

# Raport techniczny aplikacji Literaki

Stan wykonania, architektura i integracja Android MVP

Projekt: Paweł Szczeszek, Bartosz Gabruk

25 maja 2026

## Spis treści

<b>1</b>	<b>Podsumowanie wykonanych prac</b>	<b>2</b>
1.1	Najważniejsze rezultaty . . . . .	2
<b>2</b>	<b>Zakres funkcjonalny MVP</b>	<b>2</b>
<b>3</b>	<b>Stos technologiczny</b>	<b>3</b>
<b>4</b>	<b>Architektura aplikacji</b>	<b>3</b>
4.1	Warstwa UI . . . . .	3
4.2	Nawigacja . . . . .	3
<b>5</b>	<b>Integracja z backendem</b>	<b>4</b>
5.1	Obsłużone endpointy . . . . .	5
5.2	Obsługa autoryzacji i błędów . . . . .	5
<b>6</b>	<b>Zarządzanie stanem i sesją</b>	<b>5</b>
<b>7</b>	<b>Logika planszy i ruchów</b>	<b>6</b>
<b>8</b>	<b>Struktura katalogów</b>	<b>7</b>
<b>9</b>	<b>Aktualne ograniczenia</b>	<b>7</b>
<b>10</b>	<b>Rekomendowane następne kroki</b>	<b>7</b>
<b>11</b>	<b>Wniosek końcowy</b>	<b>8</b>

# 1 Podsumowanie wykonanych prac

W ramach aplikacji mobilnej wykonano działający klient Android MVP, który komunikuje się z backendem Literaki przez REST API. Aplikacja obsługuje sesję gościa, tworzenie i dołączanie do gier, przeglądanie listy gier, widok szczegółów rozgrywki, planszę 15x15, rack gracza oraz podstawowe akcje gry: start, zagranie liter, pominięcie tury i poddanie gry.

## 1.1 Najważniejsze rezultaty

- Przygotowano projekt Android oparty o Kotlin, Gradle Kotlin DSL i Jetpack Compose.
- Zaimplementowano nawigację ekranów z wykorzystaniem Navigation Compose.
- Utworzono warstwę dostępu do API z Retrofit, Moshi i OkHttp.
- Dodano automatyczne przekazywanie tokenu Bearer do żądań autoryzowanych.
- Zaimplementowano trwałą zapis sesji użytkownika w SharedPreferences.
- Rozdzielono logikę UI do ViewModeli opartych o StateFlow i korutyny.
- Przygotowano ekran planszy z możliwością wpisywania szkicu liter bezpośrednio w komórkach.
- Dodano lokalną walidację liter przed wysłaniem ruchu do backendu.
- Utworzono testy jednostkowe dla pomocniczej logiki planszy i racka.

## 2 Zakres funkcjonalny MVP

Obszar	Opis wykonania	Status
Sesja użytkownika	Logowanie jako gość przez endpoint <code>/api/v1/auth</code> , zapis identyfikatora, nicku i tokenu.	Wykonane
Lista gier	Pobieranie gier przypisanych do aktualnego użytkownika, sortowanie malejąco po ID.	Wykonane
Tworzenie gry	Utworzenie lobby i prezentacja ID gry jako kodu dołączenia.	Wykonane
Dołączanie do gry	Wpisanie kodu/ID gry i dołączenie przez endpoint backendu.	Wykonane
Szczegóły gry	Wyświetlenie statusu, graczy, wyników, planszy, racka i historii ruchów.	Wykonane
Start gry	Obsługa rozpoczęcia gry z poziomu widoku szczegółów.	Wykonane
Ruchy gracza	Zagranie liter, pominięcie tury i poddanie gry.	Wykonane
Walidacja lokalna	Normalizacja liter, odrzucanie pól zajętych/poza planszą, kontrola racka.	Wykonane
Testy	Testy jednostkowe dla logiki szkicu planszy.	Częściowe
Synchronizacja live	Ręczne odświeżanie stanu z backendu. Brak WebSocket/pollingu automatycznego.	Do rozwoju

### 3 Stos technologiczny

Technologia	Zastosowanie
Kotlin 2.0.21	Główny język aplikacji Android.
Android Gradle Plugin 8.9.1	Konfiguracja budowania aplikacji.
Jetpack Compose + Material 3	Deklaratywny interfejs użytkownika.
Navigation Compose 2.8.4	Nawigacja między ekranami.
Lifecycle ViewModel + State-Flow	Zarządzanie stanem ekranów i przepływem danych.
Retrofit 2.11.0	Klient HTTP dla REST API.
Moshi 1.15.1	Serializacja i deserializacja JSON.
OkHttp Logging Interceptor 4.12.0	Logowanie zapytań w buildzie debug.
JUnit 4.13.2	Testy jednostkowe logiki domenowej/pomocniczej.

