Tematyka i cele projektu

Projekt ***Random Chess*** został stworzony w celu ogranizacji treningu szachowego poprzez agregowanie partii z różnych źródeł oraz rozgrywania lokalnych partii z botami offline. Zaimpelementowane funkcjonalności obejmują:

- Rozgrywanie partii w aplikacji z botem
  - Weryfikacja legalności ruchów graczy
  - Integracja z istniejącymi silnikami szachowymi - obecnie Stockfish na 4 różnych poziomach trudności
- Historia rozegranych partii, włączając:
  - Partie zaimportowane w formacie PGN – Portable Game Notation
  - Partie automatycznie pobierane z połączonych kont na portalu lichess.org
  - Partie rozegrane w naszym serwisie
- Możliwość eksportowania partii z historii w formacie PGN
- Analiza partii
  - Rozpoznawanie debiutów
  - Wyświetlanie ruchów w notacji algebraicznej
- Utrzymanie struktury klient-serwer w celu potencjalnego rozszerzenia projektu
- Szacowanie rankingu ELO wśród partii rozegranych lokalnie - osobno dla różnych rodzajów temp rozgrywki
- Przeprowadzanie turniejów w systemie szwajcarskim (tylko w bazie, w aplikacji moduł do parowania graczy)

Aplikacja

Projekt realizowaliśmy w ramach Inżynierii Danych i Programowania Obiektowego.
Wraz ze schematem bazy danych powstała aplikacja desktopowa,
w obecnej wersji łącząca serwer i interfejs użytkownika i w jednym pliku wykonywalnym.

Dokładne instrukcje znajdują się w pliku `README.md` w folderze z kodem źródłowym.

Przygotowanie bazy danych

Aplikacja domyślnie oczekuje, aby w systemie baz danych:

- istniał użytkownik `random_chess` z hasłem `random_chess`
- istniała baza danych `random_chess` z użytkownikiem `random_chess` jako właścicielem.

Skrypt `extra/create.sh` tworzy użytkownika i bazę danych, tworzy schemat bazy danych i wczytuje przykładowe dane.
Plik `create.sql` zawiera te same polecenia SQL, z wyjątkiem tworzenia użytkownika i bazy danych.

Budowanie aplikacji

Przed zbudowaniem aplikacji wymagane jest uruchomienie bazy danych,
bo podczas kompilacji generowane są klasy na podstawie schematu bazy danych.

Pliki wymagane do działania aplikacji

Należy skopiować plik z przykładową konfiguracją extra/config.example.yml do folderu:

- na Linux: `~/.local/share/rchess/config.yml`
- na Windows: `%APPDATA%/rchess/config.yml`
- na macOS: `~/Library/Application Support/rchess/config.yml`

Wymagane jest też pobranie programu Stockfish, aby w aplikacji działały rozgrywki z botami.
Należy pobrać plik wykonywalny Stockfish odpowiedni do platformy,
na której będzie uruchamiany oraz pliku
nn-1c0000000000.nnue.
Te pliki należy umieścić w tym samym folderze co `config.yml`.

Schemat bazy

Typy i domeny

game_result

Domena `game_result` na typie `game_result_type` przechowuje informacje o rozstrzygnięciu partii oraz o powodzie zakończenia gry, np. `(1-0,TIMEOUT)` lub `(1/2-1/2,FIFTY_MOVE_RULE)`

clock_settings

Domena `clock_settings` na typie `clock_settings_type` przetrzymuje informacje o tempie partii - czasie przydzielanym początkowo oraz czasie przyznawanym po każdym ruchu.

Tabele

openings

Pole	Typ	Dodatkowe informacje
`id`	SERIAL	**PRIMARY KEY**
`eco`	CHAR(3)	NOT NULL
`name`	VARCHAR(256)	NOT NULL
`partial_fen`	VARCHAR	UNIQUE NOT NULL

Tabela ta przechowuje dane o wszystkich debiutach szachowych poprzez ich nazwę, kod z Encyclopedia of Chess Opening, nazwę i pozycję reprezentującą ten debiut.
Debiut charakteryzowany jest przez układ szachownicy, gracza przy ruchu oraz możliwość wykonania roszady/en passant przez obu graczy.
Dane do tej tabeli importujemy z bazy danych lichess.org - w tym celu przygotowaliśmy instrukcję oraz skrypt.

users

Pole	Typ	Dodatkowe informacje
`id`	SERIAL	**PRIMARY KEY**
`email`	VARCHAR	UNIQUE NOT NULL
`password_hash`	VARCHAR	NOT NULL

W tej tabeli przechowywane są dane o lokalnych użytkownikach. Tabela ta jest stosunkowo niewielka ze względu na to, że większość danych przechowywana jest w bardziej uniwersalnej `service_accounts`.
Kolumna `elo`, która znajdowała się tutaj poprzednio została zastąpiona kompletnym i funkcjonalnym systemem rankingowym ELO na podstawie lokalnie rozgrywanych partii.
Poprawność adresu mailowego jest obłożona checkiem na podstawie wyrażenia regularnego.

