POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: AUTOMATYKA I ROBOTYKA
SPECJALNOŚĆ: PRZEMYSŁ 4.0

PRACA DYPLOMOWA INŻYNIERSKA

Aplikacja webowa do zarządzania ligą sportową

Web application for sport league management

AUTOR:
Jakub Pietrus

PROWADZĄCY PRACĘ:
dr inż. Mariusz Uchroński

OCENA PRACY:

WROCŁAW, 2020

Spis treści

Spis tabel

Rozdział 1

Wstęp

1.1. Wprowadzenie w tematykę

W dzisiejszych czasach każda liga sportowa ma swoją stronę internetową na której udostępnia informacje na temat przeszłych i przyszłych wydarzeń. Z każdego meczu zbierane są za pomocą odpowiedniego oprogramowania informacje, które następnie są przetwarzane i udostępniane fanom sportu w formie różnorakich statystyk i tabel. Najczęściej systemy oferujące takie usługi dostarczają pakiet narzędzi w formie kilku aplikacji, najczęściej desktopowych, dlatego zdecydowano się na napisanie systemu zawierającego wszystkie potrzebne narzędzia do prowadzenia takiej ligi w postaci aplikacji webowej.

1.2. Przegląd istniejących rozwiązań

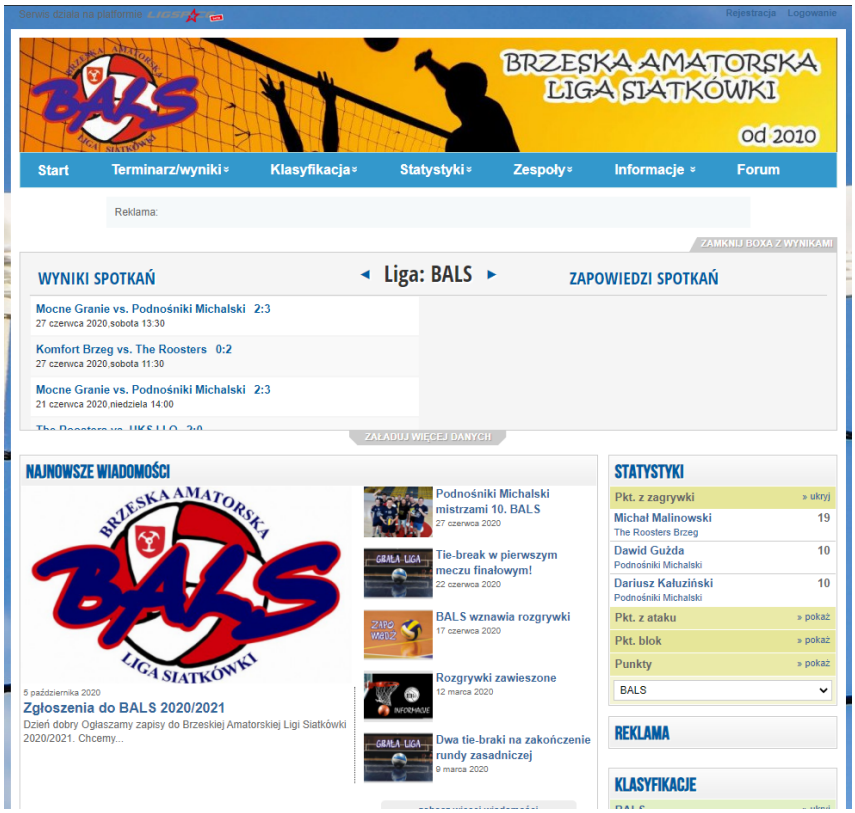
Na rynku istnieją rozwiązania wspierające zarządzanie ligą. Część z nich jest stworzona pod jeden konkretny sport, ale są też oprogramowania, które można użyć do większej ilości dyscyplin. Przykładem aplikacji stworzonej do obsługi kilku rodzajów sportów jest serwis „LigSpace.pl". Krótki opis systemu możemy przeczytać na rysunku 1.1.

Rys. 1.1: Widok zakładki „Dlaczego warto?" w serwisie „LigSpace.pl"

Na podstawie krótkiego opisu systemu na stronie internetowej(Rys. 1.2) można wywnioskować że serwis dostarcza odpowiednie narzędzia do zarządzania ligą tj. program do zbierania statystyk z możliwością relacji live, oraz własna strona internetowa.Niestety oprogramowanie do pisania statystyk oraz zrzuty ekranu nie są nigdzie dostępne, także nie ma możliwości oceny jego funkcjonalności i wygody.

Można jednak znaleźć ligi sportowe, które korzystają z usług tego serwisu - jest nim np. Brzeska amatorska liga koszykówki, której domena znajduje się pod adresem „bals.ligspace.pl”. Na rysunku 1.3 widnieje strona główna brzeskiej ligi, na której widoczny jest panel z terminarzem spotkań, klasyfikacją w różnych kategoriach oraz statystykami. Całość jest jednak bardzo słabo ostylowana i odrzuca swoim wyglądem potencjalnych użytkowników.

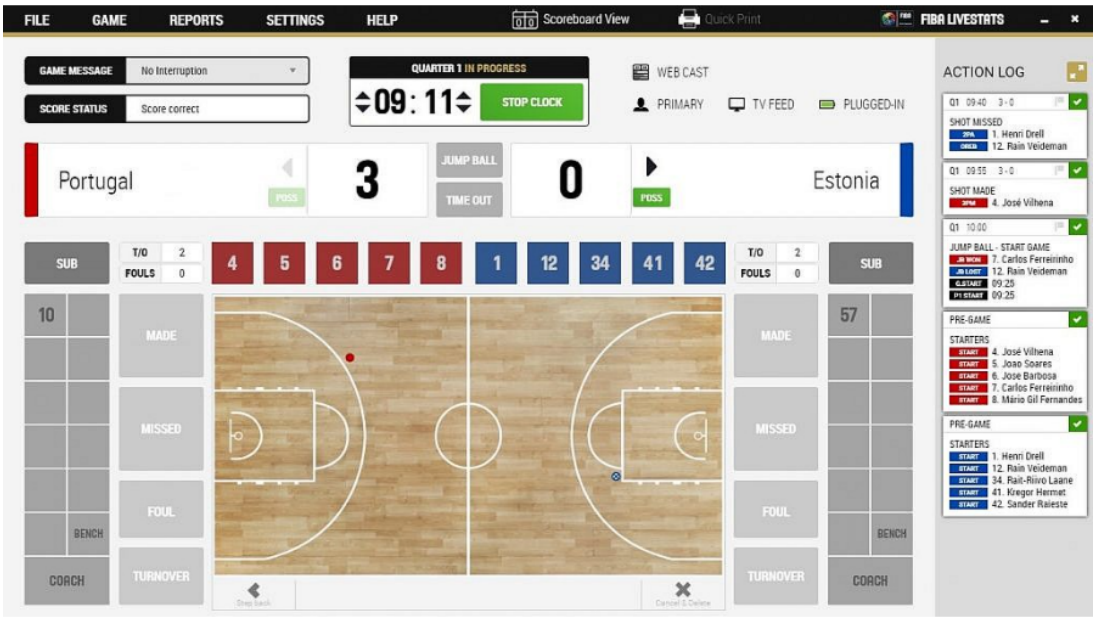
Rys. 1.2: Oferta narzędzi LigSpace



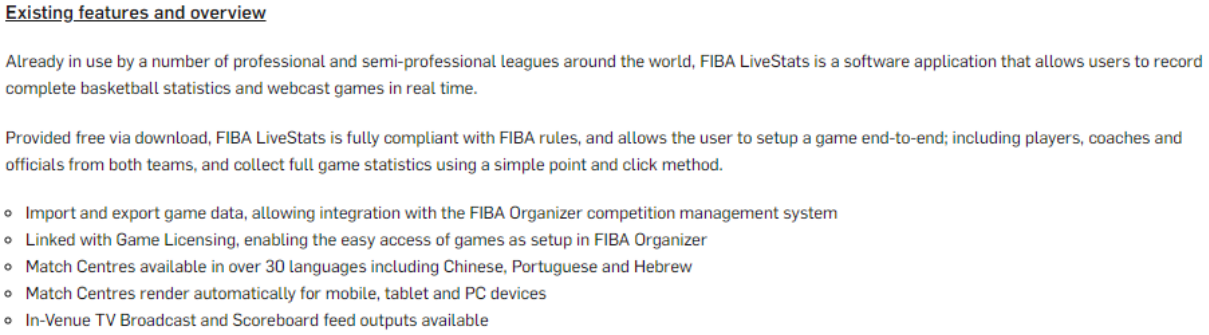
Rys. 1.3: Strona główna „bals.ligspace.pl”

Przykładem pakietu narzędzi stworzonego konkretnie pod jedną dyscyplinę (koszykówkę) jest „FIBA Organizer”, w skład którego wchodzi aplikacja desktopowa do pisania statystyk „FIBA LiveStats”(Rys. 1.4), system do zarządzania bazą zawodników o nazwie „FIBA Organizer CMS”, oraz zestaw stron w technologii WordPress, które następnie można w łatwy sposób

skonfigurować pod własną ligę. O głównych funkcjach programu „Fiba LiveStats" możemy przeczytać na rysunku (Rys. 1.5).



