MAP-06

Complete Android Jetpack Masterclass

Research Project

Sardaryan, Elina

21-01-2023

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 and CRUD Operations
- LiveData, Lifecycles and ViewModels
- Navigation Component, Navigation Graph, Safe Args
- Android Palette
- Retrofit and RxJava
- WorkManager, Sharing and Notifications

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. After the project creation and you can see that 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 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>
```

9. 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>
```

10. 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)

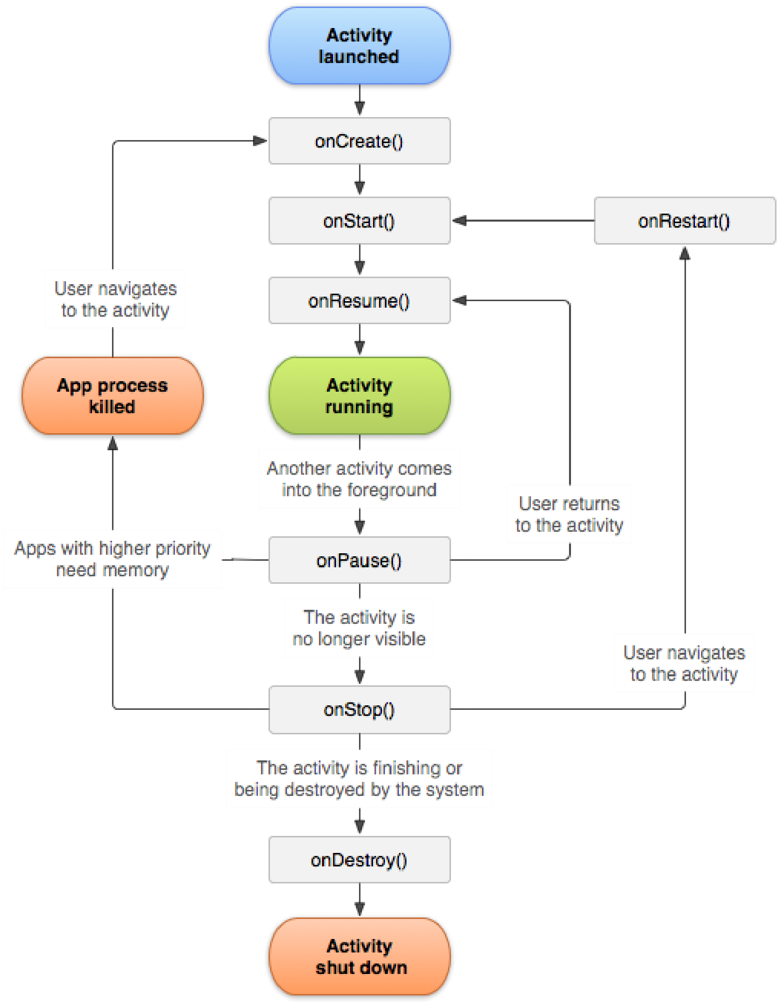
```
<item name="colorPrimary">@color/primary_light_color</item>
 <item name="colorPrimaryVariant">@color/primary_color</item>

<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 the app, the Activity instances in the app transition through different states in the 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.


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

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

Note *** - Configuring the manifest

For the App to be able to use activities, you must declare the activities, and certain of their attributes, in the manifest:

- **Declare activities**
- **Declare intent filters**
- **Declare permissions**

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 {}
```

5. Create an Another Activity to launch it via Intent and to pass the data between two activities.
6. Add the TextView to the Another Activity to just see that it is launched.
7. 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)
```

8. 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>
```

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

```
android:text="@string/splash_screen_title"
```

7. 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.

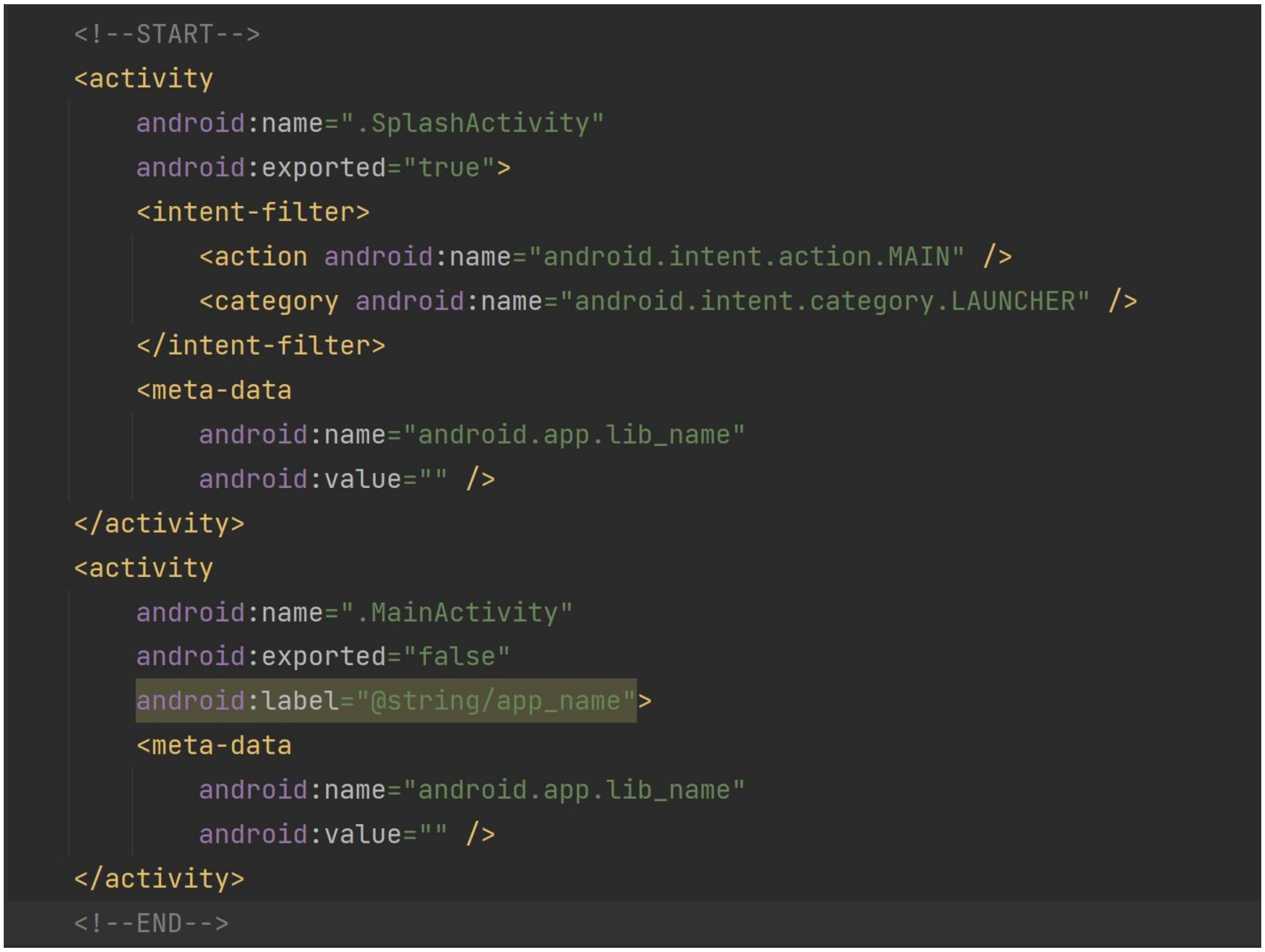
8. Access the XML layout file using the ViewBiding.

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

9. Update the content view using the ViewBinding

```
setContentView(splashBinding.root)
```

10. 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.



```xml
<!--START-->
<activity
    android:name=".SplashActivity"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    <meta-data
        android:name="android.app.lib_name"
        android:value="" />
</activity>
<activity
    android:name=".MainActivity"
    android:exported="false"
    android:label="@string/app_name">
    <meta-data
        android:name="android.app.lib_name"
        android:value="" />
</activity>
<!--END-->
```

11. Run the application and see the changes.
12. 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
     )
 }
```

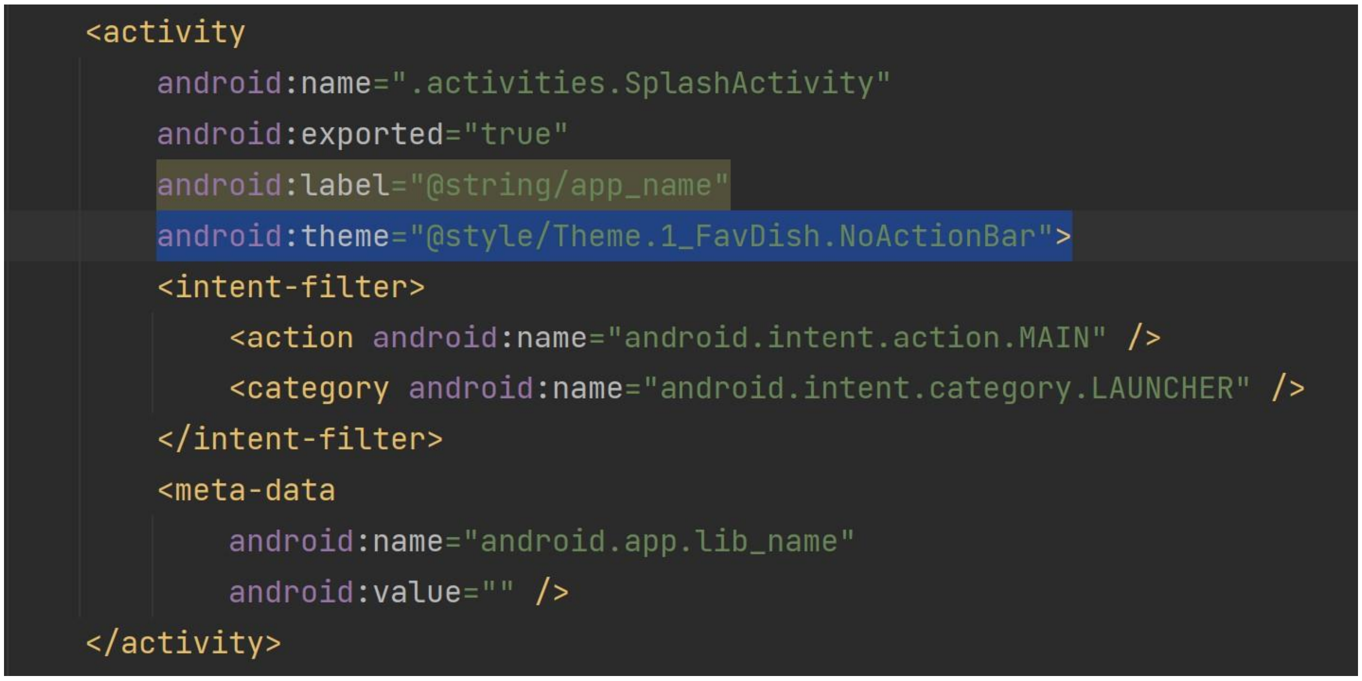
Second, go to themes.xml and type:

```
<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:

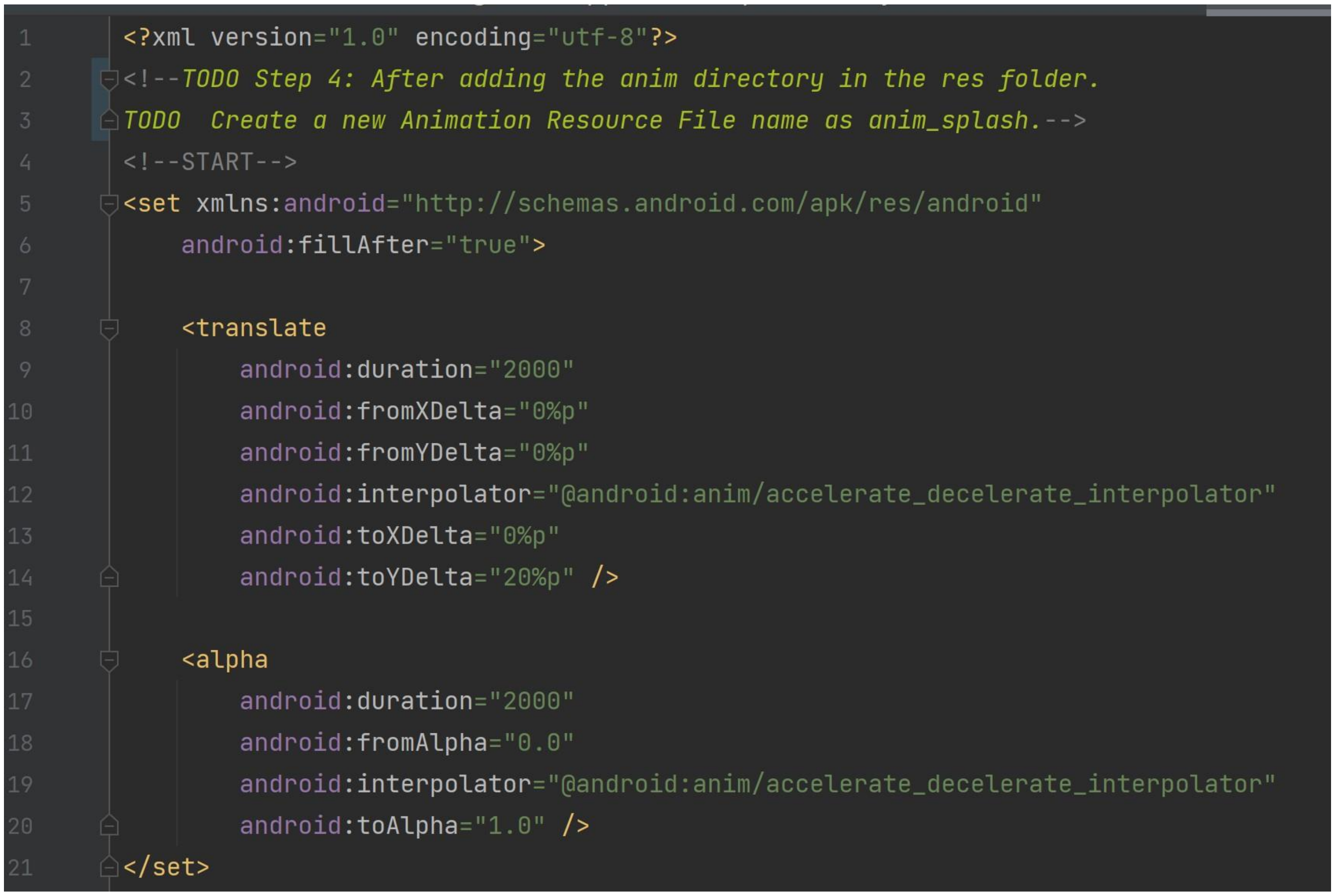
```
android:theme="@style/Theme.FavDish.NoActionBar"
```



```
<activity
    android:name=".activities.SplashActivity"
    android:exported="true"
    android:label="@string/app_name"
    android:theme="@style/Theme.1_FavDish.NoActionBar">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    <meta-data
        android:name="android.app.lib_name"
        android:value="" />
</activity>
```

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

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



```xml
<?xml version="1.0" encoding="utf-8"?>
<!--TODO Step 4: After adding the anim directory in the res folder.
TODO  Create a new Animation Resource File name as anim_splash.-->
<!--START-->
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:fillAfter="true">

    <translate
        android:duration="2000"
        android:fromXDelta="0%p"
        android:fromYDelta="0%p"
        android:interpolator="@android:anim/accelerate_decelerate_interpolator"
        android:toXDelta="0%p"
        android:toYDelta="20%p" />

    <alpha
        android:duration="2000"
        android:fromAlpha="0.0"
        android:interpolator="@android:anim/accelerate_decelerate_interpolator"
        android:toAlpha="1.0" />
</set>
```

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

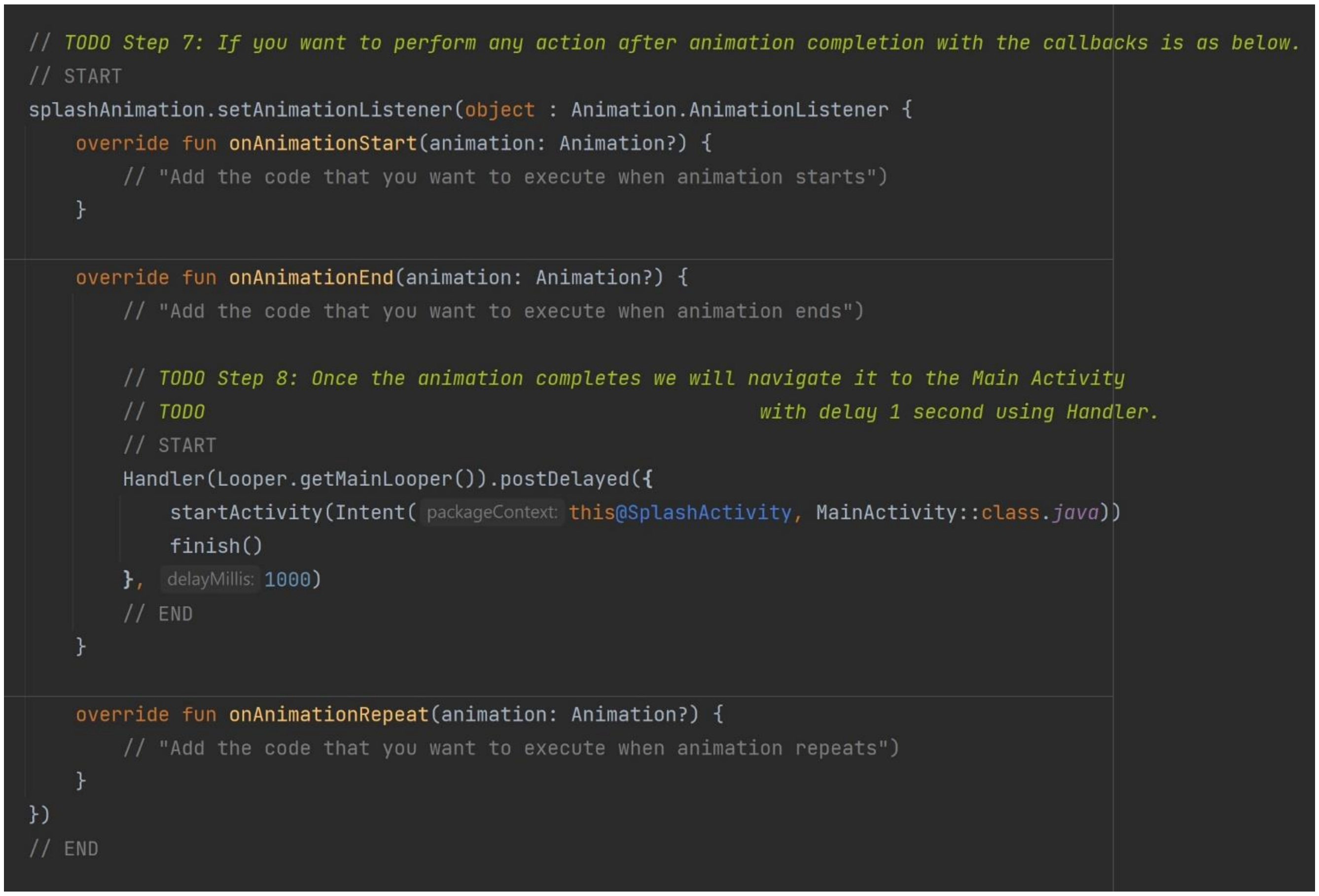
```kotlin
val splashAnimation =
AnimationUtils.loadAnimation(this@SplashActivity,
R.anim.anim_splash)
```

6. Apply the animation to TextView

```kotlin
splashBinding.tvAppName.animation = splashAnimation
```

Note*** - **tvAppName** is id of TextView in **activity_splash.xml** layout.

