Intro:

The following paper concerns our implementation of the Hoply social media app. From creating a user to commenting on your friend's posts, and even anyone else's, this app has everything, and we're going to be sharing the creation of this app from how we use the software, to the design process, to how we made it all come together.

Getting past the learning curve of making an app from scratch, the purpose of this project is to implement a remote database we've been provided with, which is written in the JSON programming language, and stored on a web server. We must fetch this data and store it in a local SQLite database, and then interact with that database using specific libraries created for this purpose. The three mandatory operations we must support are creating a user, submitting a post, and posting a comment, but those aren't much fun if we can't see the existing content of the database. This means we should also be querying all posts and comments, so we can show them to the user. As we're a two person group, we also had to implement one fun or advanced feature, and we've chosen to allow the user to update their username.

Even though we had very free range on which tools we could use, we went with the recommended way, using Android Studio with integration of the Android Room and Android Retrofit libraries.

Software and language:

Coming from writing simple Java classes to developing an entire app, we have to get familiar with Android Studio, and all the features it provides. Kotlin is the language recommended by Google for Android Studio, as it's been engineered specifically for use in this software. However, as we're used to writing in Java, an equally Turing-complete language, we will be writing our app in that instead.

We can't make an app using exclusively Java, as it needs to not only work, but also look nice. XML is a great language for doing the layout of the app, and it's the standard for Android Studio, so we're using it for our app.

To layout the app we use Android Studio widgets. Android Studio offers many different kinds of widgets ranging from a simple text view all the way up to Google Maps integration. Our app is going to be a proof of concept, so we're going relatively simple. All our fragments are based on a ConstraintLayout, which allows us to constrain widgets to each other, and adjust bias until we think it looks nice. The reason we do this instead of specifying exactly where we want everything to be is because everyone's device isn't necessarily the same aspect ratio. Using a ConstraintLayout, every device adjusts the position of widgets in relation to each other, so it looks nice on every device. Inside this constraint layout, we place TextView widgets to display text, and interact with the user through EditText windows and Buttons. To add a widget to the app we add it in XML, give it an id, and define it in a class as such:

```
Button postBtn = view.findViewById(R.id.PostBtn);
```

We call the findViewById() method on an object of type View, which is the general rectangular white box that is the base UI of any fragment. As parameter we use the aforementioned id we provided in our XML file.
If we want a button to do something, we use the provided method setOnClickListener, which lets us call the abstract method onClick, which defines what we want to happen when the button is pressed.

```
postBtn.setOnClickListener((view1) -> {
```

In this case we used a lambda instead of writing out the entire onClick method, as this is more concise. In the body of the method, we can, as an example, receive a string from the user using the same technique as we did with the button:

```
EditText contentEdit = view.findViewById(R.id.commentText);
String content = contentEdit.getText().toString();
```

Here we initialize the EditText widget, call the getText() method, and cast it to a string.

We also have to learn to navigate, and for this we can either write navigation access in XML, or we can use the nav_graph feature in Android Studio, where we draw arrows between the fragments we wish to navigate between, and it will automatically create action methods to navigate between fragments. We call it using an object of the type NavHostFragment:

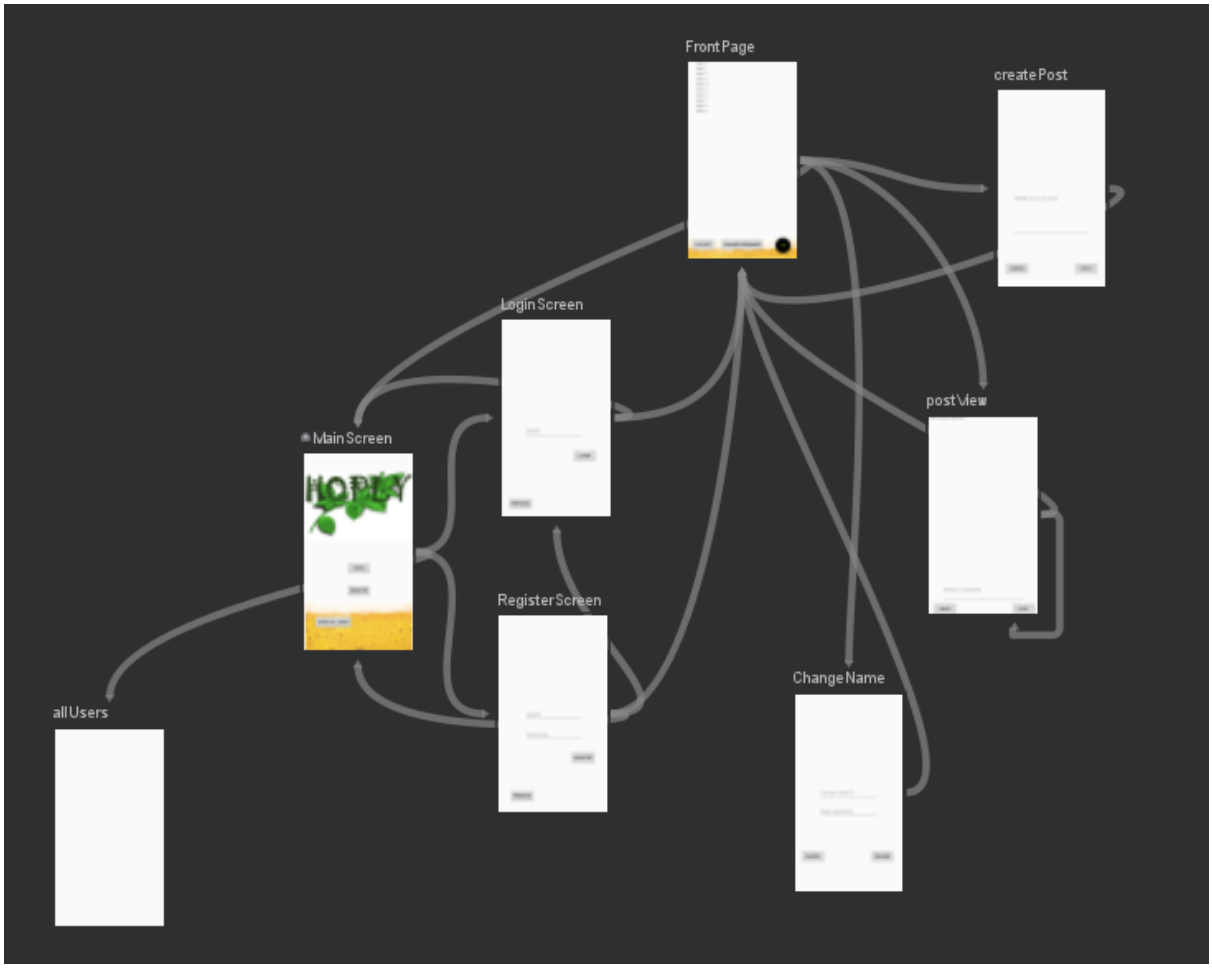
```
NavHostFragment.findNavController( fragment: PostView.this)
        .navigate(R.id.action_postView_to_FrontPage);
```

Here we use findNavController(context).navigate(action) to navigate between two fragments.

These interactions with the app can happen in two different methods: onCreateView() and onViewCreated(). The former method is called upon creation of the fragment, and the latter is called afterward, while the fragment is running. We want all layout components to load up on creation, but save the heavy, multi-threaded operations to run afterwards. We will see the importance of this later.

Now let’s go through our app, and talk about the design choices we made:

Layout and design:



The app is made up of 8 distinct fragments with each their own function. Each fragment (apart from allUsers, which we will address in our testing and debugging segment) has the option to go back to the previous fragment, which is why we've added back arrows in the nav graph. Let's go through the different fragments, talk about their function, and how it all comes together.

It all starts at the main screen, where the user can either choose to register or login as an existing user. The register screen inserts a new user into the database. It does not grant you permission to view the app. This is instead done by navigating the user to the login screen, which they can also get to directly from the MainScreen. Here we check if the user exists in the database, and if so, set the static variable thisUser as that user. This variable is important, as it is our app's way of knowing which user is currently logged into the app.

Upon arrival to the front page, the user is greeted to their username at the top of the page, letting them know which user they're logged in as. This is followed by a scroll view where we load every post in the JSON database from newest to oldest, meaning that the most recent post is at the top. For every post we print a TextView of the username, the content and the stamp, followed by a button that says "view", which we will talk about in a second. Some of the other groups implemented images as their advanced feature, which our app does not support, which means those pictures are shown as long strings on our app. To keep these from taking up the entire front page, we limit the number of characters shown for each posts content to 100. Lastly, on the front page, we have three buttons: logout,

which sets thisUser to null and navigates back to the main screen, update username, which is our bonus feature, and a floating action button which let's the user submit a post to the database. Pressing the latter navigates you to the createPost fragment, which allows the user to submit a post to the database, and navigates back to the front page when they're done.

