Module 3

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 3 Study Guide and Deliverables

Topics:	• Adapters and Adapter Views • MVM • Adapt to Different Screen Sizes
Readings:	• Online lectures • Notes and references cited in lectures • Chapters 12-14 from the "Head First" textbook (or corresponding chapters related to topics covered in this module in other textbooks)

Discussions:	• Discussion 3
	  ◦ Initial post due **Tuesday, July 26 at 6:00am ET**
	  ◦ Respond to threads posted by others due **Thursday, July 28 at 6:00am ET**

Labs:	• Lab 3 due **Tuesday, July 26 at 6:00am ET**
Assignments:	• Assignment 3 due **Tuesday, July 26 at 6:00am ET**
Assessments:	• Quiz 3 due **Tuesday, July 26 at 6:00am ET**
Live Classrooms:	• **Monday, July 25 from 6:00-8:00pm ET**

Learning Outcomes

By the end of this module, you will be able to:

- Compare list views and recycler views.
- Describe the relationship between adapters and views.
- Describe how fragments communicate with each other using Jetpack navigation.
- Explain ViewModel and LiveData.
- Describe how fragments communicate with each other through ViewModel and LiveData.
- Develop Android applications using list views or recycler views to display a list of items.
- Develop Android applications with multiple activities and fragments using MVVM architecture.

Introduction

In the last Module, we discussed the basics of user interface design, such as activities, fragments, views and viewgroups. In this module, we will discuss some advanced UI topics, including several adapter views, and fragment communication.

Topic 1: Adapters and Adapter Views

Adapter Views and Adapter

In the last module, we started developing the ProjectPortal Project. We developed the ProjectDetail and ProjectEdit screens using either just activities or activities + fragments. However, both screens only deal with a single project data. In most applications, we need to process a collection of data. For example, in the Projectportal example, we would like to list all project titles first before we view or edit the details of each project. In this topic, we will discuss the view widgets that work for a collection of data.

View Binding

Before we discuss how to display a collection of data, let us first introduce View Binding. In the previous code, we always needed to use findViewById() to get the reference of each view widget object by its id defined in the XML file. When there are quite a number of view widgets in an activity/fragment, we need to define a lot of variables and call findViewById() many times, as shown in the code below.

```
private lateinit var projTitle: EditText
private lateinit var projDesc: EditText
private lateinit var submit:Button
private lateinit var cancel:Button

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

   projTitle = view.findViewById(R.id.projTitleEdit)
   projDesc =  view.findViewById(R.id.projDescEdit)
   submit = view.findViewById<Button>(R.id.submit)
   cancel = view.findViewById\<Button>(R.id.cancel)
. . .
}
```

We can eliminate these lines of code by using view binding. Basically, you need to:

1. Enable the view binding in the module-level build.gradle file.

```
android {
   ...
   buildFeatures {
```

```
        viewBinding true
    }
}
```

2. Once the view binding is enabled, a view binding class will be automatically generated that is mapped to each layout XML file. For example, a view binding class `FragmentEditBinding` is generated that is mapped to fragment_edit.xml. Then we need to call the static inflate() method included in the generated binding class to create a view binding object. This is usually done in the onCreate() method or onCreateView() method.
3. After that we can use the binding object and the view object ID to get the references to all view widget objects , including the root view object defined based on the layout XML file. For example, binding.root is the root view for this fragment. binding.projTitleEdit is the EditText object with the id as R.id.projTitleEdit.

```
class EditFragment : Fragment() {
    // use ViewBinding
    private var _binding: FragmentEditBinding? = null
    // This property is only valid between onCreateView and onDestroyView.
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Use ViewBinding
        // FragmentDetailBinding is a generated binding class mapped to fragment_detail
        _binding = FragmentEditBinding.inflate(inflater, container, false)
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view,savedInstanceState)
       binding.projTitleEdit.setText(Project.project.title)
       binding.projDescEdit.setText(Project.project.description)
. . .
}
```

ListViews

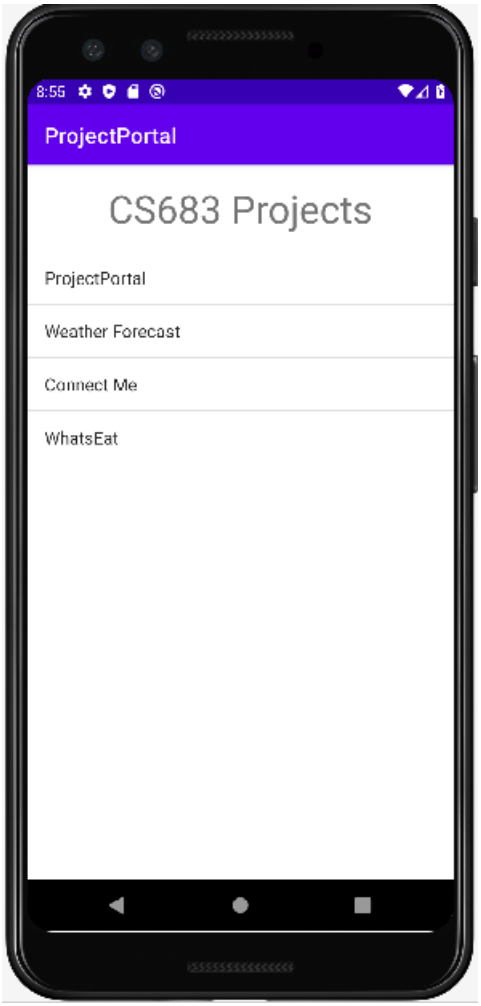
A simple approach to display a list of data is to use ListViews. A ListView can display a vertically-scrollable collection of views, where each view is used to display a single data item in the list of data, and is positioned immediately below the previous view in the list.

Let us add another fragment named ProjListFragment. The fragment_proj_list.xml is defined as below using a listview. If the collection of data is a static string array and pre-known, we can simply use the `android:entries` attribute as we used in the Spinner in the WidgetsExplore example in Module 2. The string array is defined in the `res/values/strings.xml`

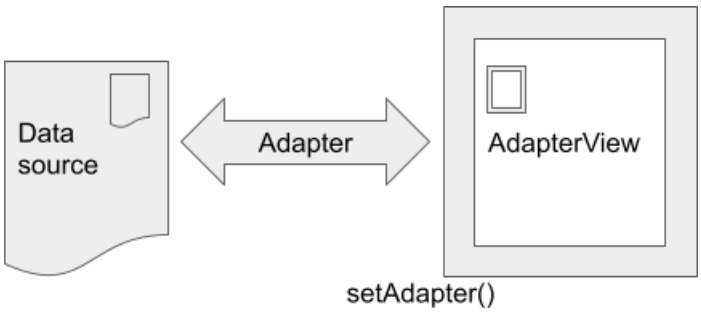
```
<androidx.constraintlayout.widget.ConstraintLayout
   android:id="@+id/main"
   android:layout_width="match_parent"
   android:layout_height="match_parent">

   <TextView
       android:id="@+id/projTitle"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_marginTop="16dp"
       android:text="CS683 Projects"
       android:textSize="16pt"
       app:layout_constraintEnd_toEndOf="parent"
       app:layout_constraintStart_toStartOf="parent"
       app:layout_constraintTop_toTopOf="parent" />

   <ListView
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:entries = "@array/project_list"
       android:layout_marginTop="16dp"
       app:layout_constraintEnd_toEndOf="parent"
       app:layout_constraintStart_toStartOf="parent"
       app:layout_constraintTop_toBottomOf="@+id/projTitle" />
   />

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

```
<resources>

   <string-array name="project_list">
       <item>ProjectPortal</item>
       <item>Weather Forecast</item>
       <item>Connect Me</item>
       <item>Whats Eat</item>
  </string-array>
</resources>
```

Run the app, you will see that a list of pre-known project titles defined in the string array is displayed on the screen.



However, in most cases, the collection of data is not pre-known, and they may also change dynamically. Then we have to bind data with the ListView dynamically. This is done through an Adapter. The ListView class is defined in the android.widget package. It is an indirect subclass of ViewGroup. It is also a kind of AdapterView. An Adapter acts as a bridge between the data source and the AdapterView. An AdapterView is a type of ViewGroup which has a collection of children views which are determined by an Adapter. Each child view is used to display a single data item in the collection of data. It is the Adapter that determines the children view and data source. Therefore, each AdapterView is associated with an Adapter. The following diagram shows the relationships between adapter and adapter view. We can either use predefined adapters or define your own adapters.



ArrayAdapter

An ArrayAdapter particularly works with arrays. Same as the array, the ArrayAdapter is also a parameterized class, in which you need to specify the array element type (ArrayAdapter<T>). It is defined in the app.widget package as the subclass of BaseAdapter.

ArrayAdapter connects the array to be displayed with the view that is used to display the array element. For example, we can create an array adapter to work with the string array by adding the following code in the onCreateView() in ProjectsListFragment.java. Here, projTitleList is a string list that holds all project titles. android.R.layout.**simple_list_item_1** is the built-in layout resource that tells the adapter to display each item in the list in a single text view. Then we set the adapter of the ListView to this array adapter.

```
val projListAdapter =  ArrayAdapter<String>(requireContext(),R.layout.simple_list_item_1,p
binding.projTitleList.adapter = projListAdapter
```

For simplicity, we define a static project list (using a companion object in Kotlin) in the Project class. The object "projects" is defined as a "val", so it cannot be reassigned. But we use a mutable list so that we can also add or delete projects from this list.

```
data class Project(val id: Int, var title: String, var description: String){
   companion object {
           val projects = mutableListOf(
           Project(0,"Weather Forecast", "Weather Forcast is an app ..."),
           Project(1, "Connect Me", "Connect Me is an app ... "),
           Project(2, "What to Eat", "What to Eat is an app ..."),
           Project(3, "Project Portal", "Project Portal is an app ..."))
   }
}
```

We can acquire all project titles from the project list.

