MODUL 13
Architecture Patterns

THEME DESCRIPTION

This module introduces students to architectural patterns they can use for their Android projects. It covers using the Model-View-ViewModel (MVVM) pattern, adding ViewModels, and using data binding. Students will also learn about using the Repository pattern for caching data and WorkManager for scheduling data retrieval and storage.

WEEKLY LEARNING OUTCOME (SUB-LEARNING OUTCOME)

Students will be able to structure their Android project using MVVM and data binding. They will also be able to use the Repository pattern with the Room library to cache data and WorkManager to fetch and save data at a scheduled interval.

TOOLS/SOFTWARE USED

- Android Studio

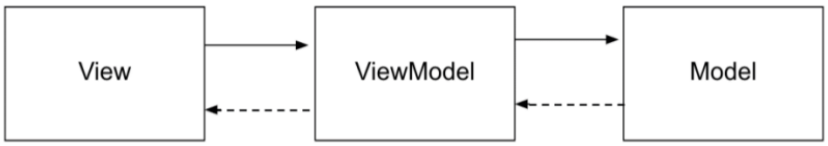
CONCEPTS

MVVM (Model - View - ViewModel)

MVVM allows you to separate the UI and business logic. When you need to redesign the UI or update the Model/business logic, you only need to touch the relevant component without affecting the other components of your app. This will make it easier for you to add new features and test your existing code. **MVVM** is also useful in creating huge applications that use a lot of data and views.

With the **MVVM** architectural pattern, your application will be grouped into three components:

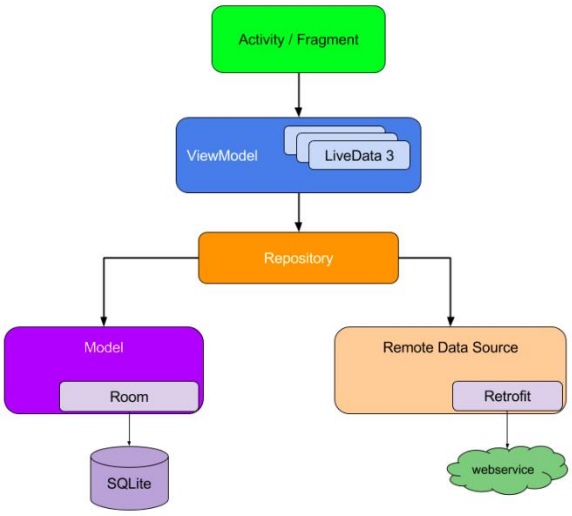
- **Model**: This represents the data layer
- **View**: This is the UI that displays the data
- **ViewModel**: This fetches data from Model and provides it to View



Repository Pattern

The **repository pattern** is a design pattern that provides us to **synchronize** the application data between web service (**API**) and your local database (**Room)**. This ensures that the data in your

local database will always be **up to date** with the data in the web service. This pattern also enables the user to access the app even when there's no connection in the area.



PRACTICAL STEPS

Part 1 - Using Data Binding in Android

1. Open Android Studio and continue your **"LAB_WEEK_12"** project.
2. Copy and paste the folder in your explorer and rename it to **"LAB_WEEK_13"**.
3. Inside the project, rename your package from **"lab_week_12"** to **"lab_week_13"**.
4. You can use **CTRL + SHIFT + F** to help you refactor the package name in all files.
5. In this part, you will be updating your **Movie List App,** and implement **Data Binding** into it.
6. To get started, update your **build.gradle (Module: app)** and add the code below.

```
buildFeatures{
   dataBinding true
}
```

7. To implement **Data Binding,** you need a way for your **View Model** to directly communicate with your **Views** without the need of using **FindViewById** or anything alike. First create a new file called **RecyclerViewBinding.kt** at the same level as **MainActivity.kt** and update it to the code below. This will allow you to call `app:list=""` inside your **RecyclerView**, and you can directly fill it with the **LiveData** or **StateFlow** from inside the **ViewModel**. You will do this in the next step.

```
@BindingAdapter("list")
fun bindMovies(view: RecyclerView, movies: List<Movie>?) {
```

```
    val adapter = view.adapter as MovieAdapter
    adapter.addMovies(movies ?: emptyList())
}
```

8. Next go to **activity_main.xml**. In here, you must include these 3 things:
    - You need to wrap everything inside a **Layout Tag** so that the binding library can generate a binding class for it.

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">
    <androidx.constraintlayout.widget.ConstraintLayout>
        ...
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

    - You need to add a **Data Element** inside your **Layout Tag** representing a layout variable for your views. This variable is bound directly to your **MovieViewModel.kt** and you can call just about anything public inside the view model.

