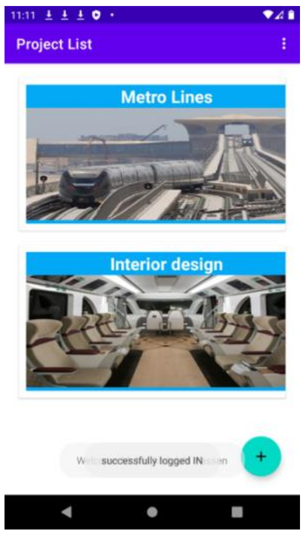
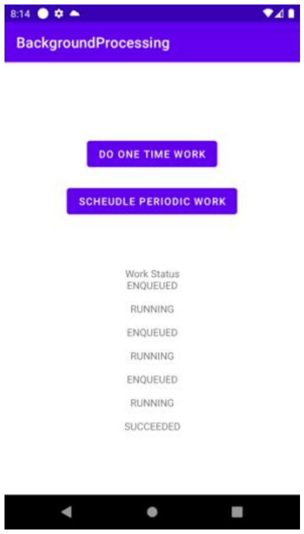
CMPS 312 Mobile App Development
Lab 12 – Cloud Firestore Authentication, Storage & Background Work with WorkManager

Objective

In this Lab you will practice how to:

- Use Firebase Authentication and security rules to secure Firestore data and application
- Create a Cloud Storage bucket to upload and download files
- Create Background Work with WorkManager and support both asynchronous one-off and periodic tasks



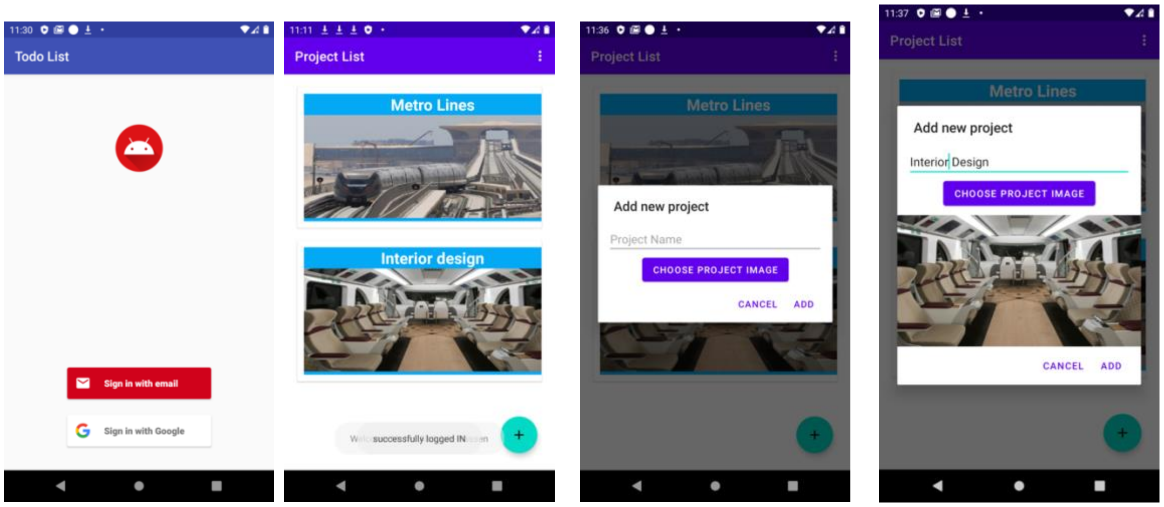


Preparation

1. Sync the Lab GitHub repo and copy the **Lab 12-Cloud Firestore and Background Processing** folder into your repository.
2. Import the **TodoList** project into Android Studio. You will probably see some compilation errors or maybe warning messages. We'll correct this in the next sections.

