UNIWERSYTET ŚLĄSKI W KATOWICACH

Wydział Nauk Ścisłych i Technicznych

PRACA INŻYNIERSKA

Platforma sieciowa służąca do parowania graczy w amatorskich turniejach szachowych

Mateusz Tarasek
Nr albumu: 317986

Promotor: dr Bartosz Dziewit

Chorzów 2020

Słowa kluczowe: system parowań, kojarzenia, platforma sieciowa, .NET, REST

Oświadczenie autora pracy

Ja niżej podpisany:

Imię i nazwisko: Mateusz Tarasek
Autor pracy dyplomowej pt. Platforma sieciowa służąca do parowania graczy w amatorskich turniejach szachowych
Numer albumu: 317986
Student Wydziału Nauk Ścisłych i Technicznych Uniwersytetu Śląskiego w Katowicach, kierunku studiów Informatyka Stosowana

Oświadczam, że ww. praca dyplomowa

- została przygotowana przeze mnie samodzielnie[1],
- nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (tekst jednolity Dz. U. z 2006 r. Nr 90, poz. 631, z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- nie była podstawą nadania dyplomu uczelni wyższej lub tytułu zawodowego ani mnie, ani innej osobie.

Oświadczam również, że treść pracy dyplomowej zamieszczonej przeze mnie w Archiwum Prac Dyplomowych jest identyczna z treścią zawartą w wydrukowanej wersji pracy.

Jestem świadomy/-a odpowiedzialności karnej za złożenie fałszywego oświadczenia.

Data

Podpis autora pracy

[1] uwzględniając merytoryczny wkład promotora (w ramach prowadzonego seminarium dyplomowego)

Streszczenie

Celem przedstawionej pracy jest stworzenie platformy sieciowej za pomocą wysokopoziomowych narzędzi oraz ogólnie przyjętych praktyk tworzenia oprogramowania. Aplikacja ma umożliwiać kojarzenie graczy w amatorskich turniejach szachowych bez potrzeby instalacji dodatkowego oprogramowania na sprzęcie lokalnym. Pomniejszym celem jest stworzenie autorskich systemów parowań pozwalających na przebieg turnieju zgodnie z założeniami przyjętymi przez Międzynarodową Organizację Szachową [FIDE]. Praca zawiera opisy działania algorytmów wraz ze schematami blokowymi, projekt platformy sieciowej oraz opis napotkanych problemów wraz z ich rozwiązaniem.

Spis treści

1 Wstęp

Przedmiotem niniejszej pracy jest platfroma sieciowa służąca do parowania graczy w amatorskich turniejach szachowych. Stworzona w tym celu aplikacja udostępnia oprogramowanie umożliwiające przeprowadzenie turniejów szachowych bez potrzeby jego instalacji na sprzęcie lokalnym. Platforma jest dedykowana dla amatorów, ponieważ nie uwzględnia oficjalnych rankingów. W pracy inżynierskiej przedstawiono rozwiązanie wykorzystujące styl architektury oprogramowania REST API. Ponadto opisano zasadę działania wybranych systemów parowań graczy i ich autorskie implementacje.

1.1 Przedstawienie problemu i jego sformułowanie

1. *Czy istnieją aplikacje umożliwiające przeprowadzanie turniejów szachowych?*

   Tak, istnieją dedykowane programy, utworzone na silnikach udpostępniających kojarzenia par zawodników w turniejach szachowych. Takie silniki są zaakceptowane przez Międzynarodowa Federację Szachową jako wzorce kojarzenia. Nie brakuje też oprogramowania stworzonego przez firmy trzecie lub osoby prywatne.

2. *Jaka jest dostępność takich aplikacji?*

   Aplikacje dedykowane, używające oficjalnych silników do kojarzeń, w większości przypadków wymagają wykupienia licencji. Często udostępniane są ich wersje próbne, których pozbawiono kluczowych funkcjonalności np. ograniczają maksymalną ilość zawodników, rund lub uniemożliwiają korzystanie z popularnych systemów kojarzeń.

   Instalacja programów należących do firm trzecich lub osób prywatnych może wiązać się z ryzykiem instalacji oprogramowania złośliwego bądź wadliwego.

   W obu przypadkach potrzebna jest instalacja dodatkowych narzędzi i bibliotek takich jak Java czy .Net Framework.

3. *Sformułowanie problemu.*

   Brak darmowego oprogramowania umożliwaiającego kojarzenie zawodników w turniejach szachowych, które jest proste w obsłudze i nie wymaga instalacji dodatkowych komponentów na maszynie lokalnej.

1.2 Cel i zakres pracy

Celem pracy jest stworzenie platformy umożliwiającej kojarzenie zawodników w turniejach szachowych. Aplikacja skierowana jest głównie do amatorów, którym nie jest potrzebny dedykowany program. Z korzyści płynących z niniejszego rozwiązania może korzystać również każda osoba, która nie chce instalować dodatkowego oprogramowania, a co za tym idzie - uważa takie rozwiązanie za prostsze w obsłudze.

Zakres pracy:

- stworzenie platformy sieciowej przy użyciu technologii REST, która umożliwia korzystanie z tego samego zasobu na różnych platformach np. przeglądarka internetowa lub urządzenia mobilne,
- odseparowanie interfejsu użytkownika od operacji na serwerze - użytkownik nie ma wpływu na to co dzieje się po stronie serwera,
- stworzenie platformy sieciowej nie wymagającej instalacji dodatkowego oprogramowania,
- udostępnienie trzech systemów parowań i potrzebnych funkcjonalności do ich wykorzystania,
- sformułowanie wniosków i perspektyw rozwoju projektu w przyszłości.

2 Wprowadzenie teoretyczne

W tym rozdziale omówiono od strony teoretycznej zaimplementowante systemy kojarzeń. Przybliżono wykorzystane narzędzia i motywy ich wyboru. Przedstawiono również ogólną koncepcję rozwiązania problemu i plany autora.

2.1 Ogólna koncepcja rozwiązania problemu

1. Podział problemu:

   Określone cele można osiągnąć poprzez podział platformy sieciowej na części w zależności od funkcjonalności jaką mają pełnić. Stworzono zatem:

   - serwer API - część algorytmiczna, która ma za zadanie udostępnić usługi związane z parowaniem graczy w turniejach szachowych. Serwer ten został napisany przy użyciu stylu architektury REST, dzięki czemu powinien być całkowicie odseparowany od działań klienta;
   - serwer web - udpostępnia interfejs graficzny oraz daje dostęp do interakcji z serwerem API.

2. Problem instalacji dodatkowego oprogramowania.

   Zgodnie z założeniami platforma sieciowa ma nie wymagać instalacji dodatkowego oprogramowania. Zminimalizowano więc niezbędne do korzystania z niej programy do minimum, którym jest przeglądarka internetowa, domyślnie dostępna na każdym systemie operacyjnym. Dokonano tego za pomocą następujących rozwiązań:

   - przeniesienie odpowiedzialności wykonywania obliczeń z urządzenia użytkownika na osobny serwer,
   - użytkownik komunikuje się z serwerem API za pomocą żądań HTTP; zamiast wykonywać program lokalnie - wysyła żądanie o wykonanie skryptu na swerwerze,
   - zwracane dane są wysyłane i odbierane w niezależnym formacie teskstowym JSON.

   Zapytania opierające się na metodach HTTP są takie same bez względu na urządenie z jakiego zostały wysłane. Wynika to z implementacji samego protokołu HTTP. Natomiast styl architektury REST daje gwarancje, że odpowiedzi serwera na takie zapytania również pozostaną jednakowe [1].

