In this final project of the term, we'll build a complex app that will download and display a collection of curated images from photo hosting/sharing site FlickrLinks to an external site., via one of its many APIs. The images are displayed in a scrolling grid by default. The user can toggle the display to place the images on a world map (which can pan and zoom) based on the coordinates embedded in the photo. The user can click any image, from either the grid or the map, to view the Flickr web page for that image.

The FancyGallery app will allow us to experiment with several advanced features of mobile app development:

- Implement a bottom navigation bar
- Make requests to REST API services via HTTPS
- Deserialize JSON into Kotlin objects
- Load network images into a recycler view grid
- Refresh cache of loaded images on demand
- Display HTML content within a WebView
- Add a progress bar to indicate loading of a WebView
- Display map tiles within a MapView with custom settings
- Share data across two distinct fragments
- Place markers onto a map at specified geo-coordinates
- Customize the display of map markers
- Handle user interactions with map markers

Some portions of this app are described in Chapters 20 and 23 from BNRG, but the majority of the development process is completely outside the scope of the textbook.

PHASE 1: Bottom Navigation and Network Access

Getting Started

A project template with starter code is once again provided via Git. Please start Project 3 by selecting "Get from VCS" from the Android Studio welcome screen, or use File | New | Project from Version Control from the menu of any existing project. In either case, specify the following remote URL, and specify any convenient directory on your development system:

```
https://www.prof-oliva.com/cs5254/2024.Spring/git/P3-FancyGallery.git
```

This will create a new project by cloning a local copy of the remote repository. The project will contain the all the starter code needed for this assignment in a single `assignment` branch.

Beyond the basic files normally created in a new Android Studio project, the repository includes the following changes and additions specific to this project:

- **build.gradle.kts** (both)
    - Dependencies have been added to support developing and testing this project.
- **AndroidManifest.xml**
    - An element has been added to request Internet access (BNRG Listing 20.11)
- **P3Test.kt**
    - This instrumented test file will help ensure the app meets the assignment requirements. As usual, the test file won't compile until you've at least mostly completed all the features. You may temporarily comment-out this entire file (except the package declaration on the first line) while you develop the app.

Bottom Navigation

Choose two Vector clip art images and add them as Drawable resources to the project. One icon should suggest a grid, and the other icon should suggest a map. The resource names are up to you, but normally icons are named starting with `ic_...`

Create a new Menu type resource called `nav_menu.xml`. Add two menu Item components, setting the first title values to `Gallery` and the second to `Map`. Add an `android:icon` for each, to associate them with the icons noted above. Don't forget to use String resources rather than hard-coded values. The id values of the menu Item components must be `gallery_fragment` and `map_fragment`. Note that the id values may need to be set in the Code tab, rather than the Design tab, due to a bug in Android Studio.

Open the `activity_main.xml` file, and add a BottomNavigationView component (found in the Containers section of the palette) to the bottom of the screen. Set the id of this component to `bottom_nav`. This component must span the full width at the bottom of the the layout, and span the height of its own content. Add an attribute no this component named `app:menu` and set its value to the `nav_menu` menu resource. The default "Hello World!" TextView should fill the remainder of the screen.

You should now see the names and icons in the bottom navigation bar of the layout preview. The Gallery navigation should be in the primary color on the left, and the Map should be in gray on the right.

When you run the app now, you should be able to select either Gallery or Map, although the only difference will be which icon/text is in the primary color.

Create a new Navigation type resource called `nav_graph.xml` and just leave it as-is for now.

Create two new layout resource files, `fragment_gallery.xml` and `fragment_map.xml`, which will correspond to new fragments we'll create next. You may use the default ConstraintLayout or convert it to a LinearLayout. For now, just place a single TextView in each layout, each with no id value. These will just display the name of the associated layout. Please use the same string resources you created for the menu above. Double-check that the correct string is displayed in each layout preview.

Create two new Kotlin class files, `GalleryFragment` and `MapFragment`, and update each to become a subclass of Fragment. Add `_binding` and `binding` properties, along with minimal override implementations for both `onCreateView()` and `onDestroyView()`, exactly as we've done in previous projects.

In `activity_main.xml`, replace the TextView with a `NavHostFragment` component. Make sure this component fills the entire remainder of the screen above the bottom navigation bar. Refer to BNRG Listing 13.3 to link the NavHostFragment with the (currently empty) nav_graph. Change the default id of this component from `fragmentContainerView` to `fragment_container`.

Open `nav_graph.xml` and it should now automatically reference `activity_main (fragment_container)` as the host.

Add GalleryFragment as the home destination, change its id to `gallery_fragment`, which will detect that this id is already in use. This conflict is expected (and required) because the id here must match the id from the nav_menu for the navigation to work as expected. Please click Continue to proceed.

