Uniwersytet Warmińsko-Mazurski w Olsztynie

Wydział Matematyki i Informatyki

**Kierunek: Informatyka**

**Sebastian Krzynówek**

**Aplikacja umożliwiająca prowadzanie, archiwizację oraz analizę statystyk**

Praca inżynierska wykonana w Katedrze Metod Matematycznych Informatyki pod kierunkiem dr. Pawła Drozdy

Olsztyn, 2022 rok

University of Warmia and Mazury in Olsztyn Faculty of Mathematics and Computer Science

**Field of Study: Computer Science**

**Sebastian Krzynówek  
  
  
Tracking, archiving and analyzing statistics application**

Engineer’s Thesis is performer in the Chair of Mathematical methods in Computer Science under supervision of Paweł Drozda, PhD

Olsztyn, 2022 rok

# **Streszczenie**

Celem pracy dyplomowej było stworzenie aplikacji umożliwiającej utworzenie klubu piłkarskiego wraz z zawodnikami oraz trenerem oraz prowadzenie wirtualnych statystyk każdego z piłkarzy. Do wykonania pracy został użyty język programowania C#, oprogramowanie APS .NET Core oraz zestaw narzędzi programistycznych Swagger.

Praca składa się z trzech rozdziałów, które opisują udział w tworzeniu aplikacji. Rozdziały dzielą się na: wprowadzenie do języka, w którym została napisana aplikacja, czyli C# oraz oprogramowania ASP .NET Core. Drugi skupia się na pojęciach takich jak Web API, czyli na typie aplikacji, którym jest aplikacja pracy inżynierskiej, clean architecture, czyli architektura, w której została napisana aplikacja oraz CQRS wraz z wzorcem projektowym Mediator. Ostatni rozdział został poświęcony efektowi pracy – aplikacji.

# **Abstract**

The aim of this diploma thesis was to create an application enabling us to create football club with coach and football players with their virtual statistics. The C# programming language, ASP .NET Core software and the Swagger development toolkit were used to do the work.

The work consists of three chapters that describe participation in the development of the application. First chapter are divided into an introduction to the language in which application was written, C# and ASP .NET Core. The second chapter focuses on concepts such as Web API (type of application), clean architecture – architecture in which application was written and CQRS along with the Mediator design pattern. The last chapter is devoted to the effect of the work – the application.

**Spis treści**

[**Streszczenie** 3](#_Toc103564007)

[**Abstract** 4](#_Toc103564008)

[Wstęp 7](#_Toc103564009)

[**Rozdział 1 C# i ASP .NET Core** 8](#_Toc103564010)

[**1.1 C#** 8](#_Toc103564011)

[1.1.1 Czym jest język programowania? 8](#_Toc103564012)

[1.1.2 Czym jest C#? 9](#_Toc103564013)

[1.1.3 IDE – środowisko programistyczne 9](#_Toc103564014)

[1.1.4 Pierwszy program 11](#_Toc103564015)

[1.1.5 Zmienne oraz operacje na zmiennych w języku programowania C# 13](#_Toc103564016)

[1.1.6 Programowanie obiektowe 16](#_Toc103564017)

[**1.2 ASP .NET Core** 17](#_Toc103564018)

[1.2.1 Czym jest ASP .NET Core 17](#_Toc103564019)

[1.2.2 Najważniejsze funkcjonalności ASP .NET Core 17](#_Toc103564020)

[**Rozdział 2 WebAPI, Clean Architecture oraz MediatR, czyli CQRS w połączeniu ze wzorcem projektowym Mediator** 19](#_Toc103564021)

[**2.1 WebAPI** 19](#_Toc103564022)

[2.1.1 Czym jest WebAPI? 19](#_Toc103564023)

[2.1.2 ASP .NET WebAPI 20](#_Toc103564024)

[**2.2 Clean architecture** 21](#_Toc103564025)

[2.2.1 Czym jest Clean architecture? 21](#_Toc103564026)

[2.2.2 Wady oraz zalety użycia Clean architecture. 22](#_Toc103564027)

[**2.3 MediatR: Wzorzec projektowy Mediator oraz CQRS** 23](#_Toc103564028)

[2.3.1 Czym jest MediatR, CQRS oraz Mediator? 23](#_Toc103564029)

[**Rozdział 3 Projekt aplikacji webowej** 25](#_Toc103564030)

[**3.1 Założenia aplikacji** 25](#_Toc103564031)

[**3.2 Struktura aplikacji** 25](#_Toc103564032)

[3.2.1 Moduł API 26](#_Toc103564033)

[3.2.2 Moduł Application 29](#_Toc103564034)

[3.2.3 Moduł Domain 31](#_Toc103564035)

[3.2.4 Moduł Infrastructure 36](#_Toc103564036)

[3.2.5 Moduł Shared 40](#_Toc103564037)

[**3.3 Omówienie kontrolerów, zapytań oraz poleceń** 42](#_Toc103564038)

[3.3.1 Coaches 42](#_Toc103564039)

[3.3.2 Players 46](#_Toc103564040)

[3.3.3 Statistics 53](#_Toc103564041)

[3.3.4 Teams 57](#_Toc103564042)

[**3.4 Działanie aplikacji** 66](#_Toc103564043)

[3.4.1 Działanie aplikacji z wykorzystaniem SwaggerUI 66](#_Toc103564044)

[**Podsumowanie** 73](#_Toc103564045)

[**Bibliografia** 74](#_Toc103564046)

# **Wstęp**

Piłka nożna to najbardziej popularny sport na świecie posiadający wiele miliardów fanów na całym świecie. Żaden inny sport nie wywołuje tak intensywnych i różnorakich doznań. Na całym świecie jest również wiele klubów piłkarskich oraz piłkarzy, którzy grają

w barwach danego klubu, także każdy kibic piłki nożnej może znaleźć jeden, lub więcej klubów, którym sympatyzuje. Co jednak, gdybyśmy chcieli, wirtualnie, utworzyć własny klub, zakontraktować najlepszego trenera i najlepszych piłkarzy na świecie oraz analizować ich statystyki?

Naprzeciw wymaganiom przychodzi niniejsza praca dyplomowa, która została utworzona w celu umożliwienia takim działaniom. Aplikacja, opisana poniżej, jest to aplikacja Web Api z wielorakimi kontrolerami akcyjnymi, która umożliwi nam podstawowe działania utworzenia klubu piłkarskiego, zakontraktowanie trenera oraz podpisanie piłkarzy, których statystyki będziemy mogli analizować oraz aktualizować z meczu na mecz.

Realizacja tematu została podzielona na rozdziały, z których pierwszy będzie koncentrował się na języku programowania C# oraz oprogramowaniu ASP .NET Core. Zawarte są tutaj informacje o początkach programowania w języku C# oraz podstawowym projekcie ASP .NET Core.

Drugi rozdział został poświęcony samej definicji Web API oraz o projekcie aplikacji, czyli architekturze, w której została napisana aplikacji oraz CQRS ze wzorcem projektowym Mediator, który został zaimplementowany w aplikacji.

W ostatnim, trzecim rozdziale, została opisana napisana przeze mnie aplikacja, zaczynając od początkowych encji bazodanowych, idąc przez repozytoria wraz z ich metodami oraz kończąc na kontrolerach, endpoint’ach akcyjnych oraz zwracanych modelach po zapytaniach bazodanowych.

# **Rozdział 1 C# i ASP .NET Core**

## **1.1 C#**

### 1.1.1 Czym jest język programowania?

Mówiąc o języku programowania mamy na myśli zbiór zasad, który określając ciąg symboli tworzy program komputerowy oraz obliczenia, jakie ten program opisuje. Pozwala on na zapis algorytmów, czy też innych zadań, które później ma wykonać komputer.

