Project 3 - Gallery

5/4/2022

40/40 Points

Attempt 1

REVIEW FEEDBACK
4/28/2022

Attempt 1 Score:
40/40

View Feedback

Unlimited Attempts Allowed

Details

Due: Wednesday, May 4
Points: 40 points
Deliverables: A ZIP file of your project
Resources:

- build.gradle (https://drive.google.com/file/d/1_qPX6hPY45ViAw-uucUkzBfOAF8ftGnO/view?usp=sharing) (app)
  - Subject to minor tweaks
- MapViewFragment (https://drive.google.com/file/d/1FuKF_j2QGGCHc1INeL4OkQRPM45rQZCS/view?usp=sharing)
- PictureUtil (https://drive.google.com/file/d/1FdePZ9qC8RCTFv0Na2QMENHH_vRLnUke/view?usp=sharing) (for using cached drawables in the map fragment)

Overview

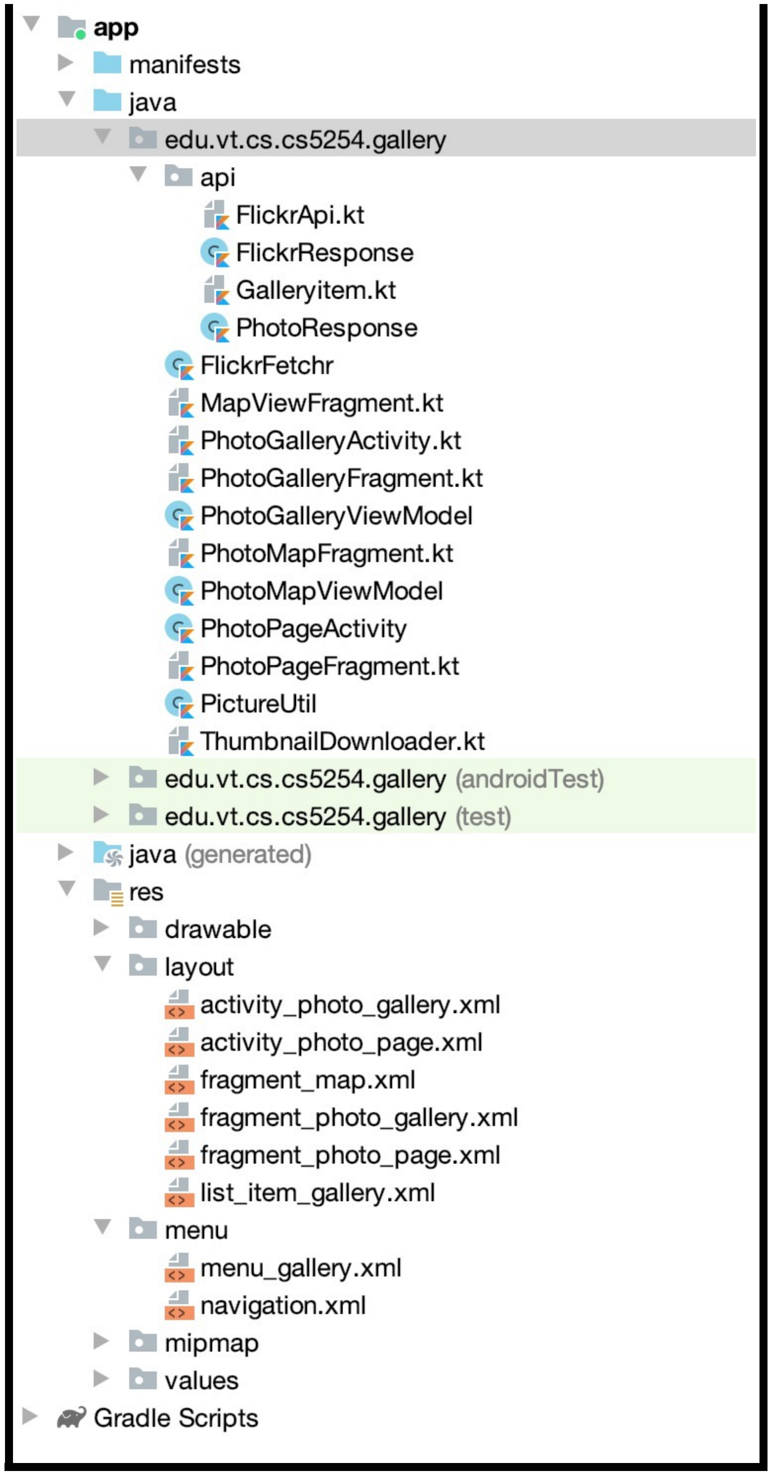
In this project you will create an application called **Gallery** (do not call it PhotoGallery like BNR) that will display photos from Flickr in both a grid view and a map view. It will also allow you to click on a photo and see the photo's Flickr page using a web view.