game_services

Pole	Typ	Dodatkowe informacje
`id`	SERIAL	**PRIMARY KEY**
`name`	VARCHAR(256)	UNIQUE NOT NULL

Jest to dość niewielka tabela przechowująca różne źródła gier niezaimportowanych przez PGN.
Przechowujemy tutaj informację o serwisach, z którymi aplikacja wspiera integrację.
W szczególności, pod indeksem `1` znajduje się wpis `Random Chess`, opisujący lokalne partie.
To oznacza, że lokalne partie traktujemy na podstawowym poziomie tak samo, jak te zaimportowane z innych serwisów, co pozwala nam przechowywać je w jednej tabeli `service_games`.

service_accounts

Pole	Typ	Dodatkowe informacje
`user_id`	INT	REFERENCES users(id) ON DELETE SET NULL
`service_id`	INT	NOT NULL REFERENCES game_services(id)
`user_id_in_service`	VARCHAR	NOT NULL
`token`	VARCHAR	NULL
`display_name`	VARCHAR(256)	NOT NULL
`is_bot`	BOOL	NOT NULL

`token` to wartość trzymana dla połączonych kont z zewnętrznych aplikacji, służąca do synchronizacji partii z nimi. Jest ona niepusta dla zewnętrznych kont nieusuniętych użytkowników.

Para **`(service_id, user_id_in_service)`** tworzy klucz podstawowy. Każdemu użytkownikowi serwisu szachowego może odpowiadać co najwyżej jeden użytkownik w naszym systemie kont.

Boty, zarówno w naszym serwisie, jak i serwisach zewnętrznych, posiadają `service_account`, ale z `user_id IS NULL` i `is_bot = TRUE`.

`service_account` istnieje dla każdego użytkownika, który połączył swoje konto w naszym serwisie z dowolnym serwisem szachowym, ale też dla użytkowników, którzy nie mają kont w naszym serwisie, a są dowolną ze stron partii przechowywanej w `service_games`. Podjęliśmy tę decyzję, ponieważ jeśli użytkownik taki później utworzy konto w naszym serwisie, to nie chcemy musieć dodawać tej samej partii drugi raz do `service_games` ani modyfikować partii w `service_games`. Zamiast tego, po podłączeniu konta jego `user_id` zostanie po prostu podłączone do już istniejącego `service_account` i będziemy musieli wyłącznie pobrać brakujące partie z odpowiedniego API do `service_games`.

Konsekwencją tej decyzji jest to, że nie chcemy nigdy usuwać `service_account` i zamiast tego po usunięciu użytkownika odłączamy od niego wszystkie jego `service_accounts`. Jest to realizowane poprzez ustawienie `ON DELETE SET NULL` w polu `user_id`.

Dodatkowo, każdy użytkownik naszego serwisu posiada dokładnie jedno odpowiadające konto w tabeli `service_accounts` o `service_id` = `1` (id naszego serwisu szachowego). Konto to przechowuje jego nazwę użytkownika w naszym serwisie, a jego `user_id_in_service` = `user_id`.
Jest to redundancja, ale:

1. ponieważ zdecydowaliśmy się na klucz podstawowy `(service_id, user_id_in_service)`, `user_id_in_service` musi być różne dla każdego użytkownika,
2. po usunięciu konta użytkownika jego odpowiadający `service_account` musi dalej istnieć (aby partie rozegrane z nim nie zniknęły). Te różne `service_accounts` w naszym serwisie pozostałe po usuniętych użytkownikach muszą być w jakiś sposób rozróżnialne i tym sposobem jest właśnie `user_id_in_service`.

Wyzwalacze `add_default_service_to_user`, `prevent_default_service_modification` oraz `prevent_default_service_deletion` służą upewnieniu się, że od utworzenia użytkownika po jego usunięcie jego odpowiadające konto o `sevice_id = 1` w `service_accounts` będzie zawsze istnieć.

Dodatkowo, ograniczenie `valid_system_account` w `service_accounts` upewnia się, że dla kont tych spełnione są powyższe założenia:

- dla użytkowników, póki ich konto istnieje, to `user_id_in_service = user_id`,
- dla botów `user_id IS NULL`.

Tabele service_games i pgn_games

Te tabele przechowują partie szachowe, które będą analizowane w naszej aplikacji. Tabela `service_games` przechowuje zsynchronizowane partie z zewnętrznych serwisów oraz partie rozegrane w naszym serwisie. Tabela `pgn_games` przechowuje partie, które zostały zaimportowane ręcznie przez użytkownika.

W późniejszej sekcji opisaliśmy, dlaczego zdecydowaliśmy się zamodelować partie szachowe właśnie w ten sposób.

Klucze podstawowe **`id`** w `service_games` i `pgn_games` mogą się powtarzać.