Na świecie istnieje tysiące języków programowania, które można podzielić, np. ze względy na zastosowanie. Gdy myślimy o stworzeniu strony internetowej od razu na myśl mogą nasuwać się języki takie jak PHP, Java lub JavaScript. Gdy naszym celem jest utworzenie Web Api, tak jak w przypadku mojej aplikacji, przede wszystkim z wykorzystaniem ASP .NET Core przoduje tutaj język C#. Chcąc napisać aplikację mobilną na prowadzenie, w popularności języka programowania, wysuwają się tutaj ( w zależności od platformy Android/IOS) Kotlin/Kotlin Native. Przy pisaniu gry komputerowej podstawą będzie poznanie języków C, C++ oraz C#. Języki, które przodują w obróbce danych oraz w wyłuskiwaniu z nich konkretnych informacji to jeżyk R oraz wszechstronny Python. Tak (według serwisu www.nexttechnology.io ) prezentuje się dziesięć najpopularniejszych języków programowania w 2022 roku:



Rysunek 1.1 Dziesięć najpopularniejszych języków programowania w 2022 według nexttechnology.io

### 1.1.2 Czym jest C#?

C# jest to nowoczesny, wysokopoziomowy, obiektowy język programowania, który umożliwia tworzenie składników programowania i ich używania. Język C# jest językiem zorientowanym obiektowo, w którym definiujemy typy oraz ich zachowanie.

Funkcje C#, które pomagając tworzyć niezawodne i trwałe aplikacje:  
- Odzyskiwanie pamięci – jest to funkcja, która automatycznie odzyskuje pamięć, która została zajęta przez nieużywane obiekty,

- Typy dopuszczane wartością *null* – takie typy chronią przed zmiennymi, które nie odwołują się do przydzielonych obiektów,

- Obsługa wyjątków – funkcja zapewnia nam ustrukturyzowane oraz rozszerzalne podejście do wykrywania oraz późniejszego obsługiwania błędów,

- Wyrażenia lambda – są to wyrażenia, które obsługują techniki programowania funkcjonalnego,

- LINQ – jest to składnia zapytania (LANGUAGE Integrated Query), która tworzy wspólny wzorzec pracy z jakimikolwiek danymi,

- Asynchroniczność – obsługa języka dla asynchronicznych operacji zapewnia składnię do tworzenia systemów rozproszonych,

- Ujednolicony system typów – wszystkie typy, w tym pierwotne, w języku C# dziedziczą po pojedynczym typie *object.* Ponad to, język C#, obsługuje też zdefiniowane przez nas typy referencyjne oraz typy wartości. Poprzez obsługę metod i typów ogólnych, język, zapewnia także większe bezpieczeństwo i wydajność typów.

### 1.1.3 IDE – środowisko programistyczne

IDE ( ang. Integrated Developmnet Environment) jest to program, czy też zespół programów, który służy nam do tworzenia, testowania, modyfikacji oraz konserwacji oprogramowania. Do rozpoczęcia programowania nie jest potrzebna IDE, wystarczy zwykły notatnik czy rozbudowany edytor tekstu, jednakże w praktyce nie jest to efektywne podejście do sprawy. IDE jest również edytorem kodu, jednak zawiera on w sobie także kompilator, narzędzia do poprawiana wyglądu kody, narzędzia służące do wykrywania błędów w składni lub praktyce pisania kodu oraz narzędzia umożliwiające testowanie i uruchomienie środowiska. Bardziej rozbudowane IDE integruje się także z Git’em, czyli rozproszonym systemem kontroli wersji.

IDE posiada wiele cech, dzięki którym praca programistów staje się coraz to bardziej wydajna oraz unowocześniona. Do najważniejszych cech rozbudowanego IDE nalezą:

- edytowanie kodu źródłowego – podstawowa cecha każdego IDE. Cecha ta umożliwia nam walidacje kodu, jak i również wskazywanie na błędy w kodzie już w trakcie samego pisania kodu.   
- wsparcie dla technologii, w której piszemy nasze oprogramowanie – umożliwia nam to istnienie kompilatora oraz interpretera kodu źródłowego a także sama możliwość uruchomienia kodu,

- debuggowanie oraz testowanie kodu – IDE posiada wbudowany debugger, który daje nam możliwość śledzenia linijka po linijce, krok po kroku, całego przebiegu działania naszej aplikacji,

- narzędzia do zarządzania projektami – dzięki temu programista może korzystać, np.

z managera plików, a także z wbudowanej w IDE przeglądarki solucji, klas czy chociażby obiektów,

- jednolity i prosty w obsłudze interfejs – IDE zawsze jest intuicyjne w obsłudze, co wspomaga efektywność pracy programisty,

- rozszerzenia, narzędzia dodatkowe – IDE umożliwia rozszerzenie swoim narzędzi o narzędzia własne lub też innych wydawców, które wspomagają pracę programisty.

Obecnie, do najbardziej popularnych IDE należą te, które zostały stworzone przez firmę JetBrains, która oferuje IDE do każdego z najważniejszych języków programowania, np. Rider - .Net lub PHPStorm – PHP. Programiści kodujący w C# zazwyczaj korzystają ze stworzonego przez Microsoft środowiska – Visual Studio w najnowszej wersji 2022.

Obraz zawierający stół

Opis wygenerowany automatycznie

Rysunek 1.2. Dziesięć najpopularniejszych darmowych IDE według portalu www.guru99.com

### 1.1.4 Pierwszy program

Po wybraniu naszego IDE, w tym wypadku jest to Visual Studio 2019, chcemy rozpocząć naszą przygodę z programowaniem od pierwszego projektu. W otwartym panelu wybieramy opcję nowy projekt oraz aplikację konsolową w języku C#. Naszym oczom ukaże się już kultowy, pierwszy kod, który polega na wypisaniu w konsoli tekstu „Hello World!”. Taki program to już tradycja, którą widzi każdy początkujący lub dopiero wchodzący w świat programowania programista.

Obraz zawierający tekst, zrzut ekranu, ekran

Opis wygenerowany automatycznie

Rysunek 1.3 Kod pierwszego programu

Przeanalizujmy istniejący w tym programie kod:

- linijka pierwsza – using System; - jest to dyrektywa, która zawiera podstawowe i bazowe klasy, które definiują wartości oraz typy danych referencyjnych, interfejsy, procedury obsługi zdarzeń i wyjątki przetwarzania. W tym wypadku jest to klasa Console, która reprezentuje standard wyjść i wejść oraz obsługę błędów dla aplikacji konsolowej. Klasa Console zawiera metodę WriteLine(), której parametrem wejściowym jest, np. string, która to wypisuje nam

w konsoli podany wcześniej parametr.

- linijka trzecia – namespace ConsoleApp1 – jest to przestrzeń nazw, w której znajduje się nasz projekt, czyli *ConsoleApp1*,

- linijka piąta – class Program – wskazuje nam to na początek klasy o nazwie *Program*

- linijka siódma – static void Main(string[] args) – jest to inicjacja statycznej metody *Main,* która jako parametr wejściowy przyjmuje tablicę stringów *args*. Przechodząc do ciała metody możemy zauważyć, że metoda, po wywołaniu, wypisze nam na konsoli tekst „Hello World!”

Zbudujmy naszą aplikację i sprawdźmy, czy faktycznie tak się stanie.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 1.4 Konsola pierwszego programu C#

Spróbujmy zmienić parametr wejściowy i zamiast tekstu „Hello World!” zmieńmy go na „Witaj świecie!”.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 1.5 Pierwsza modyfikacja kodu aplikacji

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 1.7 Potwierdzenie modyfikacji kodu

Sukces, właśnie zmodyfikowaliśmy po raz pierwszy nasz kod.

### 1.1.5 Zmienne oraz operacje na zmiennych w języku programowania C#

Zmienna w programowaniu jest to nic innego jak miejsce, obszar w pamięci naszego komputera, do której możemy przypisać wartość. Do zidentyfikowania przez IDE zmiennej służy jej nazwa, czyli pierwszy z trzech podstawowych atrybutów zmiennej. Kolejnym z nich jest jej miejsce przechowywania, które znajduje się w pamięci komputera i określane jest przez długość danych oraz adres. Ostatnim podstawowym atrybutem zmiennej jest wartość, czyli to co znajduje się w miejscu przechowywania. Zmienna, czyli, potocznie mówiąc, porcja danych musi mieć wcześniej, z góry określony typ, dzięki czemu nasze IDE będzie wiedzieć jak interpretować określoną zmienną. W języku programowania C# zmienne muszą być wcześniej zainicjowane, aby wykonywać na niej jakiekolwiek możliwe operacje.

