Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Kierunek: **INF-PPT**
Specjalność: **-**

Praca Dyplomowa Inżynierska

Serwis do przeprowadzania rozgrywek Rummikub

Adam Bednarz

Opiekun pracy
dr inż. Jakub Lemiesz

Słowa kluczowe: rummikub, bot, flutter, firebase

Wrocław 2022

Streszczenie

Celem pracy dyplomowej było stworzenie serwisu umożliwiającego przeprowadzanie rozgrywek Rummikub pomiędzy graczami lub między graczem a botem. Została stworzona aplikacja na telefon z systemem Android oraz strona internetowa dostępna w przeglądarce. Ponadto opisano algorytm dla bota wraz z analizą złożoności. Podczas implementacji serwisu wykorzystano technologie Flutter oraz Firebase. Flutter jest technologią umożliwiająca tworzenie projektów jednocześnie na różne platformy. Firebase oferuje usługi serwerowe dla projektu.

Abstract

The aim of the thesis was to create a service that allows playing Rummikub between players or between a player and a bot. An Android phone app has been created, as well as a browser-based website. Moreover, an algorithm for the bot was described along with complexity analysis. Flutter and Firebase technologies were used during the implementation of the service. Flutter is a technology that allows projects to be created simultaneously for different platforms. Firebase offers server services for the project.

Spis treści

Wstęp

Celem pracy jest zaprojektowanie i implementacja serwisu informatycznego, który umożliwia przeprowadzanie gier Rummikub.

Główne założenia funkcjonalne serwisu:

- autoryzacja użytkowników,
- zarządzanie instancjami gier,
- korzystanie z serwisu za pomocą strony internetowej lub aplikacji na telefon,
- rozgrywka z botem.

Obecnie istnieją już takie rozwiązania, które dostarczają nam podobne funkcjonalności. Jednakże motywacją tej pracy jest bliższe zapoznanie się z procesem tworzenia takiego serwisu, jak również zaznajomienie się ze złożonością gry Rummikub. Wybrano nowe technologie do implementacji serwisu, dzięki czemu praca przyczyniła się również do poznania nieznanych dotąd technologii. Czynnikiem wyróżniającym się stworzonego serwisu jest umożliwienie graczom uczestniczenie w danej rozgrywce niezależnie od tego czy korzystają z aplikacji na telefon (iOS, Android), czy też ze strony internetowej w przeglądarce.

Praca składa się z czterech rozdziałów. W rozdziale pierwszym omówiono grę Rummikub, jej zasady oraz przebieg rozgrywki. W rozdziale drugim zobrazowano projekt serwisu. Opisano jego przypadki użycia oraz proces przebiegu aktywności. W rozdziale trzecim przedstawiono technologie, które zostały użyte w projekcie informatycznym. Opisano narzędzia, z których korzystano podczas tworzenia serwisu. W rozdziale czwartym przedstawiono dokumentację techniczną systemu. Ukazano sposób w jaki zostały zaimplementowane serwer i aplikacja klienta. Do przedstawienia graficznego wykorzystano diagramy klas oraz sekwencji. W rozdziale piątym przedstawiono kwestie implementacji bota w grze Rummikub i umieszczono pseudokod algorytmu.

Rozdział 1

Opis gry Rummikub

1.1 Gra

Została stworzona przez Ephraima Hertzano i po raz pierwszy wydano ją w 1950 roku. W rozgrywce może uczestniczyć od dwóch do czterech graczy. Gra składa się z kości, które są numerowane od 1 do 13. Występują one w czterech kolorach (pomarańczowy, czerwony, niebieski, czarny). Każda kość o danym kolorze i liczbie występuje dwa razy. Ponadto występują dwie specjalne kości jako jokery. Łącznie gra zawiera 106 kości.

Gra polega na wykładaniu grup, bądź serii. Grupa to trzy lub cztery kości o różnych barwach, ale z tą samą liczbą, natomiast seria to co najmniej trzy kolejne kości o tym samym kolorze.

W przypadku braku odpowiednich kości do gry można posłużyć się dwoma zestawami kart po 52 karty i 2 jokery. Gdzie 1, 11, 12, 13 można zastąpić odpowiednio asem, waletem, damą i królem.

1.2 Rozgrywka

Zbiór wszystkich wymieszanych kości nazywany jest bankiem. Z niego na początku gry każdy gracz otrzymuje 14 kości. W przypadku pierwszego ruchu należy wyłożyć własne kości o sumie numerów tych kości co najmniej 30, bez możliwości modyfikowania kości leżących na planszy.

Po wyłożeniu pierwszego ruchu gracz może modyfikować inne wyłożone wcześniej układy. Dozwolone jest przebudowywanie układów kości (rozbijanie lub rozbudowywanie). Jednak w każdym ruchu należy wyłożyć przynajmniej jedną własną kość.

Na każdy ruch przypada ustalony wcześniej limit czasowy. Po jego upływie w przypadku braku wyłożenia kości przez gracza, lub gdy stan planszy nie spełnia zasad gry, gracz pobiera z banku jedną kość i cofa wprowadzone zmiany układów kości.

Joker w grze Rummikub symbolizuje dowolną kość. Można nim zastąpić brakujący element do utworzenia układu kości. Joker ma taką samą wartość jak kość, którą zastępuje. Można zabrać wyłożonego jokera zastępując go odpowiednią kością i wykorzystać go w innym miejscu na planszy.

Gra kończy się w momencie, gdy któryś z graczy wyłoży wszystkie swoje kości lub gdy zabraknie kości w banku. W przypadku drugim wygrywa ten gracz, który ma najmniejszą sumę liczb znajdujących się na kościach.

Rozdział 2

Projekt serwisu

W tym rozdziale przedstawiono opis projektu systemu w notacji UML uwzględniający wymagania funkcjonalne opisane we wstępie. Do opisu relacji pomiędzy składowymi systemu wykorzystano diagramy UML.

2.1 Struktura

W serwisie wykorzystano architekturę klient-serwer. Istnieje jeden serwer, do którego może podłączać się wiele klientów i odpowiada on za komunikację i zarządzanie danymi. W komponencie klienta wykorzystano architekturę trójwarstwową. Architektura ta dzieli komponent na trzy osobne części:

- warstwa prezentacji,
- warstwa biznesowa,
- warstwa danych.

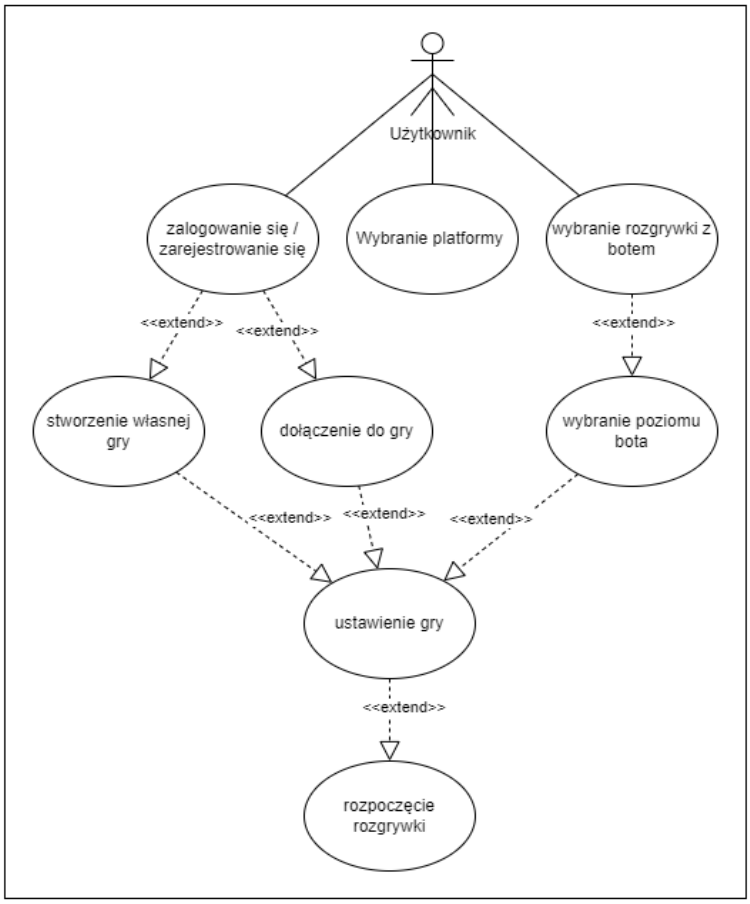
Warstwa prezentacji jest to interfejs graficzny użytkownika. Jest odpowiedzialna za interakcję z użytkownikiem (wyświetlanie i wprowadzanie danych).

Warstwa biznesowa odpowiada za przetwarzanie komunikatów od użytkownika lub ze strony serwera. Tutaj zawarta jest wszelka logika aplikacji. Przetworzone dane są przekazywane do warstwy prezentacji i/lub warstwy danych. Warstwa ta jest łącznikiem pomiędzy warstwą prezentacji, a warstwą danych.

Warstwa danych jest dostępem do danych. Obsługuje połączenie aplikacji z zewnętrznym obiektem dostarczającym dane (baza danych, serwer).

2.2 Przypadki użycia

Poniżej został przedstawiony diagram ukazujący możliwe przypadki użycia występujące w serwisie.

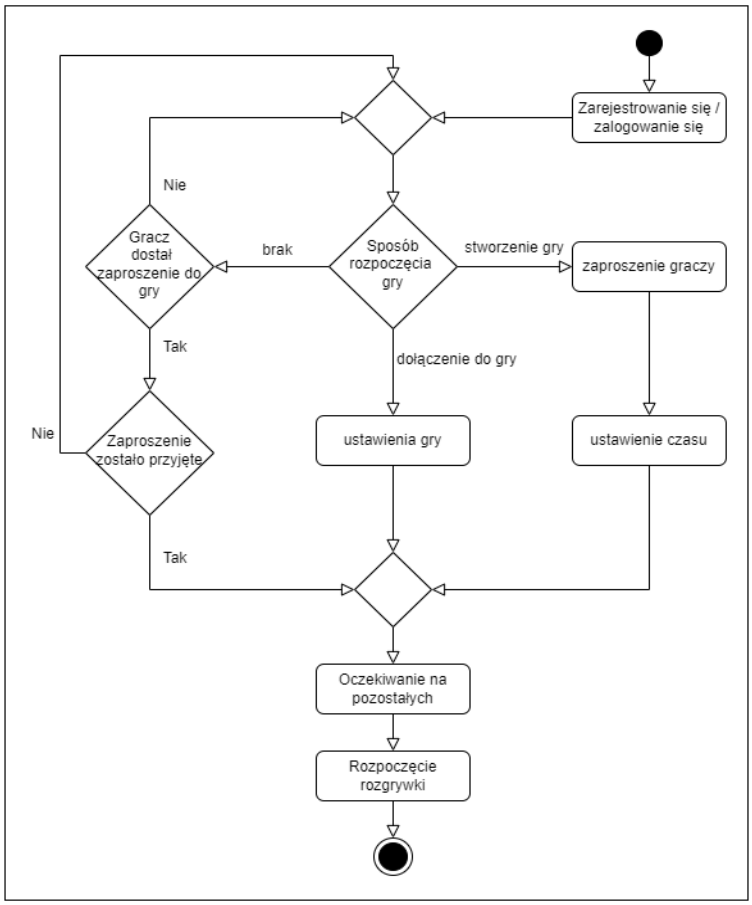


Rysunek 2.1: Diagram przypadków użycia.

Użytkownik może wybrać na jakiej platformie będzie chciał korzystać z serwisu. Ma do wyboru stronę internetową albo aplikacje na telefon. Aby móc dołączyć do gry z innymi graczami należy zalogować się lub zarejestrować w aplikacji. Następnie gracz ma możliwość stworzenia własnej gry lub dołączenia do innej. Jeśli zdecyduje się stworzyć własną grę, musi wybrać od 2 do 4 osób spośród wszystkich aktywnych graczy. Panel konfiguracji gry umożliwia zadeklarowanie dozwolonego czasu na wykonanie ruchu oraz liczbę graczy. Dołączenie do innej gry działa na zasadzie znalezienia oczekującej gry o takich samych ustawieniach, jakie wprowadził gracz, albo poprzez otrzymanie zaproszenia od innego użytkownika. Jeśli wszystkie miejsca w grze zostały wypełnione rozgrywka się rozpoczyna. W przypadku wybrania gry z botem, odbywa się ona lokalnie bez łączenia z serwerem. Użytkownik ma do dyspozycji dwa poziomy bota. Następnie również konfiguruje grę i rozpoczyna rozgrywkę.