Repeat this process to add MapFragment as a destination to the graph, and likewise change its id to `map_fragment`. Don't link the two components in the nav_graph. Next add a `tools:layout` attribute to link each component to the associated layout. This is most readily done from the Code tab.

In `MainActivity`, replace the onCreate() implementation with the following code, which links the bottom navigation with the nav host controller. This code is very lightly from the ADWS Bottom Navigation guide from the readings:

```
    super.onCreate(savedInstanceState)

    setContentView(R.layout.activity_main)

    val binding = ActivityMainBinding.inflate(layoutInflater)

    val navHost = supportFragmentManager.findFragmentById(R.id.fragment_container) as
NavHost

    binding.bottomNav.setupWithNavController(navHost.navController)

    binding.bottomNav.setOnItemSelectedListener { item ->

        NavigationUI.onNavDestinationSelected(item, navHost.navController)
```

Note: in the code above, the line `setContentView(R.layout.activity_main)` is printed struck through: ~~setContentView(R.layout.activity_main)~~

```
        true
    }
    setContentView(binding.root)
```

Try running your app now. You should see the Gallery fragment displayed as the default home fragment, and it should display the "Gallery" TextView. You should be able to use the bottom navigation bar to navigate to the Map fragment (which displays the "Map" TextView) and click back and forth between the two fragments.

Notice that using the Back gesture/button from the Map fragment will return to the Gallery fragment, because the Gallery fragment is the first (and last) screen users will see. Using the Back gesture/button from the Gallery fragment will exit the app. If you're curious about the rationale for this behavior, it's specified as one of the main Navigation PrinciplesLinks to an external site. of Android.

Gallery Fragment Photo Grid

NOTE: To complete this portion of the assignment, you'll need to sign up for a Flickr account. This currently requires only your name, age, and email address. You're highly encouraged to use your `vt.edu` email address when you sign up, so that it's clear you're using the account in an academic setting.

Follow along with BNRG Chapter 20, which begins an app called PhotoGallery. Where the textbook indicates a PhotoGalleryFragment, instead please use the GalleryFragment you've already started implementing. For example, Listing 20.3 (which has a typo in the filename; it's a Fragment not an Activity) only requires us to add one new line of code, and Listing 20.4 requires no action. Please note also that all required dependencies have already been added to the configuration files you cloned from Git.

NOTE: You can't just download textbook solution for this chapter, because we're using the PhotoGallery as just one part of a much bigger app. Also, the dependency versions are very far out of date. As you work through the textbook for this assignment, you'll need to be careful to adapt everything to our existing project, and be sure not to change anything related to the bottom navigation, including the MainActivity and its layout. The total amount of code you need to develop is actually quite small, so please be sure to consider the textual descriptions of what the code is doing.

In **Listing 20.16** the textbook asks you to hard-code your API key directly into a source code file. This is a very bad practice in general from a security standpoint. Instead, we'll place the API key it in a separate configuration file, which is excluded from Git via a `.gitignore` file. In this way we'll prevent the key from being committed into the repository along with the source code. Although your Flickr API key isn't a particularly valuable token, we'll use better security practices than the textbook.

To store the key safely, temporarily change the project view from Android to Project. Right-click the project name and select New | File then enter the name as `apikey.properties`. Within the new file, you'll just need to add one line with your API key as follows:

Within `apikey.properties`:

```
FLICKR_API_KEY="01234567...89abcdef"
```

Of course you'll need to use your actual API key within the quotes. Please don't forget to revert from Project view back to Android view after you've done this.

Next, within `build.gradle.kts (Module :app)` please uncomment lines 28 through 33.

Then, within `api/FlickrApi.kt`, instead of hard-coding your API key as a constant, you'll access it from a build configuration parameter as follows:

```
~~private const val API_KEY = "yourApiKeyHere"~~

interface FlickrApi {


    @GET(

        "services/rest/?method=flickr.interestingness.getList" +

            ~~"&api_key=$API_KEY" +~~

            "&api_key=${BuildConfig.FLICKR_API_KEY}" +

            "&format=json" +

            . . .

        )

    suspend fun fetchPhotos(): String

}
```

Important: You'll need to use Build | Clean Project followed by Build | Rebuild Project for this change to take effect.

This still isn't a perfect solution, but it eliminates the most critical problem of accidentally committing a security token into a source code repository.

For the More Curious: The build configuration works in conjunction with the `build.gradle.kts (Module :app)` file, which contains a new build feature called "buildConfig", a helper function to read the properties file into an object, and the field specification you just un-commented. The `.gitignore` file mentioned above can be found in the Project view of the project, directly under the root node. You'll notice the last line of that file lists the `apikey.properties` file, which excludes it from any commits.