```
private val projTitleList = mutableListOf<String>()
Project.projects.forEach { projTitleList.add(it.title)}
```

Test Yourself

When using a list view, if we know the list in advance, we can define the list in the strings.xml file and set the list directly using the attribute _____.

Test Yourself

What is the purpose of an adapter?

It acts as a bridge between the data source and the list view.

Test Yourself

An ArrayAdapter is a parameterized class. What is its parameter type for?

To specify the array element type.

Test Yourself

What is `android.R.layout.`*`simple_list_item_1`*?

It is a built-in layout resource that tells the adapter to display each item in the array in a single text view.

Test Yourself 3.5

Complete the following code snippet:

```
studentNameAdapter: ArrayAdapter<String>  =  _________ (requireC
studentNameListView._________ = studentNameAdapter;
```

ArrayAdapter<String>, android.R.layout.**simple_list_item_1, setAdapater**

Test Yourself

What will happen if we use:

```
val projListAdapter =
   ArrayAdapter<Project>(requireContext(), R.layout.simple_list_
```

It will display a string returned by the toString() method for each project in the list.

CardView

The list view is quite plain. To provide a better UI design, we can apply some material design principles. Card view and recycler view widgets are recommended to use as material widgets for such purposes.

A CardView is a FrameLayout with a round corner background and shadow. It is provided by the v7 support library and androidx. With a cardview, you can make your plain UI more interesting with elevation, colors, and animation. According to google, “Cards are a convenient means of displaying content composed of different elements. They’re also well-suited for showcasing elements whose size or supported actions vary, like photos with captions of variable length.”

To create a card view, you can directly use a <CardView> element in the layout file. For example, we would like to display each project and an id number in a cardview. Let us create a new layout file named project_list_item.xml. Do this by right clicking the app/src/main/res/layout folder, and choosing New Layout resource file in the popup menu.

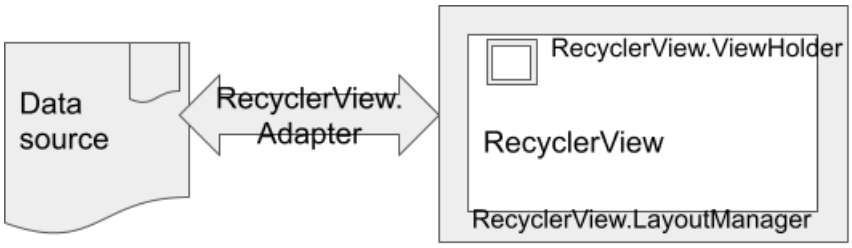
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.widget.CardView
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
```

```
    android:layout_height="wrap_content"
    android:layout_margin="8dp"
    android:id="@+id/projectCard">

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

        <TextView
            android:id="@+id/projIdView"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="@dimen/text_margin"
            android:textAppearance="?attr/textAppearanceListItem" />

        <TextView
            android:id="@+id/projTitleinCard"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_margin="@dimen/text_margin"
            android:textAppearance="?attr/textAppearanceListItem"
            android:padding="8dp"/>
    </LinearLayout>

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

RecyclerView

While the ListView is simple to use, it is not very efficient. When you have a large collection of data, a large collection of view objects will also be created, each view object for a single data item. However, only a small number of views can be displayed on the screen. Recycler view is a more advanced and flexible version of a list view that implements nested scrolling. It basically reuses a small number of views *efficiently* to provide the support for a large collection of views beyond the screen.

A RecyclerView is also a ViewGroup as ListView. However, unlike ListView, it is not a type of original AdapterView. It is introduced in the support library and now in the Androidx namespace. It directly inherits from ViewGroup. Nevertheless, it has similar behaviors and usage as those AdapterViews. It also needs to work with Adapter, which determines the children views and data source. The following diagram shows the relationships between RecyclerView classes. There are several inner classes of RecyclerView: RecyclerView.Adapter, RecyclerView.ViewHolder, and RecyclerView.LayoutManager.



To use a recycler view, you need to change both xml files and java files.

1. Add a RecyclerView element in the layout xml file (either for activity or fragment).. Here we use <RecycleView> instead of <ListView> in fragment_projects_list_recycler_view.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    tools:context=".ProjListFragment">

    <TextView
        android:id="@+id/title"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:text="CS683 Projects"
        android:textSize="16pt"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/projlist"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:layout_marginLeft="16dp"
        android:layout_marginRight="16dp"
        app:layoutManager="LinearLayoutManager"
        tools:context=".ProjListRecycleViewFragment"
        tools:listitem="@layout/fragment_proj_item"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/title" />
```

```
    />

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

2. Create a customized RecyclerView adapter class. Here we name it `MyProjListRecyclerViewAdapter`.This is more complicated than using list views, where you may use built-in adapters such as array adapters. It is described in more detail in the following sections.
3. Set the adapter of the RecycleView to your customized RecyclerView object. This is done in the Fragment class code here. You can also specify how to arrange views in the recyclerview using a layout manager. The recycler view is more flexible than a list view. You can choose linearlayout, gridlayout or staggered grid layout.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    binding.projlist?.apply{
        layoutManager = when {
            columnCount <= 1 -> LinearLayoutManager(context)
            else -> GridLayoutManager(context, columnCount)
        }
        adapter =     MyProjListRecyclerViewAdapter(Project.projects)
    }
}
```

RecyclerView.Adapter

RecyclerView.Adapter is an inner class of RecycleView. It is a static abstract class. (A static class means only one copy across all class instances.) We need to create a subclass and implement several methods in it. It is also a parameterized class with a RecyclerView.ViewHolder subclass VH as its parameter. VH (We may give a different name) is a static inner class defined in the Adapter class. It is a subclass of RecycleView.ViewHolder which is also a static abstract inner class of RecycleView. It describes an item view and metadata about its place within the RecyclerView. We need to override its constructor method.

```
abstract class Adapter<VH : RecyclerView.ViewHolder!>
```

kotlin.Any
↳ androidx.recyclerview.widget.RecyclerView.Adapter

As the main role of the adapter is to bridge the data source with Recyclerview, each adapter contains a data source which is a collection of data. Usually we set the data source through the constructor. However, it can also be set or updated through other methods. The following code shows that the `ProjectListRecyclerViewAdapter` is a subclass of `RecyclerView.Adapter` with the customized ViewHolder defined in the class as its parameter and contains a list of projects as its data source that is set through the constructor method.

```
class ProjectListRecyclerViewAdapter(
   private val projects: List<Project>
  ) : RecyclerView.Adapter<ProjectListRecyclerViewAdapter.ViewHolder>() {
 . .
}
```

We need to create an inner class of ViewHolder and override its constructor method.

```
inner class ViewHolder(binding: FragmentProjItemBinding)
   : RecyclerView.ViewHolder(binding.root) {
   val idView: TextView = binding.projIdView
   val contentView: TextView = binding.projTitleinCard
}
```

The following three abstract methods in the Adapter class need to be overridden besides the constructor method. The first two are related to its inner ViewHolder, and the third is to get the total number of data items in the data source.

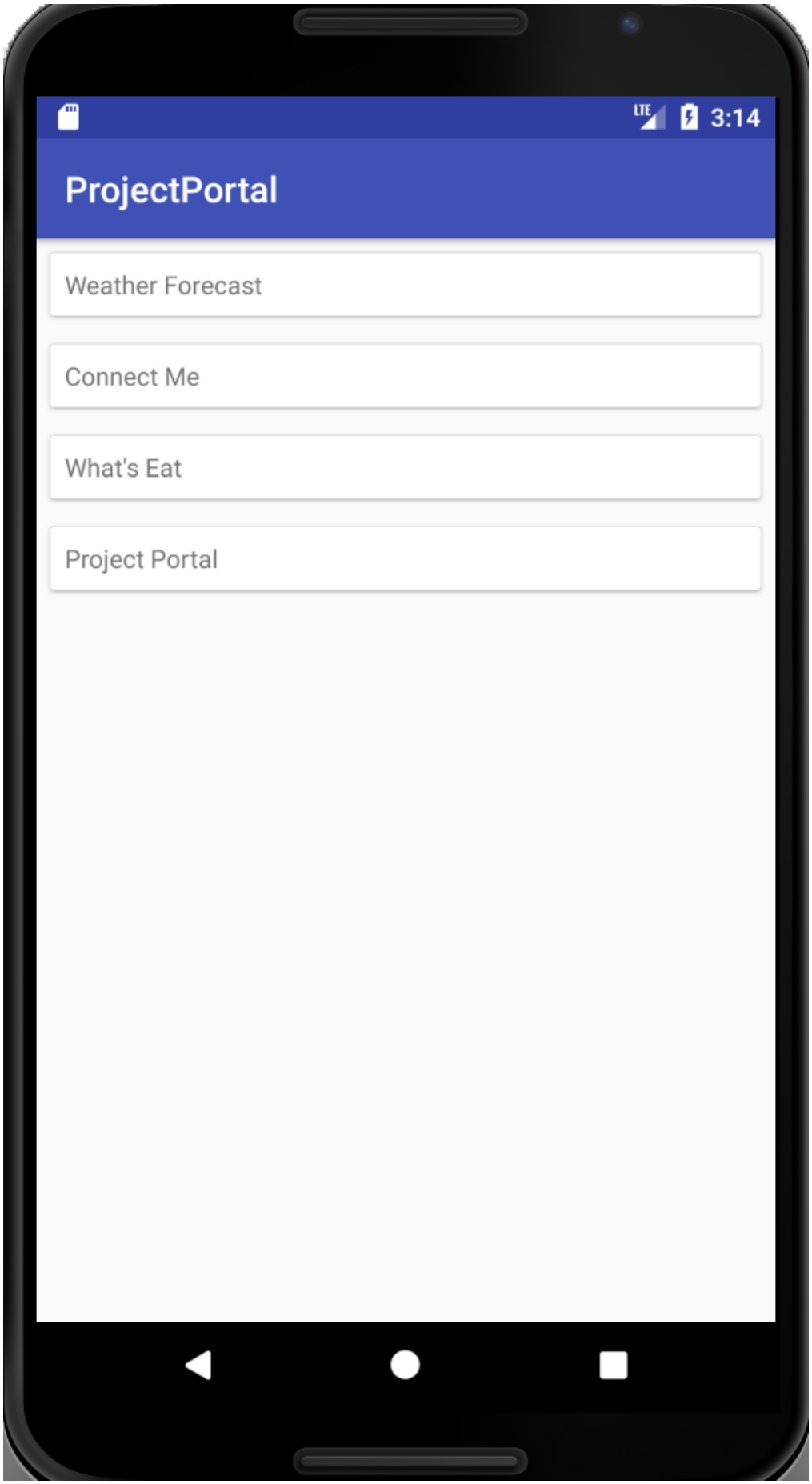
```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
// your code
}

override fun onBindViewHolder(holder: ViewHolder, position: Int) {
 // your code
}

override fun getItemCount(){
  // your code
}
```