Rys. 1.4: Zrzut ekranu interfejsu programu FIBA LiveStats udostępnionego na stronie „FibaOrganizer.com"



Existing features and overview

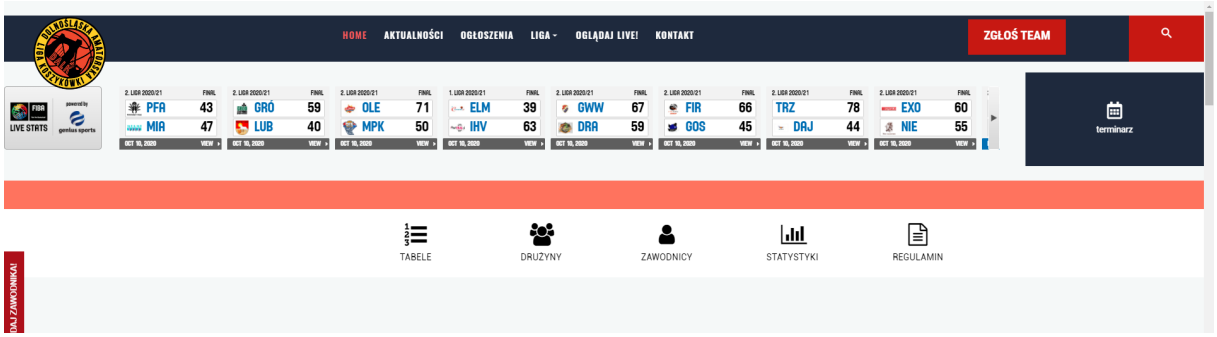
Already in use by a number of professional and semi-professional leagues around the world, FIBA LiveStats is a software application that allows users to record complete basketball statistics and webcast games in real time.

Provided free via download, FIBA LiveStats is fully compliant with FIBA rules, and allows the user to setup a game end-to-end; including players, coaches and officials from both teams, and collect full game statistics using a simple point and click method.

- Import and export game data, allowing integration with the FIBA Organizer competition management system
- Linked with Game Licensing, enabling the easy access of games as setup in FIBA Organizer
- Match Centres available in over 30 languages including Chinese, Portuguese and Hebrew
- Match Centres render automatically for mobile, tablet and PC devices
- In-Venue TV Broadcast and Scoreboard feed outputs available

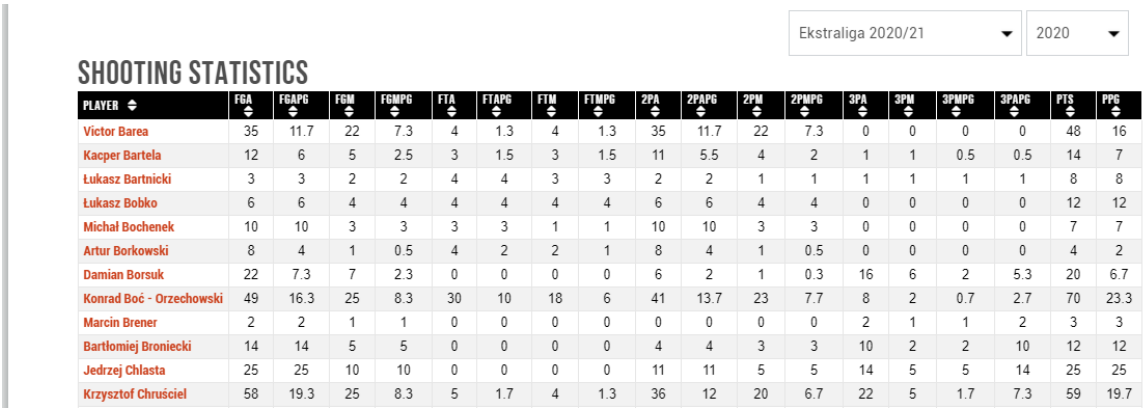
Rys. 1.5: Krótki opis aplikacji FIBA LiveStats na stronie „FibaOrganizer.com"

„Fiba LiveStats" umożliwia szybkie i wygodne zapisywanie statystyk z meczu koszykarskiego przy użyciu klikalnych kafelków wśród których do wyboru mamy zawodników przedstawionych jako kafelki z ich numerem na koszulce, wydarzenia(jak np. rzut, strata, faul), oraz typy zdarzeń które pojawiają się w okienku po wybraniu zdarzenia. W samym centrum znajduje się boisko na którym przed wybraniem zdarzenia należy zaznaczyć jego lokalizację. Po prawej stronie znajduje się pasek w którym wypisane są wszystkie zdarzenia, które zostały dodane. Można je z tego poziomu edytować bądź usunąć. Wszystkie zdarzenia na żywo przesyłane są na stronę internetową.
Przykładem ligi korzystającej z tego oprogramowania jest DALK(Dolnośląska amatorska liga koszykówki). Na rysunku 1.6 widoczna jest górna część strony internetowej „dalk.pl" w której znajduje się poziomy panel z ostatnimi meczami oraz wynikami. Jest to widget z pakietu narzędzi Fiba Orginizer.



Rys. 1.6: Widget poziomego panelu z ostatnimi meczami z pakietu narzędzi Fiba Orginizer

Znaleźć można tutaj tabele wraz ze statystykami, które można filtrować po sezonie oraz dywizji. Statystyki są przejrzyste i wygodnie się je przegląda. Jednak można znaleźć tutaj błąd - po kliknięciu na nazwisko zawodnika strona nie przekieruje nas na jego profil, lecz na stronę główną. Podobne błędy można znaleźć w wielu miejscach, strona internetowa jest niedopracowana.



PLAYER	FGA	FGAPG	FGM	FGMPG	FTA	FTAPG	FTM	FTMPG	2PA	2PAPG	2PM	2PMPG	3PA	3PM	3PMPG	3PAPG	PTS	PPG
Victor Barea	35	11.7	22	7.3	4	1.3	4	1.3	35	11.7	22	7.3	0	0	0	0	48	16
Kacper Bartela	12	6	5	2.5	3	1.5	3	1.5	11	5.5	4	2	1	1	0.5	0.5	14	7
Łukasz Bartnicki	3	3	2	2	4	4	3	3	2	2	1	1	1	1	1	1	8	8
Łukasz Bobko	6	6	4	4	4	4	4	4	6	6	4	4	0	0	0	0	12	12
Michał Bochenek	10	10	3	3	3	3	1	1	10	10	3	3	0	0	0	0	7	7
Artur Borkowski	8	4	1	0.5	4	2	2	1	8	4	1	0.5	0	0	0	0	4	2
Damian Borsuk	22	7.3	7	2.3	0	0	0	0	6	2	1	0.3	16	6	2	5.3	20	6.7
Konrad Boć - Orzechowski	49	16.3	25	8.3	30	10	18	6	41	13.7	23	7.7	8	2	0.7	2.7	70	23.3
Marcin Brener	2	2	1	1	0	0	0	0	0	0	0	0	2	1	1	2	3	3
Bartłomiej Broniecki	14	14	5	5	0	0	0	0	4	4	3	3	10	2	2	10	12	12
Jedrzej Chlasta	25	25	10	10	0	0	0	0	11	11	5	5	14	5	5	14	25	25
Krzysztof Chruściel	58	19.3	25	8.3	5	1.7	4	1.3	36	12	20	6.7	22	5	1.7	7.3	59	19.7

Rys. 1.7: Tabela ze statystykami zawodników z całego sezonu

Rozdział 2

Cel pracy

Celem pracy było stworzenie aplikacji internetowej pozwalającej na zarządzanie ligą sportową. Aplikacja ta ma pozwolić organizatorom na układanie terminarza oraz zbierania statystyk z odbywających się meczów z możliwością udostępniania ich na żywo dla zawodników/użytkowników aplikacji. System ten będzie zawierał narzędzia do zbierania informacji z odbywających się meczów, które później zostaną odpowiednio przetworzone i wyświetlone użytkownikom w formie statystyk z wybranego okresu czasu. Należało również zaimplementować system zarządzania drużynami, zawodnikami,meczami, ligami oraz dywizjami. Do wykonania aplikacji konieczne było zapoznanie się z językiem umożliwiającym napisać API, w tym przypadku padło na język C#, oraz stronę front-endową, do której wykorzystano bibliotekę Reacjt.JS. Wynik meczu jest dostępny dla użytkowników na żywo dzięki zastosowaniu rozszerzenia SignalR. Informacje zbierane z meczów zapisywane są w bazie SQL Server.