Wróćmy teraz do naszego pierwszego programu i spróbujmy zainicjować naszą pierwszą zmienną.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 1.8 Inicjacja zmiennej typu int o nazwie 'variable'

Po zainicjowaniu naszej zmiennej *variable* IDE, a w zasadzie jej kompilator otrzymuje informacje, dzięki której wie, jaki obszar w pamięci komputera musi zarezerwować, jak nazywa się dany obszar oraz gdzie będzie wartość, czyli 0, tej zmiennej przechowywana.

W języku C# wyróżniamy następujące rodzaje/typy zmiennych :

- typy proste/wartościowe – są to zmienne, które identyfikowane są za pomocą słów zarezerwowanych, inaczej mówiąc, mogą mieć one wartość przypisaną bezpośrednio. Zmienne tego typu po prostu zawierają dane. Kiedy deklarujemy zmienną typu wartościowego system od razy wie, że należy przydzielić pamięć komputera na tyle dużą, aby przechować naszą zmienną.

|  |  |  |
| --- | --- | --- |
| **Typ** | **Opis** | **Wartość domyślna** |
| Bool | Typ logiczny | False |
| Char | Znak 16-bitowy | „\0” |
| Double | 64-bitowa wartość zmiennoprzecinkowych | 0.0 |
| Float | 32-bitowa wartość zmiennoprzecinkowa | 0.0 |
| Int | 32-bitowa liczba całkowita | 0 |
| Long | 64-bitowa liczba całkowita | 0 |

Tabela 1 Przykłady typów wartościowych w języku C#

- typy referencyjne – jak sama nazwa wskazuje, zmienne tego typu nie przechowują rzeczywistych wartości zmiennych, a przechowują właśnie referencję do zmiennych. Przykładem takich zmiennych może być np. obiekty czy też typ tekstowy – string ( przykład string’a w naszym programie to kultowe „Hello World!”)

- typy wskaźnikowe – zmienne, które są zmiennymi typu wskaźnikowego przechowują

w pamięci adres do danego typu. Wskaźnik, może trzymać adres dla tylko i wyłącznie dla typów wartościowych oraz tablic.

Wracając do naszego pierwszego programu spróbujmy wykonać parę operacji na naszej zmiennej *variable*.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 1.9 Operacje matematyczne na zmiennej 'variable'

Obraz zawierający tekst

Opis wygenerowany automatycznie

1.10 Wyjścia wartości zmiennej variable po wykonaniu operacji matematycznych

Na powyższych rysunkach widać, że na zmiennej *variable* zostały wykonane operacje matematyczne takie jak, po kolei, dodanie do wartości zmiennej 1, pomnożenie wartości razy 10, podzielenie wartości na 2 oraz odjęcie od podzielonej wartości 5. Wartości zmiennej *variable* po poszczególnych operacjach matematycznych są ukazane na konsoli.

### 1.1.6 Programowanie obiektowe

Programowaniem obiektowym nazywamy w zasadzie paradygmat programowania,

w którym to definiujemy programy za pomocą obiektów. Można powiedzieć, że programowanie obiektowe służy nam do tego, żebyśmy mogli zmodularyzować, to znaczy dzielić na małe uniwersalne ‘części’, kod. Programowanie obiektowe pozwala nam bardzo dobrze usystematyzować pracę programistów przy bardzo dużych projektach. Taki rodzaj programowania pozwala lepiej zrozumieć oraz przeanalizować kod każdemu z członków zespołu developerskiego. Proces poznawania programowania obiektowego przyczyni się do poznania przez nas paru podstawowych pojęć z tej dziedziny programowania:

- klasa – jest to definicja formatu danych oraz procedur dla danego obiektu,

- obiekt – jest to instancja klasy,

- pole – jest to cecha obiektu, która może przybierać postać danych, atrybutów, zmiennych czy też właściwości,

- metoda – jest to implementacja funkcji, która znajduje się w modelu klasy

Przejdźmy teraz do naszej aplikacji. Spróbujmy stworzyć klasę pies, która będzie zawierała pole imię, rasa, wiek psa oraz metodę *Szczekaj(),* która będzie wypisywać w konsoli tekst ‘Imię psa + szczeka!’.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 1.11 Implementacja klasy Dog wraz z metodą Szczekaj()

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 1.12 Wyjście konsoli metody Szczekaj

## **1.2 ASP .NET Core**

### 1.2.1 Czym jest ASP .NET Core

ASP .NET Core jest to framework, jeden z najbardziej popularnych, który służy do tworzenia aplikacji internetowych. Framework ten został przygotowany przez firmę Microsoft. Platforma ta umożliwia tworzenie aplikacji bądź też usług internetowych, IoT oraz wdrażanie ich w chmurze bądź też lokalnie. Platforma ASP .NET Core niesie za sobą następujące zalety:

- posiada ujednolicony scenariusz na potrzeby tworzenia interfejsu użytkownika oraz API,

- platforma jest projektowana pod cel testowania,

- umożliwia ona tworzenie i uruchamianie aplikacji nie tylko na systemach Windows, lecz także na systemach macOS oraz Linux,

- platforma integruje nowoczesne struktury po stronie klienta,

- platforma ma wbudowane Depedency Injection, to jest wstrzykiwanie zależności,

- platforma jest gotowa do pracy w chmurze oraz posiada system konfiguracji oparty na środowisku,

- ASP .NET Core posiada potok żądań HTTP, który zapewnia wysoką wydajność,

- platforma posiada narzędzia, które upraszczają tworzenie nowoczesnych internetowych aplikacji

### 1.2.2 Najważniejsze funkcjonalności ASP .NET Core

Sam framework posiada niezliczone funkcje, które pozwalają na pokonywanie wszelkich przeszkód stojących na drodze developera. ASP .NET Core wychodzi naprzeciw takim przeszkodom posiadając funkcjonalności takie jak:

- obsługa wielu platform i kontenerów – aplikacje ASP .NET Core można tworzyć oraz wdrażać na wielu systemach. Microsoft, jak też i społeczność developerów, włożył ogromny wysiłek

w uczynienie wielu systemów idealną platformą do wdrażania takich aplikacji. Powszechną metodą wirtualizacji stała się wirtualizacja za pomocą *dockerów* oraz tworzenie wraz

z zarządzaniem kontenerami np. za pomocą oprogramowania również stworzonego przez Microsoft -  *Microsoft Azure*.

- asynchroniczność za pomocą *async/await* – platforma cechuje się wielkim wsparciem wykorzystania asynchroniczności w programowaniu. *Async* zaimplementowany jest

w najbardziej popularnych klasach i w prawie wszystkich bibliotekach zewnętrznych.   
  
- ujednolicenie MVC i frameworków WebApi – wraz z pojawieniem się platformy ASP .NET Core struktura MVC oraz WebApi zostały ze sobą połączone, dzięki czemu w jednej apliakcji mamy możliwość do tworzenia aplikacji internetowych obsługujących HTML oraz usługę RESTful w oparciu o np. JSON’a,

- dependency injection – jest to preferowany sposób przekazywania kontekstów do kontrolerów MVC. Kontekst w tym znaczeniu rozumiemy jako pokrycie tworzenia dzienników zdarzeń, obsługę połączenia z bazą danych oraz wiele innych akcji/zdarzeń przekazywanych do kontrolera,  
  
- globalizacja i lokalizacja – platforma ułatwia nam lokalizowanie formatów liczbowych lub dat w aplikacjach internetowych. Przez korzystanie z zasobów jesteśmy w stanie dopasować naszą aplikację internetową do wielu języków. Pliki zasobów traktowane są jako repozytoria centralne a dostęp do tłumaczeń polega na odczytaniu, w konkretnym języku, odpowiedniej etykiety. Zasoby dzielimy na lokalne, czyli charakterystyczne dla danej strony oraz globalne, czyli charakterystyczne dla całej aplikacji,

- swaggerUI – narzędzie, które ułatwia dokumentacje oraz testowanie endpoint’ów API.

