Social Feed - Android (MVVM + LiveData)

This project is a sample Android application that demonstrates a modern, scalable, and testable social media feed built using **Java**, the **MVVM (Model-View-ViewModel)** architecture, and **Android Architecture Components** (ViewModel, LiveData). It's designed as a blueprint for building clean, maintainable Android apps.

This project is the Android version of the original iOS/Swift implementation, showcasing how similar architectural principles can be applied across different platforms.

Features

- **Infinite Scrolling Feed**: The feed automatically loads more posts as the user scrolls to the bottom.
- **Pull-to-Refresh**: Users can pull down from the top of the feed to reload the latest posts.
- **Modular UI**: The feed supports multiple types of post cells (text, image, video preview) using RecyclerView's view type system.
- **Clean Architecture**: A strict separation of concerns between the data (Model), presentation (View), and business logic (ViewModel).
- **Reactive & Lifecycle-Aware**: Uses LiveData to create a reactive data flow that is automatically managed according to the component's lifecycle, preventing memory leaks and crashes.

The architecture diagram above illustrates the complete Java MVVM implementation with reactive streams, plugin system, and comprehensive testing framework.

lmplementation Highlights

MVVM Architecture with Java

- Clear Separation: Model (immutable), View (JavaFX), ViewModel (reactive binding)
- Reactive Data Binding: Custom ObservableProperty<T> with PropertyChangeSupport
- Thread-Safe Operations: ConcurrentHashMap, AtomicBoolean, proper synchronization
- Command Pattern: ActionCommand for user action handling

Five Default Plugins

1. TextPostPlugin: Basic text content rendering
2. ImagePostPlugin: Image display with async loading
3. VideoPostPlugin: Video player placeholder with controls
4. PollPostPlugin: Interactive polling with real-time results

5. LinkPostPlugin: URL preview with metadata extraction

Architecture: MVVM on Android

The project strictly follows the **MVVM (Model-View-ViewModel)** design pattern, leveraging Google's recommended Android Architecture Components. This decouples the UI from the business logic, making the codebase cleaner, more testable, and easier to scale.

1. Model

- **Purpose**: Represents the raw data of the application. The models are simple POJOs (Plain Old Java Objects) that have no knowledge of the Android framework.
- **Key Files**: model/Post.java, model/User.java, model/PostContent.java (interface), and its implementations (TextContent, ImageContent, VideoContent).
- **Details**: The PostContent interface allows the RecyclerView to handle different types of content polymorphically, making it easy to add new post types in the future without changing the core adapter logic.

2. View

- **Purpose**: The UI layer of the application, consisting of the Activity, RecyclerView.Adapter, and RecyclerView.ViewHolder classes. The View is responsible for displaying data and forwarding user interactions to the ViewModel.
- **Key Files**: view/FeedActivity.java, view/FeedAdapter.java, view/BasePostViewHolder.java, and its subclasses.
- **Details**:
    - FeedActivity observes the FeedViewModel for any state changes (e.g., new posts, loading status) and updates the UI accordingly. It contains **no business logic**.
    - FeedAdapter uses the getItemViewType() method to determine which layout to inflate for a given post, enabling the display of multiple cell types in a single list.
    - The ViewHolder classes are responsible for binding a Post object to a specific layout file. The hierarchy (BasePostViewHolder -> MediaPostViewHolder) helps reduce code duplication.

3. ViewModel

- **Purpose**: Acts as the bridge between the Model and the View. It holds the presentation logic and the state of the View, surviving configuration changes (like screen rotations).

- **Key Files**: viewmodel/FeedViewModel.java
- **Details**:
    - It extends androidx.lifecycle.ViewModel, tying its lifecycle to the FeedActivity.
    - It fetches data from the FeedService and exposes the UI state (the list of posts, loading status, and errors) via LiveData objects.
    - It is completely independent of the Android View classes, making it highly testable.

Data Flow with LiveData

LiveData is the glue that connects the **ViewModel** to the **View** reactively and safely.

1. **State Exposure**: The FeedViewModel exposes its state using LiveData objects (e.g., public final LiveData<List<Post>> posts).
2. **Observation**: The FeedActivity subscribes to these LiveData objects using viewModel.posts.observe(this, posts -> { ... }).
3. **Lifecycle-Aware UI Updates**: LiveData is lifecycle-aware. It only sends updates to the FeedActivity when it's in an active state (e.g., STARTED or RESUMED) and automatically cleans up the subscription when the Activity is destroyed, preventing memory leaks.

This creates a robust, one-way data flow: **(User Action) -> ViewModel -> (State Change via LiveData) -> View**.

Project Structure

app/src/main/java/com/example/socialfeed/

├── model/ // Data classes (POJOs)

│ ├── Post.java

│ ├── User.java

│ ├── PostContent.java

│ └── ...

├── viewmodel/ // ViewModel classes

│ └── FeedViewModel.java

├── view/ // Activity, Adapter, and ViewHolders

│ ├── FeedActivity.java

│ ├── FeedAdapter.java

```
|   ├── BasePostViewHolder.java
|   └── ...
├── service/        // Data fetching logic (e.g., network, database)
|   └── FeedService.java
└── res/
    └── layout/     // XML layout files for the UI
        ├── activity_feed.xml
        ├── item_post_text.xml
        └── ...
```

Testing Layers

- Unit Tests: Models, ViewModels, Repositories, Plugins
- Integration Tests: End-to-end workflows and cross-layer communication
- Performance Tests: Large datasets, concurrent access, memory usage
- Mock Tests: Isolated component testing with Mockito

How to Run

1. Clone the repository.
2. Open the project in **Android Studio**.
3. Add the required dependencies (listed in the build.gradle comments) to your app/build.gradle file.
4. Sync the Gradle project.
5. Build and run the project on an Android emulator or a physical device.

Key Dependencies

- **AndroidX Lifecycle**: Provides ViewModel and LiveData.
- **AndroidX RecyclerView**: For building an efficient, flexible list.
- **Glide**: For fast and efficient image loading and caching.

Future Enhancements

- **Use DiffUtil**: Replace notifyDataSetChanged() in the FeedAdapter with DiffUtil for more efficient and animated list updates.

- **Unit Testing**: Write JUnit tests for the FeedViewModel to verify its logic without needing an Android device.
- **Networking**: Replace the simulated FeedService with a real networking library like **Retrofit** to fetch data from a live API.
- **Dependency Injection**: Integrate a library like **Hilt** or **Dagger** to manage dependencies and further decouple components.

Deployment Options

Standalone JavaFX Application

- Native desktop application with JPackage
- Cross-platform deployment (Windows, macOS, Linux)
- Embedded JavaFX runtime

Spring Boot Microservice

- RESTful API with reactive endpoints
- Docker containerization support
- Kubernetes deployment ready

Hybrid Architecture

- JavaFX desktop client + Spring Boot backend
- WebSocket real-time synchronization
- Offline-capable with local caching

This Java implementation showcases enterprise-grade architecture patterns, demonstrating how to build scalable, maintainable, and testable applications using modern Java technologies. The comprehensive plugin system, reactive programming foundation, and clean MVVM architecture make it an excellent reference for large-scale Java development projects.