Rozdział 3

Założenia projektowe

3.1. Wymagania funkcjonalne

3.1.1. Wymagania od strony administratora

- Administrator ma możliwość zarządzania ligą tj. dodawania, usuwania oraz edycji sezonów, dywizji, zawodników, sędziów oraz drużyn
- Administrator ma dostęp do programu pozwalającego zapisywanie statystyk z trwającego aktualnie meczu którego wynik na żywo będzie udostępniany
- Administrator ma możliwość dodania w programie do statystyk nowego zdarzenia, oraz jego usunięcie
- Administrator ma możliwość zaplanowania meczu na przyszły termin

3.1.2. Wymagania od strony użytkownika

- Użytkownik ma możliwość rejestracji oraz logowania
- Użytkownik ma możliwość przeglądania tabel ze statystykami zawodników oraz drużyn z możliwością filtrowania po dywizji lub sezonie
- Użytkownik ma możliwość przeglądania przeszłych oraz przyszłych meczów wraz ze statystykami
- Użytkownik ma możliwość przeglądania wyniku trwających meczów na żywo bez konieczności odświeżania strony
- Użytkownik ma możliwość przeglądania list wszystkich zawodników oraz drużyn
- Użytkownik ma możliwość przeglądania profili zawodników oraz drużyna wraz z ich statystykami
- Użytkownik ma możliwość dodawania, edycji oraz usuwania zdjęć

3.2. Wymagania niefunkcjonalne

Aplikacja powinna:

- korzystać z dostępnych darmowych rozwiązań
- być wieloplatformowa
- być darmowa

Rozdział 4

Narzędzia i Technologie

Wymienione technologie zostały wybrane ze względu na ich największą znajomość oraz chęć rozwoju w ich kierunku przez autora. Użyto najnowszych dostępnych wersji platform programistycznych, które ciągle są rozwijane co umożliwia w przyszłości ciągły rozwój aplikacji.

4.1. .NET Core 3.1

.NET Core to środowisko wieloplatformowe, którego najnowszą wersję udostępniono stosunkowo niedawno, bo na początku grudnia 2019 roku. Jest ono udostępniane na zasadach otwartego oprogramowania przez firmę Microsoft i cały kod źródłowy można znaleźć na platformie Github. .NET Core może wykorzystane do budowania różnych typów aplikacji tj. mobilne, desktopowe, webowe, chmurowe, IoT, do machine learningu, gier lub mikroserwisów. Framework zawiera podstawowe funkcje potrzebne do napisania aplikacji, jednak można je w każdej chwili rozszerzyć korzystając z pakietu paczek NuGet. Wspierane języki to C#, F# oraz Visual Basic.

Źródło : Opracowano na podstawie [7]

4.2. C#

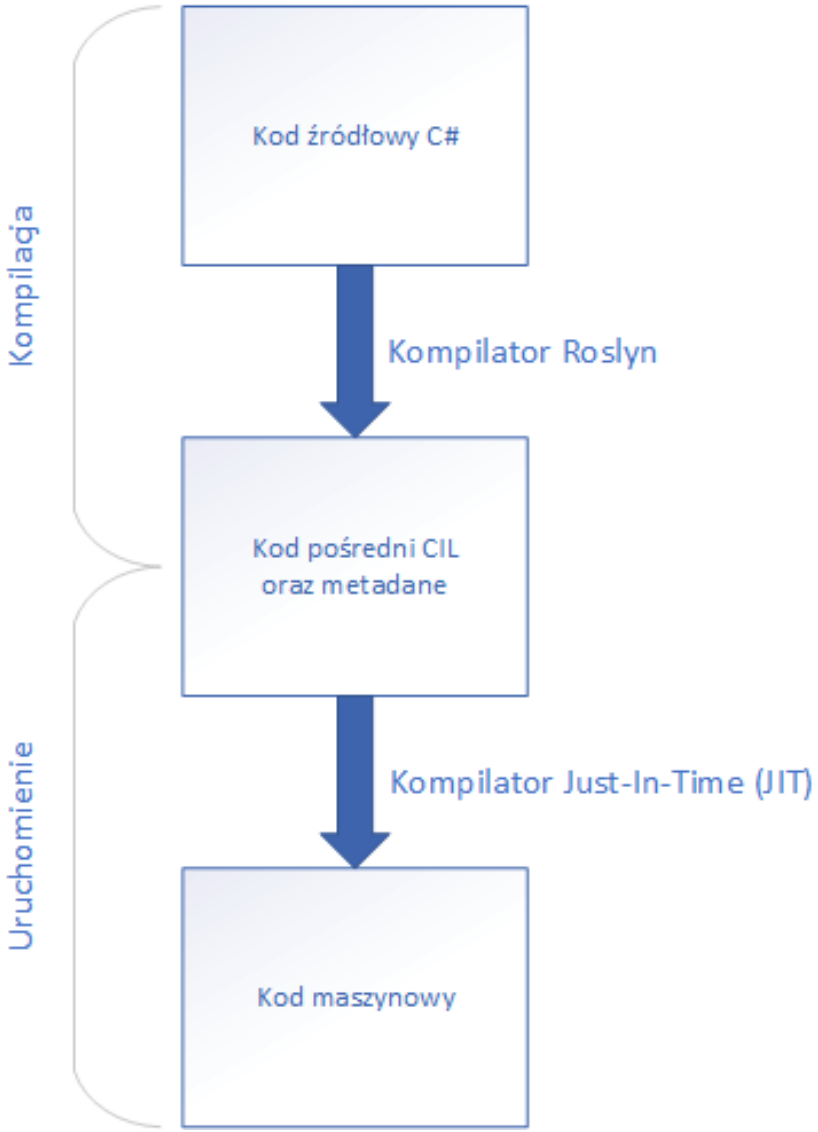
C# jest językiem programowania stworzonym przez firmę Microsoft. Pierwotnie kod źródłowy był chroniony, jednak w 2014 roku został udostępniony, a propozycje nowych funkcji są na bieżąco publikowane na platformie GitHub, pozwalając społeczności na większy wkład w rozwój języka.

Język oferuje możliwość programowania zarówno obiektowego jak i funkcjonalnego, typowania statycznego oraz dynamicznego, obsługę typów generycznych, tworzenia zapytań bazodanowych bez konieczności używania języka SQL, dzięki LINQ(Language Integrated Query). Wspiera także programowanie asynchroniczne.

Jak większość nowoczesnych języków, ten również jest wyposażony w system zarządzania pamięcią tzw. Garbage Collector, który alokuje i zwalnia pamięć za programistę. Jednak w niektórych zastosowaniach jest on niewydajny i wraz z wydaniem wersji 7.2 dodano możliwość zarządzania nią na własną rękę w razie potrzeby.

