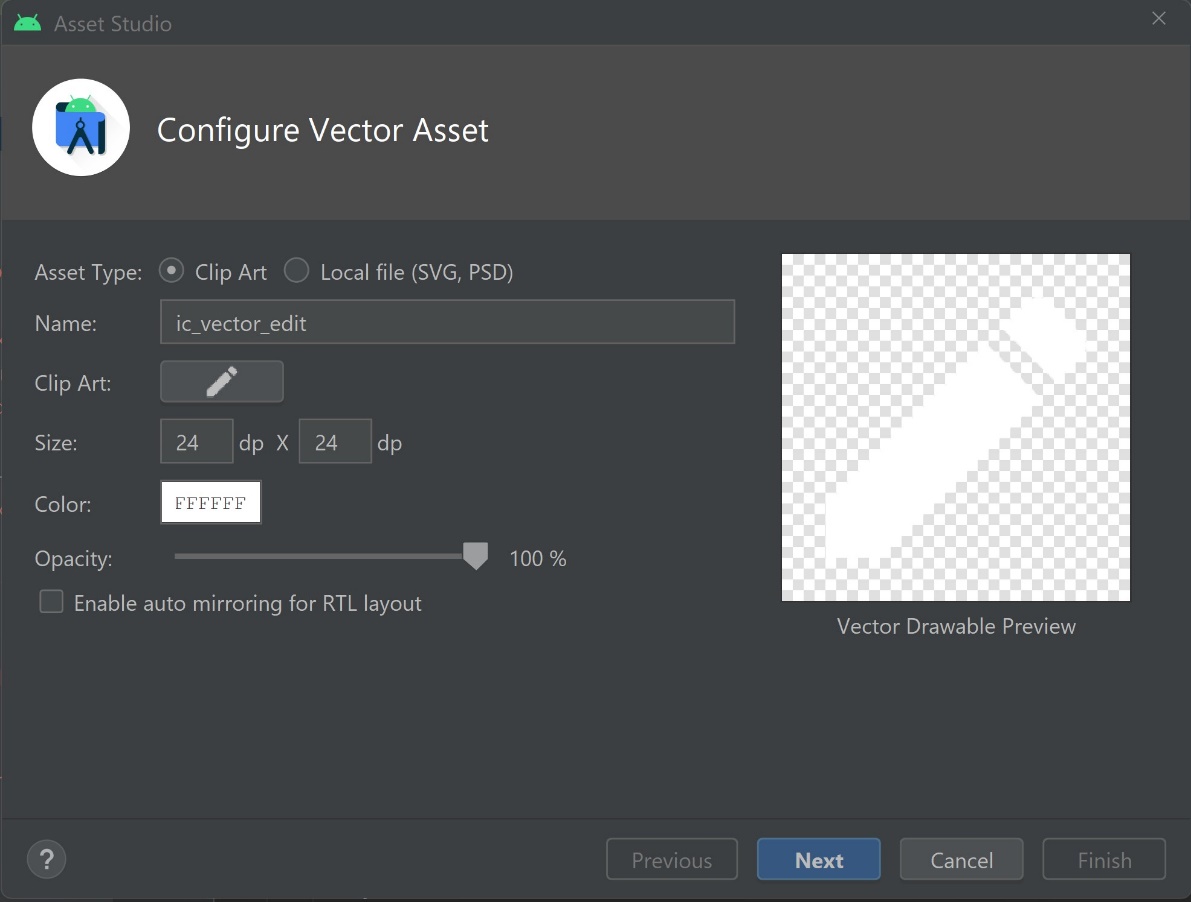
**Note \*\*\*** - Toast messages might not show while using Emulator / virtual device. If you use your actual physical phone, you will see Toast messages.

*Project # 11*  **11\_ElinaResearchProject**

In this project we will implement the functionality of image capturing using a **Camera**.

We will also replace the add image icon (camera alt) with an *edit* icon.

1. Generate and add another vector icon in drawable. Right click on **drawable -> New -> Vector Asset**



1. Find from **Clip Art** “edit” vector
2. Change the name to “**ic\_vector\_gallery**”
3. Change the color into white “#FFFFFF”
4. Click on **Next** and then **Finish**
5. Define the *Companion Object* to define the constants used in the class. We will define the constant for camera.

companion object {  
 private const val CAMERA = 1  
}

1. Start camera using the Image capture action. Get the result in the **onActivityResult** method as we are using **startActivityForResult**.

val intent = Intent(MediaStore.*ACTION\_IMAGE\_CAPTURE*)  
startActivityForResult(intent, CAMERA)

1. Override the onActivityResult method.

public override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {  
 super.onActivityResult(requestCode, resultCode, data)  
 if (resultCode == Activity.*RESULT\_OK*) {  
 if (requestCode == CAMERA) {  
  
 // *Step 9 and 10 will be here*  
  
 }  
 } else if (resultCode == Activity.*RESULT\_CANCELED*) {  
 Log.e("Cancelled", "Cancelled")  
 }  
}

1. Get Image from Camera and set it to the ImageView.

val thumbnail: Bitmap = data!!.*extras*!!.get("data") as Bitmap   
mBinding.ivDishImage.setImageBitmap(thumbnail)

1. Replace the add image icon with edit icon once the image is selected.

mBinding.ivAddDishImage.setImageDrawable(  
 ContextCompat.getDrawable(  
 this@AddUpdateDishActivity,  
 R.drawable.*ic\_vector\_edit*  
)  
)

1. When we use permissions from Dexter, we are accessing external storage. Now it works differently in the latest Api levels. From Api 30 and more it is not required anymore, so the code we have is for older devices or a device that uses an older Android version. So, to make the code work for the new devices we need to comment or delete from binding.tvCamera.setOnClickListener .

