1. Manual Code Review

1.1. Front End Review

A more detailed review of comments by file is provided here.

Pros:

- Modularity & Clarity:
  - The code is generally well-structured with clear separation between activities and composable functions.
  - Naming conventions are generally descriptive and consistent, aiding the readability of the code.
- Good Practices:
  - Proper use of Jetpack Compose makes the UI components concise and declarative.
  - Scoped coroutines and features like Parcelize are well used.
- User Experience:
  - The UI flows in a straightforward fashion and provides clear navigation between screens.
  - Composable previews and test tags are in their appropriate places.
- Error Handling & Logging:
  - Most API calls and user actions include logging, which is helpful for debugging.
  - Basic error handling is present in most API responses for most scenarios.

Cons:

- Code Duplication:
  - Significant overlap between components in UserSettingsActivity and PreferencesActivity, particularly in the UserSettingsComposables object (e.g., identical region lists and PreferenceInput functions).
- Error Handling:
  - Several try-catch blocks catch generic exceptions (`catch(e: Exception)`) without targeting specific failures, which can mask issues and complicate debugging.

    - Some API-related error responses are simply logged without providing the user any feedback.

- Non-null Assertions:
    - There is frequent use of the non-null assertion operator (`!!`), particularly when retrieving extras (e.g., `intent.getStringExtra("DiscordId")!!`) or headers from responses. If these values are missing, the app will crash. It would be better to use safe calls or check for null explicitly.
- State Management
    - UI state is managed directly within Activities, which can become unwieldy as the app scales. Using a ViewModel and adopting an MVVM pattern could improve separation of concerns and state preservation across configuration changes.
- Magic Numbers & Hard-Coded Values
    - The use of arbitrary dimension values (e.g., font sizes, offsets, widths) directly in the UI code can hinder future theming or style adjustments. Declaring these values as constants or using a centralized theme would improve maintainability.

Conclusion

The frontend code is neat, readable, and largely follows modern Android development practices. With improvements such as consolidating duplicate components, refining error and null handling, and centralizing styling, the codebase will become even more robust and maintainable. Overall, it scores an 8/10.

1.2. Back End Review

AdminController.ts	In the function *`createAdmin`*, the code should probably check if the user already has an admin record. This might allow duplicates unless DB constraints prevent it. Edit: The mocked test suite for this file lacks documentation. Potential coverage issue [for other files too].

AuthController.ts	Clear documentation, and a nice format. Variable and function names are understandable and consistent. Functions are in reasonable length. Edit: No mocked test for certain error catch recognized by JEST [although did see tests for some]
GameController.ts	The code is well-structured and easy to follow, with meaningful variable names and consistent error handling across all endpoints. CRUD operations are implemented using TypeORM, and responses follow standard HTTP conventions.
GroupController.ts	The functions in the code consider lots of edge cases. Not much comment on the functions `addDiscordUserToGuild` & `getGroupMembers`, but the names are descriptive enough so don't matter. Other functions have clear comments on why the code exists.
MatchmakingController.ts	Not much comment on the functions, but all of the code is clean with meaningful function names and variable names. The code is nice and well-structured and in fine length. Edit: unauthorized error checking is not covered by any test. in checkMatchmakingStatus(), "time-out" case was never covered by any test.
PreferencesController.ts	Not much comment on the functions, but all of the code is clean with meaningful function names and variable names. One minor issue: in `updatePreferences`, the function does not check req.params.id is NaN before calling `parseInt`.
ReportController.ts	The code is well-organized, with a clear separation of responsibilities across CRUD operations and consistent error handling using appropriate HTTP status codes. Each function performs a single responsibility efficiently, with no signs of design smells or unnecessary complexity. Error handling is implemented correctly across all endpoints using appropriate status codes and informative messages.
UserController.ts	The code is well-structured, using clear variable names, consistent error handling, and appropriate HTTP status codes. The code is efficient, avoids redundancy, and follows good design practices with no noticeable smells. Input data is assumed to be well-formed but could benefit from additional validation.
MatchmakingService.ts	Edit: some of the variable and constant names are way too long, and it's becoming difficult to read.