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">
    <data>
        <variable
            name="viewModel"
            type="com.example.lab_week_13.MovieViewModel" />
    </data>
    <androidx.constraintlayout.widget.ConstraintLayout>
        ...
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

    - Lastly, you need to add `app:list="@{viewModel.popularMovies}"` inside the **RecyclerView Element** to bind the **popularMovies StateFlow** from **MovieViewModel.kt** directly into the **RecyclerView**.

```
<androidx.recyclerview.widget.RecyclerView
    ...
    app:list="@{viewModel.popularMovies}"/>
```

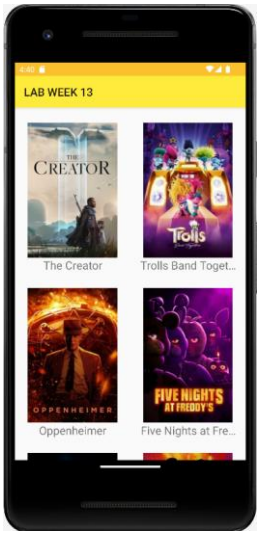
9. You might get an error, to fix that, go to **MainActivity.kt** and replace your **setContentView** with the code below. This will generate a **Binding Object** for your **activity_main.xml** to use.

```
val binding: ActivityMainBinding = DataBindingUtil
    .setContentView(this, R.layout.activity_main)
```

10. Next, bind your **movieViewModel** to the **Binding Object**. Add the code below after your **movieViewModel** declaration in **MainActivity.kt**.

```
binding.viewModel = movieViewModel
binding.lifecycleOwner = this
```

11. Lastly, remove everything that's inside the **lifecycleScope.launch**, basically removing every line after the code in **Step 11**.
12. Run your application and everything should work the same as before.



COMMIT to GITHUB at this point
Commit Message: "Commit No. 1 – add data binding"

Part 2 - Using Repository with Room

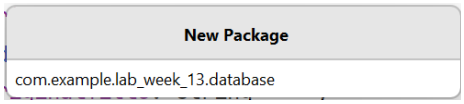
1. Continue your project from the first part. Your current repository doesn't have **Room** implemented into it, now you're going to do that.
2. Add the libraries below into your **build.gradle (Module: app)** for your **Room**.

```
implementation(libs.androidx.room.runtime)
implementation(libs.androidx.room.ktx)
kapt (libs.androidx.room.compiler)
```

3. As per usual, the first step is to add the **Entity Tag** to your **Model**. Update your **Movie.kt** and add the **Entity Tag** above your **Data Class**.

```
@Entity(tableName = "movies", primaryKeys = [("id")])
data class Movie(...)
```

4. Next, add a **DAO** into your project. Create a new **Package** called "**database**".



5. Inside the new package, create a new file called **MovieDao.kt** and update it to the code below.

```
@Dao
interface MovieDao {
   @Insert(onConflict = OnConflictStrategy.REPLACE)
   fun addMovies(movies: List<Movie>)
   @Query("SELECT * FROM movies")
   fun getMovies(): List<Movie>
}
```

6. Next, create a **Database** for your project. You will also implement the **Singleton Pattern** for your **Database Declaration**. The **Singleton Pattern** makes sure that there's only 1 single instance of your **Database** throughout multiple threads. This ensures data validity and also prevents **Race Conditions**. This pattern is crucial as creating a new instance of the **Database** is expensive. Still inside the new package, create a new file called **MovieDatabase.kt** and update it to the code below.

```
@Database(entities = [Movie::class], version = 1)
abstract class MovieDatabase : RoomDatabase() {
    abstract fun movieDao(): MovieDao
    companion object {
        // @Volatile prevents Race Condition.
        // If another thread is updating the database through the instance,
        // the value of instance will be immediately visible to the other
thread.
        // This ensures that the value of instance is always up-to-date and
the same to all execution threads.
        @Volatile
        private var instance: MovieDatabase? = null
        fun getInstance(context: Context): MovieDatabase {
            // synchronized() ensures that only one thread can execute this
block of code at a time.
            // If multiple threads try to execute this block of code at the
same time,
            // only one thread can execute it while the other threads wait
for the first thread to finish.
            return instance ?: synchronized(this) {
                instance ?: buildDatabase(
                    context
                ).also { instance = it }
            }
        }
        private fun buildDatabase(context: Context): MovieDatabase {
            return Room.databaseBuilder(
                context,
                MovieDatabase::class.java, "movie-db"
            ).build()
        }
    }
}
```

7. Next, add your **MovieDatabase** into your **MovieRepository**. Update your **MovieRepository Constructor Parameters** to the code below.

```
class MovieRepository(
    private val movieService: MovieService,
```

```
    private val movieDatabase: MovieDatabase) {
    ...
}
```

8. Now, you need to utilize that **movieDatabase** parameter. Before, you directly fetch the movie list from the **API** and display it to the user. Now, you need to check the existence of the movie list first from your **Database** using **Room**. If it exists, get data from **Room**. If it doesn't, fetch the data from the **API**, then store that data inside your **Database** using **Room**. Still inside **MovieRepository.kt**, update your **fetchMovies** function to the code below.

```
fun fetchMovies(): Flow<List<Movie>> {
    return flow {
        // Check if there are movies saved in the database
        val movieDao: MovieDao = movieDatabase.movieDao()
        val savedMovies = movieDao.getMovies()
        // If there are no movies saved in the database,
        // fetch the list of popular movies from the API
        if(savedMovies.isEmpty()) {
            val movies = movieService.getPopularMovies(apiKey).results
            // save the list of popular movies to the database
            movieDao.addMovies(movies)
            // emit the list of popular movies from the API
            emit(movies)
        } else {
            // If there are movies saved in the database,
            // emit the list of saved movies from the database
            emit(savedMovies)
        }
    }.flowOn(Dispatchers.IO)
}
```