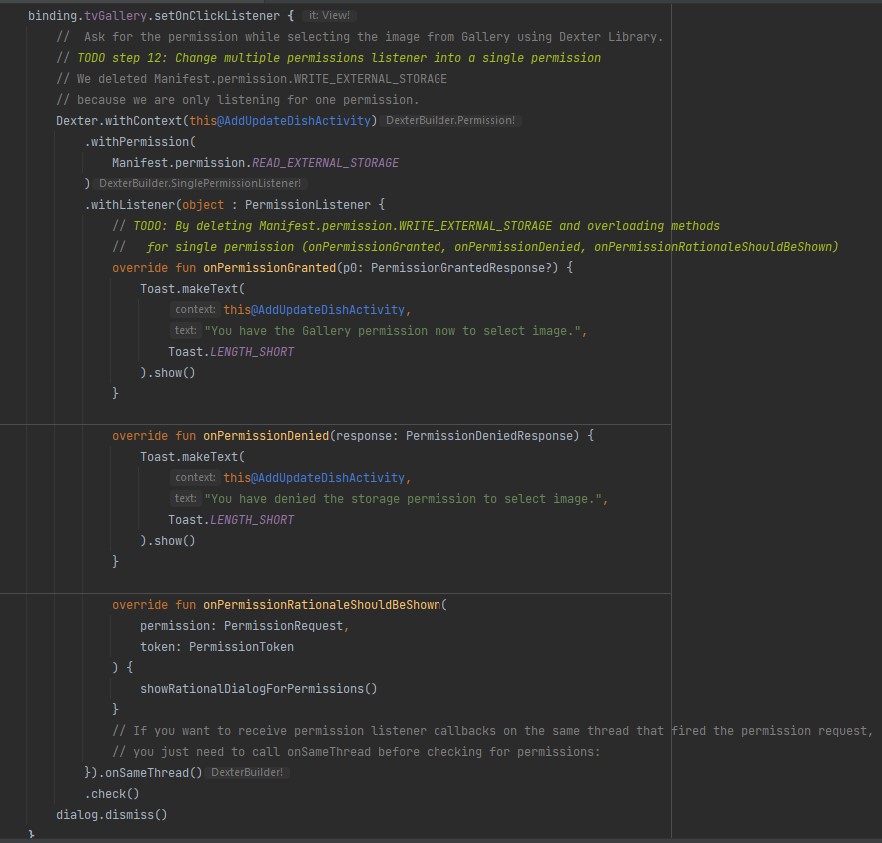
Manifest.permission.*WRITE\_EXTERNAL\_STORAGE*,

1. We do the same in tvGallery, but this time we change multiple permissions into a single permission and accordingly change the overload methods *(onPermissionGranted, onPermissionDenied, onPermissionRationaleShouldBeShown)*  with it.

override fun onPermissionGranted(p0: PermissionGrantedResponse?) {  
 Toast.makeText(  
 this@AddUpdateDishActivity,  
 "You have the Gallery permission now to select image.",  
 Toast.*LENGTH\_SHORT*  
).show()  
}

override fun onPermissionDenied(response: PermissionDeniedResponse) {  
 Toast.makeText(  
 this@AddUpdateDishActivity,  
 "You have denied the storage permission to select image.",  
 Toast.*LENGTH\_SHORT*  
).show()  
}

override fun onPermissionRationaleShouldBeShown(  
 permission: PermissionRequest,  
 token: PermissionToken  
) {  
 showRationalDialogForPermissions()  
}



1. In binding.tvCamera.setOnClickListener we check that report is not null then implement the code. So, instead of *report!!*, we now use:

report?.*let***{... code here}**

*Project # 12*  **12\_ElinaResearchProject**

In this project we will implement the functionality of selecting the image from local storage.

We will get the image URI that we select from the storage.

Also, we will replace the add image icon (camera alt) with an edit icon.

1. Add extra constant for your Gallery in **AddUpdateDishActivity**.

companion object {  
 private const val CAMERA = 1  
  
 // *TODO Step 1: Add the constant for Gallery.*  
 private const val GALLERY = 2  
}

1. Launch the gallery for Image selection using the constant.

override fun onPermissionGranted(response: PermissionGrantedResponse) {  
 // *TODO Step 2: Launch the gallery for Image selection using the constant.*  
 val galleryIntent = Intent(  
 Intent.*ACTION\_PICK*,  
 MediaStore.Images.Media.*EXTERNAL\_CONTENT\_URI*  
)  
  
 startActivityForResult(galleryIntent, GALLERY)  
}

1. Get the selected image from **Gallery**. The selected will be in form of **URI**, so set it to the dish ImageView.

else if (requestCode == GALLERY) {  
  
 data?.*let* **{**  
// Here we will get the select image URI.  
 val selectedPhotoUri = data.*data*  
  
mBinding.ivDishImage.setImageURI(selectedPhotoUri) // Set the selected image from GALLERY to imageView.  
  
