Tematyka i cele projektu

Zakres planowanej funkcjonalności zmienił się delikatnie od początkowej deklaracji, przesyłamy więc tutaj aktualny plan. Nasza baza jest zaprojektowana tak, aby wspierać w pełni wszystkie funkcjonalności podstawowe oraz rozszerzone.

Funkcjonalność podstawowa

- Rozgrywanie partii w aplikacji z botem
  - Weryfikacja legalności ruchów graczy
  - Integracja z istniejącymi silnikami szachowymi (np. Stockfish)
- Historia rozegranych partii, włączając:
  - Partie zaimportowane w formacie PGN – Portable Game Notation
  - Partie automatycznie pobierane z połączonych kont w innych serwisach szachowych udostępniających API takich jak chess.com lub lichess.org
  - Partie rozegrane w naszym serwisie
- Możliwość eksportowania partii z historii w formacie PGN
- Analiza partii
  - Rozpoznawanie debiutów
  - Wyświetlanie ruchów w notacji algebraicznej

Funkcjonalność rozszerzona

- System kont
  - Logowanie hasłem
  - Łączenie kont w innych portalach szachowych
- Architektura klient-serwer
- Możliwość przeprowadzenia rozgrywek na żywo z innymi graczami
- Dodatkowa funkcjonalność analizowania partii
  - Szacowanie rankingu Elo (wyłącznie z partii rozegranych w naszym serwisie)
  - Sugestie lepszych ruchów przez bota

Schemat bazy

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 openings przechowuje debiuty, które będą rozpoznawane dla partii poprzez `games_openings`. Planujemy oprzeć ją na https://github.com/lichess-org/chess-openings lub podobnym zasobie zbierającym debiuty. Kolumna eco to kod debiutu w Encyklopedii otwarć szachowych, a FEN to format zapisu pozycji na szachownicy. Klasyczny FEN skracamy do 4 pierwszych wartości - zapisujemy informacje o pozycji na szachownicy, możliwości roszady obu stron, kolorze przy ruchu oraz możliwości wykonania *en passant*.

users

Pole	Typ	Dodatkowe informacje
`id`	SERIAL	**PRIMARY KEY**
`email`	VARCHAR	UNIQUE NOT NULL
`password_hash`	VARCHAR	NOT NULL
`elo`	NUMERIC	NOT NULL DEFAULT 1500

Tabela users przechowuje informacje o użytkownikach w systemie kont naszego projektu. Jeżeli nie udałoby nam się zaimplementować systemu kont (jest to funkcjonalność rozszerzona) to działalibyśmy cały czas na jednym użytkowniku domyślnym.

Wstępnie w kolumnie `password_hash` planujemy przechowywać hash w formacie PHC, używając algorytmu hashowania argon2. Tabela ta ma też oczywiście ograniczenie na poprawność adresu e-mail (pochodzące ze strony emailregex.com).

Najciekawszym elementem tej tabeli jest kolumna `elo`. Jest to redundancja, ponieważ elo może być całkowicie wyliczone z rozgrywek gracza przechowywanych w tabeli `service_games`. Obliczanie elo jest jednak bardzo czasochłonne i zdecydowaliśmy, że przeliczanie go za każdym razem, gdy chcemy je odczytać, byłoby zbyt kosztowne, a liczenie go za pomocą np. materialized view w SQLu byłoby bardzo skomplikowane do zaimplementowania. Planujemy więc po każdej rozgrywce w naszej aplikacji przeliczać elo i zapisywać wynik w bazie.

game_services

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

Tabela `game_services` przechowuje identyfikator dla każdego serwisu szachowego, z którym planujemy integrację. Dodatkowo, w tabeli, pod id `1` znajduje się serwis o nazwie `Random Chess`. Jest to serwis odpowiadający partiom rozegranym w naszym serwisie. Traktujemy je tak samo, jak partie rozegrane w zewnętrznych serwisach, a więc będziemy je przechowywać w tej samej 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
`display_name`	VARCHAR(256)	NOT NULL
`is_bot`	BOOL	NOT NULL

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	NOT NULL
`date`	TIMESTAMP	
`metadata`	JSONB	

Kolumna `moves` przechowuje ruchy graczy w partii w postaci PGN, bez metadanych.

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.

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

`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`.

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.

Widoki

games

Pole	Typ	Dodatkowe informacje
`id`	INT	NOT NULL
`kind`	VARCHAR	Jeden z (`'service'`, `'pgn'`)
`moves`	VARCHAR	NOT NULL
`date`	TIMESTAMP	
`metadata`	JSONB	

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.

users_games

Pole	Type	Dodatkowe informacje
`user_id`	INT	NOT NULL
`game_id`	INT	NOT NULL
`kind`	VARCHAR	Jeden z (`'service'`, `'pgn'`)
`moves`	VARCHAR	NOT NULL
`date`	TIMESTAMP	
`metadata`	JSONB	

Widok `users_games` łączy wszystkie partie z użytkownikami, którzy mają do nich dostęp:

- Partie z `service_games` z użytkownikami, których jakieś podłączone konto z `service_accounts` jest jedną ze stron tej partii.
  Te wiersze mają pole `kind` ustawione na `service`.
- Partie z `pgn_games` z ich właścicielami według pola `owner_id`.
  Te wiersze mają pole `kind` ustawione na `pgn`.

Pola `moves`, `date` i `metadata` są w tym samym formacie co w widoku `games`.

games_openings

Pole	Typ	Dodatkowe informacje
`game_id`	INT	NOT NULL
`game_kind`	VARCHAR	Jeden z (`'service'`, `'pgn'`)
`opening_id`	INT	

Widok `games_openings` jest planowanym widokiem łączącym partie w widoku `games` z ich debiutami. Mamy zamiar zaimplementować go, pisząc funkcję, która pierwsze w oparciu na `moves` w tabeli `games` liczy partial_fen wszystkich pozycji, które wystąpiły w danej grze w kolejności. Następnie, trzeba tylko porównać kolejne elementy tej tabeli z kolumną `partial_fen` tabeli `openings`, znajdując ostatnią pozycję, której może zostać przypisany debiut i zapisując go w `opening_id`. Implementacja tego widoku była zbyt skomplikowana na pierwszy etap projektu, dlatego planujemy to zrobić w etapie drugim.

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.