In **Listing 20.28** please name this file (and the class within it) as `GalleryViewModel` instead of `PhotoGalleryViewModel`.

In **Listing 20.29** please name the view model property as `vm` rather than `photoGalleryViewModel`. When running the app after making these changes, you should see "Items received" logged only once, exactly as BNRG mentions. However please note that you will still see "Response received" logged upon each device rotation. It's not very clear from the textbook that this is actually the expected behavior.

In **Listing 20.31** and **Listing 20.32** the textbook uses inconsistent naming conventions for the holder and adapter classes, as well as for the file name. Further, these names aren't very useful within the context of our larger app. Please change the adapter file name to `GalleryItemAdapter` then change the class names within to `GalleryItemHolder` and `GalleryItemAdapter`. Of course this will require corresponding changes to these names in the remaining listings.

In **Listing 20.36** please replace the `bill_up_close` drawable with a new Vector Asset suggesting an image placeholder. Alternatively, you may copy a new PNG Image Asset into the `res/drawable` folder to act as the placeholder. Either way, you must name this drawable resource as `ic_placeholder`. Also, please add the following cache option immediately after the placeholder call, to disable disk caching (thus only allowing memory caching):

```
    . . . {
        placeholder(R.drawable.ic_placeholder)
        diskCachePolicy(CachePolicy.DISABLED)
    }
```

When you reach ***Figure* 20.12** , please also confirm that your bottom navigation still works as expected.

Gallery Page Size Limit

When you run the app and scroll all the way to the bottom, you'll notice the last row contains only a single photo. That's because we're downloading the default page size of 100 photos, and showing three photos per row.

Instead of the BNRG Chapter 20 challenge regarding paging, which is overly complicated, we're going to allow our app to specify the exact page size of the returned response. You'll need to update `api/FlickrApi.kt` to accept a *required* parameter specifying the page size. Please see the following resources for more details:

- flickr.interestingness.getList APILinks to an external site. (Arguments section)

- Retrofit info pageLinks to an external site. (Query example in URL Manipulation section)
- Retrofit Query APILinks to an external site. (See first example)

NOTE: All the Retrofit samples are written in Java, so you'll need to convert these to Kotlin.

Update the PhotoRepository class to allow fetchPhotos() to accept an *optional* parameter for the page size. Specify a default parameter value of 100. Then modify the GalleryViewModel class to call fetchPhotos() with a page size parameter of 99.

Run the app again, and now you should see the photos evenly fill exactly 33 rows of the photo grid.

PHASE 2: Caching and Web Views

Gallery Cache Reload

When the Coil library loads the images from the network, it adds them to local caches so that they don't normally need to be fetched again from the network. However, the user might want to intentionally clear the cache, in order to load the latest content available. The *interestingness* list from Flickr is generally only updated once per day, but we'll still implement a feature to allow the user to fetch the latest content from the network.

Implement the Reload feature as an app bar menu. Create a new Vector asset to attach as an icon to a menu Item with title `Reload` (use a string resource) and id `reload_menu` (set this value in the Code tab). Show the text, along with the icon, if there's sufficient room.

Add this menu via a menu provider to the gallery fragment's Activity, ensuring that it's only displayed when the gallery fragment is on the screen. Use our previous project and associated module slides as guides, keeping in mind that the BNRG approach is obsolete.

When this menu item is selected, you must first clear the photo cache from the Coil image loader. See the ImageLoader APILinks to an external site. for details on how to access the (possibly null) memory cache and clear it. The ImageLoader object can be obtained from the context object of the GalleryFragment.

Next you'll need to implement a `reloadGalleryItems()` function in `GalleryViewModel`. This function must first clear the list of gallery items, then call `update` on the MutableStateFlow, to begin the process of fetching the photos again. All of this happens in a background thread within the view model's coroutine scope. You must create the coroutine scope on your own, but it's just one line of code, and there are several examples (including below). Here's what goes within the coroutine scope:

```
    _galleryItems.value = emptyList()

    _galleryItems.update {

        loadPhotos()

    }
```

The loadPhotos() above is a private helper function, which is basically the init block of Listing 20.28. The function body is:

```
    private suspend fun loadPhotos(): List<GalleryItem> {

        return try {

            val items = photoRepository.fetchPhotos(99)

            Log.d(TAG, "Items received: $items")

            items

        } catch (ex: Exception) {

            Log.e(TAG, "Failed to fetch gallery items", ex)

            emptyList()

        }

    }
```

The init block can now just call the helper from the coroutine scope:

```
    init {

        viewModelScope.launch {

            _galleryItems.value = loadPhotos()

        }

    }
```

This significantly reduces the redundancies between the init block and the `reloadGalleryItems()` function.