Tabela 2: Najważniejsze technologie i biblioteki wykorzystane w aplikacji.

Konfiguracja projektu zakłada `compileSdk = 35`, `minSdk = 24`, `targetSdk = 35` oraz bazowy adres API ustawiony w `BuildConfig.API_BASE_URL` na `https://literaki.skiq.pl/` dla buildów debug i release.

## 4 Architektura aplikacji

Aplikacja została podzielona na czytelne warstwy: UI, ViewModel, Repository, API/model danych oraz warstwę sesji. Taki podział ogranicza zależności ekranów od szczegółów komunikacji sieciowej i ułatwia testowanie logiki niezależnej od interfejsu.

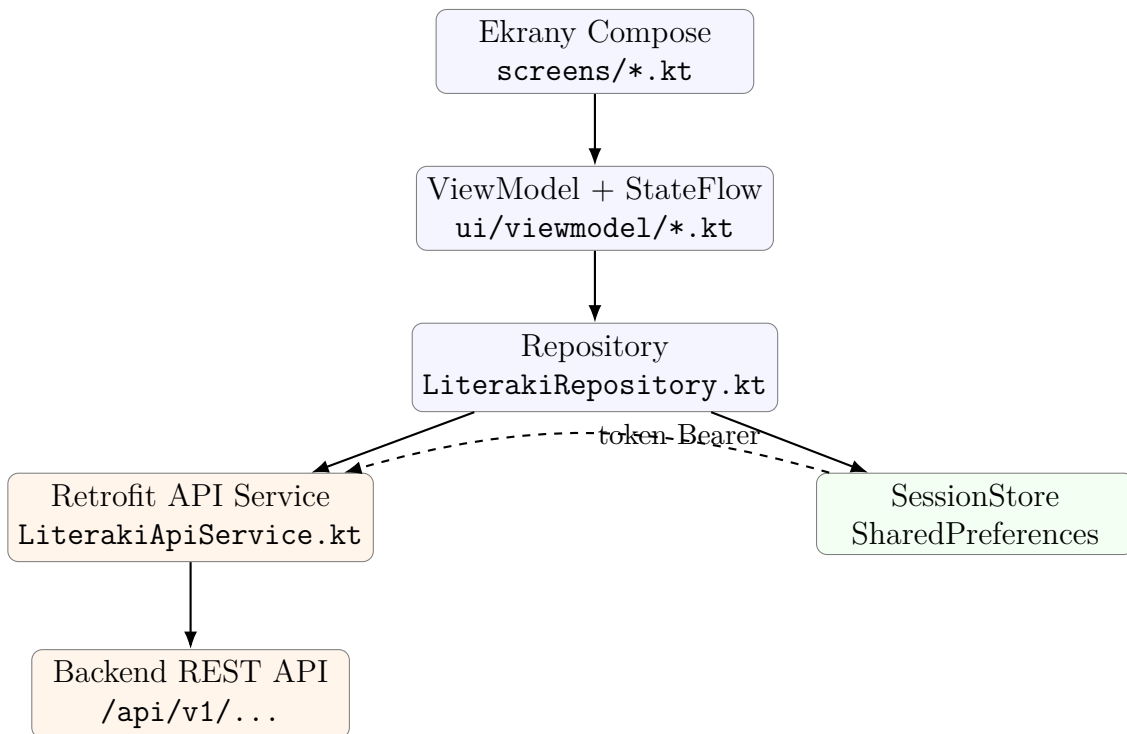
### 4.1 Warstwa UI

Interfejs użytkownika zbudowano w Jetpack Compose. W katalogu `screens` znajdują się osobne ekrany dla głównych scenariuszy:

- `SplashScreen` – decyzja, czy użytkownik ma aktywną sesję,
- `NickScreen` – podanie nicku i logowanie gościa,
- `MainMenuScreen` – ekran startowy użytkownika,
- `CreateGameScreen` – tworzenie lobby,
- `JoinGameScreen` – dołączanie po kodzie/ID,
- `OpenGamesScreen` oraz `MyGamesScreen` – listy gier,
- `GameDetailsScreen` – pełny widok rozgrywki.

### 4.2 Nawigacja

Nawigacja jest scentralizowana w `MainActivity.kt`. Definicje tras umieszczono w klasie `Screen`. Szczegóły gry przyjmują argument `gameId`, dzięki czemu aplikacja może przejść z listy lub po utworzeniu/dołączeniu bez przechowywania globalnego stanu wybranej gry.



Rysunek 1: Uproszczona architektura klienta Android.

