CMPS 312 Mobile App Development
Lab 12 – Firestore, Firebase Storage & Authentication Background Work with WorkManager

---

Objective

In this Lab you will practice how to:

- Use Firebase Authentication and security rules to secure Firestore database
- Create a Cloud Storage bucket to upload and download files
- Select a photo from the Gallery and upload it to Firebase Storage
- Take a photo using the Camera and upload it to Firebase Storage
- Use Glide library to display images from Web Urls and Uris in an Image composable.
- Sign-in using a custom Login screen using Firebase Authentication service
- Sign-up to create a user on Firebase Authentication service using Email/Password and store further user details on Firestore
- Upload set of images to Firebase Storage using Work Manager
- Create Background Work with WorkManager and support both asynchronous one-off and periodic tasks

Preparation

1. Sync the Lab GitHub repo and copy the **Lab 12-Firebase** folder into your repository.
2. Open **TodoList** project in Android Studio. Add the following dependencies to the build.gradle:

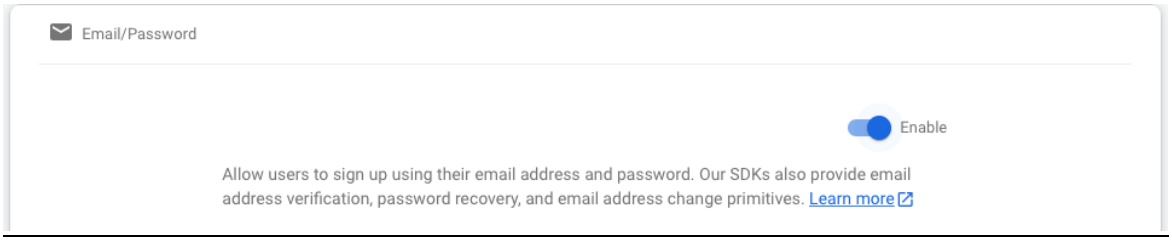
```
// Firebase Authentication
implementation 'com.google.android.gms:play-services-auth:19.2.0'

// Firebase storage
implementation 'com.google.firebase:firebase-storage-ktx:20.3.0'

// Used to display images from Web Urls
implementation("com.github.bumptech.glide:compose:1.0.0-beta01")
```

PART A: Firebase Authentication

In this lab, you will extend the Todo app implementation you created in Lab 11 and add the authentication part. Also, you will allow the user to upload and download files from firebase Storage. The rest of the app functionality will be unchanged and will allow the user to the same get, add, update delete projects and to-dos.

First configure the Firebase project you have created in the previous lab to allow only authenticated users to access the data.

1. Enable authentication using Email/Password. Then create an account to be used for testing.

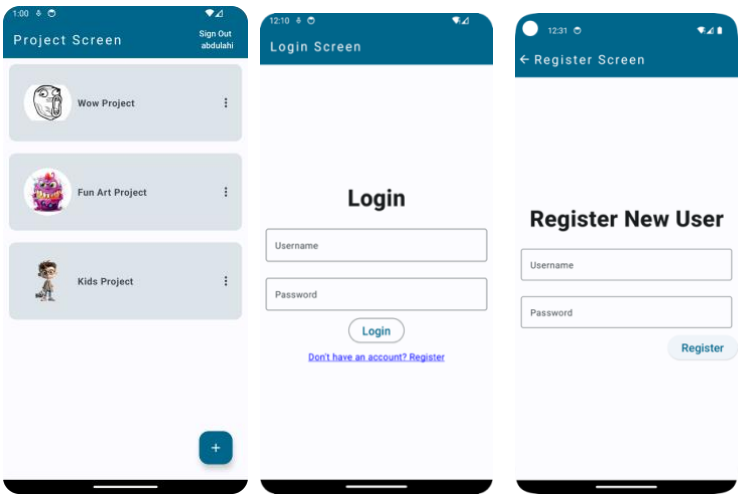


2. Open the Cloud Firestore and add the following security rule to restrict accessing the database to only authenticated users.

```
rules_version = '2';

service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow create: if request.auth != null;
      allow read, update, delete: if request.auth != null && resource.data.userId == request.auth.uid;
    }
  }
}
```

3. Create a login and registration screens enabling users to sign in and sign up through Firebase Auth. The application should prompt users to log in before accessing its features. Upon a successful login, the app should transition to the main screen. If the user is not registered, provide the option to sign up and complete the registration process.