2. Manual Test Review

a) Test completeness (all exposed api tested, main use case, error/edge case, correct assertion)

9.5/10

All API endpoints are covered. A few function tests and error checks are missing but overall complete. See the detailed breakdown below:

b) Test implementation matches requirement and design

9.5/10

[For the mocked tests] They do cover the scenarios of external failure with Discord or the database for all routes. Most of the lines for catching network 500 errors are covered in the mocked test suites. Test documentation is missing in AuthController.test.ts, and each mocked test is missing mocked behaviour as part of the test spec. [For unmocked test] The tests have sufficient comments explaining all input and expected outcomes and behavior. The test suite is organized into logical groups, with detailed comments explaining each step. Some error handling is implemented, though not for all exceptions. This is expected in unmocked tests.

c) Test well structured

10/10

Test cases are organized into no-mocked, mocked, and non-functional categories. Each category is then split into tests for each purpose.
The Mocked section is well structured, and each test case only tests one failure scenario at a time. The purpose of each test is also clear.

d) Tests are comprehensive (good coverage)

9/10.

Taking into account of their justification for JEST low coverage, if it's valid, then tests are overall comprehensive. There are a few functions and error catch statements not covered, as mentioned in the detailed breakdown in part a.

e) Non-functional well tested

10/10

Tests for verifying endpoint security clearly address non-functional requirements. It gives assertion to checks that unauthenticated requests to sensitive routes are correctly rejected with a 401 status.

f) All backend Tests run automatically

10/10

The git action includes all the backend tests and effectively sets up a secure SSH connection, deploys code to an isolated test environment, and runs backend tests using Docker Compose. It also uses commit-specific directories and includes debugging and verification steps.

Review of each file:

Non-functional Test:

EndpointSecurity.test	The test is a well-structured test suite for verifying endpoint security. It gives assertion to checks that unauthenticated requests to sensitive routes are correctly rejected with a 401 status.
ReportReasonLength .test	The test is well-organized, with clear expectations and straightforward input that's easy to follow.

Without-Mocks Test:

AdminController.test	Sufficient comments explaining all input and expected outcomes and behavior. The test suite is organized into logical groups, with detailed comments explaining each step and the purpose of every object used in the testing process.
GameController.test	Sufficient comments explaining all input and expected outcomes and behavior. The test suite is organized into logical groups, with detailed comments explaining each step and the purpose of every object used in the testing process. All error cases are included for the game API.

GroupController.test	The test suite includes thorough comments explaining all inputs and expected outcomes. All test cases have detailed explanations for each step and the purpose of every object involved.
PreferencesController.test	The test is clear and easy to follow, with both successful and error scenarios well covered. Comments are provided throughout, and the expected behaviors are documented.
UserController.test	The test is clear and easy to follow, with both successful and error scenarios well covered. Comments are provided throughout, and the expected behaviors are documented.

Mock Test:

AdminController.test	Tests are documented, and external failure scenarios are all tested (comprehensive). The inputs of test documentation should describe what is fed to the function, not the exact scenarios. Also mocked scenario is missing, but it could be guessed from the description.
AuthController.test	Test documentation is missing, but test cases cover most external failure scenarios. One mocked test did not trigger error-catch in handleRegister().
GameController.test	Tests are documented, and external failure scenarios are all tested (comprehensive). The inputs of test documentation should describe what is fed to the function, not the exact scenarios. Also mocked scenario is missing, but it could be guessed from the description.
GroupController.test	Tests are documented, and external failure scenarios are all tested (comprehensive). The inputs of test documentation should describe what is fed to the function, not the exact scenarios. Also mocked scenario is missing, but it could be guessed from the description. Backend test suite (mocked & not mocked) did not cover functions `addDiscordUserToGuild` & `getGroupMembers`
MatchmakingController.test	Tests are documented, and external failure scenarios are all tested (comprehensive). The inputs of test documentation should describe what is fed to the function, not the exact scenarios. Also mocked scenario is missing, but it could be guessed from the description. In initiateMatchmaking, unauthorized scenario error catch is never tested. In checkMatchmakingStatus, the “time-out” case is never covered by any test.
PreferencesController.test	Tests are documented, and external failure scenarios are all tested (comprehensive). The inputs of test documentation should

	describe what is fed to the function, not the exact scenarios. Also mocked scenario is missing, but it could be guessed from the description.
ReportController.test	Tests are well documented except missing “mocked behavior” line. Most external failure scenarios are covered.
UserController.test	Tests are documented, and external failure scenarios are all tested (comprehensive). The inputs of test documentation should describe what is fed to the function, not the exact scenarios. Also mocked scenario is missing, but it could be guessed from the description.