2.2 Systemy parowania w turniejach szachowych

Podczas organizacji zawodów sportowych, nie tylko szachowych, ważnym elementem jest dobór odpowiedniego sposobu kojarzenia zawodników. Można posłużyć się tu przykładem europejskich rozgrywek piłkarskich, w których początkową fazą jest rozgrywanie meczy w trybie każdy z każdym, aby później wyłonić zwycięzcę w systemie pucharowym.

Aby rozszerzyć funkcjonalność platformy zaimplementowano trzy najpopularniejszcze systemy rozgrywek.

1. System kołowy [2].

   Znany również jako system „każdy z każdym". Jest uważany za najbardziej obiektywny sposób wyłonienia zwycięzcy, gdyż pologa on na tym, że każdy uczestnik rozgrywek kolejno zmaga się ze wszystkimi przeciwnikami. Najbardziej widocznym minusem tego typu rozgrywek jest brak realnej możliwości przeprowadzenia ich dla dużej liczby graczy. Zależności pomiędzy ilością zawodników a liczbą rund są następujące:

   (a) ilość rund, które zawodnicy muszą rozegrać to:

   $$r = n - 1,$$

   gdzie:

   r - ilość rund,

   n - ilość graczy w turnieju;

   (b) ilość wszystkich pojedynków potrzebnych do rozstrzygnięcia rozgrywek można określić jako:

   $$p = \frac{n * (n - 1)}{2},$$

   gdzie:

   p - ilość wszystkich pojedynków,

   n - ilość graczy w turnieju.

2. System pucharowy [3].

   Inaczej nazywany systemem „knock-out", polega na rywalizacji graczy w bezpośrednich pojedynkach, w których z reguły zwycięzca kwalifikuje się do dalszych rozgrywek, a przegrany odpada z turnieju. Wykorzystuje się go gdy liczba zawodników jest potęgą 2. Jeżeli ten warunek nie zostanie spełniony to resztę graczy uzupełnia się wirtualnie. Skutkiem tego zabiegu część graczy przejdzie do drugiej fazy zawodów bez rozegrania pojedynku. Główną zaletą tego sposobu rozgrywania turniejów jest znacznie mniejsza liczba rozegranych rund koniecznych do wyłonienia zwycięzcy. Natomiast czynnik losowy, mający istotny wpływ na przebieg rozgrywek, powoduje iż ten system jest uważany za najmniej sprawiedliwy. Zależności pomiędzy ilością rozegranych pojedynków, a liczbą zawodników można przedstawić następująco:

   (a) ilość rund rozegranych przez zawodnika, który wygrał turniej wynosi:

   $$r = \log_2 n,$$

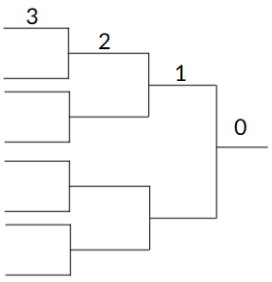
   gdzie:

   r - ilość rund,

   n - ilość graczy w turnieju.

Uzasadnienie:

Rozegrane rundy można przedstawić jako drzewo binarne, gdyż z każdą rundą odpadnie dokładnie połowa zawodników. Zatem ilość rund rozegranych przez zwycięzcę jest równa wysokości tego drzewa binarnego. Przykład dla 8 zawodników:



Rysunek 1: *Drzewo binarne dla* $n = 8$.

(b) ilość wszystkich pojedynków potrzebnych do rozstrzygnięcia rozgrywek można określić jako sumę wszystkich meczy rozegranych w poszczególnych rundach.

Niech:

$$r = \log_2 n,$$

$$p = \sum_{i=1}^{i=r} 2^{i-1}.$$

Po obliczeniu sumy ciągu geometrycznego

$$p = S_r = 1 * \frac{1 - 2^{\log_2 n}}{1 - 2},$$

otrzymujemy

$$p = n - 1,$$

gdzie:

p - ilość wszystkich pojedynków,

r - wysokość drzewa binarnego danej rozgrywki,

n - ilość graczy w turnieju.

3. System szwajcarski [4].

Potrzeba uzyskania kompromisu pomiędzy sprawiedliwym wyłonieniem zwycięzcy, a możliwością udziału dużej ilości osób w rozgrywkach, spowodowała powstanie systemu szwajcarskiego. Jest to najpopularniejszy system parowania zawodników w turniejach szachowych. Dobór par graczy zależy od ich wyników w poprzednich rundach. Dlatego w miarę możliwości gracze z jednakową ilością punktów zostają ze sobą skojarzeni.

Kojarzenie par zawodników w omawianym systemie jest o wiele bardziej skomplikowane, ponieważ system ten musi uwzględnić wiele zmiennych aby uzyskać względnie sprawiedliwy przebiegu turnieju.

Ilość rund rozgrywanych w zawodach jest z góry ustalona. Z prywatnej ankiety przeprowadzonej przez autora wśród graczy amatorskich wynika, że optymalną liczbą rund w zależności od liczby zawodników jest

$$r = \lceil \log_2 n \rceil.$$

Natomiast maksymalna ilość rund może osiągnąć granicę

$$r = n - 1,$$

gdzie:

r - ilość rund,

n - ilość graczy w turnieju.

Należy pamiętać, że przy zwiększeniu liczby rund w stosunku do ilości zawodników, system szwajcarski bardziej będzie przypominał system kołowy.

2.3 Wykorzystane narzędzia i języki programowania

1. Główne narzędzia wykorzystane przy tworzeniu serwera API udostępniającego usługi obliczeniowe:

   - .NET Core - darmowe i otwarte oprogramowanie, które pozwala uruchamiać aplikacje na platformach Windows, Linux i macOS. Technologia ta jest stosunkowo nowa i otwiera nowe możliwości tworzenia serwisów. Całkowitą innowacyjnością jest uniezależnienie narzędzia od systemu w jakim program zostaje skompilowany i uruchomiony.
   - C# - obiektowy język programowania, kompilowany do specjalnego kodu pośredniego Common Intermediate Language (CIL). Dzięki temu możliwe jest jego uruchomienie w środowiskach takich jak .NET Framework lub .NET Core. Aplikacja została napisana w tym języku ze względu na jego popularność i wsparcie techniczne firmy go rozwijającej.

2. Narzędzia bazodanowe:

   - SQLite - system zarządzania bazą danych, który pozwala na korzystanie z bazy danych bez konieczności uruchamiania osobnego procesu RDBMS (ang. Relational Database Management System). Takie rozwiązanie jest w szegółności przydatne przy początkowych fazach rozwoju aplikacji, gdzie jeszcze nie do końca wiadomo o tym jak duże będzie zainteresowanie stworzonym oprogramowaniem. Zawartość bazy danych przy uzyciu tego narzędzia jest przechowywana w jednym pliku aż do 140 TB. Wystarczająco do obsługi małych lub nawet średnich serwisów.

- .NET EF Core (.NET Entity Framework Core) - otwarte narzędzie mapowania obiektowo-relacyjnego. Pozwala na pracę z danymi w formie obiektów i ich właściwości bez potrzeby tworzenia struktury bazy danych oraz pracy z jej tabelami. Umożliwia również pracę na wyższym poziomie abstrakcji co powoduje zauważalne zmniejszenie kodu źródłowego niż w tradycyjnych aplikacjach. Głównym powodem wyboru tego narzędzia było odseparowanie działań na bazie danych od funkcjonowania platformy sieciowej.

3. Narzędzia wykorzystane przy implementacji serwera web:

    - Node JS - środowisko uruchomieniowe JavaScript pozwalające na wykonanie kodu poza przeglądarką internetową. W niniejszej pracy użyto tego narzędzia do stworzenia serwera pośredniego przekazującego polecenia użytkownika do serwera .NET. Jako dopowiedź na żądanie zwraca dynamicznie stworzoną stronę internetową.
    - Express JS - jest to standardowe narzędzie serwerowe dla Node JS. Ułatwia pisanie aplikacji internetowych oparych o API.
    - Pug - silnik szablonów stron internetowych używany do ich tworzenia po stronie serwera.