For the More Curious: The call to empty the gallery list isn't strictly necessary. It's there to make the testing much more reliable. It basically introduces an artificial delay, which helps Espresso to better recognize when the placeholders appear again, which is otherwise unreliable.

Run the app and wait until the placeholders for at least the first several rows have been replaced by loaded images. Next, when you click the reload menu, you should observe all the loaded images immediately replaced by placeholders for a moment, before starting to load again from the network. Also, while the app is running, double-check that you can still use the bottom navigation to switch to the map fragment and back.

Web Views

At this point we can view a scrollable gallery of photos. Next we want to be able to click one of the photos to show its associated web page in a custom browser-like environment.

Follow along with BNRG Chapter 23 to complete this section. Again we must be careful to take into account our existing bottom navigation, as well as our slightly different file, fragment, class, and other component names. You should notice a lot of similarities with our prior project in terms of clicking an item in a list to view the details of that item. Please first implement the "easy" way, via implicit intent.

In **Listing 23.4** you will notice that our GalleryFragment differs from the PhotoGalleryFragment of the textbook. This is because we've skipped Chapter 22. As a result, the code we add will be different (and less complex). Basically we just need to add the lambda to launch the activity:

```
vm.galleryItems.collect { items ->

    binding.photoGrid.adapter = GalleryItemAdapter(items) { photoPageUri ->

        val intent = Intent(Intent.ACTION_VIEW, photoPageUri)

        startActivity(intent)

    }

}
```

IMPORTANT: Once you've implemented **Listing 23.4** above, BNRG suggests that you should run the app at this point. Unfortunately, this won't work. There's a well-known bug in API levels since 31 that causes Chrome to freeze/crash on most emulators. You can simply keep working through the chapter without seeing the results at this point, or you can add a small configuration file to your system to enable Chrome to work properly.

The following steps are optional, but necessary to review the results of the "easy way" from BNRG Chapter 23:

1. Close your emulator
2. Download: **advancedFeatures.ini**Download advancedFeatures.ini
3. Copy this file into the `.android` folder within your user home folder, most likely:
   - Windows: `C:\Users\[your-user-name]\.android`

   - macOS: `/Users/[your-user-name]/.android`
   - Linux: `/home/[your-user-name]/.android`
4. Restart your emulator and run the app

Once you've completed the "harder" way, you may either move, remove, or rename this file (then restart the emulator) to revert it to its original configuration. Or, you may simply leave it in place as it won't harm anything.

In **Listing 23.7** the dependencies and plugins have already been added.

Starting immediately after **Listing 23.7**, and continuing through the end of the chapter, you must be careful not to interfere with the existing bottom navigation. You've already created the nav_graph, and added both GalleryFragment and the (mostly empty) MapFragment, so you won't need to add these. You will need to add the new PhotoPageFragment to the nav_graph. Please change its id to be `photo_page_fragment` for consistency with the others.

As in BNRG, please add an action arrow from GalleryFragment to PhotoPageFragment with id `show_photo`. You will then need to add an argument to PhotoPageFragment called `photoPageUri`. Its type will be Custom Parcelable `android.net.Uri` to PhotoPageFragment.called photoPageUri. All of this is essentially the same as noted in the textbook.

Skip **Listing 23.8** as this has already been done.

Continue with **Listing 23.9** and **Listing 23.10** as in the textbook, running the app as suggested after each step to examine the progress.

In **Listing 23.11** please change the `onReceivedTitle()` callback as follows:

```
    ~~val parent = requireActivity() as AppCompatActivity~~

    ~~parent.supportActionBar?.subtitle = title~~

    val parent = activity as AppCompatActivity?

    parent?.supportActionBar?.subtitle = title
```

Again, run the app after this step to confirm the proper behavior.

NOTE: Please beware that some required (and important) steps are listed only in the chapter text, and aren't associated with specific code listings, so please review this chapter carefully. Please *don't* implement the challenge at the end of the chapter.

For the More Curious: The code in BNRG Listing 23.11 doesn't cope well (crashes) if the user or a test case navigates quickly away from the photo page before the subtitle is fully loaded. Since loading the subtitle happens in the background, there's no guarantee that this will complete before the fragment has been detached from its activity.

There's still one minor problem we need to fix. When navigating Back from the PhotoPageFragment to the GalleryFragment, you may have noticed that the subtitle remains visible in the app bar. To avoid this, we need to override `onDestroyView()` of the PhotoPageFragment. There we must reset the app bar's subtitle to `null`.

Again, please also ensure at this point that you can still navigate back and forth between the Gallery and the Map (which still only shows the text "Map") via the bottom navigation bar. This should also work when the PhotoPage is being displayed from the Gallery fragment.

