MusicBrainzViewer: Music release notification and artist/album browser under the modern Android architecture

Martin Symons

May 19, 2024

1 Introduction

MusicBrainzViewer is an exploration app for the *MusicBrainz* API, retrieving artist images separately via *Fanart.TV*. Users are capable of simple music exposure tasks:

- Free-form search for artists
- Enumeration of each artist's releases
- Viewing of detailed information and images for each artist
- Following of artists to receive notifications on new releases, retrieved in the background

Priority, however, was on producing an application to *Google*'s modern architectural standards, including the use of six *Jetpack* libraries (on top of some not produced by *Google*, though strongly recommended) and adhering to the *Modern App Architecture.*

MusicBrainzViewer was produced over approximately two weeks. This progression can be seen between the original repo and its replacement.

(https://github.com/movedaccount-droid/osmviewer)

(https://github.com/movedaccount-droid/osmviewerjetpack2)

2 Architecture Overview

MusicBrainzViewer fits the *Modern App Architecture.* Thus, its structure can be split into *UI* and *Data* layers.

Beginning with the UI layer, Jetpack Compose is used to provide native Material components with built-in persistence-on-recompose and layout adjustment. Coil provides asynchronous image loading from raw URIs. Navigation between Compose components is performed via Jetpack Navigate's native support, the *NavHost* composable.

Compose provides several means to store state, but we primarily use ViewModels (and the MVVM pattern as a whole) to separate business logic from the UI layer. These are dependency injected with deeper Data layer repositories and Kotlin coroutine dispatchers (along with any further requirements) via Hilt to provide long-lived service state and ease of testing through fakes.

Data is retrieved from APIs via Retrofit2, a standard library by *Square* providing type-safe automatic deserialization of JSON responses through Gson, and global logging and header configuration through dependency-injected OkHttp objects. Entity data from either API is cached on reception into a Jetpack Room database, defining network, local and external classes for each to fit our architecture. Custom converters into Gson are provided where SQLite cannot natively handle any data. Access is provided through Daos, or paginated through Jetpack Paging 3, providing safe and adaptive access to MusicBrainz' otherwise quirky API paging structure.

Finally, requests for followed artists are made automatically via Jetpack Workmanager, choosing opportune times to make batch requests against the network.

3 Discussion

3.1 UI Design

Jetpack Compose offers the Material toolkit, allowing standards specific implementation out-of-the-box. These components are often atomic and must be assembled into their larger design; this has been performed wherever possible, including colors, sizes, paddings, and usage of components in their appropriate place. This is most visible on the header card seen on the album and artist pages, for which Material provides the container but no particular layout, despite specifying exact padding and positioning.

The Scaffold composable has been used to provide a global container across all pages, including a floating action button, tabs and search bar. The search bar drops down to cover the screen once selected - this was intended to provide a search history whilst typing, though this was not included due to time constraints. A NavHost within this layer performs all navigation through Jetpack Navigate. Users can jump between any screen in the app at any time, whilst retaining a clear Home → Artist → Album hierarchy. All events within composables is passed upwards as part of our architecture - only data and callbacks are passed down.

All halting operations take place in suspend functions away from the main thread. The app therefore remains responsive at all times unless under load, such as very fast scrolling. No ANR dialogs appear at any time. ViewModels guarantee that state, such as the viewed data, is retained throughout recompositions including after screen rotation.

Users are informed via on-screen text of key app operations, or through visual cues like icons and color differences suggested by Material. Image descriptions are provided for accessibility - most other accessibility features are provided through Compose itself.

3.2 Code Quality

Code is heavily commented and packages are organised logically. All code follows Kotlin standard practice.

All data is served to external components through repositories. This includes access to both APIs and the WorkManager implementation. Data is retrieved from APIs through services. No domain layer is present, though the application is nearing the point where this would be appropriate.

All dependencies are injected wherever possible. This includes coroutine dispatchers, which are replaced whenever running tests to force coroutines to run on a single thread, reducing flakiness. Where needed, factory methods are included for assisted injection, allowing user parameters on top of those provided by Hilt. Most dependencies persist as singletons for the app's lifetime: this is most noticed in the repositories and paging sources, where a global RateLimiter is injected to ensure no requests beyond an API's limit are made regardless of screen state or the number of client objects active.

Several advanced threading features are used; *updateFollowedCaches()* within the MusicBrainzRepository spins out several threads and manages their access to a central list via mutex, collecting all results before continuing. Multithreading in this way allows the app to complete its tasks quickly when resources are available, important especially when running in the background as a worker.