We need to override the `onCreateViewHolder()` method to create a ViewHolder object. We also need to override the `onBindViewHolder()` method to bind a single data item to a single View defined by the ViewHolder.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
   return ViewHolder(
       FragmentProjectItemBinding.inflate(
           LayoutInflater.from(parent.context),
           parent,
           false
```

```
        )
    )
}

override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    holder.titleView.text = projects[position].title
  }
}

override fun getItemCount(): Int = projects.size
```

Here is what the screen looks using the recyclerview and the cardview.



Test Yourself

Why use cardviews?

More visual improvement, support different content length.

Test Yourself

What are the advantages of using RecyclerView?

It basically uses a small number of views efficiently to provide the support for a large collection of views beyond the screen. Provide more advanced and flexible nested scrolling.

Test Yourself

What layouts can be used to arrange views in the recyclerview using a layout manager?

linearlayout, gridlayout or staggered grid layout.

Test Yourself

To create a recyclerview adapter, we need to override three abstract methods. What are those?

onCreateViewHolder(), onBindViewHolder(), and getItemCount()

Test Yourself

What is the purpose of a view holder in the recyclerview adapter?

It describes an item view and metadata about its place within the RecyclerView. We need to override its constructor method.

Topic 2: MVVM

Passing Data Through JetPack Navigation Components

Now we want to navigate between the project list fragment to the project detail fragment, so that when the user clicks on one project in the list, the project detail screen will be displayed. In the previous module, we showed how to use the JetPack navigation component to navigate between the project detail and project edit screen. We can use the same way to navigate between the project list screen and project detail screen. However, we need to know which project is clicked and pass this information to the project detail screen, so that the correct project is displayed. Further we also need to pass this information to the project edit screen, so that the edit information will be saved to the right project as well. While we can also use a companion object to define a variable such as curProjectId together with Projects, this approach should be highly discouraged. Be aware that the companion object Projects is only used temporarily for the simplicity in this example. It will be replaced later.

One method is to define arguments for a destination and a navigation action in the navigation graph. In general, *we should only use this method to pass a very small amount of data between destinations*. For example, we can add the position as an argument for the project detail fragment and project edit fragment as shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:id="@+id/nav_graph"
   app:startDestination="@id/projListRecycleViewFragment">

   <fragment
       android:id="@+id/projListRecycleViewFragment"
       android:name="edu.bu.projectportal.ProjListRecycleViewFragment"
       android:label="ProjListRecycleViewFragment" >
       <action
           android:id="@+id/action_projListRecycleViewFragment_to_detailFragment"
           app:destination="@id/detailFragment" >
           <argument
               android:name="position"
               app:argType="integer" />
       </action>
   </fragment>

   <fragment
       android:id="@+id/detailFragment"
       android:name="edu.bu.projectportal.DetailFragment"
       android:label="fragment_detail"
       tools:layout="@layout/fragment_detail" >
       <argument
           android:name="position"
           app:argType="integer"
          android:defaultValue="0"/>
       <action
           android:id="@+id/action_detailFragment_to_editFragment"
           app:destination="@id/editFragment">
           <argument
               android:name="position"
               app:argType="integer" />
       </action>
   </fragment>
   <fragment
       android:id="@+id/editFragment"
       android:name="edu.bu.projectportal.EditFragment"
       android:label="fragment_edit"
       tools:layout="@layout/fragment_edit" >
       <argument
           android:name="position"
           app:argType="integer"
```

```
            android:defaultValue="0" />
        <action
            android:id="@+id/action_editFragment_pop"
            app:popUpTo="@id/editFragment"
            app:popUpToInclusive="true" >

        </action>

    </fragment>
</navigation>
```

We will also need to add code in the DetailFragment class and the EditFragment class to receive the position as the argument. For example, the following code is added in the DetailFragment class Here arguments is a field that can be accessed directly in the onViewCreated() method Fragment.

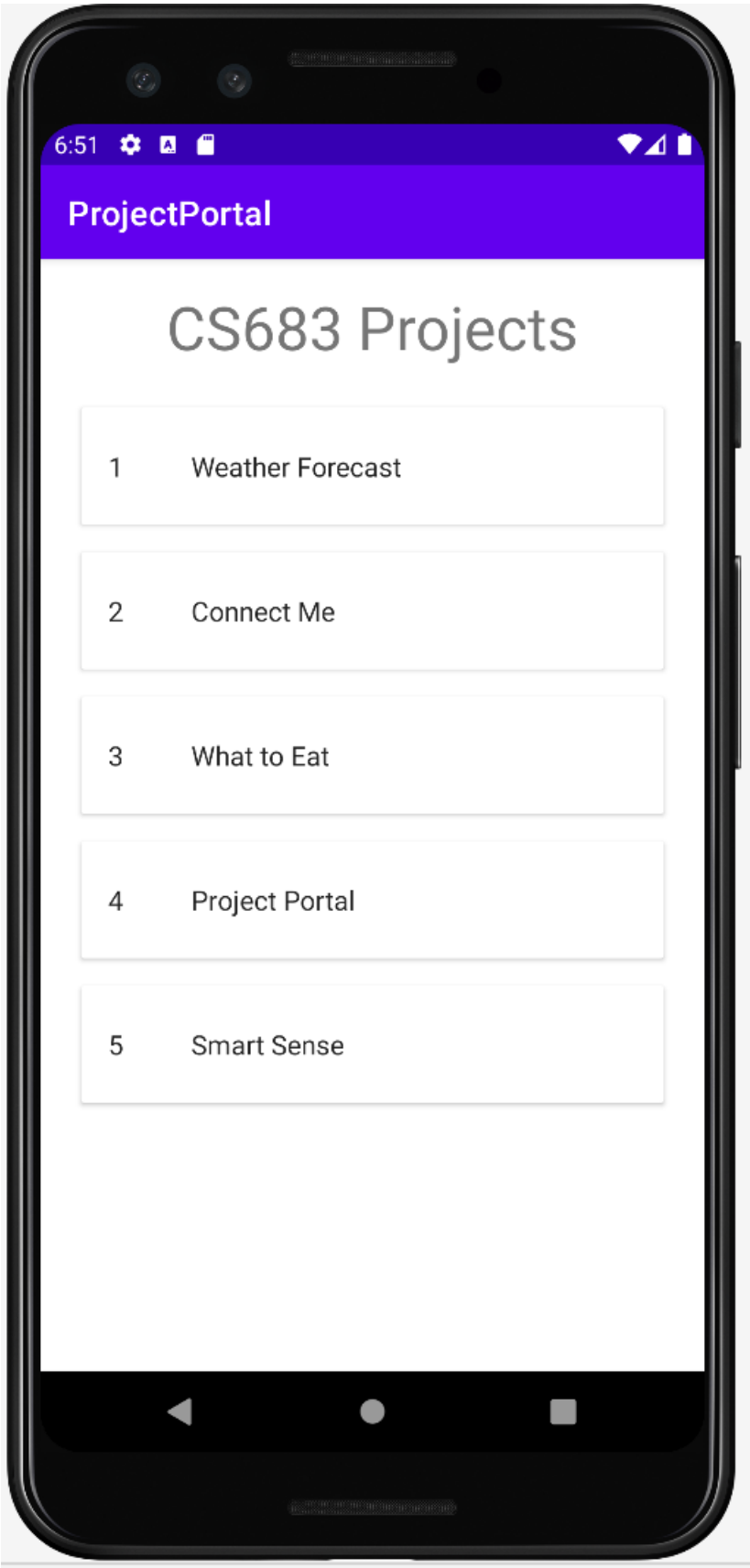
```
val position:Int = arguments?.getInt("position")?:0
binding.projTitleEdit.setText(Project.projects[position].title)
binding.projDescEdit.setText(Project.projects[position].description)
```

To respond to the user's onClick event on the project list and acquire which project is clicked in the list, we can add the following code in the onBindViewHolder() method in the adapter, where each data item is bound with a view. So we can set the onClickListener of each view to acquire the data associated with that view. Here we pass the position through the argument of the navigation action. We can also pass this position information from the detail fragment to the edit fragment. Here instead of simply using the action ID, we need to create a customized action with the information to be passed.

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    val project = projects[position]
    holder.idView.text = (project.id +1).toString()
    holder.contentView.text = project.title
    holder.cardView.setOnClickListener{
            val action = ProjListRecycleViewFragmentDirection
.actionProjListRecycleViewFragmentToDetailFragment(position)

