MODUL 12
Coroutines and Flow

THEME DESCRIPTION

This module introduces students to background operations and data manipulations with Coroutines and Flow. Students will also learn how to manipulate and display the data using Kotlin Flow operators.

WEEKLY LEARNING OUTCOME (SUB-LEARNING OUTCOME)

Students will be able to use Coroutines and Flow to manage network calls in the background. They will also be able to manipulate data with Flow operators.

TOOLS/SOFTWARE USED

- Android Studio

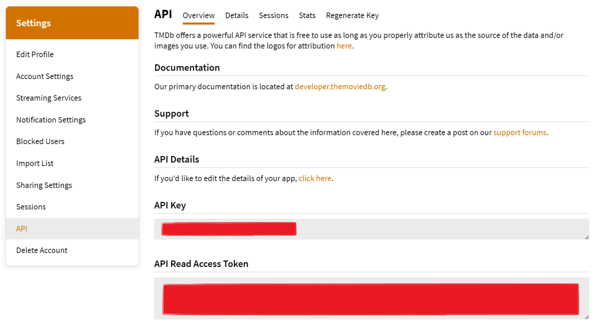
PRACTICAL STEPS

Part 1 - Building App with Coroutines

1. In this part, you will be building a **Movie List App**. The app shows a list of movies that you can press to check its details. You will be using **Coroutines** to make an **Asynchronous Call** for the **Api Call** and get all the movies.
2. To get started, download the starter app through this repository link. You will be continuing the project and implementing **Coroutines** into it.
   https://github.com/Ryuivan/LAB_WEEK_12.git

COMMIT to **GITHUB** at this point. **Commit Message: "Commit No. 1 – Init template"**.

3. Because we're using an **API**, you need an **API KEY** as per usual to send an **HTTP** call. In this app, you will be using the **themoviedb API**. Go to https://developers.themoviedb.org/ and **Register** for an account. After that, navigate to the **API Menu** and get the **API KEY**.



TMDb offers a powerful API service that is free to use as long as you properly attribute us as the source of the data and/or images you use. You can find the logos for attribution here.

Documentation

Our primary documentation is located at developer.themoviedb.org.

Support

If you have questions or comments about the information covered here, please create a post on our support forums.

API Details

If you'd like to edit the details of your app, click here.

API Key

[redacted]

API Read Access Token

[redacted]

4. To add the **API KEY** to the project, follow these steps. Update your **MovieService.kt** to the code below.

```
interface MovieService {
   @GET("movie/popular")
   // here, we are using the suspend keyword to indicate that this function
is a coroutine
   // suspended functions can be paused and resumed at a later time
   // this is useful for network calls, since they can take a long time to
complete
   // and we don't want to block the main thread
   // for more info, see:
https://kotlinlang.org/docs/flow.html#suspending-functions
   suspend fun getPopularMovies(
       @Query("api_key") apiKey: String
   ): PopularMoviesResponse
}
```

5. Next, update your **MovieRepository.kt** to the code below and add in your **API KEY**.

```
class MovieRepository(private val movieService: MovieService) {
   private val apiKey = "your_api_key_here"

   // LiveData that contains a list of movies
   private val movieLiveData = MutableLiveData<List<Movie>>()
   val movies: LiveData<List<Movie>>
       get() = movieLiveData

   // LiveData that contains an error message
   private val errorLiveData = MutableLiveData<String>()
   val error: LiveData<String>
       get() = errorLiveData
   // fetch movies from the API
   suspend fun fetchMovies() {
       try {
           // get the list of popular movies from the API
           val popularMovies = movieService.getPopularMovies(apiKey)
           movieLiveData.postValue(popularMovies.results)
       } catch (exception: Exception) {
           // if an error occurs, post the error message to the
errorLiveData
           errorLiveData.postValue(
               "An error occurred: ${exception.message}")
       }
   }
}
```