Listing 1: Model tras nawigacji.

```

sealed class Screen(val route: String) {
    object Splash : Screen("splash")
    object Nick : Screen("nick")
    object MainMenu : Screen("main_menu")
    object CreateGame : Screen("create_game")
    object JoinGame : Screen("join_game")
    object OpenGames : Screen("open_games")
    object MyGames : Screen("my_games")
    object GameDetails : Screen("game_details/{gameId}") {
        const val ARG_GAME_ID = "gameId"
        fun createRoute(gameId: Int): String = "game_details/$gameId"
    }
}

```

## 5 Integracja z backendem

Komunikacja z backendem odbywa się przez Retrofit. Interfejs `LiterakiApiService` mapuje endpointy REST na funkcje `suspend`. Modele odpowiedzi opisano jako klasy danych z adnotacjami Moshi, m.in. `ApiGame`, `ApiPlayer`, `ApiMove` i `ApiTilePlacement`.

Listing 2: Wybrane endpointy klienta API.

```

interface LiterakiApiService {
    @POST("api/v1/auth")
    suspend fun authenticateGuest(@Body request: AuthRequest): Response<AuthResponse>
}

```

```

@GET("api/v1/games")
suspend fun getGames(): Response<List<ApiGame>>

@POST("api/v1/games/{id}/join")
suspend fun joinGame(@Path("id") id: Int): Response<ApiGame>

@POST("api/v1/games/{id}/moves")
suspend fun placeTiles(
    @Path("id") id: Int,
    @Body request: PlaceTilesMoveRequest
): Response<ApiMove>
}

```

## 5.1 Obsłużone endpointy

Endpoint	Rola w aplikacji
POST /api/v1/auth	Utworzenie sesji gościa i pobranie tokenu.
GET /api/v1/me	Pobranie danych aktualnego użytkownika.
GET /api/v1/games	Pobranie listy gier użytkownika.
POST /api/v1/games	Utworzenie nowej gry/lobby.
GET /api/v1/games/{id}	Pobranie szczegółów gry.
POST /api/v1/games/{id}/join	Dołączenie do gry.
POST /api/v1/games/{id}/start	Rozpoczęcie gry.
POST /api/v1/games/{id}/moves	Zagranie liter, pass lub resign.

Tabela 3: Zakres integracji REST API.

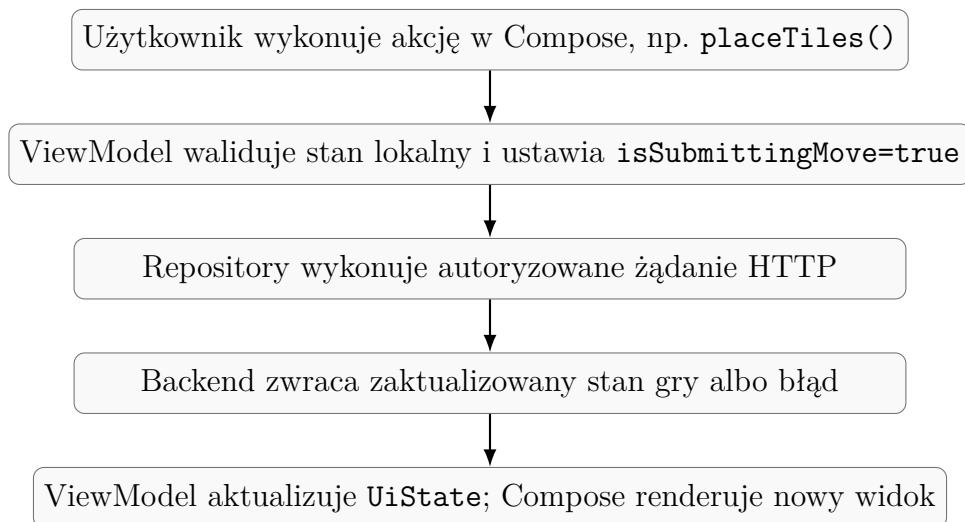
## 5.2 Obsługa autoryzacji i błędów

Warstwa `AppContainer` tworzy `OkHttpClient` z interceptorem, który dodaje nagłówek `Authorization: Bearer <token>` dla aktywnej sesji. `Repository` centralizuje obsługę odpowiedzi HTTP, parsuje komunikaty błędów z pól `error` i `errors`, a przy kodzie 401 czyści sesję użytkownika i sygnalizuje konieczność ponownego logowania.

## 6 Zarządzanie stanem i sesją

Stan każdego ekranu reprezentują niemutowalne klasy `UiState`. `ViewModele` wystawiają je jako `StateFlow`, a ekrany `Compose` subskrybują zmiany przez `collectAsState`. Operacje sieciowe wykonywane są w `viewModelScope.launch`, co separuje asynchroniczną logikę od komponentów UI.