PHASE 3: Maps and Markers

We're finally going to implement the Map portion of the app. At first we just need to display a blank map in a MapView component. The user should just be able to pan and zoom arbitrarily at this point, but eventually we'll position interactive photo markers on the map based on their source geo-location.

To do all of this, we'll use an open source library called OSMDroid, which is provided open and free by the OpenStreetMapLinks to an external site. Foundation. The interface is nearly identical to every other map provider, including Google Maps, although most of the alternatives involve complicated (and often expensive) licensing schemes.

There's nothing in BNRG to help us here, so the sections below will contain a more background information and details than usual.

Map View

There are only two required dependencies to use OSMDroid, and they've been added to the build.gradle.kts (Module:app) file.

Open the `fragment_map.xml` layout page. Right-click the TextView and select Convert View. In the popup, type "MapView" then select the `org.osmdroid.views.MapView` component and click Apply. Make sure this component fills the entire screen, and has an id value of `map_view`.

In order to use the map, it must be configured from the MapFragment, to send the application ID value as a user agent string. This acts a bit like the API key for Flickr, but it doesn't require any account sign-up.

Override the onViewCreated() function of MapFragment, then add the following contents:

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

        super.onViewCreated(view, savedInstanceState)
        Configuration.getInstance().load(
            context,
            PreferenceManager.getDefaultSharedPreferences(requireContext())
        )
        Configuration.getInstance().userAgentValue = requireContext().packageName
    }
```

NOTE: When resolving the import package for the Configuration class above, please select `org.osmdroid.config.Configuration`.

Run the app now, and select the Map fragment from the bottom navigation. You should see a rather zoomed-out map of the world. Actually you'll see quite a few world maps, tiled together to fill the screen. If you zoom in (using the `+` button) enough to see the map's annotations, you'll also notice that all the fonts are quite tiny. In fact, the more you zoom, the smaller the font will appear. You can pan to any location using a drag/swipe gesture. It's definitely a map that works, and didn't require many lines of code at all, but it obviously needs a few adjustments.

Enhanced Map View

To fix some of the limitations of the default implementation, we need to limit the maximum and minimum zoom level, and limit the view to a single map rather than a bunch of tiles. We'll also display the zoom buttons always, scale the fonts to the `sp` units of the screen, and select a bulk tile source that can handle reasonable amounts of requests from apps like ours.

We'll accomplish all of this by calling some functions on the Map component, again within the onViewCreated() function. Note that all of this can be found in the OSDroid API from the readings, but it's assembled here for convenience:

```
    Configuration.getInstance().userAgentValue = requireContext().packageName


    binding.mapView.apply {
        minZoomLevel = 1.5
        maxZoomLevel = 15.0
        setScrollableAreaLimitLatitude(
            MapView.getTileSystem().maxLatitude,
            MapView.getTileSystem().minLatitude,
            0
        )

        isVerticalMapRepetitionEnabled = false

        isTilesScaledToDpi = true

        zoomController.setVisibility(CustomZoomButtonsController.Visibility.ALWAYS)

        setTileSource(TileSourceFactory.MAPNIK)

    }
```

Run the app again, and you should see the map starts at a reasonable zoom level, and when you zoom in now the fonts should be easily readable.

However, you might notice when you leave the Map view and return from the Gallery, the map reverts to its initial center/zoom state. To fix this, we'll need to store and then restore the map center and zoom level values. However, we don't currently have a good place to store them. The Fragment will be destroyed, once it's no longer on the screen, so that won't work. Instead we'll need to create a ViewModel to store these values beyond the lifetime of the Fragment.

Create a new Kotlin class called `MapViewModel` as a subclass of `ViewModel`. We'll add two properties, `mapCenter` and `zoomLevel`, to store the state of the map. These must have reasonable initial values, because the first time the Fragment is resumed it will fetch these values. In keeping with classic object-oriented encapsulation, the properties will be read-only and set only by an explicit public function:

```
    var zoomLevel: Double = 1.5

        private set


    var mapCenter: IGeoPoint = GeoPoint(0.0, 0.0)

        private set


    fun saveMapState(zoom: Double, center: IGeoPoint) {

        zoomLevel = zoom

        mapCenter = center

    }
```

Now, update `MapFragment` to fetch the view model as private property `vm` (you already know how to do this) and to use it as a way to store/restore these values in the `onPause()` and `onResume()` overrides. Note that we must also call osmdroid default `pause()` and `resume()` functions as well, to properly handle the network connections and tile caching:

```
    override fun onPause() {
        super.onPause()
        with(binding.mapView) {
            vm.saveMapState(zoomLevelDouble, mapCenter)
            onPause()
        }
    }

    override fun onResume() {
        super.onResume()
        with(binding.mapView) {
            onResume()
            controller.setZoom(vm.zoomLevel)
            controller.setCenter(vm.mapCenter)
        }
    }
```

Now run the app, and watch how the zoom level and map center are retained when switching back and forth between the Map view and the Gallery view using the bottom navigation. However, using the Back navigation from the Map view will destroy the MapViewModel, after which returning to the Map view will display the default zoom/center. It's of course possible to store the state more persistently (and we even know how to do this using a database) but the current behavior is perfectly fine for our purposes.

Sharing Data Across Fragments

Now that the basic map is working, we need to consider how the Map fragment can access the photo information that's currently held in the Gallery fragment's associated view model. Specifically, we want to share the list of GalleryItem objects with the Map fragment, so the latter can display the photos on the map. However, Android doesn't provide a reliable way for any fragment to access any other fragment's view model, so we have a bit of a problem.

The solution for this (very common) case is to associate the shared view model with the Activity that hosts both of the fragments. Both the Map fragment and the Gallery fragment can access a view model that's associated with the Activity that hosts them (namely MainActivity).

What we need to share is currently in the GalleryViewModel, so that's the VM we'll promote to the scope of the activity. Right-click the `GalleryViewModel` filename and select Refactor | Rename to name the file `MainViewModel`. Then open the new `MainViewModel` file to confirm the class declaration has also been changed to `MainViewModel` (and change it if not). Note that this hasn't actually changed anything about the scope of the VM yet, but it will help us keep everything organized.

Open the `GalleryFragment` file, and you'll see that the Refactor operation has already updated the class name of our VM. Now we just need to make one minor change to the delegation in order to update its scope:

```
~~private val vm: MainViewModel by viewModels()~~

private val vm: MainViewModel by activityViewModels()
```

Using the delegation `by activityModels()` will associate the `MainViewModel` with the hosting activity `MainActivity`, rather than to the fragment itself.

Run the app again and you should see no changes. There should be no indication that the scope of the VM has changed behind the scenes, which is exactly what we want. The gallery should still work as it did before. However, behind the scenes, the gallery VM is now also accessible from the Map fragment!

Now open the `MapFragment` file, and add a new line of code to also access this shared (activity-scope) VM, in addition to the existing `MapViewModel`:

```
private val vm: MapViewModel by viewModels()

private val activityVM: MainViewModel by activityViewModels()
```

That's all fairly straightforward, but how do we know that it worked? Let's just temporarily add a line to the end of `onResume()` of `MapFragment` that simply logs the number of photos found in the (now-shared) VM.

```
override fun onResume() {

    . . .

    Log.w(

        "SHARED VM TEST",

        "Found ${activityVM.galleryItems.value.size} gallery items!"

    )

}
```

Launch the app again, and open Logcat, and switch to the Map fragment using the bottom navigation. Each time the Map fragment is displayed, you should see this entry displayed in Logcat:

```
edu.vt.cs5254.fancygallery W/SHARED VM TEST: Found 99 gallery items!
```

This proves that the Map fragment can actually access the list of Gallery items stored in the Main view model (formerly the Gallery view model).

Loading Coordinates from Flickr

Now that our Map fragment can access the gallery items, we need to update the Gallery to request map coordinates when fetching the item details from Flickr. Again review the API from Flickr:

- flickr.interestingness.getList APILinks to an external site.

Notice that `geo` is listed as one of the comma-delimited `extras` we can request. Let's try this out in the Flickr API explorer, to see what we'll receive as a response.

- flickr.interestingness API ExplorerLinks to an external site.

In this page, leave everything blank except check Send checkbox in the `extras` line, and enter `geo` as the associated value. Then select "JSON" output and then click the Call Method button.

In a few moments you'll see the response, which you'll notice includes a `latitude` and `longitude` for each image. However, you may also notice that most of the images have latitude and longitude values of 0. This indicates that the photo wasn't tagged with geo coordinates, so we'll eventually need to handle that case.

We're already requesting `url_s` you'll need to make a minor adjustment to our `FlickrApi` file, in order to request both `url_s` and `geo` as extras. You'll need to do this on your own, but it's fairly straightforward.

Now that we've updated the request to include the `geo` details, we need to adjust our model so that the Moshi library can parse the returned data. In the `GalleryItem` file, we need to add two new properties, `latitude` and `longitude`, both as type `Double`. Again, you'll need to do this on your own, and again you can just use the existing properties as a guide.

Back in the `MapFragment` file, we're currently only logging the overall number of GalleryItem objects found. Next we'll do something much more sophisticated.

Let's do that, and count the number of gallery items with valid (non-zero) latitude and longitude data. At the end of `onViewCreated()` of the `MapFragment` file, add this:

```
    val count = activityVM.galleryItems.value.count {

        it.latitude != 0.0 && it.longitude != 0.0

    }

    Log.w("MAPFRAGMENT", "GOT $count items with valid GEO!")
```

Run the app again with Logcat open. You should see an entry similar to this in Logcat every time you navigate to the Map fragment:

- . . . edu.vt.cs5254.fancygallery W/MAPFRAGMENT: GOT 21 items with valid GEO!

The exact number you'll see will vary from this, generally day by day and sometimes even within each day. Typically you should see something on the order of about 20 or so, but don't worry about the exact count (unless it's zero, which may indicate a problem).

Map Markers

Now that our Map fragment has access to the geo-coordinates for the photos, and can determine which are valid, let's use these valid locations to place markers on the map. The osmdroid library makes this easy to implement. Instead of logging the count of valid coordinates, we'll create a new Marker for each valid coordinate and add it to an overlay.

```
    ~~val count = activityVM.galleryItems.value.count {~~

~~        it.latitude != 0.0 && it.longitude != 0.0~~

~~    }~~

~~    Log.w("MAPFRAGMENT", "GOT $count items with valid GEO!")~~

    activityVM.galleryItems.value.filter { it.latitude != 0.0 && it.longitude != 0.0 }

        .forEach { galleryItem ->

            val marker = Marker(binding.mapView).apply {

                position = GeoPoint(galleryItem.latitude, galleryItem.longitude)

                title = galleryItem.title

            }

            _binding?.mapView?.overlays?.add(marker)

        }
```

Recall that the GalleryItem objects in the VM are actually wrapped with in a StateFlow object. This means we can readily observe the any changes to the underlying values by invoking `collect` on the StateFlow. The `collect` must be done within a coroutine scope, so we should implement a scope that's repeatedly created whenever the fragment is in the `STARTED` state. We've done this before, so it should seem familiar. Please see the `onViewCreated()` function of `GalleryFragment` how to collect on the list of GalleryItem objects from the activityVM (still called `vm` in `GalleryFragment`).

Copy this scope wrapper into the `onViewCreated()` of `MapFragment` and adapt it accordingly. Place the marker code from the listing immediately above into the new coroutine scope. You need to do this on your own. As a hint, this is one of the major changes involved:

```
    ~~activityVM.galleryItems.value.filter { it.latitude != 0.0 && it.longitude != 0.0
}~~

    activityVM.galleryItems.collect { items ->

        items.filter { it.latitude != 0.0 && it.longitude != 0.0 }
```

Run the app now and navigate to the Map fragment. You should see approximately 20 markers spanning the locations around the world where photos were taken. Click any marker and you'll see a popup appear with the title of the associated photo. Click the popup (or any other marker) to close the popup. We've successfully formed a logical linkage between our Gallery and our Map!

Photo Markers

The next step is to replace each default marker with the associated image from our gallery. We have all the access we need, so it's just a matter of loading the images into the map from the Map fragment. The Coil library does a great job of loading images, but we've only used Coil to to load an image directly into an ImageView component. Here we must do a bit of that work on our own. From the API, you can see that Each marker is capable of displaying a drawable, so let's develop a helper function near the bottom of `MapFragment` to fetch a drawable from its URL.

We'll need to fetch an `ImageLoader` object from Coil, and build an `ImageRequest` from the URL. Then we can execute the request from the loader.

```
    override fun onDestroyView() {

        . . .

    }



    private suspend fun loadDrawableFromUrl(url: String): Drawable? {
        return context?.let {

            val loader = it.imageLoader

            val request = ImageRequest.Builder(requireContext())

                .data(url)

                .build()

            return try {

                val result = loader.execute(request)

                (result as SuccessResult).drawable

            } catch (ex: Exception) {

                null

            }

        }

    }
```

Notice that this function returns `Drawable?` meaning it may return null. This is meant to indicate any problems loading the image, such as a network access timeout, or the image being temporarily unavailable. Also notice that this function must be declared as `suspend` because all network access must be done within a separate background thread, and hence this function must called from within a coroutine scope. Luckily we were already planning to call this from an existing coroutine scope.

Please update the implementation in `onViewCreated()` as follows:

```
    .forEach { galleryItem ->

        val photoDrawable = loadDrawableFromUrl(galleryItem.url)

        photoDrawable?.let { drawable ->

            val marker = Marker(binding.mapView).apply {

                position = GeoPoint(galleryItem.latitude, galleryItem.longitude)

                title = galleryItem.title

                icon = drawable

            }

            _binding?.mapView?.overlays?.add(marker)

