CSCI 448 – Lab 08A
Wednesday, March 08, 2023
LAB IS DUE BY **Friday, March 17, 2023, 11:59 PM**!!

Little Green Games has been getting positive reviews about the Samodelkin app, but users want to be able to expand the universe that characters exist in. In fact, a fan site has sprung up that creates characters specific to the Samodelkin world. The site is publicly available for anyone to access via the url: https://cs-courses.mines.edu/csci448/samodelkin/. Your team wishes to incorporate the remote character generator into the app.

Step 1 – Get Ready To Network

In order to do anything with the network, the application needs to notify the operating system of its intentions. We'll also add in some good UX to prevent errors relating to networking.

Part 1.I – Request Permission

In the manifest file, we need to list all the permissions our app requires. There will be two we use:

```
<manifest ...>
  <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
  <uses-permission android:name="android.permission.INTERNET"/>
  <application ...>
```

The first is necessary to check if we are connected to a network or not and the second is necessary to access the internet.

Part 1.II – Check If Currently Connected

We'll add ourselves another helper class to perform a common task – check if we're currently connected to the internet or not. In the `util` package, create a Kotlin object called `NetworkConnectionUtil`. This singleton object will contain a single function that checks if an active network is available. The implementation is provided below:

```
object NetworkConnectionUtil {
  fun isNetworkAvailableAndConnected(context: Context): Boolean {
    val cm = context.getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
    val activeNetwork = cm.activeNetwork
    return activeNetwork != null
            && (cm.getNetworkCapabilities(activeNetwork)
                   ?.hasCapability(NetworkCapabilities.NET_CAPABILITY_VALIDATED)
                 ?: false)
  }
}
```

We will use this to enable or disable the button that triggers the internet request as appropriate.

Part 1.III – Activate the Button!

`NewCharacterScreen` contains the currently disabled `Request Character From Web` button. We will now turn this button on. The button is called in both the landscape and portrait orientations, so we will need to abstract the button properties into a pair of variables.

Begin by adding a parameter to the `NewCharacterScreen` function called `apiButtonIsEnabled`. Update the two calls for the corresponding `NewCharacterButton` to set the `enabled` argument to this variable.

Likewise, add another parameter to the `NewCharacterScreen` function for a function object called `onRequestApiCharacter` that has no parameters and returns nothing (it should look like `onGenerateRandomCharacter`). On the corresponding `NewCharacterButton` again, set the `onClick` argument to this variable.

Part 1.IV – Enable The Button…Or Not

`NewCharacterScreenSpec` specifies how the `NewCharacterScreen` is created. That is where we need to specify the arguments for our two new parameters. Set `apiButtonIsEnabled` to the result of calling the function we created in Part 1.II.

```
apiButtonIsEnabled = NetworkConnectionUtil.isNetworkAvailableAndConnected(context)
```

For the time being, set `onRequestApiCharacter` to be an empty lambda that we will fill in later.

```
onRequestApiCharacter = { /* TODO in Step 3 */ }
```

Build, deploy, run, and check. Start with the device in airplane mode. Navigate to the new character screen and the button is disabled. Turn off airplane mode and ensure you have either WiFi or cellular data enabled. Return to the new character screen and the button should now be enabled!

Note: If you are on the new character screen and simply pull down the top window shade to change the device settings, despite the window shade appearing on top of our activity this does not pause our activity. You will need to navigate away from the new character screen to retrigger our connection check.

Step 2 – Request and Parse the Remote Character

Part 2.I – Add the Retrofit dependencies

The Retrofit library manages the entire process of making the network request and then parsing the JSON response into a POJO for us. When working with network data that uses JSON as the request and response data format, Retrofit is an excellent library to manage this process for us. Begin by adding the necessary dependencies to your `build.gradle (Module: app)` file.

```
implementation 'com.google.code.gson:gson:2.9.0'
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
```

Part 2.II – Markup the Response object

If you haven't already, in a browser view the response that https://cs-courses.mines.edu/csci448/samodelkin/ returns. The JSON structure almost matches our `SamodelkinCharacter` structure exactly, except for one property. On the `SamodelkinCharacter` class, we were already treating the `avatarAssetPath` member field uniquely for the database. We will need to make a similar mapping to the JSON form of the object. When serializing the object, denote the name to use.

```
@ColumnInfo(name = "avatar")
@SerializedName("avatar")
val avatarAssetPath: String? = null
```

Part 2.III – Create the REST Interface

Inside the `util` package, create a new package called `api`. We will place all the functionality to request and receive the remote responses in this package.

The first step is to create the interface used for the REST calls. In the `api` package, create an `interface` called `SamodelkinApiService`.