7. 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:

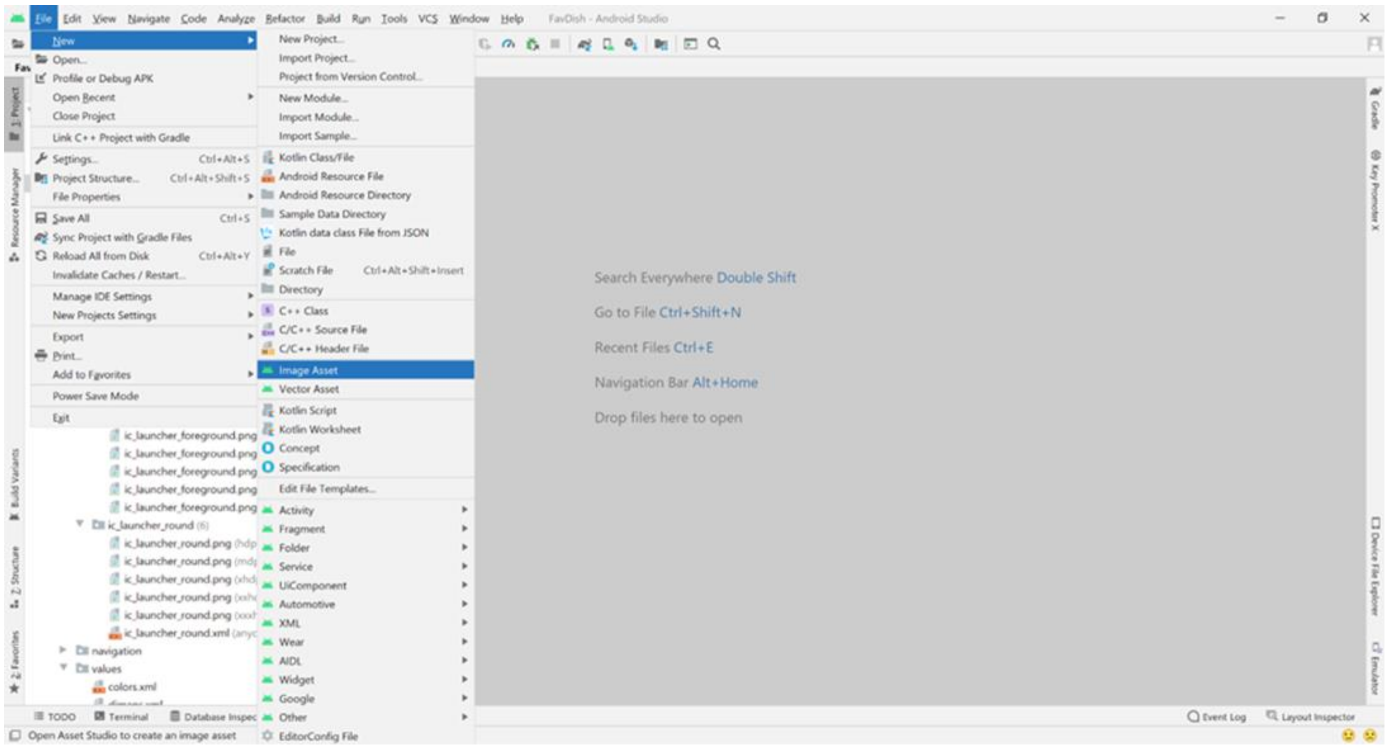


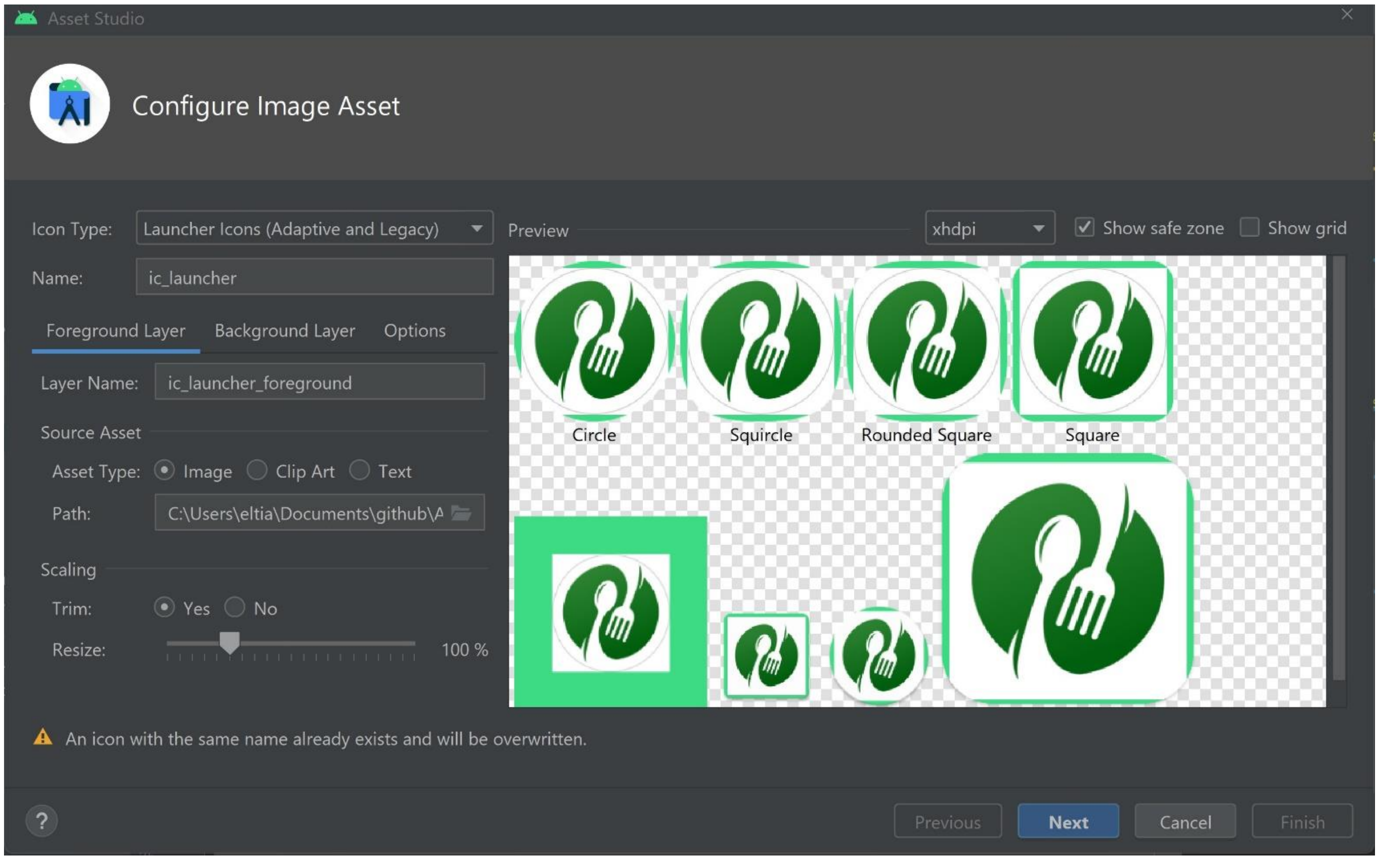
```
// TODO Step 7: If you want to perform any action after animation completion with the callbacks is as below.
// START
splashAnimation.setAnimationListener(object : Animation.AnimationListener {
    override fun onAnimationStart(animation: Animation?) {
        // "Add the code that you want to execute when animation starts")
    }

    override fun onAnimationEnd(animation: Animation?) {
        // "Add the code that you want to execute when animation ends")

        // TODO Step 8: Once the animation completes we will navigate it to the Main Activity
        // TODO                                                  with delay 1 second using Handler.
        // START
        Handler(Looper.getMainLooper()).postDelayed({
            startActivity(Intent( packageContext: this@SplashActivity, MainActivity::class.java))
            finish()
        },  delayMillis: 1000)
        // END
    }

    override fun onAnimationRepeat(animation: Animation?) {
        // "Add the code that you want to execute when animation repeats")
    }
})
// END
```

8. Run the application and see the changes.
9. 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.
10. 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**.





Icon Type: Launcher Icons (Adaptive and Legacy)

Name: ic_launcher

Layer Name: ic_launcher_foreground

Source Asset

Asset Type: [✓ Image] Clip Art Text

Path: C:\Users\eltia\Documents\github\A

Scaling

Trim: [✓ Yes] No

Resize: 100 %

xhdpi

An icon with the same name already exists and will be overwritten.

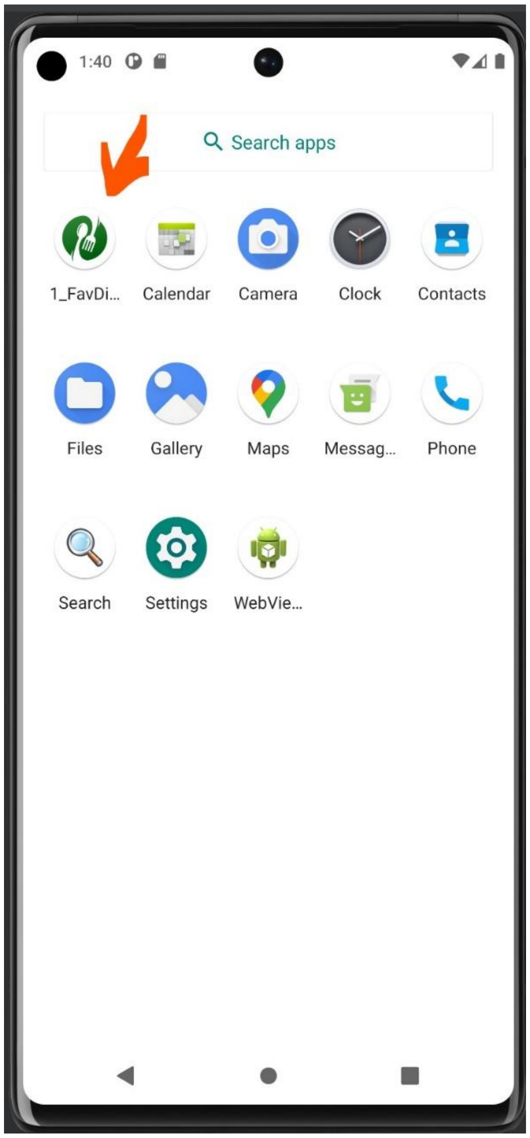
Third, choose the path and select the image you want to use.

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.

11. You will have the below app icon.

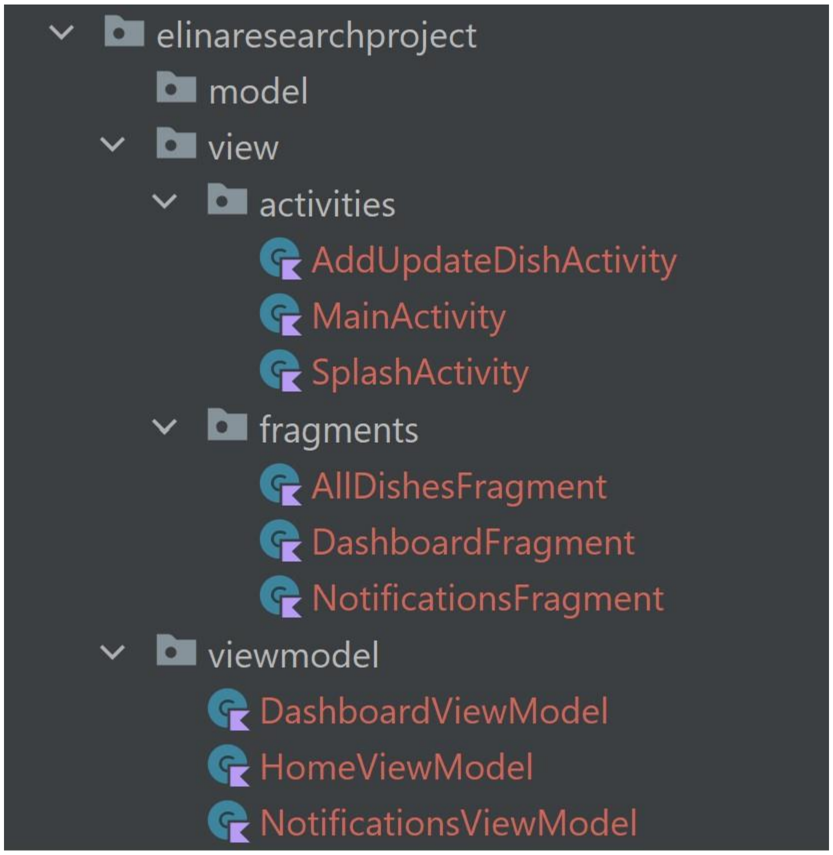


Project # 6 **6_ElinaResearchProject**

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):

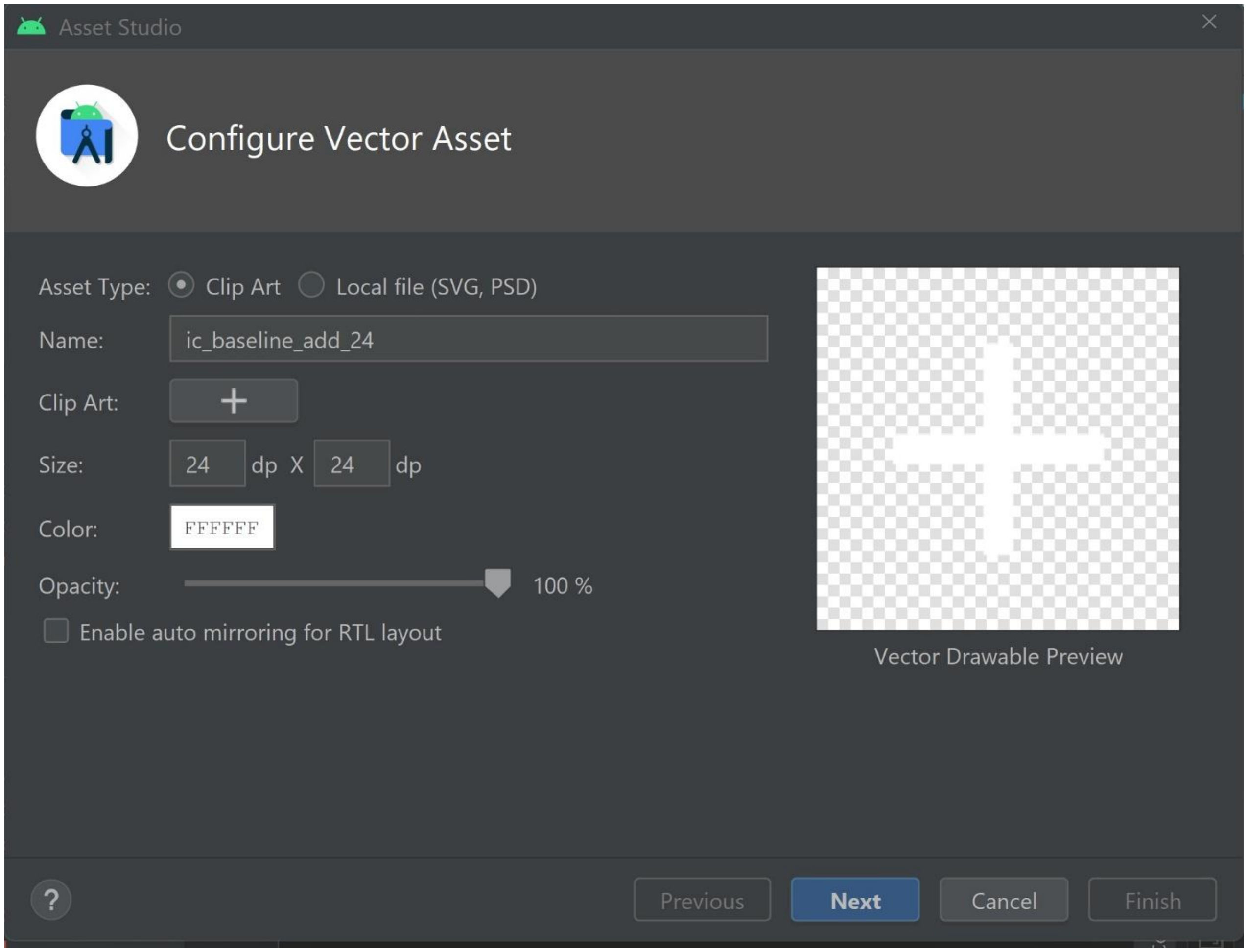


```
elinaresearchproject
    model
    view
        activities
            AddUpdateDishActivity
            MainActivity
            SplashActivity
        fragments
            AllDishesFragment
            DashboardFragment
            NotificationsFragment
    viewmodel
        DashboardViewModel
        HomeViewModel
        NotificationsViewModel
```

9. 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"
```

10. Rename **HomeFragment.kt** as **AllDishesFragment.kt**.
11. Rename the **fragment_home.xml** file as **fragment_all_dishes.xml**.
12. Create a new icon and add it in drawable. **New -> Vector Asset...**



Asset Type: [✓ Clip Art] Local file (SVG, PSD)
Name: ic_baseline_add_24
Clip Art:
Size: 24 dp X 24 dp
Color: FFFFFF
Opacity: 100 %
Enable auto mirroring for RTL layout

13. Name it as “**ic_add**”.
14. Change the color to white “**FFFFFF**”.
15. Click **Next** and then **Finish**.
16. 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.

17. 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"/>
```

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

```
<string name="action_add_dish">Add Dish</string>
```

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

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

20. 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**

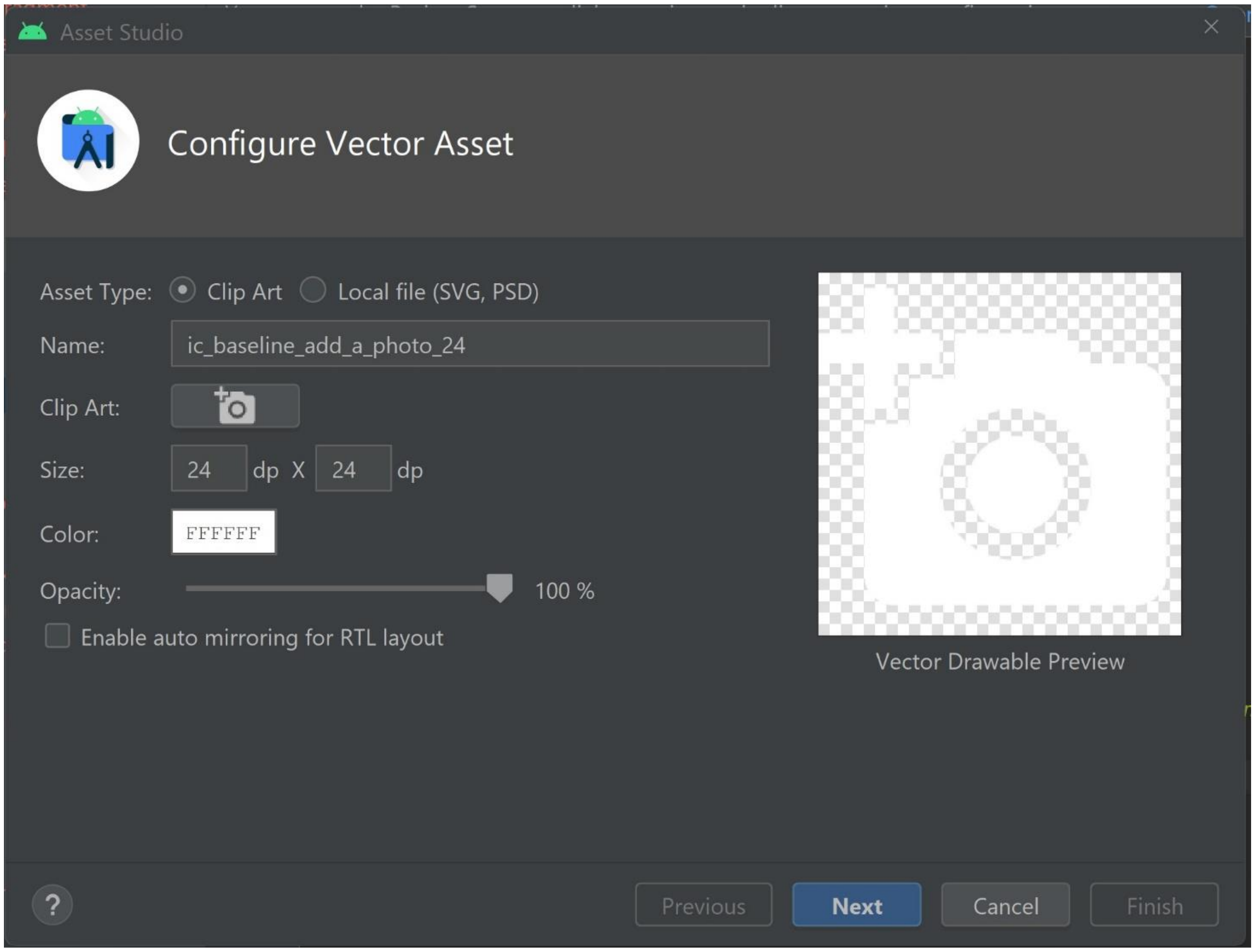
In this project we will design the **activity_add_update_dish.xml** layout file using **ConstraintLayou**t, **ScrollView**, and **FrameLayout**.

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*

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



Asset Type: [✓ Clip Art] Local file (SVG, PSD)
Name: ic_baseline_add_a_photo_24
Clip Art:
Size: 24 dp X 24 dp
Color: FFFFFF
Opacity: 100 %
☐ Enable auto mirroring for RTL layout

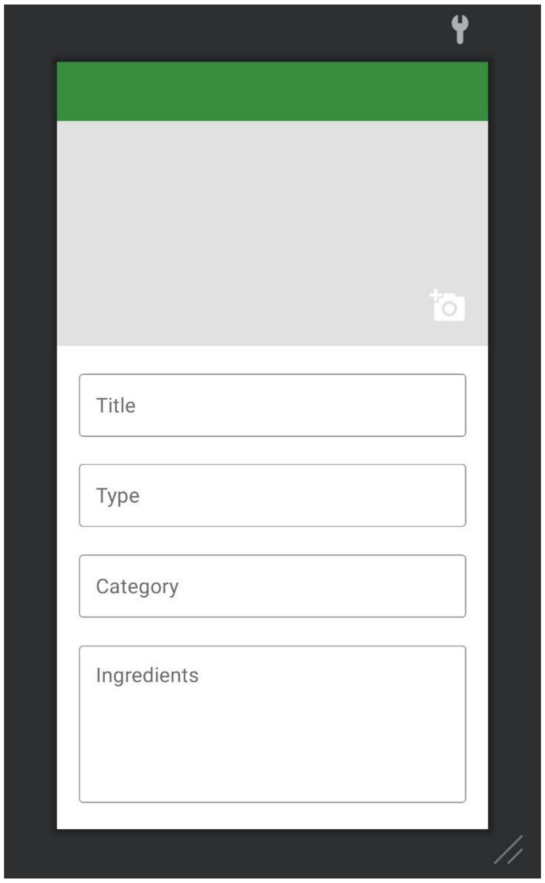
3. Find from **Clip Art** “add_a_photo” vector
4. Change the name to “**ic_add_a_photo**”
5. Change the color to white “**FFFFFF**”
6. Click on **Next** and then **Finish**
7. Add new string values in **strings.xml** as below:

```
<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>
```

8. 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>
```

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



Note *** - use as a guide activity_add_update_dish.xml

10. 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"
```

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

```
private lateinit var mBinding: ActivityAddUpdateDishBinding
```

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

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

13. 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()

  }
}
```

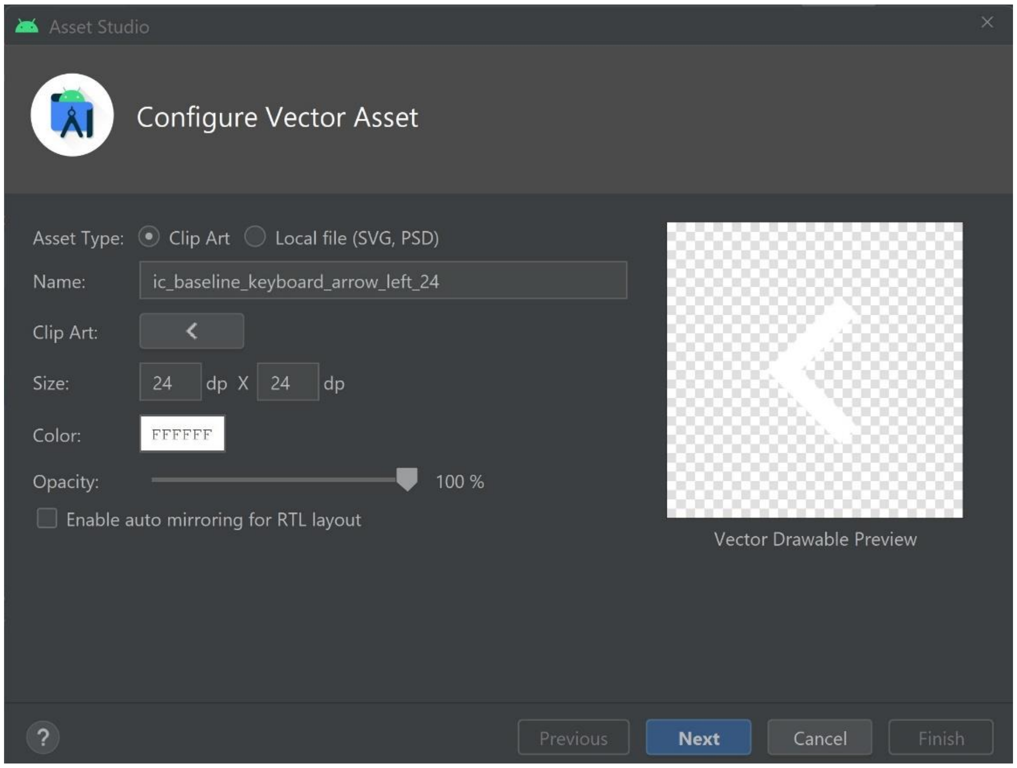
14. Call the method of **setupActionBar** in *onCreate* in **AddUpdateDishActivity.kt**

```
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**



Asset Type: [✓ Clip Art] Local file (SVG, PSD)
Name: ic_baseline_keyboard_arrow_left_24
Clip Art:
Size: 24 dp X 24 dp
Color: FFFFFF
Opacity: 100 %
Enable auto mirroring for RTL layout

2. Find from **Clip Art** "ic_arrow_back" vector
3. Change the name to "**ic_arrow_back**"
4. Change the color to white "**FFFFFF**"
5. Click on **Next** and then **Finish**
6. 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)
```

7. Implement the View.OnClickListener for **AddUpdateDishActivity** .

8. Override the onClick listener method.

