**Section 31 – Paging Library + MVVM + Data Binding – Movie App**

**1. Key Concepts Learned**

1. **Paging Library**
   * Efficiently loads large datasets in **pages** instead of all at once.
   * Improves performance, reduces memory usage, and supports infinite scrolling.
2. **MVVM Architecture with Paging**
   * **Model** → API / Data source.
   * **ViewModel** → Uses Pager and PagingData to expose paged data as LiveData or Flow.
   * **View (Activity/Fragment)** → Submits paged data to PagingDataAdapter.
3. **PagingDataAdapter**
   * A special RecyclerView adapter optimized for paged lists.
   * Works with DiffUtil.ItemCallback for efficient item updates.
4. **Data Binding in Paging**
   * Binds UI components directly to data variables in XML.
   * Minimizes boilerplate and reduces manual findViewById calls.
5. **Retrofit with Paging**
   * API is integrated into PagingSource for fetching paged results.
6. **Glide with Data Binding**
   * Loads images in RecyclerView items using custom @BindingAdapter.

**2. Step-by-Step Implementation**

**Step 1 – Add Dependencies**

// Paging 3

implementation "androidx.paging:paging-runtime:3.x.x"

// Retrofit & Gson

implementation 'com.squareup.retrofit2:retrofit:2.x.x'

implementation 'com.squareup.retrofit2:converter-gson:2.x.x'

// Glide for images

implementation 'com.github.bumptech.glide:glide:4.x.x'

annotationProcessor 'com.github.bumptech.glide:compiler:4.x.x'

// Data Binding

buildFeatures {

dataBinding true

}

**Step 2 – API Interface for Paging**

The TMDB API supports pagination via page query parameter.

public interface MovieApi {

@GET("movie/popular")

Call<MovieResponse> getPopularMovies(

@Query("api\_key") String apiKey,

@Query("page") int page

);

}

**Step 3 – PagingSource Implementation**

PagingSource<Key, Value> defines how to load each page of data.

public class MoviePagingSource extends PagingSource<Integer, Movie> {

private final MovieApi movieApi;

private final String apiKey;

public MoviePagingSource(MovieApi movieApi, String apiKey) {

this.movieApi = movieApi;

this.apiKey = apiKey;

}

@NonNull

@Override

public LoadResult<Integer, Movie> load(@NonNull LoadParams<Integer> params) {

try {

int currentPage = params.getKey() != null ? params.getKey() : 1;

Response<MovieResponse> response = movieApi

.getPopularMovies(apiKey, currentPage)

.execute(); // Synchronous for Paging

if (response.isSuccessful() && response.body() != null) {

List<Movie> movies = response.body().getResults();

return new LoadResult.Page<>(

movies,

currentPage == 1 ? null : currentPage - 1,

currentPage + 1

);

} else {

return new LoadResult.Error<>(new Exception("API error"));

}

} catch (Exception e) {

return new LoadResult.Error<>(e);

}

}

@NonNull

@Override

public Key getRefreshKey(@NonNull PagingState<Integer, Movie> state) {

return 1; // Always refresh from first page

}

}

**Key Points:**

* **load()** → Fetches data for the requested page.
* **prevKey & nextKey** → Tell Paging Library how to navigate pages.

**Step 4 – Repository**

public class MovieRepository {

private final MovieApi movieApi;

private final String apiKey;

public MovieRepository(MovieApi movieApi, String apiKey) {

this.movieApi = movieApi;

this.apiKey = apiKey;

}

public Pager<Integer, Movie> getMoviesPager() {

return new Pager<>(

new PagingConfig(20), // page size

() -> new MoviePagingSource(movieApi, apiKey)

);

}

}

**Why Pager?**

* Pager automatically creates a **Flow** or **LiveData** of paged data.

**Step 5 – ViewModel**

public class MovieViewModel extends ViewModel {

private final MovieRepository repository;

public LiveData<PagingData<Movie>> moviesLiveData;

public MovieViewModel(MovieRepository repository) {

this.repository = repository;

moviesLiveData = new Pager<>(

new PagingConfig(20),

() -> new MoviePagingSource(repository.getMovieApi(), repository.getApiKey())

).liveData.cachedIn(viewModelScope);

}

}

**Key Points:**

* .liveData → Exposes paging as LiveData.
* .cachedIn(viewModelScope) → Keeps data alive during configuration changes.