# **Rozdział 2 WebAPI, Clean Architecture oraz MediatR, czyli CQRS w połączeniu ze wzorcem projektowym Mediator**

## **2.1 WebAPI**

### 2.1.1 Czym jest WebAPI?

Interfejs programowania aplikacji – API, nazywamy zestaw definicji podprogramów oraz narzędzi, które posłużą nam do stworzenia aplikacji. Prościej, API możemy określić jako rodzaj interfejsu, który posiada wszelkie funkcje umożliwiające programistom łatwy dostęp do potrzebnych narzędzi, funkcji, systemu operacyjnego czy też innych usług wspomagających pracę programistów.   
 Istnieje dużo rodzajów interfejsów programowania, jednakże w tym rozdziale skupimy się na omówieniu interfejsu użytego przy tworzeniu aplikacji w pracy inżynierskiej – WebAPI.

WebAPI, jak wskazuje nam sama nazwa, jest to API w sieci, do którego uzyskujemy dostęp za pomocą protokołu HTTP. Do manipulowania WebAPI wykorzystujemy głównie cztery metody protokołu HTTP:

- HTTP GET – w przypadku, jeżeli chcemy, by zostały nam zwrócone jakiekolwiek informacje

z bazy danych, np. mamy do dyspozycji bazę psów, w bazie znajduje się pies o id = 2. Zakładając, że mamy nasze WebAPI wysyłamy do niego zapytanie poprzez adres URL np. api/psy/2. Na podstawie tego adresu możemy wywnioskować, że z bazy danych chcemy wyciągnąć informacje o psie, która ma numer id równy 2,

- HTTP POST – tego zapytania używamy w przypadku, gdy, biorąc za przykład naszą bazę danych psów, chcemy dodać do bazy nowy rekord. W tym zapytaniu wewnątrz wywołania przesyłamy potrzebne parametry, w tym wypadku, do dodania nowego psa do bazy danych. Zazwyczaj parametry przekazywane są w formacie JSON, np. chcemy dodać psa o imieniu *Ben,* rasie *Beagle* oraz wieku *1 rok,*  w takim razie jako parametr powinniśmy przekazać taki oto JSON:

{

„Name”: „Ben”,

„Breed”: „Beagle”,

„Age”: 1

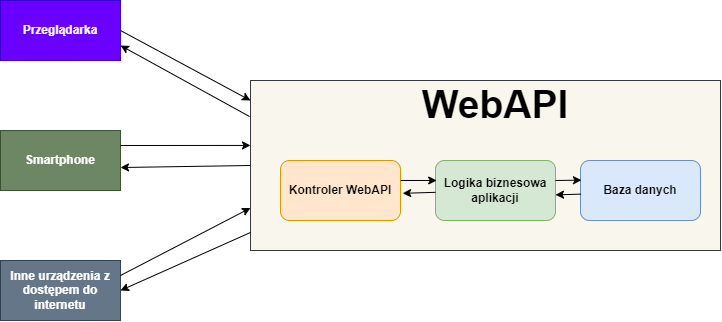
}

- HTTP PUT – tego zapytania użyjemy w przypadku, gdy chcemy zaktualizować dane danego rekordu w bazie danych. Biorąc pod uwagę przykład naszego psa Ben’a, którego dodaliśmy do bazy danych w poprzednim zapytaniu HTTP POST, załóżmy, że minął rok i należy zaktualizować jego wiek. Załóżmy również, że id naszego psa jest równe 28. Do naszego WebAPI wysyłamy zapytanie poprzez adres *api/psy/28* oraz przekazujemy JSON’a z wartościami, które chcemy zaktualizować, w tym przypadku będzie to tylko wiek, czyli:

{ „Age”: 2}

- HTTP DELETE – ostatnia z głównych metod protokołu HTTP. Zapytania tego używamy np. gdy chcemy usunąć dany rekord z bazy danych. Po raz kolejny biorąc pod uwagę nasz przykład bazy danych z psami załóżmy, że chcemy, nasz wcześniej dodany i zaktualizowany rekord psa Ben’a, skasować z bazy danych. Do WebAPI wysyłamy w tedy zapytanie poprzez adres, ten sam co w przypadku zapytania HTTP PUT, *api/psy/28*, który wskazuje nam na id psa, którego chcemy skasować z bazy danych – Ben’a.

Protokół HTTP jest również odpowiedzialny za statusy określające wynik naszego zapytania, przykładowo, gdy w zapytaniu HTTP GET chcieliśmy otrzymać informacje o psie, którego id jest równy 2, mogło okazać się, że taki rekord nie istnieje w bazie danych, wtedy zapytanie zwróci nam status 404 – Not Found, który wskazuje nam właśnie na to, że danego rekordu nie ma w bazie danych. W przypadku naszego zapytania HTTP POST, kiedy dodawaliśmy do bazy nowy rekord, operacja przeszła pomyślnie, zatem zapytanie zwróciło nam status 200 – OK, co wskazuje na poprawne przebiegnięcie procesu.



**Rysunek 2.1 Interfejs WebAPI**

### 2.1.2 ASP .NET WebAPI

ASP .NET WebAPI jest to platforma, która służy do tworzenia usług opartych właśnie na protokole HTTP, która oparta jest na ASP .NET. Taka platforma mapuje nam nasze czasowniki HTTP(GET, POST, PUT, DELET) na nazwy metod. Do jej działania potrzebna jest tylko i wyłącznie klasa, która dziedziczy po *ApiController.* Powiedzmy, że nasza klasa dziedzicząca po klasie *ApiController*, która możemy nazwać DogsController, posiada metodę *GetDog(int id).* Ta metoda mówi nam, że po jej wywołaniu otrzymamy informacje na temat psa, a konkretnie określonego psa, ponieważ w parametrze metody przekazujemy numer id psa. Framework ASP .NET WebApi zmapuje tę metodę na zapytanie GET, do którego przekażemy numer id psa, o którym informacje chcemy uzyskać. W ciele metody *GetDog(int id)* musimy napisać logikę, która na końcu zwróci nam rekord, którego oczekujemy. Końcowym wynikiem, oczywiście zakładając, że pies o numerze id równym numerowi, który przekazaliśmy w parametrze metody, istnieje, zostanie nam zwrócony status równy 200 – OK wraz z JSON’nem, zawierającym nam dane psa. Więcej o samych kontrolerach oraz metodach w rozdziale 3,

w którym to będzie opisana aplikacja.

## **2.2 Clean architecture**

### 2.2.1 Czym jest Clean architecture?

Gdy rozpoczynamy nowy projekt, jako programiści, warto jest się zastanowić od razu nie na pierwszych zadaniach czy funkcjonalnościach, które chcemy napisać a właśnie nad architekturą, w której zostanie napisana nasza aplikacja. Architektura aplikacji jest to nic innego jak wzorzec, którym posłużymy się przy budowanie i pisaniu naszej aplikacji. Poświęcony czas na wybór wzorca oraz spokojnego i efektownego zaimplementowania wzorca jest tylko to bardzo dobry fundament do przyszłego działania aplikacji oraz pracy

z samą aplikacją.

Clean architecture jest to wzorzec architektury projektowej, którego założeniem, jest wydzielenie czterech głównych funkcjonalnych modułów w projekcie. Są to:

- **Application** – moduł, który jest najbardziej i najmocniej związany z technologią, w której używamy naszej architektury. Ten moduł odpowiada za wizualną część aplikacji. Moduł **Application** podejmuje decyzje o tym, w jaki sposób będą wyświetlane dane, które zostaną dostarczone z modułu **Presentation.** Kolejnym zadaniem modułu, które będzie powiązane

z modułem **Presentation**, jest obsługa informowania warstwy o wszelkich wykonanych przez użytkownika akcjach,

- **Presentation** – ten moduł odpowiada za sterowanie/kontrolowanie całej aplikacji. W modelu tym są obsługiwane dane zadania oraz to, jaką część wizualną należy przekazać użytkownikowi. Moduł **Presentation** służy głównie do wykonywania naszej logiki biznesowej lub też pobierania jakiś danych, np. z bazy danych.