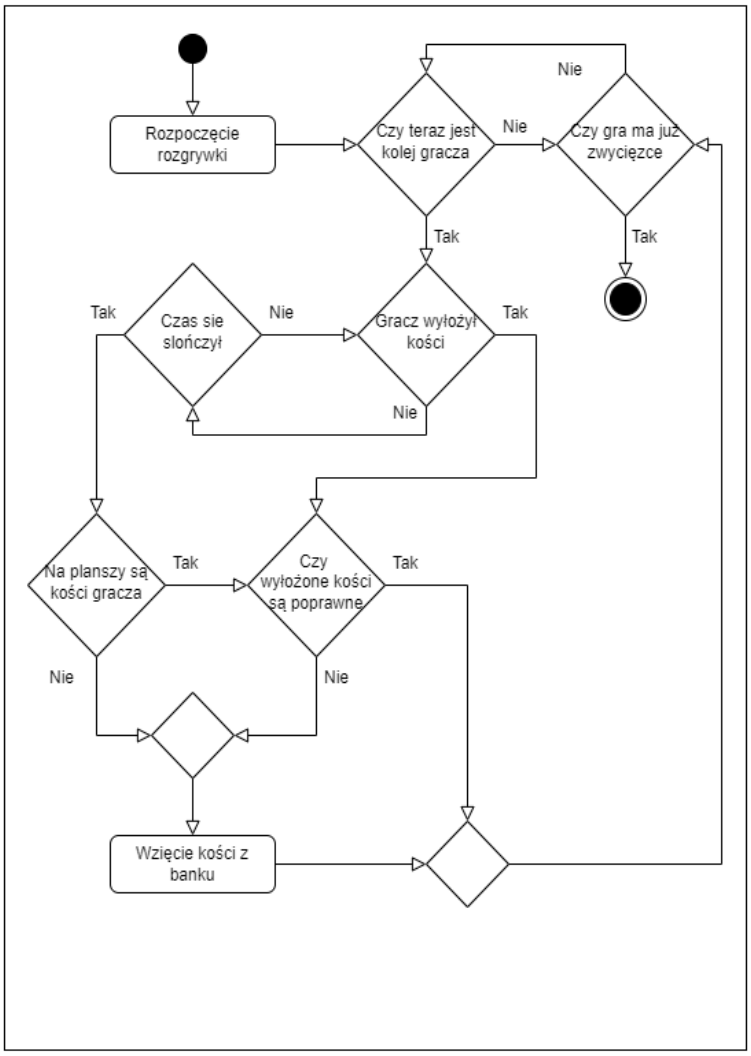
2.3 Diagramy aktywności

W tym podrozdziale zostały przedstawione dwa diagramy aktywności skupiające się na przebiegu głównych funkcjonalności aplikacji.



Rysunek 2.2: Diagram aktywności rozpoczęcie gry.

Powyższy diagram ukazuje proces przebiegu rozpoczęcia gry z innymi użytkownikami. Aby móc rozpocząć rozgrywkę z graczami trzeba posiadać konto, a więc proces rozpoczyna się od zarejestrowania się bądź zalogowania się. Następnie gracz może wybrać sposób w jaki przystąpi do rozgrywki. Ma możliwość stworzenia własnej gry i zaproszenia poszczególnych dostępnych użytkowników. Po wybraniu graczy deklaruje czas przeznaczony na ruch w grze. Drugą opcją jest dołączenie do pierwszej znalezionej gry o takiej samej konfiguracji jaką zadeklarował użytkownik i która nadal oczekuje na dołączenie innych graczy. W przypadku nieznalezienia takiej rozgrywki gracz jest dołączany do nowo utworzonej gry o podanych parametrach. Jeśli jednak tego nie zrobi, może oczekiwać na zaproszenie od innego użytkownika. Po pojawieniu się takiego zaproszenia ma możliwość przyjęcia go lub odrzucenia. Jeśli zaakceptuje to jest dołączany do gry z ustawieniami narzuconymi przez zapraszającego. Bez względu na to jaką opcje rozpoczęcia gry gracz wybierze, to w następnym kroku oczekuje, aż do gry dołączą wszyscy wymagani gracze. Proces kończy się rozpoczęciem rozgrywki.



Rysunek 2.3: Diagram aktywności rozgrywki.

Diagram ukazany na rysunku powyżej ukazuje proces przebiegu rozgrywki. Użytkownik oczekuje na własną kolej ruchu. W sytuacji, gdy po wykonaniu ruchu przez innego gracza, jest wyłaniany zwycięzca, gra dobiega końca. Natomiast jeśli następuje kolej gracza, ma on możliwość modyfikowania planszy i wykładaniu własnych kości. Na wykonanie ruchu każdy gracz ma określoną ilość czasu zadeklarowaną w trakcie tworzenia gry. Wyłożenie kości oznacza, że gracz zatwierdza modyfikacje przeprowadzone na planszy. Następnie zmiany dokonane na planszy są weryfikowane pod względem poprawności z zasadami gry. W przypadku, gdy czas upłynął zanim gracz potwierdził ruch, sprawdza się, czy na planszy znajdują się kości wyłożone przez gracza. Jeśli ich nie ma, gracz otrzymuje jedną kość z puli wolnych kości. Jeśli jednak na planszy znajdują się jakieś kości gracza, to zmodyfikowane zbiory kości są również weryfikowane pod względem poprawności. W przypadku niespełnienia zasad gry gracz otrzymuje nową kość. Jeśli gracz wyłożył wszystkie swoje kości, bądź wziął z banku ostatnią kość, gra dobiega końca.

Rozdział 3

Opis technologii

W rozdziale tym wskazano jakie podejście technologiczne zostało użyte wraz z omówieniem i zaargumentowaniem powodu takiego wyboru. Przedstawiono również jakie konkretne aspekty tych technologii zostały wykorzystane w projekcie.

3.1 Flutter

Do implementacji serwisu po stronie klienta użyto technologii Flutter. Jest to zestaw narzędzi pozwalający tworzyć natywne, wieloplatformowe aplikacje mobilne, komputerowe oraz internetowe. Flutter stworzony jest przez firmę Google, a jego pierwsza stabilna wersja ukazała się pod koniec 2019 roku. Mimo stosunkowo młodej technologii Google przyczynia się do jej dynamicznego rozwoju i zdobywania popularności wśród programistów.

3.1.1 Dart

Aplikacje we Flutterze pisze się w języku Dart. Jest to zorientowany obiektowo, statycznie typowany, wysokopoziomowy język programowania. Składnia języka Dart wzorowana była na takich językach programowania jak Java czy C#, by ułatwić programistom naukę języka. Ze względu na to, że jest to dosyć nowy język, warto uwzględnić tutaj parę charakterystycznych cech, co może być przydatne w zapoznawaniu się z kodem źródłowym tego serwisu.

Funkcja *main()* jest punktem wejścia do programu napisanego za pomocą Dart, zatem język ten nie wymaga klasy (w przeciwieństwie do Java). Wszystko, co można umieścić we zmiennej, jest obiektem. Oprócz ogólnych typów danych, istnieje typ *dynamic*. Zmienna tego typu może zawierać dowolny typ. Wszystkie typy są domyślnie *non-nullable*, więc muszą zawierać jakąś wartość. Jeśli chcemy zadeklarować zmienną z pustą wartością, po nazwie typu umieszczamy *"?"* np. *int?*. Dart używa wnioskowania o typie, więc zaleca się deklarowanie zmiennych z użyciem słowa kluczowego *var* lub *final* czy *const*. W języku tym nie występują słowa kluczowe *public*, *protected*, *private*. Domyślnie wszystkie zmienne są publiczne, jeśli jednak nazwa zmiennej będzie zaczynać się od *„_"*, to będzie to zmienna prywatna. Podczas tworzenia obiektów używanie słowa *new* jest opcjonalne. W konstruktorze można odwołać się do zmiennych instancji, aby automatycznie przypisać im wartości np. *MojaKlasa(this.zmienna1, this.zmienna2);* Dodatkowo każda klasa ma domyślne metody *set* do modyfikowania danej zmiennej oraz *get* do jej zwracania. W przypadku zmiennych prywatnych *getter* nie występuje, a w przypadku stałych *setter* nie występuje.

W implementacji serwisu często korzystano z programowania asynchronicznego. W języku Dart programowanie to charakteryzuje się klasami *Future* oraz *Stream*. Obie klasą umożliwiają wykonywanie kodu asynchronicznie. W przypadku *Future* wykonuje się jedno zapytanie i zwracana zostaje odpowiedź. Natomiast w przypadku *Stream* na jedno zapytanie może być zwracane wiele odpowiedzi.

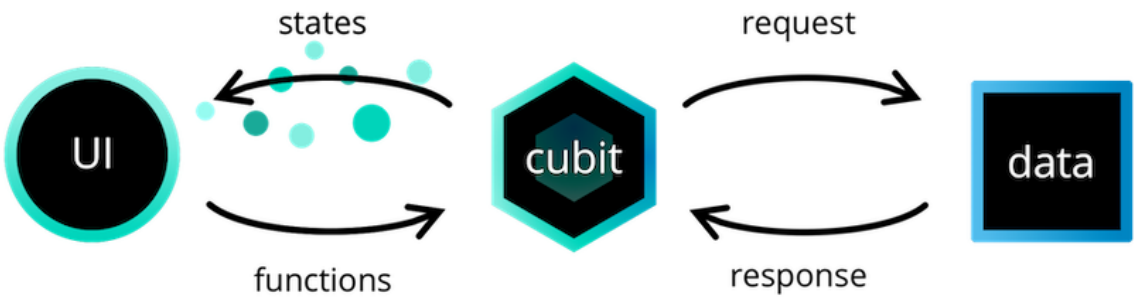
3.1.2 Widget

Każdy element interfejsu użytkownika we Flutterze jest widgetem. Widget może być elementem widocznym np. przyciskiem, ale również niewidocznym, który będzie odpowiadał za definiowanie układu elementów jak np. kolumna. Są dwa typy widgetów: statyczny oraz dynamiczny. W statycznym widgecie wszystkie pola muszą być stałe. W przypadku widgetów dynamicznych za każdym razem, gdy jego stan się zmienia jest przebudowywany. Interfejs buduje się poprzez zagnieżdżanie widgetów. Zatem cała strona bądź ekran jest widgetem jako całość, ale ta całość składa się z innych widgetów.

Ważnym aspektem w interfejsie użytkownika jest przechodzenie pomiędzy ekranami czy stronami. W Flutterze za ten mechanizm odpowiada *Navigator*, który należy do widgetów niewidocznych oraz obiekt *Route*. *Navigator* zarządza obiektami *Route* na stosie. Natomiast *Route* jest obiektem reprezentującym daną stronę czy ekran. Przesyłając do *Navigatora* obiekt *Route*, widget znajdujący się w tym obiekcie będzie umieszczany na stosie, więc tylko on będzie widoczny jako interfejs użytkownika. Zatem cofnięcie się w tył w aplikacji, będzie zrzucało dany widget ze stosu i będzie widoczny poprzedni widget.

3.1.3 BLoC

Flutter domyślnie nie oddziela kodu dotyczącego sposobu działania programu od części wizualnej. Chcąc wyodrębnić aspekty wizualne od aspektów logicznych aplikacji, należy użyć wybranego wzorca architektury. Istnieje wiele pakietów pomagających zaimplementować poszczególne wzorce. W tym projekcie zastosowano rekomendowany przez Google pakiet BLoC, który umożliwia implementacje architektury trójwarstwowej z podziałem na warstwę wizualną, logiki i danych, jak również wprowadza system zarządzania stanami w aplikacji.



Rysunek 3.1: Model struktury wzorca projektowego BLoC.

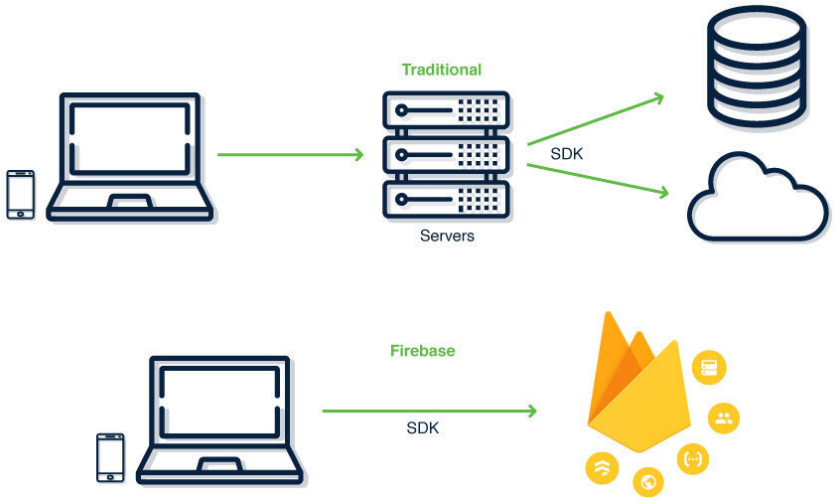
BLoC (Business Logic Component) - wzorzec projektowy pomagający oddzielić elementy wizualne projektu od części logiki działania programu. Dzieli projekt na trzy główne komponenty UI, cubit, data.

Warstwa UI jest odpowiedzialna za część wizualną aplikacji, cubit jest warstwą zawierającą mechanizmy działania aplikacji oraz pomostem pomiędzy interfejsem użytkownika, a zewnętrznymi danymi, natomiast warstwa data komunikuje się z zewnętrznymi instancjami (serwer, baza danych).

Cubit komunikuje się z warstwą danych poprzez funkcje asynchroniczne. Natomiast chcąc wpłynąć na zmianę wyglądu aplikacji korzysta z programowania reaktywnego. Mianowicie wysyła poszczególne stany strumieniem, którego dany element interfejsu aplikacji nasłuchuje. Stan jest to klasa reprezentująca dany stan poszczególnego elementu aplikacji. Relacja w drugą stronę polega na wywoływaniu bezpośrednio funkcji (synchronicznej lub asynchronicznej) na instancji cubita.

3.2 Firebase

Rolę serwera i bazy danych pełni platforma Firebase. Technologia ta, zarządzana również przez Google, określana jest jako Backend-as-a-Service. Firebase dostarcza wiele funkcjonalności wspierające tworzenie projektów w tym między innymi uwierzytelnianie użytkowników, bazę danych NoSQL, hosting oraz API odpowiedzialne za komunikację. Usługi, które świadczy Firebase są bezpłatne, jeśli nie przekroczą pewnego z góry ustalonego limitu zdefiniowanego dla poszczególnej funkcjonalności.