Upon pressing the "view" button on the front page, the user is navigated to the postView fragment, but not before we store the current post as a static variable, so postView knows which post to load up. We print the post in a scroll view to support those really long posts that we limited on the front page. Below that we have yet another scroll view containing all the comments on that post, followed by an EditText window that allows the user to submit a comment of their own. There's a reason why postView navigates back to itself in the nav graph. Whenever someone posts a comment, the page needs to refresh with said comment, and this is a nice way to do it, as a page reloads every time it is navigated to.

Now that we've addressed the UI, we now need to talk about how and why we implemented our local- as well as remote database.

Local database and Android Room:

The reason we have a local database, instead of just retrieving data from the remote database everytime we need it, is so that even if the device loses access to the internet for any reason, that data will still be stored on the device for offline access. If we didn't store information in this middle step, we could be out of sync with the database in a case like this. The best thing to do in a situation where the device loses internet access is therefore to store the information as a request on their device, and synchronize when they gain access to the internet again.

We're going to define our local database in SQLite, as that allows us to use the Android Room library. This is a library that helps us interact with an SQLite database in Java. We implement the Android Room by using three different types of classes:

An interface that defines all queries we wish to implement as methods,

An abstract class that accesses the database by calling an abstract method that gets the database instance,

A class for each relation in the database, which defines all attributes of each relation as attributes of a Java object.

The Android Room DAO interface lets us implement all CRUD-operations on the database with the annotations @Insert, @Query, @Update, and @Delete, and we are going to be using all but @Delete.

In the table classes, we use the annotation @Entity to define the name of the table and @ColumnInfo to define the name of the attributes, so they match the provided JSON database when we connect them later. Underneath this annotation, we define the attribute with its appropriate name and type in Java. We also define the primary key in each

relation, as required by the Android Room library. However, we omit foreign keys, as Room doesn't require those for every relation, and due to the fact that the JSON database already has foreign key declarations in the first place, which means we can deal with update anomalies in Retrofit later, and it will still update correctly in the local database.

Now that we have our local SQLite database set up for use, we can begin synchronizing it with the provided online database, using Android Retrofit.

Remote database access with Android Retrofit:

Much like Android Room converts SQLite into a Java interface, the Retrofit library helps us convert the JSON API into a Java interface. It still supports the CRUD-operations, but in the form of @POST, @GET, @PUT/PATCH @DELETE, and some others. @PUT replaces the existing tuple with a similar one, while @PATCH merely updates the existing tuple.

```
// Creates a static instance of the remote database, so it can be accessed from other fragments.
Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("http://caracal.imada.sdu.dk/app2020/")
        .addConverterFactory(GsonConverterFactory.create())
        .build();
remoteDB = retrofit.create(RemoteDB.class);
```

We initialize the Retrofit database by providing a URL hosting a database, and adding a converter factory that uses Gson to convert the JSON code into a Java instance.

Much like Room, we now create a new interface that defines all queries using the aforementioned operations, but with a @Header annotation containing the bearer token provided in the slides, which grants us access to the database. In all operations but fetches, we also provide @Field's defining the names of the attributes we wish to insert, and add the annotation @FormUrlEncoded to denote that the request body should use URL encoding.

We store all queried data in objects of the type Call. This means we're not actually storing the data in our app yet, as we're merely storing requests to fetch the information. These requests need to be executed and stored before we can actually access and manipulate the data. We can execute these Calls synchronously with execute(), or asynchronously with enqueue(). As we want all database interaction to steer away from the main thread, and execute every query in it's own thread, we want to use the enqueue() method with a callback running on the current thread. If we executed it on the main thread and something went wrong, the entire app could crash. Having an auxiliary thread fail is much less inconsequential.
Calling the enqueue() method requires us to implement two nested methods: onResponse() and onFailure(). If the call fails, we show an error message. If there is a response but it returns unsuccessfully, it shows another error message. If the response has a body, we know that we have successfully downloaded data from the JSON, and can now interact with the contents.

Managing threads with a ForkJoinPool

We decided, that the best way of fetching this content on separate threads would be by using a ForkJoinPool from the Java concurrency library. This object is used to execute more than one thread at the same time. Forking and Joining means splitting up a task into smaller subtasks, executing every task in its own thread, and joining the results back together to create a full result. The sum of all threads is referred to as a “thread pool”, hence the name ForkJoinPool.