Wspólne pola w tabelach service_games i pgn_games

Pole	Typ	Dodatkowe informacje
`moves`	VARCHAR(5)[]	NOT NULL
`starting_position`	VARCHAR(100)	NOT NULL
`partial_fens`	VARCHAR[]	GENERATED ALWAYS AS (generate_fen_array(starting_position, moves)) STORED
`creation_date`	TIMESTAMPTZ	NOT NULL
`result`	GAME_RESULT	NOT NULL
`metadata`	JSOB	NULL
`clock`	CLOCK_SETTINGS	NULL

Jako że wszystkie partie posiadają duże przecięcie, niezależnie od źródła, wiele pól występuje zarówno w tabeli `service_games` jak i w `pgn_games`.

Kolumna `moves` przechowuje ruchy graczy w partii w postaci długiej algebraicznej.

Kolumna `starting_position` przetrzymuje pierwsze 4 pola formatu FEN.

Kolumna `partial_fens` przechowuje kolejne stany w postaci jak powyżej wszystkich ruchów od początku do końca partii.
Jest to kolumna wygenerowana funkcją na podstawie listy ruchów i startowej pozycji. Zdecydowaliśmy się przetrzymywać to pole na stałe, gdyż zapytania do tej tabeli są bardzo częste, a obliczenie tego pola relatywnie czasochłonne.

Kolumna `creation_date` opisuje datę rozegrania albo importu, w zależności od rodzaju partii.

Kolumna `result` typu `game_result` przetrzymuje sposób zakończenia rozgrywki, jak opisano w sekcji `Typy i domeny`.

Kolumna `metadata` zawiera wszystkie niestandardowe pola metadanych pochodzących z opisu partii w postaci PGN. Dane przechowujemy w formacie JSON, choć nie spełnia to reguły atomowości, bo dokładny ich format może się różnić w zależności od serwisu, a dane te służą jedynie do wyświetlenia użytkownikowi i ponownego eksportu rozgrywki do formatu PGN, nigdy nie będziemy wykonywać zapytań dotyczących metadanych w tym polu.

Kolumna `clock` w typie `clock_settings` przetrzymuje dane o tempie, w którym partia została rozegrana. Wartość ta poza ponownym eksportem do formatu PGN służy także przy obliczaniu rankingów.

Pola występujące tylko w service_games

Pole	Typ	Dodatkowe informacje
`id`	SERIAL	**PRIMARY KEY**
`game_id_in_service`	VARCHAR	NULL
`service_id`	INT	NOT NULL REFERENCES game_services(id)
`white_player`	VARCHAR	NOT NULL
`black_player`	VARCHAR	NOT NULL
`is_ranked`	VARCHAR	NOT NULL

`game_id_in_service` to ID pochodzące z zewnętrznego API. Na pary `(game_id_in_service, service_id)` jest założone ograniczenie UNIQUE.

Dla partii pochodzących z naszego serwisu, `game_id_in_service` jest NULL, ponieważ `id` identyfikuje je jednoznacznie. Tabela ma ograniczenie upewniające się, że dla każdej takiej partii `game_id_in_service IS NULL`, a dla wszystkich partii pochodzących z zewnętrznych serwisów `game_id_in_service IS NOT NULL`.

Dla zewnętrznych serwisów `white_player` i `black_player` oznaczają id użytkownika w API tego serwisu. Pary `(white_player, service_id)` i `(black_player, service_id)` są kluczami obcymi wskazującymi na pary `(service_id, user_id_in_service)`, czyli klucz podstawowy, w tabeli `service_accounts`.

Pole `is_ranked` (fałszywe dla gier spoza naszego serwisu) opisuje, czy dana rozgrywka powinna być liczona do rankingów.

Pola występujące tylko w pgn_games

Pole	Typ	Dodatkowe informacje
`id`	SERIAL	**PRIMARY KEY**
`owner_id`	INT	NOT NULL REFERENCES users(id) ON DELETE CASCADE
`white_player_name`	VARCHAR	NOT NULL
`black_player_name`	VARCHAR	NOT NULL

`owner_id` to ID użytkownika, który zaimportował daną partię. Ponieważ w przypadku `pgn_games` partie widzi tylko właściciel, pole to ma ustawione `ON DELETE CASCADE`, aby po jego usunięciu partia także została usunięta.

rankings

Pole	Typ	Dodatkowe informacje
`id`	SERIAL	**PRIMARY KEY**
`name`	VARCHAR	NOT NULL
`playtime_min`	INTERVAL	NOT NULL
`playtime_max`	INTERVAL	NULL
`extra_move_multiplier`	INT	NOT NULL
`starting_elo`	NUMERIC	NOT NULL
`include_bots`	BOOLEAN	NOT NULL
`k_factor`	NUMERIC	NOT NULL

Wartości `playtime_min`, `playtime_max`, `extra_move_multiplier` determinują, czy dana gra może zaliczać się w dany ranking (określają "widełki").

`include_bots` opisuje, czy boty mogą mieć wartości w tym rankingu.

`starting_elo` to wartość początkowa przyporządkowywana zawodnikom w danym rankingu.