```
override fun onClick(v: View) {}
```

9. 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
        }
    }
}
```

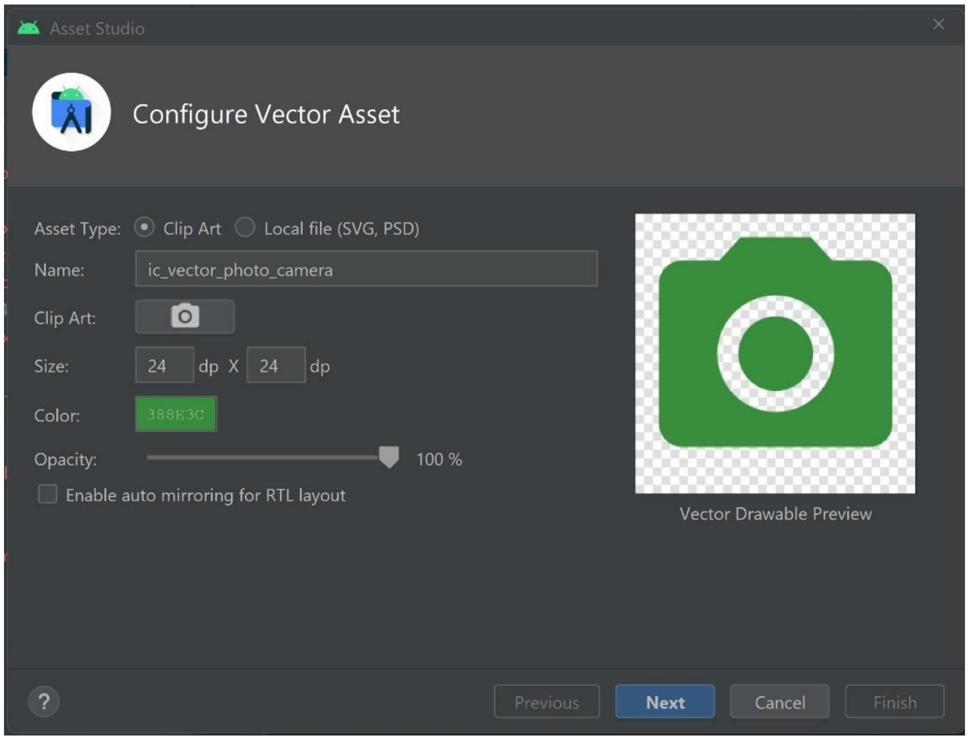
10. Assign the click event to the image button in onCreate method of **AddUpdateDishActivity**.

```
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**.

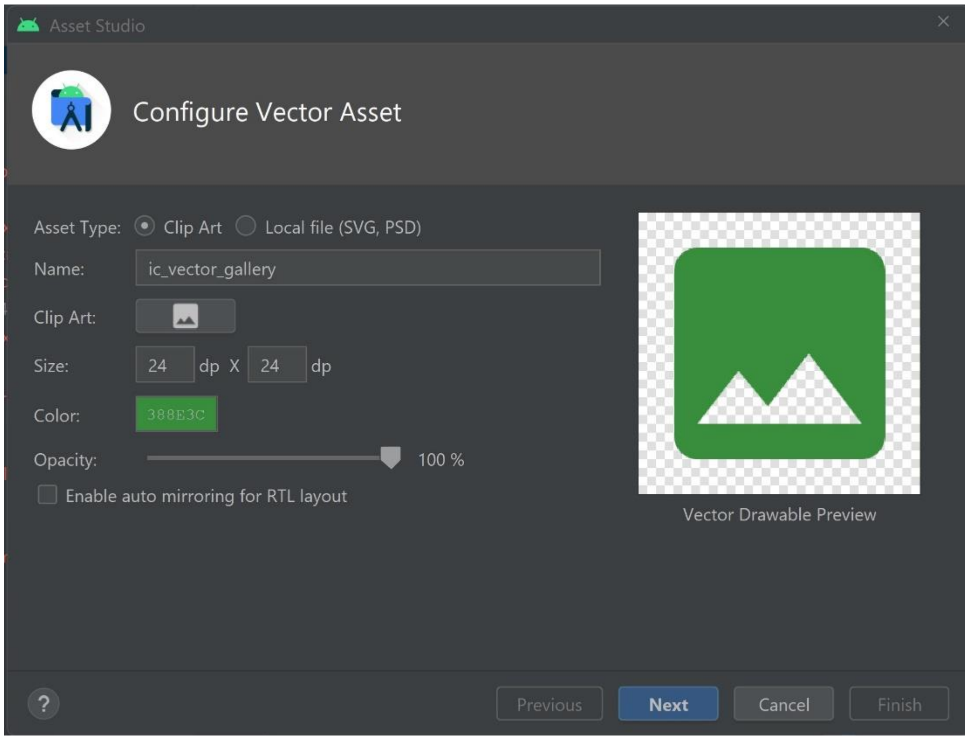
1. Generate and add vector icon in drawable. Right click on **drawable -> New -> Vector Asset**



Asset Type: (●) Clip Art ( ) Local file (SVG, PSD)
Name: ic_vector_photo_camera
Clip Art:
Size: 24 dp X 24 dp
Color: 388E3C
Opacity: 100 %
[ ] Enable auto mirroring for RTL layout

2. Find from **Clip Art** “camera alt” vector
3. Change the name to “**ic_vector_photo_camera**”
4. Change the color into “#388E3C”
5. Click on **Next** and then **Finish**

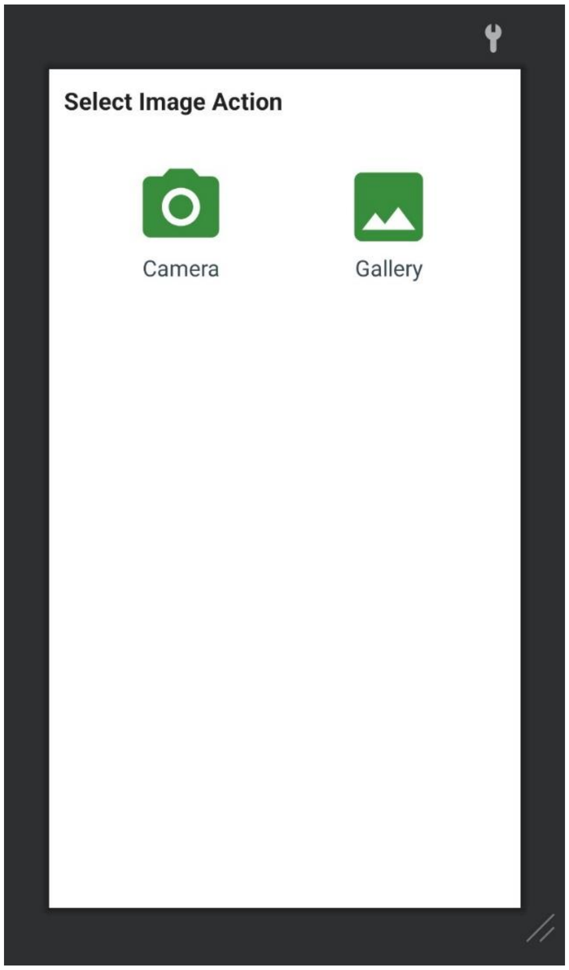
6. Generate and add another vector icon in drawable. Right click on **drawable -> New -> Vector Asset**



Asset Type: [✓ Clip Art] Local file (SVG, PSD)
Name: ic_vector_gallery
Clip Art:
Size: 24 dp X 24 dp
Color: 388E3C
Opacity: 100 %
Enable auto mirroring for RTL layout

7. Find from **Clip Art** “image” vector
8. Change the name to “**ic_vector_gallery**”
9. Change the color into “#388E3C”
10. Click on **Next** and then **Finish**

11. Design a custom dialog using the constraint layout. (See dialog_custom_image_selection.xml)



12. 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>
```

13. 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()
}
```

14. 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" />
```

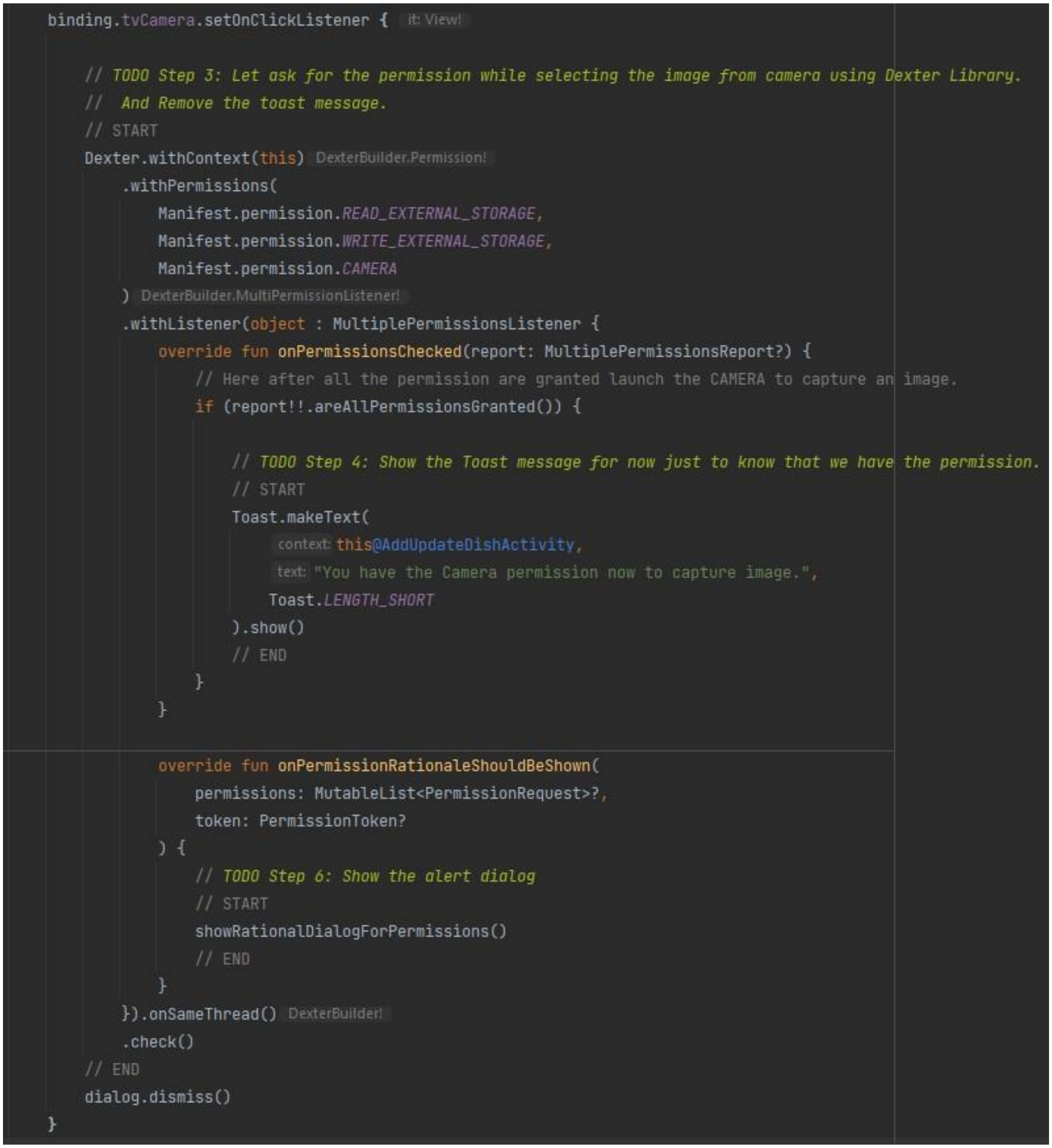
Note **** - this is not the actual permission request that we write in **AndroidManifest.xml**. Here in the Android manifest file, we are saying that we want to request for those permissions at one point in our application and that we will do it later on in our application.

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

```
implementation 'com.karumi:dexter:6.2.2'
```

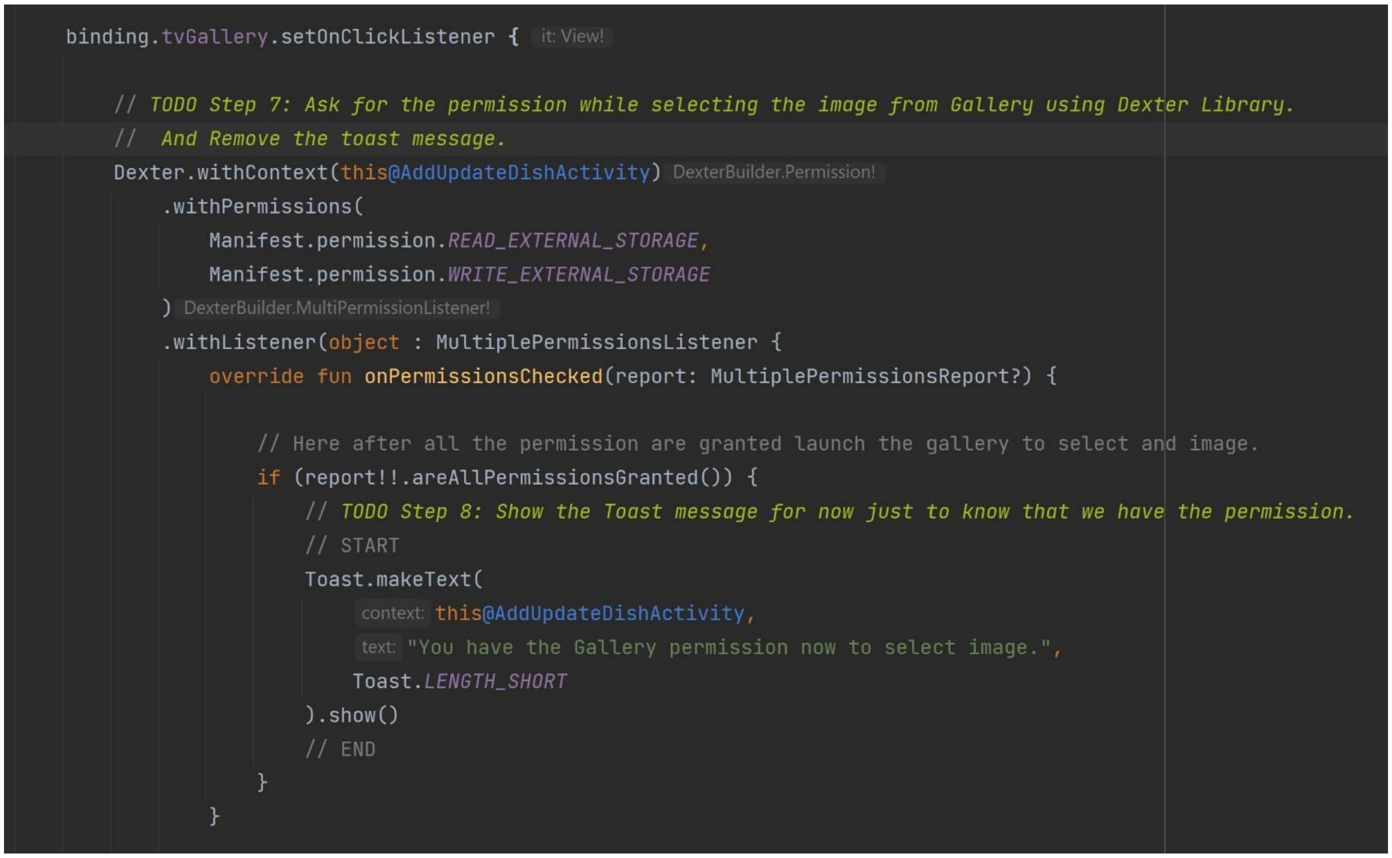
3. Go to **AddUpdateDishActivity** and ask for permissions while selecting the image from camera using **Dexter** Library. Remove the toast message.
4. Show the Toast message for now just to know that we have the permissions.
5. Create a function to show the alert message that the permission is necessary to proceed further if the user denies it. And ask him to allow it from setting.
6. Show the alert dialog.
7. Ask for the permission while selecting the image from Gallery using Dexter Library. And remove the toast message.
8. 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.



```
binding.tvCamera.setOnClickListener { it: View!

    // TODO Step 3: Let ask for the permission while selecting the image from camera using Dexter Library.
    //  And Remove the toast message.
    // START
    Dexter.withContext(this) DexterBuilder.Permission!
        .withPermissions(
            Manifest.permission.READ_EXTERNAL_STORAGE,
            Manifest.permission.WRITE_EXTERNAL_STORAGE,
            Manifest.permission.CAMERA
        ) DexterBuilder.MultiPermissionListener!
        .withListener(object : MultiplePermissionsListener {
            override fun onPermissionsChecked(report: MultiplePermissionsReport?) {
                // Here after all the permission are granted launch the CAMERA to capture an image.
                if (report!!.areAllPermissionsGranted()) {

                    // TODO Step 4: Show the Toast message for now just to know that we have the permission.
                    // START
                    Toast.makeText(
                        context: this@AddUpdateDishActivity,
                        text: "You have the Camera permission now to capture image.",
                        Toast.LENGTH_SHORT
                    ).show()
                    // END
                }
            }

            override fun onPermissionRationaleShouldBeShown(
                permissions: MutableList<PermissionRequest>?,
                token: PermissionToken?
            ) {
                // TODO Step 6: Show the alert dialog
                // START
                showRationalDialogForPermissions()
                // END
            }
        }).onSameThread() DexterBuilder!
        .check()
    // END
    dialog.dismiss()
}
```

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



```
binding.tvGallery.setOnClickListener { it: View!

    // TODO Step 7: Ask for the permission while selecting the image from Gallery using Dexter Library.
    //  And Remove the toast message.
    Dexter.withContext(this@AddUpdateDishActivity) DexterBuilder.Permission!
        .withPermissions(
            Manifest.permission.READ_EXTERNAL_STORAGE,
            Manifest.permission.WRITE_EXTERNAL_STORAGE
        ) DexterBuilder.MultiPermissionListener!
        .withListener(object : MultiplePermissionsListener {
            override fun onPermissionsChecked(report: MultiplePermissionsReport?) {

                // Here after all the permission are granted launch the gallery to select and image.
                if (report!!.areAllPermissionsGranted()) {
                    // TODO Step 8: Show the Toast message for now just to know that we have the permission.
                    // START
                    Toast.makeText(
                        context: this@AddUpdateDishActivity,
                        text: "You have the Gallery permission now to select image.",
                        Toast.LENGTH_SHORT
                    ).show()
                    // END
                }
            }
```

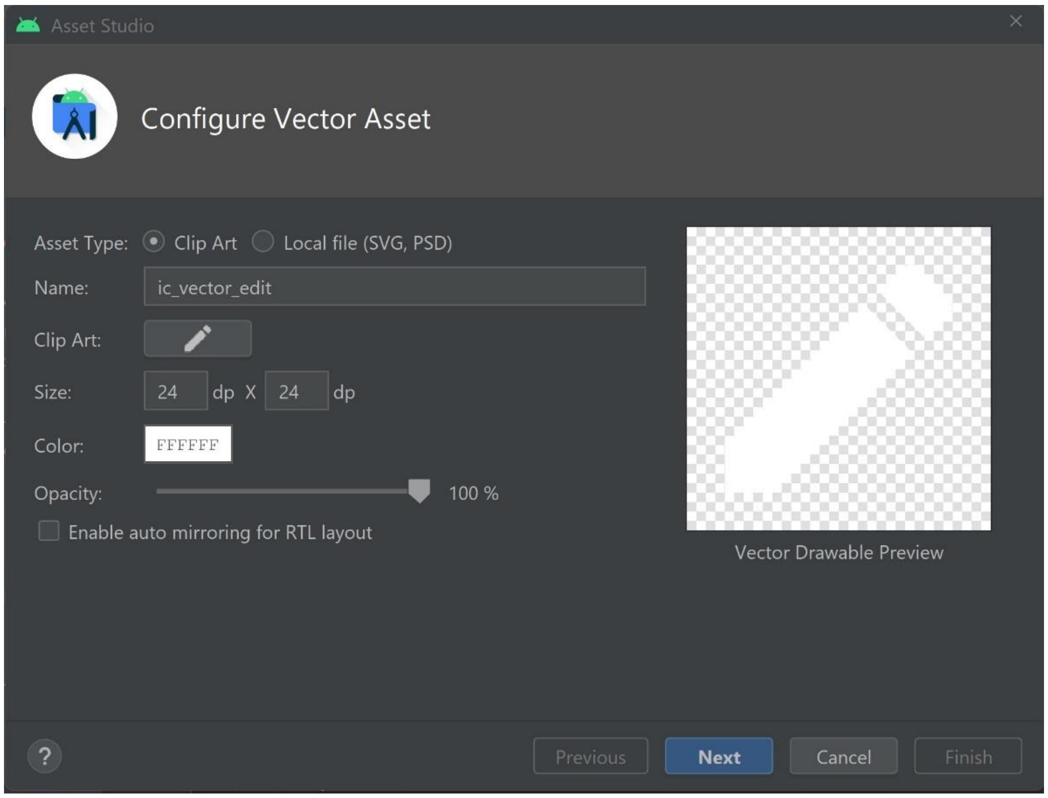
Note *** - Toast messages might not show while using Emulator / virtual device. If you use your actual physical phone, you will see Toast messages.