First add a companion object that contains a public constant called BASE_API_URL and set it to the host name of the remote endpoint.

```
companion object {
  const val BASE_API_URL = "https://cs-courses.mines.edu"
}
```

Next, add a function called getCharacter() that returns the Call object. This Call object will ultimately return a SamodelkinCharacter created from the corresponding response. We'll need to annotate the function with (1) the HTTP call method to use (2) the endpoint to access.

```
@GET("csci448/samodelkin/")
fun getCharacter(): Call<SamodelkinCharacter>
```

Be sure all imports are from the `retrofit2` package.

Part 2.IV – Fetch the Result

All of the logic to perform the request and process the response will be placed into a class called `SamodelkinFetchr`. Create this class in the `api` package.

Part 2.IV.A – Add Member Fields

Begin by setting up the `LOG_TAG` as we'll be logging the flow to aid in debugging.

Now add a private immutable data member for the `SamodelkinApiService` we just created – don't initialize it to anything just yet, we'll do that momentarily.

Also add a private immutable variable called `mCharacterState` that is a `MutableStateFlow` for a nullable `SamodelkinCharacter`. Initialize this state variable to be `null`. (Quick sanity check – how many degrees of mutability does this variable object have? It's just one!)

Then add a public immutable variable called `characterState` that is a read-only `StateFlow` for a nullable `SamodelkinCharacter`. The getter will return our private state variable `asStateFlow()`.

```
     get() = mCharacterState.asStateFlow()
```

Now in an `init` block, we can initialize the service. It is a multistep process, which is why we needed to defer initialization. First create a `Retrofit` object using the builder to specify the following build properties:

- The baseUrl is our API
- A GSON converter factory will be used to parse the JSON

```
val retrofit: Retrofit.Builder()
     .baseUrl(SamodelkinApiService.BASE_API_URL)
     .addConverterFactory(GsonConverterFactory.create())
     .build()
```

We're now able to assign a value to our service object by having Retrofit create the interface instance for us:

```
samodelkinApiService = retrofit.create(SamodelkinApiService::class.java)
```

Part 2.IV.B – Setup the Request

Add a public member function called `getCharacter()` that takes no parameters and returns nothing. The first step will be to create the request object from the service.

```
val samodelkinRequest = samodelkinApiService.getCharacter()
```

This request object is of the `Retrofit Call` class type. The `Call` class wraps the Android WorkManager and functions like the work request. Therefore, we need to `enqueue` our request to be scheduled to run. We need to give the WorkManager a callback to execute upon a response. This callback will handle a failed or successful response. We'll begin by logging which type of response we received.

```
samodelkinRequest.enqueue(object : Callback<SamodelkinCharacter> {
  override fun onFailure(call: Call<SamodelkinCharacter>, t: Throwable) {
    Log.e(LOG_TAG, "onFailure() called $t")
    // TODO - finish in Part 2.V.A
  }
  override fun onResponse(call: Call<SamodelkinCharacter>,
                          response: Response<SamodelkinCharacter>) {
    Log.d(LOG_TAG, "onResponse() called")
    // TODO - finish in Part 2.V.B
  }
}
```

At this point, we could hook up our button to this call and check the connectivity. But we're going to be optimistic that this will work and parse the response first then hook it up.

Part 2.V – Parse the Response

When we receive a response, we will update the character state. The View using this utility will be observing the state object and waiting to be notified when the response has been received.

Part 2.V.A – Handle a Failed Response

If we receive a failed response, we will update the character state to be `null`. That will signal there was no valid response received.

```
mCharacterState.update { null }
```

Part 2.V.B – Handle a Successful Response

While this is the more complex of the two scenarios, the nuanced parsing process is abstracted and handled by Retrofit for us. The `response` object received by the `onResponse()` method contains the already parsed JSON and deserialized POJO.

First, store the `body` of the `response` which corresponds to our deserialized `SamodelkinCharacter` POJO.

```
val responseCharacter = response.body()
```

Log this variable. It's possible this object could null. Setup an `if`/`else` block to check if this object is `null`. If it is `null`, then log that the character is null and then update the character state to be null (as was done in Part 2.V.A).

If it is not `null`, then we have the character to update! There are two pieces we need to massage before the character is fully ready to use:

First, the response only gives us the filename of the avatar to display – it does not give us the location. We will need to prepend the path to the filename.

Second, despite our `SamodelkinCharacter` not allowing the id to be null and specifying a default value for the field – the response character has a null id! *(The object is created using reflection not the constructor, so a different process is used to create the object and this results in the default value not being assigned.)*

We could create a new object and set every field of the constructor as appropriate. OR we can leverage the Kotlin `copy` method to clone the object and specify the fields to change on the clone. The copy method will be our solution to convert the received character into a form that works with our application.

```
val newCharacter = responseCharacter.copy(
    avatarAssetPath = "file:///android_asset/characters/${responseCharacter.avatarAssetPath}",
    id = UUID.randomUUID()
)
```

Log this `newCharacter` and then update the character state with it.