 // Replace the add icon with edit icon once the image is selected.  
 mBinding.ivAddDishImage.setImageDrawable(  
 ContextCompat.getDrawable(  
 this@AddUpdateDishActivity,  
 R.drawable.*ic\_vector\_edit*  
)  
 )  
 **}**  
**}**

*Project # 13*  **13\_ElinaResearchProject**

In this project we will implement the third-party library called **Glide** for loading the images.

For more details please visit the link and check out <https://github.com/bumptech/glide>.

After implementing the library we will load the images from **Camera** and **Storage** using **Glide**. It is very useful because it will handle most of the cases that we may face while loading the image with in build methods.

Glide is a fast and efficient image loading library for Android focused on smooth scrolling. Glide offers an easy to use API, a performant and extensible resource decoding pipeline and automatic resource pooling.

1. Add the Glide dependency in **build.gradle**(:app)

implementation 'com.github.bumptech.glide:glide:4.14.2'  
annotationProcessor 'com.github.bumptech.glide:compiler:4.14.2'

**Note\*\*\*** - in the tutorial it is also mentioned to add new repositories in the **build.gradle**(Project) as shown below

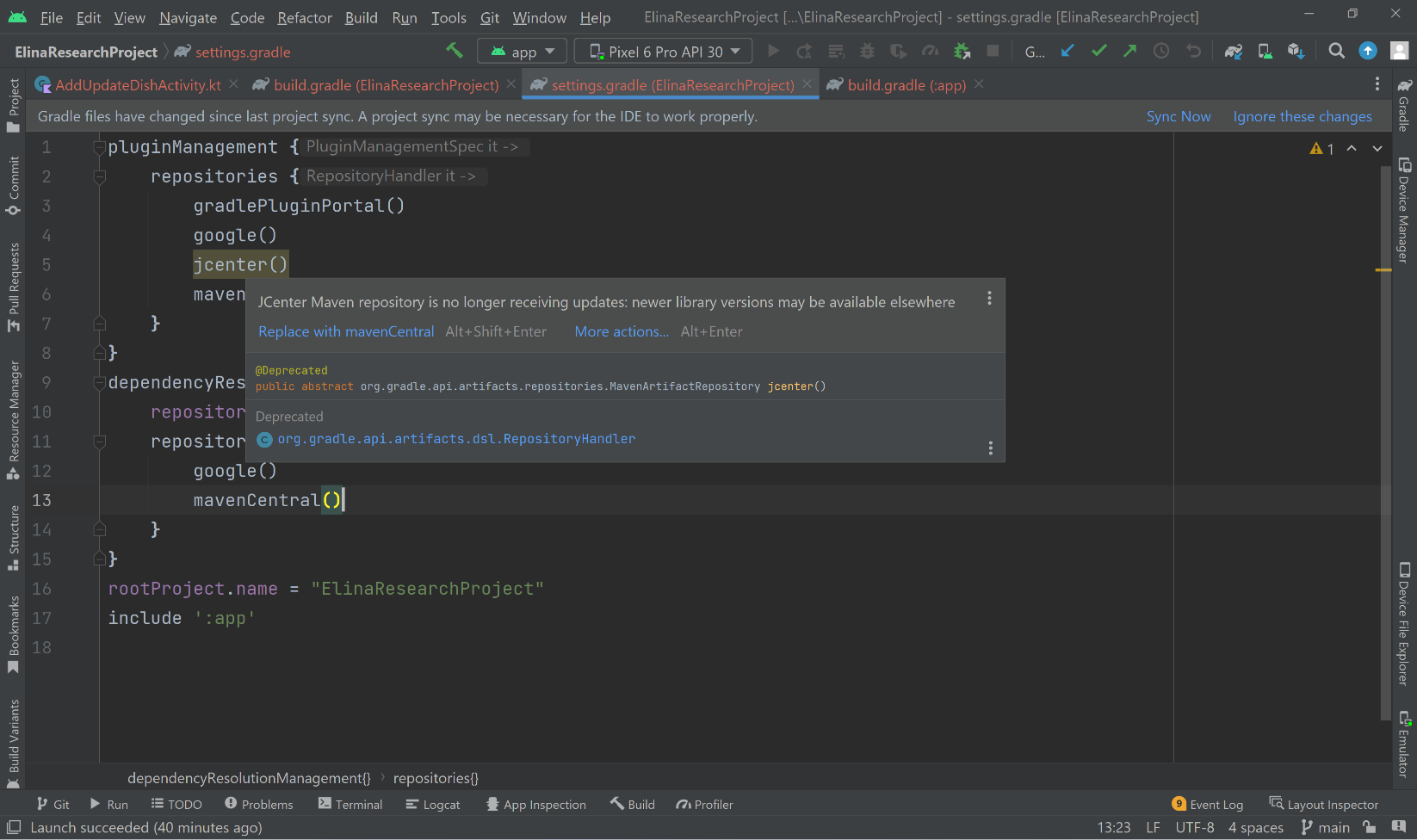
repositories **{**  
google()  
 jcenter()  
}

But starting from *Android Studio Bumblebee* build.gradle structure has been changed.

