Mobile Programming Project Documentation A.Y. 2025-2026

Project name: QuestMaster

Group: Bulletin Bills

Members:

•857321 Andrea De Luca

•938107 Viktor Pannagl

Table of Contents

1. Introduction

QuestMaster is an Android application designed to transform real-world exploration into an interactive adventure. Its primary goal is to provide a platform where users can both create and participate in custom "quests"—treasure hunts or step-by-step routes based on geolocation.

The idea stems from the desire to combine the digital world of gaming with the physical experience of urban and natural exploration. In an era where interactions are increasingly screen-mediated, QuestMaster encourages users to go outside and discover points of interest, parks, or hidden corners of their city in a fun and engaging way. It addresses the common desire for novel outdoor activities by simplifying the complex task of organizing a treasure hunt.

Unlike simple navigation apps, QuestMaster adds a layer of gamification, allowing users to create narratives, challenges, and rewards tied to physical locations. The platform manages quest progression by verifying the completion of stages via GPS and displaying instructions, leaving users free to immerse themselves in the game and the joy of discovery.

2. Architecture

The development of QuestMaster is founded on the Model-View-ViewModel (MVVM) architectural pattern. This pattern is officially recommended by Google for building robust, scalable, and maintainable Android applications because it promotes a clean separation of concerns.The data and event flow follows this structure:

View (UI Layer) <--> ViewModel (Domain Layer) <--> Repository (Data Layer) <--> Data Sources

•View (UI Layer): This layer, composed of Activities and Fragments, is responsible for rendering the user interface and capturing user

interactions. It knows nothing about the business logic or where data comes from. It simply observes the ViewModel for data changes and notifies the ViewModel of user events. In our project, this includes HomeActivity, StartActivity, FeedFragment, and QuestCreateFragment

.•ViewModel (Domain Layer): The ViewModel acts as the bridge between the UI and the application's business logic. Its primary role is to own and manage UI-related data in a lifecycle-conscious way. This means that data held by the ViewModel (e.g., the list of quests fetched from the server) automatically survives configuration changes like screen rotations, preventing data loss and redundant network calls. The ViewModel exposes data to the View via LiveData and has no direct reference to any Activity or Fragment, making it highly testable and reusable.

•Model (Data Layer): This layer is represented by the Repository Pattern. The Repository module is the single source of truth for all application data. It abstracts the data sources from the rest of the app, meaning the ViewModel doesn't know (or need to know) whether the data is coming from a remote server, a local database, or an in-memory cache. This modularity is key to our architecture, as it allows us to modify or add new data sources without altering the core business logic. Our data structures are defined in simple model classes like Quest.java, User.java, and QuestLocation.java.

This architecture makes the codebase easier to debug, test, and scale over time.

3. Design and Implementation

The app's design adheres to Material Design principles, ensuring a clean, intuitive, and modern user interface that feels at home on the Android platform.UI Layer

The user interface is implemented using Android's traditional View system. The application is built around a Single-Activity Architecture, where HomeActivity serves as the main entry point and container for various Fragment destinations. This modern approach is more memory-efficient than using multiple activities and creates a more cohesive user experience.

•Navigation: User navigation is orchestrated by the Jetpack Navigation Component. This tool allows us to define all navigation paths and transitions within a visual graph, making the application's flow easy to understand and manage from a central location.

•Layouts & Views: Screens are structured as Fragments (e.g., FeedFragment, QuestCreateFragment) with their corresponding layouts defined in XML. For displaying dynamic lists of content, such as the quest feed, we use RecyclerView. This component is highly optimized for performance, ensuring smooth scrolling even with large datasets by recycling and reusing views. Custom adapters, such as QuestAdapter and UserAdapter, are used to bind our data models to the views displayed in the list.

Domain Layer

This layer is the logical core of the application. The ViewModels are responsible for preparing and managing all data required by the UI. They fetch data from the Repository and expose it as observable LiveData

.•Unidirectional Data Flow (UDF): We follow a UDF pattern where data flows in one direction (from the ViewModel to the UI) and events flow in

the opposite direction (from the UI to the ViewModel). When the data inside a LiveData object changes (e.g., new quests are loaded), it automatically notifies its observer (the Fragment), which then updates the UI. This reactive model makes the app's state predictable and far easier to debug.