3 Implementacja systemów parowań

W tej części projektu przybliżono implementacje algorytmów wykorzystanych do obsługi systemów parowań. Sposoby ich działania opisane zostały w krokach oraz schemacie blokowym. Istotnym elementem tej części pracy jest opis rozwiązań problemów powstałych przy implementacji algorytmów. Algorytmy zostały oddzielone od platformy sieciowej i zaimportowane do niej jako moduł.

3.1 Założenia

Podczas implementacji algorytmów systemów parowań wzięto pod uwagę zasady przebiegu turniejów szachowych określone przez Międzynarodową Federację Szachową [5]. Na ich podstawie oraz z uwzględnieniem indywidualnych rozważaniań autora, przyjęto odpowiednie założenia. Kluczowym elementem stał się model gracza, który oprócz ogólnych informacji przechowuje zmienne potrzebne do typowania najlepszego przeciwnika w danej rundzie jak i wyłonienia zwycięzcy.

Ogólne założenia przyjęte przy implementacji algorytmów:

- w turnieju bierze udział n graczy,
- każdy gracz ma przypisany numer od 1 do n,
- jeżeli w turnieju wystąpiła nieparzysta ilość graczy to na koniec puli zawodników dodaje się gracza pauzującego o numerze 0,
- każdy gracz, który został skojarzony z graczem pauzującym automatycznie wygrywa grę nie rozgrywając partii,
- dwóch graczy nie może grać ze sobą więcej niż raz,
- dla każdego gracza różnica między ilością gier kolorem czarnym a ilością gier kolorem białym nie może być większa niż 2 lub mniejsza niż -2,
- żaden gracz nie może otrzymać tego samego koloru bierek 3 lub więcej razy z rzędu,
- każdy z opisanych algorytmów przyjmuje jako parametr listę modeli graczy.

Na podstawie powyższych założeń stworzono model gracza, w którym uwzględniono poniższe zmienne:

- id gracza: int,
- nazwa gracza: string,
- punkty: float,
- buchholz pełny: float,

- ilość rozegranych gier: int,
- ilość gier bierkami białymi: int,
- ilość gier bierkami czarnymi: int,
- listę ID przeciwników z którymi grał zawodnik: int[].

3.2 System Kołowy

Przy implementacji algorytmu systemu kołowego wykorzystano autorskie rozwiązanie stworzone na podstawie modelu o nazwie *patterned tournament* [6] - standardowo używanego przy parowaniu w tego typu turniejach. Zasadniczą różnicą pomiędzy tymi dwoma sposobami rozgrywania turnieju jest to, że w autorskim rozwiązaniu uwzględniono kolory bierek graczy. W opisie algorytmu podano przykład parowania dla $n = 8$ graczy.

Opis algorytmu

Krok 1: Przy pierwszym parowaniu w turnieju, algorytm tworzy dwie listy o długości $\frac{n}{2}$, wypełnia je numerami graczy i zapisuje je w pamięci.

Runda 1:

1	2	3	4
5	6	7	8

Tabela 1: *Ułożenie zawodników w rundzie 1.*

W każdym kolejnym parowaniu lub jeśli wystąpiła nieparzysta liczba zawodników, gracz o numerze 1 zostaje w miejscu a resztę uczestników obraca się według wskazówek zegara.

Runda 2:

1	5	2	3
6	7	8	4

Tabela 2: *Ułożenie zawodników w rundzie 2.*

Krok 2: Wszystkim zawodnikom z pierwszej listy przypisuje się numer przeciwnika z drugiej listy o takim samym indeksie. Powstaje zatem tablica par graczy:

Runda 1:

1	5
2	6
3	7
4	8

Runda 2:

1	6
5	7
2	8
3	4

Tabela 3: *Powstałe pary graczy.*

Krok 3: Każdemu graczowi przypisuje się odpowiedni kolor bierek. Dla partii niepatrzystych gracz w pierwszej kolumnie otrzymuje figury białe, a dla partii parzystych - figury czarne. Wyjątkiem jest gracz o numerze 1, który zawsze otrzymuje kolor przeciwny, do koloru jego bierek w poprzedniej partii. Powstaje tabela parowań z przypisanymi kolorami:

Runda 1:

Partia nr	kolor	Nr gracza		kolor
1		1	5	
2		2	6	
3		3	7	
4		4	8	

Tabela 4: *Przypisanie kolorów bierek dla rundy nr 1.*

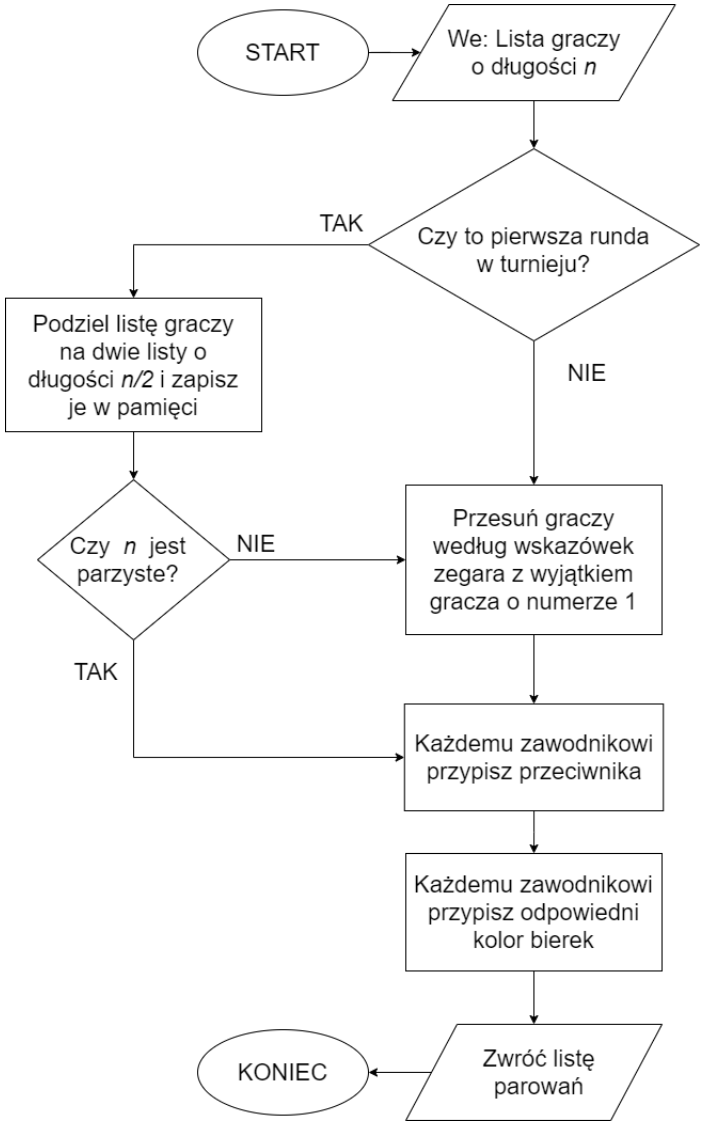
Runda 2:

Partia nr	kolor	Nr gracza		kolor
1		1	6	
2		5	7	
3		2	8	
4		3	4	

Tabela 5: *Przypisanie kolorów bierek dla rundy nr 2.*

Schemat blokowy

Poniżej przedstawiono schemat blokowy algorytmu parującego w systemie kołowym:



Rysunek 2: *Schemat blokowy dla algorytmu kołowego.*

3.3 System Pucharowy

Do implementacji systemu pucharowego użyto autorskiego rozwiązania wzorowanego na powszechnie przyjętych zasadach rozgrywek pucharowych. Założono zatem, że liczba zawodników przystępujących do turnieju powinna być potęgą liczby 2. Jeżeli ilość zawodników nie osiągnie tej liczby to listę uczestników uzupełnia się o zawodników pauzujących. Skutkuje to przejściem niektórych zawodników do następnych faz rozgrywek bez konieczności rozgrywania pierwszych meczy. W opisie algorytmu podano przykład parowania dla $n = 6$ graczy.