Kod napisany w C# jest kompilowany do języka CIL(Common Intermediate Language). Następnie trafia do CLR(Common Language Runtime) który jest środowiskiem uruchomieniowym. Przed samym uruchuchomieniem jest kompilowany do kodu maszynowego(co nazywane jest Just In Time Compilation, w skrócie JIT). Patrz rys. 4.1 Taki mechanizm sprawia że aplikacje mogą być uruchamiane na różnych systemach które mają zainstalowane środowisko uruchomie-

niowe.
Źródło : Opracowano na podstawie [5, 6]



Rys. 4.1: Proces kompilacji i uruchamiania aplikacji napisanej w języku C#, [6]

4.3. React.JS

React to javascriptowa biblioteka(choć w sieci nadal trwają spory czy nie jest to framework) służąca do budowania interfejsów użytkownika, stworzona przez firmę Facebook. Wykorzystywana jest ona często do tworzenia aplikacji typu SPA(Single Page Application). Jest to projekt typu open-source i każdy może wziąć udział w jego budowie. Tworzenie takiej aplikacji polega na tworzeniu komponentów, a następnie składanie ich w całość. Dzięki temu można tworzyć komponenty, które można wykorzystać w kilku miejscach co oszczędza czas na pisanie kodu, lub skorzystać z komponentu, który został już przez kogoś wcześniej i odpowiednio dopasować go do aplikacji.
Kiedy strona internetowa stworzona w Reactcie zostaje załadowana przeglądarka tworzy jej model obiektowy DOM(z ang.Document Object Model). Za pomocą języka javascript można korzystając z tego modelu wyszukiwać odpowiednie elementy oraz zmieniać ich właściwości,

co sprawia że strona staje się dynamiczna. Zazwyczaj taka zmiana wiąże się z koniecznością przeładowania całego drzewa HTML, co obniża wydajność działania aplikacji. React rozwiązuje ten problem tworząc wirtualną reprezentację tego drzewa tzw. Virtual DOM. Przy każdej zmianie stanu aplikacji, React nie potrzebuje ładować całego drzewa, tylko porównuje je ze stanem wcześniejszym i zmienia jedynie elementy, których stan się zmienił. Pozwala to na zysk zarówno wydajności po stronie użytkownika jak i developera, którego zmiany w kodzie są bardzo szybko gotowe do podglądu w przeglądarce.

Źródło : Opracowano na podstawie [9, 2]

4.4. TypeScript

Jedną z najbardziej znanych cech języka JavaScript jest jego dynamiczne typowanie, co potrafi przy nieuwadze programisty prowadzić do wielu błędów. Aby przed tymi błędami się zabezpieczyć zastosowano język TypeScript.
Jest on otwarto źródłowym językiem programowania będącym nadzbiorem języka JavaScript, który pozwala na opcjonalne statyczne typowanie, programowanie obiektowe oraz tworzenie interfejsów. Największą zaletą stosowania tego języka jest umożliwienie używanemu IDE(integrated development environment) dostarczenie możliwości na wykrywanie błędów podczas pisania kodu.
Dużą zaletą TypeScriptu jest wczesny dostęp do najnowszych funkcji JSa, które jeszcze nie są wspierane przez wszystkie przeglądarki, ponieważ kompilator przetłumaczy kod na starszą wersję języka, która będzie zrozumiała dla przeglądarki.

Źródło : Opracowano na podstawie [4]

4.5. REST API

REST API jest to uniwersalny interfejs HTTP do komunikacji między oprogramowaniem klienta i serwera za pomocą sieci. REST(Representational State Transfer) oznacza architekturę w której zdefiniowano konkretne metody, zasoby oraz bezstanowość, natomiast API(application programming interface) jest interfejsem. W tego typach interfejsach najczęściej wykorzystuje się metody: GET, PUT, POST, oraz DELETE, które odpowiednio pobierają, aktualizują, tworzą, oraz usuwają zasoby. Dane są najczęściej wysyłane w formacie JSON(JavaScript Object Notation).

Źródło : Opracowano na podstawie [8]

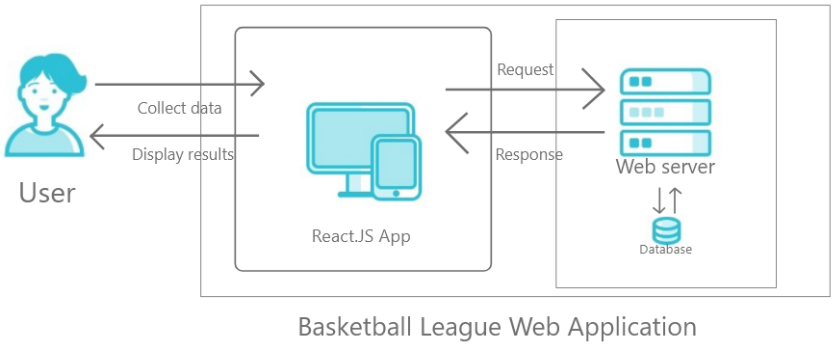
4.6. Microsoft SQL Server

Do stworzenia bazy danych wykorzystano silnik bazodanowy firmy Microsoft o nazwie SQL Server w najnowszej 19 wersji. Wybrano ją z powodu wydajnego działania, niezawodności oraz wygodnej integracji z platformą .NET.

Rozdział 5

Architektura systemu

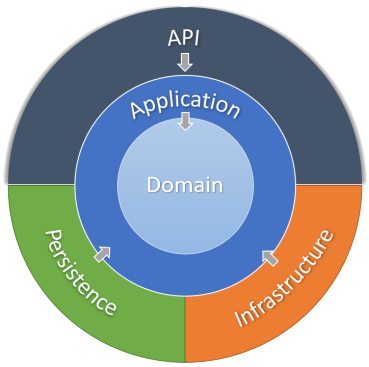
Rozdział ten poświęcono aby omówić sposób implementacji prototypu aplikacji przy pomocy wymienionych rozdział wyżej technologii. Architekturę całej aplikacji ukazano na rys. 5.1.



Rys. 5.1: Schemat architektury systemu do zarządzania ligą sportową. Źródło: Na podstawie [3]

5.1. Back-end

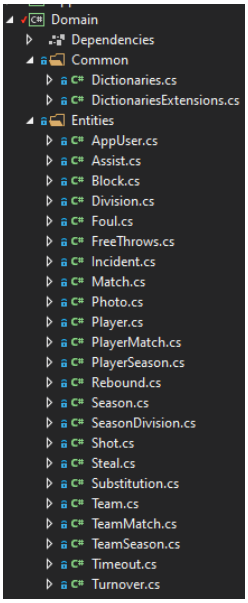
System został podzielony na warstwy tak jak na rys. 5.2. Strzałki symbolizują ich zależność od siebie. Zastosowanie tego wzorca pozwala na niezależność bazy danych, zewnętrznych serwisów oraz warstwy prezentacji(w tym przypadku jest to API do zewnętrznego serwera z warstwą prezentacyjną) i ich szybką oraz wygodną wymianę w razie potrzeby bez ingerowania w wewnętrzne warstwy aplikacji.



Rys. 5.2: Zależności między poszczególnymi warstwami aplikacji

5.1.1. Domain

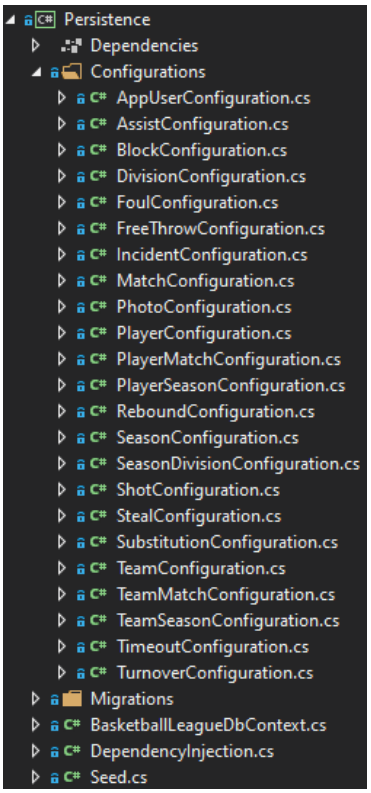
Zawiera wszystkie klasy użyte w bazie danych. Znajdują się tu także enumy oraz logika z nimi związana jak np. metoda do wydobywania opisu z obiektu typu enum.