- **Domain** – jest to moduł, który zawiera naszą całą logikę biznesową aplikacji. Moduł **Domain** składa się z wielu składowych:

- UseCase’y – jest to odwzorowanie wymagań biznesowych. W zasadzie UseCase’y umożliwiają nam komunikację pomiędzy modułami **Domain** a **Presentation**,

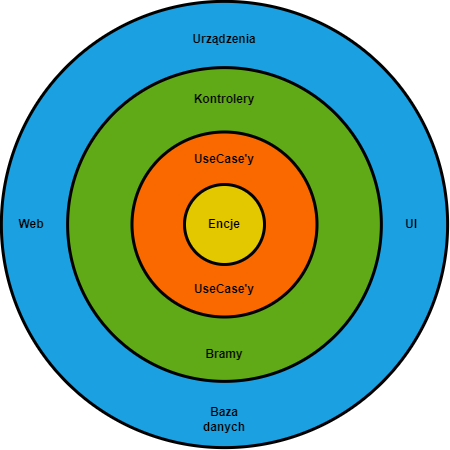
- Model – jest to zbiór danych, mogą to być np. encje bazodanowe lub obiekty, które są wykorzystywane później przez logię biznesową,

- Logic – w tym miejscu przechowujemy klasy z rozbudowaną logiką, co zapobiegnie nam tworzeniu rozbudowanych UseCase’ów,

- DataSource – jest to miejsce, które decyduje, skąd będą pobierane/modyfikowane/zapisywane/usuwane dane, czyli w tym miejscu znajdziemy odniesienie do naszej bazy danych. Prócz danych w tym modelu możemy znaleźć konfiguratory, które umożliwią nam narzucenie naszej logiki biznesowej jak mają być tworzone tabele i jak mają wyglądać obiekty w naszej bazie danych

- Mappery – odpowiadają one za to, by logika biznesowa działa w pełni bez zarzutów,

- **Data** – jest to moduł, który odpowiada za dostarczanie wszelkiego typu danych oraz modeli, które odwzorowują konkretne dane.



Rysunek 2.2 Diagram Clean Architecture

### 2.2.2 Wady oraz zalety użycia Clean architecture.

Każdy wzorzec architektury projektu ma swoje wady i zalety, dlatego trzeba brać pod uwagę jakiego typu jest nasz projekt i wtedy wybrać odpowiedni wzorzec architektury projektu.

Samo zdecydowanie się na użycie wzorca projektowego jest już dużą wartością dodaną do naszego projektu oraz przyszłej pracy w nim. Przejdźmy w takim razie do zalet clean architecture:

- łatwość utrzymania – wykorzystanie w naszym projekcie wzorca clean architecture znacznie ułatwi nam prace i utrzymanie projektu. Naprawianie błędów w naszym projekcie będzie bardzo proste, a samo lokalizowanie błędów nie zajmie nam dużo czasu,

- duża responsywność – użycie tej architektury przyspieszy nam modyfikacje naszego kodu, ponieważ nie będziemy musieli się martwić o konsekwencje w pozostałych częściach projektu,

- wiele małych klas i interfejsów – dzięki takiemu podejściu otrzymamy łatwy kod, który ma wydzielone poszczególne funkcjonalności, dzięki czemu łatwiej też będzie nam pokryć te funkcjonalności testami.

Tak jak wspomniałem wcześniej, każdy wzorzec posiada też wady. Nasz omawiany wzorzec posiada wady takie jak:

- czasochłonność – zaimplementowanie samego wzorca wymaga bardzo dużo czasu oraz spokoju developera, a późniejsze dodawane funkcjonalności oraz rozwijanie aplikacji zajmuje trochę więcej czasu przez to, że architekturę należy utrzymać w danym wzorcu,

- trudny początek – clean architecture nie należy do najprostszych wzorców architektury projektu, przez co mniej doświadczeni developerzy mogą mieć problem

z zaimplementowaniem tego wzorca w swoim projekcie.

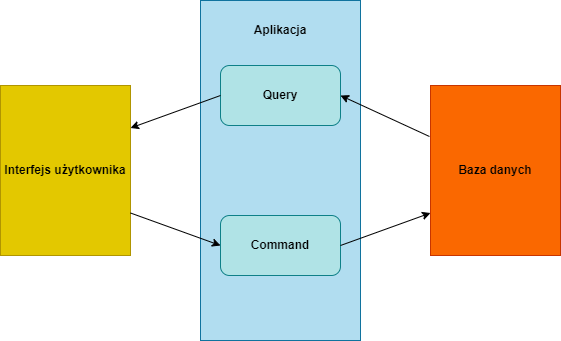
## **2.3 MediatR: Wzorzec projektowy Mediator oraz CQRS**

### 2.3.1 Czym jest MediatR, CQRS oraz Mediator?

MediatR jest niczym innym jak, możliwą do pobrania, biblioteką, która została stworzona w celu ułatwienia korzystania z dwóch wzorców projektowych – CQRS oraz Mediator. Chociaż oba wzorce w zasadzie mogą wydawać się podobne, spróbujemy przeanalizować teraz zasady, które kryją się pod każdym z nich.

Command Query Responsibility Segregation, czyli w skrócie CQRS jest to rozdzielenie odpowiedzialności, jak sama nazwa wskazuje, naszych poleceń (command’ów) oraz zapytań (queries’ów). Jeżeli myślimy o podstawowym wzorcu CRUD (Create Read Update Delete), zazwyczaj myślimy o interfejsie, który współpracując z jakimkolwiek datastorem, odpowiada za wszystkie cztery akcje. CQRS dzieli te operacje na dwa modele, gdzie zapytanie, czyli query, odpowiada za R w CRUD, a polecenie, czyli command, odpowiada za pozostałe CRU

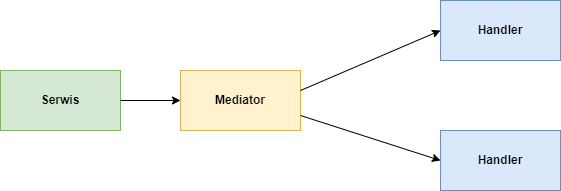
w CRUD’zie.



Rysunek 2.3 CQRS

Wzorzec w teorii nie nakłada na nas wymagań dotyczących separacji, dzięki czemu mamy pełne pole do popisu w sprawie implementacji wzorca.

Wzorzec Mediator służy nam do definiowania obiektu, który zawiera sposób interakcji w jaką wchodzę obiekty. Dzięki temu zamiast mieć wiele obiektów, które są bezpośrednio od siebie zależne, będziemy mieli obiekt ‘mediator’, który jest odpowiedzialny za wysyłanie tych interakcji pomiędzy obiektami.



Rysunek 2.4 Wzorzec Mediator

Wzorzec Mediator działa w sposób przedstawiony na rysunku powyżej. Mając serwis, która wysyła wiadomość do mediatora, mediator wywołuje usługi, które obsłużą te wiadomości (handlery). Żadne z obiektów (handlerów) nie jest bezpośrednio od siebie zależna.

Taka implementacja wzorca Mediator umożliwia nam zminimalizowanie wykresu zależności, dzięki czemu nasz kod jest prostu i bardzo łatwy do późniejszego testowania.

Biblioteka MediatR umożliwia nam połączenie obu wzorców. W myśl założenia biblioteki Mediator pomagam nam budować systemy CQRS. Komunikacja, która odbywa się pomiędzy jakimkolwiek magazynem danych a interfejsem użytkownika odbywa się za pośrednictwem MediatR. Korzystanie z biblioteki MediatR niesie za sobą wiele zalet, takich jak:

- łatwość modyfikacji – w dalszych etapach pracy w projekcie developerzy są łatwo i szybko

w stanie modyfikować lub naprawiać niedziałający kod,

- łatwość testowania – kod po zaimplementowaniu obu wzorców jest bardzo łatwy do przetestowania,

- programowanie według wzoru – cały kod w projekcie ma swój wzór, dzięki czemu łatwo jest się poruszać wewnątrz projektu oraz przyszli developerzy, którzy zaczynają pracę w takim projekcie mają łatwość w odnalezieniu się w nim,

- SOLID – zaimplementowanie obu wzorców projektowych pozwala na bardzo poważne

traktowanie zasady *Single Responsibility Principle*, która mówi nam, że nie powinna istnieć

klasa, która będzie spełniać więcej niż jedną funkcjonalność.