It is possible to add buildscript above the plugins in **build.gradle**(Project), but when you look in **settings.gradle,** you can find the repositories there.

pluginManagement **{**  
repositories **{**  
gradlePluginPortal()  
 google()  
 mavenCentral()  
 **}**  
**}**

And when you try to add jcenter(), you will see that it is deprecated and mavenCentral() is recommended instead of it, which is already in **settings.gradle** by default, so no need to add them anymore.



There are two **build.gradle** files in android studio i.e. **build.gradle(Project)** & **build.gradle(Module)**. Required blocks like **buildscript**, **dependencies** are available in **build.gradle(Module)**. Repositories have been moved to **settings.gradle** and **module** Gradle has been moved to the **app** package gradle.

1. Replace the **setImageBitmap** using **Glide** as below in **AddUpdateDishActivity**.

Glide.with(this@AddUpdateDishActivity)  
 .load(thumbnail)  
 .centerCrop()  
 .into(mBinding.ivDishImage)

Instead of

mBinding.ivDishImage.setImageBitmap(thumbnail)

1. Replace the setImageURI using Glide as below.

Glide.with(this@AddUpdateDishActivity)  
 .load(selectedPhotoUri)  
 .centerCrop()  
 .into(mBinding.ivDishImage)

Instead of

mBinding.ivDishImage.setImageURI(selectedPhotoUri)

*Project # 14*  **14\_ElinaResearchProject**

In this project we will save the image in the local storage in a separate folder to use it in our app and return the image path. We will print the image path in Log.

1. Declare a constant variable for directory name to store the images.

private const val IMAGE\_DIRECTORY = "FavDishImages"

1. Create a private function to save a copy of an image to internal storage for our App to use. Return a file absolute path. So, the function will take a parameter of type *Bitmap* and return *String*. Name the function **saveImageToInternalStorage**.

- Get the context wrapper instance:

val wrapper = ContextWrapper(*applicationContext*)

*ContextWrapper is going to take the applicationContext,so that it knows to which application this bitmap that we're trying to store is assigned to, because it needs to know that this is an image that was created with our application.*

- Initialize a new file:

var file = wrapper.getDir(  
 IMAGE\_DIRECTORY,  
 Context.*MODE\_PRIVATE*  
*)*

Image is accessible only in our app.

*The Mode Private here is File creation mode: the default mode, where the created file can only be accessed by the calling application (or all applications sharing the*  
*same user ID).*

- Mention a file name to save the image:

file = File(  
 file,  
 "${UUID.randomUUID()}.jpg"  
)

Second parameter is the name that we are going to give to the image – it is a random unique id and the extension .jpg .

- In try catch block get the file output stream, compress the bitmap, flush the stream and finally, close the stream:

try {  
 val stream: OutputStream = FileOutputStream(file)  
 bitmap.compress(Bitmap.CompressFormat.*JPEG*, 100, stream)  
 stream.flush()  
 stream.close()  
} catch (e: IOException) {   
 e.printStackTrace()  
}

- The last, but not least, return the saved image absolute path:

return file.*absolutePath*

1. Create a global variable for stored image path

private var mImagePath: String = ""

1. Save the captured image via Camera to the app directory and get back the image path. So, in the function onActivityResult when request code is CAMEAR:

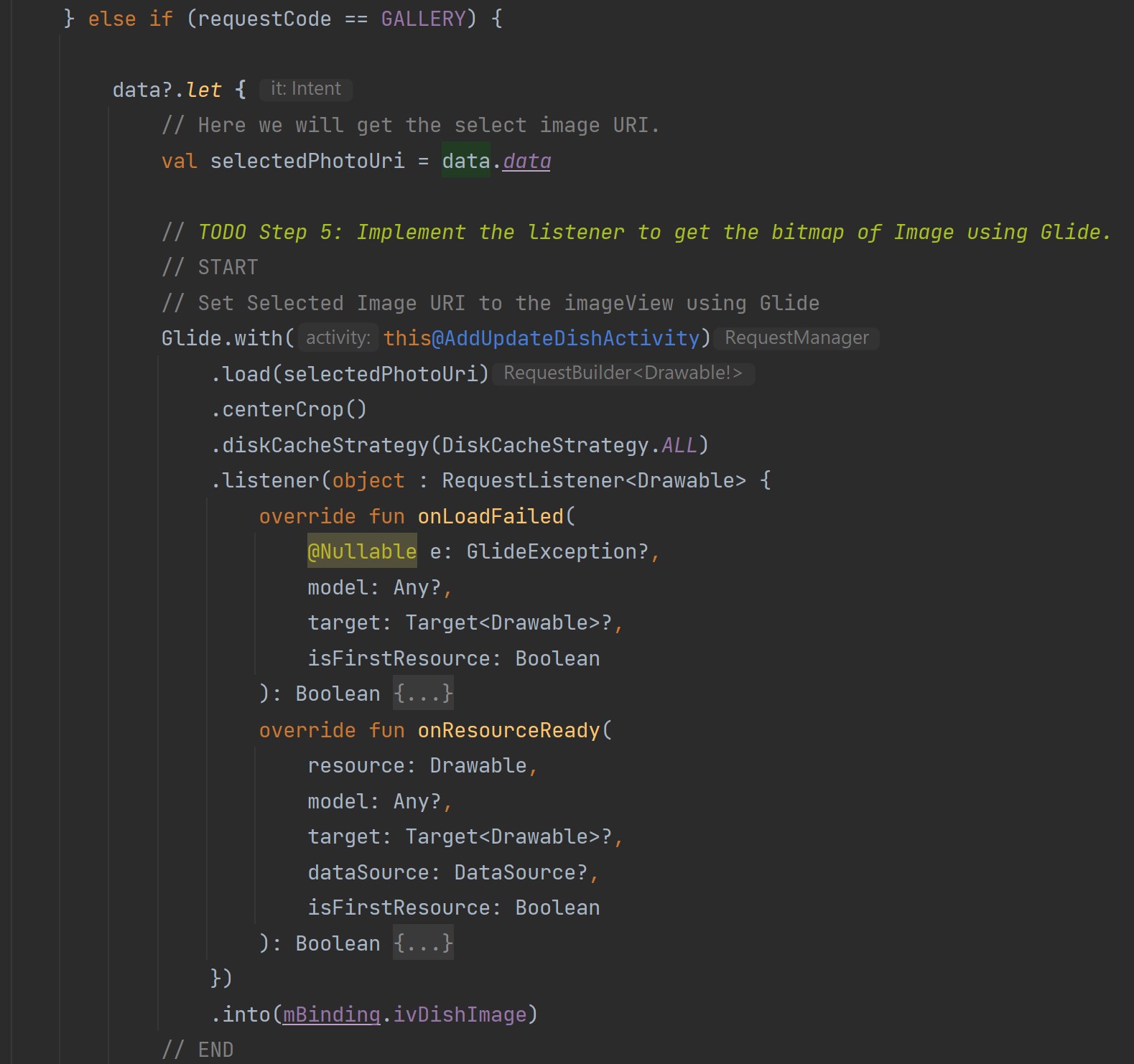
mImagePath = saveImageToInternalStorage(thumbnail)