6. Next, update your **MovieApplication.kt** to the code below.

```
class MovieApplication : Application() {
   lateinit var movieRepository: MovieRepository

   override fun onCreate() {
       super.onCreate()
       // create a Retrofit instance
       val retrofit = Retrofit.Builder()
           .baseUrl("https://api.themoviedb.org/3/")
           .addConverterFactory(MoshiConverterFactory.create())
           .build()

       // create a MovieService instance
       // and bind the MovieService interface to the Retrofit instance
       // this allows us to make API calls
       val movieService = retrofit.create(
           MovieService::class.java
       )

       // create a MovieRepository instance
       movieRepository = MovieRepository(movieService)
   }
}
```

7. Next, update your **MovieViewModel.kt** to the code below.

```
class MovieViewModel(private val movieRepository: MovieRepository)
: ViewModel() {
   init {
       fetchPopularMovies()
   }
   // define the LiveData
   val popularMovies: LiveData<List<Movie>>
       get() = movieRepository.movies
   val error: LiveData<String>
       get() = movieRepository.error

   // fetch movies from the API
   private fun fetchPopularMovies() {
       // launch a coroutine in viewModelScope
       // Dispatchers.IO means that this coroutine will run on a shared
pool of threads
       viewModelScope.launch(Dispatchers.IO) {
           movieRepository.fetchMovies()
       }
   }
}
```

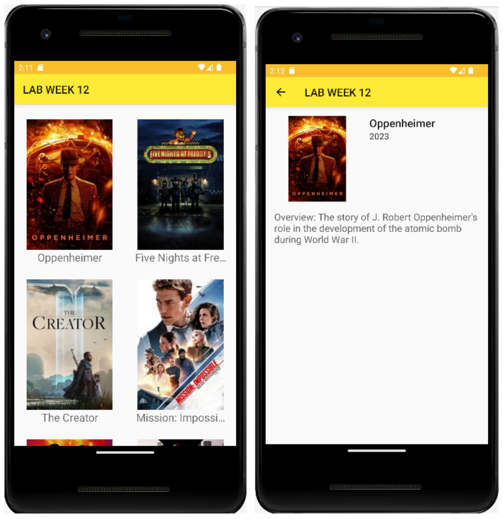
8. Here are types of **Dispatchers** that you can use for your **Coroutine**.

   - Dispatchers.Main: Used to run on Android’s main thread.
   - Dispatchers.IO: Used for network, file, or database operations.
   - Dispatchers.Default: Used for CPU-intensive work.

9. Now, open your **MainActivity.kt** and update your **onCreate** function.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    val recyclerView: RecyclerView = findViewById(R.id.movie_list)
    recyclerView.adapter = movieAdapter

    val movieRepository = (application as MovieApplication).movieRepository

    val movieViewModel = ViewModelProvider(
        this, object : ViewModelProvider.Factory {
            override fun <T : ViewModel> create(modelClass: Class<T>): T {
                return MovieViewModel(movieRepository) as T
            }
        })[MovieViewModel::class.java]
    movieViewModel.popularMovies.observe(this) { popularMovies ->
        val currentYear =
Calendar.getInstance().get(Calendar.YEAR).toString()

        movieAdapter.addMovies(
            popularMovies
                .filter { movie ->
                    // aman dari null
                    movie.releaseDate?.startsWith(currentYear) == true
                }
                .sortedByDescending { it.popularity }
        )
    }