`k_factor` to wewnętrzna stała określająca zmiany w danym rankingu.

elo_history

Pole	Typ	Dodatkowe informacje
`id`	SERIAL	**PRIMARY KEY**
`service_id`	VARCHAR	NOT NULL, stale równe 1
`user_id_in_service`	VARCHAR	NOT NULL
`ranking_id`	INT	NOT NULL REFERENCES rankings(id)
`game_id`	INT	NOT NULL REFERENCES service_games(id)
`elo`	NUMERIC	NOT NULL
`previous_entry`	INT	NULL

Tabela ta w naturalny sposób tworzy historię zmian rankingów ELO dla lokalnych kont.

`service_id` wraz z `user_id_in_service` tworzą klucz obcy na tabelę `service_accounts`.
`ranking_id` wskazuje podobnie na ranking, w którym zmiana się dokonała, a `game_id` wskazuje na grę, która była powodem tej zmiany.

`elo` opisuje nową wartość rankingu, a `previous_entry` wskazuje na poprzednią zmianę tego rankingu dla danego gracza.

Podczas dodawania wartości do tej tabeli upewniamy się, że każda gra może wpłynąć na ELO danego gracza tylko raz.

swiss_tournaments

Pole	Typ	Dodatkowe informacje
`tournament_id`	SERIAL	**PRIMARY KEY**
`round_count`	INT	NOT NULL
`starting_position`	VARCHAR	NOT NULL
`is_ranked`	BOOLEAN	NOT NULL
`ranking_id`	INTEGER	NULL REFERENCES rankings(id)
`time_control`	CLOCK_SETTINGS	NOT NULL

`round_count` to informacja o maksymalnej liczbie rund, w jakiej rozgrywany jest dany turniej - wartość charakterystyczna dla turniejów w systemie szwajcarskim.

`starting_position` to wartość startowej pozycji obowiązująca wszystkie rozgrywki zaliczane do danego turnieju, podobnie jak `time_control` zadaje wymaganie o konkretnym tempie rozgrywki.

`ranking_id` wskazuje na jeden z typów rankingu, który będzie używany do obliczania performance rating dla tego turnieju (o tym więcej w widoku `swiss_tournaments_players_points`). Ta wartość musi być niepusta nawet dla turniejów nierankingowych. Wymagamy także, aby `time_control` było zgodne z podanym typem rankingu.

`is_ranked` opisuje, czy dany turniej jest rankingowy. Rankingowe turnieje wymagają, by wszystkie ich gry były rankingowe, i odwrotnie.

tournaments_games

Pole	Typ	Dodatkowe informacje
`tournament_id`	INT	NOT NULL REFERENCES swiss_tournaments(tournament_id)
`game_id`	INT	NOT NULL REREFERENCES service_games(id)
`round`	INT	NOT NULL

Tabela ta łączy turnieje z zarejestrowanymi dla niego partiami. Zaimpelementowane triggery i checki zapewniają, że rozgrywka zgodna jest z wymaganiami zapewnionymi przez turniej oraz że gracze są w turnieju zarejestrowani.

Nałożona na kolumnę `tournament_id` klauzula `ON DELETE CASCADE` zapewnia, że gra zostanie usunięta z tego rejestru, gdy zostanie usunięty jej turniej.

tournaments_players

Pole	Typ	Dodatkowe informacje
`service_id`	INT	NOT NULL DEFAULT 1 REFERENCES game_services(id)
`tournament_id`	INT	NOT NULL RERERENCES swiss_tournaments(tournament_id)
`user_id_in_service`	VARCHAR	NOT NULL

Tabela ta łączy turnieje z zarejestrowanymi graczami. Przy dołączeniu sprawdzana jest zgodność z wymaganiami nałożonymi na dany turniej.

Pole `service_id` stale ustawione na `1` jest w celu ustawienia klucza obcego złożonego z pól `service_id, user_id_in_service` na tabelę `service_accounts`.
To dość hakerskie rozwiązanie i ta wartość jest właściwie zbędna (stosujemy je też w innym miejscu), ale język nie pozwala nam wstawić stałej wartości do klucza.

Dodatkowy trigger zapobiega usuwaniu z tej tabeli zawodników tak długo, jak ich partie są zarejestrowane do danego turnieju.

byes

Pole	Typ	Dodatkowe informacje
`tournament_id`	INT	NOT NULL RERERENCES swiss_tournaments(tournament_id)
`round`	INT	NOT NULL DEFAULT 1 REFERENCES game_services(id)
`user_id_in_service`	VARCHAR	NOT NULL

Istnienie tej tabeli jest koniecznością ze względu na sposób działania parowania systemu szwajcarskiego. Wymagania nałożone na ten system oraz choćby nawet nieparzysta liczba graczy sprawia, że może nie udać się sparować wszystkich graczy w danej rundzie.
Gracz niesparowany otrzymuje w ten sposób darmowy punkt.

tournaments_ranking_reqs

Pole	Typ	Dodatkowe informacje
`tournament_id`	INT	NOT NULL REFERENCES swiss_tournaments(tournament_id)
`ranking_type`	INT	NOT NULL REFERENCES rankings(id) DEFAULT 1
`required_value`	NUMERIC	NOT NULL

Tabela ta opisuje nałożone na dany turniej wymaganie minimum punktów ELO osiągniętych w danym rankingu.