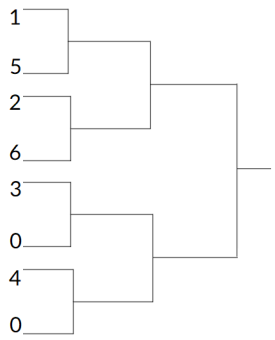
Opis Algorytmu

Krok 1: Przy pierwszym parowaniu w turnieju algorytm dodaje zawodników pauzujących do listy uczestników, tak aby liczba graczy była równa potędze 2:

1	2	3	4	5	6	0	0

Tabela 6: *Lista zawodników po dodaniu graczy pauzujących.*

Krok 2: Zostaje wyznaczony środek listy uczestników, po czym każdemu zawodnikowi z pierwszej połowy przypisuje numer przeciwnika z drugiej połowy. Powstaje zatem drabinka par graczy:



Rysunek 3: *Drabinka turnieju pucharowego dla* $n = 6$.

Drabinka zostaje zapisana w pamięci algorytmu do ponownego wykorzystania w następnych parowaniach.

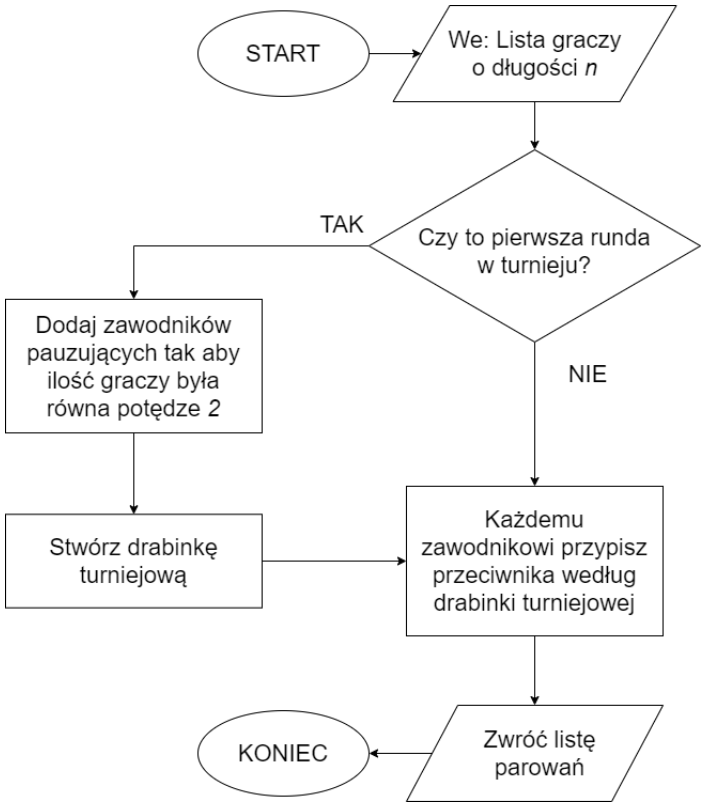
Krok 3: W każdym kolejnym parowaniu zwracane zostają pary przeciwników zdefiniowane według powstałej drabinki.

1	5
2	6
3	0
4	0

Tabela 7: *Lista par przeciwników dla 1 rundy.*

Schemat blokowy

Poniżej przedstawiono schemat blokowy algorytmu parującego w systemie pucharowym:



Rysunek 4: *Schemat blokowy dla algorytmu pucharowego.*

3.4 System Szwajcarski

Do implementacji systemu szwajcarskiego również użyto autorskiego rozwiązania. Stworzony algorytm opiera się na nadaniu każdemu zawodnikowi odpowiednich wag do gry z danym przeciwnikiem. Aby turniej przebiegał jak najbardziej sprawiedliwie, skrypt próbuje kojarzyć uczestników z podobną ilością punktów i preferowanym kolorem bierek. Ze względu na poziom skomplikowania umieszczono również fragmenty kodu źródłowego.

Opis Algorytmu

Krok 1: W pierwszym kroku algorytm sprawdza czy dany zawodnik może grać przeciwko drugiemu.

```
public bool CanPlay(Player other){
    if (this.PlayedWith(other)){
        return false;
    }
    if (other.CanPlayAs(Player.Opossite(this.PrefColor())) &&
        this.CanPlayAs(Player.Opossite(other.PrefColor()))){
            return true;
    }
    return false;
}
```

Jeżeli gracz grał już z danym przeciwnikiem lub nie może grać kolorem preferowanym przez jego przeciwnika to jego ID nie zostaje wpisane do listy wag. Wynikiem czego jest całkowite pominięcie tej pary zawodników w bieżącym kojarzeniu.

Krok 2: Przy każdym parowaniu algorytm wylicza odpowiednie wagi dla każdego gracza. Przyjęto, że im większa waga tym większe jest prawdopodobieństwo skojarzenia z danym zawodnikiem.

- Waga ze względu na ilość punktów.

```
private float GetScoreWeight(
        Player currPlayer, Player other){
    float scoreWeight;
    if(other.Score < currPlayer.Score){
        scoreWeight = other.Score - currPlayer.Score;
    }else{
        scoreWeight = currPlayer.Score - other.Score;
    }
    return scoreWeight;
}
```

Waga zostaje wyliczona poprzez odjęcie większej ilości punktów od mniejszej. To zapewnia, że w przypadku gdy gracze mają taką samą ilość punktów o kojarzeniu zadecyduje ilość gier rozegranych danym kolorem bierek.

- Waga ze względu na ilość gier rozegranych danym kolorem bierek.

```
private float GetColorWeight(
        Player currPlayer, Player other){
    float blackWeight = Math.Abs(
        currPlayer.GamesAsBlack - other.GamesAsBlack);
    float whiteWeight = Math.Abs(
        currPlayer.GamesAsWhite - other.GamesAsWhite);
     return blackWeight + whiteWeight;
}
```

Metoda nadaje większą wagę parom, które rozegrały więcej gier przeciwnym kolorem. Zatem, w skrajnych przypadkach wyniesie ona 0 - gdy obojętnym jest jaki kolor bierek zostanie przypisany danemu graczowi oraz maksymalnie 4 gdy zawodnicy osiągną limit gier danym kolorem.

Krokiem pośrednim jest przemnożenie powstałych wag przez określone zmienne.

```
float scoreMultiplier = 1.5f;
float colorMultiplier = (CurrentRound - 1)/8f;
```

Wraz z postępem turnieju zmiejsza się pula graczy, z którymi mogą grać zawodnicy a co za tym idzie ważność dobrania odpowiedniego koloru bierek przeważa istotność zdobytych punktów. W skrajnym przypadku dla rundy $n-1$ gracz najsilniejszy rozegra partię z graczem nasłabszym. Dzięki temu zabiegowi przebieg turnieju wydaje się bardziej naturalny.

Krok 3: Wybierane są pary zawodników, które mają największą wagę.

```
private List<int> GetBestPairing(
        List<Weight> weightsPerPlayer){
    int highestPriorityID = GetHighestPrority(weightsPerPlayer);
    Weight playerWeights = weightsPerPlayer[highestPriorityID];
    int bestOpponent = GetBestOpponent(playerWeights);
    return new List<int>(){ highestPriorityID, bestOpponent };
}
```

Ważnym jest aby uniknąć zdarzenia, w którym gracze zostaną bez przeciwnika. Dlatego zaimplementowano algorytm decydujący o tym, który zawodnik w pierwszej kolejności musi zostać skojarzony. Następnie wybranemu uczestnikowi przypisuje się rywala z największą wagą.