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**



Asset Type: (● Clip Art) Local file (SVG, PSD)
Name: ic_vector_edit
Clip Art:
Size: 24 dp X 24 dp
Color: FFFFFF
Opacity: 100 %
Enable auto mirroring for RTL layout

2. Find from **Clip Art** "edit" vector
3. Change the name to "**ic_vector_gallery**"
4. Change the color into white "#FFFFFF"
5. Click on **Next** and then **Finish**

6. 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
}
```

7. 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)
```

8. 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")
    }
}
```

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

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

10. 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
    )
)
```

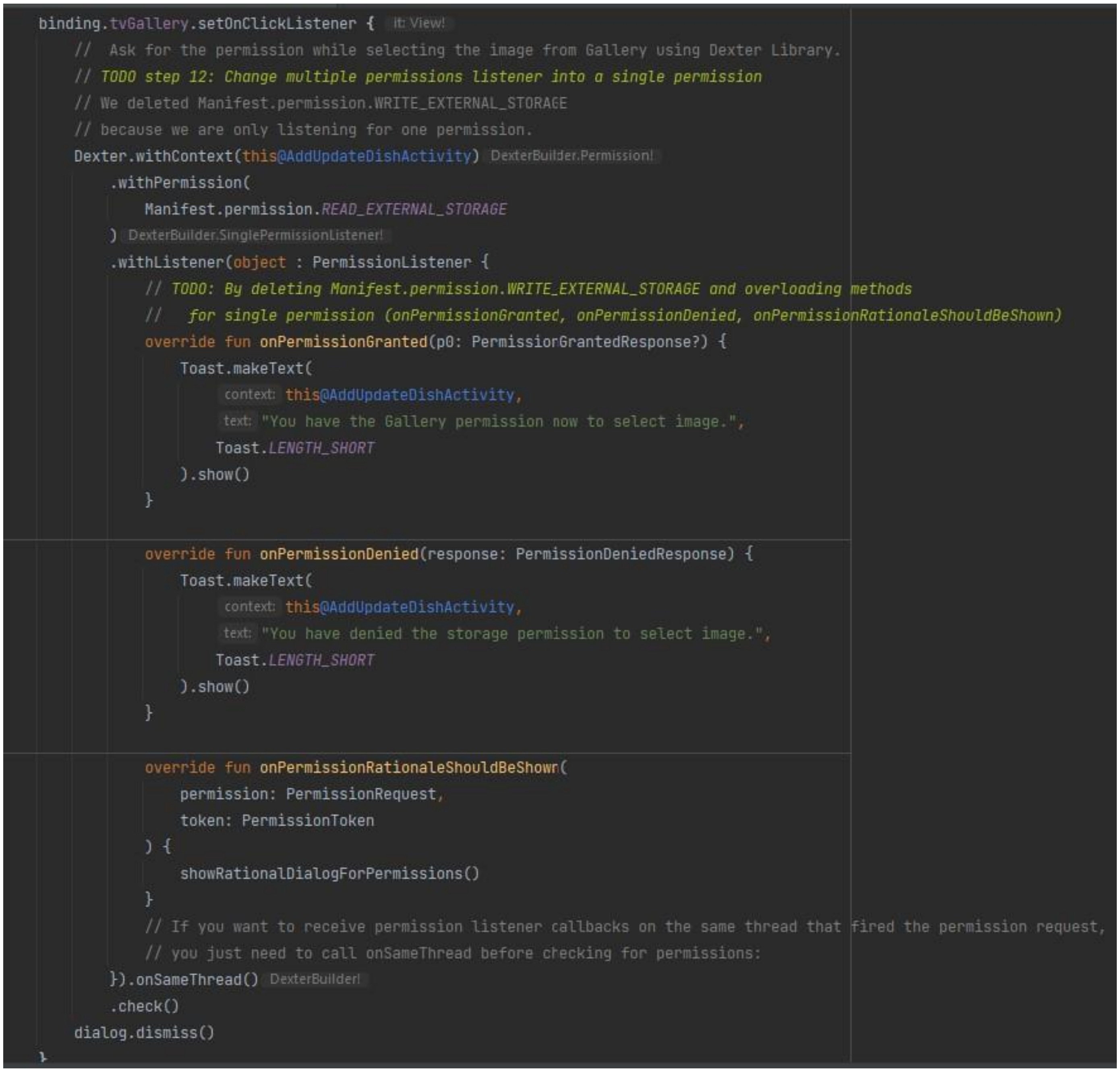
11. When we use permissions from Dexter, we are accessing external storage. Now it works differently in the latest Api levels. From Api 30 and more it is not required anymore, so the code we have is for older devices or a device that uses an older Android version. So, to make the code work for the new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

```
Manifest.permission.WRITE_EXTERNAL_STORAGE,
```

12. 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()
}
```



```
binding.tvGallery.setOnClickListener {  it: View!
    //  Ask for the permission while selecting the image from Gallery using Dexter Library.
    // TODO step 12: Change multiple permissions listener into a single permission
    // We deleted Manifest.permission.WRITE_EXTERNAL_STORAGE
    // because we are only listening for one permission.
    Dexter.withContext(this@AddUpdateDishActivity) DexterBuilder.Permission!
        .withPermission(
            Manifest.permission.READ_EXTERNAL_STORAGE
        ) DexterBuilder.SinglePermissionListener!
        .withListener(object : PermissionListener {
            // TODO: By deleting Manifest.permission.WRITE_EXTERNAL_STORAGE and overloading methods
            //   for single permission (onPermissionGranted, onPermissionDenied, onPermissionRationaleShouldBeShown)
            override fun onPermissionGranted(p0: PermissionGrantedResponse?) {
                Toast.makeText(
                    context: this@AddUpdateDishActivity,
                    text: "You have the Gallery permission now to select image.",
                    Toast.LENGTH_SHORT
                ).show()
            }

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

            override fun onPermissionRationaleShouldBeShown(
                permission: PermissionRequest,
                token: PermissionToken
            ) {
                showRationalDialogForPermissions()
            }
            // 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:
        }).onSameThread() DexterBuilder!
        .check()
    dialog.dismiss()
}
```

13. 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
}
```

2. 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)
}
```

3. 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

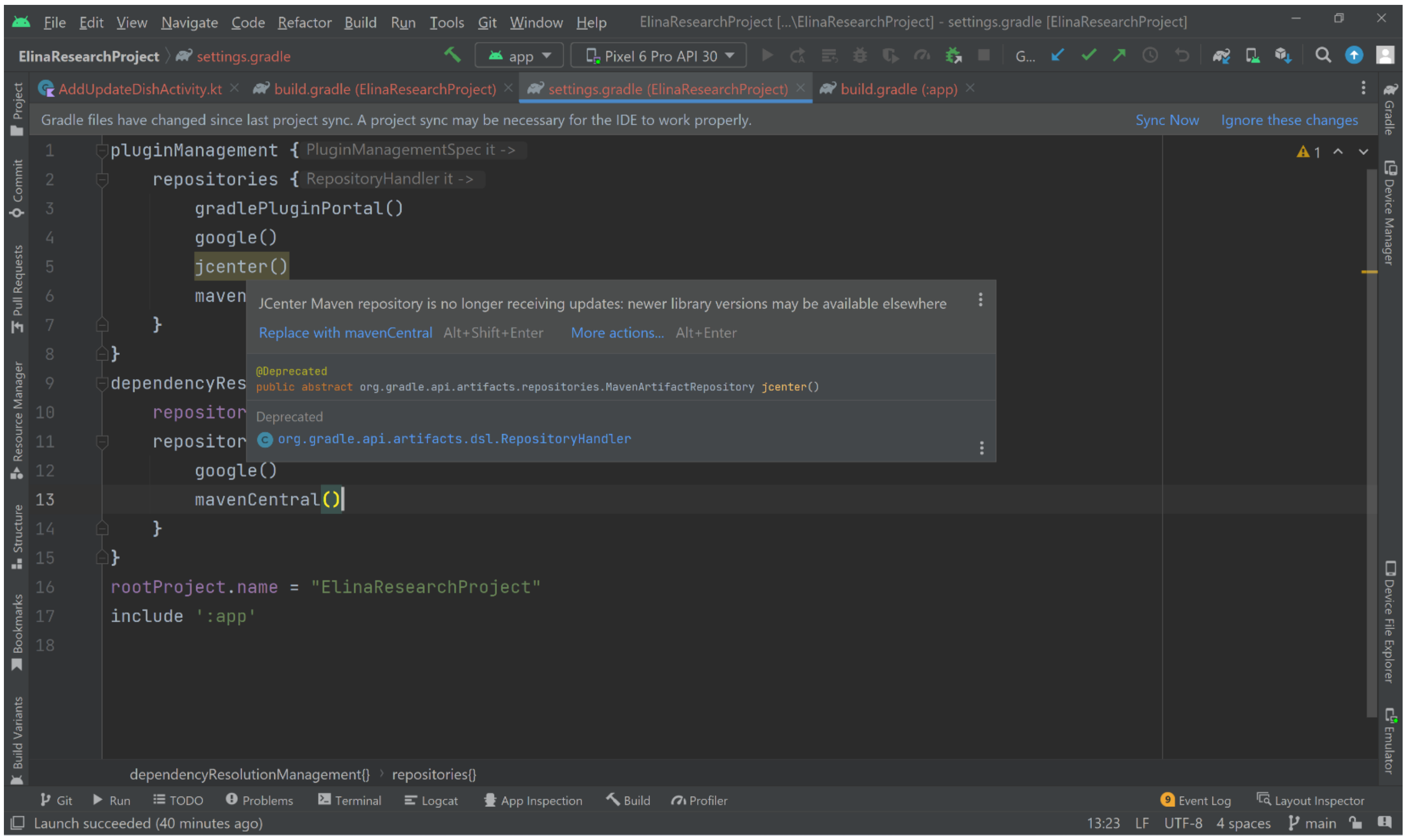
```
repositories {
    google()
    jcenter()
}
```

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.



ElinaResearchProject [...\ElinaResearchProject] - settings.gradle [ElinaResearchProject]

Gradle files have changed since last project sync. A project sync may be necessary for the IDE to work properly.

```
pluginManagement { PluginManagementSpec it ->
    repositories { RepositoryHandler it ->
        gradlePluginPortal()
        google()
        jcenter()
        maven
    }
}
dependencyRes
    repositor
    repositor
        google()
        mavenCentral()
    }
}
rootProject.name = "ElinaResearchProject"
include ':app'
```

JCenter Maven repository is no longer receiving updates: newer library versions may be available elsewhere

Replace with mavenCentral Alt+Shift+Enter More actions... Alt+Enter

```
@Deprecated
public abstract org.gradle.api.artifacts.repositories.MavenArtifactRepository jcenter()
```

Deprecated

org.gradle.api.artifacts.dsl.RepositoryHandler

Launch succeeded (40 minutes ago)

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.

2. 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)
```

3. 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"
```

2. 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
```

3. Create a global variable for stored image path

```
private var mImagePath: String = ""
```

4. 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:

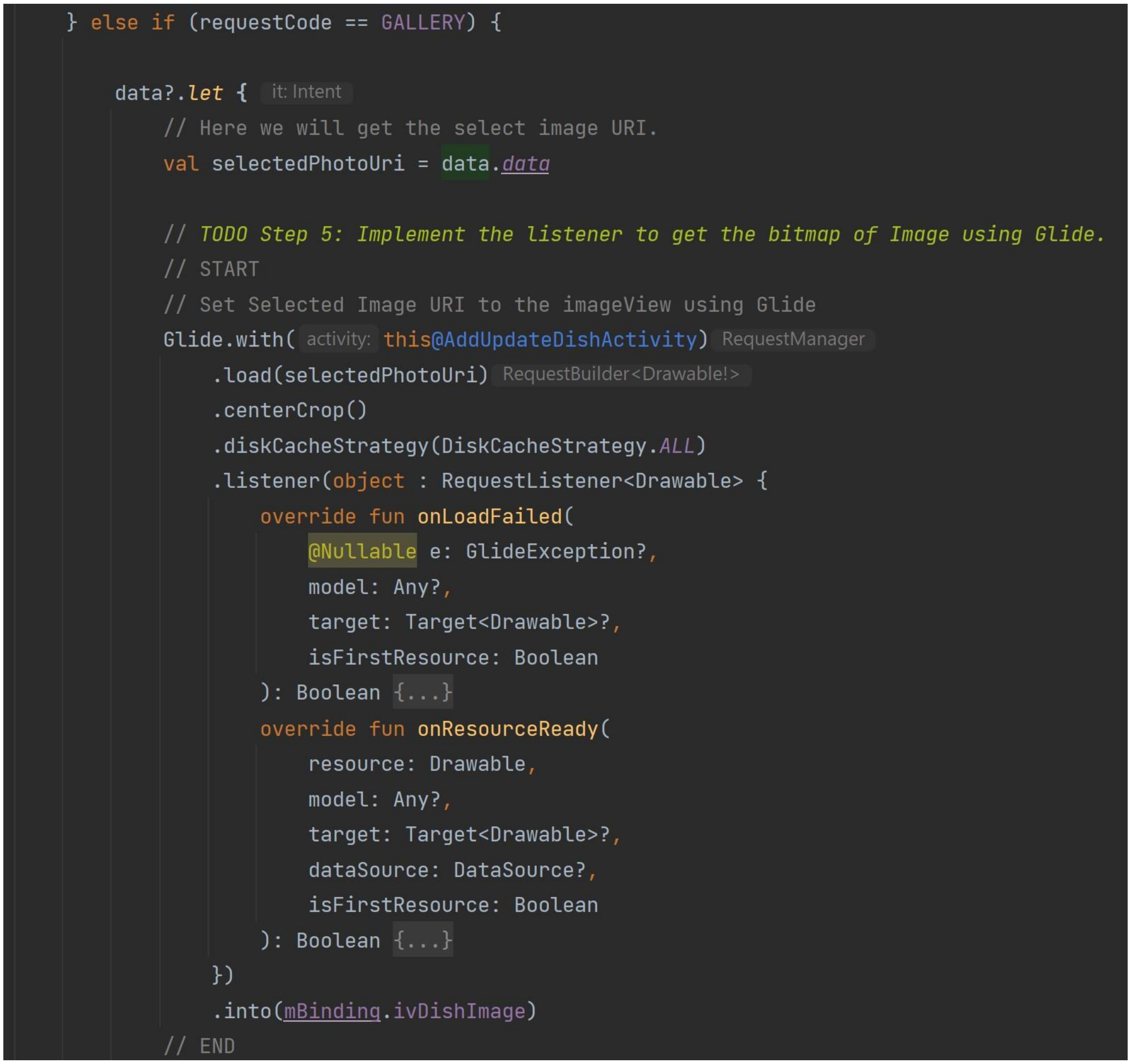
```
mImagePath = saveImageToInternalStorage(thumbnail)
```

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.

5. 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`.



```
} else if (requestCode == GALLERY) {

    data?.let { it: Intent
        // Here we will get the select image URI.
        val selectedPhotoUri = data.data

        // TODO Step 5: Implement the listener to get the bitmap of Image using Glide.
        // START
        // Set Selected Image URI to the imageView using Glide
        Glide.with( activity: this@AddUpdateDishActivity) RequestManager
            .load(selectedPhotoUri) RequestBuilder<Drawable!>
            .centerCrop()
            .diskCacheStrategy(DiskCacheStrategy.ALL)
            .listener(object : RequestListener<Drawable> {
                override fun onLoadFailed(
                    @Nullable e: GlideException?,
                    model: Any?,
                    target: Target<Drawable>?,
                    isFirstResource: Boolean
                ): Boolean {...}
                override fun onResourceReady(
                    resource: Drawable,
                    model: Any?,
                    target: Target<Drawable>?,
                    dataSource: DataSource?,
                    isFirstResource: Boolean
                ): Boolean {...}
            })
            .into(mBinding.ivDishImage)
        // END
```

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

```kotlin
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"
```

2. 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
}
```

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

```
fun dishCategories(): ArrayList<String> {
    val list = ArrayList<String>()
    list.add("Pizza")
    list.add("BBQ")
```

```kotlin
    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
}
```

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

```kotlin
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
}
```

5. Create and design the custom list dialog layout. It should include one **TextView** for *Dialog Title* and a **RecyclerView** for different scrollable options.

6. 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.

7. Add color value for divider in **colors.xml** file.

```
<color name="divider_line_color">#e0e0e0</color>
```

8. 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
```

```kotlin
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
```

```kotlin
override fun onBindViewHolder(holder: ViewHolder, position: Int)
{

    val item = listItems[position]
    holder.tvText.text = item
}
```

```
- Gets the number of items in the list
```

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

9. 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.
```

```kotlin
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()
}
```

10. Assign the click events to the EditText fields.

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

11. 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
}
```

12. 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
```

2. Replace the *dialog* variable with the global variable `mCustomListDialog` in `customItemsListDialog` method.
3. 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)
        }
    }
}
```

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

```
holder.itemView.setOnClickListener {

    if (activity is AddUpdateDishActivity) {
        activity.selectedListItem(item, selection)
    }
}
```

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

```
mBinding.btnAddDish.setOnClickListener(this)
```

6. 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_instructi
ons),
        Toast.LENGTH_SHORT
    ).show()
}
```

7. 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>
```

8. 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**

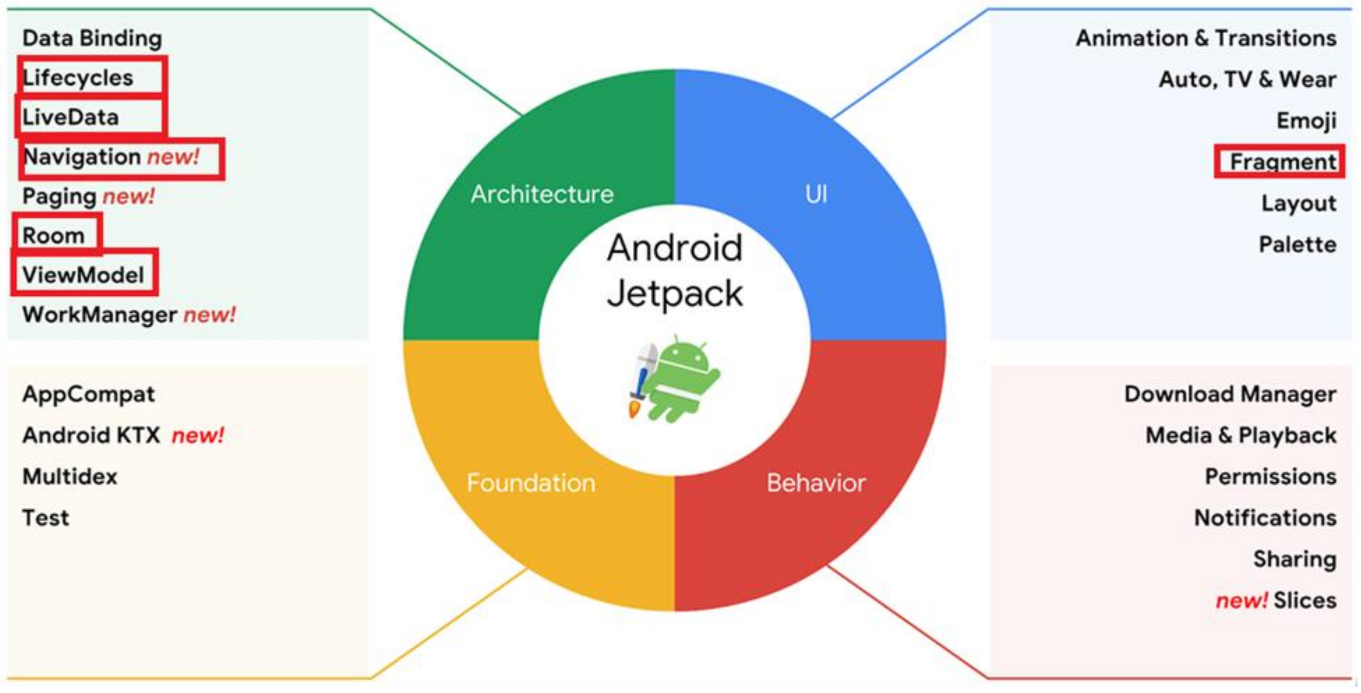
In this project we will start implementing the **Android ROOM Database** that is also a part of *Android Jetpack*.

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.



LiveData: A data holder class that can be observed. Always holds/caches the latest version of data and notifies its observers when data has changed. LiveData is lifecycle aware. UI components just observe relevant data and don't stop or resume observation. LiveData automatically manages all of this since it's aware of the relevant lifecycle status changes while observing.

ViewModel: Acts as a communication center between the Repository (data) and the UI. The UI no longer needs to worry about the origin of the data. ViewModel instances survive Activity/Fragment recreation.

Entity: Annotated class that describes a database table when working with Room.

Room database: Simplifies database work and serves as an access point to the underlying SQLite database (hides SQLiteOpenHelper). The Room database uses the DAO to issue queries to the SQLite database.

SQLite database: On device storage. The Room persistence library creates and maintains this database for you.

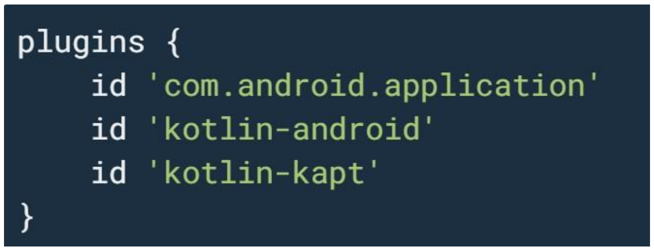
DAO: Data access object. A mapping of SQL queries to functions. When you use a DAO, you call the methods, and Room takes care of the rest.

Repository: A class that you create that is primarily used to manage multiple data sources.

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

1. Apply the kapt annotation processor Kotlin plugin by adding it in the plugins section defined on the top of your build.gradle (Module: app) file.

```
id 'kotlin-kapt'
```



```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
    id 'kotlin-kapt'
}
```

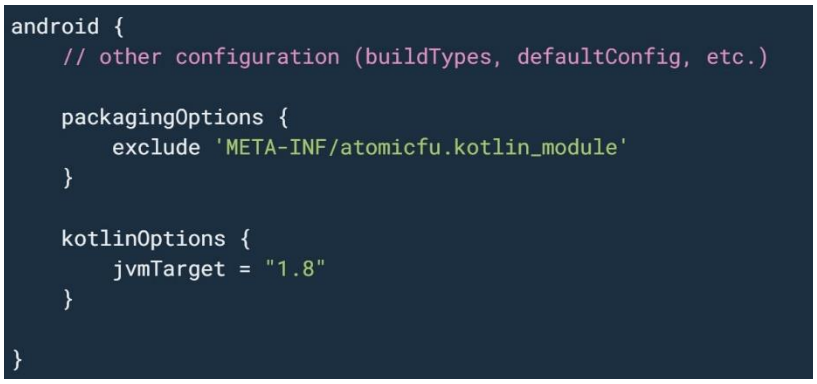
2. Add the "packagingOptions" .