            it.findNavController().navigate(action)
        }
}
```

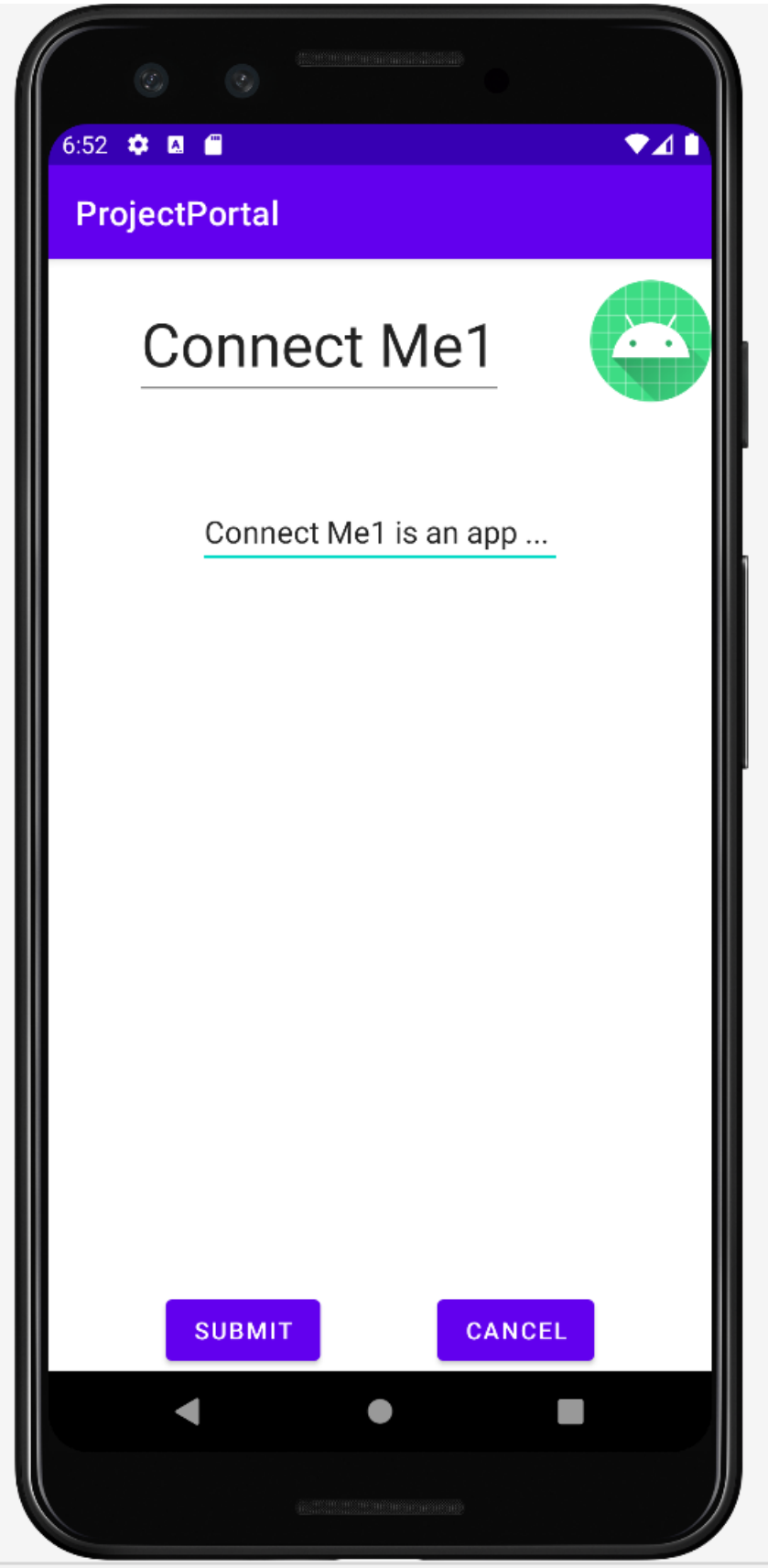
Here is the execution result.

Here is the execution result for the normal screen size:


6:51
ProjectPortal
CS683 Projects
1 Weather Forecast
2 Connect Me
3 What to Eat
4 Project Portal
5 Smart Sense


6:51
ProjectPortal


Connect Me
Connect Me is an app ...



Test Yourself

When would you use navigation destination arguments?

Pass a small amount of data between destinations.

Test Yourself

How would you set the argument?

Create a customized action using Directions.

ViewModel and LiveData

While we can use destination arguments to pass a small amount of data, it is not recommended to use this if you need to pass a large amount of data. In addition, the data pass can only happen when switching screens between fragments. Rather, a shared ViewModel is usually recommended to use for fragments to communicate with each other or with the host activity.

Problems of data in UI controllers (activities or fragments)

Currently for simplicity, we just use a simple companion object to hold all projects. In reality, we will need to access data from some permanent storage units such as databases, or files, or from the network. If we handle all these data accessing actions inside UI controllers, it will be affected by their life cycles. The Android framework manages the life cycles of activities and fragments. If the framework decides to destroy or re-create a UI controller in response to certain user actions or device events, any transient UI-related data you store in them is lost. In addition, the return of the asynchronous calls (such as reading data from a file or database) that the UI controllers made may cause potential memory leaks.

Separation of Duty Principle and ViewModel

Following *the separation of duty principle*, it is better to separate the presentation of data from storing and managing data. The **ViewModel** class is introduced in Android to store and manage UI-related data in a lifecycle conscious way. This enables UI controllers (activities and fragments) to just focus on the presentation of data, thus they are much cleaner and simpler. The ViewModel class allows data to survive configuration changes such as screen rotations. Any data contained in the ViewModel will stay until its lifecycle owner finishes.

LiveData

ViewModel objects often contain LiveData objects. LiveData is an lifecycle-aware observable data holder class. First, let us understand what an observable object is. The observer pattern is used here. An observable object can be observed by multiple observers. Whenever the observable object changes, all its observers are notified automatically. Usually In MVC architecture, model objects are observables, and view objects are observers. So when model data changes, views can be updated automatically. As an observable, LiveData provided by Android supports this auto notification. The developer needs to specify its observers, and how the observer will respond to the change when it receives the notification.

LiveData is also lifecycle-aware. This means, when the values of a livedata object change, not all its observers are notified. Only those observers who are in an **active lifecycle state (STARTED or RESUMED**) are notified. For example, if both DetailFragment and EditFragment observe a project livedata object, and only DetailFragment is currently in the active state (on the screen), when the project value changes, only DetailFragment will be notified, and so its view can be changed. This improves the performance and prevents data leakage.

A wrapper that can be used with any data, including objects that implement Collections, such as List. A LiveData object is usually stored within a ViewModel object and is accessed via a getter method, as demonstrated in the following example:

LiveData is defined as an abstract class with a parameter type T. It is a wrapper class that can be used with any data. We can set or get its value which is of type T. We can call the observe() method to add an observer into the observer list.

```
observe(owner: LifecycleOwner, observer: Observer<Any!>)
```

The `observer` is defined in androidx as:

```
interface Observer<T>
```

With a single abstract method which should be overridden to specify the actions responding to the change of the livedata objects.

```
onChanged(t: T!): unit
```

Usually, A ViewModel object contains one or multiple LiveData objects. However, ViewModel objects must never observe changes to LiveData objects. Rather, the corresponding UI controllers (activities or fragments) observe LiveData objects.

Passing Data between Fragments Using ViewModel

ViewModel is an ideal choice when you need to share data between multiple fragments or between fragments and their host activity. To create a ViewModel class, we need to subclass the abstract ViewModel class. To create a ViewModel object, we cannot simply use its constructor, instead we can use a ViewModel provider or create a customized factory method. To use ViewModelProvider, we need to first create a ViewModelProvider object by providing a ViewModelStoreOwner (activity, or fragment etc) and then use the get() method to get the corresponding ViewModel object.

```
ViewModelProvider(owner: ViewModelStoreOwner!)

<T : ViewModel!> get(modelClass: Class<T!>!)
```

> The ViewModel remains in memory until the ViewModelStoreOwner to which it's scoped goes away permanently. Once a ViewModel object is first instantiated, the subsequent calls to retrieve the ViewModel using the same scope always returns the same existing ViewModel along with the existing data until the ViewModelStoreOwnerStoreowner's lifecycle has permanently ended.

Therefore, by specifying the same ViewModelStoreOwner such as the hosting activity scope (using the `requireActivity()` method), we can share ViewModel objects between fragments within the same activities and their hosting activities. We can also use parent fragment scope (requireParentFragment()) to share between parent and child fragments. We can also scope a ViewModel to the lifecycle of a destinations' NavBackStackEntry (e.g. `NavController.getBackStackEntry(R.id.nav_graph)`) for all fragments in the backstack.

The ProjectPortal Example

Now let us use ViewModel and LiveData for the ProjectPortal Example. First, we need to define the ViewModel class. We will create two ViewModel classes, each containing a LiveData. One for the Project List that is used for the project list fragment, another is the Current Project that will be shared among all fragments. We use val for the contained LiveData object to prevent re-assigning it to different objects. We also only expose the immuable data to the public and keep the mutable data private to prevent the change of its value outside.

```
class ProjectListViewModel: ViewModel(){

    private val _projectList: MutableLiveData<MutableList<Project>> = MutableLiveData()
    val projectList:LiveData<MutableList<Project>>
    get() = _projectList

    init{
        _projectList.value = Project.projects
    }
}
```

```
class CurProjectViewModel: ViewModel(){
    private val _curProject: MutableLiveData<Project> = MutableLiveData()
    val curProject: LiveData<Project>
        get() = _curProject

    // initialize the current project to be
    // the first project in the list
    init {
        _curProject.value?.let {
            Project.projects[0]
        }
    }

    fun setCurProject(project: Project){
        _curProject.value = project
    }

    fun updateCurProject(title:String, desp:String){
        _curProject.value?.apply{
            this.title = title
            this.description = desp
        }
    }

}
```

In the fragments, we will need to instantiate them or get the references to them. We use this (the fragment itself) as the scope for ProjectListViewModel, but the activity as the scope of CurProjectViewModel. We will also need to observe the LiveData objects defined in them.

```
class ProjListRecycleViewFragment : Fragment() {

    ...

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

       val viewModel =     ViewModelProvider(requireActivity()).get(CurProjectViewModel::c
       val listViewModel =        ViewModelProvider(this).get(ProjectListViewModel::class.

. . .
}
```

```
class DetailFragment : Fragment(R.layout.fragment_detail) {

   override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
       val viewModel =          ViewModelProvider(requireActivity()).get(CurProjectViewMod

       viewModel.curProject.observe(viewLifecycleOwner, Observer {
           projTitle.text =  it?.title?:""
           projDesc.text = it?.description?:""
       })
```

In the EditFragment, the CurProjectViewModel is used similarly. In addition to observing its livedata object, we also need to change the data when the submit button is clicked, so that this change will be notified to project detail fragment and the project list fragment.

```
        class EditFragment : Fragment() {

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

       val viewModel =         ViewModelProvider(requireActivity()).get(CurProjectViewMode

       viewModel.curProject.observe(viewLifecycleOwner, Observer {
           binding.projTitleEdit.setText(it.title)
           binding.projDescEdit.setText(it.description)
       })
```

```
        binding.submit.setOnClickListener {
            viewModel.updateCurProject( binding.projTitleEdit.text.toString(),
                binding.projDescEdit.text.toString() )
           view.findNavController().navigate(R.id.action_editFragment_pop)
        }

. . .

    }
```

Since we use recyclerview in the project list fragment, and the data source is defined by its adapter, it is a little tricker to use ViewModel there. Instead of using the ViewModel directly in the adapter class, it is cleaner to have all ViewModel code in the fragment. We will use an interface and set a callback function to pass the viewmodel handling from fragment into the adapter.

In the adapter class, we define an interface OnProjectClickListener and pass the callback through the constructor.