The project will have functionality very similar to chapters 24, 25, and 29 of BNR. In addition, it will have a map view that shows your photos on a map of the world based on the GPS data in those photos. It will also have a bottom navigation bar that allows you to switch between the gallery and the map views.

Video Demo

Here is a short (~1 minute) video demo of how the app should behave.

The files your finished application contains will look something like this.



```
▼ app
  ▶ manifests
  ▼ java
    ▼ edu.vt.cs.cs5254.gallery
      ▼ api
          FlickrApi.kt
          FlickrResponse
          Galleryitem.kt
          PhotoResponse
        FlickrFetchr
        MapViewFragment.kt
        PhotoGalleryActivity.kt
        PhotoGalleryFragment.kt
        PhotoGalleryViewModel
        PhotoMapFragment.kt
        PhotoMapViewModel
        PhotoPageActivity
        PhotoPageFragment.kt
        PictureUtil
        ThumbnailDownloader.kt
    ▶ edu.vt.cs.cs5254.gallery (androidTest)
    ▶ edu.vt.cs.cs5254.gallery (test)
  ▶ java (generated)
  ▼ res
    ▶ drawable
    ▼ layout
        activity_photo_gallery.xml
        activity_photo_page.xml
        fragment_map.xml
        fragment_photo_gallery.xml
        fragment_photo_page.xml
        list_item_gallery.xml
    ▼ menu
        menu_gallery.xml
        navigation.xml
    ▶ mipmap
    ▶ values
▶ Gradle Scripts
```

Implementation: Photo Gallery and Web View

Use a combination of BNR chapters 24, 25, and 29, along with Dr. K's slides and Q&A recordings to implement the gallery portion of the project and the web-view portion of the project. For the WebView, it should be easy enough to follow along with Chapter 29 in BNR. However, please note the following:

- Do both parts (easy way and hard way). The "easy way" sets you up for the "hard way", and the hard way is not actually that hard
- Instead of VisibleFragment(), use Fragment()
- Instead of (intent.data), use (intent.data as Uri)

Implementation: Bottom Navigation

Make sure you have the latest version of the **build.gradle (https://drive.google.com/file/d/1_qPX6hPY45ViAw-uucUkzBfOAF8ftGnO/view?usp=sharing)** file installed. In this section, we will create a bottom navigation bar that will switch between you gallery view and an empty map view. Before we can even get to the empty map view, you have to have a Google map key. Follow along with the document below to see how to sign up for a Google cloud platform account and obtain an API key.

google-map-key.pdf (https://drive.google.com/file/d/15Ait1-UgTbPr6GfWRlhRi8DobGaEZmb7/view?usp=sharing)

Once you have a Google map key, you will have to place it in your Android manifest. Inside the <application> section of the manifest, place the following code, with your key substituted for the false key value. Note that all Google map keys start with `AIza`.

```
<meta-data
    android:name="com.google.android.geo.API_KEY"
    android:value="AIzaXXXXXXX-XXXXXXXXXXXXXXXXXXXXXXXXXXX"/>
```

Create a layout file for the PhotoMapFragment that your will add. First, create a layout resource named `fragment_photo_map.xml`, using a constraint view. Inside the top-level view, add a MapView with ID of `map_view`. Its height should be wrap_content and its width should be 0dp. Constrain all sides of the map view to its parent.