Once we have taken an image from the camera, that's where we would want to do the setting up of the image.

Log.i("ImagePath", mImagePath)

Log the image path to test on Logcat. So, when you run the app, go to Logcat, choose Info and search for “imagepath”. You will see the image path there.

1. In the function onActivityResult when request code is GALLERY, implement the listener to get the bitmap of Image using Glide. RequestListener will listen for <Drawable>, we need to implement its methods onLoadFailed and onResourceReady.



1. Get the Bitmap and save it to the local storage and get the Image Path

override fun onResourceReady(  
 resource: Drawable,  
 model: Any?,  
 target: Target<Drawable>?,  
 dataSource: DataSource?,  
 isFirstResource: Boolean  
): Boolean {  
 // *Step 6: Get the Bitmap and save it to the local storage and get the Image Path.*  
resource.*let* **{**  
val bitmap: Bitmap = resource.*toBitmap*()  
  
 mImagePath = saveImageToInternalStorage(bitmap)  
 Log.i("ImagePath", mImagePath)  
 **}**  
return false  
}

* If the load failed, so in onLoadFailed method we will just Log:

Log.e("TAG", "Error loading image", e) and return false so that the error placeholder that we use for our application can be placed. Our placeholder will be generated by Glide if the load fails.

*Project # 15*  **15\_ElinaResearchProject**

In this project we will create a custom list dialog to select the Type, Category, and Cooking Time in Minutes.

We will define a list’s values hardcoded. Users have to select them from the list that we have described.

1. Create a new package as “**utils**”. Create a **Constants** object in the “**utils**” package and define the constant values that we can use throughout the application.

const val DISH\_TYPE: String = "DishType"  
const val DISH\_CATEGORY: String = "DishCategory"  
const val DISH\_COOKING\_TIME: String = "DishCookingTime"

1. Define the Dish Types list items in **Constants** object.

fun dishTypes(): ArrayList<String> {  
 val list = ArrayList<String>()  
 list.add("breakfast")  
 list.add("lunch")  
 list.add("snacks")  
 list.add("dinner")  
 list.add("salad")  
 list.add("side dish")  
 list.add("dessert")  
 list.add("other")  
 return list  
}

1. Define the Dish Category list items in **Constants** object.

fun dishCategories(): ArrayList<String> {  
 val list = ArrayList<String>()  
 list.add("Pizza")  
 list.add("BBQ")  
 list.add("Bakery")  
 list.add("Burger")  
 list.add("Cafe")  
 list.add("Chicken")  
 list.add("Dessert")  
 list.add("Drinks")  
 list.add("Hot Dogs")  
 list.add("Juices")  
 list.add("Sandwich")  
 list.add("Tea & Coffee")  
 list.add("Wraps")  
 list.add("Other")  
 return list  
}

1. Define the Dish Cooking Time list items in **Constants** object.

fun dishCookTime(): ArrayList<String> {  
 val list = ArrayList<String>()  
 list.add("10")  
 list.add("15")  
 list.add("20")  
 list.add("30")  
 list.add("45")  
 list.add("50")  
 list.add("60")  
 list.add("90")  
 list.add("120")  
 list.add("150")  
 list.add("180")  
 return list  
}

1. Create and design the custom list dialog layout. It should include one **TextView** for *Dialog Title* and a **RecyclerView** for different scrollable options.
2. Create and design the custom list item layout. It should include one **TextView** for *Item Value* and one divider as **View** the height of which can be 1 sdp.
3. Add color value for divider in **colors.xml** file.