3.3 Data Management

Data management is the largest obstacle in an API-driven application. Reconciliation of the cached and networked resources is performed primarily in the MusicBrainzRepository, the largest class in the project. The application is offline-first in accordance with Android recommendations - the user can disable the network and still see their followed artists. Primarily, however, caching provides a performance gain. MusicBrainz and Fanart.tv both provide large blobs of data much more than needed to render a single screen - caching allows us to make use of this data without waste, when screens lower on the hierarchy are loaded. Outdated caches are purged in the background as part of the sync worker.

API data is retrieved responsibly - beyond the mentioned rate limiting, OkHttp configuration injects a User-Agent header into all requests for identification. The Room database holding this data is queried both with simple lookups and updates and more advanced queries to retrieve data with relationships, mostly via custom SQL returning multimap objects as part of current Google guidance.

3.4 Android Key Concepts

WorkManager integration has already been discussed, but users are also provided notifications for new releases from artists they follow.

3.5 Testing

JUnit testing is passing for the two most involved actions, pruning of old cached data and retrieval of new releases from followed artists. Both of these actions are tied to passing time and external events, and would be difficult to observe otherwise. Fakes for each Dao in the app are produced to provide controlled data. Test data is generated through factory methods for convenience. Since pruning tests are performed on Room itself, these are instrumented, using Room's built-in in-memory-database feature to reset the database between each test.

Functionality was additionally regularly tested throughout development with multiple full-runs of functionality, with several bugs removed as a result. This includes off-by-one image loading errors, navigation edge cases causing ViewModels to receive bad data and discrepancies with paged data retrieval, which can be seen through the commit history.

Error handling is seen throughout the code to prevent crashes in exceptional cases, like the network going down or APIs responding with error codes.

4 User Guide

A two-minute video user guide can be seen on YouTube here.
(https://www.youtube.com/watch?v=HtqgOV0sPC8)

5 Remaining Problems

Several problems still remain with the final solution. We elaborate on their origin and their potential future fixes.

5.1 Room and Early Optimization

What does a performant Room query look like? Answering this requires a solid foundation of SQL along with Room's idiosyncrasies. Having little experience in either, repositories in MusicBrainzViewer are designed to perform the fewest queries possible for each action, an approach seen mostly in large-scale architectures like the APIs we are querying from. On a local device, especially with multithreading, this is unnecessary. As a result, many queries are more complex than they should be, resulting in bloated Dao classes, and some repository functions work with limited or strange data that they likely should not. This has a knock-on impact on testing: very large functions taking from several points of data require tests to be designed with internal implementation knowledge; this approach cannot scale and massively increases the effort and failure rate behind generated tests.

The correct approach would likely include a domain layer, providing compound lookups on simpler Dao requests when needed. The repositories could then focus on combining this data into their external outputs. This would lead to a greater number of queries made, but at little cost to user experience.

5.2 Room and API design

When querying APIs directly, it is appropriate to design network and local classes referencing only the observed response. When this data must be cached and queried locally it must instead mirror the external schema, so that database expansion can be performed logically without a large blast radius. The response will also be consistent: there are no unexpected changes at the network layer, and so local models are free to perform more rigorous adaptation of the response without maintenance costs. MusicBrainzViewer was not developed this way, and suffers as a result: numerous relationships are not well-defined as in MusicBrainz itself. When attempting to get notification data, this finally broke: following the M:N artist-to-release-group relationship after the 1:1 release-group-to-notification relationship was not feasible. The complexity of the prior issue is thus compounded.

Room itself is undergoing a large shift from operating as a replacement to SQL to a lower-level convenience layer, and moving from POJO intermediate objects for relations to a cleaner multimap approach. Finding the right approach is therefore a task in itself: still, it appears that multimap is best used for any queries up to M:N relationships, beyond which heavily annotated POJOs must be used. In any future project this approach would be applied consistently for all local relations.

5.3 Other Issues

Other smaller issues are seen:

- The search bar has strange padding when closing due to the introduction of the back button. It also does not respond to three-button navigation. Both are the result of the same problem: the Material SearchBar component should not be relied on alone for this functionality. Instead, tapping search should open a new view completely, with a custom shared element transition applied to scale the search bar separate from the scaffold.

- Whilst error handling is in place to prevent crashing, much of it is not shown back to the user. As a result, most failures simply result in a blank screen or long-lasting progress indicator.

6 Conclusion

MusicBrainzViewer is a feature-simple, architecturally complex application including cutting edge Android development standards. It includes functionality to search, view and follow artists, as well as detailed information on each. Through the inclusion of many data management and querying libraries, caching and paging functionality has been included to improve performance. Issues remaining with its design have been identified, and a forward path earmarked for future development.