PART B: Cloud Firebase Storage

In this section, we add the images for the project and upload them to firebase storage



1. Add the following dependencies to use Firebase Storage and to download and display images inside your adapter

   // Firebase storage

   implementation 'com.google.firebase:firebase-storage-ktx:19.2.0'

   // Glide image download library

   implementation 'com.github.bumptech.glide:glide:4.11.0'

   kapt 'com.github.bumptech.glide:compiler:4.11.0'

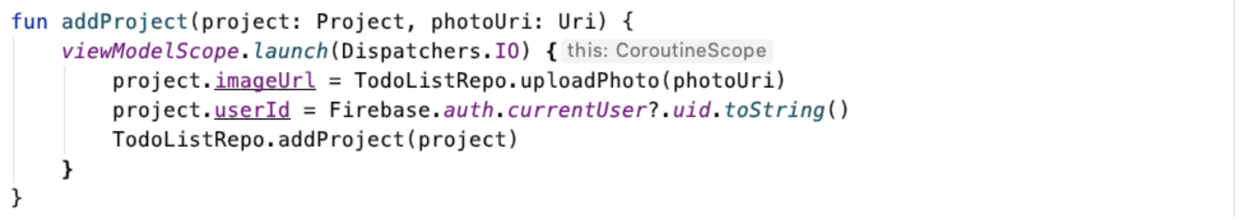
2. Create a function called **uploadPhoto** inside the TodoListRepo that takes Photo URI and upload it to the firebase storage. Then the function should return the uploaded photo URI

```
suspend fun uploadPhoto(photoUri: Uri): String {
    var timestamp : String  = SimpleDateFormat( pattern: "yyyyMMdd_HHmmss").format(Date())
    var fileName : String  = "IMAGE_" + timestamp + "_.png"

    var storageRef : StorageReference  = FirebaseStorage.getInstance().reference.child( pathString: "images")
        .child(fileName)

    val task : UploadTask.TaskSnapshot!  = storageRef.putFile(photoUri).await()
    return storageRef.downloadUrl.await().toString()
}
```

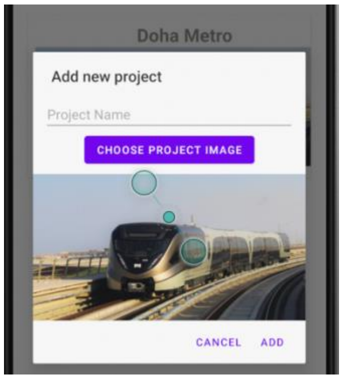
3. Modify the **addProject** function inside the ViewModel to call this function and pass the photo URI



```
fun addProject(project: Project, photoUri: Uri) {
    viewModelScope.launch(Dispatchers.IO) { this: CoroutineScope
        project.imageUrl = TodoListRepo.uploadPhoto(photoUri)
        project.userId = Firebase.auth.currentUser?.uid.toString()
        TodoListRepo.addProject(project)
    }
}
```

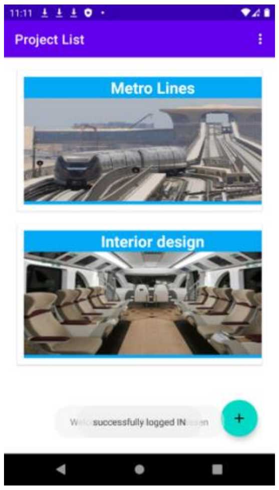
4. Create a function inside the **ProjectListFragment** called **openGallery** that allows you to select an image from the gallery.

5. Call this function when the dialog box's **Choose Project Image** button is clicked.
6. Add the **onActivityResult** function and get the image URI. Also, show this image in the dialog box



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

   projectViewModel.addProject(newProject, photoUri)
8. Test your application



PART C: Background Processing using Work Manager

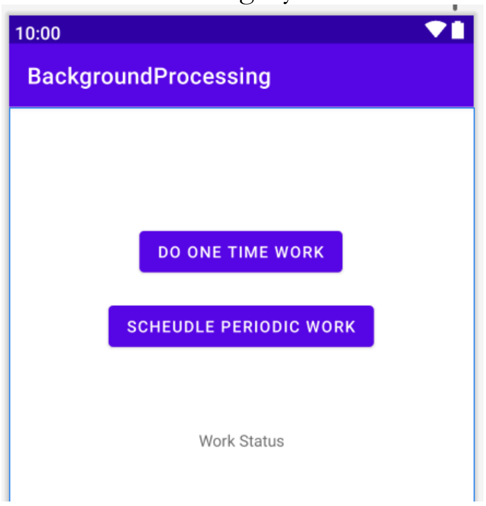
In this section, you will create a simple application that schedules work in the background. You will also learn how to get the status of running work.