"packagingOptions" will exclude the atomic functions module from the package and prevent warnings

```
packagingOptions {
    exclude 'META-INF/atomicfu.kotlin_module'
}
```

Some of the APIs you will use require 1.8 **jvmTarget**, so add that to the android block as well if it does not exist already.

```
kotlinOptions {
    jvmTarget = '1.8'
}
```



```
android {
    // other configuration (buildTypes, defaultConfig, etc.)

    packagingOptions {
        exclude 'META-INF/atomicfu.kotlin_module'
    }

    kotlinOptions {
        jvmTarget = "1.8"
    }

}
```

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

Reference Link: https://developer.android.com/training/data-storage/room

```
Room components
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'
```

Note***

Get the most current version numbers from the AndroidX releases page.

Room allows you to create tables via an Entity. Let's do it now.

4. Create a new package name as “**entities**” in the “**model**” package. After creating the **entities** package, create a new Kotlin class file called **FavDish** containing the **FavDish** data class with the entities (Dish Items) that we want to insert in the database. Each property in the class represents a column in the table. Room will ultimately use these properties to both create the table and instantiate objects from rows in the database.

5. To make the **FavDish** class meaningful to a Room database, you need to create an association between the class and the database using Kotlin annotations. You will use specific annotations to identify how each part of this class relates to an entry in the database. Room uses this extra information to generate code.
   If you type the annotations yourself (instead of pasting), Android Studio will auto-import the annotation classes.

Note***

If you paste the code in, you can move the cursor over each highlighted error and use the "Project quick fix" keyboard shortcut (**Alt+Enter** on Windows/Linux, **Option+Enter** on Mac) to import classes quickly.

6. Here is the code:

```
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
)
```

Let's see what these annotations do:

- @Entity(tableName = **"fav_dish_table"**) - Each @Entity class represents a SQLite table. Annotate your class declaration to indicate that it's an entity. You can specify the name of the table if you want it to be different from the name of the class. This names the table "**fav_dish_table**".
- @PrimaryKey - Every entity needs a primary key. To keep things simple, each word acts as its own primary key. You can autogenerate unique keys by annotating the primary key.
- @ColumnInfo(name = **"cooking_time"**) - Specifies the name of the column in the table if you want it to be different from the name of the member variable. This names the column " **cooking_time** ".
- Every property that's stored in the database needs to have public visibility, which is the Kotlin default.

You can find a complete list of annotations in the Room package summary reference.

See also Defining data using Room entities.

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

First of all, let's see what is DAO and then continue the project with actually implementing DAO in the project.

What is the DAO?

In the DAO (data access object), you specify SQL queries and associate them with method calls. The compiler checks the SQL and generates queries from convenience annotations for common queries, such as @Insert.

Room uses the DAO to create a clean API for your code.

The DAO must be an interface or abstract class.

By default, all queries must be executed on a separate thread.

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.

Implement the DAO

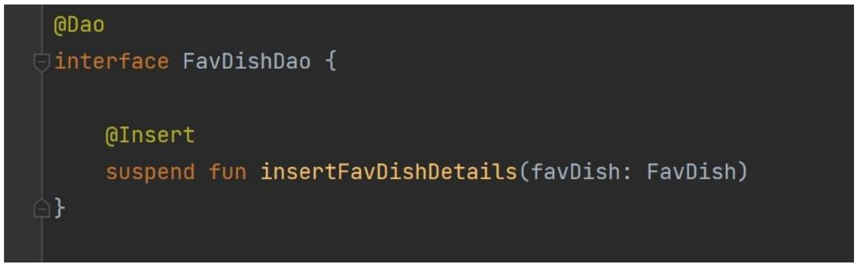
1. Create a new package name as "**database**" in the "**model**" package.

2. Create an interface name as **FavDishDao** that we will use to specify SQL queries and associate them with method calls.
   As we know, DAO must either be an interface or an abstract class.
   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 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.*

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

```
@Insert
suspend fun insertFavDishDetails(favDish: FavDish)
```



```
@Dao
interface FavDishDao {

    @Insert
    suspend fun insertFavDishDetails(favDish: FavDish)
}
```

Let's walk through it:

- FavDishDao is an interface; DAOs must either be interfaces or abstract classes.
- The @Dao annotation identifies it as a DAO class for Room.
- suspend fun insertFavDishDetails(favDish: FavDish) : Declares a suspend function to insert a favorite dish.
- The @Insert annotation is a special DAO method annotation where you don't have to provide any SQL! (There are also @Delete and @Update annotations for deleting and updating rows.)

Learn more about Room DAOs.

Project # 19 **19_ElinaResearchProject**

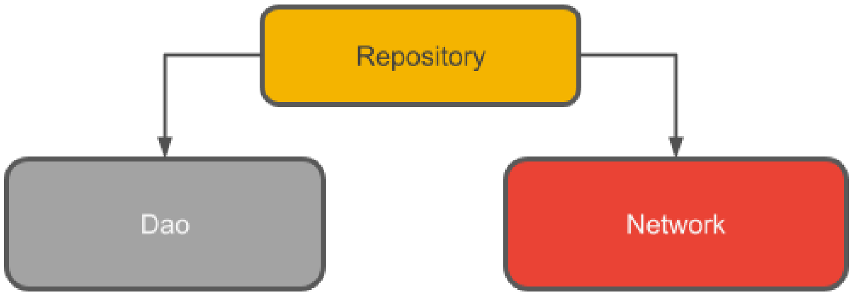
In this project we will create a Kotlin class file called **FavDishRoomDatabase**, as well as a Kotlin class file named as **FavDishRepository**.

What is a Room database?

- Room is a database layer on top of an SQLite database.
- Room takes care of mundane tasks that you used to handle with an SQLiteOpenHelper.
- Room uses the DAO to issue queries to its database.
- By default, to avoid poor UI performance, Room doesn't allow you to issue queries on the main thread. When Room queries return Flow, the queries are automatically run asynchronously on a background thread.
- Room provides compile-time checks of SQLite statements.

What is a Repository?

A repository class abstracts access to multiple data sources. The repository is not part of the Architecture Components libraries but is a suggested best practice for code separation and architecture. A Repository class provides a clean API for data access to the rest of the application.



Why use a Repository?

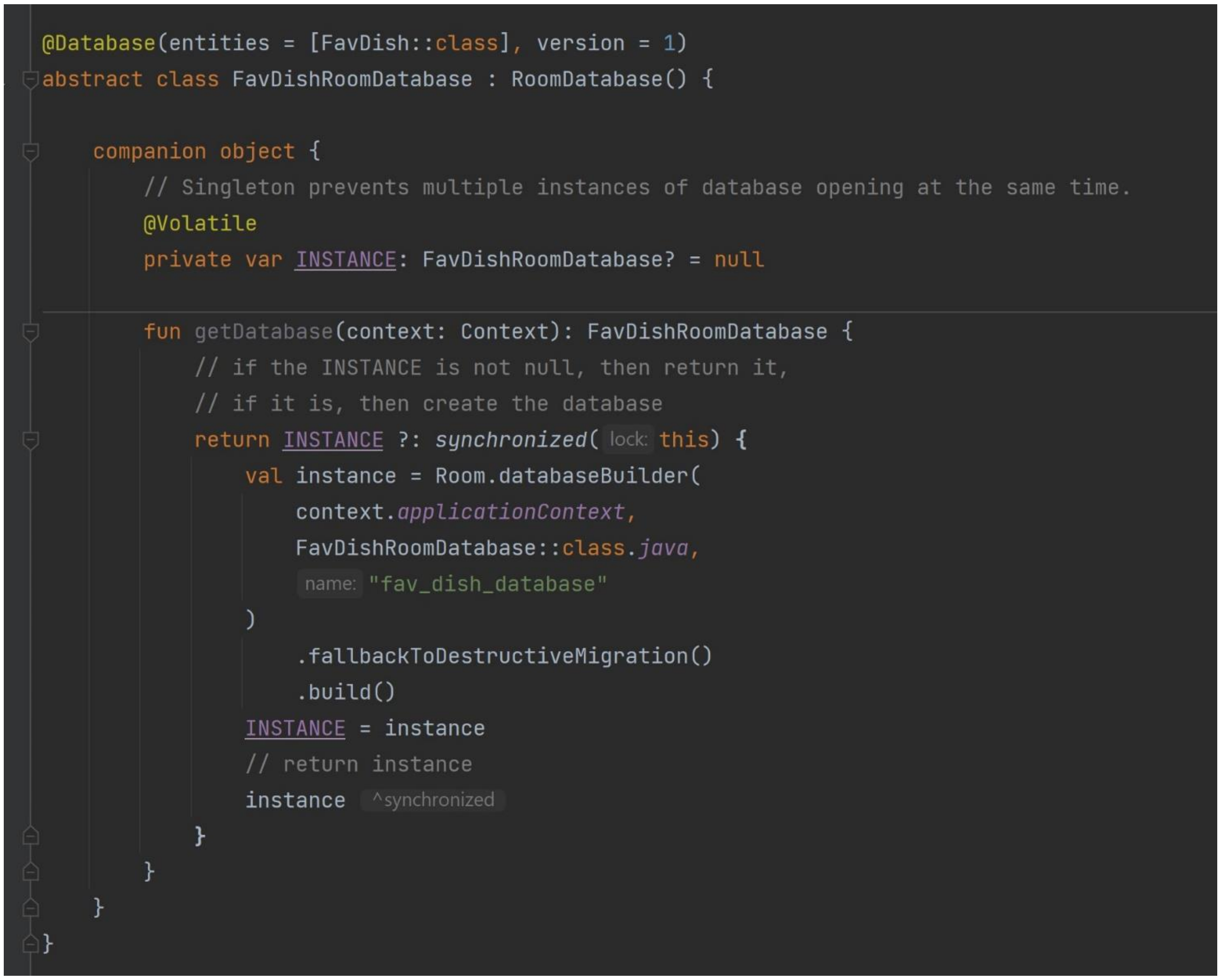
A Repository manages queries and allows you to use multiple backends. In the most common example, the Repository implements the logic for deciding whether to fetch data from a network or use results cached in a local database.

Implement the Room database

Your Room database class must be abstract and extend RoomDatabase. Usually, you only need one instance of a Room database for the whole app.

1. Create a Kotlin class file name called **FavDishRoomDatabase** in “**database**” package and add this code to it:

```
@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
            }
        }
    }
}
```



```kotlin
@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( lock: this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    FavDishRoomDatabase::class.java,
                     name: "fav_dish_database"
                )
                    .fallbackToDestructiveMigration()
                    .build()
                INSTANCE = instance
                // return instance
                instance  ^synchronized
            }
        }
    }
}
```

Let's walk through the code:

- You annotate the class to be a Room database with @Database and use the annotation parameters to declare the entities that belong in the database and set the version number. Each entity corresponds to a table that will be created in the database.
- You define a singleton, FavDishRoomDatabase, to prevent having multiple instances of the database opened at the same time.
- getDatabase returns the singleton. It'll create the database the first time it's accessed, using Room's database builder to create a RoomDatabase object in the application context from the FavDishRoomDatabase class and names it "fav_dish_database".

Note***

When you modify the database schema, you'll need to update the version number and define a migration strategy.

For example, a destroy and re-create strategy can be sufficient. But for a real app, you must implement a migration strategy. See Understanding migrations with Room.

In Android Studio, if you get errors when you paste code or during the build process, select **Build >Clean Project**. Then select **Build > Rebuild Project**, and then build again.

Implementing the Repository

2. Create a Kotlin class file name called **FavDishRepository** in "**database**" package. Declare the **DAO** as a *private property* in the repository constructor. Pass in the DAO instead of the whole database, because you only need access to the DAO, since the DAO contains all the read/write methods for the database.

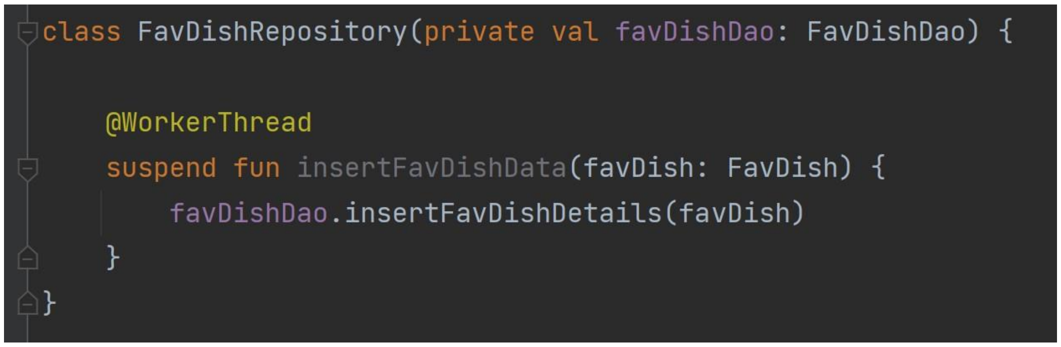
```
class FavDishRepository(private val favDishDao: FavDishDao) {...}
```

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

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

The suspend modifier tells the compiler that this needs to be called from a coroutine or another suspending function.

By default Room executes suspend queries off the main thread. Therefore, we don't need to implement anything else to ensure we're not doing long running database work off the main thread.



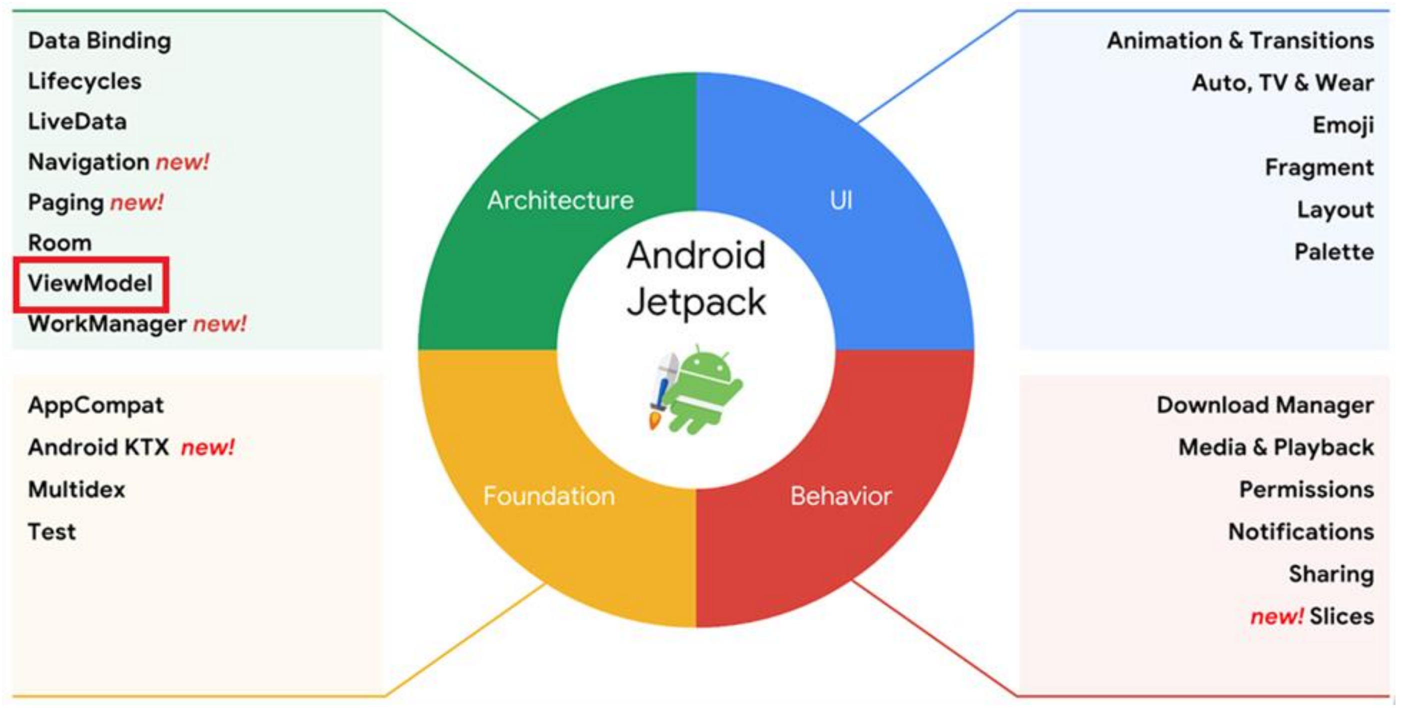
```kotlin
class FavDishRepository(private val favDishDao: FavDishDao) {

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

Repositories are meant to mediate between different data sources. In this simple example, you only have one data source, so the Repository doesn't do much.

See the BasicSample for a more complex implementation.

Project # 20 **20_ElinaResearchProject**

In this project we will create a **ViewModel** class to define the required function. Also, we will create an application class to create a database instance.



1. Create a Kotlin class file name as FavDishViewModel in “viewmodel” package.

- The ViewModel is part of the lifecycle library.

- The ViewModel's role is to provide data to the UI and survive configuration changes.

- You can also use a ViewModel to share data between fragments.

- A ViewModel acts as a communication center between the Repository and the UI.

```
class FavDishViewModel(private val repository: FavDishRepository)
: ViewModel() { ...

}
```

2. In this FavDishViewModel class create a function to deal with the repository while data insertion.

   Launching a new coroutine to insert data.

```
fun insert(dish: FavDish) = viewModelScope.launch {
    // Call the repository function and pass the details.
    repository.insertFavDishData(dish)
}
```

3. Create a ViewModelFactory provider class.

To create the ViewModel we implement a ViewModelProvider.Factory that gets as a parameter the dependencies needed to create FavDishViewModel: FavDishRepository.

By using viewModels and ViewModelProvider.Factory then the framework will take care of the lifecycle of the ViewModel.

It will survive configuration changes and even if the Activity is recreated, you'll always get the right instance of the FavDishViewModel class.

```
class FavDishViewModelFactory(private val repository:
FavDishRepository) : ViewModelProvider.Factory {
  override fun <T : ViewModel> create(modelClass: Class<T>):T{
    if(modelClass.isAssignableFrom(FavDishViewModel::class.java){
         @Suppress("UNCHECKED_CAST")
       return FavDishViewModel(repository) as T
    }
    throw IllegalArgumentException("Unknown ViewModel class")
  }

}
```

4. Go to FavDishRoomDatabase abstract class in “**database**” package and create abstract function that we can access from the application class to initialize the repository class.

```
abstract fun favDishDao(): FavDishDao
```

5. Create an “application” package and FavDishApplication class in “application” package,where we can define the variable scope to use throughout the application.

```
class FavDishApplication : Application() {…

}
```

6. Define the variables to create an instance of Room database and use it throughout the application.

```
private val database by lazy {

   FavDishRoomDatabase.getDatabase(this@FavDishApplication)

}
```

A variable for repository.

```
val repository by lazy {

   FavDishRepository(database.favDishDao())

}
```

Using "by lazy" so the database and the repository are only created when they're needed rather than when the application starts.

The "lazy" keyword used for creating a new instance that uses the specified initialization function and the default thread-safety mode [LazyThreadSafetyMode.SYNCHRONIZED].

If the initialization of a value throws an exception, it will attempt to reinitialize the value at next access.

Note that the returned instance uses itself to synchronize on. Do not synchronize from external code on the returned instance as it may cause accidental deadlock.

7. Go to AndroidManifest and link the application class with application. Add the name parameter in the application tag.

```
android:name=".application.FavDishApplication"
```

Project # 21 **21_ElinaResearchProject**

In this project we will finally insert the dish details into the database and print the success message in Log and Display the Toast.

1. Go to AddUpdateDishActivity and create an instance of the ViewModel class so that we can access its methods in our View class.

```
private val mFavDishViewModel: FavDishViewModel by viewModels {
    FavDishViewModelFactory((application as

    FavDishApplication).repository)
}
```

To create the ViewModel we used the viewModels delegate, passing in an instance of our FavDishViewModelFactory.
This is constructed based on the repository retrieved from the FavDishApplication.

2. Go to “**utils**” package and in *Constants* object create constants that we will use for the Image Source while inserting the Dish details.

```
const val DISH_IMAGE_SOURCE_LOCAL: String = "Local"
const val DISH_IMAGE_SOURCE_ONLINE: String = "Online"
```

3. Go to AddUpdateDishActivity and create an instane of entity class and pass the required values to it. Also, remove the Toast Message.

```
val favDishDetails: FavDish = FavDish(
    mImagePath,
    Constants.DISH_IMAGE_SOURCE_LOCAL,
    title,
    type,
    category,
    ingredients,
    cookingTimeInMinutes,
    cookingDirection,
    false
)
```

4. Pass the value to the ViewModelClass and display the Toast message to acknowledge.

```
mFavDishViewModel.insert(favDishDetails)

Toast.makeText(
    this@AddUpdateDishActivity,
    "You successfully added your favorite dish details.",
    Toast.LENGTH_SHORT
).show()
```

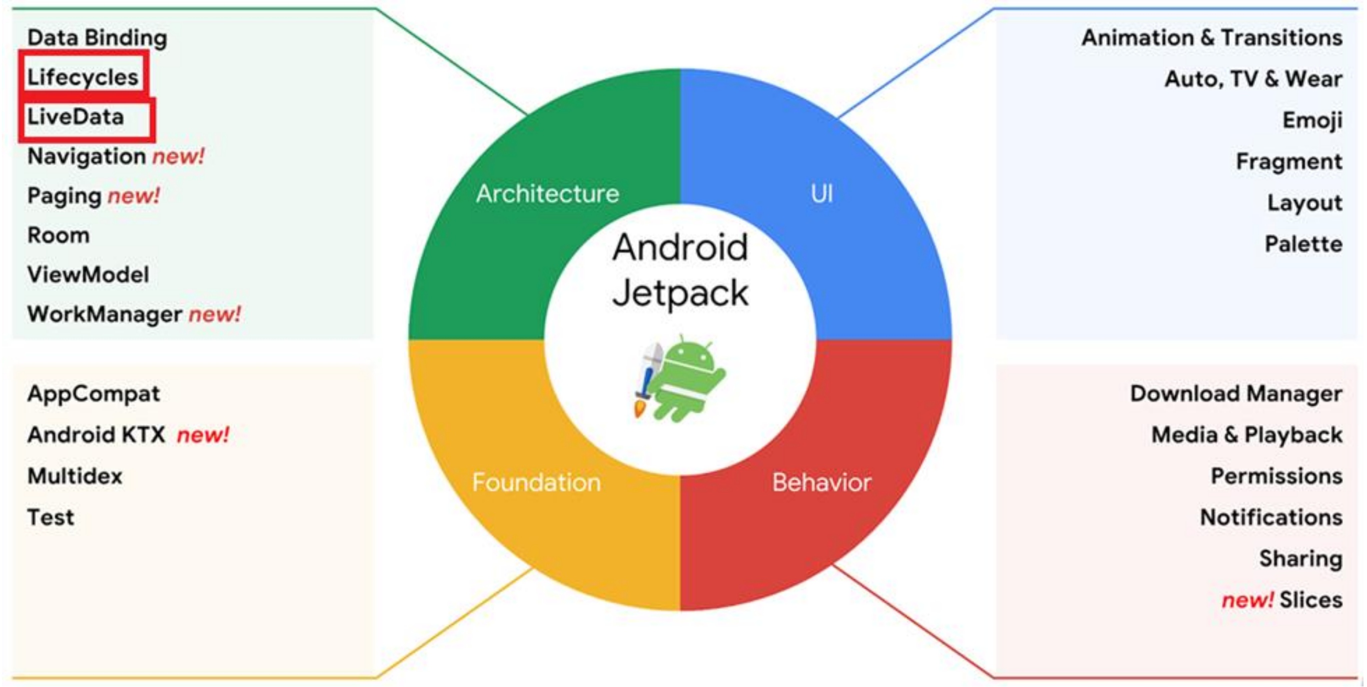
You even print the log if Toast is not displayed on emulator

```
Log.e("Insertion", "Success")
finish()
```

Project # 22 **22_ElinaResearchProject**

In this project we will get the list of inserted dishes from the local database and print them in the Log for now.

We will use the LifeCycle and LiveData library of Android Jetpack.



1. Go to FavDishDao inerface and create a function to get the list of dishes from database using @Query.