4. Once Logged In in the display on the top app bar the name of the currently login user and allow the user to signout.



5. Inside the **SignInViewModel** implement the following methods,

```
fun registerUser(email: String, password: String)
fun signIn(email: String, password: String)
fun signOut()
```

6. Modify the **observeProject** function you created last week inside the **TodoListRep** to get projects using the logged-in user

```
fun observeProjects(): Flow<List<Project>> = callbackFlow {
    val snapShotListener = projectCollectionRef
        .whereEqualTo("userId", FirebaseAuth.getInstance().currentUser?.uid)
        .addSnapshotListener { values, err ->
            if (err != null)
                return@addSnapshotListener
            //parse the values
            val projects = values!!.toObjects(Project::class.java)
            trySend(projects)
        }
    awaitClose { snapShotListener.remove() }
}
```

7. Test your implementation.

PART B: Cloud Firebase Storage

In this section, we allow the user to select an image either from the gallery or take a picture using the camera then upload the image to firebase storage. You will also learn how to display images from Firebase Storage using Glide library.

1. When creating a project provide the ability for the user select an image either from the gallery or take a picture using the camera then upload the image to firebase storage.
2. Create a function called **uploadImage** inside the TodoListRepo that takes Photo URI and upload it to the firebase storage. Then the function should return the Url of uploaded photo.

```
suspend fun uploadPhoto(photoUri: Uri): String {
    var timeStamp = SimpleDateFormat("yyyymmdd_HHmmss").format(Date())
    var imageName = "Image_$timeStamp.png"

    var storageReference = FirebaseStorage.getInstance()
        .reference.child("images").child(imageName)
    storageReference.putFile(photoUri).await()
    return storageReference.downloadUrl.await().toString()
}
```

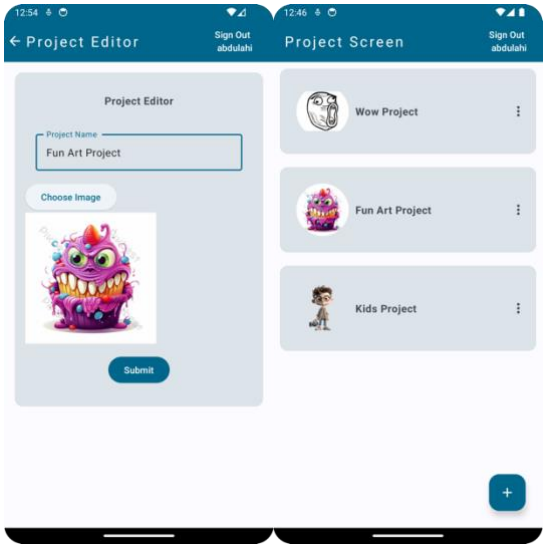
3. Modify the **addProject** function inside the ViewModel to call this function and pass the photo URI

```
suspend fun addProject(project: Project, imageUri: Uri?) {
    if (imageUri != null)
        project.imageURL = uploadPhoto(imageUri)
    project.userId = FirebaseAuth.getInstance().currentUser?.uid.toString()
    projectCollectionRef.add(project)
}
```

4. When the user presses on submit button, then call the **addProject** function inside the project view model and pass the **newProject** object and the **ImageUri**.