Rys. 5.3: Pliki i foldery znajdujące się w warstwie domain

5.1.2. Persistence

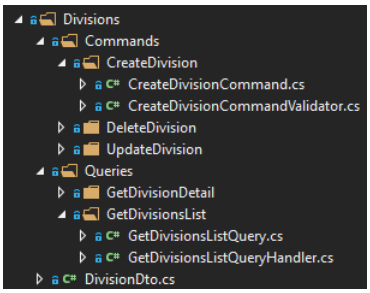
Odpowiada za konfiguracje poszczególnych obiektów bazodanowych oraz za połączenie z bazą danych. Oddzielenie tej warstwy od warstwy domain pozwala na szybką i wygodną zmianę dostarczyciela bazy danych bez potrzeby edytowania samych obiektów. Znajdują się tu także migracje.



Rys. 5.4: Pliki i foldery znajdujące się w warstwie persistence

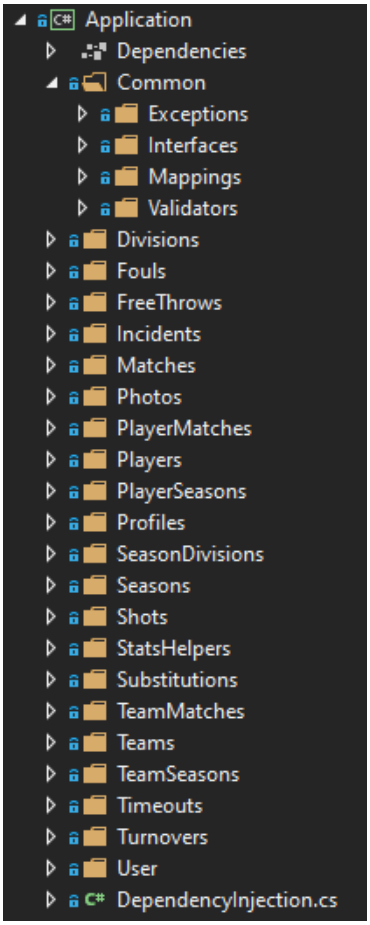
5.1.3. Application

Zawiera całą logikę aplikacji. Jest zależna od warstwy domain i nie ma zależności w innych warstwach lub projektach. Zawiera komendy oraz zapytania (commands/queries) które są obsługiwane przez rozszerzenie MediatR patrz rys. 5.5. Pozwala to na oddzielenie zapytań pobie-



Rys. 5.5: Struktura komend i zapytań dla obiektu division

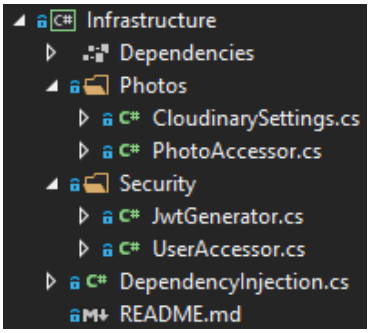
rających dane z bazy od komand które zmieniają stan bazy danych.W tej klasie zdefiniowane są interfejsy implementowane później na zewnątrz tej warstwy. Np. gdy aplikacja potrzebowała dostępu do serwisu wysyłania zdjęć do chmury, nowy interfejs został dodany do aplikacji, a jego implementacja została stworzona w warstwie infrastructure.



Rys. 5.6: Pliki i foldery znajdujące się w warstwie application

5.1.4. Infrastructure

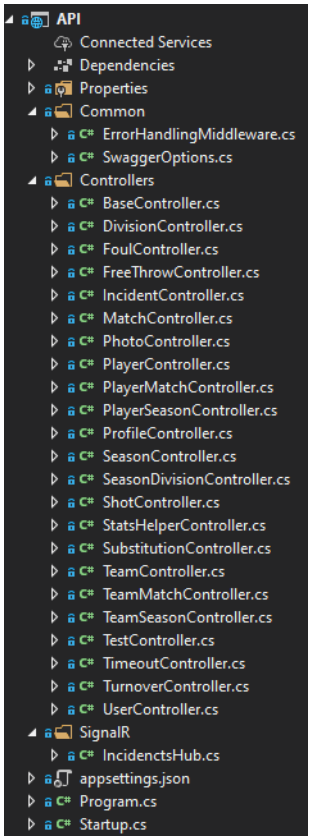
Zawiera klasy zapewniające dostęp do zewnętrznych zasobów takich jak chmura do przechowywania zdjęć, generator tokenów autoryzacyjnych. Klasy te są następnie implementowane przy pomocy dependency injection za pomocą interfejsów stworzonych w warstwie application.



Rys. 5.7: Pliki i foldery znajdujące się w warstwie infrastructure

5.1.5. API

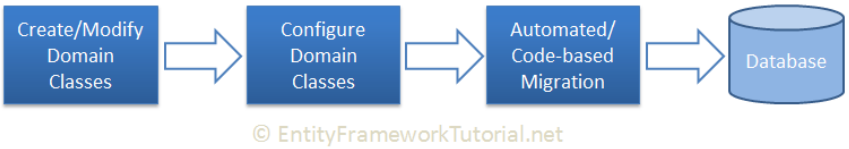
Zawiera controllery które są endpointami przez które wysyłane są zapytania z aplikacji klienckiej. Znajdują się tu również klasy konfigurujące rozszerzenie SignalR służące do wysyłania wyników meczu na żywo, oraz klasa pośrednicząca w przechwytywaniu błędów. W klasie Startup znajduje się konfiguracja obsługi żądań wysyłanych do aplikacji, a także wstrzykiwane są serwisy i interfejsy.