Krok 4: Dobierany jest kolor bierek zawodników. Odbywa się to według prostej zasady - białe bierki otrzymuje uczestnik z mniejszą ilością gier tym kolorem.

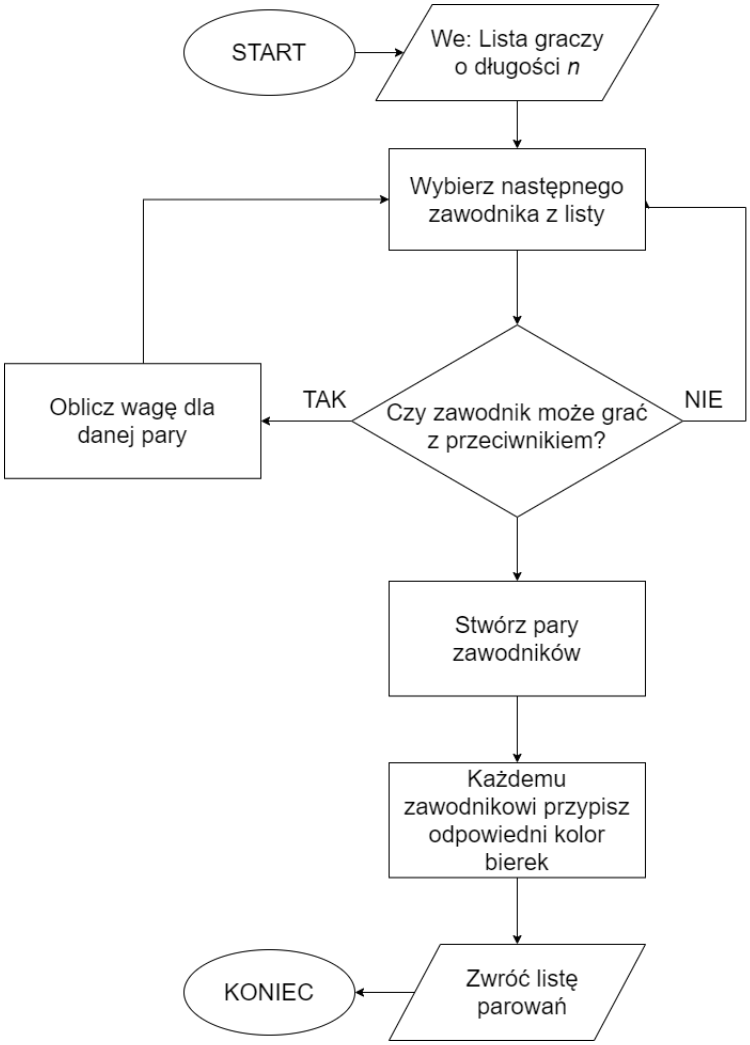
Krok 5: Zwracana jest lista par, w kórych pierwszy zawodnik gra bierkami białymi a drugi czarnymi. Poniżej przedstawiono przykład dla $n = 8$:

1	5
6	2
3	7
8	4

Tabela 8: *Lista par przeciwników dla przykładowej rundy w systemie szwajcarskim.*

Schemat blokowy

Poniżej przedstawiono schemat blokowy algorytmu parującego w systemie szwajcarskim:



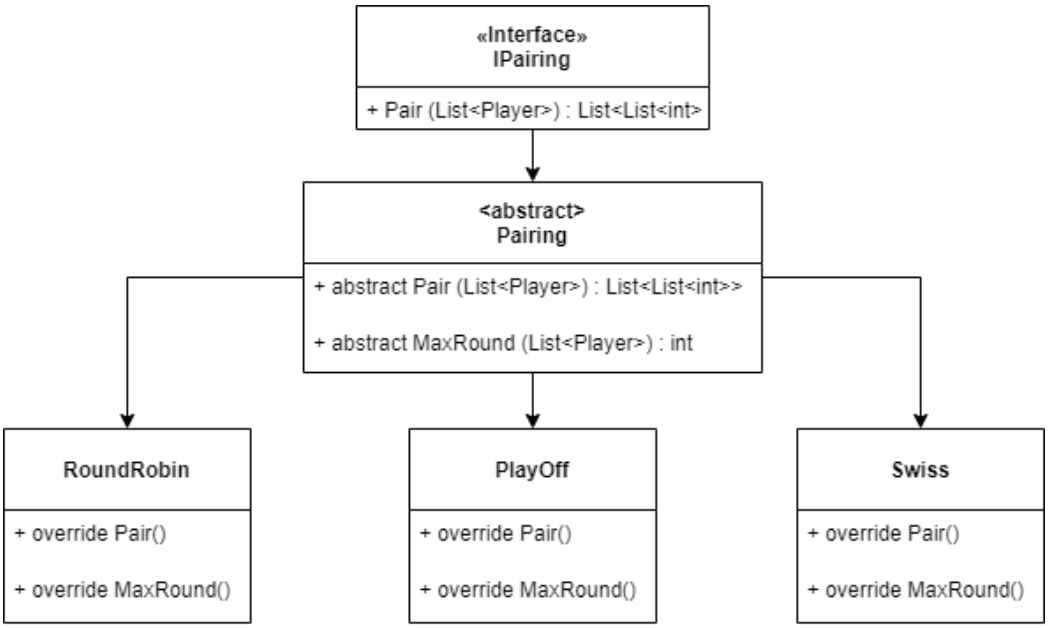
Rysunek 5: *Schemat blokowy dla algorytmu szwajcarskiego.*

3.5 Ogólny zarys schematu klas

Stosując ogólnie przyjęte praktyki programowania, klasy stworzono tak aby:

- każda klasa miała jedną odpowiedzialność - odseparowano od siebie implementacje systemów parowań poprzez stworzenie dla nich osobnych klas,
- były otwarte na rozszerzenia i zamknięte na modyfikacje - wprowadzono klasę abstrakcyjną, która nadaje przymus implementacji metody parowania. Klasy po niej dziedziczące mają jednolity sposób zwracania przetworzonych danych. Ponadto w przypadku gdy zostanie dodany kolejny system kojarzenia nie będzie potrzeby zmiany kodu źródłowego w innych miejscach dopóki implementuje metodę abstrakcyjną,
- zależności pomiędzy nimi wynikały z abstrakcji, a nie implementacji - wszystkie klasy nadrzędne korzystające z systemów parowań nie muszą zwracać się osobno do każdego systemu lecz do interfejsu, które implementują.

Diagram schematu klas:



Rysunek 6: *Diagram schematu klas parowań.*

Przy implementacji algorytmów skorzystano z wzorca Fabryki (ang. Factory Pattern). Dzięki czemu zminimalizowano ilość kodu potrzebną do ich obsługi po stronie serwera API oraz otwarto platformę na rozszerzenia.

```
public class PairingFactory{
    public static Pairing GetPairing(string PairingSystem){
        return PairingSystem switch
        {
            "Kołowy" => new RoundRobin(),
            "Pucharowy" => new PlayOff(),
            "Szwajcarski" => new Swiss(),
            _ => new RoundRobin(),
        };

    }
}
```

Wywołanie metody Pair() w klasie nadrzędnej jest niezależne od ilości zaimplementowanych klas. Wybór parowania odbywa się z zewnątrz i nic nie stoi na przeszkodzie aby dodać kolejny system kojarzenia.

```
 public List<List<int>> pairTournament(string PairingSystem){
    this.PairingSystem = PairingFactory.GetPairing(PairingSystem);
    var pairs = this.PairingSystem.Pair(this.Players);
    return pairs;
}
```

3.6 Napotkane problemy i ich rozwiązania

Wobec przyjętych założeń i oddzielenia implementacji algorytmów od platformy sieciowej, wystąpiły poniższe problemy:

1. Ogólne:
   (a) przypisanie koloru bierek przy założeniu, że algorytm nie może wpływać na zmienne znajdujące się w modelu,
   (b) lista graczy podana przy wywołaniu parowania jest wykorzystywana do inicjalizacji pól. Zapamiętywanie zawodników gdy podane zostały całe ich modele.
2. W systemie kołowym:
   (a) przy nieparzystej ilości graczy została naruszona zasada ilości gier bierkami białymi i czarnymi.
3. W systemie szwajcarskim:
   (a) przy pierwszym podejściu do implementacji tego systemu użyto algorytmu zachłannego - kojarzył graczy po kolei nie zwracając uwagi na to co może wydarzyć się później. Dlatego powstawały sytuacje,

w których turniej nie mógł być kontynuowany, ponieważ dwóch lub więcej graczy nie posiadało przeciwnika.