        }

    }
```

Run the app and navigate to the Map fragment. Scroll around and you'll see each marker is display as photo that was taken at the marked location. This is great progress, but there are two minor issues you may notice.

First, you may notice that when you initially select the Map fragment after launching the app, not all the markers will be displayed at first. We'd like to show them all, once they're loaded. This may be difficult to detect if you have a reasonably fast development platform and network connection, but it's something we should address.

Second, click one of the markers, which will center it, and try to zoom in many times. You'll likely notice that the marker seems to shift or drift somewhat if you zoom far enough. That's because the default marker's anchor is set to the bottom center, where its pointer indicates. However, since we're using photos, we'd like the photos to be anchored at the center, directly on top of the associated location.

The first issue can be resolved by invalidating the MapView component after the forEach has completed, forcing it to be rendered again after all the markers have been loaded. The second issue can be fixed easily by explicitly setting the anchor point of the marker to be at the horizontal and vertical center:

```
                    icon = drawable
                    setAnchor(Marker.ANCHOR_CENTER, Marker.ANCHOR_CENTER)
                }
                _binding?.mapView?.overlays?.add(marker)
            }
        }
    _binding?.mapView?.invalidate()
```

For the More Curious: You may notice we're using `_binding` here rather than `binding`. That's because this lambda may take a significant amount of time to complete in the background, yet several test cases (and some human users!) only wait long enough for the first one (or few) of the markers to load, then they move on to other tasks. The problem is that this entire fragment may be gone by the time some of the markers have loaded. Using `_binding` lets the code gracefully degrade and do nothing if the fragment has gone away. Similarly the `loadDrawableFromUrl()` function earlier in this section handles the case where the context has gone away before the drawable has been fetched.

When you launch the app now, you'll see the images won't seem to drift nearly as much when you zoom in, and the view will refresh itself once all of the photo markers have loaded (again this might be difficult to detect).

Interactive Markers

At this point, the default action of clicking a photo marker is to center the map on the marker and show the photo title in a popup. As our last feature, we'd like to take this another step further. We'll still show the popup upon the first click, but we'll navigate to the Photo Page if the marker is clicked again (while the popup is displayed).

In the osmdroid library, each marker can be associated (they call it "related") with any arbitrary object. We'll use this to associate each marker with the GalleryItem at this location. Then we'll specify a listener to handle the marker click logic:

```
            icon = it
            setAnchor(Marker.ANCHOR_CENTER, Marker.ANCHOR_CENTER)
            relatedObject = galleryItem
            setOnMarkerClickListener { marker, _ ->
                if (marker.isInfoWindowShown) {
                    val item = marker.relatedObject as GalleryItem
                    findNavController().navigate(
                        MapFragmentDirections.showPhotoFromMarker(item.photoPageUri)
                    )
                } else {
                    showInfoWindow()
                }
                true
            }
        }
        _binding?.mapView?.overlays?.add(marker)
```

Meanwhile, even after importing `findNavController()` you'll notice an error in red, because `MapFragmentDirections` hasn't been defined yet. To fix this, we must update the `nav_graph` accordingly. The `MapFragment` needs a navigation leading to the photo page, similar to how the `GalleryFragment` leads to the photo page. You'll need to do this on your own, making sure the action id matches the code above.

Once you've completed this step, run the app again. The markers in your app should still show the title popup as before upon the first click. But now clicking a second time should take you to the photo page fragment for that photo, and using the Back button/gesture from there should return you to the Map. This is great progress, and we're actually almost done now. There are just two minor issues remaining to address.

You've probably noticed that markers will often overlap each other, until we zoom in fairly substantially. It would be nice if clicking a marker would raise it to display in full above the other markers. There's no explicit "z" index, but we can simply remove the marker from the list of overlays, then add it back to the end of the list. This will ensure it's the last marker to be drawn, which means it will be effectively displayed "above" all the others.

You might also notice that we've actually lost a minor feature that we had before we implemented interactive markers. Clicking a marker used to smoothly shift the map so that the clicked marker appeared at the center of the screen. Our custom listener above doesn't do that, but we can add that feature easily enough.

Let's fix both of these issues with just a few lines of code:

```
    setOnMarkerClickListener { marker, _mapView ->
        mapView.apply {
            controller.animateTo(marker.position)
            overlays.remove(marker)
            overlays.add(marker)
        }
        if (marker.isInfoWindowShown) {
```

Run the app again, and notice that clicking a photo marker will now center the map around it. It will also raise it above the other markers, so that it will be fully visible even if there were any markers partially overlapping it.

Congratulations! You've completed the rather extensive FancyGallery app. Hopefully you've enjoyed the process of completing this assignment.