```
        class MyProjListRecyclerViewAdapter(
        private val projects: List>Project>,
       private val onProjectClickListener: OnProjectClickListener)
    : RecyclerView.Adapter<MyProjListRecyclerViewAdapter.ViewHolder>() {

    interface OnProjectClickListener {
        fun onProjectClick(project:Project);
    }
```

In the, we will use this callback to set the `onClickListener` of the ViewHolder object.

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val project = projects[position]
        holder.idView.text = (project.id +1).toString()
        holder.contentView.text = project.title
        holder.cardView.setOnClickListener{
            onProjectClickListener.onProjectClick(project)
        }
    }
```

In the project list fragment, we observe the view model and pass this callback to the adapter.

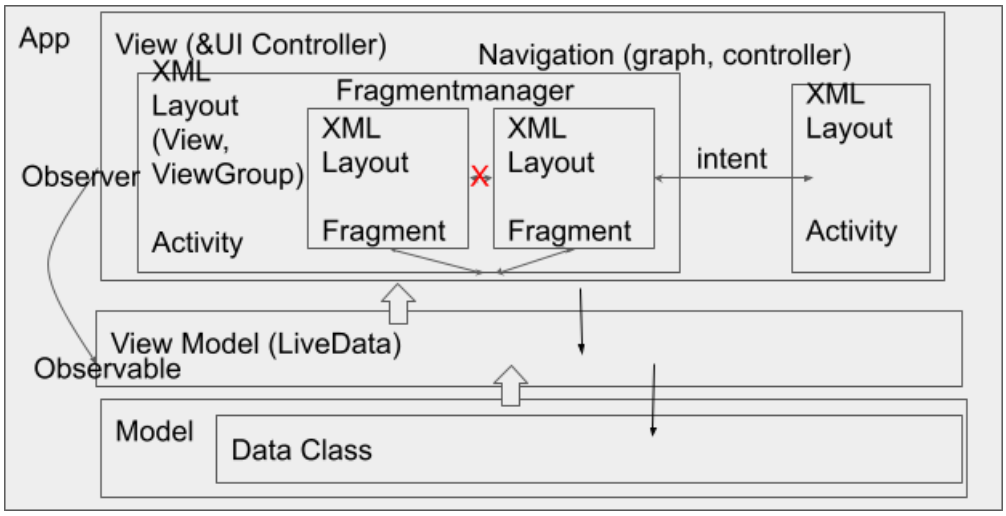
```
binding.projlist.apply {
    layoutManager = when {
        columnCount <= 1 -> LinearLayoutManager(context)
        else -> GridLayoutManager(context, columnCount)
    }

      val myAdapter = MyProjListRecyclerViewAdapter(
        listViewModel.projectList?.value ?: emptyList(),
        object : MyProjListRecyclerViewAdapter.OnProjectClickListener {
            override fun onProjectClick(project: Project) {
                viewModel.setCurProject(project)
                it.findNavController().navigate(
R.id.action_projListRecycleViewFragment_to_detailFragment)

            }
        })

    this.adapter = myAdapter

    listViewModel.projectList.observe(viewLifecycleOwner, Observer {
        myAdapter.notifyDataSetChanged()
    })

    viewModel.curProject.observe(viewLifecycleOwner, Observer {
        myAdapter.notifyDataSetChanged()
   })

}
```

The execution is the same as using the navigation and destination arguments.

MVC and MVVM

MVC (Model View Controller) is a very commonly used software architecture pattern. Some people argue that Android also uses MVC, where view objects are View and activities/fragments are controllers. Sometimes, activities or fragments are called UI controllers.

A variant of MVC is MVVM (Model View ViewModel), where ViewModel is a kind of controller, responsible for the data handling for View. With the ViewModel introduced in Android, MVVM is recommended in Android applications, where View in MVVM is basically activities or fragments (UI controllers). The following diagram shows the relationship between them.



In an Android Application, the View subsystem includes View/ViewGroup and UI controllers such as Activity or Fragments objects. The XML layout files specify the attributes of View objects associated with each screen. Activities communicate mainly through intents, while fragments within the same activities cannot communicate directly with each other, but through FragmentManager. The JetPack Navigation component simplifies the communication, making using the FragmentManager transparent to the developer. By adding a ViewModel subsystem, we can move all UI-related data handling into view models. With contained observable live data, observers in the View subsystem can update UI automatically whenever data changes. So the View subsystem doesn't directly interact with Model, rather, the ViewModel subsystem accesses the Model to get whatever data that UI needs. The data flows from the bottom to top, while the control flows from the top to bottom. That means, the models are not aware of view models, and view models are not aware of views. We shall only reference view models in the View subsystem, and handle UI data in the View Model subsystem, NOT vice-versa.

Topic 3: Adapt to Different Screen Sizes

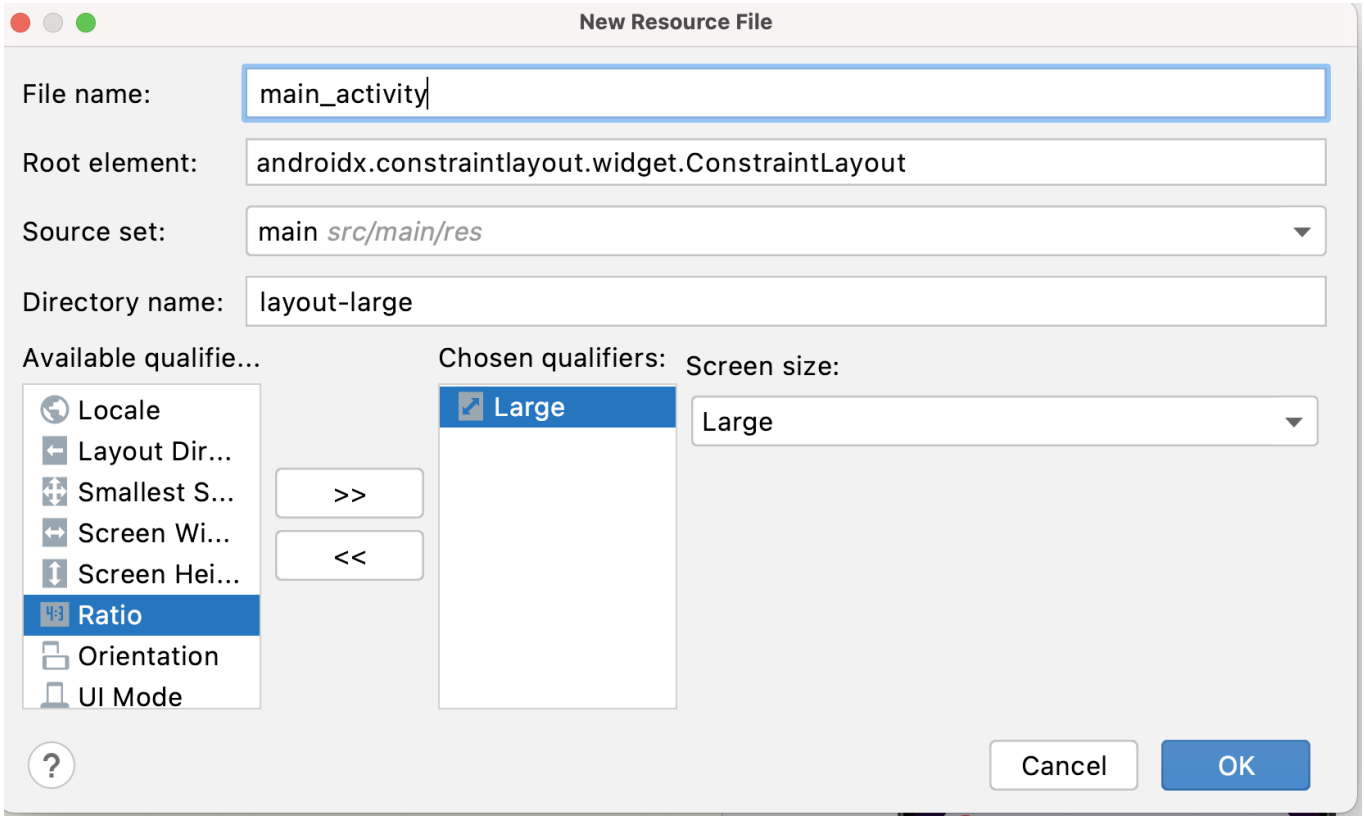
Adapt to Different Screen Sizes

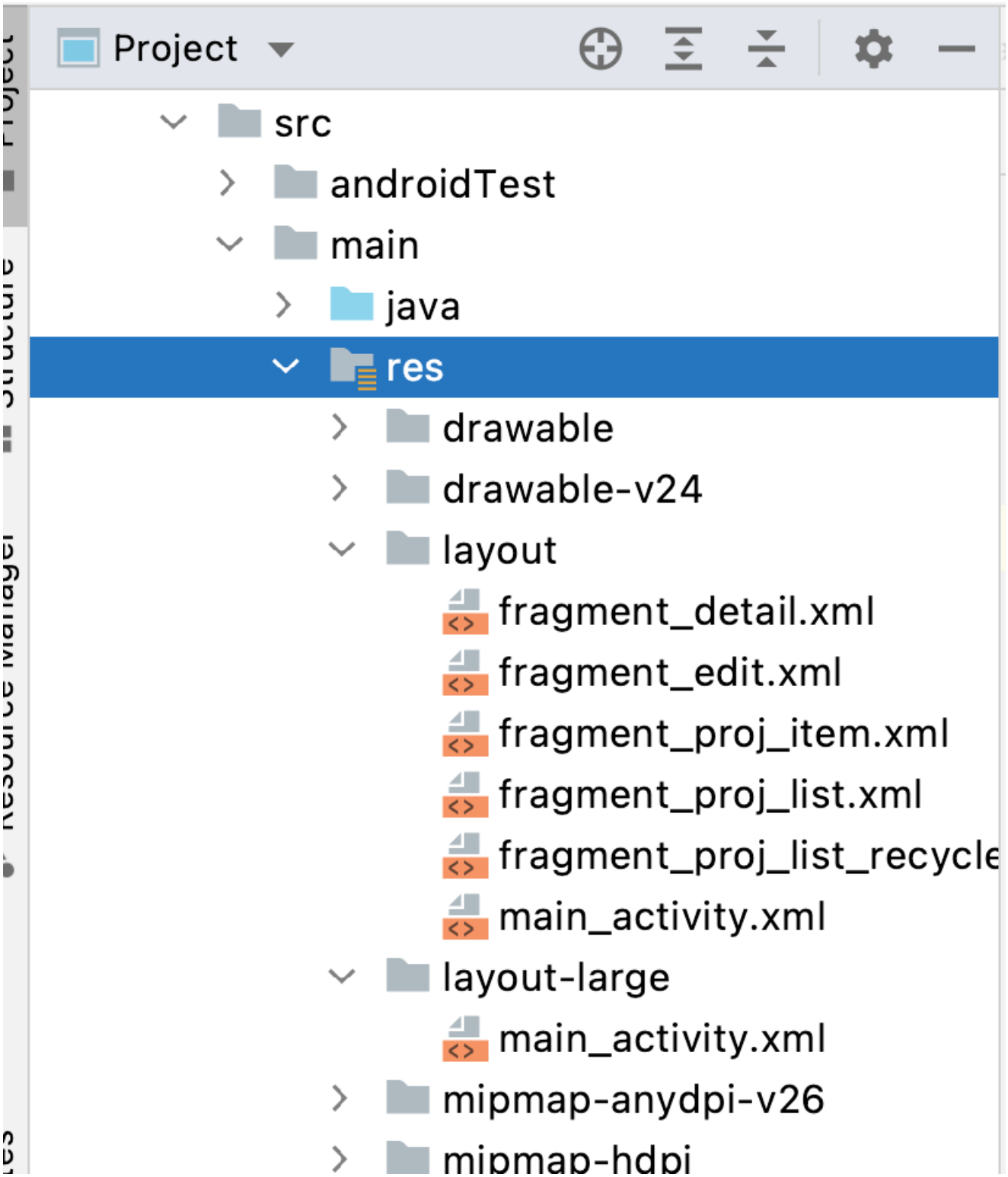
Currently It looks the same on a phone and a tablet, and It looks pretty bad on a big screen. It is better to display both fragments side by side on a large screen.

