LAB 2: ASYNCHRONOUS TASKS

Guillaume HAMEL

EFREI PARIS J.-F. Lalande

Lab2: Asynchronous tasks

I - Github repositories

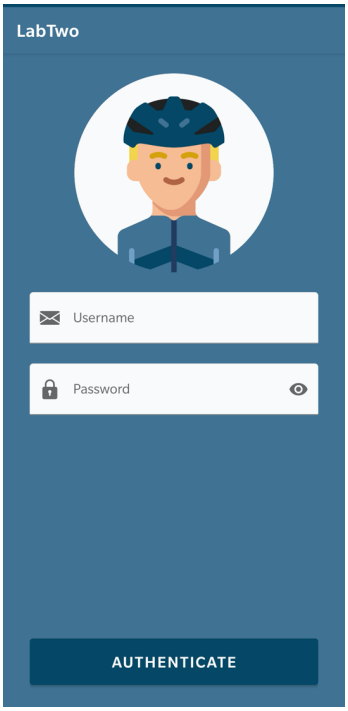
Here are the links to both repositories used to track code during the development process. Due to a small misunderstanding of the instructions, the first part (questions 1 to 10) wasn't tracked in a GitHub repository at first and were added later.

Part 1:
Part 2:

II - Architecture

a) Authentication

At first, I developed a small app to test the connection process to a remote server as well as a Login User Interface. As always in an Android Java app, the plain UI side and the Logic side are implemented separately.



I naturally decided to put some efforts on the UI side to make something clear, user friendly and beautiful, in order to allow me to practice my UI/UX skills during the process.

I started by setting up standards to use throughout all the application interface to ensure continuity in design. Once some dimensions (for margin, padding and text size) and colors stored in the appropriates resources files, I created this Login interface through xml.

I choose to use Material Design standards to easily produce a clean and coherent interface with very handful widgets such as those textfields with moving hint text and a very convenient show/hide password button.

I have also put a random image which conveniently matched my color scheme as a way to show the design potential without putting too much time in it.

Once the interface ready, I started programming the logic necessary to test that authentication functionality. The *OnCreate* function handle the naming of the accessed resources such as the two fields and the button, as well as setting up the *OnClickListener* to code the logic operating after a click on our authenticate button.

Considering calling a remote server can be a long task depending on many factors, we execute the call in a separate thread. This way, it allows the application to create the view and then add the remaining elements once the remote task is completed.

The *OnClickListener* will start by getting the text from both *TextFields* and prepare a statement to send to the remote server with the user input. We then get a response – hopefully a success, maybe a failure – that we process by printing in the Logcats and formatting a String to send. We read the JSON we got back and determine wether the request was successful or not. In the case of a success, we can update a TextView meant to show a successful connection.

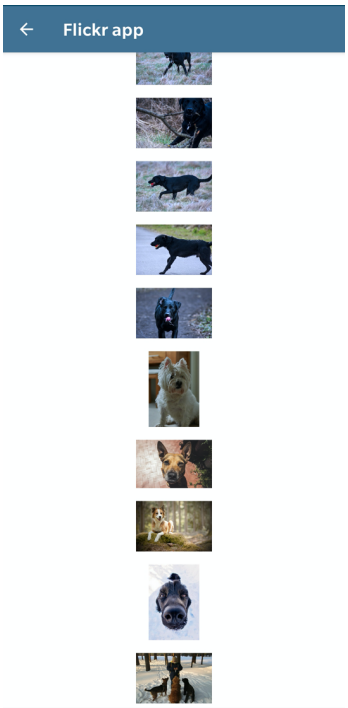
As we are on a separate thread, we are not able anymore to access to the views and change them. To achieve the requested behavior, we create a new *Runnable* to run on the *UIThread*.