Rysunek 3.2: Porównanie struktury tradycyjnego serwera z platformą Firebase.

3.2.1 Firestore

Firestore jest to baza danych NoSQL w chmurze. Jej strukturą są kolekcje, które zawierają dokumenty, które te z kolei zawierają dane, ale także mogą zawierać kolejne kolekcje. Definiując zapytanie do bazy danych zwracane są jedynie dokumenty z poszczególnej kolekcji lub grup kolekcji. Firestore umożliwia tworzenie prostych zapytań SQL, które filtruje wyniki na podstawie wartości danych znajdujących się w dokumencie.

Firebase umożliwia po stronie klienta implementacje słuchaczy, którzy będą niezwłocznie informowani, gdy w obserwowanym dokumencie zaszła zmiana.

Firestore umożliwia konfigurację dostępu do bazy danych za pomocą definiowania reguł bezpieczeństwa. Dla każdego dokumentu oraz pola można zdefiniować zakres dostępu (czytanie, modyfikacja, usuwanie) dla poszczególnego użytkownika.

3.2.2 Funkcje Firebase

Umożliwiają zaimplementowanie własnego kodu po stronie serwera. Są wywoływane poprzez zapytania HTTPS lub w odpowiedzi na zdarzenia powstałe w bazie danych Firestore. Możliwe jest wywoływanie bezpośrednio z poziomu aplikacji. Do zapytania dołączane są tokeny uwierzytelniania Firebase i tokeny sprawdzania aplikacji. Dostępne są dwa języki programowania, w których można zaimplementować funkcje Firebase. Można wybrać język Javascript lub jego odpowiednik ze statycznym typowaniem Typescript.

3.2.3 Konsola Firebase

Jest to strona internetowa, która oferuje dostęp do wszystkich funkcjonalności Firebase. Aby korzystać z platformy Firebase należy założyć tam własny projekt i skonfigurować go z docelową aplikacją. Po wstępnej konfiguracji otrzymuje się panel kontroli nad platformą Firebase dla projektu. Widzi się cały ruch generowany przez klientów w poszczególnych funkcjonalnościach np. liczba zapisów do bazy danych. Za pomocą konsoli można konfigurować i modyfikować każdą usługę. Zapewnia bezpośrednio dostęp do bazy danych, zarządzanie zarejestrowanymi użytkownikami, monitorowanie procesu działania serwisu.

3.2.4 Emulator

Korzystając z funkcjonalności funkcji Firebase napotyka się na problem długiego czasu oczekiwania na zaktualizowanie kodu źródłowego programisty w chmurze Firebase (czas trwania około minuty). Oczekiwanie dłuższe niż parę sekund sprawia, że czas tworzenia kodu źródłowego po stronie serwera znacząco się wydłuża. Kolejną sprawą jest fakt, że w trakcie, gdy dany projekt jest już dostępny dla użytkowników, nie jest wskazana jego modyfikacja. Zatem trzeba tworzyć kopie projektu, które przeznaczone są do modyfikacji, co w przypadku, gdy nad projektem pracuje wielu programistów, staje się jeszcze bardziej nieefektywne. Ostatnim aspektem są koszty. W momencie programowania i testowania funkcjonalności serwera może dochodzić do generowania dużej ilości zapytań, co z kolei sumuje się do liczby zapytań i ewentualnych płatności.

Z powodu wyżej wymienionych problemów wraz z rozwojem platformy Firebase w 2019 roku oficjalnie został zaprezentowany Firebase Local Emulators. Emulator ten pozwala uruchomić wszystkie usługi Firebase lokalnie na własnym komputerze. Dzięki czemu szybko można nanieść zmiany w funkcjach Firebase, nie modyfikując właściwego projektu, a wykonywane zapytania nie są zliczane.

Każda usługa udostępniana przez platformę Firebase ma swój własny emulator, który jest uruchamiany na osobnym porcie. Zatem można uruchomić jedynie te emulatory tych funkcjonalności, które są wykorzystywane w projekcie. Ponadto fakt uruchomienia lokalnie funkcji Firebase umożliwia debugowanie kodu.

Rozdział 4

Implementacja serwisu

Rozdział ten zawiera dokumentację techniczną projektu. Zobrazowano sposób w jaki założenia projektowe, zostały zaimplementowane przy użyciu wybranych technologii. Zgodnie ze strukturą serwisu w podrozdziale pierwszym opisano sposób implementacji serwera, a podrozdziale drugim klienta. Przedstawiona ogólna dokumentacja miała na celu bliższe zapoznanie się ze sposobem działania serwisu pod względem informatycznym.

4.1 Serwer

Przy implementacji serwera wykorzystano głównie trzy moduły Firebase takie jak uwierzytelnianie, firestore (baza danych NoSQL) oraz funkcje. Komunikacja pomiędzy serwerem, a klientem opiera się na dwóch sposobach. Klient może bezpośrednio nasłuchiwać zmian zachodzących w bazie danych Firestore, albo wywoływać funkcje Firebase.

4.1.1 Baza danych

W bazie danych czasu rzeczywistego Firestore zostały umieszczone dwie kolekcje - *Users*, *Games*. W kolekcji *Users* są przechowywane dokumenty indeksowane unikatowymi kodami ID użytkownika, które są przydzielane w czasie rejestracji. Każdy dokument zawiera pola *name* i *active*. Pole *name* odnosi się do nazwy użytkownika, a pole *active* jest wartością logiczną wskazującą, czy dany użytkownik jest dostępny w grze (jest zalogowany i obecnie nie znajduje się w rozgrywce z innymi graczami). Opcjonalne pole *invitation* jest mapą zawierającą klucze *gameId* oraz *player*, których wartości wskazują kolejno na kod ID gry i nazwę gracza, który zaprosił danego gracza do swojej gry. W kolekcji *Games* są przechowywane dokumenty indeksowane automatycznie generowanymi unikatowymi kodami ID przy tworzeniu dokumentu i które są przypisane jako kody ID gier. Każdy dokument zawiera trzy pola: *available* (wskazuje ile graczy może jeszcze dołączyć do gry), *currentTurn* (wskazuje jakiego gracza jest teraz kolej), *size* (wskazuje na liczbę graczy w grze). Ponadto dokument danej gry zawiera jeszcze subkolekcje *playersQueue*, *playersRacks*, *pool*, *state*.

Subkolekcja *playersQueue* jest to zbiór dokumentów indeksowanych kodami ID graczy, którzy uczestniczą w tej grze. Każdy dokument zawiera pole *name* z nazwą gracza oraz pole *initialMeld*, które wskazuje czy dany gracz wyłożył już rozdanie początkowe.
Subkolekcja *playersRack* to zbiór dokumentów z automatycznie generowanymi kodami ID. Dokumenty te przedstawiają kości, które są przydzielone graczowi. Każdy dokument zawiera dwa pola *color* oraz *number*.

Subkolekcja *pool* to z kolei zbiór dokumentów z automatycznie generowanymi kodami ID. Zbiór tych dokumentów przedstawia bank w grze, czyli wszystkie kości, które nie znajdują się na planszy ani nie są przydzielone do gracza. Każdy dokument ma pola *color* oraz *number*.

Subkolekcja *state* zawiera dokładnie jeden dokument *sets* o najbardziej złożonej strukturze. W dokumencie *sets* znajdują się wszystkie zbiory, które są wyłożone na planszy. Struktura tego dokumentu składa się z mapy, w której klucze to pozycja pierwszej kości z danego zbioru na planszy, a więc moment, w którym rozpoczyna się dany zbiór kości. Wartości tej mapy to tablica kości, gdzie każda kość jest w postaci mapy z kluczami *color* i *number*. Taki sposób przedstawienia stanu planszy wynika głównie z optymizacji kosztów modyfikowania bazy danych. Nie rozbito każdego zbioru kości na osobne dokumenty, ponieważ zwiększa to nam liczbę zliczanych zapisów do bazy danych. Ponadto w tym przypadku nie ma potrzeby korzystania z właściwości oferowanych przez dokumenty takie jak śledzenie zmian w danym dokumencie czy możliwość tworzenia prostych zapytań SQL na zbiorze dokumentów.

4.1.2 Uwierzytelnianie

Tak jak wspomniano wyżej występują dwa sposoby komunikacji klienta z serwerem. W przypadku funkcji Firebase do zapytań HTTPS z aplikacji, automatycznie są dołączane tokeny uwierzytelniania. W przypadku nasłuchiwaniu zmian zachodzących w bazie danych lub operacjach zaczytywania i modyfikowania bazy danych Firestore za proces autoryzacji dostępu są odpowiedzialne reguły bezpieczeństwa Firestore.

W tym projekcie reguły bezpieczeństwa Firestore są zdefiniowane następująco:

- w kolekcji *users* użytkownik ma dostęp jedynie do dokumentu z własnym ID, na którym może wykonywać operacje czytania i modyfikacji,
- w kolekcji *games* użytkownik ma dostęp jedynie do dokumentu gry, w którym on sam jest jednym z graczy. W tym dokumencie będzie miał dostęp do pól zdefiniowanych w dokumencie oraz do subkolekcji *playersQueue* oraz *playersRacks*. Co więcej w *playersRacks* będzie miał dostęp jedynie do dokumentu z własnym polem ID. We wszystkich dostępnych miejscach użytkownik ma prawo jedynie wykonywać operacje czytania.

4.1.3 Funkcje

Za pomocą funkcji Firebase został zaimplementowany kod serwera serwisu. Jego głównymi zadaniami jest zarządzanie instancjami gier oraz uniemożliwienie prób oszukiwania w czasie rozgrywki przez graczy za pomocą ingerencji w kod źródłowy aplikacji po stronie klienta.

Zbiór zdefiniowanych funkcji został uporządkowany w trzy podzbiory. W pierwszym podzbiorze znajdują się wszystkie możliwe do wywołania przez użytkownika funkcje serwera. Drugim podzbiorem jest klasa *GameUtils*, zawierająca statyczne funkcje odpowiedzialne za zarządzanie instancjami gier. Trzecim podzbiorem jest klasa *GameLogic*, zawierająca statyczne funkcje, które odpowiadają za walidację ruchów użytkowników w grze i implementację logiki rozgrywki gry.

Charakterystyka funkcji serwera możliwych do wywołania przez użytkownika:

- *createGame* - jako paramtry wejściowe przyjmuje ID gracza i jego nazwę, listę nazw graczy zaproszonych do gry oraz czas przeznaczony na wykonanie ruchu w trakcie gry.
  Wywołuje ona metodę *createGame* z klasy *GameUtils*. Następnie wyszukuje poszczególnych graczy i informuje ich o zaproszeniu do gry.
  Parametrem zwracanym jest ID nowo utworzonej gry.
- *searchGame* - jako parametry wejściowe przyjmuje ustawienia gry: liczbę graczy i czas przeznaczony na ruch gracza oraz ID gracza i jego nazwę.
  Wywołuje funkcję *findGame* z *GameUtils*. Jeśli gra zostanie znaleziona wywoływana jest funkcja *addToGame*. Następnie jeśli gracz zajął ostatnie wolne miejsce w grze, wywoływana jest funkcja *startGame* z *GameUtils*. Jednak jeśli nie znaleziono gry, wywoływana jest funkcja *createGame* z klasy *GameUtils*. Aby uchronić się przed wyszukiwaniem i modyfikowaniem bazy danych jednocześnie przez kilka wywołań funkcji *searchGame* przez różnych graczy, proces wyszukania gry i zapisu do

niej odbywa się poprzez transakcje.
Parametrem wyjściowym jest ID gry.

- *addToExistingGame* - jako parametry wejściowe przyjmuje ID gry oraz ID gracza i jego nazwę. Funkcja szuka instancji gry, a następnie wywołuje *addToGame* z *GameUtils*. Operacje na bazie danych również odbywają się z pomocą transakcji.

- *putTiles* - jako parametry wejściowe przyjmuje ID gracza i gry oraz zbiór zbiorów kości znajdujących się na planszy.
Znajduję instancję gry i subkolekcje graczy uczestniczących w niej, a następnie wywołuje metodę *checkTurn* z *GameLogic*. Potem nadaje prawo ruchu kolejnemu graczowi w kolejce. Dalej wywołuje metodę *addNewTiles* z *GameLogic*. Jeśli metoda ta zwróci, że gracz jest zwycięzcą, wskazuje zwycięzcę. Jeśli zwróci, że gracz nie wykonał żadnego ruchu lub próbował oszukiwać, to doda do jego zbioru kości kolejną kość z banku. W przypadku, gdy była to ostatnia kość z banku, to wywołuje metodę *pointTheWinner* z *GameLogic*.

- *leftGame* - jako parametry wejściowe przyjmuje ID gracza i gry.
Usuwa gracza z kolejki graczy w grze. Jeśli akurat ten gracz miał prawo ruchu, daje możliwość ruchu następnemu graczowi.

Charakterystyka funkcji statycznych serwera z klasy *GameUtils*:

- *createGame* - tworzy dokument gry w kolekcji *games*. Dodaje założyciela do subkolekcji *playersQueue* oraz tworzy wszystkie możliwe kości w postaci dokumentów i dodaje je do subkolekcji *pool*.
Zwraca ID gry.