Rys. 5.8: Pliki i foldery znajdujące się w warstwie api

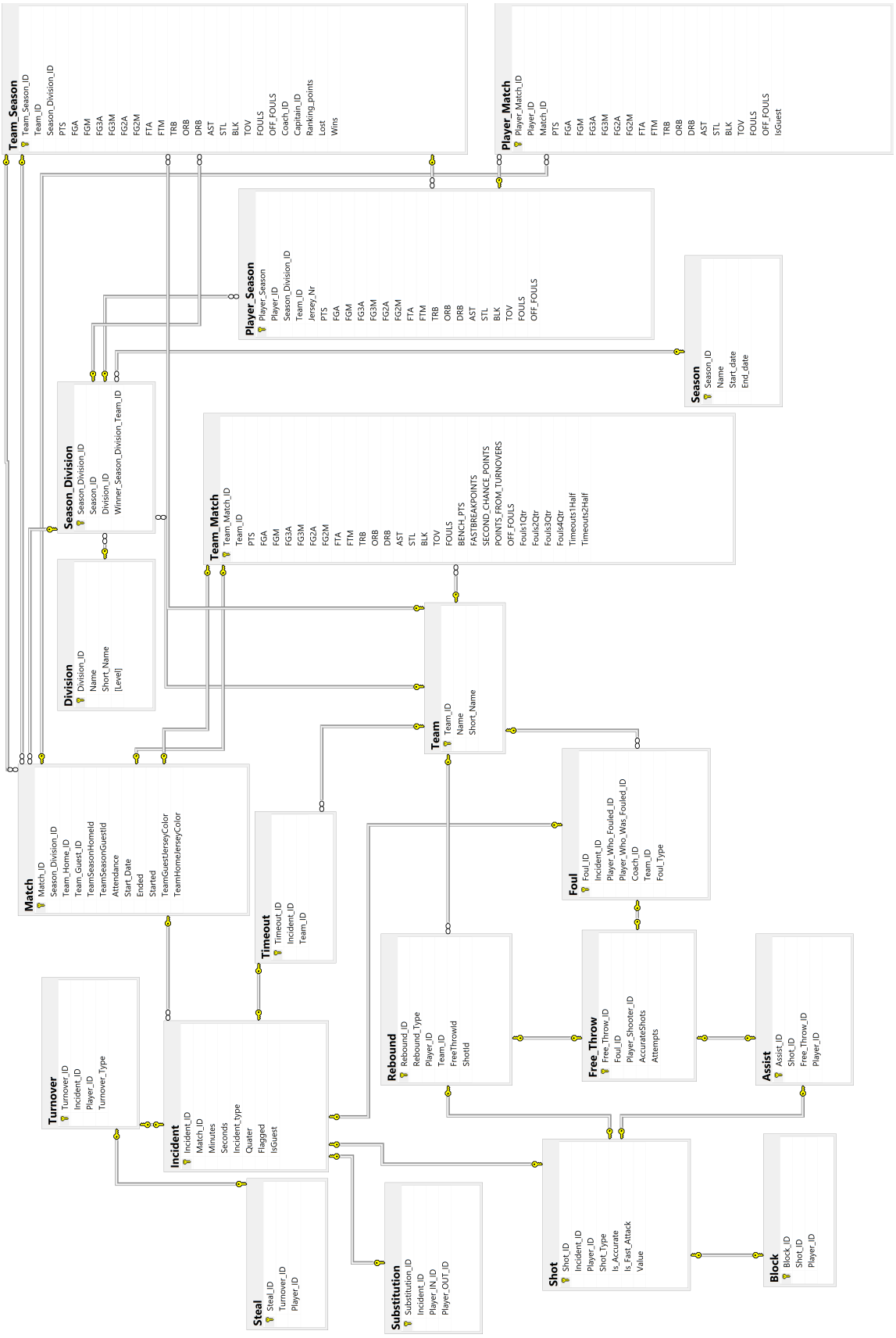
5.2. Baza danych

Bazę danych stworzono stosując podejście Code-First, czyli najpierw zostały utworzone obiekty w kodzie reprezentujące poszczególne tabele, a następnie zostały one zmapowane przy użyciu rozszerzenia Entity Framework Core na SQL-ową bazę danych. Podejście takie pozwala na szybkie i wygodne wprowadzanie zmian dzięki migracjom. Schemat tego podejścia przedstawiono na rys. 5.9.

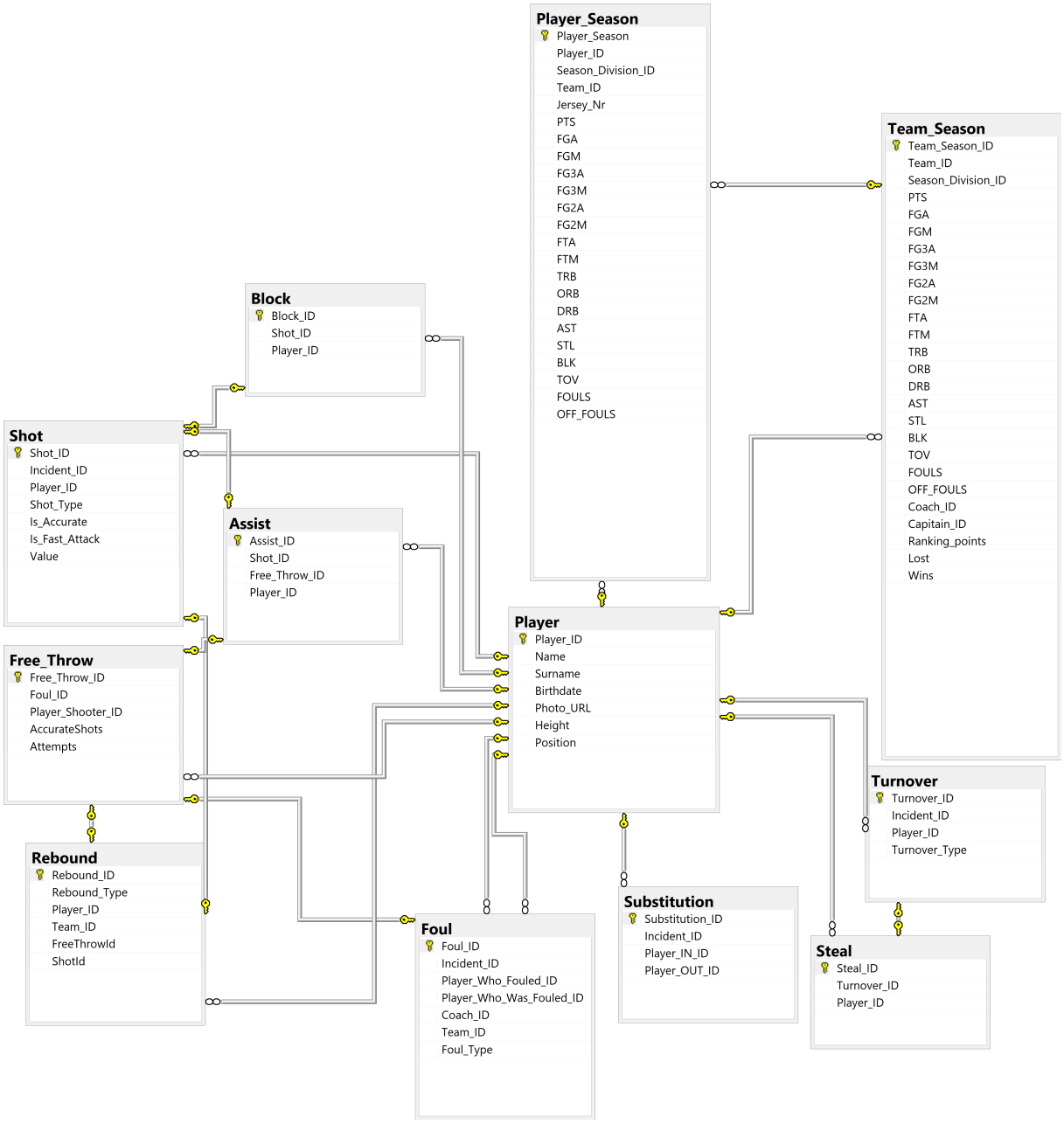


Rys. 5.9: Schemat tworzenia bazy danych w podejściu Code-First, [1]

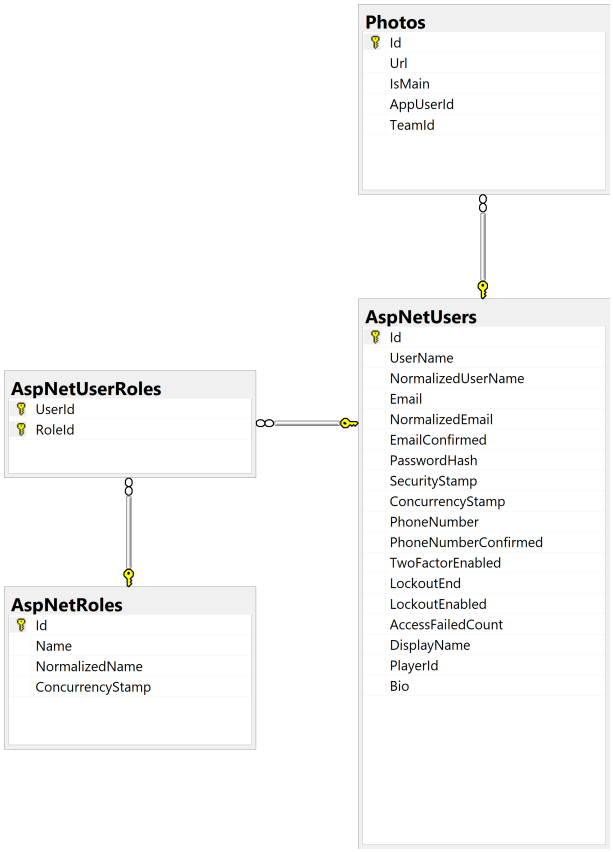
Schemat zaprojektowanej bazy danych na potrzeby aplikacji przedstawiono na rys. 5.10. Jednak z powodu dużej ilości tabel i relacji w nie znalazła się na nim tabela *Player* oraz *AspNetUsers*. Tabelę *Player* wraz z jej relacjami przedstawia rys. 5.11, natomiast tabela *AspNetUsers* znajduje się na rys. 5.12. W bazie przechowywane są informacje o każdym wydarzeniu które zostało dodane przez program do statystyk. Każde wydarzenie jest powiązane z tabelą *Incident* która zawiera informacje o meczu, czasie oraz kwarcie w której zdarzenie miało miejsce i typie incydentu. Tabele odpowiadające za gromadzenie danych o zdarzeniach to: *Shot*, *Substitution*, *Foul*, *FreeThrow*, *Assist*, *Rebound*, *Block*, *Steal*, *Turnover*. Z każdym takim zdarzeniem jest związany zawodnik/zawodnicy lub drużyna/drużyny. W tabeli *Season* gromadzone są informacje na temat stworzonych sezonów, analogicznie tabela *Division* gromadzi informacje na temat stworzonych dywizji. W tabeli *Season_Division* znajdują się dywizje dla danego sezonu. Z tabelą tą związane są tabele *Player_Season* oraz *Team_Season*, które magazynują informacje na temat zawodnika oraz drużyny w danym sezonie oraz dywizji. Tabela *Match* magazynuje informacje na temat meczu, czasie jego startu, końca, drużyn które ze sobą grają (występuje tu relacja zarówno do tabeli *Team_Match* oraz *Team_Season*) oraz koloru koszulek. Występują tu relacje z tabelą *Player_Match* oraz *Team_Match* w których magazynowane są statystyki z powiązanych meczów. W tabeli *AspNetUsers* gromadzone są informacje na temat konta użytkownika. Występuje tu relacja z tabelą *Photos* w której mieści się link do zdjęcia, oraz z tabelą *AspNetUserRoles* w której gromadzona jest informacja o rolach które są przypisane do danego użytkownika. Role natomiast magazynowane są w tabeli *AspNetRoles*.