With some of our operations we need to return something from the thread. One example is when we need to get all the posts for the front page, and to do this we use FutureTasks. A FutureTask is an operation that is executed on a thread, but when you call the get() method on the operation, it waits for the thread to finish and then returns that result to the FutureTask. This makes sure that we get all the data we want from our threads, without the risk of losing data because the thread didn’t finish.

For our methods that insert data into the JSON database, we don’t need to use a FutureTask, as we don’t need to retrieve anything from the thread. For these operations we just add them to the ForkJoinPool to be handled with the other threads.

Connecting databases in the MainActivity:

The MainActivity class contains all the “master” code we wish to run for the entire app, not just a stand alone fragment. This means it’s a great place for us to initialize our databases, and synchronize them on startup. We declare the databases as static objects, which makes it possible to access the database from the fragments, which is very important.

If we were to create an instance of the database in each fragment, we would have a chance to have different data across our fragments. In our app it is important that the posts, comments, and users on our front page are the same when accessed from other fragments, like PostView and CreatePost.

Features

We have 3 basic features, which are creating a user, posting a post and posting a comment on every post. These methods all use the Retrofit method @POST, which simply adds it to the remote database. Since we use a local database with our remote database, we first add the entity to the local database and then to our remote database. This is because we want the user to view them as soon as possible.

As our fun/advanced feature, we wanted to do something that interacts with the database in a way that is different than our other features. We decided to make a feature that allows the user to update their username. From the front page we have a button that navigates to the ChangeName fragment. Here the user types their UserID as well as the name they want to update to. We verify that this UserID is equal to that of the logged in user, if so we update the username.

For this we use the Retrofit method called @PATCH. We use @PATCH instead of @PUT here, because the only attribute we update is the name of the user. Therefore there is no reason to update the entire entity with a @PUT method. As we only insert data and don't retrieve any, our enqueue method have no body besides toasting error codes in case the name isn't updated correctly.

Code choices:

Post/User/Comment:
These classes represent the three relations in the database. They give these relations primary keys and attributes in a similar way to that of SQL. Each attribute gets a variable and a name.

DAO/LocalDatabase/RemoteDB
DAO is an interface used by Android Room and it defines queries for our java program to execute on our local database. As an example, our getAllUsers() method in DAO is equivalent to calling the query: "SELECT * FROM users" in SQL. This way, every time we call the method getAllUsers() it calls that query on our database and returns the result as a list of users, which we then can use in our java program.

LocalDatabase is our implementation of the abstract class, which is used by Android Room. When we create an instance of the class, we can manipulate it in Java to get, insert, update and delete data. This is where we store all the posts, comments and users of our app, so we don't have to make calls over the internet to the remote database every time we want data from the database.

RemoteDB is an interface used by Android Retrofit, and it is very similar to the DAO interface used by Android Room. The difference is that this interface defines all of the queries we want to make on the remote database instead of the local database. This means every time we want to query the remote database, we need to define a method for that query in this interface.

Testing / debugging:

To test our connection to the databases, both local and remote, we chose to add a fragment that displays the content of one of the relations. We chose to print all users of the database, as we implemented users in our app before posts and comments, as users have no foreign key dependencies. If the users are all displayed on the allUsers fragment, that

means we've successfully queried information from the database. We can access this fragment from the MainScreen, but we have to be careful to not ever ship an app with a feature like this, as it should be impossible for anyone to see each others user_id. However, as this is only a school project, we simply comment out navigation to this fragment, and toast "Feature disabled" to anyone trying to access the feature.

Connecting to the local database using Android Room was relatively straight forward, but we still had to think about the database design a little, as constraints have to be defined by us. On insert declarations, we have to handle onConflict situations. Where we, in SQL, would normally type "ON DELETE <ACTION>", in Java we used the Room method OnConflictStrategy.IGNORE. This means that if the existing entity already exists we ignore it, as to not overwrite an existing user.

Using the Retrofit library was less straight forward. Many sources recommend adding @ForeignKey declarations. However, after testing this with the comments relation, since this relation has multiple foreign keys, we ran into some issues. Every time we were to run our app we kept getting the same SQL error, saying we were violating foreign key declarations. In the end we realized we wouldn't even need to use them, as they were already declared in the JSON database, which means we didn't have to worry about anomalies.