**Step 6 – PagingDataAdapter**

public class MovieAdapter extends PagingDataAdapter<Movie, MovieAdapter.MovieViewHolder> {

public MovieAdapter() {

super(DIFF\_CALLBACK);

}

private static final DiffUtil.ItemCallback<Movie> DIFF\_CALLBACK =

new DiffUtil.ItemCallback<Movie>() {

@Override

public boolean areItemsTheSame(@NonNull Movie oldItem, @NonNull Movie newItem) {

return oldItem.getId() == newItem.getId();

}

@Override

public boolean areContentsTheSame(@NonNull Movie oldItem, @NonNull Movie newItem) {

return oldItem.equals(newItem);

}

};

@NonNull

@Override

public MovieViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {

MovieItemBinding binding = DataBindingUtil.inflate(

LayoutInflater.from(parent.getContext()),

R.layout.movie\_item, parent, false);

return new MovieViewHolder(binding);

}

@Override

public void onBindViewHolder(@NonNull MovieViewHolder holder, int position) {

Movie movie = getItem(position);

holder.binding.setMovie(movie); // Bind data directly

}

static class MovieViewHolder extends RecyclerView.ViewHolder {

MovieItemBinding binding;

public MovieViewHolder(MovieItemBinding binding) {

super(binding.getRoot());

this.binding = binding;

}

}

}

**Step 7 – Data Binding in Layout**

**movie\_item.xml**

<layout xmlns:android="http://schemas.android.com/apk/res/android">

<data>

<variable

name="movie"

type="com.example.app.Movie" />

</data>

<LinearLayout ...>

<ImageView

android:id="@+id/imagePoster"

android:layout\_width="100dp"

android:layout\_height="150dp"

app:imageUrl="@{movie.posterPath}" />

<TextView

android:id="@+id/textTitle"

android:text="@{movie.title}" />

<TextView

android:id="@+id/textReleaseDate"

android:text="@{movie.releaseDate}" />

</LinearLayout>

</layout>

**Step 8 – Binding Adapter for Glide**

@BindingAdapter("imageUrl")

public static void loadImage(ImageView view, String url) {

Glide.with(view.getContext())

.load("https://image.tmdb.org/t/p/w500" + url)

.into(view);

}

**Step 9 – Activity/Fragment**

public class MainActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

ActivityMainBinding binding = DataBindingUtil.setContentView(this, R.layout.activity\_main);

MovieAdapter adapter = new MovieAdapter();

binding.recyclerView.setLayoutManager(new LinearLayoutManager(this));

binding.recyclerView.setAdapter(adapter);

MovieViewModel viewModel = new ViewModelProvider(this).get(MovieViewModel.class);

viewModel.moviesLiveData.observe(this, adapter::submitData);

} }

**3. Tools, Libraries, and APIs Used**

* **Paging 3** → Incremental data loading.
* **Retrofit** → API calls.
* **Gson** → JSON parsing.
* **Data Binding** → Binding UI and data directly.
* **Glide** → Image loading.
* **DiffUtil** → Efficient list updates.

**4. Best Practices & Modern Alternatives**

* Use **Kotlin coroutines + Flow** instead of Java synchronous .execute() calls.
* Combine **RemoteMediator** with Paging 3 for **offline caching** via Room.
* Always use DiffUtil in PagingDataAdapter (already implemented here).
* Keep **API keys secure** (local.properties, NDK).
* Add **LoadStateAdapter** for showing loading/error at list ends.
* Use sealed class UiState for loading/error/success in UI.

**5. Part B – Important but Missing from Section**

1. **LoadState Handling**
   * Show progress bar while new page loads.
   * Retry button on network failure.
2. **Offline Mode**
   * Combine Paging 3 with Room (RemoteMediator) for caching.
3. **UI Testing**
   * Test Paging flows using TestPagingSource and mock API.
4. **Error Logging**
   * Use Timber for structured logging.
5. **Accessibility**
   * Content descriptions for images.
6. **PagingConfig Tweaks**
   * Tune prefetchDistance and enablePlaceholders for performance.