CS Android Programming: Trivia Game (network services)

The rules. Recall the rules for flipped classrooms: do the work alone or with a partner that is unique for the semester. You can find the github link in Canvas.

Only one student needs to submit this code, but both students may do so if they wish. This assignment should be submitted through github (sorry, partners can't share the repository). Include a README at the top directory level that contains both student's EIDs.

Everything that you need to do is marked in the code with this prefix: `// XXX`
There might be hints in the comments. If you do not see a XXX, you do not need to change the code and you should not change the code.

Background and web site. Here is where the magic really starts to happen inside our apps as they start to communicate with the world at large. We will build a trivia game that uses our layout and display skills to present questions fetched over the network.

Watching the video is pretty helpful for getting a handle on the dynamics of the assignment.

You can visit `https://opentdb.com/api_config.php` to better understand the web service we are interacting with. The video goes over what we are trying to get out of the service and how we will achieve that goal.

Your task. You will fill in some retrofit/Gson code that is going to make the network request and build your Kotlin objects from the returned JSON. I give you most of this code, but I want you to write key pieces. The structure is nearly identical to CatNet, so please read that code carefully to give you ideas.

Then we have some practice with ViewModels and Views. The view model is pretty simple. The View (MainFragment) does mostly UI stuff that you understand well at this point. I hope turning the background on a TextView to green doesn't freak anyone out. But the View does need to observe LiveData, so there is that to practice.

- **AndroidManifest.xml** All good.
- **content_main.xml** Let's start by putting in three fragments, with IDs `q1`, `q2`, and `q3`. We want them equal sized, without being hard coded. They are inside a constraint layout.

  Because these fragments are being used for navigation, they need to be navigation hosts and they need a navigation graph. We do this by including these lines.
  `android:name="androidx.navigation.fragment.NavHostFragment"`
  `app:navGraph="@navigation/nav`$_{g}$*raph*"

  You should take a look at the layout. The swipe refresh layout must have only a single child, which is why it is the root element. I tried to have a constraint layout as the root element and the swiperefreshlayout had a linear layout child, but for the life of me, I couldn't get it to work. I blew hours on this one little layout detail.

- ~~main_fragment.xml needs no modification.~~

- navigation/nav_graph.xml We have a very simple graph with a single fragment (MainFragment) and a global action to navigate to that fragment while passing it an integer parameter. Please add the global action.

- **api/Repository.kt** This is the bridge between the API and your application. In our case, that doesn't mean a huge amount of complex logic. But you will need to define a certain kind of function and notice the input parameters to the Repository class.

- **api/TriviaApi.kt** You need to provide the annotation and return type for the `getThree` function. The annotation let's retrofit know how to turn a call to this function into a network request. I give you the function prototype because it has the `@Query` annotation, which we did not see in CatNet. The function prototype let's your program know how it should be called to give and get the correct information to/from the network request. In our case, we are passing the difficulty level as a parameter to the network request. Everything else (like the amount and category) should be "hardcoded" into the URL. See CatNet to get you started with the `@GET` annotation.

  I configured some basic HTTP logging, which can really come in handy when debugging your network-connected apps.

  Finally, there is a data class defined to capture all of the fields in the response. Let's talk about that.

- **api/TriviaQuestion.kt** This file contains the model for the network request. It maps the names in the JSON (e.g., `@SerializedName("category")`) to declarations in the Kotlin data class (e.g., `val category: String`). One of these bindings is missing and you must provide it.

- **MainActivity.kt** Initialize the viewModel.

  `navigateToMainFragment` looks a bit different from things we have seen before. We want to navigate to the main fragment in three different navigation controllers, one per layout (with ids `q1`, `q2`, and `q3`). This function takes an id and an argument and navigates to the main fragment and passes that argument. You need to think about how to pass parameters to this function from `onCreate`, where it is called.

  We provide the live data observe method, but you fill in the contents. It is a one-liner.

  When it comes to managing the spinner, I got you.

- **MainFragment.kt** Initialize the view model. Note, we use the same view model here and in the activity. We provide the navigation arguments declaration.

  For `setClickListeners`, these handle the logic of checking if the user got the question correct and changes the background color to red or green. Which color goes with which answer? Let common sense be your guide.

  Note the function `fromHtml`. When the trivia game sends you strings, it uses an ugly HTML encoding that looks terrible if you just display it to the user (if there are any special characters like an apostrophe or a greater than sign). Call `fromHtml` on all of the question text to make sure it looks nice.

In `onViewCreated` you need to observe something in the ViewModel, but that part isn't written yet. You will observe the list of trivia questions. When it changes (via LiveData), you will do some game stuff here. Read the comments. This part has the most logic.

- **MainViewModel.kt** You need to add some variables, like an API object and a repository and you are going to want to export a list of questions as livedata to your view. Recall the trick of making MutableLiveData in the ViewModel that is "observed" as LiveData by the view.

  You should initialize the state of the view model by making a network request for questions.

  Recall that the function that makes the network request is doing IO. See CatNet for some help with the syntax.

Hints: Start by getting your viewModel to make a netwwork request. With the tracing we have, you should see the URL in the run tab. Make sure that the URL is correct by copying it and pasting into a browser.

Hints: Once you get things somewhat hooked up, you might find the network activity indicator from the swiperefreshlayout just stuck on your screen. That is a problem you should fix. Ask Mr. Google about it.