- *addToGame* - dodaje gracza do istniejącej już gry.
Zwraca ID gry oraz wartość logiczną dotyczącą występowania wolnych miejsc w grze.

- *findGame* - za pomocą parametrów ustawień gry (liczba graczy, czas na ruch) szuka wolnej instancji gry.
Zwraca instancję gry lub null.

- *startGame* - przydziela graczom po 14 kości losowo wybranych z subkolekcji *pool*. Nadaje prawo do wykonania ruchu pierwszego graczowi w subkolekcji *playersQueue*.

Charakterystyka funkcji statycznych serwera z klasy *GameLogic*:

- *checkTurn* - sprawdza, czy dany gracz ma prawo wyłożyć kości.

- *addNewTiles* - przeprowadza walidację nowego zbioru kości na planszy. Sprawdza, czy zbiory są poprawnie ułożone, czy występują wszystkie poprzednie kości na planszy, czy nowo położone kości należą do gracza. W przypadku wyłożenia początkowego sprawdza czy gracz wyłożył nowe ułożenia o wartości co najmniej 30 punktów oraz czy poprzednie ułożenie nie zostały zmodyfikowane. Po spełnieniu tych warunków usuwa nowo wyłożone kości z puli gracza i modyfikuje stan planszy. W przypadku wyłożenia początkowego odznacza, że zostało wykonane.
Zwraca informację, czy kości zostały poprawnie dodane lub informację, że gracz zwyciężył w przypadku usunięcia wszystkich kości z jego puli.

- *getTileFromPool* - wybiera jedną kość z subkolekcji *pool* i przydziela je danemu graczowi.
Zwraca informację, czy w *pool* znajdują się jeszcze dostępne kości.

- *pointTheWinner* - sumuje wszystkie wartości kości w poszczególnych pulach graczy. Wskazuje zwycięzcę lub zwycięzców w przypadku równości sum.

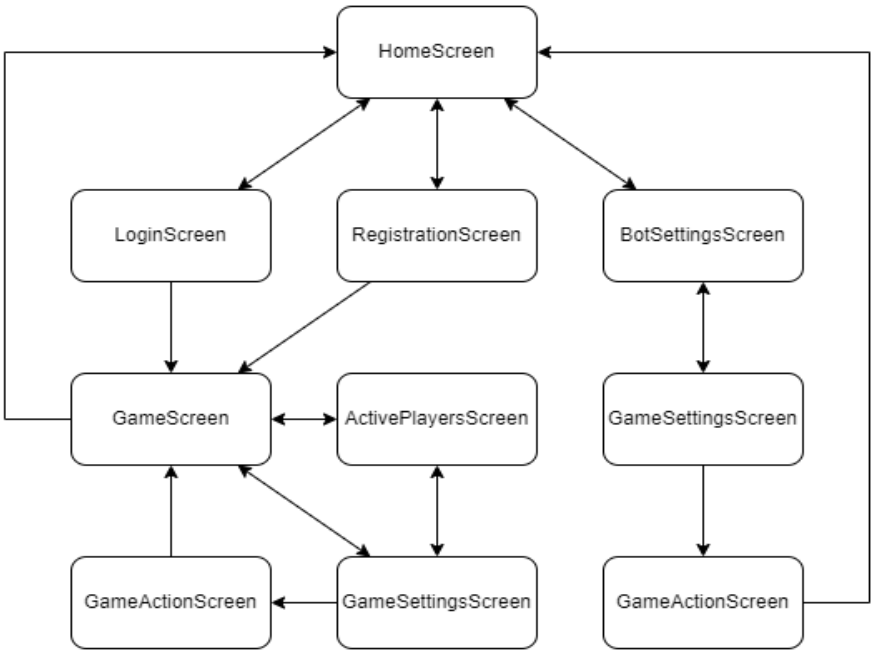
4.2 Klient

Warstwa klienta w serwisie napisana we Flutterze jest dostępna jako aplikacja na telefon albo jako strona internetowa. Z uwagi na wieloplatformowość Fluttera kod źródłowy obu wersji klienta jest wspólny z odpowiednimi konfiguracjami i modyfikacjami na poszczególne platformy.

4.2.1 Warstwa prezentacji

W warstwie prezentacji zdefiniowany jest wygląd aplikacji klienta. Warstwa ta składa się z bezstanowych widgetów, które generują interfejs graficzny. Klasy te odpowiadają za odbieranie komunikatów ze strony użytkownika jak i przekazują lub odbierają komunikaty ze strony logiki aplikacji. Zdarzenia te mogą mieć wpływ na wygląd danego widgeta. Komunikacja pomiędzy poszczególnymi stronami aplikacji jest również zdefiniowana w tej warstwie.

Nawigacja pomiędzy stronami odbywa się poprzez dynamiczną nawigacją z wygenerowanymi trasami. Jest to jeden ze sposobów komunikacji we Flutterze, dzięki któremu za cały proces nawigacji odpowiada klasa *app_router*, a w poszczególnych widgetach podaje się nazwę ścieżki i obiekty do przekazania pomiędzy stronami. Główną zaletą korzystania z tego sposobu jest eliminacja bezpośrednich zależności pomiędzy widgetami. Ponadto zmniejsza się objętość kodu, a sam kod staje się bardziej przejrzysty.



Rysunek 4.1: Diagram nawigacji.

HomeScreen jest ekranem startowym aplikacji, od którego zaczyna się każda inna ścieżka. W programie można podążyć dwoma głównymi ścieżkami. Pierwszą ścieżką jest wybranie gry z botem lokalnie na urządzeniu użytkownika bez konieczności połączenia z internetem. Następnie wybraniu ustawień bota i kolejno ustawień gry i przejściu do rozgrywki. Po wyjściu z gry użytkownik wraca do ekranu startowego. Drugą ścieżką jest z kolei zalogowanie się lub utworzenie konta gracza. Następnie użytkownik będzie miał do wyboru sposób rozpoczęcia gry. Jeśli zdecyduje się na utworzenie własnej gry z innymi aktywnymi użytkownikami, przechodzi do strony *ActivePlayersScreen*, a dalej do ustawień gry. Jeśli jednak będzie chciał dołączyć do istniejącej już gry, aplikacja przekierowuje go bezpośrednio do *GameSettingsScreen*. Po przejściu do rozgrywki gracz w każdej chili może opuścić grę. W takim przypadku znajdzie się z powro-

tem w *GameScreen*. Z którego również może nastąpić wylogowanie się użytkownika i powrót do ekranu startowego.

Charakterystyka poszczególnych klas w warstwie prezentacji:

- *AppRouter* - klasa odpowiadająca za nawigację w aplikacji. Zawiera zdefiniowane ścieżki, które wskazują na poszczególne strony. Jest odpowiedzialna za tworzenie nowego ekranu i dołączanie do niego odpowiednich zależności i przekazywanych parametrów.
- *HomeScreen* - klasa zawierająca trzy przyciski kolejno prowadzące do *RegirstrationScreen*, *LoginScreen*, *BotSettingsScreen*. Ekran startowy aplikacji.s
- *RegirstrationScreen* - klasa zawierająca formularz z danymi do rejestracji (email, nazwa gracza, hasło, potwierdzenie hasła). Przeprowadza walidację formatu email oraz długości hasła i jego identyczności z potwierdzającym hasłem.
- *LoginScreen* - klasa zawierająca formularz z potrzebnymi danymi do zalogowania użytkownika (email, hasło).
- *BotSettingsScreen* - klasa zawierająca dwa przyciski, odpowiadające za wybrany poziom bota. Przekierowują one do ekranu *GameSettings* z argumentami *playerId* oraz *serverType*, co w przypadku gry z botem będzie równoznacze z numerem ID gracza *0* oraz typ serwera *basicBot*, *advancedBot*.
- *GameScreen* - klasa zawierająca dwa przyciski, które definiują sposób dołączenia do rozgrywki. Prowadzą one do stron *ActivePlayersScreen*, *GameSettingsScreen* i jako parametr przekazują ID gracza. Ponadto, gdy użytkownik chce powrócić do wcześniejszej strony zostanie wyświetlone okno dialogowe, wymuszające potwierdzenie decyzji, gdyż wiążę się ona z wylogowaniem użytkownika i powrotem do ekranu startowego. W przypadku, gdy użytkownik zostanie zaproszony do gry przez innego gracza, również zostanie wyświetlone okno dialogowe, w którym gracz będzie mógł przyjąć lub odrzucić zaproszenie.
- *ActivePlayersScreen* - składa się z wyszukiwarki, listy aktywnych użytkowników oraz przycisku. Lista aktywnych użytkowników jest sortowana względem wyrazu wpisanego do wyszukiwarki. Użytkownik wybiera z listy graczy, których chce zaprosić do gry. Po kliknięciu w przycisk przechodzi do *GameSettings* z parametrami *playerId* (ID użytkownika), *selectedPlayers* wybrani gracze, *serverType* (nazwę serwera).
- *GameSettingsScreen* - klasa zawierająca dwa pola przeznaczone do konfiguracji liczby uczestników i czasu przeznaczonego na ruch gracza oraz przycisk. Po kliknięciu w przycisk ukazuje się obracające kółeczko i napis z liczbą graczy brakujących do rozpoczęcia rozgrywki. Po pojawieniu się wymaganej liczby graczy nastąpi przekierowanie do *GameActionScreen* z parametrami *gameId*, *playerId*, *serverType*.
- *GameActionScreen* - klasa obrazująca całą rozgrywkę w grę Rummikub. W górnej części znajduję się panel, na którym widać wszystkich uczestników i ilość pozostałego czasu dla aktualnie grającego gracza oraz przycisk zatwierdzający ruch. W środkowej części znajduje się plansza, na której można umieszczać swoje kości tworząc lub rozbudowując obecnie znajdujące się tam kości. W dolnej części umieszczony jest zbiór kości, które aktualnie posiada gracz. Komunikaty informacyjne o występujących zdarzeniach w czasie rozgrywki są pokazywane w postaci znikających dymków (toasty).

4.2.2 Warstwa logiki

W warstwie logiki zdefiniowane są zasady działania aplikacji. Z racji wykorzystanego wzorca projektowego we Flutterze klasy te są w postaci cubitów. Instancje tych klas są tworzone w klasie *AppRouter* i odpowiednio są powiązane z klasami w warstwie prezentacji. Do każdego cubita przypisany jest zbiór klas, które reprezentują poszczególne stany. Klasy stanowe są przekazywane za pomocą strumienia do odpowiednich klas w warstwie prezentacji.

Charakterystyka poszczególnych cubitów:

- *AuthCubit* - klasa ta jest powiązana z czynnościami dotyczącymi bezpośrednio konta użytkownika. Przeprowadza rejestrację, logowanie i wylogowanie gracza. Ponadto powiadamia użytkownika, gdy ten zostanie zaproszony do gry przez innego gracza i w przypadku akceptacji dołącza go do gry.
- *GameSettingsCubit* - klasa odpowiadająca za konfigurację gry. Zawiera stan zawierający wybrane ustawienia gry. Odpowiada również za proces dołączania do gry (wyszukanie gry lub jej stworzenie oraz oczekiwanie na pozostałych graczy).
- *ActivePlayersCubit* - klasa pobiera aktywnych użytkowników i odpowiednio je filtruje na podstawie podanej przez użytkownika wartości. Zawiera stan wskazujący na obecnie wybranych graczy.
- *GameActionPanelCubit* - klasa zarządzająca rozgrywką. Przechowuje stan, który zawiera listę uczestników oraz pozostały czas na ruch obecnie grającego. Zarządza kolejką graczy, jak również informuje o wyniku końcowym gry. Ponadto umożliwia użytkownikowi opuszczenie rozgrywki przed zakończeniem.
- *GameActionBoardCubit* - klasa odpowiadająca za planszę gry. Stan klasy zawiera listę obecnie ułożonych zestawów kości na planszy. Cubit ten odpowiada za aktualizowanie planszy w przypadku wyłożeń gracza lub innych graczy. W przypadku wyłożeń użytkownika przeprowadza walidację oraz zgodność z regułami gry i odpowiednio modyfikuje stan planszy.
- *GameActionRackCubit* - klasa zawierająca stan, który przechowuje kości użytkownika. Jest odpowiedzialna za modyfikację tego stanu w przypadku wyłożeń użytkownika lub gdy użytkownik dostaje kość z banku.

4.2.3 Warstwa danych

Warstwa danych składa się z dwóch interfejsów *AuthRepository* oraz *GameRepository*. Pierwszy interfejs odpowiada za zarządzanie kontem użytkownika, a drugi instancją gry. W aplikacji obecnie zaimplementowane są dwa rozwiązania będące warstwą danych: firebase, bot.

