KBTU KAZAKH-BRITISH TECHNICAL UNIVERSITY

Assignment 4
Mobile Programming
Working with Databases and Retrofit in
Kotlin Android Applications

Prepared by:
Seitbekov S.
Checked by:
Serek A.

Almaty, 2024

Table of Contents

Executive Summary

The given report represents the design and implementation of Weebnet, which is an anime discovery application implemented in Kotlin using Room, Retrofit, and Jetpack Compose. The application integrates local database management and network requests with caching, guaranteeing a seamless user experience even in offline scenarios. It also applies clean architecture with a repository pattern for modularity and scalability of the code. Features centered around the user included profiles, favorites, and personal interaction. A Netflix-inspired UI boosted this in terms of visual aspects. Testing will be done, which includes paying great attention to API services and how the database functions. Below is a report on all developments the platform has undergone, challenges posed, and solutions that were implemented.

Introduction

Modern Android applications manage a large amount of data and must keep the local and remote data sources in sync. Databases and RESTful APIs are its backbone. Weebnet, brings together such technologies to create a beautiful and responsive anime discovery experience. Room will be used for efficient data persistence locally and for caching, while Retrofit handles network interactions with an anime API. The dark theme, inspired by Netflix, makes it both intuitive and engaging for users.

This report is supposed to document the integration of Room and Retrofit in Android development, featuring advanced techniques such as data caching, lifecycle-aware components, and dependency injection. Full-cycle activities range from setting up a database and integrating API to testing and deployment strategies.

Working with Databases in Kotlin Android

Overview of Room Database

Room is a persistence library which abstracts SQLite operations to make database access less error-prone and safer. Unlike raw SQLite, Room gives compile-time query validation and seamless integration with Kotlin coroutines and Flow, hence providing type safety and increasing developer productivity. This allowed Weebnet, while using Room, to provide offline access to user data smoothly, thus improving the reliability and satisfaction of users.

Room also provides migration support for changing the database schema without losing data integrity. This is essential for applications whose requirements change with time. Furthermore, Room's integration with other Jetpack components simplifies state management and lifecycle-aware operations.

Figure 1. Weebnet Room Database

Above, the AnimeDatabase class defines the configuration of the database. The entities, such as User and CachedTopAnime, represent the schema, while the DAOs provide the methods for data access. Simplicity and reliability make Room a perfect fit for maintaining the persistent data in an Android application.

The other strong advantage of Room is the possibility to handle schema migrations by increasing the number of the version and providing a migration strategy so the developer can update the database without losing the existing data.

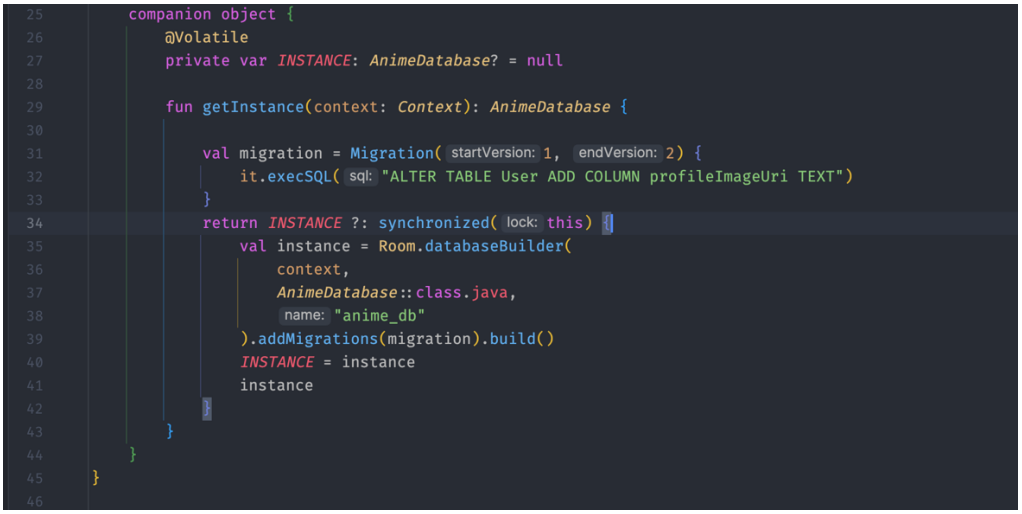

Figure 2. Room Database Migration

This way, it will be non-destructive updates of databases, and compatibility with the older versions of the app will be maintained.

Data Models and DAO

Entities in Room define the database table structure. Each field in the entity represents a column. Any other constraint for each field is described by using annotations. This User entity would be a table for the user's profile.

```kotlin
import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "users")
data class User(
    @PrimaryKey
    val username: String,
    @ColumnInfo(name = "password")
    val password: String,
    @ColumnInfo(name = "profile_image_uri")
    val profileImageUri: String? = null
)
```

Figure 3. User Data Class

The User entity enforces data integrity with the @PrimaryKey annotation, ensuring unique usernames. Optional fields such as profileImageUri provide the flexibility to store more information for the user. This allows for future scalability and maintains simplicity.

In general, DAOs define how the programs access data stored in the database. For example, the UserDao manages the creation, query, and update of users' information.

```kotlin
import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import androidx.room.Update
import com.example.assignment_4.model.User

@Dao
interface UserDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun registerUser(user: User)

    @Query("SELECT * FROM users WHERE username = :username")
    suspend fun getUser(username: String): User?

    @Update
    suspend fun updateUser(user: User)
}
```

Figure 4. User DAO

The registerUser method inserts or updates the user information to avoid the duplication of user data. The getUser method returns details of the user by username, and updateUser updates the records that already exist. All this isolation of database logic really cleans up the codebase for maintainability and testing.