```
@Query("SELECT * FROM FAV_DISHES_TABLE ORDER BY ID")
fun getAllDishesList(): Flow<List<FavDish>>
```

- When data changes, you usually want to take some action, such as displaying the updated data in the UI.

- This means you must observe the data so when it changes, you can react.

- To observe data changes, we will use **Flow** from *kotlinx-coroutines*.

- Use a return value of type **Flow** in your method description, and **Room** generates all necessary code to update the **Flow** when the database is updated.

- A **Flow** is an async sequence of values

- **Flow** produces values one at a time (instead of all at once) that can generate values from async operations like network requests, database calls, or other async code.

- It supports *coroutines* throughout its API, so you can transform a flow using coroutines as well.

2. Go to FavDishRepository and create a variable for the dishes list to access it from ViewModel.

Room executes all queries on a separate thread.

Observed Flow will notify the observer when the data has changed.

```
val allDishesList: Flow<List<FavDish>> =
favDishDao.getAllDishesList()
```

3. Get all the dishes list from the database in the ViewModel to pass it to the UI. Go to FavDIshViewModel and write the code as follows:

```
val allDishesList: LiveData<List<FavDish>> =
repository.allDishesList.asLiveData()
```

Using LiveData and caching what allDishes returns has several benefits:

- We can put an observer on the data (instead of polling for changes) and only update the UI when the data changes.

- Repository is completely separated from the UI through the ViewModel.

4. Go to AllDishesFragment and create a ViewModel instance to access the methods.

```
private val mFavDishViewModel: FavDishViewModel by viewModels {
    FavDishViewModelFactory((requireActivity().application as
FavDishApplication).repository)
}
```

- To create the ViewModel we used the viewModels delegate, passing in an instance of our FavDishViewModelFactory.

- This is constructed based on the repository retrieved from the FavDishApplication.

5. In AllDishesFragment override the onViewCreated method and get the dishes list and print the title in Log for now.

```
override fun onViewCreated(view: View, savedInstanceState:
Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    mFavDishViewModel.allDishesList.observe(viewLifecycleOwner) {
dishes ->
        dishes.let {
            for (item in it){
                Log.i("Dish Title", "${item.id} ::
                ${item.title}")
            }
        }
    }
}
```

- Add an observer on the LiveData returned by getAllDishesList.

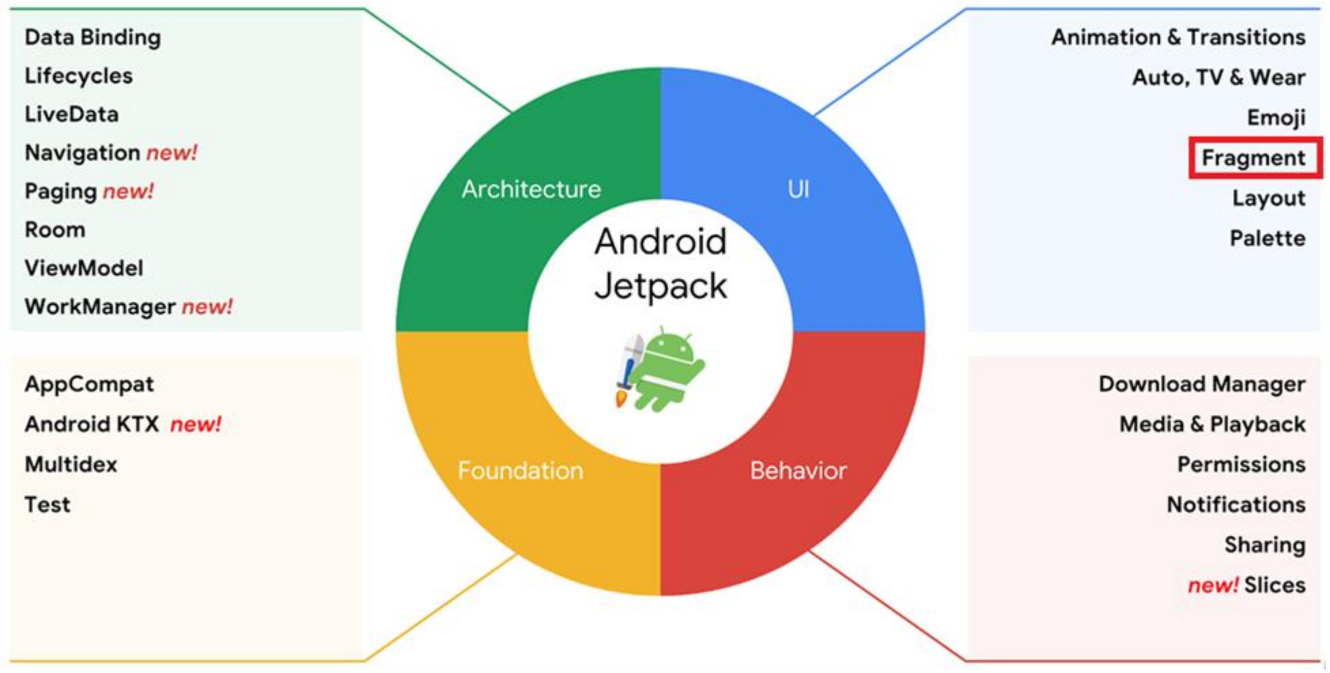
- The onChanged() method fires when the observed data changes and the activity is in the foreground.

Project # 23 **23_ElinaResearchProject**

In this project we will display the list of dishes into UI that we have printed in the Log in the last version.

We will have a look at the fragment library of Android Jetpack.

We will also implement the ViewBinding in fragment display the list of inserted dishes into UI using RecyclerView.



1. Design all dishes Fragment layout. In **fragment_all_dishes.xml** in "**layout**" create RecyclerView with id *rv_dishes_list*, which will contain one TextView with id *tv_no_dishes_yet*. The latter will display "No dishes added yet!!!" when there is no dish to display to the user in the home page.

2. Add the string value in the strings.xml value.

```
<string name="label_no_dishes_added_yet">No dishes added
yet!!!</string>
```

3. Design the dish item layout. Create **item_dish_layout.xml** in “**layout**”, which will contain a CardView with a LinearLayout that includes one ImageView and one TextView with ids iv_dish_image and tv_dish_title respectively. ImageView will display the dish image and TextView will display the title of the dish.

4. In “adapters” package create favorite dish adapter. Create a class **FavDishAdapter** that takes a parameter type Fragment and extends RecyclerView Adapter.

```
class FavDishAdapter(private val fragment: Fragment) :
    RecyclerView.Adapter<FavDishAdapter.ViewHolder>() {...}
```

In the **FavDishAdapter** class

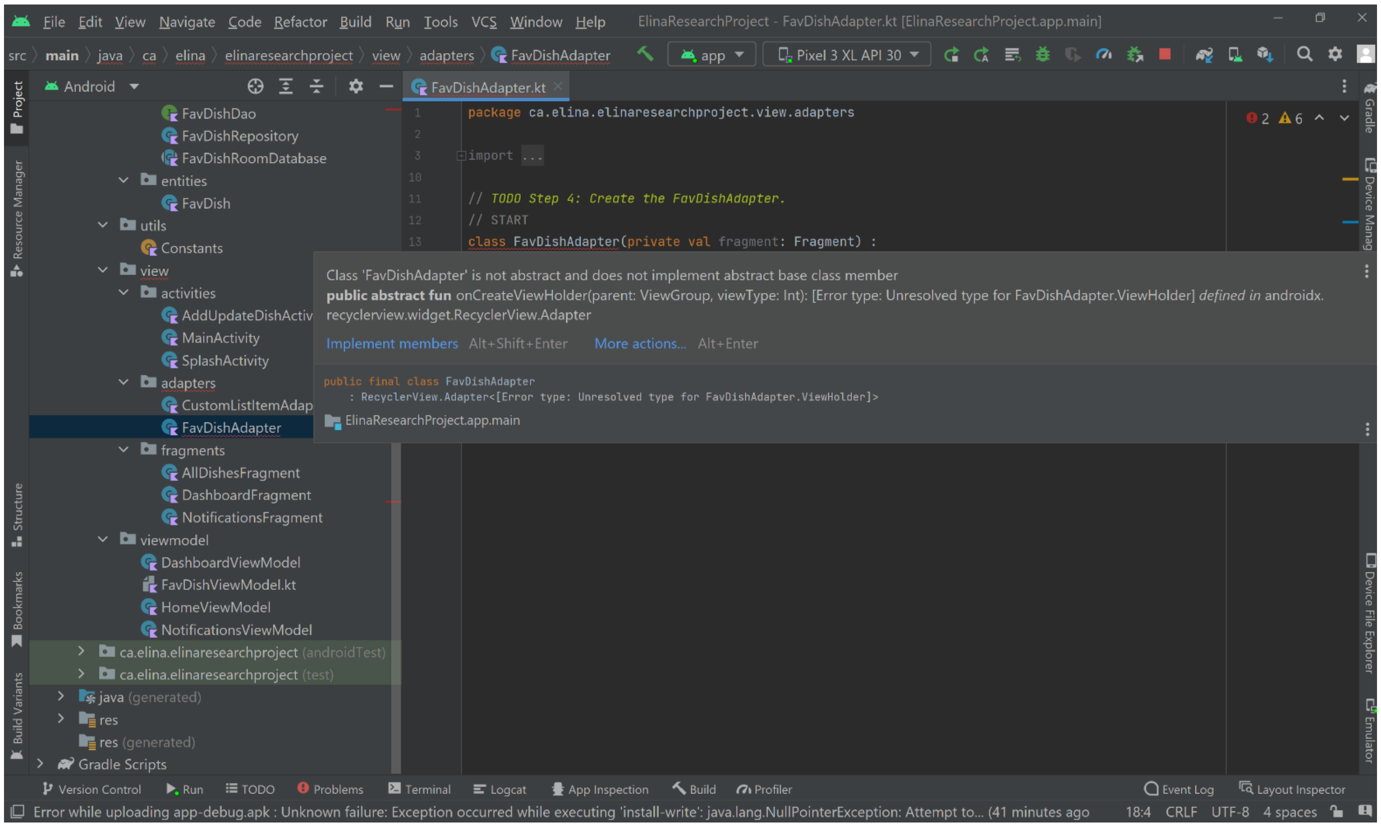
- create a variable **dishes**, which will be a list of dishes.

```
private var dishes: List<FavDish> = listOf()
```

- create our own ViewHolder that describes an item view and metadata about its place within the RecyclerView. It will take an argument view which is **`ItemDishLayoutBinding`** and it is our **item_dish_layout.xml**.

```
class ViewHolder(view: ItemDishLayoutBinding) :
RecyclerView.ViewHolder(view.root) {

    val ivDishImage = view.ivDishImage
    val tvTitle = view.tvDishTitle
}
```

- implement abstract class members



```
package ca.elina.elinaresearchproject.view.adapters

import ...

// TODO Step 4: Create the FavDishAdapter.
// START
class FavDishAdapter(private val fragment: Fragment) :
```

Class 'FavDishAdapter' is not abstract and does not implement abstract base class member **public abstract fun** onCreateViewHolder(parent: ViewGroup, viewType: Int): [Error type: Unresolved type for FavDishAdapter.ViewHolder] *defined in* androidx.recyclerview.widget.RecyclerView.Adapter

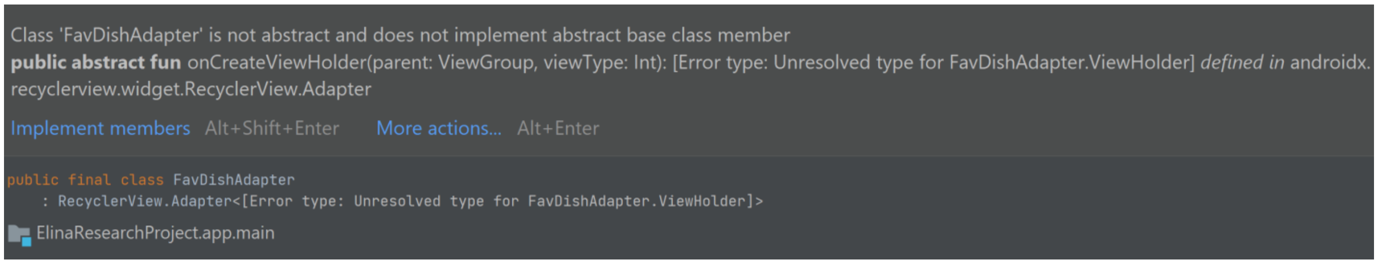
Implement members Alt+Shift+Enter More actions... Alt+Enter

```
public final class FavDishAdapter
    : RecyclerView.Adapter<[Error type: Unresolved type for FavDishAdapter.ViewHolder]>
```

ElinaResearchProject.app.main

Error while uploading app-debug.apk : Unknown failure: Exception occurred while executing 'install-write': java.lang.NullPointerException: Attempt to... (41 minutes ago

You will get the warning that you need to implement missing members.



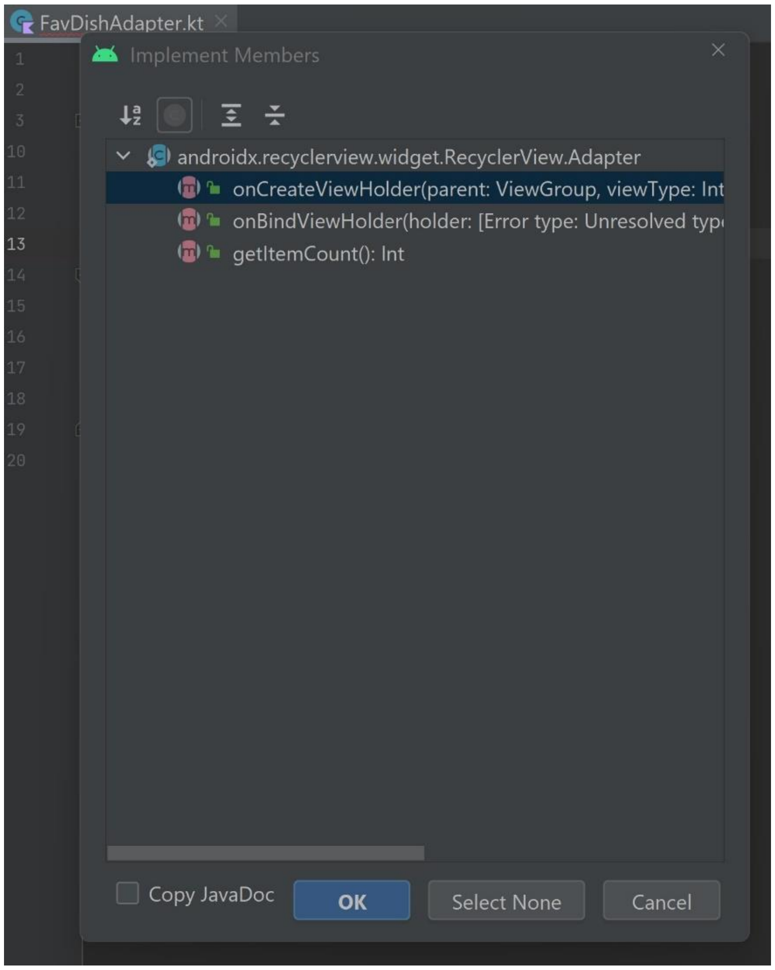
Class 'FavDishAdapter' is not abstract and does not implement abstract base class member
public abstract fun onCreateViewHolder(parent: ViewGroup, viewType: Int): [Error type: Unresolved type for FavDishAdapter.ViewHolder] *defined in* androidx.recyclerview.widget.RecyclerView.Adapter

Implement members Alt+Shift+Enter More actions... Alt+Enter

```
public final class FavDishAdapter
    : RecyclerView.Adapter<[Error type: Unresolved type for FavDishAdapter.ViewHolder]>
```

ElinaResearchProject.app.main

- Click on implement members and choose all three of them.



androidx.recyclerview.widget.RecyclerView.Adapter
onCreateViewHolder(parent: ViewGroup, viewType: Int
onBindViewHolder(holder: [Error type: Unresolved typ
getItemCount(): Int

- override onCreateViewHolder, which Inflates the item views which is designed in xml layout file (item_dish_layout.xml) and initializes some private fields to be used by RecyclerView.

```
override fun onCreateViewHolder (parent: ViewGroup,
viewType:Int):  ViewHolder {
    val binding: ItemDishLayoutBinding =  ItemDishLayoutBinding

        .inflate(LayoutInflater.from(fragment.context), parent,

        false)
     return ViewHolder(binding)

}
```

- override onBindViewHolder, that binds each item in ArrayLisr to a view.

It is called when RecyclerView needs a new {@link ViewHolder} of the given type to represent an item.

This new ViewHolder should be constructed with a new View that can represent the items of the given type.

You can either create a new View manually or inflate it from an XML layout file.

```
override fun onBindViewHolder(holder: ViewHolder, position: Int)
{
    val dish = dishes[position]

    Load the dish image in the ImageView.
    Glide.with(fragment)
        .load(dish.image)
        .into(holder.ivDishImage)

    holder.tvTitle.text = dish.title
}
```

- override getItemCount, which gets the number of items in the list.

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

5. In AllDishaesFragment remove the unused code and create a global variable for the ViewBinding.

```
private lateinit var mBinding: FragmentAllDishesBinding
```

6. In AllDishaesFragment in the function onCreateView remove the extra code and initialize the mBinding variable.

```
View {
    mBinding = FragmentAllDishesBinding
        .inflate(inflater, container, false)
    return mBinding.root
}
```

7. In AllDishaesFragment in the function onViewCreated initialize the RecyclerView and bind the adapter class.

```
Set the LayoutManager that this RecyclerView will use.
mBinding.rvDishesList.layoutManager =
GridLayoutManager(requireActivity(), 2)

Adapter class is initialized and list is passed in the param.
val favDishAdapter = FavDishAdapter(this@AllDishesFragment)

adapter instance is set to the recyclerview to inflate the items.
mBinding.rvDishesList.adapter = favDishAdapter
```

8. Create a function that will have the updated list of dishes that we will bind to the adapter class. In FavDishAdapter write the following code:

```
fun dishesList(list: List<FavDish>) {
    dishes = list
    notifyDataSetChanged()
}
```

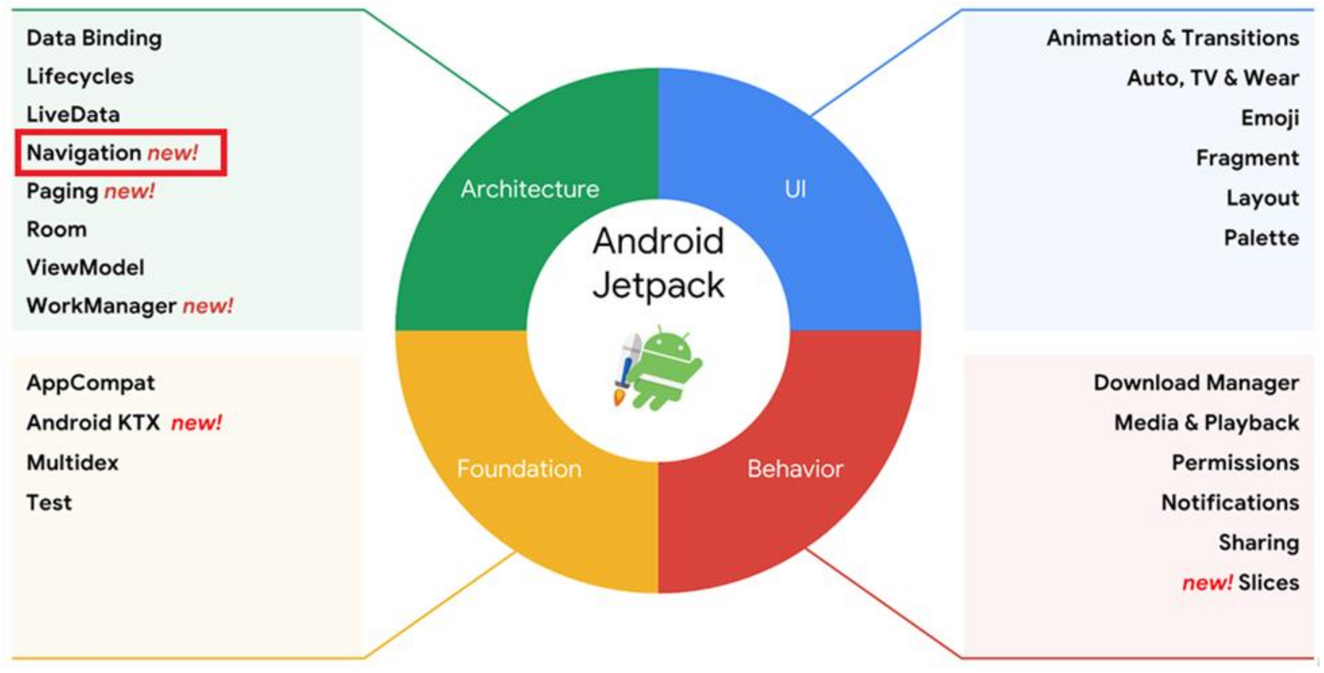
9. In AllDishaesFragment pass the dishes list to the adapter class.