<color name="divider\_line\_color">#e0e0e0</color>

1. Create a custom list adapter to use it while showing the list item in the RecyclerView. In “**view**” package add “**adapters**” package and in there create **CustomListItemAdapter** class, which will take three parameters of type Activity, List<String> and String consecutively.

private val activity: Activity,  
private val listItems: List<String>,  
private val selection: String

The class will extend RecyclerView.Adapter<> and so we need to overload three methods, which are onCreateViewHolder, onBindViewHolder and getItemCount.

Also create an inner class ViewHolder, that will describe an item and metadata about its place within the RecyclerView.

class ViewHolder(view: ItemCustomListLayoutBinding) : RecyclerView.ViewHolder(view.*root*) {  
 // Holds the TextView that will add each item to  
 val tvText = view.tvText  
}

- Inflates the item views which is designed in xml layout file

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {  
 val binding: ItemCustomListLayoutBinding =  
 ItemCustomListLayoutBinding.inflate(LayoutInflater.from(activity), parent, false)  
 return ViewHolder(binding)  
}

- Binds each item in the ArrayList to a view

override fun onBindViewHolder(holder: ViewHolder, position: Int) {  
  
 val item = listItems[position]  
 holder.tvText.*text* = item  
}

- Gets the number of items in the list

override fun getItemCount(): Int {  
 return listItems.size  
}

1. Create a function to launch the Custom List dialog and pass the required parameters in it.

*title - Define the title at runtime according to the list items.*

*itemsList - List of items to be selected.*

*selection - By passing this param you can identify the list item selection.*

private fun customItemsListDialog(title: String, itemsList: List<String>, selection: String) {  
 val customListDialog = Dialog(this)  
  
 val binding: DialogCustomListBinding = DialogCustomListBinding.inflate(*layoutInflater*)  
  
 /\* Set the screen content from a layout resource.  
 The resource will be inflated, adding all top-level views to the screen. \*/  
 customListDialog.setContentView(binding.*root*)  
  
 binding.tvTitle.*text* = title  
  
 // Set the LayoutManager that this RecyclerView will use.  
 binding.rvList.*layoutManager* = LinearLayoutManager(this)

// Adapter class is initialized and list is passed in the param.  
 val adapter = CustomListItemAdapter(this, itemsList, selection)

// adapter instance is set to the recyclerview to inflate the items.  
 binding.rvList.*adapter* = adapter

//Start the dialog and display it on screen.  
 customListDialog.show()  
}

1. Assign the click events to the EditText fields.

mBinding.etType.setOnClickListener(this)  
mBinding.etCategory.setOnClickListener(this)  
mBinding.etCookingTime.setOnClickListener(this)

1. Perform the action of the view and launch the dialog.

* For “**Type”** EditText

R.id.*et\_type* -> {  
 customItemsListDialog(  
 *resources*.getString(R.string.*title\_select\_dish\_type*),  
 Constants.dishTypes(),  
 Constants.DISH\_TYPE  
 )  
 return  
}

* For “**Category”** EditText

R.id.*et\_category* -> {  
 customItemsListDialog(  
 *resources*.getString(R.string.*title\_select\_dish\_category*),  
 Constants.dishCategories(),  
 Constants.DISH\_CATEGORY  
 )  
 return  
}

* For “**Cooking Time in Minutes”** EditText

R.id.*et\_cooking\_time* -> {  
  
 customItemsListDialog(  
 *resources*.getString(R.string.*title\_select\_dish\_cooking\_time*),  
 Constants.dishCookTime(),  
 Constants.DISH\_COOKING\_TIME  
 )  
 return  
}

1. Add new string values in **strings.xml** file.

<string name="title\_select\_dish\_type">SELECT DISH TYPE</string>  
<string name="title\_select\_dish\_category">SELECT DISH CATEGORY</string>  
<string name="title\_select\_dish\_cooking\_time">SELECT COOKING TIME IN MINUTES</string>

*Project # 16*  **16\_ElinaResearchProject**

In this project we will assign the click event to the list item and set the result to the view.

We will validate the entries to store in the local database.

1. Define the custom list dialog as a global variable and initialize it in the function as it was defined previously. We do this in order we could display and dismiss the dialog out of the method’s scope.

private lateinit var mCustomListDialog: Dialog

1. Replace the *dialog* variable with the global variable mCustomListDialog in customItemsListDialog method.
2. Create a function to set the selected item to the view.

fun selectedListItem(item: String, selection: String) {  
 when (selection) {  
 Constants.DISH\_TYPE -> {  
 mCustomListDialog.dismiss()  
 mBinding.etType.setText(item)  
 }  
 Constants.DISH\_CATEGORY -> {  
 mCustomListDialog.dismiss()  
 mBinding.etCategory.setText(item)  
 }  
 else -> {  
 mCustomListDialog.dismiss()  
 mBinding.etCookingTime.setText(item)  
 }  
 }  
}

1. Define the ItemView click event and send the result to the base class.

holder.itemView.setOnClickListener **{**  
  
if (activity is AddUpdateDishActivity) {  
 activity.selectedListItem(item, selection)  
 }  
}

1. Assign the click event to the Add Dish button.

mBinding.btnAddDish.setOnClickListener(this)

1. Perform the action on button click.

Define the local variables and get the EditText values.  
For Dish Image we have the global variable defined already.

val title = mBinding.etTitle.*text*.toString()

.*trim* **{ it** <= ' ' **}**  
**val** type = mBinding.etType.*text*.toString()