9. Lastly, go to your **MovieApplication.kt** and update the declaration of **movieRepository** to the code below.

```
// create a MovieDatabase instance
val movieDatabase =
    MovieDatabase.getInstance(applicationContext)
```

```
// create a MovieRepository instance
movieRepository =
   MovieRepository(movieService, movieDatabase)
```

10. Run your application and it should work the same as before. But, if you **Disconnect** your phone from your network, the movie list is still there, which is now cached in the database.
11. In case you got the **Error** below.

```
An error occurred: NOT NULL constraint failed: movies.backdrop_path (code
1299 SQLITE_CONSTRAINT_NOTNULL)
```

Go to **Movie.kt**, and update the **backdrop_path** column to the code below.

```
val backdrop_path: String? = "",
```

COMMIT to GITHUB at this point
Commit Message: "Commit No. 2 – update to Room"

Part 3 - Using WorkManager

1. Continue your project from the second part. Your project is almost done, but there's still 1 problem. Currently there's no way of refreshing your **Database** with the **Latest Version** of data from the **API**. Usually you want to fix this by fetching data from the **API** at an interval or even when you're not opening the app. This way, the cached data will always be **Up to Date**. You can do this by utilizing **WorkManager**.
2. Add the libraries below into your **build.gradle (Module: app)** for your **WorkManager**.

```
implementation (libs.androidx.work.runtime)
```

3. First, make a function that will be called at an interval by the **Worker**. Go to your **MovieRepository.kt** and add the code below as a new function in your repository.

```
// fetch movies from the API and save them to the database
// this function is used at an interval to refresh the list of popular
```

```
movies
suspend fun fetchMoviesFromNetwork() {
    val movieDao: MovieDao = movieDatabase.movieDao()
    try {
        val popularMovies = movieService.getPopularMovies(apiKey)
        val moviesFetched = popularMovies.results
        movieDao.addMovies(moviesFetched)
    } catch (exception: Exception) {
        Log.d(
            "MovieRepository",
            "An error occurred: ${exception.message}"
        )
    }
}
```

4. Now, create a new file called **MovieWorker.kt** and update it to the code below.

```
class MovieWorker(
    private val context: Context, params: WorkerParameters
) : Worker(context, params) {
    // doWork() is called in a background thread
    // this is where you put the code that you want to run
    override fun doWork(): Result {
        // get a reference to the repository
        val movieRepository = (context as MovieApplication)
            .movieRepository
        // launch a coroutine in the IO thread
        CoroutineScope(Dispatchers.IO).launch {
            movieRepository.fetchMoviesFromNetwork()
        }
        return Result.success()
    }
}
```

5. Lastly, schedule your worker to call the `fetchMoviesFromNetwork` function every hour. Add the code below at the end of your **onCreate Callback** in the **MovieApplication.kt**.

```
// create a Constraints instance
```

```
// this will be used to specify the conditions that must be met
// in order to run worker tasks
val constraints = Constraints.Builder()
    // only run the task if the device is connected to the internet
    .setRequiredNetworkType(NetworkType.CONNECTED)
    .build()
// create a WorkRequest instance
// this will be used to schedule a background task
val workRequest = PeriodicWorkRequest
    // run the task every 1 hour
    // even if the app is closed or the device is restarted
    .Builder(
        MovieWorker::class.java, 1,
        TimeUnit.HOURS
    ).setConstraints(constraints)
    .addTag("movie-work").build()
// schedule the background task
WorkManager.getInstance(
    applicationContext
).enqueue(workRequest)
```

6. Run your application and it should work the same as before. But, now if you check back again after 1 hour or more, if there's a new movie added into the **API**, then the movie list should be updated.

COMMIT to GITHUB at this point
Commit Message: "Commit No. 3 – update to WorkManager"

ASSIGNMENT

Answer these questions based on the tutorial:

1. Why is **MVVM** important? Which files represent **Model**, which files represent **View**, and which files represent **ViewModel**?
2. In **Part 1**, you implemented **Data Binding**, why is this more efficient than using the normal method?
3. In **Part 2**, you implemented the **Singleton Pattern**, why is this important?
4. In **Part 2 & 3**, you implemented the **Repository Pattern**, why is this important?
5. In **part 3**, you implemented the **Worker Manager**, is there another way to refresh your database with the latest data other than using **Worker**?

Write your answer in a **Text File** and name it **LAB_WEEK_13.txt**.

COMMIT to GITHUB at this point
Commit Message: "Commit No. 4 – assignment"