When you create a fragment in Android Studio, it gives you two methods that are run when the fragment is shown, the first one is onCreateView. This method is called instantly once the fragment is navigated to, and executes the XML view, so the user can see the layout of the fragment. The other method is onViewCreated, which is executed after the view has been created. This method is normally used for methods that have a long execution time, because if you were to run slow algorithms in the onCreateView, layout would only be shown once the slow operations have finished, which is not ideal, as the user would be staring at a blank screen for a while, waiting for all contents to load before they can even use the app. As an example, in our FrontPage fragment we want to show a feed of every post in the app. Here we reserve the onCreateView method for button creation and layout, and move creating the feed of posts into the onViewCreated method. We do this because the algorithm for printing the posts is slow.

We also had trouble with our Update Username feature when we tried to update it in the remote database. At first we were using @PUT, but then we realized that we should be using @PATCH, which was the better option in our case. As we were only updating one attribute in the user, because @PUT might also lead to anomalies. We also couldn't get the URL right. We kept getting code 404's testing the feature, which is an error that occurs when the page returns "not found". That meant we didn't have the right URL. We fixed this by adding the method updateUser() in the ChangeName fragment, which calls updateUsername as we would do normally, but adding "eq." to the new name in order to complete the URL for that username. Now the feature works.

In order to know whether your code will hold up to users abusing the interface, you have to abuse it yourself first. For a long time our app was working, until we started going crazy with the shared database. We noticed that the database didn't require for a post to have a userID, which would make our app crash every time we loaded the front page, if the

database were to contain a post without a userID. We fixed this by simply checking for a userID everytime we add a post to the local database. This wasn't the only flaw in our database. SQLite has a limit to the size of the content, and some posts exceeded that limit. Therefore, we had to omit posts that exceeded this limit.

Things to improve:

There were quite a few features we wanted to implement into our app, but not all the features made it into the final product. We decided that if we couldn't reliably implement these functionalities without either making our app slow or unstable, we should leave them out.

We chose to synchronise with the remote database only on startup, but this was not always the ideal thing to do. Originally we wanted to synchronise every few seconds, which would make the app way more practical for a social media platform.
After testing this we found out that with the current way we synchronise with the database, we would risk adding the same posts multiple times, which would flood the app with the same posts, and make it almost unusable.
Therefore we decided to only update the app on startup, because this decreases the chance of adding multiple entries of the same entity.

Deletion seems like a straight forward feature to implement, but this feature gave us a lot of problems. Sometimes it would delete the wrong entity and sometimes it would make the app crash, and after a long time of testing we found that our app was more stable without this feature.

When creating our app we found out that many of the features we wanted to implement were almost impossible with the given database. The given database only has very few attributes, so implementing something like profile pictures and user specific messages wouldn't be possible. If we were able to create our own attributes in the remote database, we would be able to add a bunch more features, that would make the user experience way more convenient. This could also include passwords if we wanted, but we don't think that would be a good idea either as we're not experts in hashing algorithms just yet.

The design of our app is very primitive and not very complex. This is something we decided to do on purpose from the beginning. The priority of this project is to communicate with the databases both local and remote, which we have achieved. Therefore we have not spent a lot of time on the design and layout of the app. The only thing it does is give the user a couple of options and displays the data of the database. For a future project spending more time on the layout would improve on the user experience quite a bit, which would provide the feel of a more professional and complete app.

One thing we could have done to improve the design a bit, is use a ListView. In this app we're only using a for-each loop to print a new TextView followed by a new Button for each post. An optimized way to do it, would be to use a ListView. Here we create a custom view as a template for each post, and make an adapter class which translates a post in the database into an object of this layout, and prints them in a scroll view. In reality a

RecyclerView would probably be the better option, as it is an improved but similar version of the ListView widget, which saves space by recycling used cells. However, we found this functionality would distract from the purpose of the app, so we omitted it.

Conclusion:

Now that we've reached the end of the paper, we can conclude whether or not we've done what we set out to do from the beginning. We made a database for us to store the online data locally, and then printed the contents on each their respective fragments. We implemented the three basic functionalities:

From the register screen we insert a user into the database, unless the user already exists, in which case we toast the user to use another user_id.
From the FrontPage we navigate to the CreatePost fragment, where the user can insert a post into the database.
They can also choose to click "view" on any post, and navigate to a fragment where they can insert and view comments.

Even though we had a few issues, we also managed to implement the feature for the user to update their username. With the inclusion of this advanced/fun feature, we've met the minimum requirements for this assignment, but the learning hasn't stopped there. We also learned how to build an entire app from scratch, which turned out to be easier said than done. In the future, we will be able to use this experience to create even better apps, with better UI's and original functionalities.