Step 3 – Trigger the Request and Display the Response

The final piece is to connect the utility to our Presentation layer in View. Everything here on out will take place in the `NewCharacterScreenSpec`.

Part 3.I – Get and Observe the **`SamodelkinFetchr`** State

Inside of the `Content()` method, after we've already setup the character state object that we used with the randomly generated characters we will store an instance of our `SamodelkinFetchr` class and remember it (this way if our screen gets recomposed we won't make a second new object, we'll use the original instance).

```
val characterState = remember { ... }
val samodelkinFetchr = remember { SamodelkinFetchr() }
```

The next step is to observe the fetchr's character state and collect it off the flow. Here is where we see an advantage of using StateFlow instead of just State. StateFlows are lifecycle aware. Since our View (i.e. the activity) can exist in different states, be killed and recreated, the flow will respect and respond to these state changes. If the activity is killed, then we'll stop collecting from the flow. If the activity is recreated, then we are able to collect from the prior flow that was already streaming. There is no new State object created, there is just the single hot flow emitting data. This is accomplished by collecting on a separate coroutine from our View. Whenever we collect from a StateFlow in the View, we will want to setup the collection like follows moving forward:

```
val apiCharacterState = samodelkinFetchr.characterState
  .collectAsStateWithLifecycle(context = coroutineScope.coroutineContext)
```

Wait, where did `coroutineScope` come from? We need to pass it down from `MainActivity`. Quick! Get the band aids to patch!

Part 3.I.A – Add **`CoroutineScope`** to the Interface

In `IScreenSpec`, add a parameter to the `Content` method for the coroutine scope.

```
coroutineScope: CoroutineScope
```

Part 3.I.B – Patch Each Concrete Screen

We've already setup `NewCharacterScreenSpec` to work with this new interface. However, on `ListScreenSpec` and `DetailScreenSpec`, likewise add the new parameter to the method.

Part 3.I.C – Patch the NavHost

The `SamodelkinNavHost` calls each screen's `Content` method. Add the corresponding parameter to the `SamodelkinNavHost` and then set the argument in the `screen.Content` call.

Part 3.I.D – Supply the Value

The `SamodelkinNavHost` gets called from `MainActivity`. After we've remembered our `navController`, likewise remember our `coroutineScope`.

```
val coroutineScope = rememberCoroutineScope()
```

We can now pass this object to the argument for our NavHost.

Part 3.II – Respond to Newly Collected Characters

We're going to take our first foray into composable side effects. Usually, we don't want the call of our composable function to change any state – just display the state. But sometimes, that is unavoidable. What we need to occur is for when we see one state change then we need to update the second state. What's occurring in our View?

- We are storing the state of the displayed character.
- We are listening to the state of the remote character.
- When we are notified of a new remote character, we need to transfer this to the displayed character.
- And thus, our change of state.

We can control the execution of these side effects and have them occur at specified times on a separate coroutine. The LaunchedEffect will be called once when state changes. It will observe a state value, when the value changes it will launch its associated coroutine.

```
LaunchedEffect(key1 = apiCharacterState.value) {
  // transfer updated state here
}
```

Now we can go through the state transfer process safely.

First, get a reference to the value of our `apiCharacterState`.

```
val apiCharacter = apiCharacterState.value
```

Log this object. If this object is not `null`, then log we have a valid character and set the `characterState` value to this object. If the object is `null`, then log we have a null object and set the `characterState` value to be a random character again. *(Note: We could also use this opportunity to display an error message to the user that the connection could not be made.)*

Part 3.III – Request a Remote Character!!

And now, the moment we've all been waiting for – the event to kick off this entire process and check if it worked. Inside the `onRequestApiCharacter` lambda, make the call to `samodelkinFetchr.getCharacter()`. This kicks off a sequence of events:

- Make the request object for the REST call
- Enqueue the request and wait for the response
- Receive the response

- Convert the JSON to a POJO
- Process the received POJO to a valid `SamodelkinCharacter` that works with our app
- Update the API character state
- Collect the new state
- Transfer the API state to the displayed state
- Recompose the View with the new character

To check that's all working: build, deploy, and run the app. Have the Logcat open to follow the flow. Press the request from web button and check out the expanded array of characters we can now generate!

Step 4 – Deploy Your App & Submit

When Lab08 is fully complete, you will submit a video of your working app to Canvas. Demonstrate the following actions inside the app:

- Begin with the device in airplane mode
- Go to the new character screen (the button is disabled)
- Return to the list screen
- Turn off airplane mode, ensure device is connected to network
- Go to the new character screen (the button is enabled)
- Press the request character from web button
- Press the request character from web button again
- Save the character

Then stop the recording. Save it as webm format, name the video `<username>_L08.webm`, and upload this file to Canvas Lab08.

With the new functionality completed, you sign off the third ticket and begin thinking about completing the Samodelkin app.

LAB IS DUE BY **Friday, March 17, 2023, 11:59 PM**!!