Lab 9 - Arduino programming and Android Retrofit

Part I - Android app

Android supports an extensive choice of data transfer and connection management options. Data can be transferred using Java Sockets, upon which we can make HTTP calls. REST APIs provided by remote Web services can be accessed using HTTP calls, and for that we use HTTP clients, such as HttpsURLConnection or OkHttp. However, converting data from these Web services (usually structured as JSON or XML) to Java objects that our Android app uses can be tedious if done manually. Libraries such as Retrofit and Volley make this process easier.

In this part of the lab, you will code an Android app that will connect to the Arduino system you will be designing in the second part of the lab assignment (next week) via an HTTP GET request, download the latest image available and display it on screen. Ultimately, you will also apply some basic image processing techniques to try to enhance the downloaded image.

This app will not be very complex, so you can start from scratch - create a new Android project with an empty activity.

Setting up Retrofit and OkHttp client

To use Retrofit you need to add the following to your module's dependencies in the build.gradle file:

```
implementation 'com.squareup.retrofit2:retrofit:2.8.1'

implementation 'com.squareup.retrofit2:converter-gson:2.6.2'
```

HTTP Client definition and instantiation

Create a new public interface called **HttpCalls**. In this interface we define the HTTP GET request as Kotlin function calls:

```
@GET("image.jpg")

fun downloadImage(): Call<ResponseBody?>?
```

Examine the above lines, make sure you understand what they mean. Next, create a Kotlin class **RetroClient**. This helper class will, for a given HTTP API interface, instantiate and build a Retrofit object. Go ahead and implement this class. When building the Retrofit object, the baseURL should point to the IP address and port of the Arduino board. Before you have the chance to build your own system, you can use the following URL "`http://212.235.189.130:8080/`", which is the IP and port address of the Arduino system which is already up and running somewhere on the FRI premises, for test purposes.

The **layout** of your app should allow for calling the `downloadImage()` method and display the received image file from the Arduino board in the app. The easiest way to do this is by using an **ImageView** and a **Button**. Go ahead and add these two to your layout. Next, you should in **MainActivity** implement the function that is called when clicking the button. This should make the call to the `downloadImage()` function using Retrofit:

```
val call: Call<ResponseBody?>? = RetroClient.instance?.api?.downloadImage()
```

```
call?.enqueue(object : Callback<ResponseBody?> {

override fun onResponse(call: Call<ResponseBody?>, response: Response<ResponseBody?>)
{
      assert(response.body() != null)
      val bitmap = BitmapFactory.decodeStream(response.body()!!.byteStream())
      if (bitmap != null) {
         binding.imageView.setImageBitmap(bitmap)
      } else {
         Toast.makeText(this@MainActivity, "Invalid SN", Toast.LENGTH_SHORT).show()
      }
   }

override fun onFailure(call: Call<ResponseBody?>, t: Throwable) {
    Toast.makeText(this@MainActivity, "Network Failed", Toast.LENGTH_SHORT).show()
}
})
```

Take a minute to understand how the above functions work. When the call is made, if successful, the onResponse function is called. Here, we want to extract the image stream from the response body and pass it to the imageView to be displayed. If the call was not successful, we will show an error message using a Snackbar.

When an Android app communicates with servers using a cleartext network traffic, such as HTTP, it could raise a risk of eavesdropping and tampering of content. Third parties can inject unauthorized data or leak information about the users[1]. That is why developers are encouraged to secure traffic in real-world apps (by using HTTPS, for example). Starting with Android 9 (API level 28), cleartext support is disabled by default. As such, you will have to add the following attribute to your application in the AndroidManifest.xml file:

```
android:usesCleartextTraffic="true"
```