Rys. 5.10: Schemat bazy danych wraz z relacjami (bez tabeli *Player*)



Rys. 5.11: Tabela *Player* wraz z tabelami relacyjnymi



Rys. 5.12: Tabela *AspNetUsers* wraz z tabelami relacyjnymi

Rozdział 6

Implementacja

6.1. Rejestracja, logowanie oraz role

Do stworzenia użytkowników wykorzystano wbudowany w .NETa interfejs API o nazwie Identity, który pozwala na zarządzanie użytkownikami, hasłami, danymi profilu, rolami oraz tokenami. Po rejestracji informacje na temat użytkownika zachowywane są w bazie, jego hasło zostaje zhaszowane (tym zajmuje się Identity, przez co nie było potrzeby implementacji tego na własną rękę). Przed samym procesem tworzenia użytkownika następuje sprawdzenie, czy użytkownik o podanych danych nie znajduje się już w bazie i w takim przypadku proces przerywany zostaje wyjątkiem.
Jeżeli jest to nowy użytkownik, na podstawie przesłanych danych tworzony jest obiekt *AppUsera* oraz przypisana zostaje mu rola. W przypadku gdy użytkownik zostanie pomyślnie dodany do bazy zwracany jest obiekt *User* wraz z wygenerowanym tokenem pozwalającym na automatyczne zalogowanie się do aplikacji. Kod odpowiadający za rejestrację nowego użytkownika znajduje się na listingu 6.1.

Listing 6.1: Rejestracja nowego użytkownika

```
1  if (await _context.AppUser.AnyAsync(x => x.Email == request.Email,
       ↪ cancellationToken))
2  throw new BadRequestException("Email already exist");
3
4  if (await _context.AppUser.AnyAsync(x => x.UserName == request.
       ↪ UserName, cancellationToken))
5  throw new BadRequestException("Username already exist");
6
7  var user = new AppUser
8  {
9    DisplayName = request.DisplayName,
10   Email = request.Email,
11   UserName = request.UserName,
12 };
13
14 var result = await _userManager.CreateAsync(user, request.Password);
15 await _userManager.AddToRoleAsync(user, "USER");
16
17 if (result.Succeeded)
18 {
19   return new Dto.User
20   {
21     DisplayName = user.DisplayName,
22     Username = user.UserName,
23     Token = _jwtGenerator.CreateToken(user),
```

```
24      };
25    }
26    throw new Exception("Problem creating user");
```

Podczas logowania następuje sprawdzenie czy użytkownik istnieje w bazie i czy hasło jest prawidłowe. Jeżeli tak z bazy zostaje pobierana rola przypisana do użytkownika oraz adres url zdjęcia jeżeli takowe wcześniej zostało wrzucone i generowany jest token. W przypadku niepowodzenia zwracany jest wyjątek braku autoryzacji. Przedstawia to listing 6.2.

Listing 6.2: Logowanie

```
1   var user = await _context.AppUser.Include(x => x.Photos).
        ↪ FirstOrDefaultAsync(x => x.Email == request.Email,
        ↪ cancellationToken);
2
3   if (user == null)
4   throw new NotFoundException(nameof(Dto.User), request.Email);
5
6   var result = await _signInManager.CheckPasswordSignInAsync(user,
        ↪ request.Password, false);
7
8   if (result.Succeeded)
9   {
10    var roles = await _userManager.GetRolesAsync(user);
11
12    return new Dto.User
13    {
14      Username = user.UserName,
15      DisplayName = user.DisplayName,
16      Token = _jwtGenerator.CreateToken(user),
17      Image = user.Photos.FirstOrDefault(x => x.IsMain)?.Url,
18      Role = roles.FirstOrDefault()
19    };
20  }
21  throw new UnauthorizedException(request.Email);
```

Logowanie jest konieczne, aby mieć dostęp do reszty funkcjonalności aplikacji. Podczas każdego requesta do każdego endpointa (za wyjątkiem logowania oraz rejestracji), sprawdzany jest token i w przypadku jego braku, lub nieważności zwracany jest kod 401, świadczący o braku autoryzacji.

Użytkownik może posiadać jedną z dwóch ról - Admin lub User. Podczas rejestracji domyślnie każdemu nowemu użytkownikowi przypisywana jest rola User. Zmiany można dokonać jedynie z poziomu bazy danych. Role te różnią się dostępem do poszczególnych funkcjonalności aplikacji. Użytkownik nie ma dostępu do metod związanych z zarządzaniem ligą oraz meczami, natomiast Admin ma dostęp do wszystkich funkcjonalności systemu. Efekt ten osiągnięto dodając nagłówki z autoryzacją roli przy odpowiednich metodach w kontrollerach. Autoryzację metody rolą przedstawiono na przykładzie tworzenia meczu na listingu 6.3.

Listing 6.3: Autoryzacja rolą metody tworzenia meczu

```
1   [HttpPost]
2   [Authorize(Roles = "Admin")]
3   public async Task<IActionResult> Create([FromBody] CreateMatchCommand
        ↪ command)
4   {
5     await Mediator.Send(command);
6     return NoContent();
7   }
```

6.2. Profil użytkownika

Aplikacja posiada funkcjonalność wgrywania zdjęć, które można następnie wykorzystać jako avatar. Zdjęcie zostaje przypisane do użytkownika, który je uploaduje, oraz zostaje wybrane jako zdjęcie główne jeżeli jest to pierwsze zdjęcie wrzucone przez użytkownika. Kod odpowiadający za wgrywanie zdjęcia znajduje się na listingu 6.4.

Listing 6.4: Dodawanie nowego zdjęcia

```
var photoUploadResult = _photoAccessor.AddPhoto(request.File);

var user = await _context.AppUser.Include(x => x.Photos)
.SingleOrDefaultAsync(x => x.UserName == _userAccessor.
    ↪ GetCurrentUsername(), cancellationToken);

var photo = new Photo
{
  Url = photoUploadResult.Url,
  Id = photoUploadResult.PublicId
};

if (!user.Photos.Any(x => x.IsMain))
  photo.IsMain = true;

user.Photos.Add(photo);

var success = await _context.SaveChangesAsync(cancellationToken) > 0;
if (success) return photo;
throw new Exception("Problem saving changes");
```

Zdjęcie wrzucone zostaje na serwis chmurowy cloudinary, a w bazie przechowywany jest link do wrzuconego zdjęcia.

Zaimplementowano również możliwość podglądu profilu innych użytkowników. Na podstawie przesłanej w requeście nazwy użytkownika z bazy pobierane są informacje na jego temat wraz ze zdjęciem głównym jeżeli takowe posiada. Do znalezienia użytkownika wykorzystano metodę SingleOrDefaultAsync, co oznacza że w przypadku braku użytkownika o podanej nazwie funkcja rzuca wyjątek. Sposób pobierania danych na temat profilu przedstawia listing 6.5.