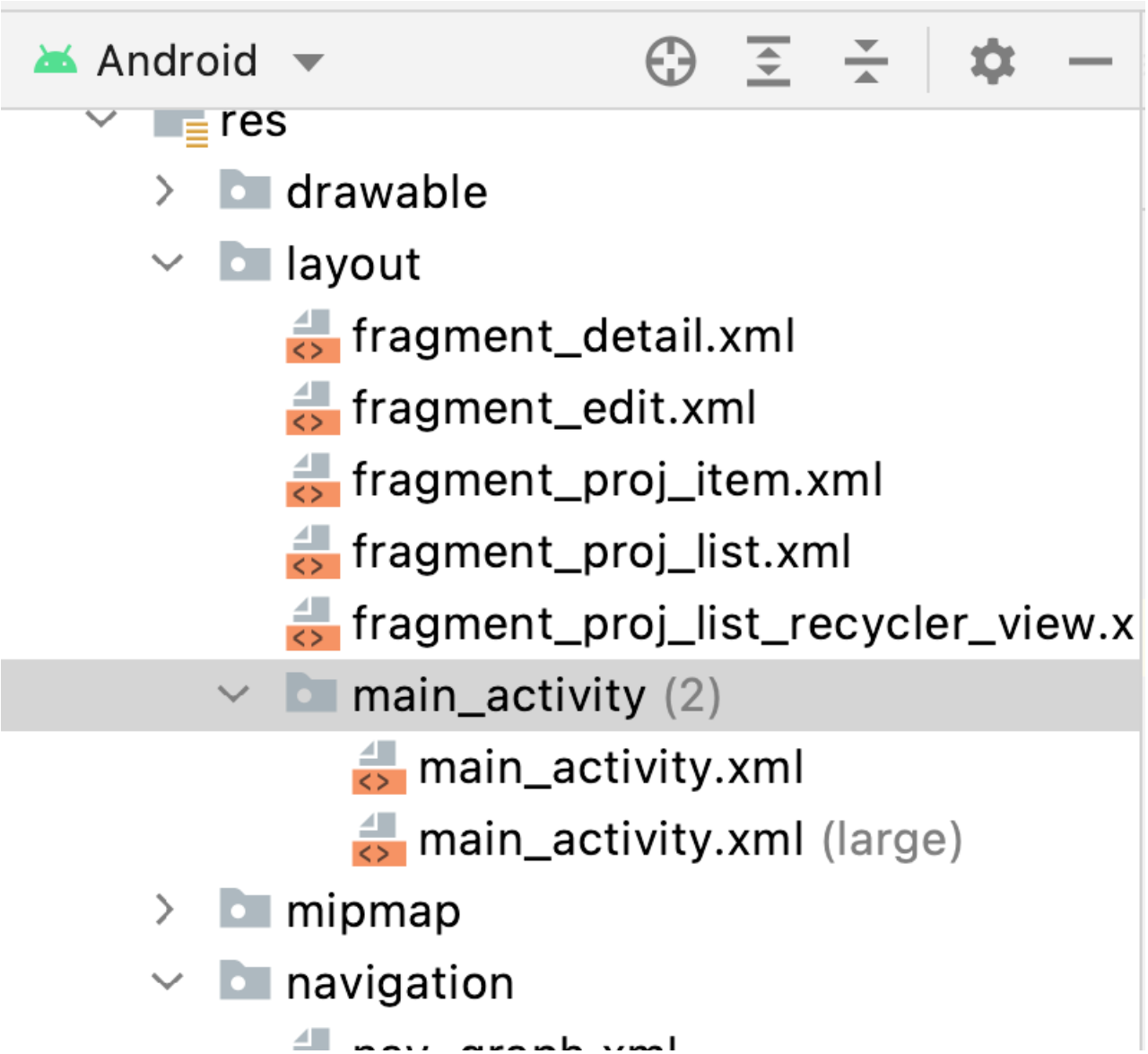
To adapt our application to different screen sizes, we can put screen specific resources in screen specific folders. The screen specific folder name can include screen size, density, orientation and aspect ratio with each part separated by hyphens. The available screen sizes are small, normal, large and xlarge. The available densities are:

- ldpi (low) ~(approximately)120dpi
- mdpi (medium) ~160dpi
- hdpi (high) ~240dpi
- xhdpi (extra-high) ~320dpi
- xxhdpi (extra-extra-high) ~480dpi
- xxxhdpi (extra-extra-extra-high) ~640dpi

For example, we can create another folder named layout-large to hold layout files for a large screen. In the project tool window, when we use "project view", we can see an overview of the file structure. Here, in the res/ folder, we create another subfolder "layout-large", and add a different layout file for the main activity. When creating subfolders for different screen sizes, make sure they cover all cases. For example, if we have a folder "layout-land" to specify layout files in landscape , we will also need a "layout-port" to specify layout files in portrait.


New Resource File
File name:
main_activity
Root element:
androidx.constraintlayout.widget.ConstraintLayout
Source set:
main src/main/res
Directory name:
layout-large
Available qualifie...
Locale
Layout Dir...
Smallest S...
Screen Wi...
Screen Hei...
Ratio
Orientation
UI Mode
>>
<<
Chosen qualifiers:
Large
Screen size:
Large
Cancel
OK


Project
src
androidTest
main
java
res
drawable
drawable-v24
layout
fragment_detail.xml
fragment_edit.xml
fragment_proj_item.xml
fragment_proj_list.xml
fragment_proj_list_recycle
main_activity.xml
layout-large
main_activity.xml
mipmap-anydpi-v26
mipmap-hdpi



We will have two different main_activity.xml. One is for the devices with normal screen size, and the other is for the devices with large screen size. The first one only contains one fragment, while the second contains two fragments side by side. Make sure the root level ID should be the same. Since on the large screen device, we will not have navigation between the list fragment and the detail fragment. We can break the original navigation graph into two. We will extract the navigation only between the detail and the edit fragment into nav_graph.xml and then use that as a nested navigation graph in the listnav_graph.xml for the list and detail fragment navigation for the normal screen devices. We use another list_graph.xml for just the list fragment and add an argument for the list fragment to indicate whether it is for large screen devices.

```
<?xml version="1.0" encoding="utf-8"?>
<navigation
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:id="@+id/nav_graph"
   app:startDestination="@+id/detailFragment">

   <fragment
       android:id="@+id/detailFragment"
```

```
        android:name="edu.bu.projectportal.DetailFragment"
        android:label="fragment_detail"
        tools:layout="@layout/fragment_detail" >
        <argument
            android:name="position"
            app:argType="integer"
            android:defaultValue="0"/>
        <action
            android="@+id/action_detailFragment_to_editFragment"
            app:destination="@id/editFragment">
        </action>
    </fragment>
    <fragment
        android:id="@+id/editFragment"
        android:name="edu.bu.projectportal.EditFragment"
        android:label="fragment_edit"
        tools:layout="@layout/fragment_edit" >
        <argument
            android:name="position"
            app:argType="integer"
            android:defaultValue="0"/>
        <action
            android:id="@+id/action_editFragment_pop"
            app:popUpTo="@id/editFragment"
            app:popUpToInclusive="true" >

        </action>
    </fragment>
</navigation>
```

```
<?xml version="1.0" encoding="utf-8"?>
<navigation
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/list_graph.xml"
    app:startDestination="@+id/projListRecycleViewFragment">

    <fragment
        android:id="@+id/projListRecycleViewFragment"
        android:name="edu.bu.projectportal.ProjListRecycleViewFragment"
        android:label="ProjListRecycleViewFragment" >
        <argument
            android:name="large-screen"
```

```
            app:argType="boolean"
            android:defaultValue="true"/>
        </fragment>

</navigation>
```

```
<?xml version="1.0" encoding="utf-8"?>
<navigation
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/listnav_graph"
    app:startDestination="@id/projListRecycleViewFragment">

    <fragment
        android:id="@+id/projListRecycleViewFragment"
        android:name="edu.bu.projectportal.ProjListRecycleViewFragment"
        android:label="ProjListRecycleViewFragment" >
        <argument
            android:name="large-screen"
            app:argType="boolean"
           android:defaultValue="false"/>
        <action
            android:id="@+id/action_projListRecycleViewFragment_to_nav_graph"
            app:destination="@id/nav_graph" />
    </fragment>
    <includeapp:graph="@navigation/nav_graph" />

</navigation>
```

Now, we can use listnav_graph in the main_activity.xml for the normal screen devices, and nav_graph in the main_activity.xml for the large screen devices. Here is the main_activity for normal screen devices.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
```