Adding an alert dialog while downloading the image file

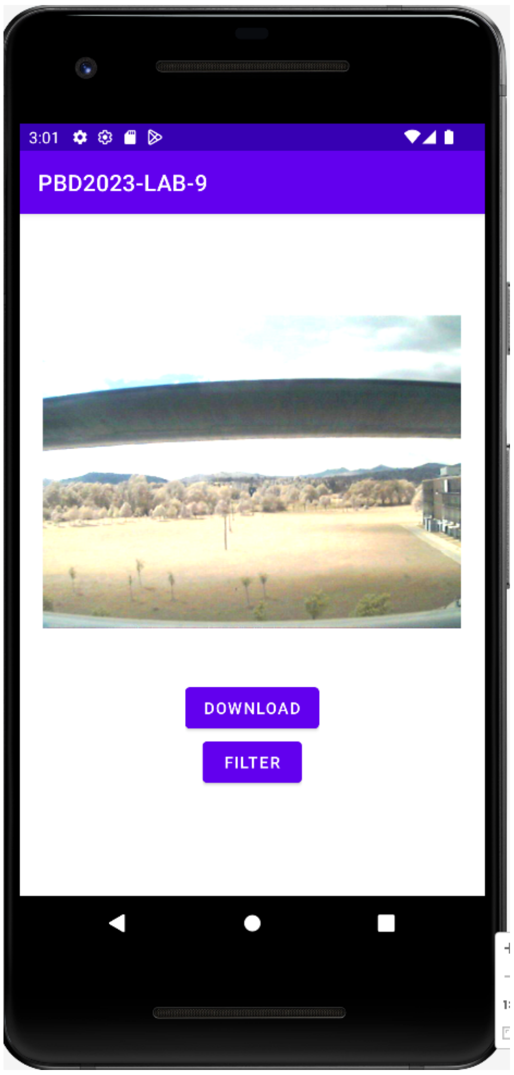
Now, build and run your app. Clicking the download button should, eventually, make the latest captured image from the Arduino system be downloaded and displayed in your Android app. You might notice it takes a longer than expected time for this to happen, after the moment you click Download. This is not surprising, since the Arduino is a low-end uC with a very slow speed of the serial communication between the uC and the other system components such as the uSD card adapter and Ethernet shield. However, you should make sure the user is informed that clicking Download actually does something and that it might take some time. Consequently, it would be nice to show an informative message on the screen after clicking the button and until the image is shown. You could do this by using an alertDialog - this should be set up and displayed before the call:

```
 // Set up alert dialog before call
        val alertDialogBuilder = AlertDialog.Builder(this)
            .setTitle("Status")
            .setMessage("Download in progress. Please wait...")
        val alertDialog = alertDialogBuilder.show()
 // show it
        alertDialog.show()
```

[1] https://medium.com/@son.rommer/fix-cleartext-traffic-error-in-android-9-pie-2f4e9e2235e6

Dismiss it using `alertDialog.dismiss()`, either when a response is received, or when failure has been detected. Test the app again and make sure the alert works as expected.

Enhancing the image



As you might have noticed, the image quality is not great. This is because of the limitations of the camera used for acquiring it (low resolution, lack of automatic image enhancement techniques, etc…). In the last part of this lab assignment, let's try to make the image look better. So far, you've been using ImageView objects to display images. Android has a "smarter" version of ImageView, called ImageFilterView[2]. This performs the same functions as a regular ImageView, but in addition allows for several basic image processing techniques for adjusting the saturation, brightness, warmth and contrast for the image. These can be dynamically adjusted by accessing the corresponding attributes of the ImageFilterView object. Let's try them out!

In the layout xml file, change the ImageView type to ImageFilterView and add a second button beneath the "Download" button. In the MainActivity, set a onClickListener to this button that will call method to do some basic image enhancement:

```
  if (binding.imageView.drawable != null) {
            binding.imageView.saturation = 2F
            binding.imageView.contrast = 2F
        } else {
          Snackbar.make(binding.root,   "No   image
downloaded", Snackbar.LENGTH_SHORT).show()
  }
```

In the above code, we are setting the ImageFilterView's saturation to 2, which means "hyper saturated" and its contrast settings to 2, corresponding to high contrast. Test the app again and see how this new feature works. Take some time to experiment with the other ImageFilterView attributes as well.

If you did not attend this lab assignment in-person at FRI you can commit your solution to a private repository named **PBD2023-LAB-9** in your Bitbucket account. The repository must contain the Android project. User **pbdfrita** must be added as a read only member of this repository. The code must be committed by Sunday (May 14th) 23:59.

Happy coding!

[2] https://developer.android.com/reference/androidx/constraintlayout/utils/widget/ImageFilterView