AD. 1a

Biorąc pod uwagę zasady programowania SOLID, klasa systemu kołowego nie może zajmować się aktualizacją zmiennych modelu gracza. Musi jednak przypisać kolor bierek każdemu z zawodników. Problem ten rozwiązano w taki sposób, że dla każdej pary w liście parowań osoba na pierwszym miejscu ma kolor biały, a osoba na miejscu drugim - czarny.

AD. 1b

Gdy klasa nadrzędna uruchamia algorytm parujący, to przekazuje do niego również listę modeli graczy. Jak wcześniej pokazano, algorytm systemu kołowego dzieli tę listę na dwie mniejsze. Przy naiwnym podejściu można zapamiętywać całe obiekty i na nich wykonywać operacje. Jednak takie działania są mało opłacalne, gdyż koszt operacji jest duży, a ponadto dane zostają powielone. Problem ten został rozwiązany poprzez zapamiętanie samych ID graczy, dzięki czemu zwiększono wydajność algorytmu.

AD. 2a

Ze schematu blokowego wynika, że przy nieparzystej ilości zawodników zostaje pominięte pierwsze parowanie. Bez tego kroku ilość gier bierkami białymi i czarnymi dla niektórych graczy wykazuje dysproporcje.

	Nr gracza						
	1	2	3	4	5	6	7
1		■		■	■		■
2	■		■		■		
3		■		■		■	■
4	■		■		■		■
5		■		■		■	
6	■			■	■		■
Razem	3B/3CZ	3B/3CZ	4B/2CZ	2B/4CZ	2B/4CZ	4B/2CZ	2B/4CZ

Tabela 9: *Tabela kolorów graczy w danych partiach bez pominięcia pierwszej rundy (dla $n = 7$).*

Warto wspomnieć, że gracz o numerze 6 w partii nr 3 nie miał przeciwnika, więc automatycznie wygrał grę nie musząc grać czarnymi bierkami. Nieoficialnie dla niego różnica pomiędzy grami czarnymi figurami a białymi wynosi -3, łamiąc przy tym założenia. Aby rozwiązać ten problem pominięto pierwsze parowanie i dodano kolejne.

	Nr gracza						
	1	2	3	4	5	6	7
2	■		■		■		
3		■		■		■	■
4	■		■		■		■
5		■		■		■	
6	■			■	■		■
7		■	■			■	
Razem	3B/3CZ	3B/3CZ	3B/3CZ	3B/3CZ	3B/3CZ	3B/3CZ	3B/3CZ

Tabela 10: *Tabela kolorów graczy w danych partiach z pominięciema pierwszej rundy i dodaniem ostatniej (dla $n = 7$).*

Jak widać, stosunek gier białych do czarnych się unormował nie łamiąc przy tym przyjętych założeń.

AD. 3a

Problem rozwiązano poprzez zaimplementowanie dodatkowego algorytmu, który wykrywa zawodnika o największym priorytecie kojarzenia.

```
private int GetHighestPrority(List<Weight> weightsPerPlayer){
    int highestProrityID = -1;
    int count = (int)(Math.Pow(2, 15));
    foreach (Weight playerWeight in weightsPerPlayer){
        if (playerWeight.Choices.Count < count){
            count = playerWeight.Choices.Count;
            highestProrityID = playerWeight.ID;
        }
    }
    return highestProrityID;
}
```

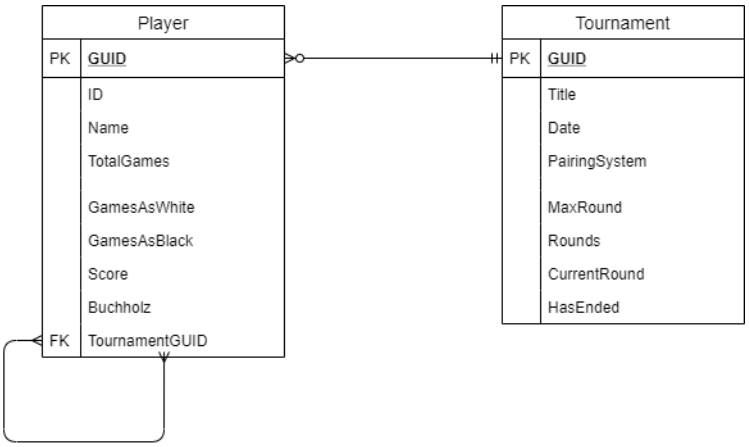
Metoda zwraca ID zawodnika, który posiada najmniejszy wybór przeciwników spośród wszystkch graczy.

4 Projekt bazy danych

Utworzona baza danych posiada minimum pól potrzebnych do przeprowadzenia turnieju szachowego. Zadbano o to aby nie nastąpiła redundancja danych. Zapis informacji o przeciwnikach, z którymi grał zawodnik wymaga odniesienia się tabeli "Player" do samej siebie. Powstałą relację wiele do wielu zastąpiono osobną encją, która umożliwia przechowanie tego typu danych.

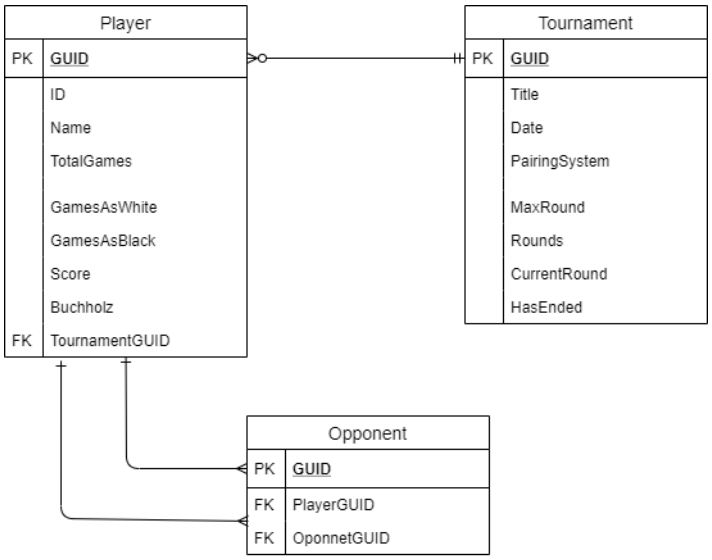
4.1 Schemat ERD

Na rysunku 7 zaprezentowano schemat ERD przed rozwiązaniem relacji wiele do wielu.



Rysunek 7: *Schemat ERD przed rozwiązaniem relacji wiele do wielu*

Rysunek 8 przedstawia schemat ERD po dodaniu dodatkowej encji i rozwiązaniu relacji wiele do wielu.