.*trim* **{ it** <= ' ' **}**  
**val** category = mBinding.etCategory.*text*.toString()

.*trim* **{ it** <= ' ' **}**  
**val** ingredients = mBinding.etIngredients.*text*.toString()

.*trim* **{ it** <= ' ' **}**  
**val** cookingTimeInMinutes = mBinding.etCookingTime.*text*.toString()

.*trim* **{ it** <= ' ' **}**  
**val** cookingDirection = mBinding.etDirectionToCook.*text*.toString()

.*trim* **{ it** <= ' ' **}**

Validations when the field is empty when {...}:

* Image

TextUtils.isEmpty(mImagePath) -> {  
 Toast.makeText(  
 this@AddUpdateDishActivity,  
 *resources*.getString(R.string.*err\_msg\_select\_dish\_image*),  
 Toast.*LENGTH\_SHORT*  
).show()  
}

* Title

TextUtils.isEmpty(title) -> {  
 Toast.makeText(  
 this@AddUpdateDishActivity,  
 *resources*.getString(R.string.*err\_msg\_enter\_dish\_title*),  
 Toast.*LENGTH\_SHORT*  
).show()  
}

* Type

TextUtils.isEmpty(type) -> {  
 Toast.makeText(  
 this@AddUpdateDishActivity,  
 *resources*.getString(R.string.*err\_msg\_select\_dish\_type*),  
 Toast.*LENGTH\_SHORT*  
).show()  
}

* Category

TextUtils.isEmpty(category) -> {  
 Toast.makeText(  
 this@AddUpdateDishActivity,  
 *resources*.getString(R.string.*err\_msg\_select\_dish\_category*),  
 Toast.*LENGTH\_SHORT*  
).show()  
}

* Ingredients

TextUtils.isEmpty(ingredients) -> {  
 Toast.makeText(  
 this@AddUpdateDishActivity,  
 *resources*.getString(R.string.*err\_msg\_enter\_dish\_ingredients*),  
 Toast.*LENGTH\_SHORT*  
).show()  
}

* Cooking time in minutes

TextUtils.isEmpty(cookingTimeInMinutes) -> {  
 Toast.makeText(  
 this@AddUpdateDishActivity,  
 *resources*.getString(R.string.*err\_msg\_select\_dish\_cooking\_time*),  
 Toast.*LENGTH\_SHORT*  
).show()  
}

* Cooking direction

TextUtils.isEmpty(cookingDirection) -> {  
 Toast.makeText(  
 this@AddUpdateDishActivity,  
 *resources*.getString(R.string.*err\_msg\_enter\_dish\_cooking\_instructions*),  
 Toast.*LENGTH\_SHORT*  
).show()  
}

1. Define all the string values to the string.xml file

<string name="err\_msg\_select\_dish\_image">Select dish image.</string>  
<string name="err\_msg\_enter\_dish\_title">Enter dish title.</string>  
<string name="err\_msg\_select\_dish\_type">Select dish type.</string>  
<string name="err\_msg\_select\_dish\_category">Select dish category.</string>  
<string name="err\_msg\_enter\_dish\_ingredients">Enter dish ingredients. </string>  
<string name="err\_msg\_select\_dish\_cooking\_time">Select dish cooking time. </string>  
<string name="err\_msg\_enter\_dish\_cooking\_instructions">

Enter dish cooking instructions.</string>

1. Show the Toast Message for now that you dish entry is valid.

Toast.makeText(  
 this@AddUpdateDishActivity,  
 "All the entries are valid.",  
 Toast.*LENGTH\_SHORT*  
*).show()*

*Project # 17*  **17\_ElinaResearchProject**

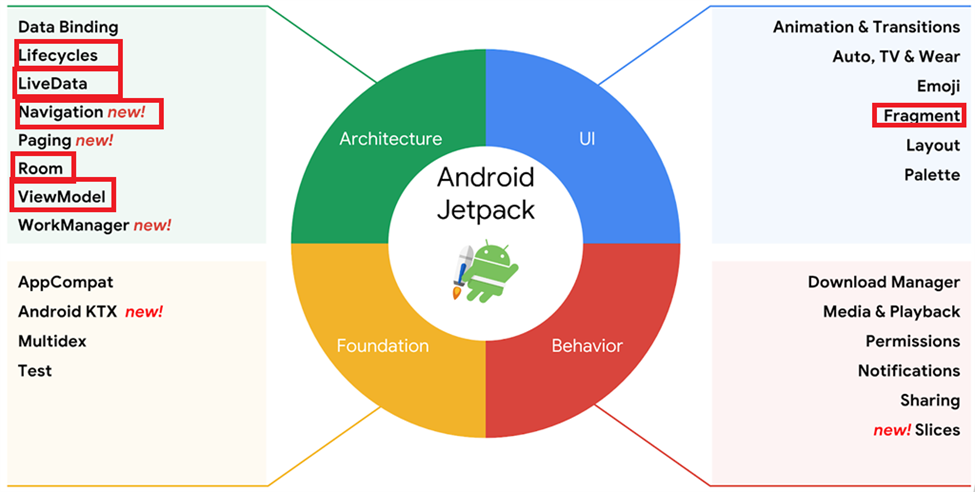
In this project we will start implementing the Android ROOM Database that is also a part of Android Jetpack.