Add the file **MapViewFragment (https://drive.google.com/file/d/1FuKF_j2QGGCHc1INeL4OkQRPM45rQZCS/view?usp=sharing)** to your project. This file will handle a most of the administrative functions of getting the map-view up and running. Notice, in particular, that the class has three public methods that are not lifecycle methods:

- **onCreateMapView**

- **onMapViewCreated**
- **setMarkerIcon**

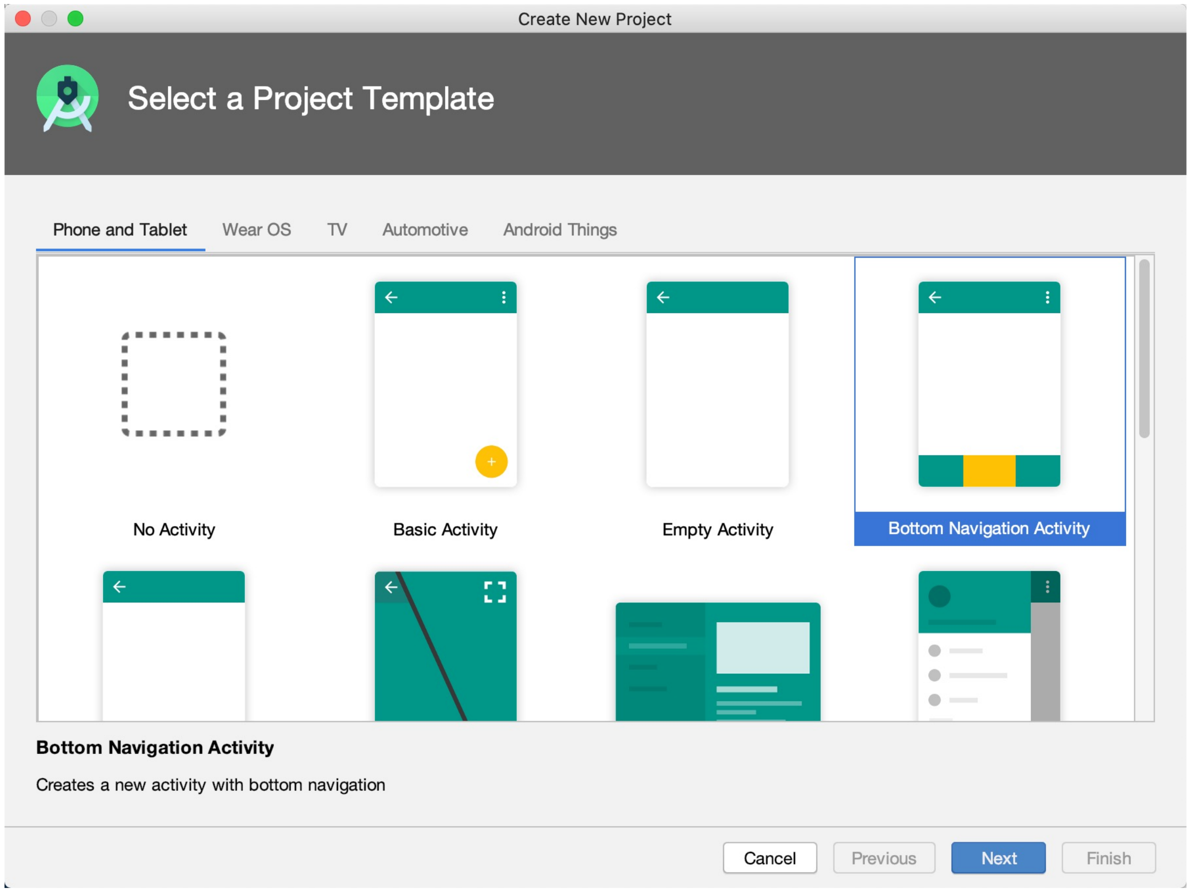
The first two help set up the map-view object. They will be called by PhotoMapFragment. The third function is used to place photos on the map and will be used in a later section.

Now create class PhotoMapFragment that extends MapViewFragment. For now, we're only going to implement two functions.