# **Rozdział 3 Projekt aplikacji webowej**

## **3.1 Założenia aplikacji**

Głównym założeniem pracy było stworzenie aplikacji, która pozwoli na przeniesienie swojego klubu wraz z piłkarzami oraz trenerem z gry sportowej „Fifa” do aplikacji i po każdym rozegranym turnieju, który odbywa się raz w tygodniu, będziemy mogli aktualizować

i zapisywać statystyki piłkarzy, takie jak np. bramki czy asysty. Koniec końców aplikacja może służyć również do „wirtualnej zabawy” poprzez tworzenie swojego klubu piłkarskiego

i przypisywaniu do niego piłkarzy, jakich tylko chcemy. Efektem końcowym jest aplikacja webowa o nazwie „FutStats” ( nazwa pochodzi od internetowego trybu gry „Fifa” – „Fifa Ultimate Team” oraz od słowa statystyki), która została napisana przy użyciu języka programowania C#, framework’u ASP .NET Core oraz wzorców CQRS, Mediator, Clean Architecture a za interfejs użytkownika posłuży nam SwaggerUI. Aplikacja przeznaczona jest dla każdego fana piłki nożnej, który lubi wcielić się w role wirtualnego zarządcę klubu.

## **3.2 Struktura aplikacji**

Aplikacja została zgodnie z profilaktyką pisania architektury aplikacji Clean Architecture.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.1 Clean Architecture aplikacji FutStats

### 3.2.1 Moduł API

Przejdźmy teraz do omówienia co poszczególne modułu w naszej architekturze posiadają.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.2 Moduł API

Zaczynając od modułu API i idąc po kolei w eksploratorze solucji, mamy do czynienia

z folderem *Abstractions,* który zawiera klasę *ApiBaseController,* po której będą dziedziczyć wszystkie nasze kontrolery akcyjne. Idąc dalej, znajdują się foldery takie jak *Coach, Player, Statistic* oraz *Team*, a w nich kolejno *InputModels, ResultModels* oraz kontrolery. Jest to część aplikacji przechowująca rodzaje zapytań w protokole HTTP ( kontrolery). InputModels są to modele, które będą potrzebne, aby wykonać jakieś polecenie (command), np. *CreateCoachInputModel* jest to model, jak sama nazwa wskazuje, który będzie nam potrzebny do polecenia utworzenia rekordu trenera w bazie danych. ResultModels są to modele, na które będą mapowane nasze zwroty z zapytań (query) do bazy danych, np. *CoachesResultModel*, jak sama nazwa wskazuje, będzie to model, na który będzie mapowany wynik zapytania o wszystkich trenerów do bazy danych. Dalej mamy plik *appsettings.json*, który jest plikiem konfiguracyjnym, gdzie najważniejszym miejscem w tym configu jest nasz *connection string* do bazy danych:

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.3 Plik konfiguracyjny appsettings.json

W tym przypadku mamy bazę danych, która istnieje na komputerze lokalnym, aby zmienić bazę na inną, np. serwerową, musimy zmienić nasz *DefaultConnection*  i w miejscu wyznaczonym "**Server=localhost**;Database=FutStats;Trusted\_Connection=True;MultipleActiveResultSets=true"

zmienić pogrubioną wartość. Z naszego connection string’a wynika, że baza danych nazywać się będzie *FutStats,* czyli tak samo jak aplikacja.

Następnie, w dalszej części modułu API widzimy klasę *Program.cs*, który zawiera metodę odpowiadającą za zbudowaniem nam aplikacji – *CreateHostBuilder*

*Obraz zawierający tekst

Opis wygenerowany automatycznie*

Rysunek 3.4 Klasa Program.cs

Moduł ten kończymy na klasie *Startup.cs*, w której to mamy wstrzyknięcie zależności (dependency injection), czyli zarejestrowanie wszystkich serwisów, repozytoriów, zapytań, poleceń, dbContextu oraz zarejestrowanie swaggera, który będzie służyć jako UI

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.5 Wstrzyknięcie zależności w klasie Startup.cs

W klasie tej posiadamy również konfigurację naszej aplikacji.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.5 Konfiguracja aplikacji w klasie Startup.cs

### 3.2.2 Moduł Application

Kolejnym modułem w naszej aplikacji jest moduł *Application*.

Obraz zawierający tekst

Opis wygenerowany automatycznie

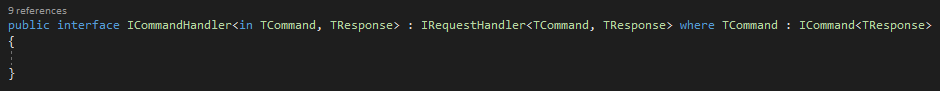
Rysunek 3.6 Moduł Application

Moduł ten zawiera jedynie folder Messaging, w którym wydzielone są foldery Abstractions oraz Commands. Pierwszy z nich zawiera dwa interfejsy, które będą służyły później w pisaniu klas obsługi poleceń.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.7 Interfejs ICommand.cs



Rysunek 3.7 Interfejs ICommandHandler.cs

Drugi z nich – Commands, zawiera podfoldery Coach, Player, Statistic oraz Team

a w nich wszystkie commandy (polecenia), które zostały użyte w aplikacji. Folder Coach posiada:

- CreateCoachCommand.ts – polecenie, które obsłuży nam utworzenie nowego trenera

w bazie danych.

Folder Player posiada:

- *CreatePlayerCommand.ts* – polecenie, które obsłuży nam utworzenie nowego gracza w bazie danych,

- *UpdatePlayerOfStatisticCommand.ts* – polecenie, które obsłuży nam zmodyfikowanie rekordu piłkarza od przypisanie do niego statystyk.

Folder Statistic posiada:

- *CreateStatisticCommand.ts –* polecenie, które obsłuży nam utworzenie nowego rekordu statystyk w bazie danych,

-  *UpdateStatisticCommand.ts –* polecenie, które obsłuży nam modyfikowanie istniejącego rekordu statystyki.

Folder Team posiada:

- *CreateTeamCommand.ts -*  polecenie, które obsłuży nam utworzenie nowego rekordu drużyny w bazie danych,

- *FireCoachCommand.ts –* polecenie, które obsłuży nam usunięcie trenera z istniejącej drużyny,

- *HireCoachCommand.ts -*  polecenie, które obsłuży nam przypisanie trenera do istniejącej drużyny,

- *SignPlayerCommand.ts –* polecenie, które obsłuży nam przypisanie piłkarza do istniejącej drużyny.

### 3.2.3 Moduł Domain

Następnym modułem jest moduł Domain, który przechowuje nam encje bazodanowe, wykorzystywane enumy oraz interfejsy repozytoriów, które przechowują głowy metod.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.8 Moduł Domain

W folderze Entities istnieją wszystkie klasy, które są modelami rekordu bazodanowego:

- *CoachEntity.cs –* model trenera, który jako klucz główny będzie posiadał Id typu Guid a reszta jego właściwości to Imię, Nazwisko, Narodowość oraz dwie wartości typu nullable Guid TeamId oraz TeamEntity Team, ponieważ może istnieć trener, który nie jest przypisany do żadnej drużyny,

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.9 Encja CoachEntity

- *PlayerEntity.cs –* model piłkarza, który jako klucz główny będzie posiadał Id typu Guid a reszta jego właściwości to Imię, Nazwisko, NumerNaKoszulce, Pozycja oraz cztery właściwości typu nullable Guid StatisticEntityId oraz StatisticEntity StatisticEntity, ponieważ może istnieć piłkarz nie posiadający statystyk, nullable Guid TeamId oraz TeamEntity Team, ponieważ może istnieć piłkarz, który jest wolny na rynku – nie jest w żadnej drużynie,

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.10 Encja PlayerEntity

- *StatisticEntity.cs –* model statystyki, która jako klucz główny będzie posiadała Id typu Guid

a reszta jej właściwości to Gole, Asysty, ŻółteKartki, CzerwoneKartki oraz dwie wartości nullable Guid PlayerId oraz PlayerEntity Player, ponieważ może istnieć statystyka nie przypisana do żadnego piłkarza

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.11 Encja StatisticEntity

- *TeamEntity.cs –* model drużyny piłkarskiej, która jako klucz główny będzie posiadała Id typu Guid a reszta jej właściwości to Nazwa, nullable kolekcja Piłkarzy, ponieważ może istnieć drużyna bez piłkarzy, nullable Trener, ponieważ może istnieć drużyna bez trenera oraz Tydzień, który określa, numer tygodnia (turnieju), w którym brała udział ta drużyna.