Wpisy z tej tabeli są usuwane po usunięciu odpowiadającego im turnieju.

tournaments_ranked_games_reqs

Pole	Typ	Dodatkowe informacje
`tournament_id`	INT	NOT NULL REFERENCES swiss_tournaments(tournament_id)
`ranking_type`	INT	NOT NULL REFERENCES rankings(id) DEFAULT 1
`game_count`	INT	NOT NULL

Tabela ta opisuje nałożone na dany turniej wymaganie rozegrania pewnej liczby rankingowych gier zaliczonych do danego typu rankingu.

Podobnie jak inne wpisy łączące się z danym turniejem, wpisy z tej tabeli także są usuwane po usunięciu turnieju.

Widoki

games

Pole	Typ	Dodatkowe informacje
`id`	INT	NOT NULL
`kind`	VARCHAR	Jeden z (`'service'`, `'pgn'`)
`starting_position`	VARCHAR(100)	NOT NULL
`moves`	VARCHAR(5)[]	NOT NULL
`partial_fens`	VARCHAR[]	GENERATED ALWAYS AS (generate_fen_array(starting_position, moves)) STORED
`creation_date`	TIMESTAMPTZ	NOT NULL
`result`	GAME_RESULT	NOT NULL
`metadata`	JSOB	NULL
`clock`	CLOCK_SETTINGS	NULL
`result`	GAME_RESULT	NOT NULL
`service_id`	INT	NULL (puste dla kind `'pgn'`)
`white_service_account`	VARCHAR	NULL (puste dla king `'pgn'`)
`black_service_account`	VARCHAR	NULL (puste dla kind `'pgn'`)
`is_ranked`	BOOLEAN	NULL (puste dla kind `'pgn'`)
`pgn_owner_id`	INT	NULL (puste dla kind `'service'`)
`pgn_white_player_name`	VARCHAR	NULL (puste dla kind `'service'`)
`pgn_black_player_name`	VARCHAR	NULL (puste dla kind `'service'`)

Widok `games` jest UNION `service_games` i `pgn_games`. `kind` jest równy `'service'` dla partii pochodzących z `service_games` i `'pgn'` dla partii pochodzących z `'pgn_games'`. `id` nie jest unikatowe dla wszystkich jego elementów, ale para `(id, kind)` już jest.

W szczególności, wartości unikalne dla jednego z tych rodzajów są `NULL` dla wierszy pochodzących z drugiego.

users_games

Pole	Typ	Dodatkowe informacje
`user_id`	INT	NOT NULL, dotyczy `service_accounts`
`game_id`	INT	NOT NULL, dotyczy `service_games` albo `pgn_games`
`kind`	INT	Jeden z (`'service'`, `'pgn'`)
`moves`	VARCHAR(5)[]	NOT NULL
`creation_date`	TIMESTAMPTZ	NOT NULL
`result`	GAME_RESULT	NOT NULL
`metadata`	JSOB	NULL

Widok ten łączy użytkowników z posiadanymi przez nich grami - tutaj także znajduje się pole `kind` oznaczające źródło pochodzenia partii. Gra jest posiadana przez użytkownika, gdy jest ona grą PGN i jest on oznaczony jako jej właściciel, lub jest ona grą serwisową i jego service_account jest jedną z jej stron.
Podobnie jak wcześniej, para (`game_id`, `kind`) jest unikalna, mimo że żadna z tych wartości pojedynczo niekoniecznie musi taka być.

games_openings

Pole	Typ	Dodatkowe informacje
`id`	INT	NOT NULL
`kind`	VARCHAR	Jeden z (`'service'`, `'pgn'`)
`opening_id`	VARCHAR	NULL
`move_no`	VARCHAR	NULL

Widok ten łączy wszystkie gry z ich debiutami przy pomocy funkcji. Tam, gdzie wykrycie debiutu jest możliwe (istnieje jakikolwiek wpis w bazie, który można dopasować), tam wartość `opening_id` wskazuje na odpowiedni wpis.
Dodatkowo kolumna `move_no` trzyma informację o tym, w którym ruchu dany debiut został wykryty.

games_rankings

Pole	Typ	Dodatkowe informacje
`game_id`	INT	NOT NULL, dotyczy `service_games`
`ranking_id`	INT	NOT NULL, dotyczy `rankings`

Widok `games_rankings` łączy grę z wszystkimi rankingami, na które wpływa. Robi to, sprawdzając,
czy ustawienia zegara gry są zgodne z ustawieniami rankingu, oraz, jeśli jedną ze stron gry jest bot,
czy ranking pozwala na boty.

current_ranking

Pole	Typ	Dodatkowe informacje
`service_id`	INT	NOT NULL, dotyczy `service_accounts`
`user_id_in_service`	INT	NOT NULL, dotyczy `service_accounts`
`ranking_id`	INT	NOT NULL, dotyczy `rankings`
`elo`	NUMERIC	NOT NULL
`elo_history_id`	INT	NOT NULL, dotyczy `elo_history`

Widok `current_ranking` pozwala zobaczyć na obecne stany rankingów wszystkich graczy. Dla danego
`service_id`, `user_id_in_service` oraz `ranking_id` znajduje się tam krotka pokazująca obecny stan
rankingu dla danego konta.