   ```
   projectViewModel.addProject(newProject, photoUri)
   ```

5. Change to app to display project images downloaded from Firebase storage using Glide library.
6. Test your implementation



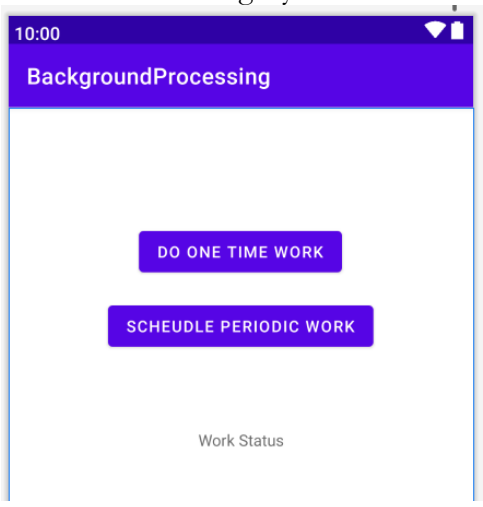
PART C: Background Processing using Work Manager

In this section, you will create a simple application that schedules work in the background. You will also learn how to get the status of running work.

1. Create a new project and call it Background Processing
2. Add the following dependency

   ```
   implementation "androidx.work:work-runtime-ktx:2.7.0"
   ```

3. Create the following layout



4. Create a worker class and name it **MyWorker** that implements **CoroutineWorker**
5. Override the doWork() function

6. Create another method called aLongRunningTask

```
suspend fun aLongRunningTask() {
    Log.d( tag: "TAG",  msg: "aLongRunningTask started executig ")
    delay( timeMillis: 1000 * 3)
    Log.d( tag: "TAG",  msg: "aLongRunningTask finished executing ")
}
```

7. Call this method insdie the doWork function

```
override suspend fun doWork(): Result = coroutineScope { this: CoroutineScope
    aLongRunningTask()
    Result.success() ^coroutineScope
}
```

8. Create a onetime request object inside the MainActivity and enqueue it when the One Time Button is clicked

9. Create a periodic request and enqueue it when the schedule periodic button is clicked

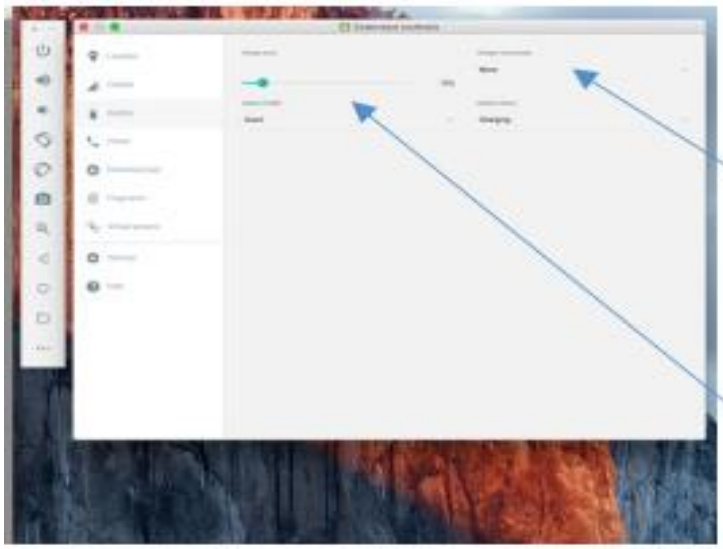
```
val periodicRequest : PeriodicWorkRequest
        = PeriodicWorkRequestBuilder<MyWorker>( repeatInterval: 15, TimeUnit.DAYS )
    .build()
```

10. Add the following listener to show the status of the work

```
WorkManager.getInstance( context: this)
    .getWorkInfoByIdLiveData(oneTimeRequest.id)
    .observe( owner: this) { it: WorkInfo!
        workStatusTv.append("${it.state.name}\n\n")
    }
```

11. Test your implementation.
12. Add the following constraints and test your code again by modifying the emulator properties

```
val constraint : Constraints  = Constraints.Builder()
    .setRequiresBatteryNotLow(true)
    .setRequiredNetworkType(NetworkType.CONNECTED).build()

val periodicRequest : PeriodicWorkRequest
        = PeriodicWorkRequestBuilder<MyWorker>( repeatInterval: 15, TimeUnit.DAYS )
    .setConstraints(constraint)
    .build()

val oneTimeRequest : OneTimeWorkRequest  = OneTimeWorkRequestBuilder<MyWorker>()
    .setConstraints(constraint)
    .build()
```



Make it
“NONE”

Use the Slider to
make the battery
LOW or OKAY