For more information, you can visit:

<https://developer.android.com/codelabs/android-room-with-a-view-kotlin#0>.

We will add all the required dependencies that we are going to use throughout the implementation.

We will also introduce the libraries that are highlighted with a red box and going to add and use them.



So, first of all we need to do the setups in our build.gradle(:app).

1. Apply the "kapt" plugin

id 'kotlin-kapt'

1. Add the "packagingOptions" and also add the "kotlinOptions" if it does not exist already.

"packagingOptions" will exclude the atomic functions module from the package and prevent warnings  
packagingOptions **{**  
exclude 'META-INF/atomicfu.kotlin\_module'  
}

kotlinOptions **{**  
jvmTarget = '1.8'  
}

1. Add all the required dependencies libraries. A few of them are added already by Android Studio while creating the project with **BottomNavigationActivity**.

Room components  
Reference Link: https://developer.android.com/training/data-storage/room  
def room\_version = '2.2.6'  
implementation "androidx.room:room-ktx:$room\_version"  
kapt "androidx.room:room-compiler:$room\_version"

These dependencies are already added if you have created the project with BottomNavigationActivity.  
Lifecycle components  
implementation 'androidx.lifecycle:lifecycle-livedata-ktx:2.5.1'  
implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.5.1'  
  
Navigation  
implementation 'androidx.navigation:navigation-fragment-ktx:2.5.3'  
implementation 'androidx.navigation:navigation-ui-ktx:2.5.3'

1. Create a new package name as “**entities**” in the “**model**” package. After creating the entities package, create data class **FavDish** with the entities (Dish Items) that we want to insert in the database.

Define the Table name  
@Entity(tableName = "fav\_dishes\_table")

data class FavDish(

The name of the column in the table is by default the name of the member variable.  
 @ColumnInfo val image: String,  
 @ColumnInfo val imageSource: String, // Local or Online  
 @ColumnInfo val title: String,  
 @ColumnInfo val type: String,  
 @ColumnInfo val category: String,  
 @ColumnInfo val ingredients: String,

Specifies the name of the column in the table if you want it to be different from the name of the member variable.  
 @ColumnInfo(name = "cookingTime") val cooking\_time: String,  
 @ColumnInfo(name = "instructions") val direction\_to\_cook: String,  
 @ColumnInfo(name = "favoriteDish") var favorite\_dish: Boolean = false,  
 @PrimaryKey(autoGenerate = true) val id: Int = 0  
)

*Project # 18*  **18\_ElinaResearchProject**

In this project we will create a Dao interface and define the insert function with the entity class that we have created in the previous version.

We will also start using the Kotlin Coroutines. We will explain it with a separate demo as Room has Kotlin coroutines support. So we can simply use it.

For more information about the Kotlin Coroutines, you can visit:

<https://developer.android.com/codelabs/kotlin-coroutines#0>

1. Create a new package named as “**database**”in the “**model**” package.
2. Create an interface name as **FavDishDao** that we will use to specify SQL queries and associate them with method calls. DAOs must either be interfaces or abstract classes. With @Dao annotation, we are marking it to be a Dao. And inside of this interface, we can now go ahead and prepare the methods that we are willing to use throughout our application.
3. We will create a function for insert and we will pass a parameter of type FavDish entity class that we have created.

*All queries must be executed on a separate thread. They cannot be executed from* ***Main Thread,*** *or it will cause a crash.*   
***Room*** *has* ***Kotlin Coroutines*** *support.*  
*This allows your queries to be annotated with the* ***suspend*** *modifier and then called from a coroutine or from another suspension function.*

A function to insert favorite dish details to the local database using Room.

@Insert  
suspend fun insertFavDishDetails(favDish: FavDish)

*Project # 19*  **19\_ElinaResearchProject**

In this project we will create a Kotlin class file called **FavDishRoomDatabase**, as well as a Kotlin class file named **FavDishRepository**.

1. Create a Kotlin class file name called **FavDishRoomDatabase** in “**database**” package.

@Database(entities = [FavDish::class], version = 1)  
abstract class FavDishRoomDatabase : RoomDatabase() {  
  
 companion object {  
 // Singleton prevents multiple instances of database opening at the  
 // same time.  
 @Volatile  
 private var INSTANCE: FavDishRoomDatabase? = null  
  
 fun getDatabase(context: Context): FavDishRoomDatabase {  
 // if the INSTANCE is not null, then return it,  
 // if it is, then create the database  
 return INSTANCE ?: *synchronized*(this) **{**  
val instance = Room.databaseBuilder(  
 context.*applicationContext*,  
 FavDishRoomDatabase::class.*java*,  
 "fav\_dish\_database"  
 )  
 .fallbackToDestructiveMigration()  
 .build()  
 INSTANCE = instance  
 // return instance  
 instance  
 **}**  
}  
 }  
}

1. Create a Kotlin class file name called **FavDishRepository** in “**database**” package.

*A Repository manages queries and allows you to use multiple backend. The DAO is passed into the repository constructor. This is because it only needs access to the DAO, since the DAO contains all the read/write methods for the database.*

class FavDishRepository(private val favDishDao: FavDishDao) {...}

1. Create a **suspend** function to insert the data and annotate it with **@WorkerThread**.

@WorkerThread  
suspend fun insertFavDishData(favDish: FavDish) {  
 favDishDao.insertFavDishDetails(favDish)  
}