```
mFavDishViewModel.allDishesList.observe(viewLifecycleOwner) {
dishes ->
    dishes.let {

        Pass the dishes list to the adapter class.
        // START
        if (it.isNotEmpty()) {

            mBinding.rvDishesList.visibility = View.VISIBLE
            mBinding.tvNoDishesAddedYet.visibility = View.GONE

            favDishAdapter.dishesList(it)
        } else {

            mBinding.rvDishesList.visibility = View.GONE
            mBinding.tvNoDishesAddedYet.visibility= View.VISIBLE
        }
        // END
    }
}
```

Project # 24 **24_ElinaResearchProject**

In this project we will refactor some labels and icons that we will use in our app. So it makes more sense while building the application further.

We will also get acquainted with the concept of Navigation Library a part of Android Jetpack while refactoring.



Overview of **Navigation** from official Documentation of Android Jetpack. For more information, please visit https://developer.android.com/guide/navigation

Navigation refers to the interactions that allow users to navigate across, into, and back out from the different pieces of content within your app. Android Jetpack's Navigation component helps you implement navigation, from simple button clicks to more complex patterns, such as app bars and the navigation drawer. The Navigation component also ensures a consistent and predictable user experience by adhering to an established set of principles.

The Navigation component consists of three key parts that are described below:

- Navigation graph: An XML resource that contains all navigation-related information in one centralized location. This includes all of the individual content areas within your app, called *destinations*, as well as the possible paths that a user can take through your app.
- NavHost: An empty container that displays destinations from your navigation graph. The Navigation component contains a default NavHost implementation, NavHostFragment, that displays fragment destinations.
- NavController: An object that manages app navigation within a NavHost. The NavController orchestrates the swapping of destination content in the NavHost as users move throughout your app.

As you navigate through your app, you tell the NavController that you want to navigate either along a specific path in your navigation graph or directly to a specific destination. The NavController then shows the appropriate destination in the NavHost.

The Navigation component provides a number of other benefits, including the following:

- Handling fragment transactions.
- Handling Up and Back actions correctly by default.
- Providing standardized resources for animations and transitions.
- Implementing and handling deep linking.
- Including Navigation UI patterns, such as navigation drawers and bottom navigation, with minimal additional work.
- Safe Args - a Gradle plugin that provides type safety when navigating and passing data between destinations.
- ViewModel support - you can scope a ViewModel to a navigation graph to share UI-related data between the graph's destinations.

In addition, you can use Android Studio's Navigation Editor to view and edit your navigation graphs.

Steps to implement our project are:

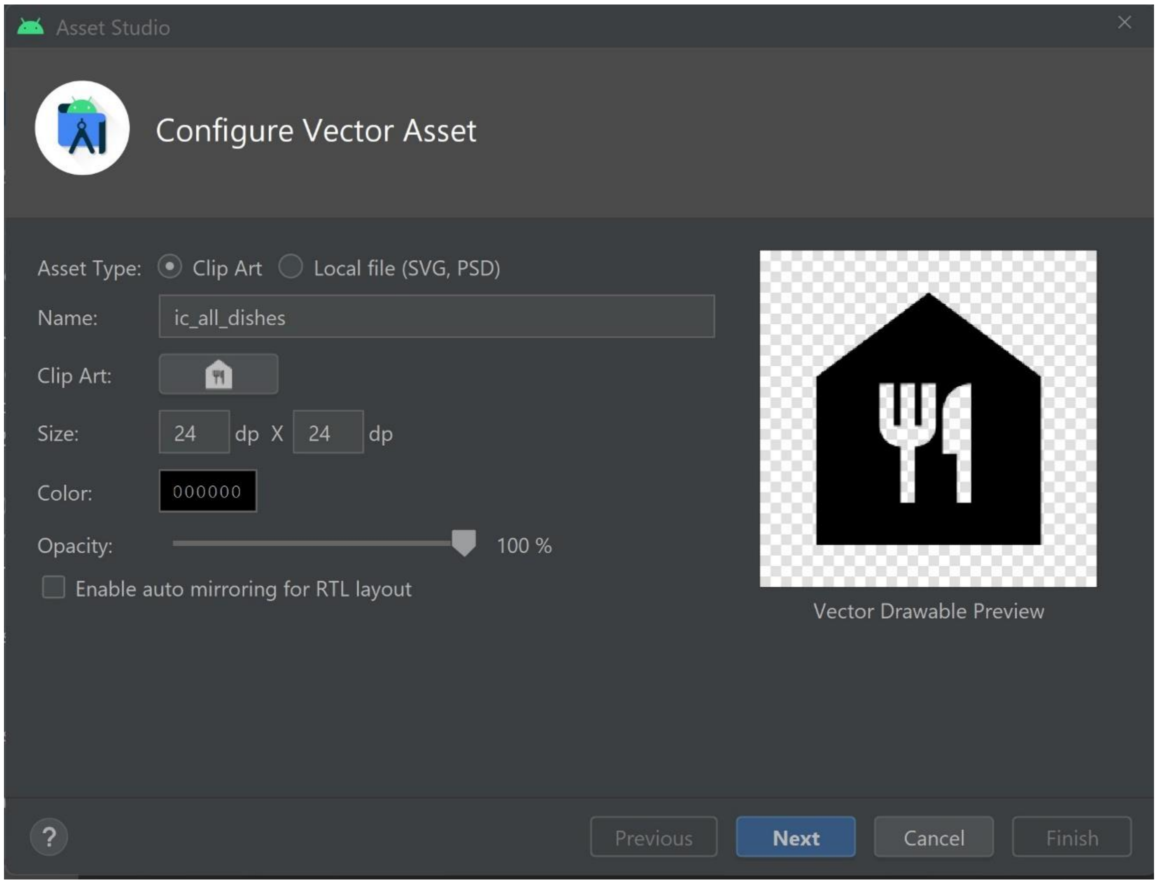
1. Go to the “**fragments**” package and refactor *DashBoardFragment* to **FavoriteDishesFragment**.
2. Go to the “**fragments**” package and refactor *NotificationsFragment* to **RandomDishFragment**.
3. Go to the **res** -> **layout** and refactor *fragment_dashboard.xml* to **fragment_favorite_dishes.xml**.
4. Go to the **res** -> **layout** and refactor *fragment_notifications.xml* to **fragment_random_dish.xml**.
5. Go to the **res** -> **navigation** and in mobile_navigation.xml rename the ids of fragment as per the name of Fragments as below and also update their label in the string.xml file. Also refactor the "app:startDestination" as below:

```
<navigation

     ...

     app:startDestination="@+id/navigation_all_dishes" >


<fragment
    android:id="@+id/navigation_all_dishes"

     ...

     tools:layout="@layout/fragment_all_dishes"/>

<fragment
    android:id="@+id/navigation_favorite_dishes"

     ...

     tools:layout="@layout/fragment_favorite_dishes"/>

<fragment
    android:id="@+id/navigation_random_dish"

     ...

     tools:layout="@layout/fragment_random_dish"/>
```

6. Go to the **strings.xml** and update the titles as follows:

```
<string name="title_all_dishes">All Dishes</string>
<string name="title_favorite_dishes">Favorite</string>
<string name="title_random_dish">Random Dish</string>
```

7. Add a new vector in drawable for *all dishes icon* and name it as **ic_all_dishes.xml**. You can choose any icon from Vector Asset Clip Art or download a new vector from other open sources.


Asset Studio

Configure Vector Asset

Asset Type: [✓ Clip Art] Local file (SVG, PSD)
Name: ic_all_dishes
Clip Art:
Size: 24 dp X 24 dp
Color: 000000
Opacity: 100 %
Enable auto mirroring for RTL layout

For this project we use all vectors from *Clip Art*. And we chose color **black**.

8. Add a new vector in drawable for *favorite dish icon* and name it as **ic_favorite_dish.xml**. You can choose any icon from Vector Asset Clip Art or download a new vector from other open sources.

9. Add a new vector in drawable for *random dish icon* and name it as **ic_random_dish.xml**. You can choose any icon from Vector Asset Clip Art or download a new vector from other open sources.

10. Go to the **res** -> **menu** and update the icons that we have added in drawable.

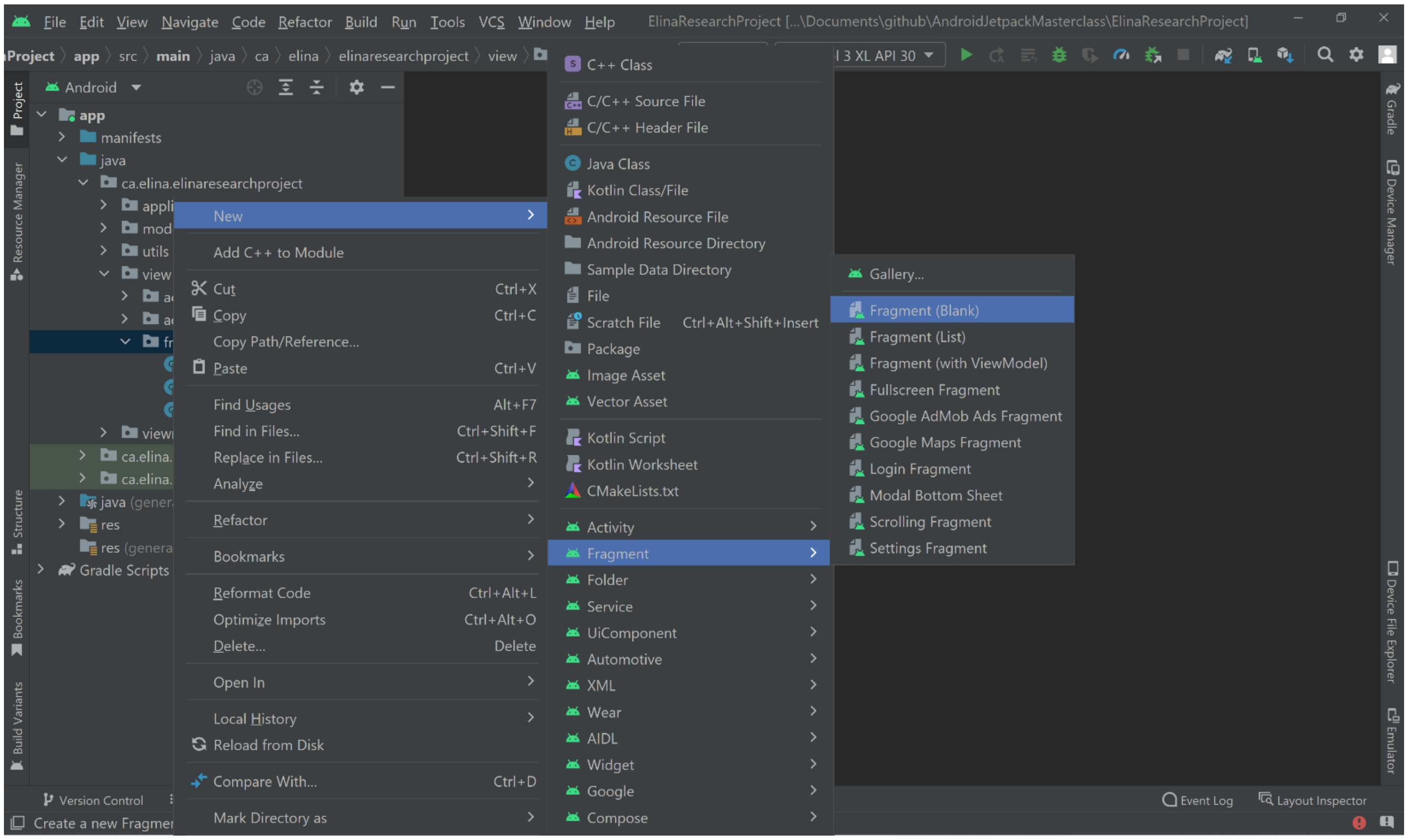
```
<item
    android:id="@+id/navigation_all_dishes"
    android:icon="@drawable/ic_all_dishes"
    android:title="@string/title_all_dishes" />

<item
    android:id="@+id/navigation_favorite_dishes"
    android:icon="@drawable/ic_favorite_dish"
    android:title="@string/title_favorite_dishes" />

<item
    android:id="@+id/navigation_random_dish"
    android:icon="@drawable/ic_random_dish"
    android:title="@string/title_random_dish" />
```

Project # 25 **25_ElinaResearchProject**

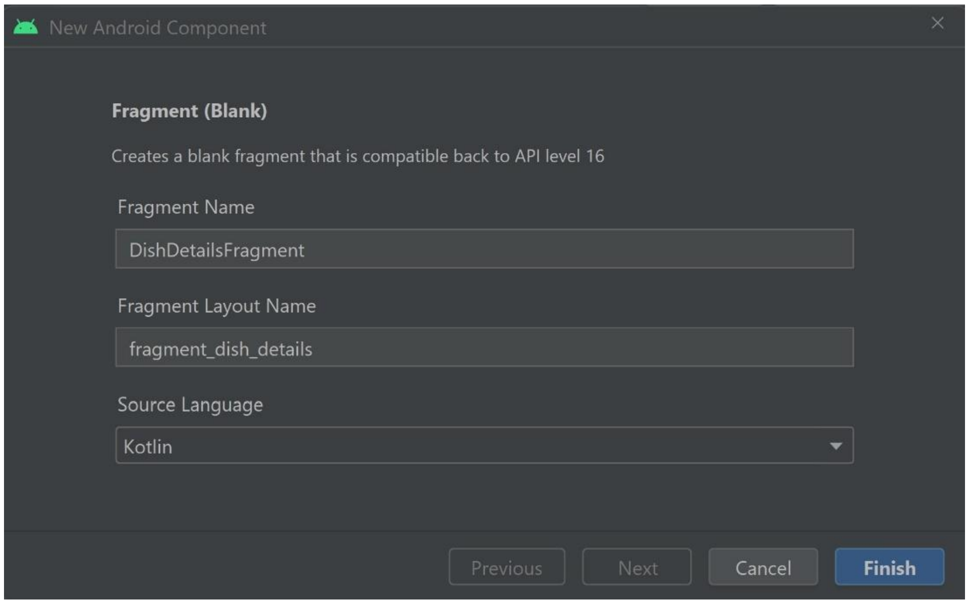
In this project we will create a DishDetailsFragment and navigate to it from the list of items using Navigation.

1. Create a new fragment in "**fragments**" package and name it **DishDetailsFragment**.

Right click on "**fragments**" package -> **New** -> **Fragment** -> **Fragment (Blank)**



Put *Fragment Name* as **DishDetailsFragment**.



Fragment (Blank)

Creates a blank fragment that is compatible back to API level 16

Fragment Name: DishDetailsFragment

Fragment Layout Name: fragment_dish_details

Source Language: Kotlin

Remove all the unused code that is added by Android Studio and write the code below:

```
class DishDetailsFragment : Fragment() {

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

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_dish_details,

        container, false)
    }
}
```

2. Go to the **res** -> **layout** and align the text to the center and add a text "Dish Details Fragment Layout".

```
android:gravity="center"

android:text="Dish Details Fragment Layout"
```

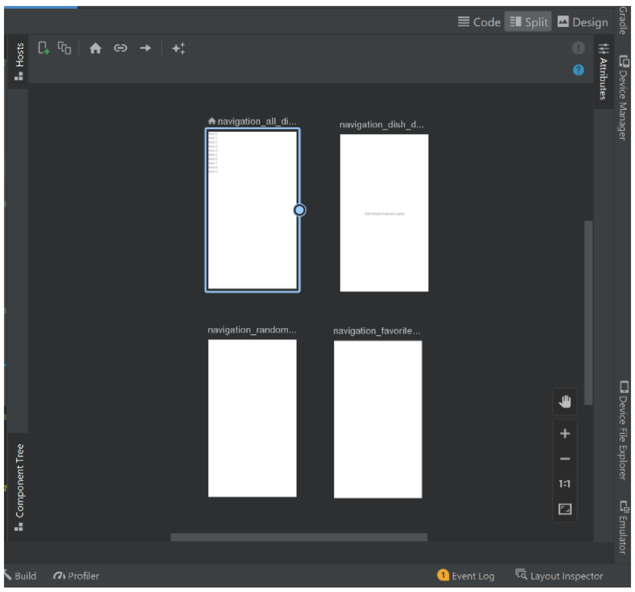
3. Go to the **navigation** -> **mobile_navigation.xml** and Add the Dish Details Fragment in navigation view.

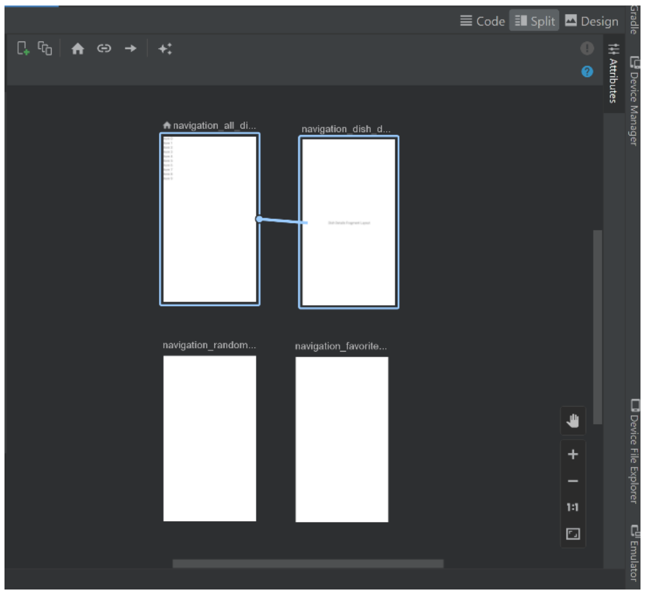
```
<fragment
  android:id="@+id/navigation_dish_details"
  android:name="your.package.view.fragments.DishDetailsFragment"
  android:label="@string/title_dish_details"
  tools:layout="@layout/fragment_dish_details"/>
```

4. Go to the **values** -> **strings.xml** and add the title to the dish details.

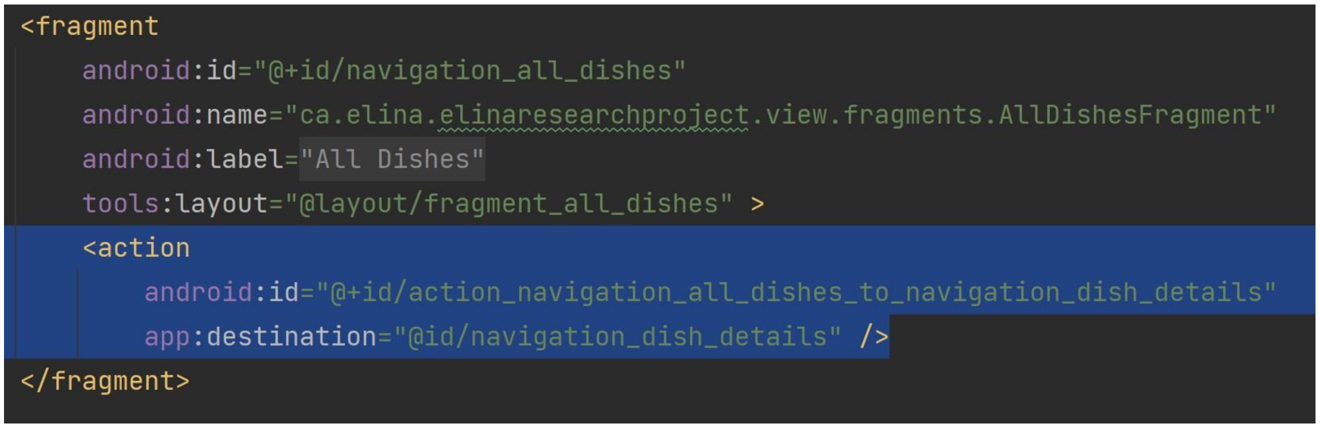
```
<string name="title_dish_details">Dish Details</string>
```

5. Go to the **navigation** -> **mobile_navigation.xml** and add the action to navigate to the *Dish Details Fragment* from *All Dishes* Fragment.





Drag from navigation all dishes and drop to navigation dish details to create an action.



```
<fragment
    android:id="@+id/navigation_all_dishes"
    android:name="ca.elina.elinaresearchproject.view.fragments.AllDishesFragment"
    android:label="All Dishes"
    tools:layout="@layout/fragment_all_dishes" >
    <action
        android:id="@+id/action_navigation_all_dishes_to_navigation_dish_details"
        app:destination="@id/navigation_dish_details" />
</fragment>
```

You will get the fragment action with default id so you need to change it to:

```
<action
    android:id="@+id/action_all_dishes_to_dish_details"
    app:destination="@id/navigation_dish_details" />
```

After adding the action re-build the project to generate the required classes.
To view the auto generate class you can go to "java (generated) and
" your.package.view.fragments" under the project package structure.

6. Go to gradle(Module: app) and add in plugins:

either

```
id("androidx.navigation.safeargs.kotlin")
```

or

```
id 'androidx.navigation.safeargs.kotlin'
```

7. Go to gradle(Project) and add:

either

```
buildscript {
    dependencies {
        classpath("androidx.navigation:navigation-safe-args-
gradle-plugin:2.5.3")
    }
}
```

Note*** this should be before plugins

or

```
id 'androidx.navigation.safeargs.kotlin' version '2.5.3' apply
false
```

8. Go to AllDishesFragment and create a function to navigate to the Dish Details Fragment.

```
fun dishDetails(){

    findNavController()
        .navigate(AllDishesFragmentDirections

        .actionAllDishesToDishDetails())

}
```

9. Go to FavDishAdapter and in the function onBindViewHolder assing the click event to the ItemView and perform the required action.

```
holder.itemView.setOnClickListener {
    if (fragment is AllDishesFragment) {
        fragment.dishDetails()
    }
}
```

Project # 26 **26_ElinaResearchProject**

In this project we will handle the click event of the home back button and hide the BottomNavigationView in the DishDetailsFragment. As you have checked in the previous version, the home back button is not working, and BottomNavigationView is visible in the DishDetailsFragment.

1. In MainActivity create ViewBinding variable.

```
private lateinit var mBinding: ActivityMainBinding
```

2. In MainActivity create the NavController variable.

```
private lateinit var mNavController: NavController
```

3. In MainActivity initialize the mBinding variable and set the content view.

```
mBinding = ActivityMainBinding.inflate(layoutInflater)
setContentView(mBinding.root)
```

4. In MainActivity remove the code below as we are going to use ViewBinding.

```
val navView: BottomNavigationView = findViewById(R.id.nav_view)
```

5. In MainActivity override the onSupportNavigateUp method.

```
override fun onSupportNavigateUp(): Boolean {...}
```

6. In MainActivity add the navigateUp and pass the required parameters.
   This will navigate the user from DishDetailsFragment to AllDishesFragment when user clicks on the back button.

```
return NavigationUI.navigateUp(mNavController, null)
```

7. In MainActivity create a function to hide the BottomNavigationView with animation.