Listing 6.5: Pobieranie danych na temat profilu

```
var user = await _context.AppUser.Include(x => x.Photos)
.SingleOrDefaultAsync(x => x.UserName == request.Username,
    ↪ cancellationToken);

return new Profile
{
  DisplayName = user.DisplayName,
  Username = user.UserName,
  Image = user.Photos.FirstOrDefault(x => x.IsMain)?.Url,
  Photos = user.Photos,
  Bio = user.Bio
};
```

6.3. Zarządzanie ligą

Jest to implementacja funkcjonalności do których dostęp ma jedynie użytkownik z rolą Admin. Aplikacja pozwala na dodawanie, edycje oraz usuwanie sezonów, dywizji, drużyn oraz zawodników. Do każdego z wymienionych obiektów zaimplementowała została również walidacja

wysłanych danych przy użyciu rozszerzenie Fluent Validation. Sposób implementacji tych podstawowych operacji CRUD przedstawiono na przykładzie obiektu sezonu. Jeżeli dane w zapytaniu przejdą walidację, której implementację przestawia listing 6.6 to następuje przypisanie danych do obiektu *Season*. Jeżeli w zapytaniu znajdują się dane na temat dywizji które mają zostać przypisane do tego sezonu to również są one dodawane, w przypadku ich braku operacja zostaje pominięta i następuje zapisanie sezonu do bazy. Jeżeli operacja przebiegła prawidłowo zwracany jest Id sezonu, w innym przypadku rzucany jest wyjątek, co przedstawia listing 6.7

Listing 6.6: Walidacja danych

```
RuleFor(x => x.Name)
.NotEmpty();
RuleFor(x => x.StartDate)
.NotEmpty();
```

Listing 6.7: Dodawanie nowego sezonu

```
var newSeasonDivisions = new List<SeasonDivision>();
var season = new Season
{
  Name = request.Name,
  StartDate = request.StartDate,
  EndDate = request.EndDate
};

if (request.DivisionsId.Any())
{
  foreach (var divisionId in request.DivisionsId)
  {
    var seasonDivision = new SeasonDivision
    {
      Season = season,
      DivisionId = divisionId,
    };
    newSeasonDivisions.Add(seasonDivision);
  }
  _context.SeasonDivision.AddRange(newSeasonDivisions);
}
else
{
  _context.Season.Add(season);
}

var success = await _context.SaveChangesAsync(cancellationToken) > 0;
if (success) return season.Id;

throw new Exception("Problem saving changes");
```

Podczas edycji pobierany jest sezon za pomocą Id, jeśli nie zostaje on znaleziony rzucany jest wyjątek. Po znalezieniu obiektu poszczególnym własnościom przypisywane są nowe dane jeśli takowe zostały przysłane, w przypadku ich braku zachowana zostaje stara wartość. Sprawdzane są również przysłane dywizje które są przypisane do sezonu, w przypadku nowych zostają one dodanych. Usuwanie dywizji z sezonu znajduje się w innej metodzie. Podczas wysyłania danych do edycji również następuje walidacja, analogiczna do tej podczas dodawania. Kod odpowiedzialny za edycję znajduje się na listingu 6.8.

Listing 6.8: Edycja sezonu

```
var newSeasonDivisions = new List<SeasonDivision>();

```

```
3   var entity = await _context.Season.FindAsync( request.Id);
4
5   if (entity == null)
6     throw new NotFoundException(nameof(Season), request.Id);
7
8   entity.Name = request.Name ?? entity.Name;
9   entity.StartDate = request.StartDate ?? entity.StartDate;
10  entity.EndDate = request.EndDate ?? entity.EndDate;
11
12  if (request.DivisionsId.Any())
13  {
14    var seasonDivisionsInDb = _context.SeasonDivision.Where(x => x.
        ↪ SeasonId == entity.Id).Select(x => x.DivisionId).ToList();
15    foreach (var divisionId in request.DivisionsId)
16    {
17      var seasonDivision = new SeasonDivision
18      {
19        SeasonId = entity.Id,
20        DivisionId = divisionId,
21      };
22      newSeasonDivisions.Add(seasonDivision);
23    }
24    newSeasonDivisions.RemoveAll(x => seasonDivisionsInDb.Contains(x.
        ↪ DivisionId));
25    _context.SeasonDivision.AddRange(newSeasonDivisions);
26  }
27
28  var success = await _context.SaveChangesAsync(cancellationToken) > 0;
29  if (success) return Unit.Value;
30
31  throw new Exception("Problem saving changes");
```

Usuwanie obiektu wykonywane jest na podstawie Id wysłanego w zapytaniu, dzięki po którym szuka się sezonu do usunięcia. Gdy takiego nie ma rzucany jest wyjątek. W przypadku powodzenia sezon zostaje usunięty a zmiany zostają zapisane w bazie. W przypadku problemów z zapisem rzucany jest wyjątek. Omawianą sytuację przedstawia listing 6.9.

Listing 6.9: Usuwanie sezonu

```
1   var entity = await _context.Season.FindAsync(request.Id,
      ↪ cancellationToken);
2
3   if (entity == null)
4     throw new NotFoundException(nameof(Season), request.Id);
5
6   _context.Season.Remove(entity);
7
8   var success = await _context.SaveChangesAsync(cancellationToken) > 0;
9   if (success) return Unit.Value;
10
11  throw new Exception("Problem saving changes");
```

6.3.1. Zarządzanie drużynami oraz zawodnikami

Drużyny oraz zawodników przypisuje się do sezonów oraz dywizji. Tworzy się w ten sposób obiekty *PlayerSeason* oraz *TeamSeason*. Obiektowi zostaje przypisany Id sezonu oraz dywizji, którego ma dotyczyć. Zapisany zostaje także Id drużyny do której w danym sezonie oraz dywizji zostaje zapisany. Warto zaznaczyć że jest to referencja do obiektu *TeamSeason*, a nie *Team*. Kod odpowiadający za tworzenie obiektu przedstawia poniższy listing 6.10.

Listing 6.10: Przypisywanie zawodnika do sezonu oraz dywizji

```
var seasonDivision = await _context.SeasonDivision.FirstOrDefaultAsync
    ↪ (
x => x.SeasonId == request.SeasonId && x.DivisionId == request.
    ↪ DivisionId, cancellationToken);

if (seasonDivision == null)
  throw new NotFoundException(nameof(SeasonDivision), request.SeasonId
     ↪ );

var entity = new PlayerSeason
{
  PlayerId = request.PlayerId,
  SeasonDivisionId = seasonDivision.Id,
  TeamId = request.TeamId,
  JerseyNr = request.JerseyNr,
};

_context.PlayerSeason.Add(entity);

var success = await _context.SaveChangesAsync(cancellationToken) > 0;
if (success) return Unit.Value;

throw new Exception("Problem saving changes");
```

6.4. Zarządzanie meczami i program do pisania statystyk

6.5. Przesyłanie wyniku meczu na żywo

Rozdział 7

Opis działania aplikacji

7.1. Strona domowa

7.2. Tabele

7.3. Terminarz

7.4. Statystyki

7.5. Zawodnicy

7.6. Drużyny

7.7. Panel zarządzania meczami

7.8. Panel zarządzania ligą

Rozdział 8

Podsumowanie

ma się odnosić do celu pracy, spinają co chcieliśmy zrobić a co się udało

Literatura

[1] What is Code-First? https://www.entityframeworktutorial.net/code-first/what-is-code-first.aspx [dostęp dnia 20 października 2020].

[2] E. P. Alex Banks. *Learning React, 2nd Edition*. O'Reilly Media, 2020.

[3] M. Dabbs. The Fundamentals of Web Application Architecture. https://reinvently.com/blog/fundamentals-web-application-architecture/ [dostęp dnia 07 listopada 2020].

[4] P. Dixon. What is TypeScript and why would I use it in place of JavaScript? Paz. 2012. https://stackoverflow.com/questions/12694530/what-is-typescript-and-why-would-i-use-it-in-place-of-javascript [dostęp dnia 20 października 2020].

[5] I. Griffiths. *Programming C# 8.0*. O'Reilly Media, 2019.

[6] K. Grudzień. Jak działa C# ? Mar. 2019. https://it-depends.dev/jak-dziala-csharp.html [dostęp dnia 20 października 2020].

[7] M. J. Price. *C# 8.0 and .NET Core 3.0 - Modern Cross-Platform Development*. O'Reilly Media, wydanie 1, 2019.

[8] R. P. Przemysłowy.pl. Interfejsy REST API w praktyce przemysłowej. Luty 2020. https://portalprzemyslowy.pl/utrzymanie-ruchu-produkcja/utrzymanie-ruchu-lean-tqm-uslugi-dla-przemyslu/interfejsy-rest-api-w-praktyce-przemyslowej/ [dostęp dnia 20 października 2020.

[9] M. Żywczok. O mocnych i słabych stronach ReactJS. List. 2019. https://geek.justjoin.it/zalety-wady-reactjs [dostęp dnia 20 października 2020].

Indeks rzeczowy