Database Setup and Repository Pattern

The Repository pattern abstracts the data sources and exposes a unified API to the UI layer. In Weebnet, the AnimeRepository consolidates logics for fetching data from Room and Retrofit; thus, the app will adhere to clean architecture.

```
class AnimeRepository @Inject constructor(
    private val apiService: AnimeApiService,
    private val userDao: UserDao,
    private val favoriteAnimeDao: FavoriteAnimeDao,
    private val cachedTopAnimeDao: CachedTopAnimeDao,
    private val cachedUpcomingAnimeDao: CachedUpcomingAnimeDao,
    @ApplicationContext private val context: Context
) {

    companion object {
        private const val CACHE_EXPIRATION_MINUTES = 60
    }

    suspend fun getTopAnime(): List<Anime> {
        return withContext(Dispatchers.IO) {
            val cachedList =
                cachedTopAnimeDao.getCachedTopAnime().map { list -> list.map { it.toAnime() } }
                    .first()
            val lastCacheTime = cachedTopAnimeDao.getCachedTopAnime().map { list ->
                if (list.isNotEmpty()) list.first().cachedAt else 0L
            }.first()

```

Figure 5. Anime Repository

By encapsulating database and API logic, the repository simplifies interaction between the data and UI layer. For example, a call to getTopAnime prefers returning data from the cache to save network usage, and uses the API only when it needs to:

The repository also handles users, like creating a new user or updating a user profile information:

```
    suspend fun updateUserProfileImage(username: String, profileImageUri: String?) {
        val user = userDao.getUser(username)
        if (user ≠ null) {
            val updatedUser = user.copy(profileImageUri = profileImageUri)
            userDao.updateUser(updatedUser)
        }
    }

```

Figure 6. Update User Profile Image Function

This centralizes the logic, enabling reuse of code and maintaining the same codebase.

User Interface Integration

Weebnet uses Jetpack Compose to power its UI, making great use of its declarative and reactive design to build beautiful and functional screens. The composables automatically recompose when any app state changes occur; there is no need to manually update the UI. For example, the AnimeList Composable dynamically renders a list of favorite anime.

Ensuring that changes in the data in the database get reflected instantly on the screen, this approach makes for much simpler UI logic: only the items visible within the list are rendered, thus enhancing performance. The following code snippet illustrates how the AnimeList composable displays a list of anime stored in favorites:

```
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.wrapContentHeight
import androidx.compose.foundation.lazy.LazyRow
import androidx.compose.foundation.lazy.items
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.navigation.NavController
import com.example.assignment_4.model.Anime

@Composable
fun AnimeList(
    navController: NavController,
    animeList: List<Anime>,
    favoriteAnimeIds: List<Int>,
    onFavoriteClick: (Anime) -> Unit
) {
    LazyRow(
        modifier = Modifier
            .fillMaxWidth()
            .wrapContentHeight()
    ) {
        items(animeList) { anime ->
            AnimeItem(
                anime = anime,
                navController = navController,
                onFavoriteClick = onFavoriteClick,
                isFavorite = favoriteAnimeIds.contains(anime.mal_id)
            )
        }
    }
}
```

Figure 7. Composable AnimeList

The LazyRow component will efficiently render the list, loading only the currently visible items. Each of the items in the list will be a row with an image, title, and a delete button. The onFavoriteClick callback simplifies users' interactions by enabling adding of items to the Favorites list with a single tap. Compose's declarative syntax reduces complexity in your codebase, thereby making it more readable and maintainable.

AnimeDetailScreen composable - supplies detailed information about the selected anime. This screen will contain the big image of anime, title, synopsis, and some other details about the anime with the possibility to add to favorite:

Figure 8. Composable AnimeDetailScreen

The AnimeDetailScreen composable skillfully combines visuals and functionality, allowing an intuitive user experience. Scaffold ensures a consistent layout, providing a top app bar and area for content. This screen is integrated with data from AnimeViewModel to show real-time updates, such as when an anime is added to the list of favorites.

Lifecycle Awareness

It's a kind of programming that is lifecycle aware, hence allowing efficient management of resources and maintaining stability in an application. In Weebnet, the associated UI component's lifecycle would bind long-running operations of the ViewModel to the database query or API call. Consequently, it would prevent memory leaks and ensure operations get automatically canceled once the ViewModel is cleared.

For example, viewModelScope is used to launch coroutines in AnimeViewModel: this scopes all database and API operations safely to the lifecycle of the ViewModel:

Figure 9. AnimeViewModel

In the example, the stateIn function is used to transform a Flow into a StateFlow, for the UI layer to consume the data in a reactive way. The addFavorite function runs in a coroutine asynchronously, updating the database without blocking the UI. These lifecycle-aware practices improve app stability and enhance the user experience.

Another important feature of a lifecycle awareness is handling seamlessly configuration changes. For example, when the user rotates a device, the ViewModel has the current state and so there are no unnecessary reloads or data loss, which is especially helpful on operations involving network calls and database queries, as to reload such data might create delays or redundant processing.

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun HomeScreen(
    navController: NavController, animeViewModel: AnimeViewModel
) {
    val animeList by animeViewModel.animeList.collectAsState()
    val upcomingAnimeList by animeViewModel.upcomingAnimeList.collectAsState()
    val favoriteAnimeList by animeViewModel.favoriteAnimeList.collectAsState()
    val favoriteAnimeIds = favoriteAnimeList.map { it.mal_id }

    Scaffold(
        topBar = {
            TopAppBar(title = { Text( text: "Weebnet") }, actions = {
                IconButton(onClick = {
                    animeViewModel.logout()
                    navController.navigate( route: "login") {
                        popUpTo( route: "main") { inclusive = true }
                    }
                }) {
                    Icon(
                        Icons.AutoMirrored.Filled.ExitToApp, contentDescription = "Logout"
                    )
                }
```

Figure 10. HomeScreen Composable

In the example, a HomeScreen composable that subscribes to state flows exposed by the ViewModel. As this Compose will automatically trigger and reflect changes in data because this minimizes the risks of inconsistency. The lifecycle-aware component will make this robust architecture for the app.

Using Retrofit in Kotlin Android

Overview of Retrofit

Retrofit is a type-safe HTTP client for Android and Java that makes interaction with RESTful APIs quite easy. It can also automatically convert JSON responses into Kotlin objects, saving developers from manually parsing JSON data and thus reducing boilerplate code and minimizing errors. Due to its seamless integration with Kotlin coroutines, Retrofit will handle asynchronous programming nicely, keeping the main thread responsive during network operations.

In the Weebnet application, Retrofit is already set up to work with the Jikan API, which

provides data about anime. It includes the configuration for the base URL and the library used for serialization, Gson. This will keep all the API requests and responses nicely organized.