- **onCreateView** - make this call through to the parent's onCreateMapView. You already have access to all the parameter values. For mapLayoutResouceId, pass in R.layout.fragment_photo_map. For mapViewResourceId, pass in R.id.map_view.
- **onViewCreated** - make this call through to the parent's onMapViewCreated. You already have access to the view and the bundle. For the "block", pass in an empty function: `{ googleMap -> {} }`. We will fill in the details later.

Bottom Nav implementation

The easiest way to implement the Bottom Nav bar is to "borrow" code from Android Studio's Bottom Navigation Activity template.



Bottom Navigation Activity

Creates a new activity with bottom navigation

Before you do, create new image assets for the navigation bar. You will need two of them:

- **ic_gallery** – An icon for your gallery fragment, which is the default fragment
- **ic_map** – An icon for your map fragment

Note that Android's bottom navigation template uses a light background for the navigation bar, so the icons it uses are black. If you want to use a dark background for the navigation bar, you will have to adjust the color of the bar and the color of your icons.

Once you have the icons, create a dummy project (like TestBottomNav) in order to get the code. There are four main files that you will need to modify your project.

- **mobile_navigation.xml** – Create a "Navigation" resource with this same name in your Gallery project and paste in this code. Then modify the code so that it makes sense for the Gallery project. Note that we only navigate between 2 fragments (not 3), so you can eliminate one. Give the fragments IDs of navigation_gallery and navigation_map. Also, you will want to create titles for the icons in strings.xml: "Gallery" and "Map".

- **bottom_nav_menu.xml** – Create a "Menu" resource with this same name in your Gallery project and paste in this code. Then modify the code so that is makes sense for the Gallery project.
- **activity_main.xml** – Replace the Frame layout in activity_photo_gallery.xml with the code in here. If you see extra white space at the top of your screen when you run your app, remove the top padding in the constraint layout.
- **MainActivity.kt** – Replace the code in PhotoGalleryActivity with the code from this file. Make sure you replace the imports as well. As long as you are using the most up-to-date gradle file we gave you, you should not have problems. You will have to modify the set of fragment ID's to reflect what we are using in the Gallery app.

Run your application. Everything should work as before, but now you should have a bottom navigation bar. When you click on the map icon it should take you to a view with a Google map but no photos.

There are a couple of things to notice here. First, because of the way we implemented the bottom navigation, and therefore the photo gallery activity, we no longer need a companion object for the photo gallery fragment. The companion object held the new-instance function. Second, when you are in the map view and you click back into the gallery view, you will briefly see your placeholder images while the Flickr images are downloaded. That is because we have not yet cached the thumbnails. We will walk through that in a later section.

Implementation: Add Markers to the Map

In a recycler view, information is stored in view holder; in a map view, information is stored in a marker. In this section we're going to add marker for all the downloaded photos that have a valid latitude-longitude coordinate. All lat-lon pairs except (0, 0) are considered valid.

To obtain latitude and longitude form the Flickr API:

1. Add latitude and longitude properties to GalleryItem
2. Add "geo" to the list of extras in FlickrApi: "&extras=url_s,geo"

Once we set up gallery items to hold location information, we are going to set up the photo map fragment to download the gallery items, just as we did with photo gallery fragment. It's not very efficient to download all the photo data again, but we will fix that later. To download the photos, do the following.

1. Create PhotoMapViewModel with the same implementation as PhotoGalleryViewModel
2. Declare and initialize the view-model in PhotoMapFragment

3. Observe the gallery-item live-data in onViewCreated. Instead of updating an adapter with the gallery items, simply call a method named updateUI and pass in the gallery items
4. Implement updateUI(List<GalleryItem>)

Here is the implementation for updateUI.

```
private fun updateUI(galleryItems: List) {

        // if the fragment is not currently added to its activity, or
        // if there are not gallery items, do not update the UI
        if (!isAdded || galleryItems.isEmpty()) {
                return
        }

        Log.i(TAG, "Gallery has has " + galleryItems.size + " items")

        // remove all markers, overlays, etc. from the map
        googleMap.clear()

        val bounds = LatLngBounds.Builder()
        val latLngGalleryItems =
                galleryItems.filterNot { it.latitude == "0" && it.longitude == "0" }
        for (item in latLngGalleryItems) {
                // log the information of each gallery item with a valid lat-lng
                Log.i(
                        TAG,
                        "Item id=${item.id} " +
                                        "lat=${item.latitude} long=${item.longitude} " +
                                        "title=${item.title}"
                )
                // create a lan-lng point for the item and add it to the lat-lng bounds
                val itemPoint = LatLng(item.latitude.toDouble(), item.longitude.toDouble())
                bounds.include(itemPoint)

                // create a marker for the item and add it to the map
                val itemMarker = MarkerOptions().position(itemPoint).title(item.title)
                val marker = googleMap.addMarker(itemMarker)
                marker.tag = item.id
        }

        Log.i(TAG, "Expecting ${latLngGalleryItems.size} markers on the map")
}
```