Charakterystyka interfejsu *AuthRepository*:

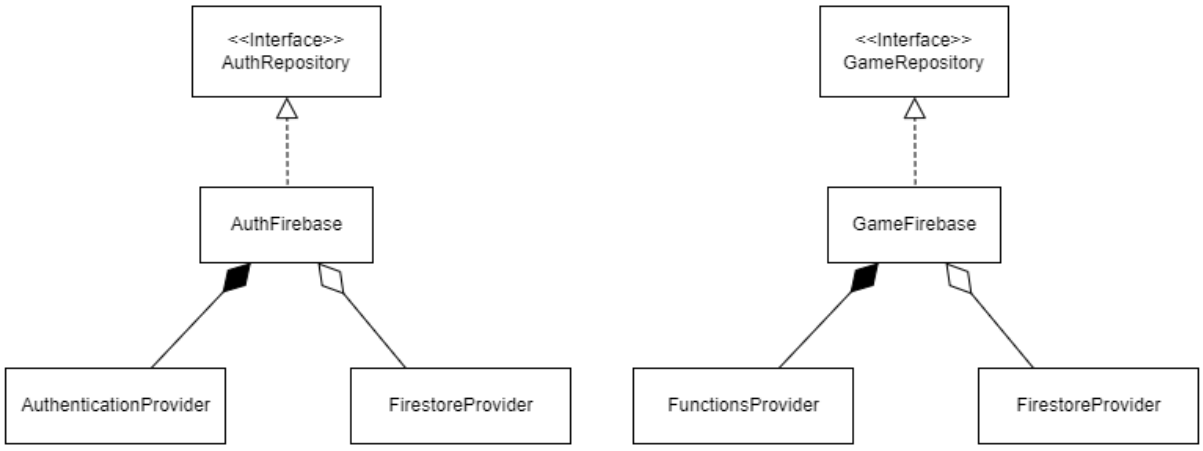
- *signUp* - metoda asynchroniczna, która jako parametry wejściowe przyjmuje email, nazwę oraz hasło użytkownika. Odpowiada za zarejestrowanie się do serwisu. Zwraca klasę *Player*, która zawiera nazwę i ID gracza.
- *logIn* - metoda asynchroniczna, która jako parametry wejściowe przyjmuje email oraz hasło użytkownika. Odpowiada za zalogowanie się do serwisu. Zwraca klasę *Player*, która zawiera nazwę i ID gracza.
- *logOut* - metoda asynchroniczna, która jako parametry wejściowe przyjmuje ID gracza. Odpowiada za wylogowanie się z serwisu.
- *invitationToGame* - metoda, która jako parametry wejściowe przyjmuje ID użytkownika. Zwraca strumień, przez którego będą wysyłane zaproszenia do gry.
- *activePlayers* - metoda, która jako parametry wejściowe przyjmuje ID użytkownika. Zwraca strumień, przez którego będą wysyłani obecnie aktywni gracze.

Charakterystyka interfejsu *GameRepository*:

- *createGame* - metoda asynchroniczna, która jako parametry wejściowe przyjmuje ID gracza, listę nazw zaproszonych graczy oraz czas przeznaczony na ruch w grze. Jest wywoływana w momencie, gry gracz tworzy własną grę i zaprasza innych graczy. Zwraca ID nowo utworzonej gry.

- *searchGame* - metoda asynchroniczna, która jako parametry wejściowe przyjmuje ID gracza, liczbę graczy w grze oraz czas przeznaczony na ruch w grze. Jest wywoływana w momencie, gdy gracz chce znaleźć grę o podanych parametrach. Zwraca ID znalezionej gry lub w przypadku jej braku zwraca ID nowo utworzonej gry.
- *joinGame* - metoda asynchroniczna, która jako parametry wejściowe przyjmuje wartość logiczną wskazującą na dołączenie do rozgrywki oraz ID gry. Jest wywoływana w momencie, gry gracz otrzymał zaproszenie do gry.
- *missingPlayers* - metoda, która jako parametry wejściowe przyjmuje ID gry. Zwraca strumień, przez którego będzie wysyłana liczba brakujących graczy do rozpoczęcia rozgrywki.
- *playerTiles* - metoda, która jako parametry wejściowe przyjmuje ID użytkownika oraz ID gry. Zwraca strumień, przez którego będą wysyłane kości należące do gracza.
- *playersQueue* - metoda, która jako parametry wejściowe przyjmuje ID gry. Zwraca strumień, przez którego będą wysyłani gracze uczestniczący w grze.
- *gameStatus* - metoda, która jako parametry wejściowe przyjmuje ID gry. Zwraca strumień, przez którego będzie wysyłany aktualny stan gry, czyli gracz mający ruch, czas przeznaczony na ruch oraz zwycięzca gry.
- *tilesSets* - metoda, która jako parametry wejściowe przyjmuje ID gry. Zwraca strumień, przez którego będą wysyłane zbiory kości znajdujące się na planszy.
- *putTiles* - metoda asynchroniczna, która jako parametry wejściowe przyjmuje ID gry oraz zbiory kości znajdujące się na planszy. Odpowiada za przekazanie wykonanego ruchu gracza.
- *leaveGame* - metoda asynchroniczna, która jako parametry wejściowe przyjmuje ID gry, ID użytkownika. Obsługuje wyjście gracza z rozgrywki.

W przypadku firebase zdefiniowane są dwie klasy *AuthFirebase* oraz *GameFirebase*, które implementują odpowiednie interfejsy. Klasy te są odpowiedzialne za komunikację z platformą Firebase.



Rysunek 4.2: Diagram klas warstwy danych firebase.

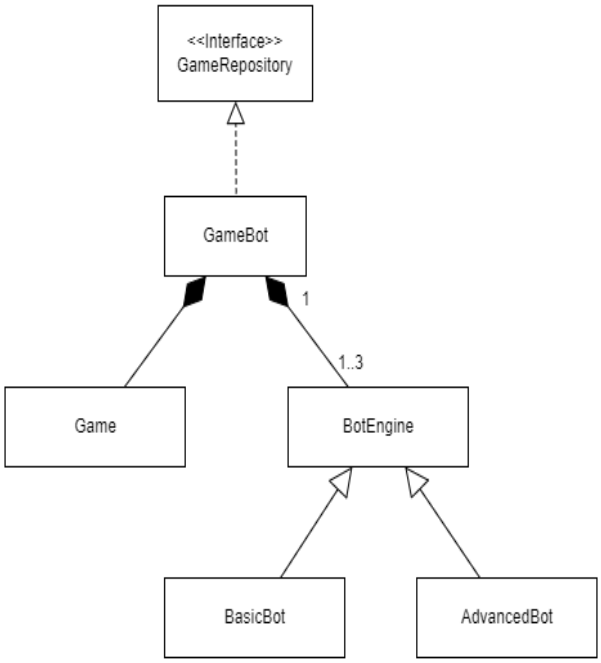
Klasa *AuthFirebase* jest w relacji z klasą *AuthenticationProvider* w postaci agregacji całkowitej (kompozycji). Oznacza to, że klasa główna (*AuthFirebase*) ma na własność klasę częściową (*AuthenticationProvider*) i kontroluje cykl życia części. Natomiast pomiędzy *AuthFirebase*, a *FirestoreProvider* występuje

relacja agregacji częściowej. To znaczy, że element częściowy (*FirestoreProvider*) należy do elementu głównego (*AuthFirebase*), ale nie jest od niego zależny. Usunięcie elementu głównego, nie wpłynie na element częściowy. W przypadku *GameFirebase* występuje ona w relacji częściowej z *FirestoreProvider*, a w relacji agregacji całkowitej z *FirestoreProvider*. Zatem z klasy *FirestoreProvider* korzysta się w obu klasach implementujących interfejsy warstwy danych.

Klasa *AuthenticationProvider* odpowiada za połączenie się z usługą *Firebase Authentication* i przeprowadzanie operacji zarejestrowania, zalogowania i wylogowania użytkownika.

Klasa *FirestoreProvider* odpowiada za połączenie się z usługą *Firebase Firestore* (baza danych czasu rzeczywistnego). Klasa ta czyta lub zapisuje dane do bazy danych. Ponadto tworzy strumienie, które nasłuchują zmiany zachodzące w wybranych miejscach w bazie danych.

Klasa *FunctionsProvider* odpowiada za połączenie się z usługą *Firebase Functions*. Wywołuje dedykowane funkcje po stronie serwera. W przypadku bota zdefiniowana jest jedna klasa *GameBot*, która implementuje interfejs *GameRepository*. Klasa ta zarządza instancją gry, która jest tworzona lokalnie na urządzeniu użytkownika. Biorąc pod uwagę rozwiązanie w postaci bota, konto użytkownika nie jest w użyciu.



Rysunek 4.3: Diagram klas warstwy danych bot.

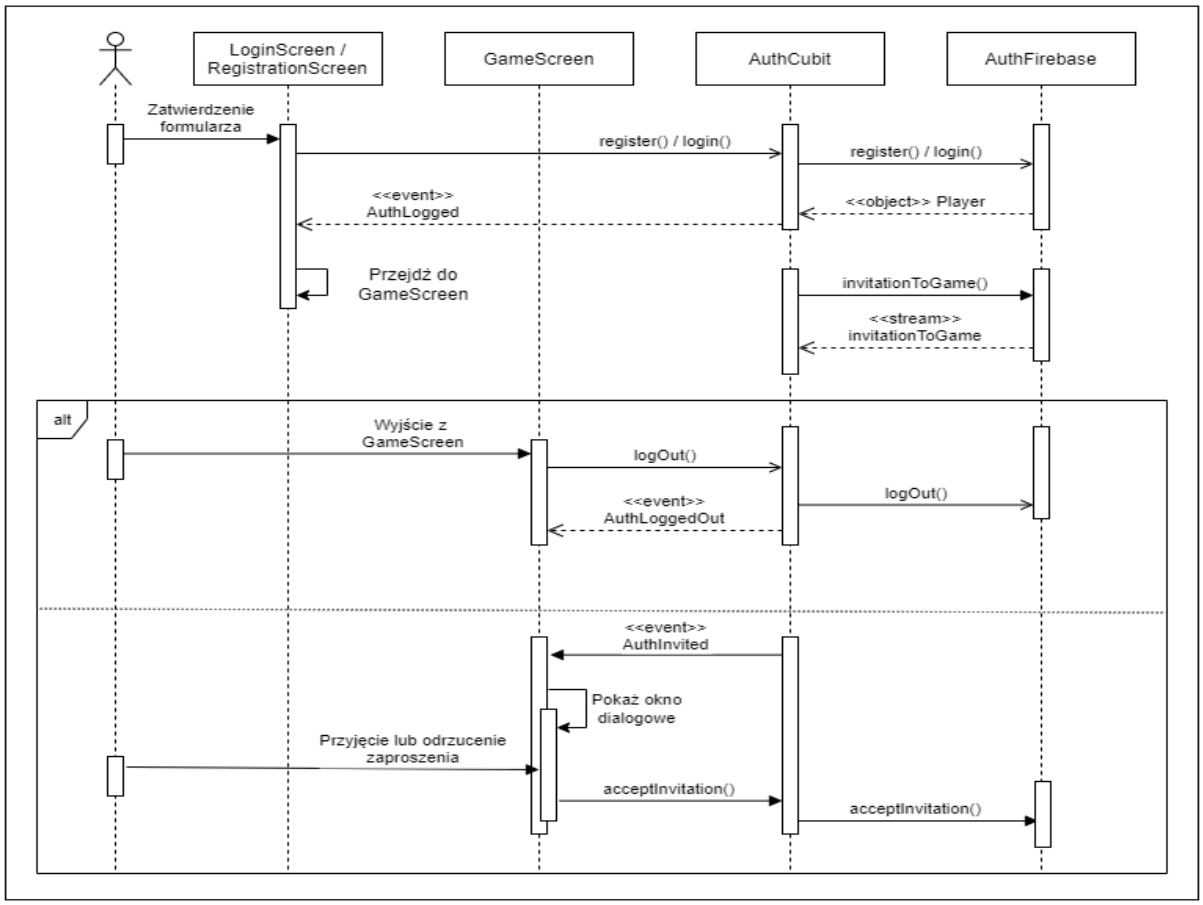
Klasa *GameBot* występuje w relacji agregacji całkowitej z klasami *Game* oraz *BotEngine*. Klasa główna może zawierać od 1 do 3 instancji klasy *BotEngine*. To ograniczenie wzięło się z zasady, że w grę Rummikub może grać od 2 do 4 graczy. Klasy *BasicBot* i *AdvancedBot* dziedziczą po klasie *BotEngine*.

Klasa *Game* jest instancją gry. Tworzy wszystkie możliwe kości w grze i losowo przydziela każdemu uczestnikowi. Przechowuje w sobie listę dostępnych kości do rozdania, listę kości przypisanych każdemu graczowi oraz zbiór zestawów kości, które są obecnie na planszy.

Klasa *BotEngine* ma na celu imitować prawdziwego gracza. Analizuje obecny układ kości na plaszy oraz swoje własne kości i na tej podstawie wykonuje wyłożenie. Klasy *BasicBot* oraz *AdvancedBot* różnią się zaimplementowanym algorytmem wykonującym ruch. Algorytmy te wraz z opisem złożoności gry są przedstawione w następnym rozdziale.