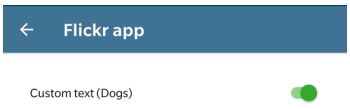
b) Flickr App

We will now be using the Flick API to fetch and retrieve images from the remote server. Once again, retrieving the data from the server is a tedious task sometimes requiring a lot of execution time. Plus, the images need to be downloaded and the relatively big amount of data will take a long time to fully be ready to appear on screen. We will use the same tricks as in the previous section and create a separated thread as well as some asynchronous tasks in order to have an efficient load time despite every connection issue we could encounter.

As always, the first part of this project is the production of a clear and user-friendly interface to quickly navigate through the different functionalities.

In order to fulfill all of the requested functionalities, I had to implement different view into the application. The first one allows the user to get an image and navigate to the over activities. The second one is a list of images retrieved from Flicker and the last one allow the user to select a custom word to use with the list activity aforementioned.



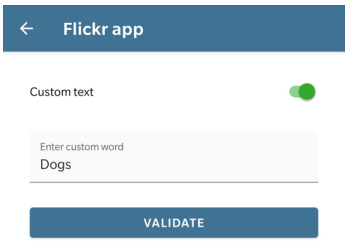


As for the logic involved, the first activity consists of two navigational buttons (to access the other activities) and a third button (at the top of the screen) to retrieve a single image. This work by requesting an image and using a converter to parse data bites to a Bitmap images to display on the screen. The request part is done in a separated thread to allow the *onCreate* function to operate in parallel of the download. This way, the image will be displayed only when ready and won't disturb the rest of the application.

The List Activity is a bit more complex. I had to use a *ListView* in order to conveniently display an array of images. As for the image in the previous activity, I needed to use some thread-related tricks to allow the execution of the whole activity to not be bothered by the ongoing downloading of the numerous pictures available.

This time, we had to process the JSONObject differently to store not one but an unknown number of picture related data. I iterated over the returned object to store each picture separately and conveniently access each of them. We then had to use some kind of asynchronous task in order to ease the downloading part by queuing the pictures url into an object meant to download them in order. Once downloaded, another element took care of adding them to the *ListView* to display them as soon as they were ready.

As for the technique used to display all the images in the interface, the ListView worked in pair with a custom *BaseAdapter* meant to manage the different data fed to him. For each and every picture passed to the adapter, it inflate a layout meant as a component and update the correct elements, for instance the source of the *ImageView*.



Last activity is a preference interface with a single item allowing the user to input a custom word and activate the function. As a default, the List Activity will fetch and retrieve tree pictures, but once this option activated using the switch button, a *TextField* will prompt the user to choose a custom word which will be used as a parameter for the flickr query.

Once the new word inputted, just hit validate and the word will be stored as a shared preference for later use. You can also disable the custom word option to use the default tree parameter.

III – Asynchronous tasks

As stated before, many of our tasks take a certain amount of time to complete and we cannot let the user wait with a white screen as everything is setting up. To cope with this issue, I used asynchronous tasks and thread to allow the application to quickly populate views and display complex elements as soon as they are available without restraining the use of the application in any way.

The first asynchronous task purpose is to get a single image from the remote server. It works in three phases. Firstly, the *doInBackground* method fetch and retrieve an *JSONObject* from the server. Once the execution completed, the *onPostExecute* method is executed and take that *JSONObject* as an argument to process it. This method gets the needed information from the object and put them in the view in the right element. It also launches an second asynchronous tasks meant to download the bitmap image, which is another long process.

As for the List Activity, it is a bit more complex as we need to do the actions an undetermined amount of time. The *AsyncFlickrJSONDataForList* java class purpose is once again to retrieve a *JSONObject* redirecting towards several pictures. The *onPostExecute* method will then add to the custom adapter all the media links and trigger an update of the view.

The adapter will then use a dedicated library to queue all the Bitmap images and download them separately and update the *ListView* subsequently every time a download complete.

This way, when the List Activity is called to be displayed on screen, it will be quickly constructed and shown empty, then images will start to appear one by one as they are downloaded. This happen so fast the user won't necessarily see it but it is indeed a good boost to the fluidity of the loading time.