Run your application. You should see markers on the map view for gallery items with location information. You log message will tell you how many photos (out of 100) have valid coordinates.

Implementation: Replace Markers with Photos

To replace the markers with photos, we only need to modify the photo-map fragment to use the thumbnail downloader. It will be similar to the way photo-gallery fragment uses the thumbnail downloader, but we will work with markers instead of view-holders.

- Declare a thumbnail downloader with Marker as its type.
- In onCreate, initialize the thumbnail downloader with a new handler (call it responseHandler) and a function. The function you pass in should take a marker and a bitmap and use setMarkerIcon to associate them.
- Add the fragment lifecycle observer in onCreate and remove it in onDestroy. Add the view lifecycle observer in onCreateView and remove it in onDestroyView.
- In updateUI, you should have a loop (either for or forEach will work) that sets up map markers for valid gallery items. At the end of that loop, use the thumbnail downloader to queue the URL with its associated marker.

And that's it! Run your application and now you should have photos displaying instead of default markers in your map view.

Implementation: Click Photo for WebView

To start a web-view when clicking on a photo, you need to implement the OnMarkerClickListener.

- Make PhotoMapFragment implement Google map's OnMarkerClickListener.
- Declare latLngGalleryItems as a var, and assign it to an empty list of gallery items. In updateUI, remove the "val" declaration from latLngGalleryItems -- now you are updating the field.
- In onViewCreated, we called onMapViewCreated and passed in a function as the last parameter:
  `googleMap -> {}`
  Change this to set the on-marker-click listener to the PhotoMapFragment:
  `googleMap -> googleMap.setOnMarkerClickListener(this@PhotoMapFragment)`
- Finally, implement onMarkerClick by doing the following.
  1. Get the ID from the marker's tag
  2. Log a message saying that the marker (include its ID) has been clicked on
  3. Get the gallery item from the list of lat-lng gallery items; you can use the find method to check which item has a matching ID.
  4. Get the URI of the item
  5. If the URI is null, return false

6. Otherwise, create a new intent for PhotoPageActivity and start it (as was done PhotoHolder of PhotoGalleryFragment)
7. Return true

Run the application. Now clicking a photo should take you to the web view and display that photo.

Implementation: Caching Gallery Items

Item List to Item Map in Map View

In PhotoMapFragment, change the field latLngGalleryItems (which is a list) to geoGalleryItemMap (which is a map). The renaming to "geo" should help avoid spelling errors. The map will map ID's (strings) to gallery items. Initialize the declared field to an empty map.

In updateUI, when you populate the map, filter the gallery items into a list (as before), then use the function "associateBy" to associate the values (gallery items) with their IDs. Once you have the map, iterate through the values of the map. In onMarkerClick, get the gallery item (and then the url) from the ID.

Eventually, we're going to move the map into the view-model (where it belongs). But before we do, we are going to cache the gallery items inside the repository.

Caching Gallery Items

We are going to store the gallery items in the repository (FlickrFetchr), so that both photo gallery fragment and photo map fragment can access the same items.

Move the local variable responseLiveData out of fetchPhotos and make it a field. Remove the return type from fetchPhotos and remove the return statement at the end of the method. Add a guard to the beginning the method: if responseLiveData already has something in it, it means that the gallery items have already been loaded, so return from the method. Now change both view models so they initialize galleryItemLiveData to FlickFetcher.responseLiveData. Note: it's probably a good idea to rename responseLiveData to galleryItemsLiveData since that is what it represents.