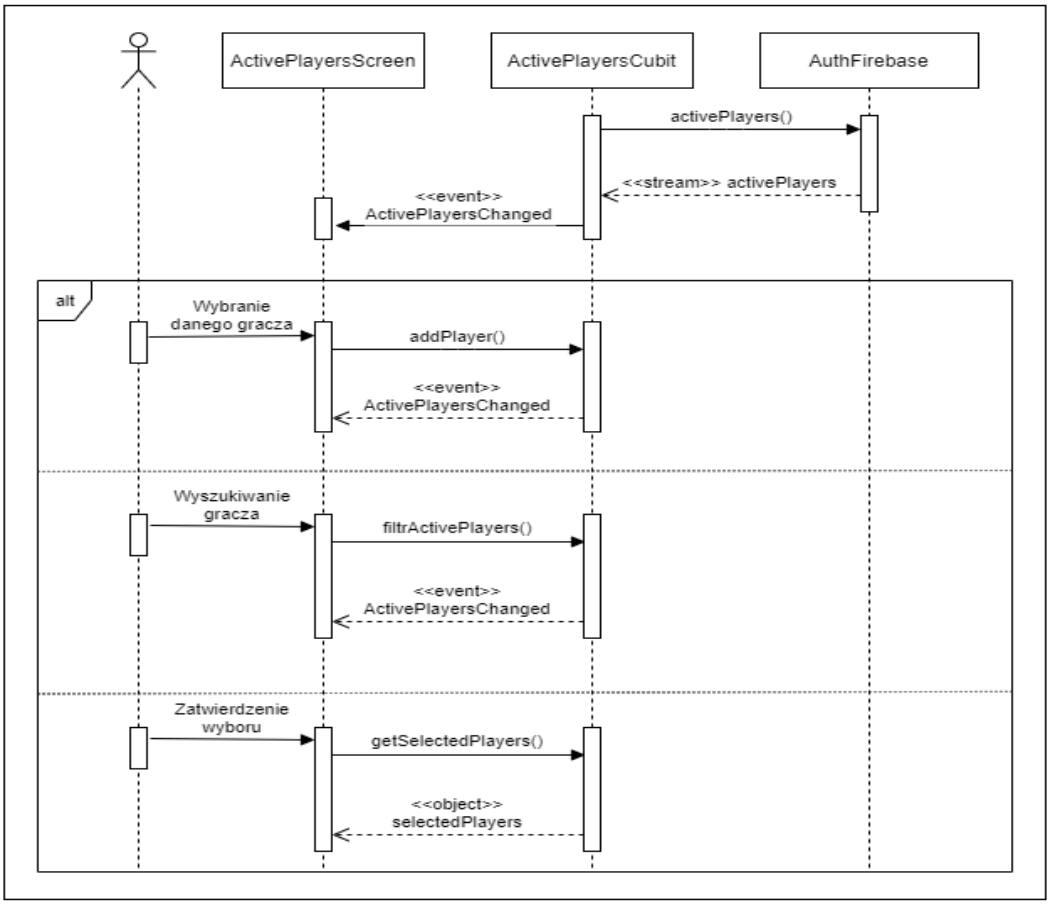
4.2.4 Przepływ zdarzeń

W poprzednich podrozdziałach przedstawiono opisy poszczególnych warstw. Ostatni podrozdział dotyczy przepływów zdarzeń pomiędzy warstwami, aby zobrazować mechanikę działania systemu. Do przedstawienia graficznego wykorzystano diagramy sekwencji w notacji UML. Dla czytelności diagramów metody są zapisywane bez podawania ich parametrów. W interakcjach również nie uwzględniono obsługi wyjątków. Bardziej szczegółowe opisy znajdują się poniżej każdego diagramu.



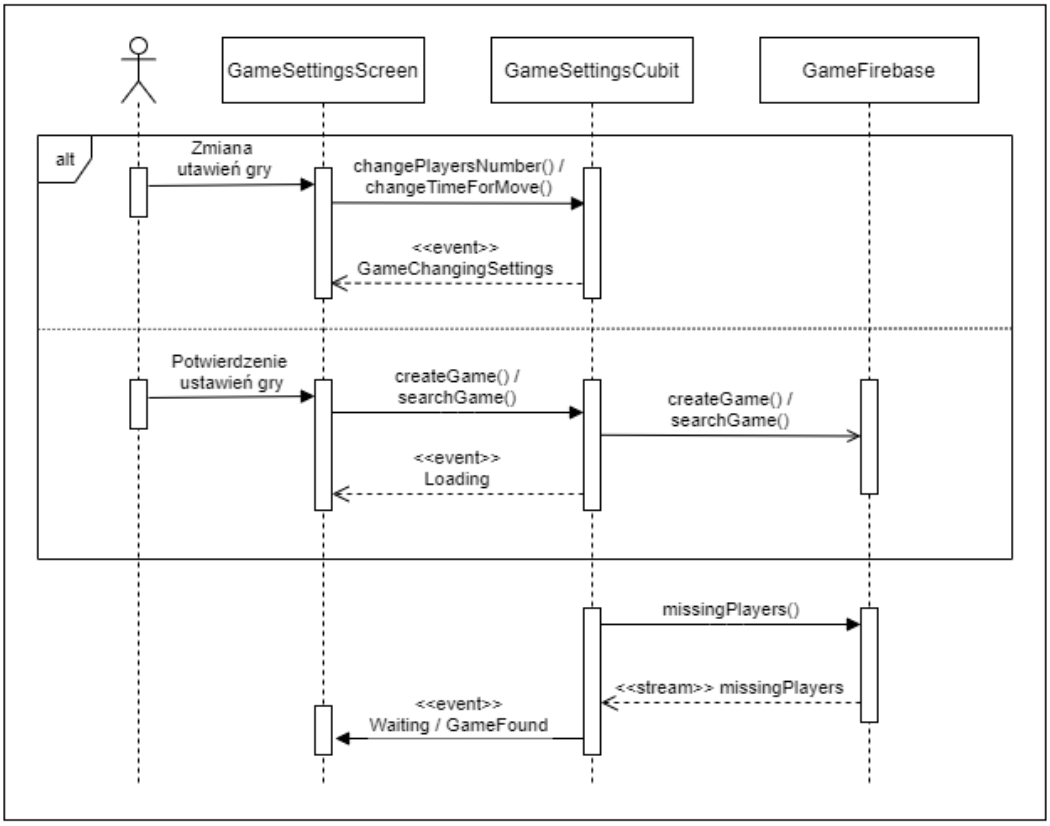
Rysunek 4.4: Diagram sekwencji obsługi konta użytkownika.

Powyższy diagram przedstawia interakcje pomiędzy warstwami, które są związane z klasą *AuthCubit*. Klasa ta po stronie warstwy prezentacji komunikuje się z klasami *RegistrationScreen*, *LoginScreen* oraz *GameScreen*. W przypadku rejestracji lub logowania klasa *AuthCubit* jest pośrednikiem między warstwą prezentacji, a warstwą danych. Wywołuje asynchroniczą metodę *login* z parametrami email oraz hasło, lub metodę *register* (rozszerzona o dodatkowy parametr: nazwa użytkownika). Z warstwy prezentacji w odpowiedzi dostajemy obiekt *Player*, który zawiera dwa atrybuty: ID gracza oraz jego nazwa. Następnie *AuthCubit* przesyła strumieniem klasę stanu *AuthLogged*, która zawiera otrzymany obiekt. *LoginScreen* bądź *RegistrationScreen* po otrzymaniu tego stanu, nawigują do strony *GameScreen*, przekazując referencję do *AuthCubit* oraz parametry gracza. Ponadto po zalogowaniu / zarejestrowaniu użytkownika klasa *AuthCubit* pobiera obiekt listenera z *AuthFirebase*, za pomocą którego będzie nasłuchiwać pojawiające się zaproszenia innych użytkowników do rozgrywki. W przypadku interakcji *GameScreen* z *AuthCubit* występują dwa od siebie niezależne procesy. Pierwszy z nich polega na obsłudze wylogowania użytkownika. Natomiast drugi odpowiada za obsługę zaproszeń do gry. *AuthCubit* po otrzymaniu komunikatu ze strumienia o zaproszeniu, emituję klasę stanu *AuthInvited*, która zawiera nazwę zapraszającego i kod ID gry. W przypadku gdy gracz odmówi udziału, zostanie przekazana metoda *acceptInvitation(false)*.



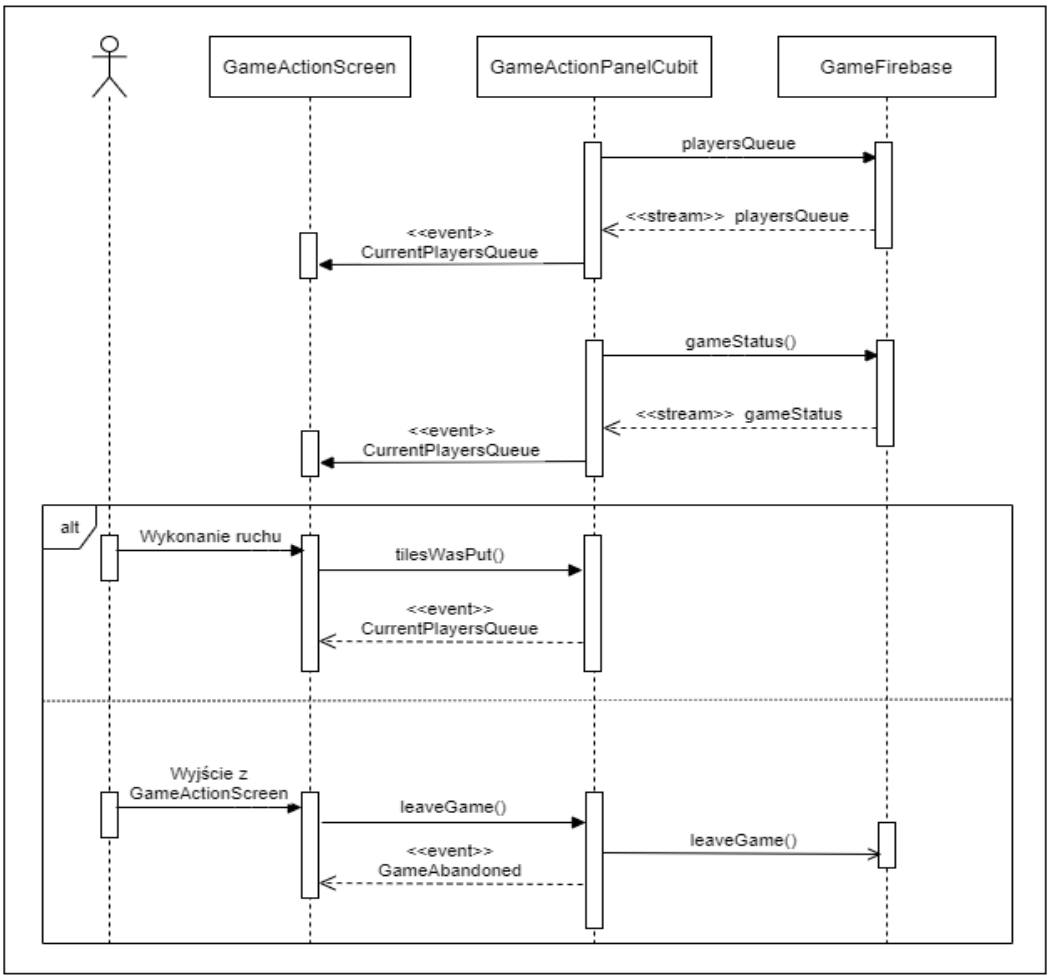
Rysunek 4.5: Diagram sekwencji wyszukiwania aktywnych użytkowników.

Proces na rysunku 4.5 rozpoczyna się pobraniem strumienia z *AuthFirebase*, dzięki któremu można otrzymywać aktualną listę aktywnych użytkowników w grze. Za każdym razem, gdy lista użytkowników ulegnie zmianie, będzie przekazywana do *ActivePlayersScreen* klasa stanu *ActivePlayersChanged*. Klasa ta zawiera w sobie aktualną listę aktywnych użytkowników, ale również zawiera listę dotychczas wybranych przez użytkownika graczy. Następnie występują trzy możliwe procesy, które odpowiadają działaniom użytkownika. W momencie, gdy gracz wybiera kogoś z listy, przekazywana jest nazwa wybranego gracza do *ActivePlayersCubit* i zwracana jest zaktualizowana klasa stanu *ActivePlayersChanged*. Użytkownik może również wyszukiwać gracza po nazwie. W tym przypadku za każdym razem zwracana jest klasa stanu, w której lista użytkowników jest odpowiednio przefiltrowana. Ostatnią możliwością jest zatwierdzenie wyboru. W takim przypadku do *ActivePlayersScreen* jest przekazywana ostateczna lista wybranych graczy do zaproszenia.