```
47      private fun getRetrofit(context: Context): Retrofit {
48          return Retrofit.Builder().baseUrl(BASE_URL).client(provideOkHttpClient(context))
49              .addConverterFactory(GsonConverterFactory.create()).build()
50      }
51
52      fun getApiService(context: Context): AnimeApiService {
53          return getRetrofit(context).create(AnimeApiService::class.java)
54      }
```

Figure 11. Retrofit Initialization

This snippet initializes Retrofit with the Jikan API's base URL. GsonConverterFactory takes care of JSON responses deserialization and serialization, while create binds the already defined service interface to the instance of Retrofit.

Retrofits' declarative design abstracts away the complex operation of HTTP into simple function invocations that are easy for a developer to implement features rather than dealing with managing network requests. It works along with coroutine-based architecture and ensures API calls run on a background thread and hence does not freeze the UI.

API Service Definition

AnimeApiService is the interface describing endpoints that will fetch information about anime. Every function utilizes Retrofit annotations - @GET is used for specifying the HTTP method while @Path for dynamic values injection in URL. All these mean that the service is going to be as expressive and concise as possible.

```
8   interface AnimeApiService {
9       @GET("seasons/upcoming")
10      suspend fun getUpcomingAnime(): AnimeResponse
11
12      @GET("top/anime")
13      suspend fun getTopAnime(): AnimeResponse
14
15      @GET("anime/{id}")
16      suspend fun getAnimeDetails(@Path("id") id: Int): AnimeDetailResponse
17  }
```

Figure 12. AnimeApiService

This implementation provides the list of trending and upcoming anime through the getTopAnime and getUpcomingAnime, respectively. The getAnimeDetails retrieves details about specific anime. Every method is set to suspend, which gives it a capability to execute in coroutine context.

For instance, the getAnimeDetails function replaces the {id} placeholder in the URL using the @Path annotation with the ID of the anime provided. It constructs the URL dynamically and

makes it easier to interact with the endpoint. That makes the service reusable and flexible.

Retrofit also provides the possibility of query parameters for things like filtering or pagination. This is not in use for Weebnet but could look like this:

```
12
13      @GET("top/anime")
14      suspend fun getTopAnime(@Query("page") page: Int): AnimeResponse
15
```

Figure 13. Retrofit Query Parameter

Data Models

Data models play an important role in mapping JSON responses to Kotlin objects. These models ensure type safety, allowing the developer to operate on structured data instead of manually parsing it. The Anime class, for example, is a simple mapping that describes the core of the anime data:

```
3   data class Anime(
4       val mal_id: Int,
5       val title: String,
6       val images: Images,
7       val synopsis: String?,
8       val episodes: Int?,
9       val score: Double?
10  )
11
12  data class Images(
13      val jpg: ImageUrl
14  )
15
16  data class ImageUrl(
17      val image_url: String
18  )
```

Figure 14. Anime Data Class

This will create a nested structure that follows JSON hierarchy and will help more when it comes to image URL and other details. To be specific, ImageUrl class holds the URL for anime's image and Anime which binds all the key fields together, such as a title and synopsis.

Now to provide detail about an individual anime, AnimeDetailResponse model:

```
1   package com.example.assignment_4.model
2
3   data class AnimeDetailResponse(
4       val data: Anime
5   )
6
```

Figure 15. AnimeDetail Data Class

These models abstract away the complexities of JSON to let the app interact with structured Kotlin objects. Gson, integrated with Retrofit, will automatically map this JSON response to these models at the time of API calls.

API Calls and Response Handling

Retrofit provides a straightforward way of calling APIs by abstracting all the complex HTTP operations into simple, declarative methods. The AnimeRepository acts as the bridge between the API and the UI in Weebnet, taking care of all API-related interactions. An example is the getTopAnime method, which retrieves the list of trending anime using Retrofit and sends the data to the ViewModel for display in the UI.

```
37      suspend fun getTopAnime(): List<Anime> {
38          return try {
39              val response = apiService.getTopAnime()
40              response.data
41          } catch (e: Exception) {
42              emptyList()
43          }
44      }
```

Figure 16. GetTopAnime Function

Here, the try-catch block ensures that if the API call fails, the app won't crash. If the response is successful, it returns the data property of the AnimeResponse object; otherwise, it returns an empty list for fallback so that the app does not break in case of network disruption.

Another very important API call is the getAnimeDetails call, which returns a detailed version of an anime. This method uses Retrofit's path parameters to create the request URL dynamically:

```
134     suspend fun getAnimeById(animeId: Int): Anime? {
135         return withContext(Dispatchers.IO) {
136             if (NetworkUtils.hasNetwork(context)) {
137                 try {
138                     val response = apiService.getAnimeDetails(animeId)
139                     response.data
140                 } catch (e: Exception) {
141                     e.printStackTrace()
142                     null
143                 }
144             } else {
145                 null
146             }
147         }
148     }
```

Figure 17. Get Anime By ID function

The @Path annotation in the service definition replaces {id} in the endpoint dynamically with the actual anime ID passed. This enables the app to fetch any anime's details by simply passing the ID as an argument.

Caching Responses

This feature is crucial because caching significantly optimizes the performance by ensuring data is available on network outages. To cache API responses from Trending and Upcoming anime, Weebnet relies on Room. With Room caching, the application displays some data even in cases of no Internet or very slow network access. The caching system first identifies entities that describe what information will be cached. The cached schema for Trending Anime, for example, looks something like this:

```
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "cached_top_anime")
data class CachedTopAnime(
    @PrimaryKey val mal_id: Int,
    val title: String,
    val synopsis: String?,
    val image_url: String,
    val episodes: Int?,
    val score: Double?,
    val cachedAt: Long
)
```

Figure 18. CachedAnime Data Class Model

The cachedAt field keeps track of the time when the data was cached, so the app can tell if the cache is still valid. A similar entity is defined for upcoming anime, CachedUpcomingAnime. The DAO provides methods to manage the cached data, including inserting, retrieving, and clearing records:

```
@Dao
interface CachedTopAnimeDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun cacheTopAnime(animeList: List<CachedTopAnime>)

    @Query("SELECT * FROM cached_top_anime ORDER BY cachedAt DESC")
    fun getCachedTopAnime(): Flow<List<CachedTopAnime>>

    @Query("DELETE FROM cached_top_anime")
    suspend fun clearCachedTopAnime()
}
```

Figure 19. CachedTopAnimeDao

getCachedTopAnime returns cached data as a Flow for immediate updates in UI each time the cache changes. cacheTopAnime removes previously set data and adds the new records. clearCachedTopAnime clears all the entries.

Repository combines caching with actual API calls, taking first cached data and refreshing when it is necessary:

```
37      suspend fun getTopAnime(): List<Anime> {
38          return withContext(Dispatchers.IO) {
39              val cachedList =
40                  cachedTopAnimeDao.getCachedTopAnime().map { list -> list.map { it.toAnime() } }
41                      .first()
42              val lastCacheTime = cachedTopAnimeDao.getCachedTopAnime().map { list ->
43                  if (list.isNotEmpty()) list.first().cachedAt else 0L
44              }.first()
45
46              val currentTime = System.currentTimeMillis()
47              val isCacheValid =
48                  TimeUnit.MILLISECONDS.toMinutes( duration: currentTime - lastCacheTime) < CACHE_EXPIRATION_MINUTES
49
50              if (isCacheValid && cachedList.isNotEmpty()) {
51                  cachedList
52              } else {
53                  if (NetworkUtils.hasNetwork(context)) {
54                      val response = apiService.getTopAnime()
55                      val animeList = response.data
56
57                      cachedTopAnimeDao.clearCachedTopAnime()
58                      cachedTopAnimeDao.cacheTopAnime(animeList.map { it.toCachedTopAnime(currentTime) })
59
60                      animeList
61                  } else {
62                      cachedList
63                  }
64              }
65          }
66      }
```

Figure 20. Get Top Anime With Cache Function

This implementation ensures the use of cached data in cases of availability, reducing network use and improving performance. CACHE_EXPIRATION_MINUTES will ensure that the cache won't serve stale data without at least updating to some new information.

Testing

Testing will ensure that AnimeApiService talks to the RESTful API correctly and that the app behaves as expected for various responses. For this, it uses MockWebServer to mock API responses without a live backend. This allows the isolation of the API service for testing, allowing the behavior of each method to be verified under controlled conditions. MockWebServer and a Retrofit service which would point to the mock would be set up; that way, all API calls from tests would go to the fake backend:

Figure 21. MockWebServer Setup

Here, `mockWebServer.url ("/") ` provides a temporary base URL for tests. This URL will be used by a Retrofit builder to forward all API requests to the mock, eliminating actual network calls during tests.

After the tests have been completed, the MockWebServer is shut down, which helps in freeing up resources and prevents port conflicts:

Figure 22. MockWebServer Shutdown

This sets up a clean environment for the test cases, which means side effects are minimized, and one gets reliable results.

The getTopAnime method fetches a list of trending anime from the API. For this test, it first enqueues a mock response in the MockWebServer that will behave like the API. It includes a predefined JSON body:

Figure 23. Get Top Anime Test Function

In this test, the mock server returns a response of 200 OK with the topAnimeResponse JSON. The apiService.getTopAnime() call retrieves the mocked data, and assertions are done to verify the request path, the response is ok, and the data content. For instance, the test asserts that the title of the first anime in the response is "Cowboy Bebop."

getAnimeDetails retrieves detailed information of a specific anime. Testing this method will be very similar to the previous example using MockWebServer to simulate the API response and validate the behavior of the method:

```
@Test
fun `getAnimeDetails should return anime details`() = runTest {
    // Given
    val animeId = 1
    val mockResponse =
        MockResponse().setResponseCode(200).setBody(MockResponses.animeDetailResponse)
    mockWebServer.enqueue(mockResponse)

    // When
    val response = apiService.getAnimeDetails(animeId)

    // Then
    val request = mockWebServer.takeRequest()
    Assert.assertEquals( message: "Request path should be /anime/1", expected: "/anime/$animeId", request.path)
    Assert.assertNotNull( message: "Response should not be null", response)
    Assert.assertEquals(
        message: "Anime title should be 'Cowboy Bebop'", expected: "Cowboy Bebop", response.data.title
    )
}
```

Figure 24. Get Anime Details Test Function

Here, the animeId variable is the ID of the anime that needs to fetch. Here, it verifies that the proper endpoint (/anime/1) has been called and that response data such as anime title and synopsis is properly returned.

The tests depend on predefined mock JSON data, held in the MockResponses object. These responses, similar in structure and content to real API responses, guarantee production-like scenarios of tests:

```
object MockResponses {
    val topAnimeResponse = """
    {
        "data": [
            {
                "mal_id": 1,
                "title": "Cowboy Bebop",
                "synopsis": "In the year 2071 ... ",
                "images": {
                    "jpg": {
                        "image_url": "https://example.com/image.jpg"
                    }
                }
            },
            {
                "mal_id": 2,
                "title": "Fullmetal Alchemist: Brotherhood",
                "synopsis": "Two brothers ... ",
                "images": {
                    "jpg": {
                        "image_url": "https://example.com/image2.jpg"
                    }
                }
            }
        ]
    }
    """.trimIndent()
```

Figure 25. Mock Response

These mock responses are structurally identical to AnimeResponse and AnimeDetailResponse classes for compatibility with service methods. By centralizing mock data, tests remain clean and maintainable.

In general, integrating MockWebServer with JUnit makes sure the tests of AnimeApiService are well performed. By mocking real behavior, these tests will check method correctness and guarantee that an app will work correctly in case of a successful response and in cases of errors. Such testing will guarantee the reliability of the core functionality of the application and enhance its stability in general.

Conclusion

The Weebnet app proves to be a modern Kotlin Android application that embeds Room Database and Retrofit perfectly. The app gives guarantees of efficient local data storage by means of the usage of Room, considering the support of offline access and caching mechanisms. All of this enhances the user experience, especially in cases where the network condition is poor. Using Retrofit keeps the communication with RESTful APIs simple, abstracting complex HTTP operations into declarative, reusable methods. Coupled with Jetpack Compose, the application is well-equipped to provide a dynamically changing and visually stunning UI that reacts to every data change.

Lifecycle-aware programming ensures resource management with ViewModel and LiveData to efficiently avoid memory leaks while preserving the state over configuration changes. The repository pattern provides an architecture that can be used to maintain cleaner code: it encapsulates data logic for a meaningful data representation and provides it as the single source of truth in the application. The different error handling strategies enhance app stability to let unexpected scenarios, such as API failures, be appropriately handled by the application itself.