```
    app:defaultNavHost="true"
    app:navGraph="@navigation/listnav_graph"
    tools:context=".MainActivity"/>
```

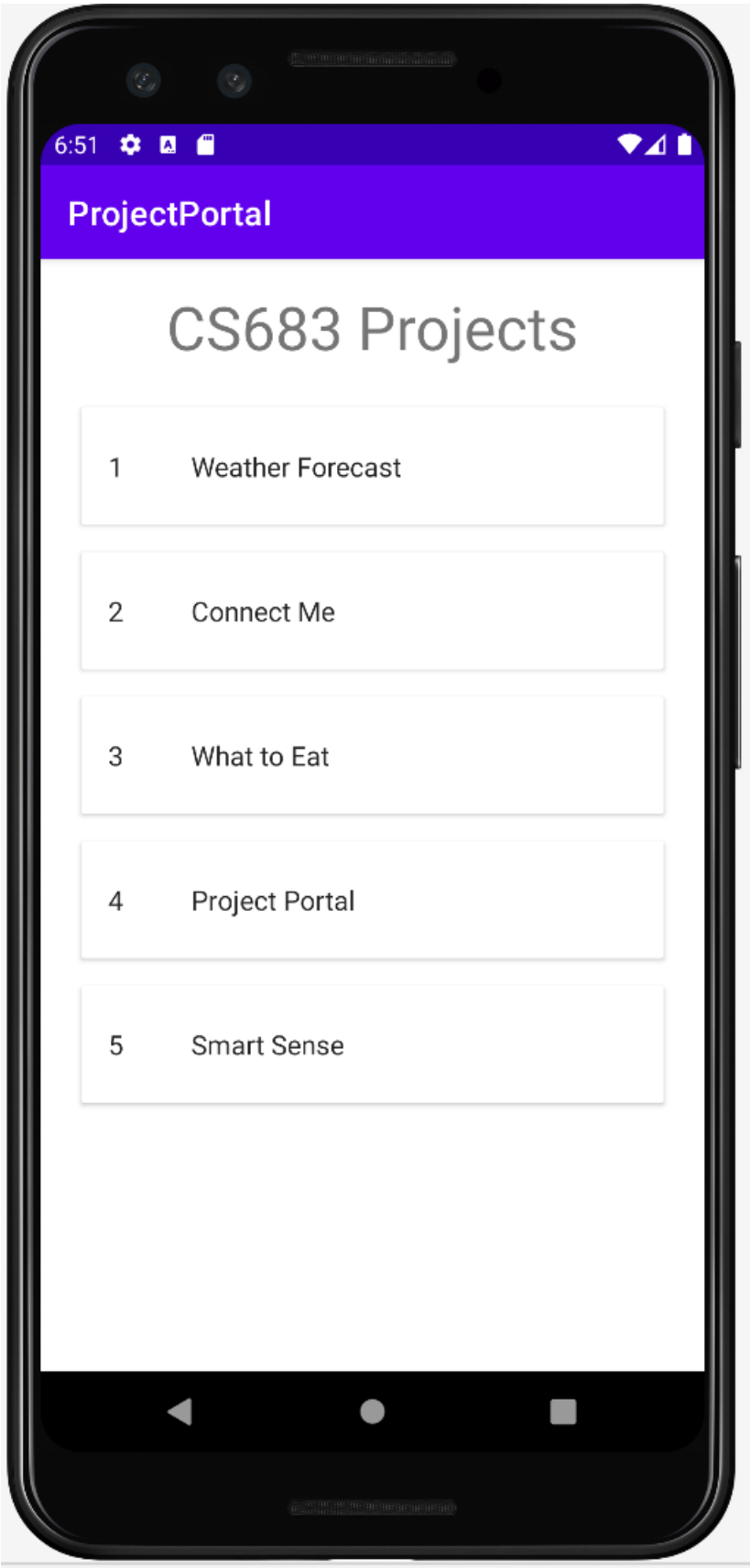
And this is for large screen devices.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <androidx.fragment.app.FragmentContainerView

android:name="androidx.navigation.fragment.NavHostFragment"
        android:id="@+id/list_fragment"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        app:defaultNavHost="true"
        app:navGraph="@navigation/list_graph"
        android:layout_weight="0.4" />


    <androidx.fragment.app.FragmentContainerView
       android:id="@+id/item_fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="0.6"
        app:defaultNavHost="true"
        app:navGraph="@navigation/nav_graph" />

</LinearLayout>
```

In the list fragment, we will modify the onProjectClick() call back when creating the adapter. We will only navigate to the detail fragment if this fragment id is R.id.main which is defined in the main_activity.xml for small screen devices.

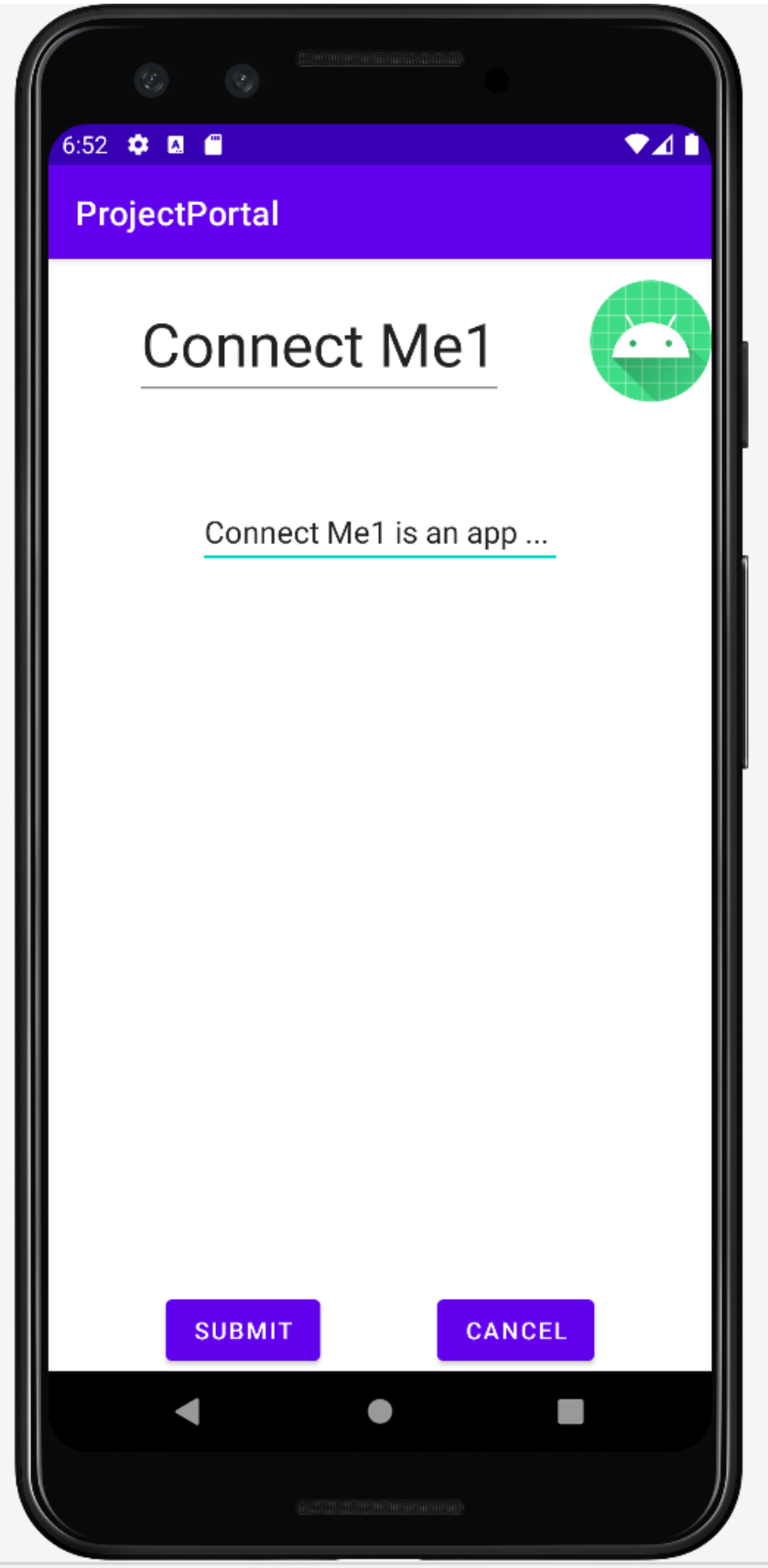
```
val myAdapter = MyProjListRecyclerViewAdapter(
  listViewModel.projectList?.value ?: emptyList(),
  object : MyProjListRecyclerViewAdapter.OnProjectClickListener {
      override fun onProjectClick(project: Project) {
          viewModel.setCurProject(project)
          if (!largerScreen) {
              view.findNavController()?.navigate(
                  R.id.action_projListRecycleViewFragment_to_nav_graph
              )
          }

      }
  })


private var largeScreen = false


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

   arguments.let {
       columnCount = it.getInt(ARG_COLUMN_COUNT)
   }
   arguments?.let {
       largeScreen = it.getBoolean(ARG_LARGE_SCREEN)
   }

}
```

Here is the execution result for the normal screen size, which is the same as before:

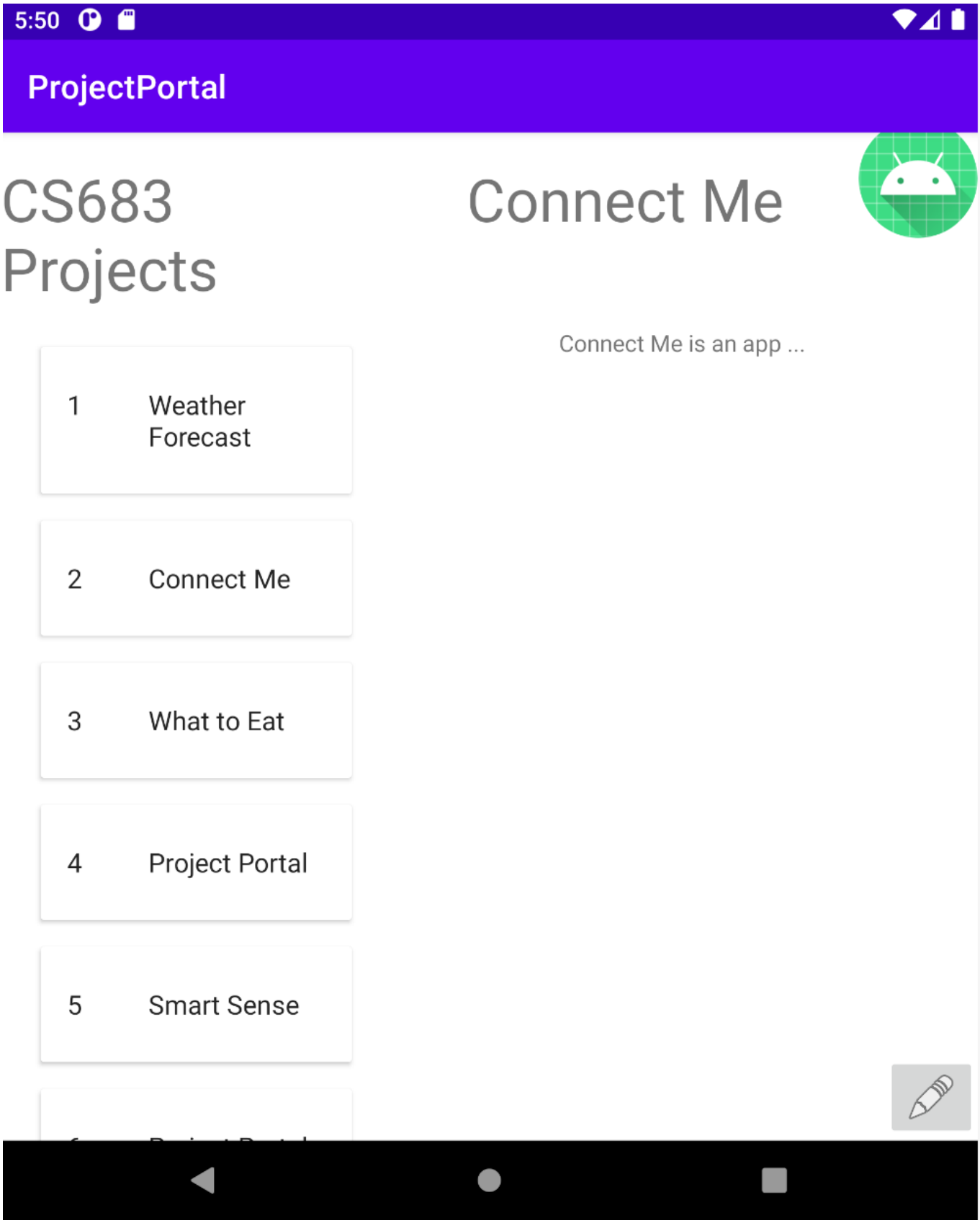

6:51
ProjectPortal
CS683 Projects
1 Weather Forecast
2 Connect Me
3 What to Eat
4 Project Portal
5 Smart Sense


6:51
ProjectPortal

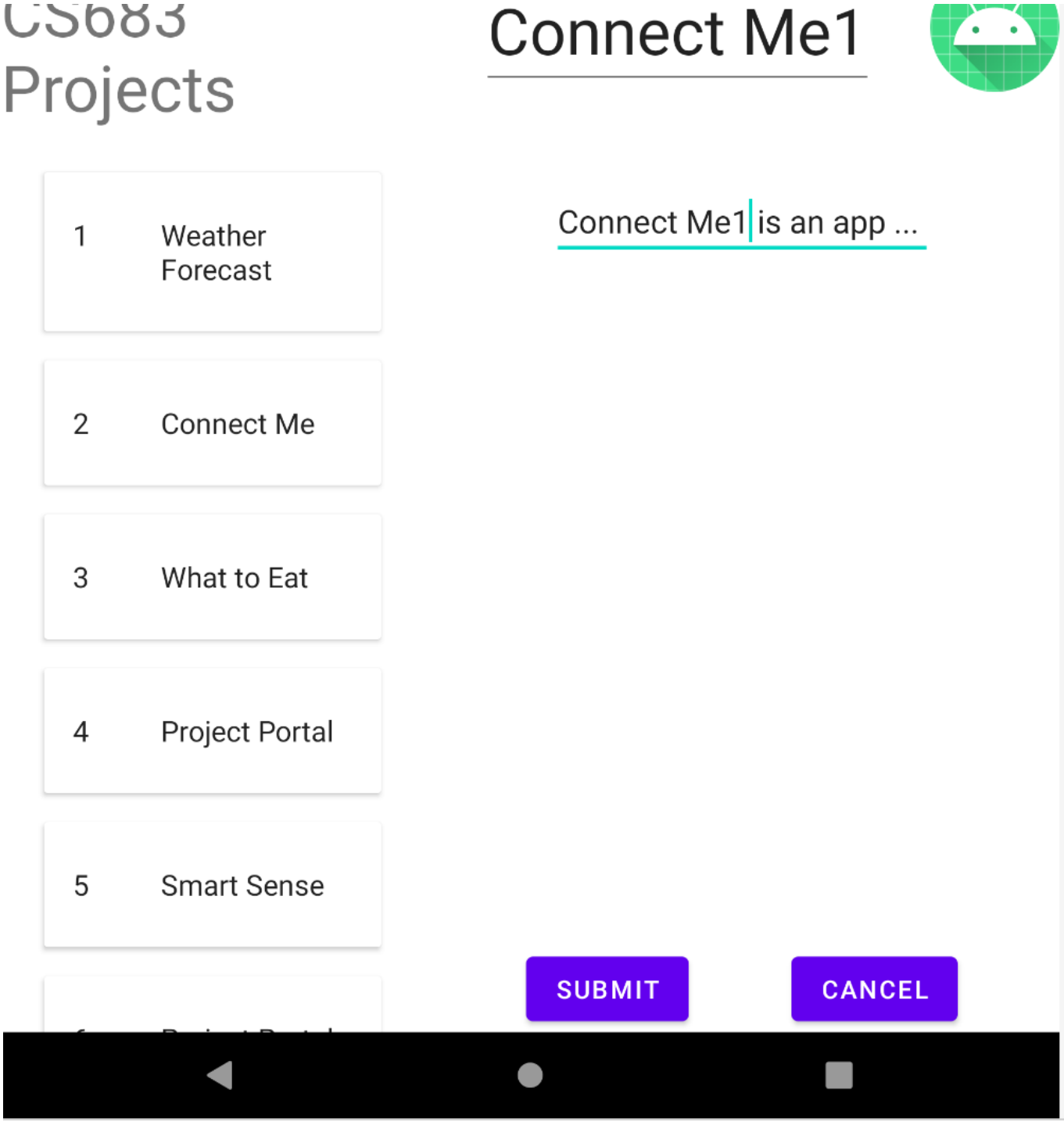

Connect Me
Connect Me is an app ...



Here is the execution result for the large screen size:


5:50
ProjectPortal
CS683 Projects
Connect Me
Connect Me is an app ...
1 Weather Forecast
2 Connect Me
3 What to Eat
4 Project Portal
5 Smart Sense


5:51
ProjectPortal



In addition to use alternative UI layouts according to the screen configuration in resource directory folders with different qualifier name, such as res/layout, res/layout-sw600dp, res/layout-land/, res/layout-w600dp, res/layout-large/, etc, we also need to make sure to use view dimensions that allow the layout to resize, such as using constraint layout, weight in Linear layout, match_parent, wrap_ content, etc. We need to avoid hard code size. If you need to deal with image file scaling, you may also need to provide bitmaps that can stretch with the views using nine patch files. For more details, please read Support different screen sizes.

To support multi-panel screens that adapt automatically for different screen sizes, SlidingPanelLayout is introduced in AndroidX. With the sliding pane, Initially, the list pane is shown filling the window. When the user taps an item the list pane is replaced by the detail pane for that item, which also fills the window. The SlidingPaneLayout can help manage the logic easier. The following XML layout file uses the sliding pane layout.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.slidingpanelayout.widget.SlidingPaneLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:id="@+id/main_activity"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context=".MainActivity">

   <androidx.fragment.app.FragmentContainerView
       android:layout_width="200dp"
       android:layout_height="match_parent"

android:name="androidx.navigation.fragment.NavHostFragment"
       android:id="@+id/list_fragment"
       app:defaultNavHost="true"
       app :navGraph="@navigation/list_graph"
       android:layout_gravity="start" />

   <androidx.fragment.app.FragmentContainerView
       android:layout_width="300dp"
       android:layout_height="match_parent"
       android:id="@+id/detailContainerId"

android:name="androidx.navigation.fragment.NavHostFragment"
       app:defaultNavHost="true"
       app:navGraph="@navigation/nav_graph"
       android:layout_weight="1"
       />

</androidx.slidingpanelayout.widget.SlidingPaneLayout>
```

With the slidingPaneLayout, there are always two panes on the same screen no matter the screen size. The two panes can overlap and one can slide away. So we don't need the navigation defined in listnav_graph at all.

```
 val myAdapter = MyProjListRecyclerViewAdapter(
               listViewModel.projectList?.value ?: emptyList(),
               object : MyProjListRecyclerViewAdapter.OnProjectClickListener {
                   override fun onProjectClick(project: Project) {
                       viewModel.setCurProject(project)
```

```
//                      if (!largeScreen) {
//                              view.findNavController()?.navigate(
//                                  R.id.action_projListRecycleViewFragment_to_nav_graph
//                              )
//                          }

                    }
                })
```

Test Yourself

To inform Android to use layout files for a different type of screen, we can create a separate layout folder in the res/ folder. Which folder name below is valid? (Check all that are true.)

(Note: Think of each answer as a separate question. The system will mark each one as correct or incorrect, whether or not the box was supposed to be checked. If you do not check the box and it was not supposed to be checked, it will also mark it correct.)

- layout-smartphone
- layout-large-land
- layout-normal

Conclusion

In this module, we have discussed several advanced topics related to UI design. First we discussed a simple way to display lists using listviews. Then we introduced cardviews and recycleviews which provide a more flexible way to display lists as well as visual improvement. The key to using these views is to define proper adapters. After that, we discussed several approaches to communicate between fragments. One is to use destination arguments in the navigation component. The other is to use ViewModel. We discussed why and how to use ViewModel and LiveData to handle UI-related data. Finally, we show how to make your app adapt to different screen sizes.