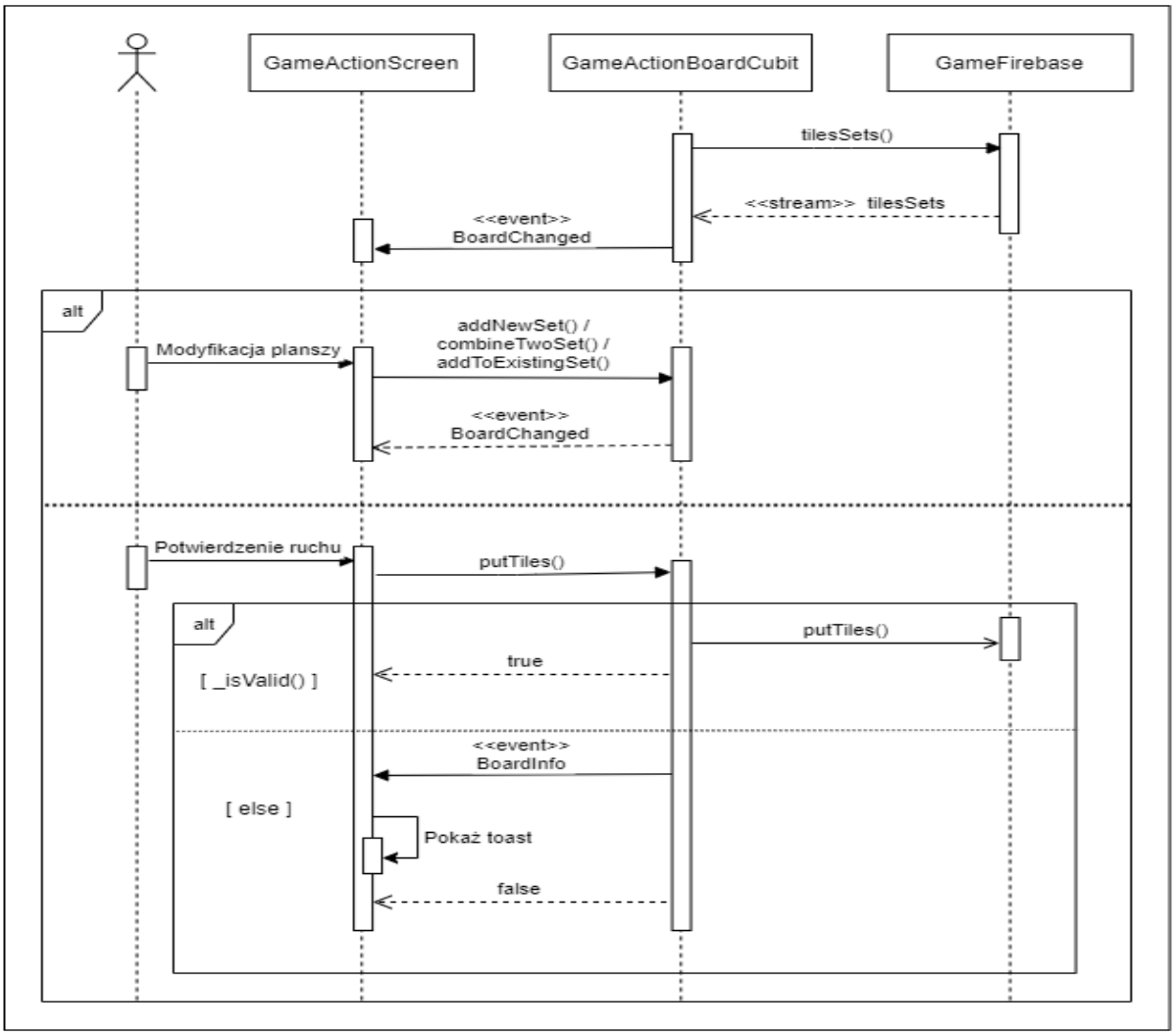
Rysunek 4.6: Diagram sekwencji konfiguracji gry.

Dwa początkowe procesy ukazane na diagramie zależą od tego z jakimi parametrami została utworzona klasa *GameSettingsCubit*. W przypadku, gdy gracz szuka gry do dołączenia może odpowiednio skonfigurować ustawienia gry, zmieniając liczbę graczy oraz czas przeznaczony na wykonanie ruchu. Jeśli jednak gracz wybrał opcję stworzenia własnej gry z wybranymi graczami, może jedynie zmienić czas przeznaczony na ruch. Klasa stanu *GameSettingsChanged* zawiera aktualnie wybrane ustawienia. Gdy użytkownik potwierdzi ustawienia za pośrednictwem *GameSettingsCubit* w *GameFirebase* jest wywoływana metoda *createGame* (parametry: kod ID gracza, lista nazw zaproszonych graczy, czas przeznaczony na ruch w grze) lub metoda *searchGame* (parametry: kod ID gracza, liczba graczy w grze, czas przeznaczony na ruch w grze). *GameSettingsScreen* po otrzymaniu eventu *Loading*, zastępuję dotychczasowy widok obracającym się kółeczkiem. W ostatnim procesie metoda *missingPlayers* z parametrem ID gry, zwraca strumień, który powiadamia o dołączających graczach do gry. Klasa stanu *Waiting* zawiera liczbę brakujących graczy do rozpoczęcia gry. Klasa *GameFound* zawiera ID gry i wskazuje, że gra jest już gotowa do rozpoczęcia. W przypadku, gdy gracz przyjął zaproszenie do gry wykona się jedynie ostatni proces z pominięciem dwóch pierwszych, ponieważ gracz nie ma już wpływu na ustawienia gry.



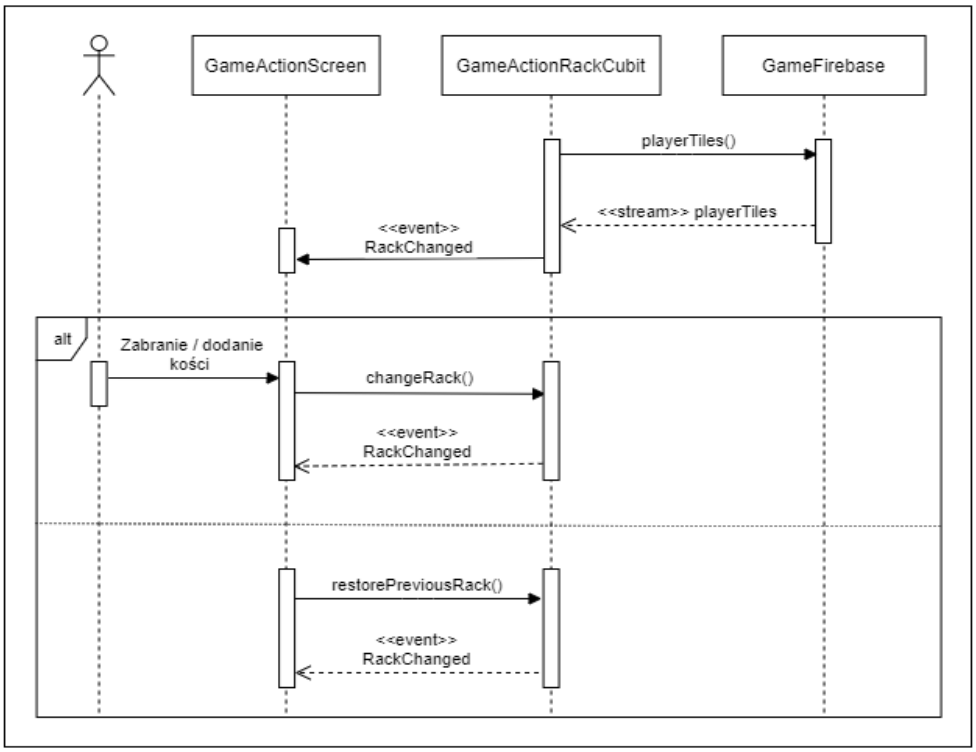
Rysunek 4.7: Diagram sekwencji panelu sterowania gry.

Klasa *GameActionPanelCubit* jest odpowiedzialna za zarządzanie stanem gry. Korzysta z dwóch strumieni danych. Strumień *playersQueue* przekazuje aktualną listę uczestników gry, a *gameStatus* wskazuje, jaki gracz ma obecnie prawo ruchu oraz czy pojawił się już zwycięzca gry. Informacje te są przekazywane do *GameActionScreen*. Klasa stanu *CurrentPlayersQueue* zawiera listę uczestników w grze oraz czas, który pozostał dla gracza mającego prawo ruchu. Kolejny proces przedstawia sytuację, gdy użytkownik wykonał wyłożenie kości. Wtedy informuje o tym *GameActionPanelCubit*, które zeruje i przestaje odliczać czas jego ruchu. Ostatni proces dotyczy zdarzenia, gdy użytkownik zamierza opuścić grę przed jej zakończeniem. Cubit wywołuje metodę *leaveGame* z parametrami: ID gracza oraz ID gry.



Rysunek 4.8: Diagram sekwencji planszy rozgrywki.

Klasa *GameActionBoardCubit* jest odpowiedzialna za zarządzanie planszą gry. Pobiera strumień z *GameFirebase*, który będzie przekazywał modyfikacje planszy przeprowadzone przez innych użytkowników. Klasa stanu *BoardChanged* zawiera listę zbiorów kości. Następnie są obsługiwane dwa procesy dotyczące planszy. Proces pierwszy aktualizuje stan planszy za każdym razem, gdy gracz dokona modyfikacji pod warunkiem, że ma prawo ruchu. Modyfikacje mogą odbywać się trzema sposobami: *addNewSet* dodanie kości do planszy tworzy nowy zbiór, *combineTwoSet* dodanie kości łączy dwa zbiory, *addToExistingSet* kość jest dodwana do istniejącego zbioru. Drugi proces następuje, gdy gracz chce zatwierdzić swoje zmiany przeprowadzone na planszy. W takim przypadku *GameActionBoardCubit* przeprowadza walidację stanu planszy. Metoda *_isValid* wskazuje, czy otrzymany zbiór kości jest zgodny z zasadami gry. Jeśli tak to zostanie wywołana funkcja dodania kości po stronie *GameFirebase*, w przeciwnym wypadku zostanie wyświetlone powiadomie o tym, że plansza nie jest ułożona w poprawny sposób. W momencie, gdy czas gracza minął, a ten nie zdążył wyłożyć kości o poprawnej konfiguracji, plansza wraca do stanu sprzed modyfikacji ze strony danego gracza.



Rysunek 4.9: Diagram sekwencji zestawu kości gracza.

Klasa *GameActionRackCubit* jest odpowiedzialna za zarządzanie zbiorem kości gracza. Pobiera strumień z *GameFirebase*, który będzie przekazywał zmiany zachodzące w zbiorze kości (w przypadku braku wyłożenia gracz będzie otrzymywał kość z banku). Klasa stanu *RackChanged* zawiera listę kości. W momencie, gdy gracz wykłada kości na plansze, albo z powrotem wkłada (może wziąć z planszy tylko swoje kości), klasa *GameActionRackCubit* jest informowana i aktualizuje listę kości gracza. W przypadku gdy czas gracza minął jest wykonywana funkcja *restorePreviousRack*, poprzez którą zmiany dokonane przez gracza w czasie swojego ruchu są anulowane. *GameActionRackCubit* zapamiętuje stan zestawów kości w momencie rozpoczęcia ruchu przez gracza.

Rozdział 5

Bot

W ostatnim rozdziale przedstawiono analizę złożoności gry Rummikub oraz opisano jakie podejścia zostały zrealizowane do zaprogramowania bota. Podczas rozważania złożoności gry oraz implementacji algorytmu dla zaawansowanego bota skorzystano z artykułu „The Complexity of Rummikub Problems" [3]. Zauważmy, że w związku z zastosowaniem architektury trójwarstwowej, po stronie klienta bot zastępuje warstwę danych bez konieczności modyfikowania logiki aplikacji.

5.1 Podstawowy bot

Implementacja podstawowego bota miała na celu umożliwienie przeprowadzanie rozgrywek na niskim poziomie trudności. Jest on przeznaczony dla osób, które dopiero poznają grę Rummikub, a jego zasada działania polega na formowaniu serii oraz grup (patrz: następny rozdział) z własnego zbioru kości. Ponadto bot może modyfikować zbiory znajdujące się na planszy poprzez dodanie swoich kości na ich początku lub końcu danego zbioru.

5.2 Złożoność gry

Gra Rummikub składa się z 106 kości, wśród których są dwa jokery. Joker może reprezentować sobą dowolną inną kość. Zatem 104 kości są numerowane od 1 do 13, w czterech różnych kolorach i każda kość występuje dwa razy. Kości można grupować w serie (rosnący ciąg liczb o tym samym kolorze) lub w grupy (kości o takiej samej liczbie z różnymi kolorami). Problemem jest znalezienie najlepszego wyłożenia podczas danego ruchu.

Liczbę sposobów ułożenia grup z danej puli kości oznaczono jako $G(k, m)$, gdzie k to jest liczba kolorów, a m to jest liczba kopii. Na początek wzięto pod rozważanie przypadek, gdy $k = 4$, a $m = 1$. W takim przypadku dla danej wartości kości można nie formować żadnej grupy, formować cztery grupy o długości trzy lub jedną o długości cztery. Zatem $G(4,1) = 1 + \binom{4}{3} + 1 = 6$. W przypadku, gdy $m = 2$ pojawia się kopia każdej kości, a zatem są dwa zestawy, dla których można policzyć $G(4,1)$. Czyli $G(4,2) < G(4,1)^2$. Jest to górna granica, ponieważ pewne przypadki zostaną policzone więcej niż raz. Podczas tworzenia grup nie bierze się pod uwagę kolejności w jakiej te grupy są tworzone oraz przypadki, które łącznie dają taki sam zbiór kości, nie powinny być rozróżnialne. Pomimo tego, że ograniczenie górne nie jest ścisłe, liczba możliwych sposobów utworzenia grup rośnie nadal wykładniczo, powołując się na artykuł [3]. Z tego powodu w implementacji algorytmu na początku formuje się serie, a dopiero później grupy z pozostałych kości.

Wybierając podejście, w którym najpierw formułuje się serie, a później grupy, dzieli się zbiór wszystkich kości (plansza oraz kości gracza) ze względu na wartość danej kości. Dla każdego podzbioru w kolejności od kości o wartościach 1 aż do wartości 13 tworzy się wszystkie możliwe kombinacje użycia poszczególnych kości do danych serii. Natomiast z kości, które pozostały w podzbiorach tworzy się możliwie jak największe grupy.

Zatem wystarczy dla każdego podzbioru o danej wartości kości, zdefiniować wszystkie możliwe do uzyskania konfiguracje serii. Aby seria spełniała zasady gry musi mieć co najmniej długość 3. Z tego powodu serie koduje się za pomocą cyfr 0, 1, 2, 3. Każdy stan gry będzie przedstawiony w postaci *[wartość, konfiguracja serii]*.