    movieViewModel.error.observe(this) { error ->
        if (error.isNotEmpty()) {
            Snackbar.make(recyclerView, error, Snackbar.LENGTH_LONG).show()
        }
    }
}
```

10. **Run** your application. Movies should start to appear without blocking the main thread. If you click one of the movies, an activity containing the details of the movie should be displayed.



LAB WEEK 12
Oppenheimer
Five Nights at Fre...
The Creator
Mission: Impossi...
LAB WEEK 12
Oppenheimer
2023
Overview: The story of J. Robert Oppenheimer's role in the development of the atomic bomb during World War II.

COMMIT to **GITHUB** at this point. **Commit Message: "Commit No. 2 – Building app with coroutines"**.

Part 2 - Upgrading your App with Flow

1. You've built an app that uses **Coroutines** to suspend heavy functions in a way that makes the app handle **API Calls** asynchronously. You can upgrade your app by using **Flow** to retrieve the **Data Stream** Asynchronously.
2. Go to your **MovieRepository.kt** and remove both of the **movies** and the **error LiveData**. Also replace the **fetchMovies** function to the code below.

```
// fetch movies from the API
// this function returns a Flow of Movie objects
// a Flow is a type of coroutine that can emit multiple values
// for more info, see: https://kotlinlang.org/docs/flow.html#flows
fun fetchMovies(): Flow<List<Movie>> {
   return flow {
       // emit the list of popular movies from the API
       emit(movieService.getPopularMovies(apiKey).results)
       // use Dispatchers.IO to run this coroutine on a shared pool of
threads
   }.flowOn(Dispatchers.IO)
}
```

3. Next, open your **MovieViewModel.kt**. Again, remove the **movies** and **error LiveData** and the **fetchPopularMovies** function. Replace all of them with the code below.

```
// define the StateFlow in replace of the LiveData
// a StateFlow is an observable Flow that emits state updates to the
collectors
// MutableStateFlow is a StateFlow that you can change the value
private val _popularMovies = MutableStateFlow(
   emptyList<Movie>()
)
val popularMovies: StateFlow<List<Movie>> = _popularMovies

private val _error = MutableStateFlow("")
val error: StateFlow<String> = _error

// fetch movies from the API
private fun fetchPopularMovies() {
   // launch a coroutine in viewModelScope
   // Dispatchers.IO means that this coroutine will run on a shared pool of
threads
   viewModelScope.launch(Dispatchers.IO) {
       movieRepository.fetchMovies().catch {
           // catch is a terminal operator that catches exceptions
from the Flow
           _error.value = "An exception occurred: ${it.message}"
       }.collect {
           // collect is a terminal operator that collects the values from
the Flow
           // the results are emitted to the StateFlow
           _popularMovies.value = it
       }
   }
}
```

4. Lastly, open your **MainActivity.kt**. Remove the lines of code for observing **popularMovies** and **error** from **MovieViewModel**. Replace them with the code below.

```
// fetch movies from the API
// lifecycleScope is a lifecycle-aware coroutine scope
lifecycleScope.launch {
   // repeatOnLifecycle is a lifecycle-aware coroutine builder
   // Lifecycle.State.STARTED means that the coroutine will run
   // when the activity is started
   repeatOnLifecycle(Lifecycle.State.STARTED) {
       launch {
           // collect the list of movies from the StateFlow
           movieViewModel.popularMovies.collect {
               // add the list of movies to the adapter
               movies ->movieAdapter.addMovies(movies)
           }
       }
       launch {
           // collect the error message from the StateFlow
           movieViewModel.error.collect { error ->
               // if an error occurs, show a Snackbar with the error
message
               if (error.isNotEmpty()) Snackbar
                   .make(
                       recyclerView, error, Snackbar.LENGTH_LONG
                   ).show()
           }
       }
   }
}
```

5. **Run** your application. The app should work the same way as before but now you have **Flow** implemented into your app.

COMMIT to **GITHUB** at this point. **Commit Message: “Commit No. 2 – Upgrading app with flow”**.

ASSIGNMENT

Continue your **LAB_WEEK_12** project. You may notice when implementing **Flow** into your app, you’ve replaced **LiveData** with **StateFlow**. But in the process, you’ve also removed the **Data Filtering**. Implement the same data filter (descending by popularity) to the **StateFlow**.

COMMIT to **GITHUB** at this point. **Commit Message: “Commit No. 3 – Assignment”**.