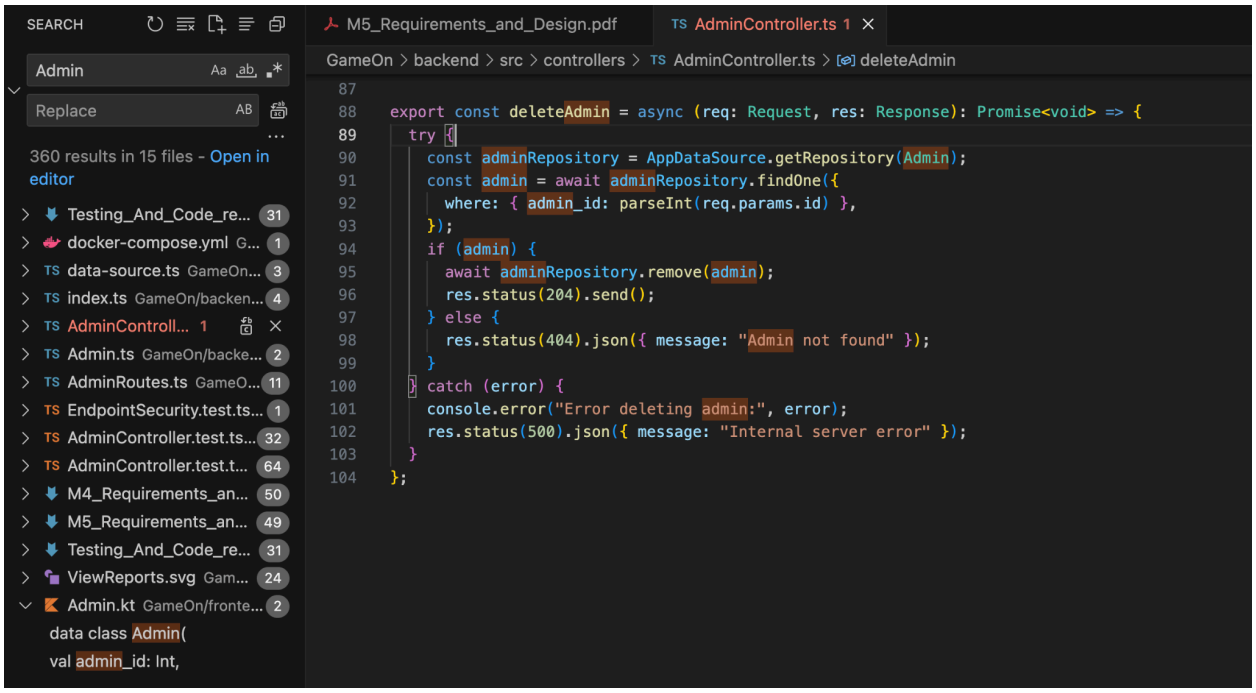
3. Automated Code Review
    a) Codacy runs with the required setup: 10/10

    Codacy does indeed run, and I can see that it’s working from the group’s Codacy page. 10/10

    b) All remaining Codacy Issues are well-justified: 1/10

    Based upon the group’s documentation, they seem to have focused more on writing tests than fixing Codacy problems. While their document says 576 errors, it appears from the updated Codacy numbers on the website that the faulty errors were resolved. As such, they continue to have 60 issues of varying severity, including critical. As a result, while it is understandable that focusing on writing tests was a priority, it does not fully justify the inattention to fixing seemingly any Codacy errors, and a 1/10 for time prioritization is generous.

4. Fault: Admin module design in the system
    - It’s unclear how the Admin class works in the system. In the backend, there are routes and functions for handling admin-related functionality. However, in the front end, the Admin class is declared but never used. There’s nowhere in the front-end code that creates or deletes an Admin. Since only the Admin handles reports, it is unclear how this would happen in the real system.
    - Furthermore, due to the ambiguity above, there is no privilege check in the backend Admin endpoint, before making admin-related changes (create & delete). This is a serious problem, as anyone could make themselves admin and ban other users.



Addition (Detailed Frontend Comments)

Front End - Activities and Composables

File	Pros	Cons
AuthActivity.kt	• Concise and focused on authentication flow. • Uses LoadingScreen composable clearly.	None.
BannedActivity.kt	• Very short and self-contained. • Simple, clear composable for “banned” UI.	None.