W grze Rummikub możliwe jest stworzenie ośmiu pełnych serii (zbiór kości 1-13). Z tego powodu, że występują cztery kolory oraz każda kość ma swoją kopię. Zatem stan gry będzie określony jako $[v, abcdefgh]$, gdzie $v \in \{1..13\}$ oraz $(a, b, c, d, e, f, g, h) \in \{0, 1, 2, 3\}$. Założono, że (a, b) przedstawiają serie koloru czarnego, (c, d) koloru niebieskiego, (e, f) koloru pomarańczowego, (g, h) koloru czerwonego.

Dla przykładu, stan gry składający się z serii kości $(2, 3, 4, 5)$ koloru niebieskiego oraz $(1, 2, 3)$ koloru czerwonego, będzie zakodowany w następujący sposób:

$$[1, 00000001], [2, 00010002], [3, 00020003], [4, 00030000], [5, 00030000], [6, 00000000], \ \ldots\ , [13, 00000000]$$

5.3 Zaawansowany bot

Podczas implementacji bota zmodyfikowano algorytm opisany w artykule [3]. Mianowicie algorytm *MaxScore* oprócz zwracania największej możliwej sumy wartości wszystkich wyłożonych na planszę kości, zwraca również konfigurację serii, która dla danej wartości kości dała najlepszy wynik oraz gwarantuje, że wszystkie kości na planszy będą ponownie wykorzystane.

Pseudokod 5.1: MaxScore

Input: *value*, *runs*, *tableTiles*
Output: *Result*

```
1  if value > 13 then
2      return Result(0, zeroRuns);
3  if results[value, runs] ≠ null then
4      return results[value, runs];
5  results[value, runs] ← Result(−∞, zeroRuns);
6  for possibleRuns ∈ MakeRuns(value, runs) do
7      runScores ← GetRunScores(possibleRuns, runs, tableTiles, tiles, value);
8      groupScores ← GetGroupScores(tableTiles, tiles, value);
9      if runScores = −1 or groupScores = −1 then
10         continue;
11     maxScore ← MaxScore(value + 1, possibleRuns, tableTiles);
12     sum ← runScores + groupScores + maxScore.scores;
13     if results[value, runs].scores < sum then
14         results[value, runs] ← Result(sum, possibleRuns);
```

Algorytm jest wykonywany rekurencyjnie dla kolejnych wartości kości $(1 - 13)$. Parametrem wyjściowym jest obiekt *Result*, który zawiera dwa pola: *scores* (najlepszy wynik dla danej wartości kości) oraz *runs* konfiguracja serii, która dała najlepszy wynik. Na początku oprócz sprawdzenia czy dana wartość *value* nie przekroczyła maksymalnej wartości, sprawdza czy dany stan gry, nie został już poprzednio obliczony. *zeroRuns* oznacza konfigurację serii, gdzie wszystkie serie mają długość zero. Następnie następuje iteracja po wszystkich możliwych nowych konfiguracjach serii. Funkcja *MakeRuns* tworzy nowe konfiguracje na podstawie obecnej konfiguracji serii i dostępnych kościach o wartości *value*. Funkcja *GetRunScores* oblicza punkty za serie na podstawie porównania nowej konfiguracji *possibleRuns* z poprzednią *runs*. *GetGroupScores* z kolei zwraca liczbę punktów otrzymanych za grupy kości o wartości *value*. Obie funkcje zwracają wartość -1, gdy kości należące do planszy nie zostaną wyłożone. W takim przypadku dane *possibleRuns* jest odrzucane i następuje dalsza iteracja. W przeciwnym wypadku jest wywoływana rekurencyjnie funkcja *MaxScore*, której przekazywana wartość kości powiększona o jeden, nowa konfiguracja kości oraz *tableTiles*. Po zsumowaniu wyniku sprawdzane jest, czy otrzymywany wynik jest obecnie największym możliwym wynikiem dla tego stanu gry. Najlepszy wynik otrzymany w poszczególnym stanie

jest zapisywany do tablicy wszystkich stanów wraz z konfiguracją serii, która została wybrana dla tego stanu.

Po wykonaniu algorytm zwraca najlepszy możliwy wynik z pierwszą wybraną konfiguracją serii. Dzięki temu, że dla każdej wartości *value* wynik jest zapisywany do tablicy *result* z indeksem $[value, runs]$ można odtworzyć przebieg konfiguracji serii. Zatem iterując po wszystkich wartościach *value* można odtworzyć utworzone serie. Wiedząc jakie kości zostały użyte do stworzenia serii można wygenerować grupy dla poszczególnych wartości kości. W implementacji zaawansowanego bota pominięto obecność jokerów w grze. Ich dodanie wprowadziłoby pewne modyfikacje w algorytmie $MaxScore$. Kwestia ta pozostała w aspekcie dalszego rozwoju serwisu.

Poniżej omówiono działanie metody *MakeRuns* wywołanej w linii 6 algorytmu 5.1. Jako parametry wejściowe przyjmuje ona wartość kości *value* oraz konfigurację serii *runs* otrzymaną z poprzedniej wartości kości. Funkcja zlicza liczbę dostępnych kości o danej wartości *value* i na tej podstawie tworzy wszystkie możliwe nowe konfiguracje serii. Dla każdego koloru mamy dwie serie. W zależności od liczby dostępnych kości o tym kolorze są trzy możliwości modyfikacji: niedodanie kości do żadnej serii, dodanie kości do jednej z serii, dodanie kości do dwóch serii. W przypadku, gdy tylko jedna seria zostanie przedłużona, a serie te nie różnią się długością, nie rozróżnia się, do której serii została dodana kość. Gdy do danej serii nie jest dodawana kość, seria ta jest przerywana. W poniższych przykładach dla uproszczenia wzięto pod uwagę jedynie modyfikacje serii koloru czarnego.

Przykład 5.1 *Konfiguracja serii runs ma postać* - 010000000*, więc zawiera jedną serię koloru czarnego o długości* 1*. Do dyspozycji są dwie kości koloru czarnego o danej wartości. Możliwe do otrzymania nowe konfiguracje serii wyglądają następująco:* 000000000 *(nie dodanie żadnej kości do serii koloru czarnego),* 020000000 *(dodanie kości do drugiej serii koloru czarnego),* 100000000 *(dodanie kości do pierwszej serii koloru czarnego),* 120000000 *(dodanie kości do obu serii koloru czarnego).*

Przykład 5.2 *Konfiguracja serii runs ma postać* - 110000000*, więc zawiera dwie serie koloru czarnego o długości* 1*. Do dyspozycji są dwie kości koloru czarnego o danej wartości. Możliwe do otrzymania nowe konfiguracje serii wyglądają następująco:* 000000000 *(nie dodanie żadnej kości do serii koloru czarnego),* 020000000 *(dodanie kości do drugiej serii koloru czarnego),* 220000000 *(dodanie kości do obu serii koloru czarnego).*

Przykład 5.3 *Konfiguracja serii runs ma postać* - 130000000*, więc zawiera dwie serie koloru czarnego o długości* 1 *oraz* 3*. Do dyspozycji są dwie kości koloru czarnego o danej wartości. Możliwe do otrzymania nowe konfiguracje serii wyglądają następująco:* 000000000 *(nie dodanie żadnej kości do serii koloru czarnego),* 030000000 *(dodanie kości do drugiej serii koloru czarnego),* 200000000 *(dodanie kości do pierwszej serii koloru czarnego),* 230000000 *(dodanie kości do obu serii koloru czarnego).*

Poniżej omówiono strukturę *tableTiles* wykorzystaną w algorytmie 5.1. Struktura zawiera dla każdego koloru dwa liczniki kości (jeden licznik dla $value - 2$, drugi licznik dla $value - 1$), które znajdują się na planszy, a zostały użyte w seriach o długości mniejszej niż 3. Jest ona potrzebna, aby mieć pewność, że wszystkie kości znajdujące się na planszy ponownie zostały wyłożone. Dla przypadku, gdy seria o długości mniejszej niż 3 zostanie przerwana, kości nie są już brane pod uwagę w dalszych wyłożeniach. W przypadku, gdy wśród nich będzie kość należąca do planszy, złamana zostanie zasada gry.

Poniżej omówiono działanie metody *GetRunScores* wywołanej w linii 7 algorytmu 5.1. Jako parametry wejściowe przyjmuje nową konfigurację serii *possibleRuns*, poprzednią konfigurację serii *runs*, *tableTiles*, dostępne kości *tiles*, daną wartość kości *value*. Metoda ta zwraca wynik na podstawie porównania *possibleRuns* z *runs*. W poniższych przykładach dla uproszenia wzięto jedynie pod uwagę modyfikacje jednej serii koloru czarnego.

Przykład 5.4 *Konfiguracja serii runs* - 010000000*, nowa konfiguracja serii possibleRuns* - 020000000*. Ze zbioru tiles usuwana jest czarna kość o wartości value. W tableTiles licznik* $value - 1$ *staje się licznikiem* $value - 2$*, a licznik* $value - 1$ *wynosi* 1*, gdy usunięta kość znajdowała się na planszy. Seria ma długość mniejszą niż 3, więc zwracany jest wynik* 0*.*

Przykład 5.5 *Konfiguracja serii runs* - 020000000*, nowa konfiguracja serii possibleRuns* - 000000000*. Seria została przerwana, więc kości koloru czarnego o wartościach value* $-$ 2 *oraz value* $-$ 1 *są dodawane do zbioru tiles. Na podstawie tableTiles można wywnioskować, czy te kości należą do planszy. W takim przypadku zwrócony będzie wynik* -1*, w przeciwnym razie zwracany jest wynik* 0*.*

Przykład 5.6 *Konfiguracja serii runs* - 020000000*, nowa konfiguracja serii possibleRuns* - 030000000*. Z tiles usuwana jest czarna kość o wartości value. Liczniki w tableTiles są zerowane z powodu zakończenia serii. Zwracany jest wynik value* $-$ 2 $+$ *value* $-$ 1 $+$ *value.*

Przykład 5.7 *Konfiguracja serii runs* - 030000000*, nowa konfiguracja serii possibleRuns* - 030000000*. Zwracany jest wynik value.*

Poniżej omówiono działanie metody *GetGroupScores* wywołanej w linii 8 algorytmu 5.1. Jako parametry przyjmuje *tableTiles*, dostępne kości *tiles*, daną wartość kości *value*. Funkcja ta sprawdza, czy kości o wartości *value* są w stanie utworzyć grupę/y (grupa zawiera minimum 3 kości). Usuwa wykorzystane kości ze zbioru *tiles*. Zwraca wynik *value* pomnożony przez liczbę wykorzystanych kości. W przypadku, gdy kość w grupie nie należy do planszy, natomiast kopia tej kości, która należała do planszy, została użyta w serii o długości mniejszej niż 3, kość ta nie jest już uwzględniana w *tableTiles*. W przypadku grup mamy pewność, że kości zostaną wyłożone. Ze względu na to, że każda kość ma swoją kopię w grze, w trakcie tworzenia serii oraz grup priorytetowo są brane kości należące do planszy.

Podsumowanie

Obecnie serwis jest gotowy do wdrożenia. Wszystkie wskazane funkcjonalności zostały zaimplementowane. Serwis działa zarówno na stronie w przeglądarce jak i w postaci aplikacji mobilnej na telefon z systemem Android. Użytkownicy mogą zakładać własne konta i grać z innymi użytkownikami przez internet lub z botem bez potrzeby zakładania konta i łączności z internetem.

Główny fundament serwisu został stworzony, więc obecnie można rozszerzać serwis o nowe funkcjonalności. Najbardziej pożądaną kwestią jest dodanie bota po stronie serwera. Funkcjonalność ta przydałaby się w momencie, gdy chcemy zagrać w grę z większą liczbą osób, a niestety w pewnej chwili w serwisie nie ma zbyt wielu graczy. Ponadto należałoby rozwinąć możliwości związane z kontami użytkowników. Udostępniać statystyki dotyczące danego użytkownika, wprowadzić możliwość komunikacji tekstowej między graczami. Warto byłoby też pochylić się nad ulepszeniem wizualizacji aplikacji (dodaniem animacji czy efektów głosowych). Ważną kwestią jest również dalsza praca nad algorytmem zaawansowanego bota i uwzględnienie w nim możliwości wystąpienia dwóch dodatkowych kości z jokerami.

W projekcie tym udało się pomyślnie wykorzystać technologie Flutter oraz Firebase. Wymagało to przeznaczenia pewnej ilości czasu na ich naukę z uwagi na to, że są to dosyć młode technologie oraz serwis ten jest pierwszym większym projektem korzystającym z tych technologii dla autora. Z pewnością zapoznanie się z wieloplatformową technologią budowania aplikacji było cennym doświadczeniem. Wartością dodaną tego projektu jest szczególnie możliwość zapoznania się ze złożonością gry Rummikub.

Bibliografia

[1] Firebase technology. Web pages: https://firebase.google.com/.

[2] A. Miola. *Flutter Complete Reference*. 2020.

[3] J. N. van Rijn, F. W. Takes, J. K. Vis. The complexity of rummikub problems. 2016.

Załącznik A

Zawartość płyty CD

Do pracy dyplomowej została dołączana płyta CD, na której znajduje się projekt zawierający kod źródłowy zaimplementowanego serwisu. Zamieszczony został również plik binarny (o rozszerzeniu apk) umożliwiający instalację aplikacji na telefonie z systemem Android. Ponadto umieszczone zostały zdjęcia obrazujące wygląd serwisu.