Caching by Room minimizes dependency on the network and thus greatly improves performance by serving cached data when appropriate. It also does extensive testing for API interactions using MockWebServer, which assures the reliability and correctness of this under different conditions. This is how these technologies and design principles together help build a scalable, maintainable, and user-friendly application.

Recommendations

1. Advanced caching: It already caches top and upcoming anime, and extending this into the other features, such as the anime information detail, will even further prolong offline usability. Consider more granular cache expiration strategies dependent on the specific API endpoint called.
2. Search Functionality: Incorporate a search feature to enable the user to find an anime of their choice either by title or genre. This would be greatly enhanced by adding a search endpoint to AnimeApiService and adjusting the UI to accommodate this.
3. Push Notifications: Let the users know when new anime is released, or if there is an update in their favorite list, or according to their preferences. This could be implemented using Firebase Cloud Messaging.
4. Localization: support for many languages, thus widening their targeted audience. Jetpack Compose has made the procedure for this quite easy - basically just using Android's inter nationalization in its implementation.
5. Enhanced UI Customization: While the current UI follows the best practices of Material Design, it would be better to add more to the branding. Think about adding animations and transitions for enhanced user engagement and interaction.
6. Analytics Integration: Integrate third-party services like Firebase Analytics. This will help in collecting information about user interactions, understand behaviors, and track usages of features for future development decisions.

References

1. Android Developers Documentation - https://developer.android.com/
2. Retrofit Library Documentation - https://square.github.io/retrofit/
3. Kotlin Coroutines Guide - https://kotlinlang.org/docs/coroutines-overview.html
4. Room Persistence Library - https://developer.android.com/jetpack/androidx/releases/room
5. MockWebServer Documentation - https://github.com/square/okhttp/tree/master/mockwebserver
6. Material Design for Android - https://m3.material.io/
7. Gson Library Documentation - https://github.com/google/gson
8. Jikan API Documentation - https://jikan.moe/
9. Mockk Documentation - https://mockk.io/
10. Kotlin Flow and StateFlow Guide - https://kotlinlang.org/docs/flow.html

Appendices

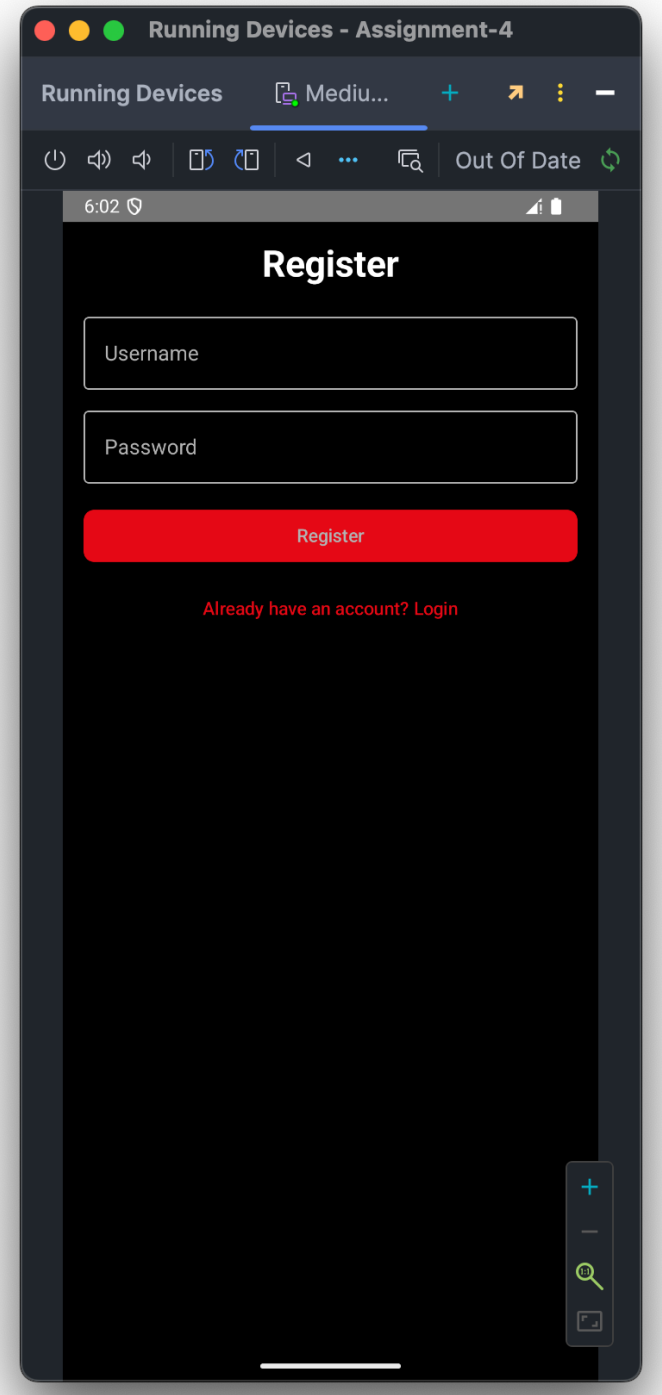

Figure 26. Register Page

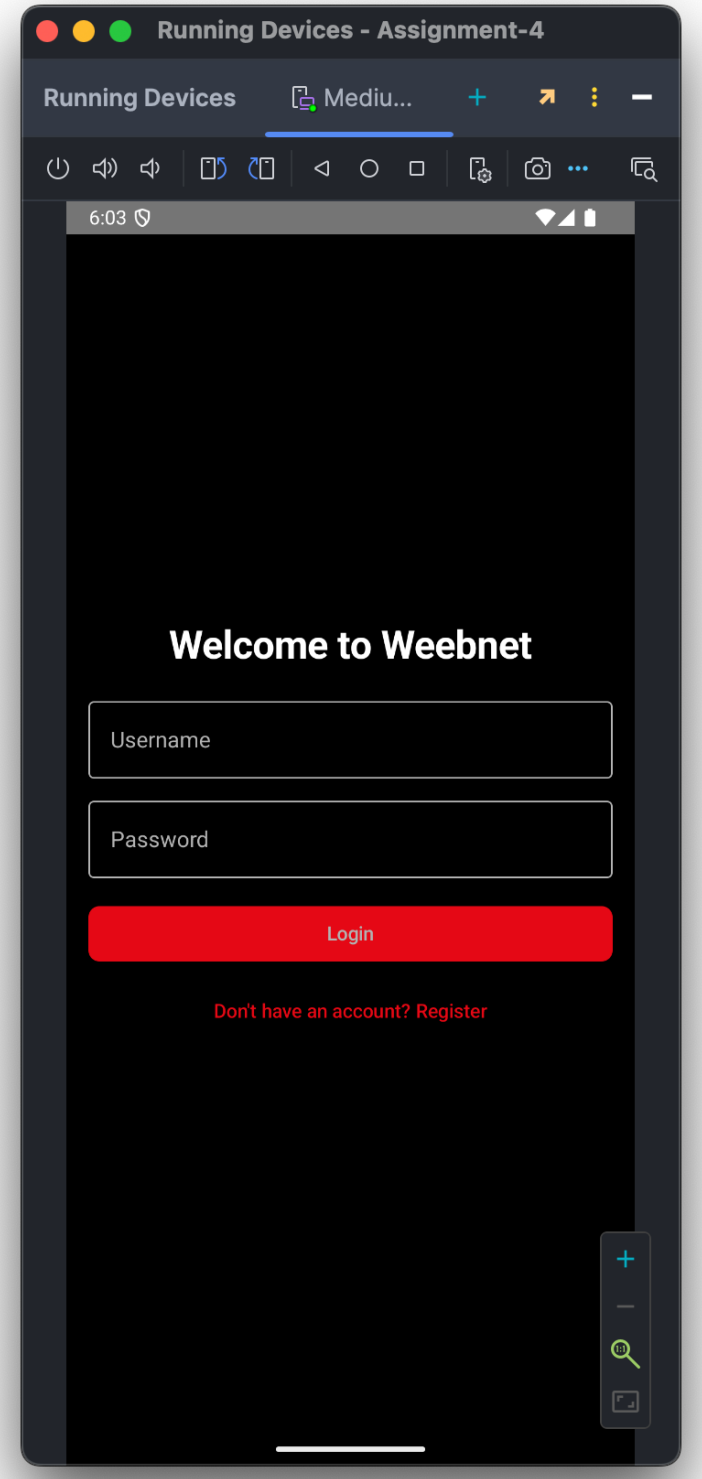

Figure 27. Login Page

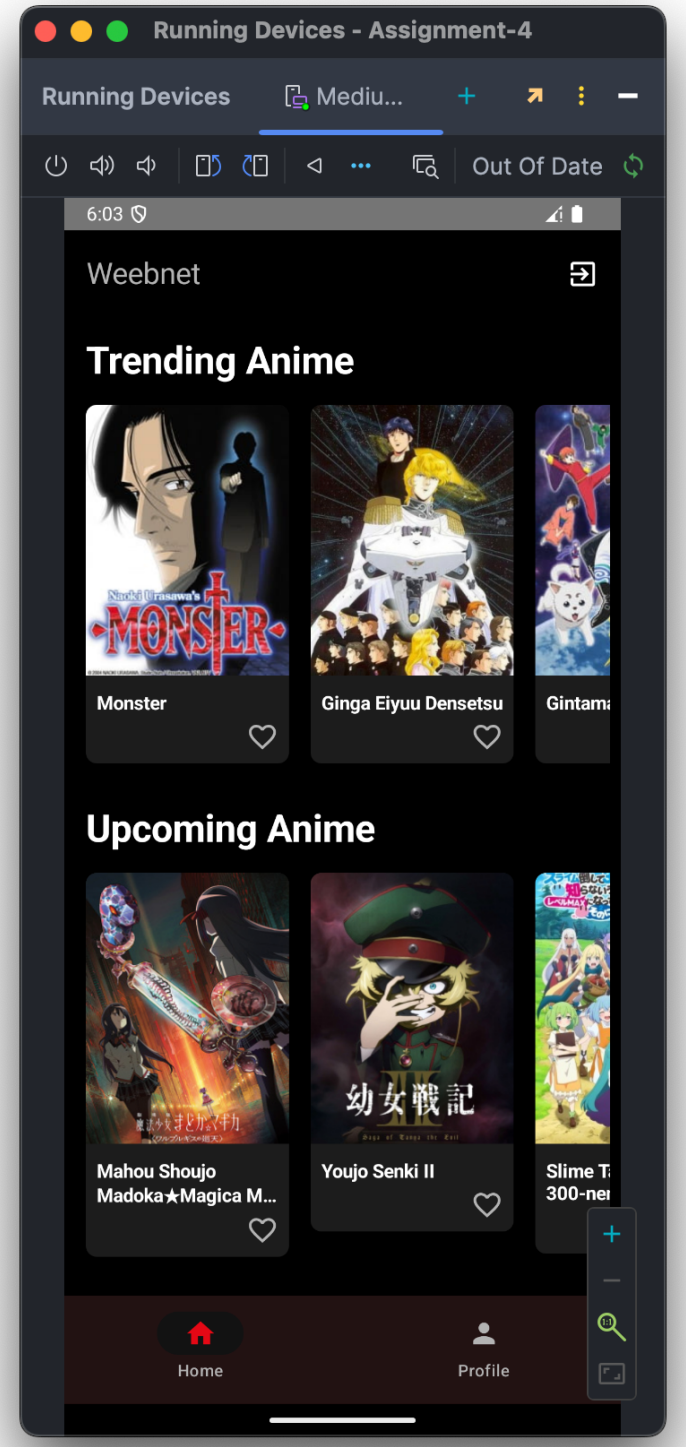

Figure 28. Home Page

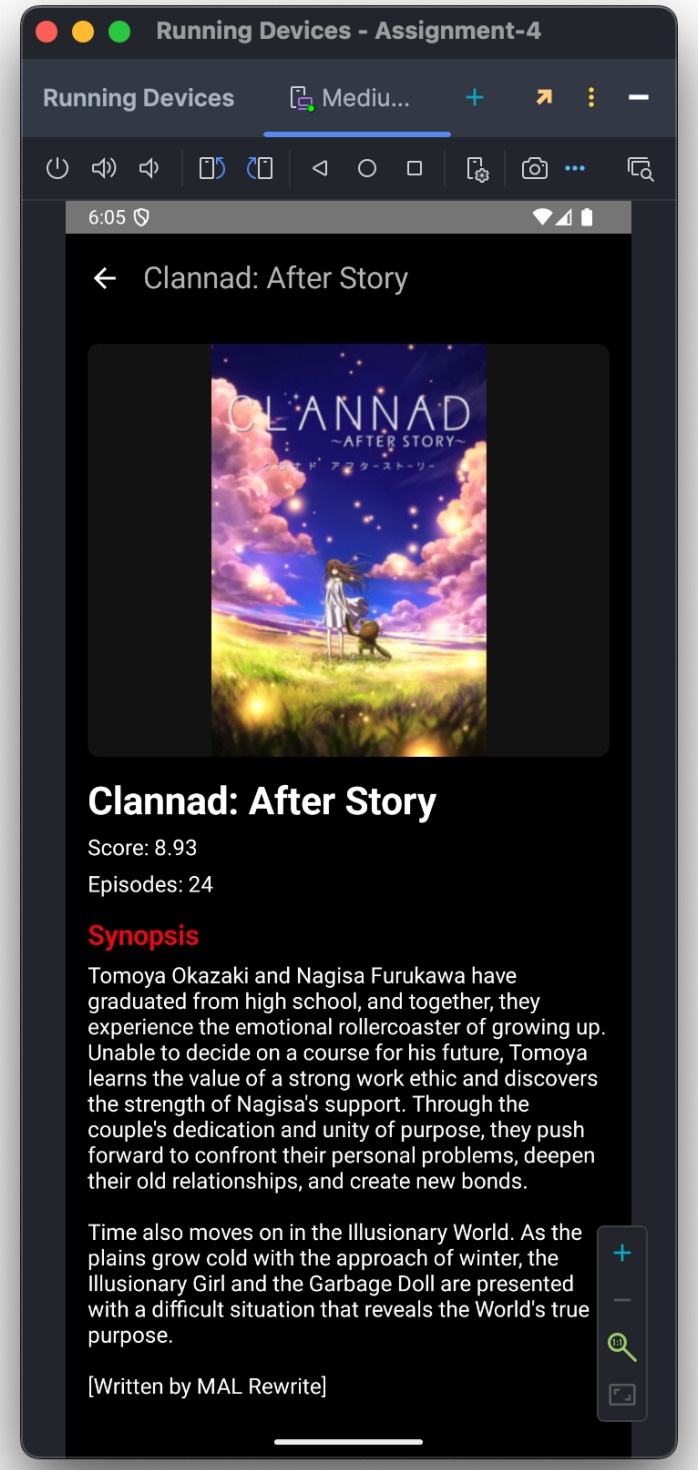

Figure 29. Detailed Page

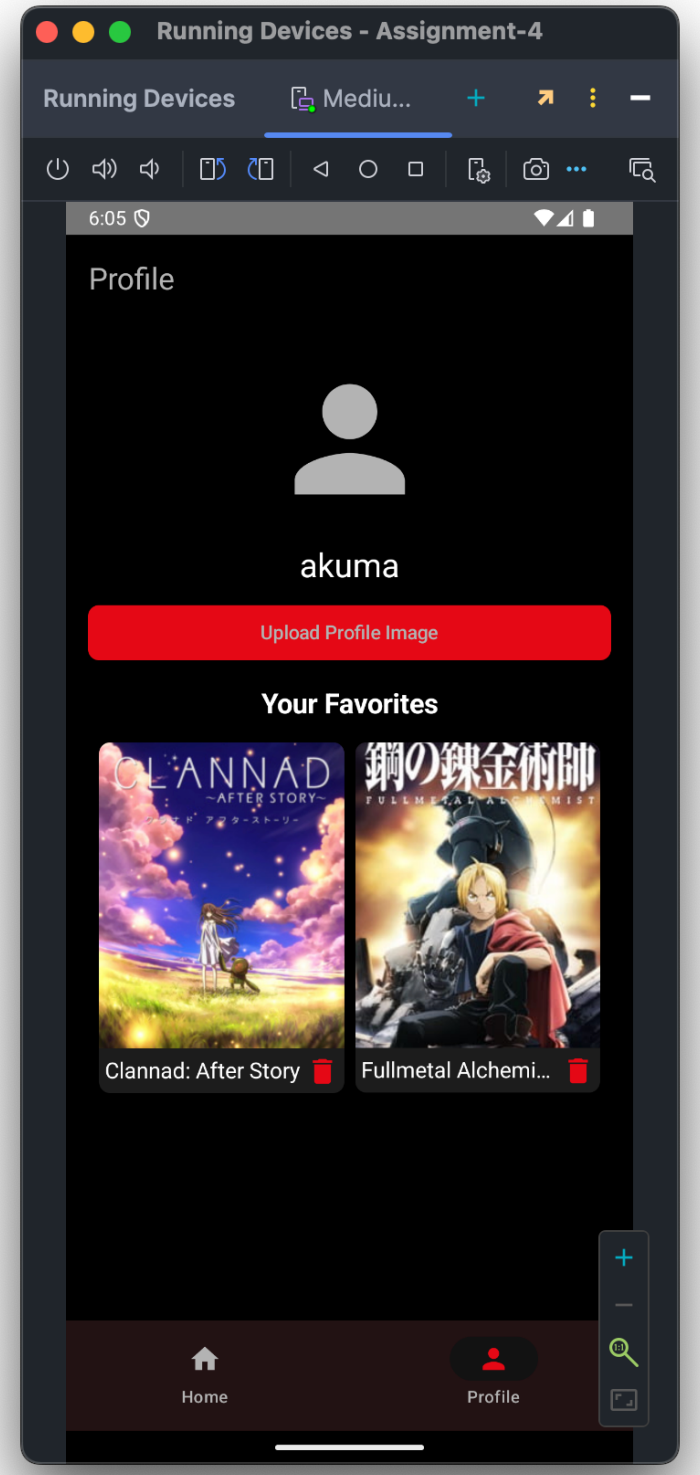

Figure 30. Profile Page

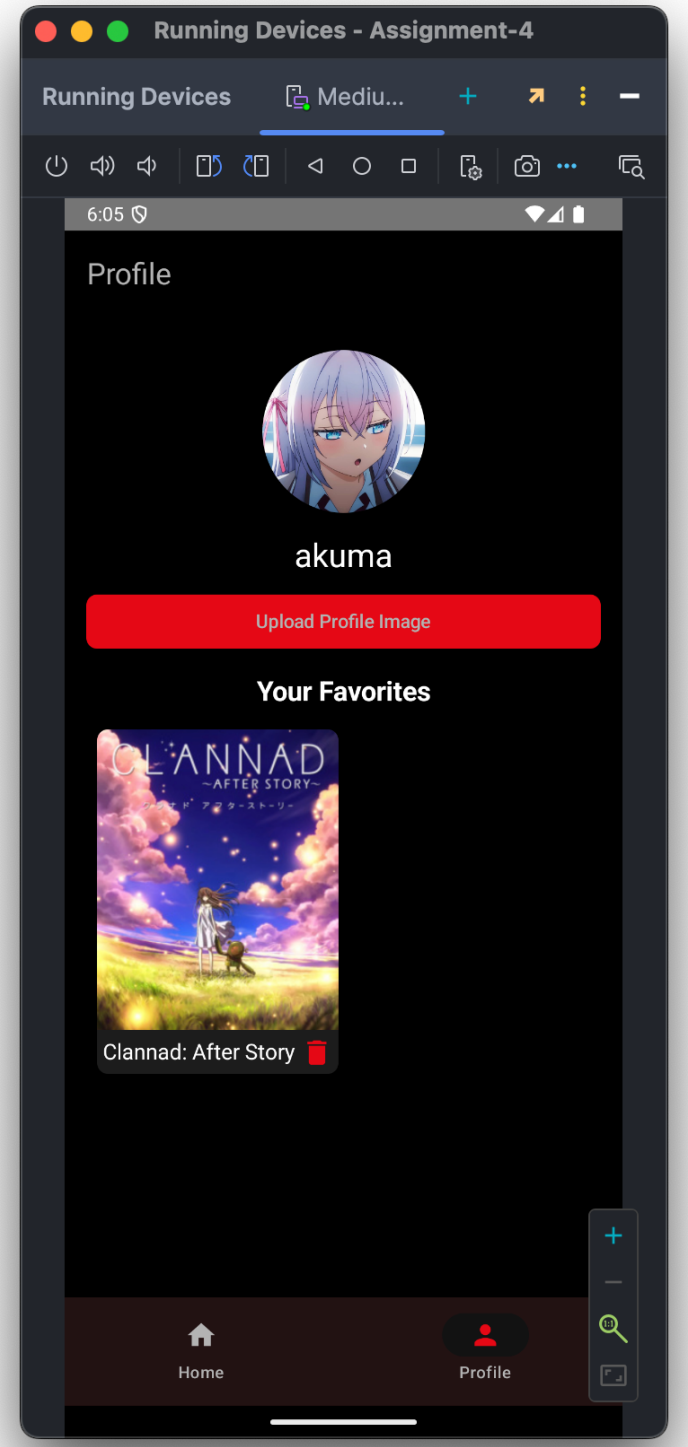

Figure 31. Updated Profile Page with Profile Image