Sesja przechowuje trzy wartości: `userId`, `username` i `token`. Dane są odczytywane przy starcie aplikacji przez `SplashViewModel`, który decyduje o przekierowaniu na ekran nicku albo do menu głównego.



Rysunek 2: Przepływ stanu dla akcji gracza.

## 7 Logika planszy i ruchów

Wydzielono pomocniczy moduł `BoardDraft.kt`, który odpowiada za spójność lokalnego szkicu liter. Plansza ma rozmiar 15x15, a komórki identyfikowane są kluczem tekstowym w formacie `x,y`. Przed wysłaniem ruchu aplikacja:

1. normalizuje wpisaną wartość do jednej wielkiej litery,
2. usuwa puste wpisy i pola poza zakresem planszy,
3. ignoruje próby nadpisania komórek już zajętych na planszy,
4. buduje listę `ApiTilePlacement` posortowaną po współrzędnych,
5. sprawdza, czy rack gracza zawiera wszystkie użyte litery w odpowiednich licznosciach.

Listing 3: Lokalna walidacja liter względem racka.

```
fun rackContainsAllLetters(rack: List<String>, letters: Collection<String>):  
    Boolean {  
        val rackCounts = rack  
            .map(::normalizeDraftLetter)  
            .filter(String::isNotBlank)  
            .groupBy { it }  
            .eachCount()  
            .toMutableMap()  
  
        letters  
            .map(::normalizeDraftLetter)  
            .filter(String::isNotBlank)  
            .forEach { letter ->  
                val remaining = rackCounts[letter] ?: return false  
                if (remaining <= 0) return false  
                rackCounts[letter] = remaining - 1  
            }  
  
        return true  
    }
```

Właściwe reguły gry pozostają po stronie backendu. Klient wykonuje walidację UX, aby szybciej informować użytkownika o oczywistych błędach, ale ostateczne zatwierdzenie ruchu nadal zależy od odpowiedzi API.

## 8 Struktura katalogów

Listing 4: Najważniejsza struktura projektu.

```
LiterakiApp/  
  app/  
    API.md  
    build.gradle.kts  
    src/main/java/com/example/literakiapp/  
      MainActivity.kt  
      LiterakiApplication.kt  
    data/  
      AppContainer.kt  
      SessionStore.kt  
      api/LiterakiApiService.kt  
      model/ApiModels.kt  
      repository/LiterakiRepository.kt  
    navigation/Screen.kt  
    screens/*.kt  
    ui/viewmodel/*.kt  
    ui/theme/*.kt  
    src/test/java/com/example/literakiapp/  
      BoardDraftTest.kt  
  gradle/libs.versions.toml  
  scripts/Test-LiterakiApiFlow.ps1
```

## 9 Aktualne ograniczenia

- Odświeżanie rozgrywki jest ręczne; brak automatycznego pollingu lub WebSocketów.
- Aplikacja bazuje na prostym modelu sesji gościa; brak pełnego logowania kontem, rejestracji i odzyskiwania dostępu.
- Walidacja reguł gry jest głównie backendowa; klient waliduje jedynie lokalny szkic i rack.
- Brak rozbudowanych testów UI Compose i testów integracyjnych z mockowanym API.
- Brak konfiguracji wariantów środowisk, np. osobnego adresu API dla development/staging/production poza aktualnym BuildConfig.
- Brak mechanizmu obsługi konfliktów stanu, gdy drugi gracz wykona ruch między ręcznymi odświeżeniami.

## 10 Rekomendowane następne kroki

1. Dodać automatyczną synchronizację stanu gry: polling z limitem częstotliwości albo kanał WebSocket/SSE.

2. Rozszerzyć testy o ViewModele z fake repository oraz testy błędów API.
3. Dodać testy UI dla głównych ścieżek: logowanie, tworzenie lobby, dołączenie i wykonanie ruchu.
4. Uporządkować warianty środowisk przez product flavors lub osobne konfiguracje Gradle.
5. Rozważyć Dependency Injection, np. Hilt/Koin, jeśli projekt będzie dalej rósł.
6. Poprawić UX planszy: podświetlenie pól, premie, podpowiedzi legalności ruchu oraz lepsza historia punktacji.
7. Dodać obsługę trybu offline/ponawiania zapytań dla typowych błędów sieciowych.

## 11 Wniosek końcowy

Udało się wykonać spójny technicznie klient Android MVP dla Literaków. Aplikacja ma kompletny przepływ użytkownika od sesji gościa po podstawową rozgrywkę, korzysta z nowoczesnego stosu Androidowego i ma rozdzielone odpowiedzialności między UI, ViewModel, Repository oraz API. Największą wartością obecnej wersji jest działająca integracja z backendem oraz przygotowana baza architektoniczna do dalszego rozwoju funkcji czasu rzeczywistego, testów i bogatszego UX rozgrywki.