Data Layer

The Data Layer is built on the Repository Pattern, establishing a single source of truth.

•Repository: The QuestRepository, for example, provides a clean API for the domain layer (e.g., getQuests(), saveQuest(quest)).

•Data Sources: Internally, the repository manages calls to two types of data sources:

1.Remote Data Source: This component is responsible for all network communication. It interacts with our backend services on Firebase (Firestore) to save and retrieve quests, user profiles, and other dynamic data.

2.Local Data Source: We plan to use the Room persistence library to implement a local database. This will serve as a cache for data fetched from the network, enabling fast data access and providing a basic offline mode where users can still view previously loaded content. The repository will contain the logic to decide whether to fetch fresh data from the network or serve it from the local cache.4. FeaturesAuthentication and RegistrationAccess to the app is managed by Firebase Authentication. The StartActivity orchestrates the entire login and registration flow. Users can:

•Register with an email and password.

•Log in with their existing credentials.

•Use Google Sign-In for a faster, more streamlined authentication experience.Quest CreationAuthenticated users can create new quests via the QuestCreateFragment. This flow is centered on an interactive map powered by the Google Maps SDK.

•Visual Design: Users can visually design their adventure by tapping on the map to place waypoints.•Stage Definition: Each waypoint represents a stage in the quest, defined by the QuestLocation model. Creators can enrich each stage with a title and a descriptive clue or puzzle.

•Saving: Once the quest is complete, the entire Quest object is saved to Firebase Firestore, instantly making it available for other users in the community to discover and play.

Quest Participation

The FeedFragment displays a list of available quests for users to join.

•Progression: The app displays the first stage on the map with its clue. By using the ACCESS_FINE_LOCATION permission, the app tracks the user's GPS position.

•Completion: When a user gets within a predefined radius of a waypoint, the app marks that stage as "completed" and automatically reveals the next one, creating a seamless sense of progression until the adventure is finished.

•Interaction: Users can leave comments on quests they've played, fostering a sense of community.

User ProfileA dedicated profile section allows users to:

•View and manage the quests they have created.•

Keep track of the quests they have completed.•

Manage account settings and log out.

5. Testing

To ensure code quality, a multi-layered testing strategy is essential:

•Unit Tests: These are used for ViewModels and Repositories. Using JUnit and a mocking library like Mockito, we can verify the business logic in complete isolation from the Android framework. For example, a unit test can confirm that a ViewModel calls the correct method on its Repository dependency when a user action is simulated.

•Integration Tests: These tests verify the interaction between different components. For example, we will write integration tests to confirm that our Room database queries for caching are working correctly.

•UI Tests: Using the Espresso framework, we can write automated tests that simulate full user journeys. These tests run on a device or emulator and assert that the UI behaves as expected. For instance, a UI test could simulate a user logging in, navigating to the quest creation screen, filling in the details, and successfully saving the quest, verifying that the correct UI state is shown at each step.

6. Future Developments

To further enrich the QuestMaster experience, we plan to explore the following features:

•Private Quests and Sharing: Allow users to create private quests that can only be shared with friends via a unique link or code.

•Multimedia Clues: Enable creators to upload images, audio, or video as clues for quest stages, using Firebase Storage to host the media.

•Leaderboards and Points: Introduce a scoring system based on quest completion time or the number of quests finished to encourage friendly competition.

•Improved Offline Mode: Allow users to download entire quests, including map data and all clues, enabling a full offline gameplay experience in areas with poor connectivity.

•Augmented Reality (AR) Integration: Use ARCore to display clues, 3D models, or navigational aids in the real world through the device's camera, creating a more immersive and futuristic adventure.7. Appendix (Technologies Used)

•Languages: Java, Kotlin

•Architecture: MVVM (Model-View-ViewModel)

•UI: Android View System, XML Layouts, Material Design 3

•Navigation: Jetpack Navigation Component

•Asynchrony: Kotlin Coroutines and Flow

•Networking and Backend:

•Firebase Authentication for user management

•Firebase Firestore for real-time data storage

•Firebase Storage (for future multimedia files)

•Local Database: Room (for caching and offline support)

•Maps: Google Maps SDK for Android

•External Libraries

:•Glide for efficient image loading and caching

- Google Play Services (Auth, Maps)
- Material Components for Android