Rysunek 8: *Schemat ERD po rozwiązaniu relacji wiele do wielu.*

4.2 Proces tworzenia bazy danych

Tworzenie bazy danych za pomocą narzędzia EF Core rozpoczyna się od zdefiniowania silnika, który będzie podstawą do automatycznego generowania i wykonywania skryptów SQL lub NoSQL. Jako, że do komunikacji między bazą danych a działaniami użytkownika użyto narzędzia ASP.NET API to taką definicję umieszczono w metodzie „ConfigureServices" w klasie „Startup".

```
public void ConfigureServices(IServiceCollection services){
    services.AddDbContext<TournamentContext>(opt =>
        opt.UseSqlite("Filename=TournamentPlatform.db"));
}
```

Następnie określa się zbiory danych w kontekście klasy reprezentującej sesję połączenia bazodanowego.

```
public class TournamentContext : DbContext{
    public TournamentContext(
        DbContextOptions<TournamentContext> options)
        : base(options){ }

    public DbSet<TournamentModel> Tournaments { get; set; }
    public DbSet<PlayerModel> Players { get; set; }
    public DbSet<Opponent> Opponents { get; set; }
}
```

Kolejnym krokiem jest zdefiniowanie modelu tabeli. Typ danych zostaje dobrany automatycznie na podstawie typu zmiennych. Relacje między tabelami określa się poprzez oznaczenie dodatkowymi adnotacjami przed deficnicją pól. W przykładzie pokazano model tabeli „Tournament".

```
public class TournamentModel{
    [Key]
    public Guid ID { get; set; }
    [Required]
    public string Title { get; set; }
    [Required]
    public DateTime Date { get; set; }
    [Required]
    public string PairingSystem { get; set; }
    public int MaxRound { get; set; }
    public int Rounds { get; set; }
    public int CurrentRound { get; set; }
    public bool HasEnded { get; set; }
    [ForeignKey("PlayerModel")]
    public List<PlayerModel> Players { get; set; }
}
```

Po zdefiniowaniu wszystkich klas modeli, potrzebnych do stworzenia bazy danych, można wywołać poniższe polecenie w konsoli, znajdując się w folderze głównym projektu. Stworzy ono szkielet migracji inicializującej początkowy zestaw tabel dla modelu.

```
dotnet ef migrations add InitialCreate
```

Aby finalnie utworzyć bazę danych należy wykonać polecenie:

```
dotnet ef database update
```

5 Projekt platformy sieciowej

5.1 Serwer API

Serwer API odpowiedzialny za logikę platformy zaimplementowano przy pomocy narzędzia ASP.NET API. Dla podstawowych operacji bazodanowych stworzono punkty końcowe oparte na protokołach HTTP (podanych w nawiasach kwadratowych) wzorując się na skrócie CRUD (od ang. create, read, update i delete). Zgodnie z przyjętym formatem danych, każde żądanie przyjmuje oraz zwraca pliki tekstowe w formacie JSON.

Punkty końcowe obsługujące operacje na tabeli "Tournaments". Przy ścieżkach pominięto podanie nazwy domeny i portu:

- api/TournamentModels [POST] - metoda umożliwiająca dodanie modelu Turnieju. W przypadku powodzenia zwraca status 201, a w przypadku niepowodzenia status 404,
- api/TournamentModels [GET] - metoda umożliwiająca odczytanie zasobów. W przypadku powodzenia zwraca status 200, a w przypadku niepowodzenia status 404,
- api/TournamentModels/tournamentID [PUT] - metoda umożliwiająca modyfikację już istniejącego zasobu. Przyjmuje parametr tournamentID, dzięki któremu jednoznacznie określa jaką tabelę ma zaktualizować. W przypadku powodzenia zwraca status 200, a w przypadku niepowodzenia status 404,
- api/TournamentModels/tournamentID [DELETE] - metoda umożliwiająca usunięcie istniejącego turnieju. Nie jest możliwym usunięcie całej kolekcji jednym poleceniem. W przypadku powodzenia zwraca status 200, a w przypadku niepowodzenia status 404.

Oprócz standardowych operacji dodano punkty końcowe umożliwiające korzystanie z autorskich systemów parowań:

- api/TournamentModels/tournamentID/hasEnded [GET] - metoda sprawdzająca czy dany turniej został zakończony. W przypadku powodzenia zwraca status 200 wraz ze zmienną logiczną true/false, a w przypadku niepowodzenia status 404,
- api/TournamentModels/tournamentID/highscore [GET] - metoda zwracająca wyniki zawodników w danym turnieju posortowane po ilości punktów oraz punktów Buchholz. W przypadku powodzenia otrzymujemy listę graczy, a w przypadku niepowodzenia status 404,
- api/TournamentModels/tournamentID/pair [GET] - metoda zwracająca parowania dla następnej rundy w turnieju. W przypadku powodzenia otrzymujemy listę par [gracz, gracz], a w przypadku niepowodzenia status 404,

- api/TournamentModels/tournamentID/setScores [POST] - metoda przyjmuje wyniki jakie gracze osiągnęli w danej rundzie. W przypadku powodzenia - dodaje odpowiednią ilość punktów zawodnikom i zwraca status 200, a w przypadku niepowodzenia zwraca status 404.

5.2 Serwer internetowy

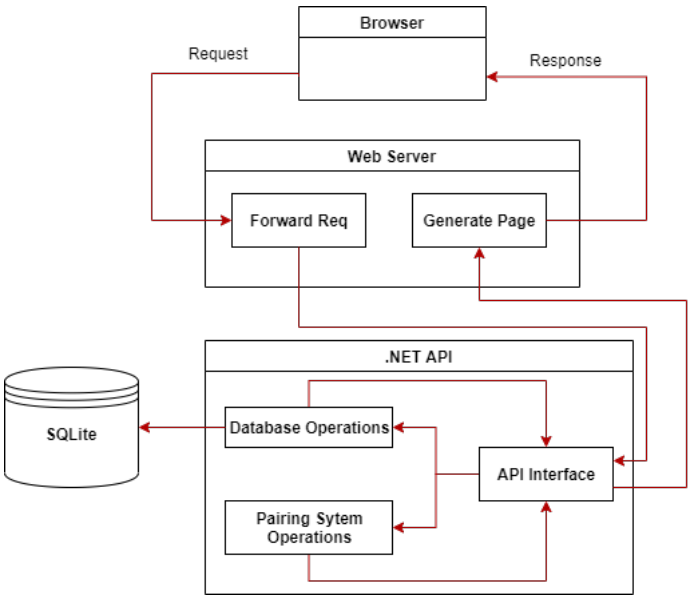
Serwer Web obsługujący żądania użytkowników został napisany przy pomocy narzędzia NodeJS i ExpressJS. Składa się z punktów końcowych, które służą do komunikacji z serwerem API. W odpowiedzi na żądania zwraca dynamicznie wygenerowane strony HTML stworzone na podstawie danych przechowywanych w bazie SQLite.

Poniżej znajduje się schemat działania interfejsu graficznego ze względu na punkty końcowe.

- / - ścieżka do strony domowej. Można przedostać się z niej bezpośrednio do stron odpowiadających za utworzenie i wyszukanie istniejącego turnieju,
- /createTournament - na tej stronie znajduje się formularz, przez który można stworzyć turniej. Umożliwia on wybór jego nazwy, daty oraz systemu parowania. Wysłanie formularza kończy się przekierowaniem na stronę edycji turnieju. W kroku pośrednim użyty zostaje punkt końcowy „/postTournament”,
- /editTournament - pod tą ścieżką znajduje się kilka formularzy umożliwiających pełną edycję turnieju jak i jego zawodników. Pozwala na przejście do parowania następnej rundy oraz wyników. Dostępne formularze oraz punkty końcowe, z których korzystją:
  - edycja pól turnieju - /putTournament,
  - stworzenie gracza i dodanie go do turnieju - /postPlayer,
  - dynamicznie wygenerowane formularze do edycji każdego gracza z osobna - /putPlayer,
  - dynamicznie wygenerowane przyciski usunięcia danego gracza z turnieju - /deletePlayer.
- /pairTournament - strona pokazująca parowanie dla wybranego turnieju oraz umożliwia przesłanie wyników danej rundy. Pośrednio używa „/sendScores”,
- /standings - wyświtla tabelę wyników dla danego turnieju,
- /findTournament - wyświetla wszystkie utworzone turnieje z możliwością ich usunięcia. Ponadto pozwala na wyszukanie rozgrywek po ich nazwie.