ListReportsActivity.kt	• Uses composables (ReportTitle, ReportList) effectively. • Logical navigation flows.	• onCreate is long; setContent could be broken into smaller functions. • ViewExistingGroups composable is lengthy and had compile issues.
LoginActivity.kt	• Straightforward usage of Custom Tabs. • Minimal and focused code.	None.
MainActivity.kt	• Clear separation of UI components (MainContent, FindGroup, ReportsSection). • Logical polling and state updates.	• onCreate is lengthy (multiple lifecycle launches). • Uses magic numbers for sizes.
PreferencesActivity.kt	• Clearly fetches game list and user preferences. • Uses PreferenceInput composable to reduce code duplication.	• Takes many parameters in composables. • Shares duplicated code with UserSettingsActivity (e.g., region list, PreferenceInput).
ReportsActivity.kt	• Reports composable is concise and self-contained. • Preview functions work well.	• onCreate is very long. • High parameter count in Reports composable.
StartupActivity.kt	• Extremely short and focused solely on checking login status.	None.
UserSettingsActivity.kt	• Provides basic settings functionality with header and footer.	• onCreate is overly long and hard to maintain. • Duplicate region list and PreferenceInput code already found in PreferencesActivity. • Violates DRY principle.

ViewGroupActivity.kt	• Good navigation and clear composable structure (GroupMembers, GoToDiscord, MainContent).	• Uses a deprecated method. • GroupMembers error handling is unclear (errorMessage condition always false). • Some redundant string templates.
ViewReportsActivity.kt	• Neatly formatted with clear composable sections (ReportDetails, InputReadable).	None.
UI Theme Files & Other Composables	• Overall theming is consistent; composables like Avatar, LoginButton, Logo, ReportButton, ReportTitle are clear.	• Some composables (e.g., DropdownInput, Header) have long inline code. • Hard-coded “magic numbers” in styling.

Front End - API Interfaces

File	Pros	Cons
AuthApi.kt	• Clear endpoints for login, Discord callback, register, and logout. • Straightforward Retrofit interface.	None (error handling is expected at the caller level).
GamesApi.kt	• Simple and concise endpoint to fetch games.	None.
GroupsApi.kt	• Provides endpoints for getting group members and group URL.	• Inconsistent path formatting (mix of relative/absolute paths).
MatchmakingApi.kt	• Well-defined request/response models. • Clear separation of initiate and status check endpoints.	None significant.
PreferencesApi.kt	• Covers creation, fetching, and updating of preferences.	• Inconsistent use of leading slashes in endpoint paths.

ReportsApi.kt	• Defines endpoints for fetching, creating, and resolving reports.	• createReport returns a generic type (Response<Any>), reducing clarity.
UsersApi.kt	• Simple interface to fetch user groups based on Discord ID.	• Default parameter value ("session") may be non-intuitive. • Could use safer response handling.

Front End - API Methods

File	Pros	Cons
Auth.kt	• Implements login, callback, registration, and logout flows. • Uses Intents for navigation.	• Heavy use of non-null assertions (!!) when retrieving headers or extras. • Error handling is minimal (simply returns on unexpected status).
Games.kt	• Straightforward API call to fetch games with logging.	• Lacks try/catch around network exceptions. • Returns an empty list on failure, which might hide errors.
Groups.kt	• Uses try/catch to handle network errors and returns empty list on failure.	• Generic error logging could be more descriptive.
Matchmaking.kt	• Clearly maps backend status messages to local status strings. • Handles 404 as “not_in_progress.”	• Return logic could be clearer for unknown statuses.
Preferences.kt (API methods)	• Functions for create, update, and fetch are consistent.	• Some functions rely on try-catch blocks that return entire blocks, making returns less clear. • Heavy use of non-null assertions.

Reports.kt (API methods)	• Separates report submission, fetching, and resolution well.	• Force-unwrapping responses (!!) in getReports and getReportById can lead to crashes. • Error responses are only logged.
SessionDetails.kt	• Manages session data effectively with SharedPreferences. • Provides convenient helper functions.	• Stores sensitive data in plain SharedPreferences.
Users.kt (API methods)	• Simple method to fetch user groups.	• Uses force-unwrapping (!!) on the response body. • Could check for null discordId more safely.
Api.kt	• Straightforward Retrofit instance creation with custom Gson.	• Always creates a new Retrofit instance instead of caching it. • Requires explicit initialization (init(context)).
PersistentCookieJar.kt	• Implements persistent cookie storage using SharedPreferences. • Converts cookies to/from strings simply.	• Force-unwrapping in cookie parsing can lead to crashes if parsing fails.