Run your application. There should be no errors, but the photos are never loaded. That's because fetchPhotos is no longer called. Eventually, we want to call this in two instances: when the application starts and when a reload is forced. We don't have a reload button yet so we'll focus on loading when the app starts.

Put a method in the gallery view-model called loadPhotos that calls through to fetchPhotos, and call loadPhotos in PhotoGalleryFragment::onCreate, just after you initialize the view model.

Run your application. The photos should load now, but when you switch to the map view, the application will probably crash. If you look at LogCat, you will see something like "lateinit property googleMap has not been initialized". To fix this, add the following method in MapViewFragment:

```
protected fun mapIsInitialized() = this::googleMap.isInitialized
```

Use this method at the beginning of updateUI to check if the map is initialized, and return if it isn't. This will avoid the error, but it still won't put photos on your map. That's because you are no longer fetching the items from Flickr (the gallery view already did that), and the map that you are getting from Google is not available yet.

We're going to fix this, but before we do, we are going to return to the gallery-item map and fix the view-model for the map fragment.

Put Item-Map into View-Model

Add geoGalleryItemMapLiveData to your map-view's view model:

```
val geoGalleryItemMapLiveData: LiveData<Map<String, GalleryItem>> =
    Transformations.switchMap(galleryItemsLiveData) {items ->
        val geoGalleryItemMap =
            items.filterNot { it.latitude == "0" && it.longitude == "0" }
                .associateBy { it.id }
    MutableLiveData<Map<String, GalleryItem>>(geoGalleryItemMap)
}
```

This updates the geo-map whenever the item list changes. Instead of observing the item-list in PhotoMapFragment::onViewCreated, observe the geo-map. When the geo-map is updated, assign it to the geo-map field that you created earlier. Get rid of the argument from updateUI, remove the local geo-map, and use the goe-map field instead.

Update the UI when the GoogleMap Loads

MapViewFragment has a method called onMapViewCreated that effectively takes a block of code to be executed once the Google map is obtained. Right now we are using it (inside PhotoMapFragment::onViewCreated) to set the photo map fragment to be a marker-click listener for the map. After that statement, add updateUI().

Implementation: Caching Thumbnails

Add the following annotation and parameter to galleryItem. This is where the thumbnail will be stored.

```
    @Expose(serialize = false, deserialize = false)
    var drawable: Drawable? = null // local cache for thumbnail
```

Add the following function to FlickrFetchr. This ensures that the drawable will be added to the appropriate gallery item in the repository's list.

```
    fun storeThumbnail(id: String, drawable: Drawable) {
        galleryItemsLiveData.value?.find { it.id == id }?.drawable = drawable
    }
```

Add the following call-through function to PhotoGalleryViewModel. This calls through to the repository.

```
    fun storeThumbnail(id: String, drawable: Drawable) {
        FlickrFetchr.storeThumbnail(id, drawable)
    }
```

Add the following to PhotoGalleryFragment::onCreate when initializing the ThumbnailDownloader.

```
  photoGalleryViewModel.storeThumbnail(photoHolder.galleryItem.id, drawable)
```

This should be at the end of the block (argument for onThumbnailDownloaded). For this to work, galleryItem in PhotoHolder must be public.

Add the following to the end of onBindViewHolder (in PhotoAdapter)

```
    holder.bindDrawable(galleryItem.drawable ?: placeholder)
    if (galleryItem.drawable == null || galleryItem.drawable == placeholder) {
        thumbnailDownloader.queueThumbnail(holder, galleryItem.url)
    }
```

If you run the application now, what happens? See the fix in the followup video to the Q&A in Module 12. If you would like to implement getting access to the gallery-item drawable from the map fragment, please see related posts on Piazza. However, that will not be graded.

Implementation: Reload Button

Implement a reload button for both the PhotoGalleryFragment and the PhotoMapFragment. The reload button should force the application to fetch photos from Flickr.

Preview Unavailable

Gallery.zip

Download

(https://canvas.vt.edu/files/23085557/download?download_frd=1&verifier=vvwm5uScQtSG8TDFGIfnHIeakcDwLtRUI7Owx7h4)

You are unable to submit to this assignment as your enrollment in this course has been concluded.