Jeśli dane konto nie może brać udziału w danym rankingu (np. ponieważ jest botem i ranking nie
pozwala na udział botów), widok ten nie będzie posiadał dla niego krotki. Jeśli za to może brać
w nim udział, tylko jeszcze nie rozegrał partii, to widok będzie zawierał wartość z `elo` równym
`starting_elo`.

`elo_history_id` wskazuje na odpowiednią wartość w `elo_history`, żeby np. można było poprzez
`previous_entry` zobaczyć poprzednią wartość elo.

tournaments_reqs

Pole	Typ	Dodatkowe informacje
`tournament_id`	INT	NOT NULL
`ranking_type`	INT	NOT NULL
`game_count`	INT	NULL (puste tylko dla wymagań z `tournaments_ranking_reqs`)
`required_value`	NUMERIC	NULL (puste tylko dla wymagań z `tournaments_ranked_games_reqs`)

Widok ten łączy dane z `tournaments_ranking_reqs` oraz `tournaments_ranked_games_reqs` i trzyma wszystkie wymagania do dołączenia do turniejów.

swiss_tournaments_players_points

Pole	Typ	Dodatkowe informacje
`tournament_id`	INT	NOT NULL
`user_id_in_service`	VARCHAR	NOT NULL
`round`	INT	NOT NULL
`points`	NUMERIC	NOT NULL
`performance_rating`	NUMERIC	NOT NULL

Kolumna `tournament_id` wskazuje na turniej, a kolumna `user_id_in_service` na id gracza w `service_accounts`, który był zarejestrowany w `tournaments_players` w tym turnieju.

Widok ten zawiera informację o punktacji i `performance_rating` dla każdego turnieju i zarejestrowanej rundy.
Zapytanie o krotki z danego turnieju i rundy ujawnia liczby punktów oraz rankingi turniejowe użytkowników w nim zarejestrowanych.

`points` to liczba punktów będąca wielokrotnością `0.5`, zliczająca punkty uzyskane przez gracza nie później niż dana runda.

Performance rating to wartość używana do rozstrzygania remisów - opisuje ona estymowany ranking w trakcie turnieju na podstawie rankingów przeciwników.

swiss_tournaments_round_standings

Pole	Typ	Dodatkowe informacje
'place`	INT	NOT NULL
`points`	NUMERIC	NOT NULL
`performance_rating`	NUMERIC	NOT NULL
`user_id_in_service`	VARCHAR	NOT NULL
`tournament_id`	NUMERIC	NOT NULL
`round`	INT	NOT NULL

Widok ten łączy informacje z poprzedniego widoku `swiss_tournaments_players_points` i dodaje pole `place`, które dla danego zawodnika, rundy i turnieju zapamiętuje jego miejsce.

Napotkane problemy

Modelowanie partii szachowych

W trakcie projektowania bazy natrafiliśmy na problem tego, jak modelować partie szachowe. Nasz program przechowuje jednocześnie partie zaimportowane ręcznie przez graczy, które mają jednego właściciela i są widoczne tylko dla niego, jak i partie z serwisów szachowych, które powinny być widoczne dla obu stron. Mamy więc dwa różne typy partii, które mają jednocześnie ze sobą dużo wspólnego, i chcemy móc operować na nich razem, ale mają też różne pola w zależności od typu. Rozważyliśmy wiele różnych sposobów modelowania tych danych w bazie i poniżej wymieniamy część z nich w skrócie, włącznie z wadami każdego podejścia:

1. Jedna tabela `games` z kolumnami obu typów i checkami weryfikującymi, że kolumny jednego typu są ustawione na wartości inne niż `NULL`, a kolumny drugiego typu wypełnione są `NULL`ami.\
   Wady: Każdy wiersz ma dużą liczbę `NULL`i. Duża redundancja: `NOT NULL` w jednej sekcji znaczy, że cała druga sekcja jest `NULL`.
2. Tabela `games` ze wspólnymi kolumnami oraz osobne tabele `service_games` i `pgn_games`. Tabela `games` posiada pola z kluczami obcymi do `service_games.id` i `pgn_games.id` oraz check sprawdzający, czy dokładnie jeden z tych kluczy obcych jest `NOT NULL`.\
   Wady: możliwość powstania sieroty w `service_games` lub `pgn_games` (a więc np. `pgn_game` która ma właściciela, a nie ma faktycznej rozgrywki). Istnienia takiej sieroty nie da się wykryć triggerami blokującymi jej powstanie, ponieważ trigger taki zupełnie uniemożliwiałby stworzenie wiersza w `pgn_games` i `service_games` (ponieważ potrzebowałyby ono istnienia wiersza w `games`, który potrzebuje istnienia wiersza w `pgn_games` albo `service_games`). W takiej sytuacji można zawsze odnosząc się do `pgn_games` albo `service_games` pierwsze robić INNER JOINa z `games` aby upewnić się, że gra istnieje, ale nie jest to najładniejsze rozwiązanie.
3. Tabela `games` ze wspólnymi kolumnami oraz tabele `service_games` i `pgn_games`. Tabele `service_games` i `pgn_games` posiadają pola `game_id` będące kluczami obcymi do `games.id`.\
   Wady: możliwość posiadania partii w `games`, która jest podłączona do 0 partii w `pgn_games` i `service_games`, lub jednocześnie do `pgn_games` i `service_games`. Podobnie jak w pomyśle 2., problemu z możliwością posiadania 0 partii w `pgn_games` i `service_games` nie da się naprawić triggerem (choć możliwość posiadania dwóch już tak).
4. Tabela `games` ze wspólnymi kolumnami oraz tabele `service_games` i `pgn_games` dziedziczące od games. Tabela `games` z zablokowaną możliwością tworzenia wierszy bezpośrednio, pozwalając na wstawianie wierszy tylko do `service_games` i `pgn_games`.\
   Wady: Niestety dziedziczenie w Postgresie nie dziedziczy żadnych checków, włącznie z

kluczami obcymi i głównymi. Oznacza to, że w tabelach `service_games` i `pgn_games` mógłby być wiersz posiadający to samo `id` (choć to dałoby się jeszcze naprawić triggerami). Większym problemem jest jednak, że do takich tabel wcale nie da się odnosić kluczami obcymi, ponieważ klucz obcy zwracający się do `games` nie sprawdza wcale tabel dziedziczących. Daje to wielkie ograniczenia na potencjalne przyszłe rozszerzanie bazy, dlatego nie zdecydowaliśmy się na to rozwiązanie.\
Aby współpracować z systemem dziedziczenia w Postgresie, tabela `games` musi istnieć, choć nie przechowuje żadnych wierszy.

5. Tabela `games` ze wspólnymi kolumnami oraz tabele `service_games` i `pgn_games`. Tabela `games` posiada pola z kluczami obcymi do `service_games.id` i `pgn_games.id` (z checkami podobnymi do 2.). `Service_games.id` i `pgn_games.id` są symetrycznymi obowiązkowymi kluczami obcymi wskazującymi na z powrotem na klucze obce w `games`.\
   Wady: rozwiązanie to duplikuje klucze obce, wskazując w obie strony na raz - jest to redundancja.
   Wymagany jest wyzwalacz do weryfikowania czy te klucze są spójne, czyli że jeśli wiersz *A* z tabeli `pgn_games`/`service_games` wskazuje na wiersz *B* z `games`, to *B* wskazuje z powrotem na *A*.
6. Tabela `games` ze wspólnymi kolumnami oraz tabele `service_games` i `pgn_games`. Dodatkowe pole `game_type` we wszystkich trzech tabelach - `GENERATED ALWAYS AS('pgn')` w `pgn_games`, analogicznie w `service_games`, w games będące równe `'pgn'` albo `'service'`. Para `(id, type)` będąca foreign key z `service_games` i `pgn_games` w `games`.\
   Wady: konieczność stworzenia dodatkowych kolumn `GENERATED` w `service_games` i `pgn_games` (które muszą być `STORED`, ponieważ Postgres nie implementuje w tej chwili kolumn `VIRTUAL`), możliwość stworzenia sierot w `games` (choć sieroty te są mniej problematyczne niż w `pgn_games` oraz `service_games`, bo raczej przy przeglądaniu bazy nie odwołujemy się do `games` bezpośrednio, tylko przez `pgn_games` albo `service_games`).
7. Finalne rozwiązanie: tabele `pgn_games` i `service_games`, bez tabeli `games`. Kolumny, które w innych rozwiązaniach znajdowały się w `games`, w tym rozwiązaniu są przeniesione do `pgn_games` i `service_games`. W razie potrzeby możliwość robienia UNION na tabelach.\
   Wady: część kolumn w `pgn_games` i `service_games` jest identyczna, co wymaga czujności przy modyfikowaniu struktury tabel. Zapytania o wszystkie partie wymagają odwołania się do dwóch tabel zamiast jednej (albo do widoku `games`, który robi to samo), tak jest np. w `games_openings`. `id` nie stanowi samo jednoznacznego identyfikatora wszystkich partii (ale `(id, kind)`, gdzie `kind` oznacza na rodzaj partii `service`/`pgn`, już tak). Nie da się zrobić jednego klucza obcego do obu typów partii.