```
fun hideBottomNavigationView() {
    mBinding.navView.clearAnimation() // in case there are
animations running, we get rid of them

mBinding.navView.animate().translationY(mBinding.navView.height.t
oFloat()).duration = 300
}
```

8. In MainActivity create a function to show the BottomNavigationView with Animation.

```
fun showBottomNavigationView() {
    mBinding.navView.clearAnimation()
    mBinding.navView.animate().translationY(0f).duration = 300
}
```

9. In AllDishesFragment call the hideBottomNavigationView function when user wants to navigate to the DishDetailsFragment.

```
if(requireActivity() is MainActivity){
    (activity as MainActivity?)!!.hideBottomNavigationView()
}
```

10. Override the onResume method and call the function to show the BottomNavigationView when user is on the AllDishesFragment.

```
override fun onResume() {
    super.onResume()
    if(requireActivity() is MainActivity){
        (activity as MainActivity?)!!.showBottomNavigationView()
    }
}
```

Project # 27 **27_ElinaResearchProject**

In this project we will pass the required dish details to the DishDetailsFragment using **Parcelable** and **SafeArgs**.

1. Go to gradle(Module: app) and apply the parcelize plugin.

```
id 'kotlin-parcelize'
```

2. Go to the **model**-> **entities** -> **FavDish** and Annotate a class with @Parcelize and implement Parcelable.

```
@Parcelize

@Entity(tableName = "fav_dishes_table")
data class FavDish( ...

) : Parcelable
```

Parcelable is similar to Serializable.
It makes an object into a String that can easily pass from one screen to another.

3. Go to the **res**-> **navigation**-> **mobile_navigation.xml** and add the required arguments to the dish details fragment and rebuild the project to generate the required class.

```
<fragment … >

    <argument
        android:name = "dishDetails"
        app:argType = "ca.you.package.model.entities.FavDish"/>

</fragment>
```

4. Go to the **view**-> **fragments**-> **AllDishesFragment** and add the required param as below to the function dishDetails.

```kotlin
fun dishDetails(favDish: FavDish){ …}
```

5. Go to the **view**-> **adapters**-> **FavDishAdapter** and pass "dish" parameter to dishDetails.

```kotlin
fragment.dishDetails(dish)
```

6. Go to the **view**-> **fragments**-> **AllDishesFragment** and add the required argument to the action in the dishDetail function.

```kotlin
findNavController()
    .navigate(
        AllDishesFragmentDirections.actionAllDishesToDishDetails(
            favDish // the required argument
        )
    )
```

7. Go to the **view**-> **fragments**-> **DishDetailsFragment** and override the onViewCreated method as below.

```kotlin
override fun onViewCreated(view: View, savedInstanceState: 
Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    ...
}
```

8. In the overridden function get the required arguments that we have passed through action and print a few details in the Log for now.

```kotlin
val args: DishDetailsFragmentArgs by navArgs()
Log.i("Dish Title", args.dishDetails.title)
Log.i("Dish Type", args.dishDetails.type)
```

Project # 28 **28_ElinaResearchProject**

In this project we will design the Dish Details screen layout and populate the data in the UI.

1. Go to the **res**-> **layout**-> **fragment_dish_details.xml** and design dish details layout however you want, which should include:

- Dish Title

- Type

- Category

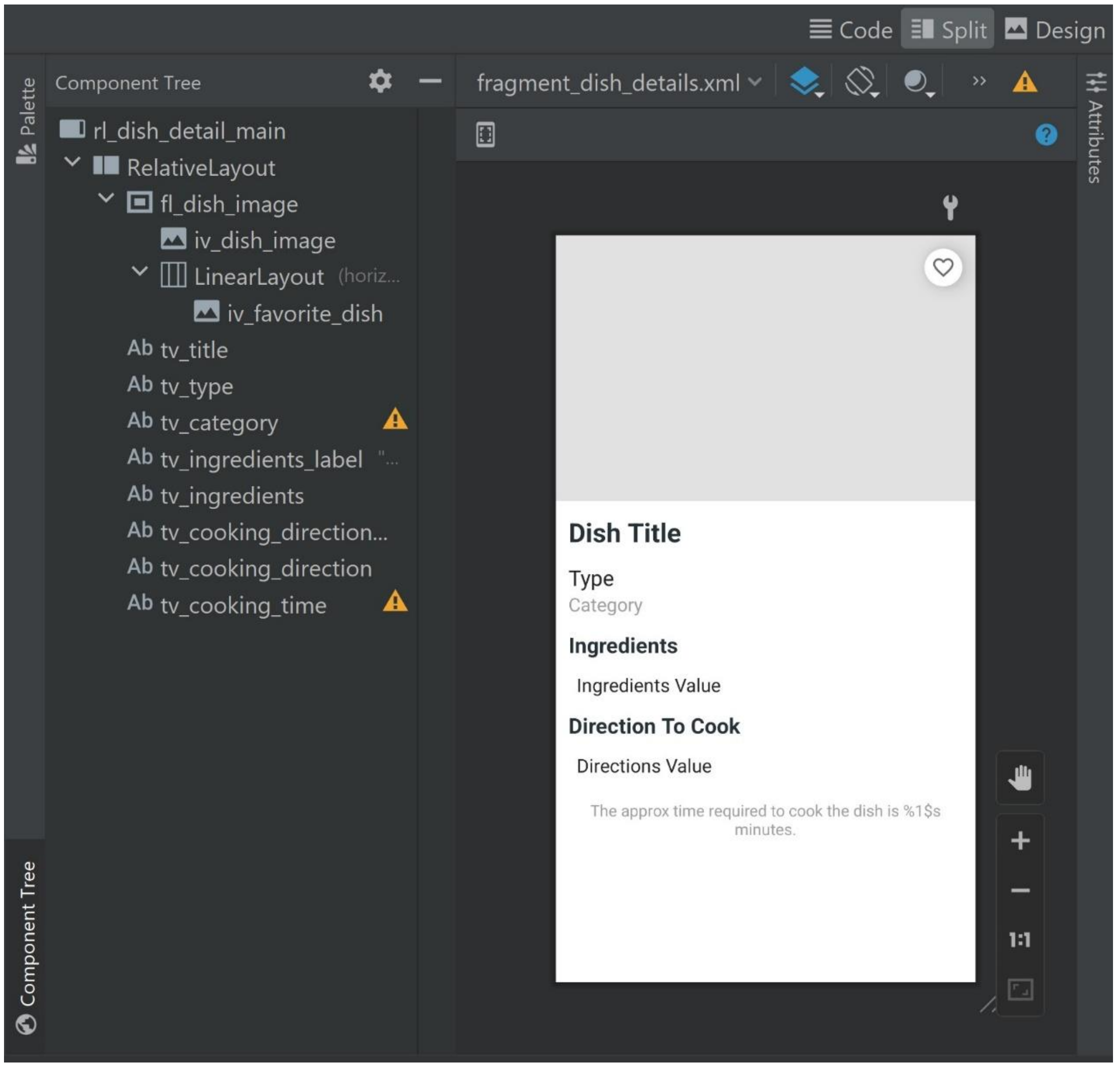
- Ingredients

- Cooking Time

- Direction to Cook

In this project we used ScrollView -> RelativeLayout -> FrameLayout (which includes ImageView , LinearLayout (which includes ImageView)) and eight TextViews for dish title, type, category, “Ingredients” as a title, ingredients value, “Direction to Cook” as a title, directions value and cooking time. See **fragment_dish_details.xml**.

This is the component tree and the UI we made.



rl_dish_detail_main
- RelativeLayout
  - fl_dish_image
    - iv_dish_image
    - LinearLayout (horiz...
      - iv_favorite_dish
  - Ab tv_title
  - Ab tv_type
  - Ab tv_category
  - Ab tv_ingredients_label "...
  - Ab tv_ingredients
  - Ab tv_cooking_direction...
  - Ab tv_cooking_direction
  - Ab tv_cooking_time

Dish Title

Type

Category

Ingredients

Ingredients Value

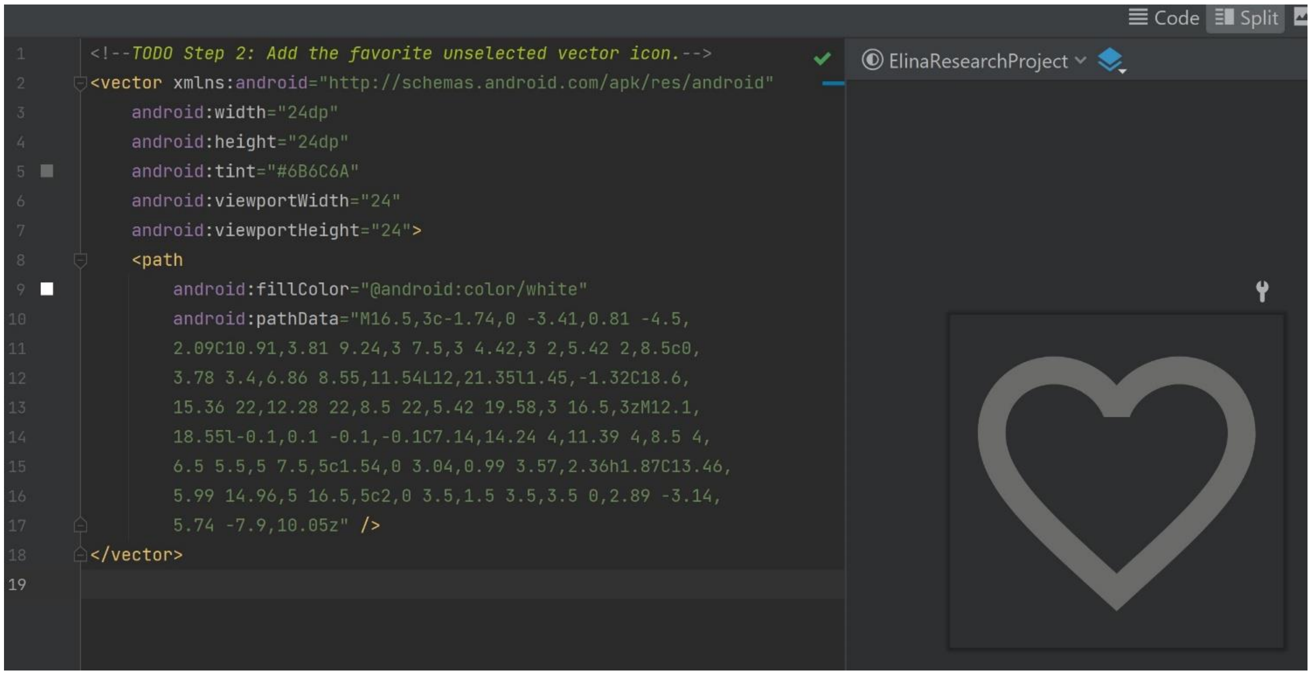
Direction To Cook

Directions Value

The approx time required to cook the dish is %1$s minutes.

2. Create an icon in drawable for unselected favorites. Name it as **ic_favorite_unselected.xml**.

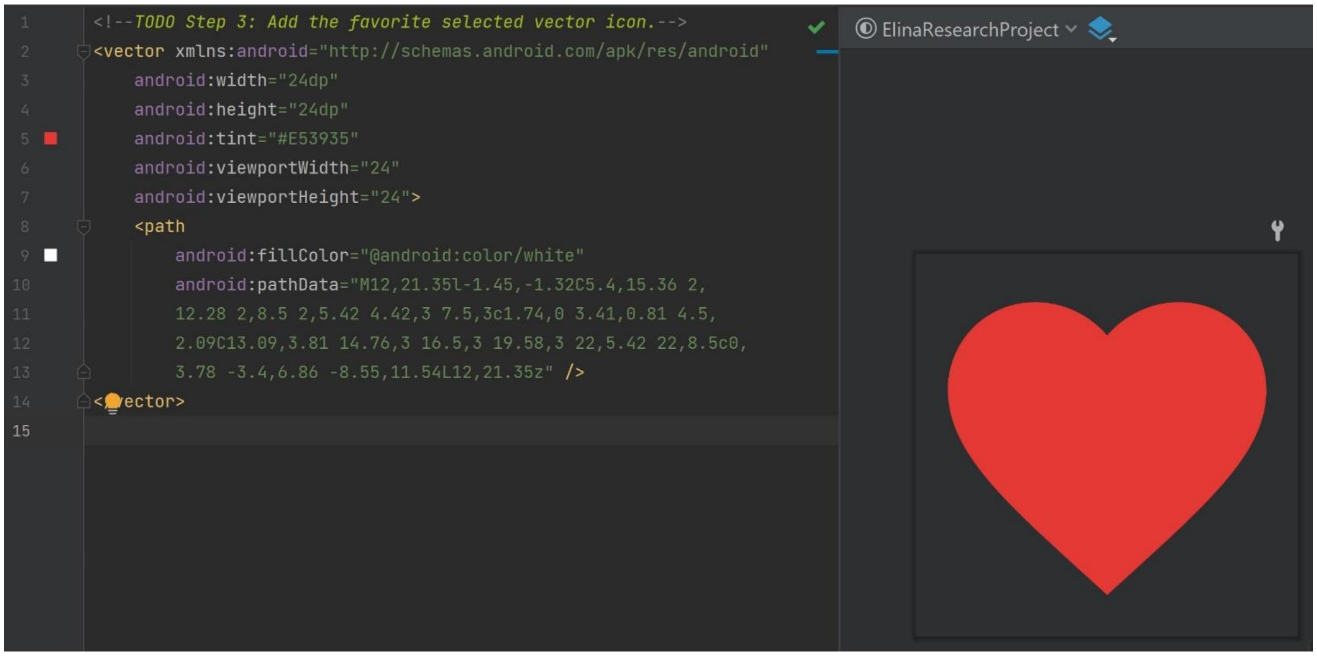
This is the one we are using for this project.



```xml
<!--TODO Step 2: Add the favorite unselected vector icon.-->
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    android:width="24dp"
    android:height="24dp"
    android:tint="#6B6C6A"
    android:viewportWidth="24"
    android:viewportHeight="24">
    <path
        android:fillColor="@android:color/white"
        android:pathData="M16.5,3c-1.74,0 -3.41,0.81 -4.5,
        2.09C10.91,3.81 9.24,3 7.5,3 4.42,3 2,5.42 2,8.5c0,
        3.78 3.4,6.86 8.55,11.54L12,21.35l1.45,-1.32C18.6,
        15.36 22,12.28 22,8.5 22,5.42 19.58,3 16.5,3zM12.1,
        18.55l-0.1,0.1 -0.1,-0.1C7.14,14.24 4,11.39 4,8.5 4,
        6.5 5.5,5 7.5,5c1.54,0 3.04,0.99 3.57,2.36h1.87C13.46,
        5.99 14.96,5 16.5,5c2,0 3.5,1.5 3.5,3.5 0,2.89 -3.14,
        5.74 -7.9,10.05z" />
</vector>
```

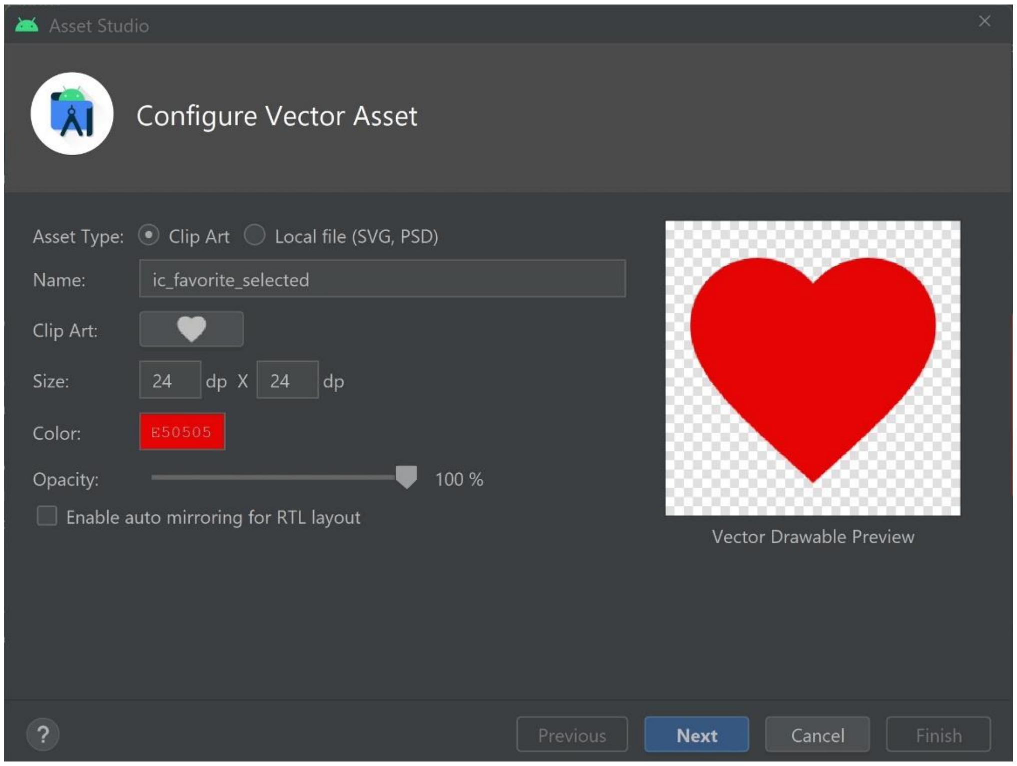
3. Create an icon in drawable for selected favorites. Name it as **ic_favorite_selected.xml**.

This is the one we are using for this project.



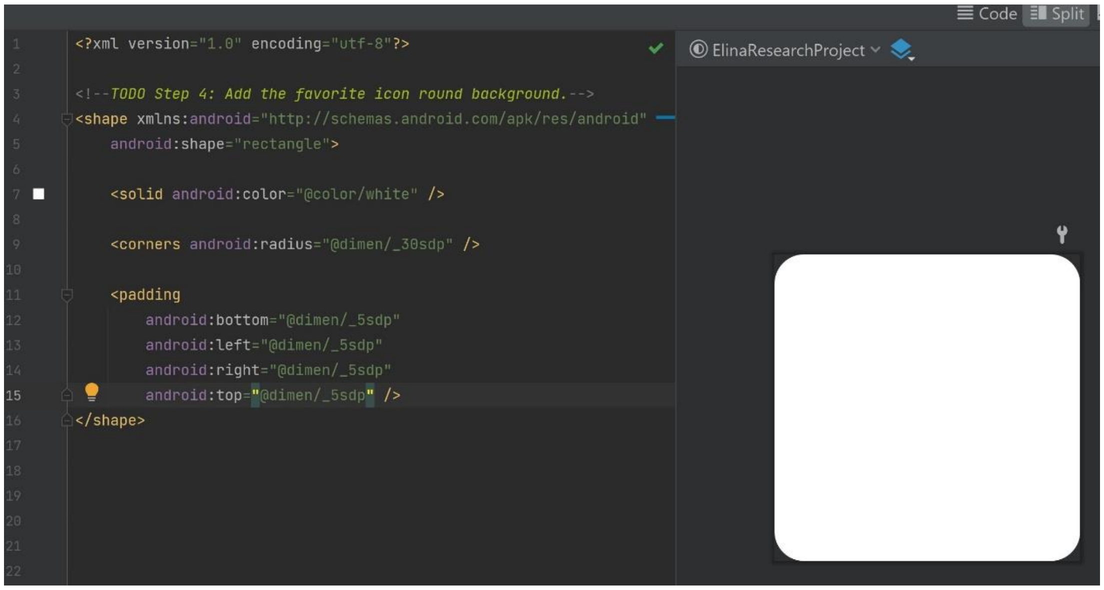
```xml
<!--TODO Step 3: Add the favorite selected vector icon.-->
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    android:width="24dp"
    android:height="24dp"
    android:tint="#E53935"
    android:viewportWidth="24"
    android:viewportHeight="24">
    <path
        android:fillColor="@android:color/white"
        android:pathData="M12,21.35l-1.45,-1.32C5.4,15.36 2,
        12.28 2,8.5 2,5.42 4.42,3 7.5,3c1.74,0 3.41,0.81 4.5,
        2.09C13.09,3.81 14.76,3 16.5,3 19.58,3 22,5.42 22,8.5c0,
        3.78 -3.4,6.86 -8.55,11.54L12,21.35z" />
</vector>
```

As we already know, we can create a vector by right clicking on **drawable** then selecting **New** and **Vector Asset** and create a vector icon as below:



Asset Type: [✓ Clip Art] Local file (SVG, PSD)
Name: ic_favorite_selected
Clip Art:
Size: 24 dp X 24 dp
Color: E50505
Opacity: 100 %
Enable auto mirroring for RTL layout

4. Create favorite icon round background in drawable as below:



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

<!--TODO Step 4: Add the favorite icon round background.-->
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle">

    <solid android:color="@color/white" />

    <corners android:radius="@dimen/_30sdp" />

    <padding
        android:bottom="@dimen/_5sdp"
        android:left="@dimen/_5sdp"
        android:right="@dimen/_5sdp"
        android:top="@dimen/_5sdp" />
</shape>
```

We are going to use this background for our favorite buttons selected and unselected.

5. Go to the **values** -> **colors.xml** and add two new colors

```
<color name="blue_grey_900">#263238</color>
<color name="grey_500">#9E9E9E</color>
```

6. Go to the **values** -> **strings.xml** and add a new string value.

```
<string name="lbl_estimate_cooking_time">
    The approx time required to cook the dish is %1$s minutes.
</string>
```

7. Go to the **view**-> **fragments** and in **DishDetailFragment** create a ViewBinding nullable variable.

```
private var mBinding: FragmentDishDetailsBinding? = null
```

8. In **DishDetailFragment** initialize the mBinding variable.

```
mBinding = FragmentDishDetailsBinding.inflate(inflater,
container, false)
return mBinding!!.root
```

9. In **DishDetailFragment** override the onDestroy function to make the mBinding null to avoid memory leaks. It is good practice to do it.

```
override fun onDestroy() {
    super.onDestroy()
    mBinding = null
}
```

10. In **DishDetailFragment** remove the log and populate the data to UI from the function onViewCreated.

```
args.let {

    try {
        Load the dish image in the ImageView.
        Glide.with(requireActivity())
            .load(it.dishDetails.image)
            .centerCrop()
            .into(mBinding!!.ivDishImage)
    } catch (e: IOException) {
        e.printStackTrace()
    }

    mBinding!!.tvTitle.text = it.dishDetails.title
    mBinding!!.tvType.text =
        it.dishDetails.type.capitalize(Locale.ROOT)

    mBinding!!.tvCategory.text = it.dishDetails.category
    mBinding!!.tvIngredients.text = it.dishDetails.ingredients
    mBinding!!.tvCookingDirection.text =
it.dishDetails.directionToCook
    mBinding!!.tvCookingTime.text =
        resources.getString(R.string.lbl_estimate_cooking_time,
it.dishDetails.cookingTime)
}
```