Obraz zawierający tekst

Opis wygenerowany automatycznie

*Rysunek 3.11 Encja TeamEntity*

W folderze Enums istnieją dwa enum’y:

- Nationalities – przechowujący możliwe narodowości do wyboru

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.13 Enum Nationalities

- Positions – przechowujący wszystkie możliwe pozycje piłkarza na boisku

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.14 Enum Positions

Kolejnym folderem jest folder Repositories, który posiada interfejs IRepository.ts, po którym dziedziczy interfejs IEntitesRepository.ts po którym to dziedziczą wszystkie pozostałe interfejsy w folderze Entities.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.15 Interfejs IRepository.ts

Interfejs ten zawiera wszystkie definicje metod, które będą wykorzystywane później przy okazji zapytań, np. GetAll – pobierz wszystkie lub poleceń, np. Add - dodaj rekord do bazy danych.

Ostatnim intefejsem w module Domain jest interfejs IUnitOfWork, który zawiera definicję metody Complete, używanej przy okazji poleceń (commandów).

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.16 Intefejs IUnitOfWork

### 3.2.4 Moduł Infrastructure

Obraz zawierający tekst

Opis wygenerowany automatycznie

*Rysunek 3.17 Moduł Infrastructure*

Następnym modułem jest moduł Infrastructure, który podzielony jest na dwie części/foldery. Pierwszy z nich to Database, który posiada folder Migrations, który przechowuje wszystkie migracje, na podstawie których będzie tworzona nasza baza danych oraz folder Repositories.

Folder Repositories zawiera:

-klasę UnitOfWork, która dziedziczy po interfejsie IUnitOfWork oraz zawiera ciało metody Complete

Obraz zawierający tekst

Opis wygenerowany automatycznie

*Rysunek 3.18 Klasa UnitOfWork*

- klasę Repository, która dziadziczy po interfejsie IRepository oraz zawiera ciała metod

Obraz zawierający tekst

Opis wygenerowany automatycznie

*Rysunek 3.19 Klasa Repository*

- folder FutStatsRepositories, który zawiera EntitiesRepositoryBase, który dziedziczy po klasie Repository oraz interfejsie IEntitiesRepository

Obraz zawierający tekst

Opis wygenerowany automatycznie

*Rysunek 3.20 Klasa EntitiesRepositoryBase*

oraz inne repozytoria naszych encji, które dziedziczą po swoich interfejsach oraz po klasie EntitiesRepositoryBase, np. klasa CoachRepoository:

Obraz zawierający tekst

Opis wygenerowany automatycznie

*Rysunek 3.21 Klasa CoachRepository*

Jako ostatnie w folderze Database znajduje się klasa FutStatDbContext, która zawiera informacje, na podstawie których zostanie stworzona nasza baza danych. Klasa zawiera informacje jakie będą poszczególne tabele w bazie danych oraz jak mają być budowane te tabele, np. warunki na utworzenie tabeli z trenerami wyglądają następująco:

- musi posiadać klucz głównym, którym jest Id,

- Id jest wymagane,

- tabela posiada kolumnę Imię, która ma maksymalną długość znaków 50 oraz jest wymagana,

- tabela posiada kolumnę Nazwisko, która ma maksymalną długość znaków 50 oraz jest wymagana,

- tabela posiada kolumnę Narodowość, która jest wymagana

Obraz zawierający tekst

Opis wygenerowany automatycznie

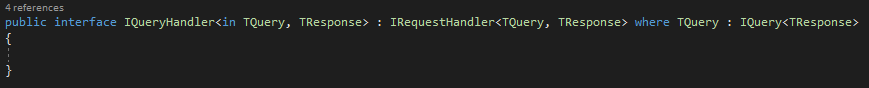
*Rysunek 3.22 Klasa FutStatsDbContext*

Kolejną częścią/folderem w naszym module jest folder Messaging, który zawiera foldery Abstractions oraz Queries. Folder Abstractions zawiera dwa interfejsy, które będą służyły później w pisaniu klas obsługi zapytań.

Obraz zawierający tekst

Opis wygenerowany automatycznie

*Rysunek 3.23 Interfejs IQuery*

**

*Rysunek 3.24 Interfejs IQueryHandler*

Drugim folderem w folderze Messaging jest folder Queries, która zawiera podfoldery Coach, Player, Team. Poszczególne foldery zawierają wszystkie istniejące zapytania, Folder Coach zawiera:  
- GetAllCoachesQuery – zapytanie, które zwróci nam wszystkich istniejących w bazie danych trenerów.

Folder Player zawiera:

- GetAllPlayersQuery – zapytanie, które zwróci nam wszystkich istniejących w bazie danych piłkarzy,

- GetPlayerWithStatisticsQuery – zapytanie, które zwróci nam konkretnego gracza wraz z jego statystykami.

Folder Team zawiera:

- GetTeamQuery – zapytanie, które zwróci nam konkretną drużynę wraz z jej trenerem oraz piłkarzami.

### 3.2.5 Moduł Shared

Moduł Shared jest ostatnim modułem, który znajduję się w strukturze architektury mojej aplikacji. Jest to najmniejszy ze wszystkich modeli i w zasadzie służy do tego, aby przechowywać tutaj rozszerzenia lub też serwisy, z których będziemy korzystali w więcej niż jednym miejscu w naszym kodzie – zapobiegnie to niepotrzebnej duplikacji, która nie chcemy, żeby miała miejsce.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.25 Moduł Shared

Moduł Shared jest bardzo podatny na aktualizacje i rozszerzanie. W mojej aplikacji znajduje się tylko i wyłącznie folder Infrastracture, który zawiera klasę będącą rozszerzeniem do EntityFramework’a – EntityFrameworkExtensions.cs.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.25 Klasa rozszerzenie EntityFrameworkExtensions.cs

## **3.3 Omówienie kontrolerów, zapytań oraz poleceń**

### 3.3.1 Coaches

Kontrolery są to klasy, które dziedziczą, w naszym przypadku, po klasie ApiBaseController oraz zawierają metody protokołu HTTP, które obsługują nasze polecenia lub zapytania. Przejdźmy w pierwszej kolejności do omówienia kontrolera CoachesController.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.26 Klasa CoachesController

Kontroler CoachesController zawiera ścieżkę api/coaches oraz dwie metody GetAllCoaches() oraz AddCoach();

Pierwsz z nich jest to zapytanie HTTP GET do bazy danych, które zwróci nam model CoachesResultModel, czyli listę trenerów wraz z ich imieniem, nazwiskiem oraz narodowością.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.27 Klasa CoachesResultModel

Zapytanie może nam zwrócić dwa statusy. W przypadku powodzenia – status 200 OK, natomiast w przypadku niepowodzenia – status 400 BadRequest. Przejdźmy teraz do klasy, która obsługuje dane zapytanie.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.28 Klasa GetAllCoachesQuery

Klasa ta zawiera dwie klasy. Pierwsza to CoachesResult, czyli model na jaki mają być mapowane dane oraz klasa Handler, która zawiera naszą całą logikę. W klasie Handler wstrzykujemy nasz FutStatDbContext, czyli naszą bazę, do której później się odniesiemy, aby wyciągnąć listę wszystkich rekordów trenerów z bazy danych.