5.3 Zasada działania platformy

Użytkownik wysyła żądanie poprzez przeglądarkę internetową. Serwer internetowy je przetwarza i przekazuje dalej do serwera API. Interfejs API decyduje co zrobić z żądaniem po czym odsyła odpowiedź. Silnik szablonów generuje odpowiednią stronę na podstawie otrzymanych danych. Serwer internetowy wysyła wygenerowaną stronę użytkownikowi.



Rysunek 9: *Diagram działania platformy sieciowej.*

5.4 Perspektywy rozwoju

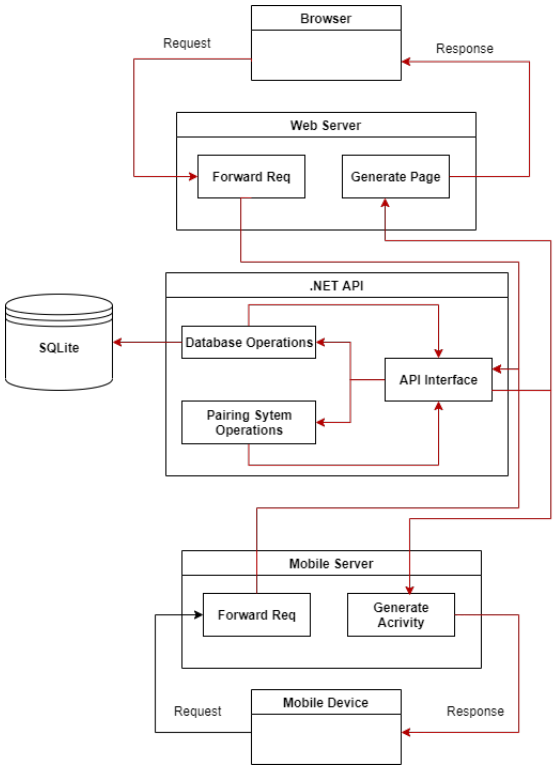
Najbardziej oczywistym, lecz zarazem trafnym pomysłem na rozwój aplikacji sieciowej jest rozszerzenie jej na urządzenia mobilne. Popularność smartfonów i tabletów z roku na rok rośnie coraz bardziej. Zastąpiły one wiele przedmiotów codziennego użytku. Grają coraz większą rolę w życiu człowieka, a jednym z tego powodów jest fakt z jaką łatwością dają dostęp do informacji. Wiele serwisów internetowych posiada aplikacje mobilne będące ich odwzorowaniem, aby były bardziej dostępne dla odbiorców.

Aby rozwój aplikacji sieciowej mógł być rozszerzony na inne platformy trzeba spełnić kilka warunków:

- operacje na bazie danych nie mogą być zależne od serwisu internetowego - zachodzi potrzeba udostępnienia ich wykonywania poza jednym obszarem,
- działanie aplikacji sieciowej musi być niezależne od działania aplikacji mobilnej - pomoże w tym skonstruowanie wspólnego interfejsu żądań, z którego obydwie będą korzystały; interfejs nie może być modyfikowalny przez aplikacje, gdyż zachodziłaby potrzeba wymiany informacji między nimi i ciągłego zapisywania ich stanów,

- aplikacja sieciowa jak i mobilna nie powinny o sobie wiedzieć, a tym bardziej mieć na siebie wpływ,
- zmiany dokonane przez użytkownika muszą być widoczne na obu platformach.

Platforma sieciowa stworzona w ramach niniejszej pracy spełnia postawione warunki. Proces tworzneia aplikacji mobilnej może odbyć się przy pomocy punktów końcowych serwera API. Wykorzystanie tych samych metod protokołu HTTP i formatu danych sprawi, że będzie kompatybilna z już istniejącym serwisem sieciowym. Rozpatrzony został poniższy przykład:



Rysunek 10: *Diagram działania platformy sieciowej po dodaniu kolejnej aplikacji.*

Obydwie aplikacje mogą działać niezależnie od siebie i korzystać z tej samej bazy danych jak i implementacji systemów parowań.

6 Wnioski

Przebieg i efekt pracy nad platformą sieciową prowadzi do wniosku mówiącego, że przeniesienie usług umożliwiających kojarzenie turniejów szachowych ze sfery lokalnej do sieciowej jest całkowicie możliwe.

Pierwszym powodem dlaczego to nie zostało do tej pory wykonane przez firmy komercyjne może być fakt iż Międzynarodowa Federacja Szachowa, mająca duży autorytet, utrzymuje tradycyjne rozwiązania, które mimo wszystko działają.

Drugim powodem mogą być ograniczenia techniczne. Rozbudowane algorytmy kojarzenia zawodników w dużym stopniu obciążają urządzenia. W przypadku rozgrywania wielu turniejów na raz mogłaby zajść sytuacja, że zabrakłoby mocy obliczeniowej serwera aby obsłużyć wszytkie z nich. Potrzebne by było wprowadzenie kolejkowania żądań co prowadziłoby do znacznych opóźnień.

7 Podsumowanie

Celem niniejszej pracy było stworzenie platformy sieciowej służącej do parowania w amatorskich turniejach szachowych w oparciu o styl architektury oprogramowania REST. Przy tworzeniu platformu sieciowej użyto wysokopoziomowych narzędzi takich jak .NET Core i NodeJS.

Wszystkie postawione cele zostały zrealizowane:

- projekt udostępnia rozwiązanie chmurowe kojarzenia par zawodników,
- interfejs graficzny użytkownika został odseparowany od działań na serwerze. Umożlwia to rozszerzenie aplikacji na inne platformy np. urzadzenia moblilne,
- stworzono systemy parowań, które używają autorskich rozwiązań,
- do korzystania z systemów parowań wystarczy zwykła przeglądarka bez potrzeby instalacji dodakowego orogramowania,
- przedstawiono poropozycję dalszego rozwoju.

Spełniając wszystkie założenia platforma sieciowa stała się bardziej przyswajalna dla osób niezwiązanych z IT niż tradycyjne oprogramowania w zakresie przeprowadzania turniejów szachowych.

8 Literatura

[1] R. T. Fielding, R. N. Taylor: *Principled Design of the Modern Web Architecture.*
ACM Transactions on Internet Technology Vol. 2, Issue 2 (2002)

[2] R. V. Rasmussen , M. A. Trick: *Round robin scheduling – a survey.*
European Journal of Operational Research Vol. 188, Issue 3 (2008)

[3] T. Vu, A. Altman, Y. Shoham: *On the Complexity of Schedule Control Problems for Knockout Tournaments.*
Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems Vol. 1 (2009)

[4] L. Csatò: *On the ranking of a Swiss system chess team tournament.*
Annals of Operations Research Vol. 254, Issue 1–2 (2017)

[5] FIDE Handbook, *General Rules and Technical Recommendations for Tournaments*, (2020)
https://handbook.fide.com

[6] J. H. Dinitz: *Designing Schedules for Leagues and Tournaments.*
Mathematics and Statistics, University of Vermont (2005)

[7] W. Stronka: *Optymalizacja kojarzenia w pary uczestników fazy pucharowej rozgrywek sportowych.*
Praca doktorska (2018)

[8] M. Henz, T. Müller, S. Thiel: *Global Constraints for Round Robin Tournament Scheduling.*
European Journal of Operational Research Vol. 153, Issue 1 (2004)