Upewnienie się, że dla każdego użytkownika istnieje systemowy `service_account`

Z powodu decyzji o traktowaniu partii rozegranych w naszym systemie w taki sam sposób jak tych rozegranych w innych systemach, musimy upewnić się, że każdy użytkownik posiada dokładnie jedno systemowe `service_account` o `service_id = 1` i z jego `user_id`. Tutaj też rozważyliśmy kilka możliwości:

1. Przechowywanie w `users` bezpośredniego foreign key do odpowiadającego konta systemowego.\
   Wady: niepotrzebna duplikacja danych.
2. Dziedziczenie `users` od `service_accounts`.\
   Wady: mimo tego, że wygląda to, jak dobre rozwiązanie, niestety spotykamy te same problemy, co w podejściu 4 z modelowania partii szachowych. Fakt, że inne tabele nie mogłyby zwracać się do kont systemowych poprzez foreign key, zupełnie psułby np. foreign key z `service_games` do `service_accounts`.
3. Finalne rozwiązanie: stworzenie triggerów `add_default_service_to_user`, `prevent_default_service_modification`, `prevent_default_service_deletion` oraz checka `valid_system_account`, które weryfikują poprawność i istnienie kont systemowych.

Wykrywanie debiutów dla partii

Dla każdej partii debiutem jest najpóźniejsza pozycja pasująca do jakiegoś wpisu w tabeli debiutów. Aby definitywanie go wyznaczyć, konieczne jest zasymulowanie pozycji z całej partii.
Wykrywanie debiutu i przechowywanie danych o nich nie jest więc tak trywialne, a potrzebowaliśmy odpowiedniego rozwiązania dla naszych potrzeb.

1. Zapisywanie debiutu dla partii na stałe po umieszczeniu w tabeli na podstawie zapisanych ruchów\
   Wady: Wartości częściowych FEN-ów, które używamy do obliczania debiutów, są i tak stale używane w aplikacji, więc nawet stracilibyśmy w ten sposób na nieprzechowywaniu tych informacji. Dodatkowo, choć relatywnie rzadko, baza debiutów też może być aktualizowana, co spowoduje przedawnienie się danych.
2. Finalne rozwiązanie: Widok wykorzystujący funkcję liczącą debiut dla danej partii.\
   Wady: To rozwiązanie i tak wymagało kaskady dodatkowych funkcji i jest bardziej czasochłonne. Dodatkowo polegamy w tym miejscu na poprawności danych wejściowych, czego nie można zagwarantować, gdy użytkownik po prostu ręcznie wrzuci tam byle co. Zyskujemy jednak na nie przechowywaniu redundantnych informacji o debiutach w bazie.

Usuwanie zawodników i partii turniejowych

1. Usuwanie gier i zawodników dowolne.\
   Wady: Po usunięciu użytkownika pozostają w tabeli nieusunięte partie, powiązane z turniejem, ale nie z żadnym graczem.
2. Zakazanie usuwania zawodników, tylko gier.\
   Wady: Oczywiście nie uwzględniamy możliwości wycofania się z turnieju, nawet nie rozegrawszy żadnej partii.
3. Finalne rozwiązanie: Możliwość usuwania gier dowolna, zawodników tylko tych bez gier.\
   Wady: Brak dowolności usuwania zawodników, nawet jeśli biorą udział w turnieju w np. tylko jednej partii

Ustalanie poziomu izolacji w transakcji:

Chcieliśmy, aby procedura `update_ranking_after_game` wywoływana przez trigger `update_rankings_on_game_insert` działała w isolation level `REPEATABLE READ`, ponieważ pierwsze robi selecta, a potem aktualizuje bazę na podstawie tych informacji, i mogłaby wprowadzić do bazy
błędne wartości, gdyby okazało się, że między selectem a insertem zaszedł drugi insert. Nie mogliśmy jednak znaleźć żadnego sposobu na ustawienie poziomu izolacji wewnątrz procedury ani triggera.

Analogiczny problem spotkaliśmy w procedurze `recalculate_ranking`. Służy ona do ponownego przeliczenia całego rankingu, gdy np. zostanie zmodyfikowany. Chcieliśmy ustawić w niej poziom izolacji `SERIALIZABLE`, ponieważ rekalkulacja rankingu jest delikatną operacją i nie chcieliśmy, żeby coś mogło pójść nie tak, ale spotkaliśmy ponownie ten sam problem.

Mieliśmy jeszcze pomysł, który wyglądał, jakby mógł zadziałać - gdy triggery te zostaną wywołane z niższym poziomem izolacji niż wymagane, wyrzucą błąd, wymuszając wywołanie ich w sposób odpowiedni. Okazuje się jednak, że jOOQ, biblioteka do interakcji między Kotlinem a bazą danych, którą używamy, wcale nie wspiera ustawiania poziomów izolacji. Musieliśmy więc odrzucić ten pomysł.