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

```
// Kotlin + coroutines

implementation "androidx.work:work-runtime-ktx:2.4.0"
```

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 one time request object inside the MainActivity and enque 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

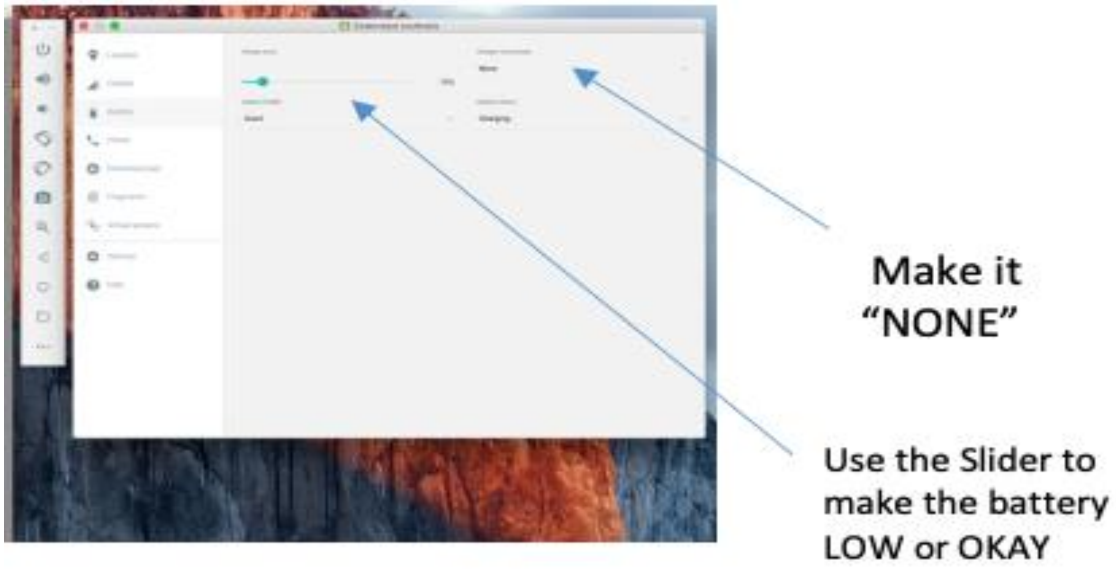
```
WorkManager.getInstance( context: this)
    .getWorkInfoByIdLiveData(oneTimeRequest.id)
    .observe( owner: this) { it: WorkInfo!
        workStatusTv.append("${it.state.name}\n\n")
    }
```

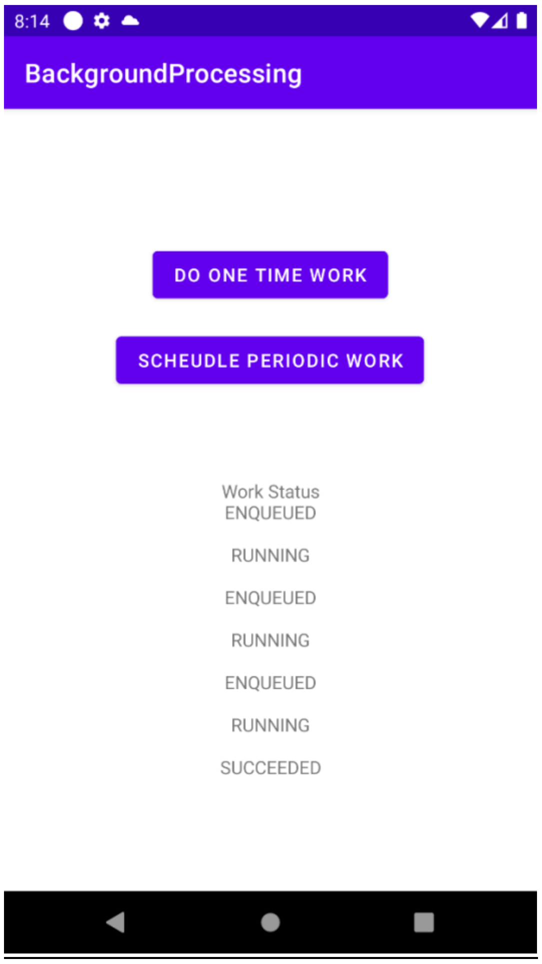
11. Test your code
12. Add the following constraints and test your code again by modifying the emulator properties

```
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()
```





```
Work Status
ENQUEUED

RUNNING

ENQUEUED

RUNNING

ENQUEUED

RUNNING

SUCCEEDED
```