Drugą metodą w naszym kontrolerze jest metoda AddCoach(), która jest zapytaniem HTTP POST, które obsługuje polecenie CreateCoachCommand. Ta metoda przyjmuje parametr, którym jest CreateCoachInputModel, którym jest model danych potrzebnych do stworzenia rekordu trenera w bazie danych.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.29 Klasa CreateCoachInputModel

Przejdźmy teraz do klasy, która obsługuje nam to polecenie.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.30 Klasa CreateCoachCommand

Konstruktor tej klasy przyjmuje nasz input model, który jest potrzebny do stworzenia rekordu trenera w bazie danych. CreateCoachCommand posiada także klasę Validator, która odpowiada za walidację danych, w tym przypadku walidacja jest ustawiona w następujący sposób :

- Imię musi być przynajmniej trzy-literowe oraz nie przekraczać 50 znaków,

- Nazwisko musi być przynajmniej trzy-literowe oraz nie przekraczać 50 znaków,

- Narodowość musi pochodzić z enum’a Nationalities.

W klasie Handler wstrzykujemy nasz ICoachRepository wraz z IUnitOfWork, dzięki czemu będziemy mogli skorzystać z metody Add() oraz metody Complete(), aby dodać nowo utworzony rekord do bazy.

### 3.3.2 Players

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.31 Klasa PlayersController

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.31 Klasa PlayersController

Kontroler PlayersController zawiera ścieżkę api/players oraz cztery metody GetPlayers(), GetPlayer(), AddPlayer() oraz AssignStatistic().

Pierwsza z nich to zapytanie HTTP GET do bazy danych, który zwróci nam model PlayersResultModel, czyli listę piłkarzy z ich imieniem, nazwiskiem, numerem na koszulce oraz pozycją na boisku.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.32 Klasa PlayersResultModel

Przejdźmy teraz do klasy, która obsługuje dane zapytanie. Klasa ta zawiera dwie klasy. Pierwsza to PlayersResult, czyli model na jaki mają być mapowane dane oraz klasa Handler, która zawiera naszą całą logikę. W klasie Handler wstrzykujemy nasz FutStatDbContext, czyli naszą bazę, do której później się odniesiemy, aby wyciągnąć listę wszystkich rekordów trenerów z bazy danych.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.33 Klasa GetAllPlayersQuery

Kolejną metodą w naszym kontrolerze jest zapytanie HTTP GET, które zwróci nam PlayerWithStatisticReslutModel, czyli model piłkarza wraz z jego statystykami. Metoda ta jako parametr wejściowy przyjmuje Id typu Guid, które jest po prostu Id piłkarza, którego wraz ze statystykami chcemy wyciągnąć z bazy danych.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.34 Klasa PlayerWithStatisticResultModel

Przejdźmy teraz do klasy, która obsługuje dane zapytanie. Klasa ta zawiera dwie klasy. Pierwsza to PlayerWithStatisticResult, czyli model na jaki mają być mapowane dane oraz klasa Handler, która zawiera naszą całą logikę. W klasie Handler wstrzykujemy nasz FutStatDbContext, czyli naszą bazę, do której później się odniesiemy, aby wyciągnąć listę wszystkich rekordów piłkarzy z bazy danych.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.35 Klasa GetPlayerWithStatisticQuery

Kolejną metodą w naszym kontrolerze jest metoda AddPlayer(), która jest zapytaniem HTTP POST, które obsługuje polecenie CreatePlayerCommand. Ta metoda przyjmuje parametr, którym jest CreatePlayerInputModel, którym jest model danych potrzebnych do stworzenia rekordu piłkarza w bazie danych. Modelem tym jest imię, nazwisko numer na koszulce oraz pozycja piłkarza na boisku.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.36 Klasa CreatePlayerInputMode

Przejdźmy teraz do klasy, która obsługuje nam to polecenie.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.37 Klasa CreatePlayerCommand

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.37 Klasa CreatePlayerCommand

Konstruktor tej klasy przyjmuje nasz input model, który jest potrzebny do stworzenia rekordu trenera w bazie danych. CreatePlayerCommand posiada także klasę Validator, która odpowiada za walidację danych, w tym przypadku walidacja jest ustawiona w następujący sposób :

- Imię musi być przynajmniej trzy-literowe oraz nie przekraczać 50 znaków,

- Nazwisko musi być przynajmniej trzy-literowe oraz nie przekraczać 50 znaków,

- Pozycja na boisku musi pochodzić z enum’a Positions,

- Numer na koszulce musi być od 1 do 99

W klasie Handler wstrzykujemy nasz IPlayerRepository wraz z IUnitOfWork, dzięki czemu będziemy mogli skorzystać z metody Add() oraz metody Complete(), aby dodać nowo utworzony rekord do bazy.

Ostatnią metodą w naszym kontrolerze jest zapytanie HTTP PUT, które obsługuje polecenie UpdatePlayerOfStatisticCommand. Parametrem wejściowym do tej metody jest UpdatePlayerOfStatisticValueInputModel, którym jest model danych potrzebnych do przypisania statystyk do piłkarza. Modelem tym jest id statystyki oraz id piłkarza.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.38 Klasa UpdatePlayerOfStatisticValueInputModel

Przejdźmy teraz do klasy, która obsługuje nam to polecenie.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.39 Klasa UpdatePlayerOfStatisticCommand

Konstruktor tej klasy przyjmuje nasz input model, który jest potrzebny do przypisania statystyk do piłkarza. Klasa Handler posiada obsługę dwóch wyjątków:

- przypadek, gdy statystyka o id równym id podanemu w parametrze wejściowym nie istnieje,

- przypadek, gdy piłkarz o id równym id podanemu w parametrze wejściowym nie istnieje.

W klasie Handler wstrzykujemy nasz IPlayerRepository, IStatisticRepository wraz z IUnitOfWork, dzięki czemu będziemy mogli zaktualizować rekordy oraz skorzystać z metody Complete(), aby dane rekordy zostały zaktualizowane w bazie danych.

### 3.3.3 Statistics

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.40 Klasa StatisticsController

Kontroler StatisticsController zawiera ścieżkę api/statistics oraz dwie metody AddStatistic() wraz z UpdateStatistic().

Pierwsza z nich to zapytanie HTTP POST, które obsługuję polecenie CreateStatisticCommand. Ta metoda przyjmuje jako parametr wejściowy CreateStatisticInputModel, którym jest model danych potrzebnych do stworzenia rekordu statystyki w bazie danych. Modelem tym jest liczba goli, liczba asyst, liczba żółtych oraz czerwonych kartek.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.41 Klasa CreateStatisticInputModel

Przejdźmy teraz do klasy, która obsługuje nam to polecenie.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.42 Klasa CreateStatisticCommand

Konstruktor tej klasy przyjmuje nasz input model, który jest potrzebny do stworzenia rekordu statystyki w bazie danych. CreateStatisticCommand posiada także klasę Validator, która odpowiada za walidację danych, w tym przypadku walidacja jest ustawiona

w następujący sposób :

- liczba goli musi być większa od zera,

- liczba asyst musi być większa od zera,

- liczba żółtych kartek musi być większa od zera,

- liczba czerwonych kartek musi być większa od zera.

W klasie Handler wstrzykujemy nasz IStatisticRepository wraz z IUnitOfWork, dzięki czemu będziemy mogli skorzystać z metody Add() oraz metody Complete(), aby dodać nowo utworzony rekord do bazy.

Ostatnią metodą w naszym kontrolerze jest zapytanie HTTP PUT, które obsługuje polecenie UpdateStatisticCommand. Parametrem wejściowym do tej metody jest UpdateStatisticInputModel, którym jest model danych potrzebnych do zaktualizowania statystyki. Modelem tym jest id statystyki oraz wartości nullable po kolei bramki, asysty, żółte oraz czerwone kartki, ponieważ nie jest powiedziane, że po każdym meczu wszystkie ze statystyk zostaną zaktualizowane.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.43 Klasa UpdateStatisticInputModel

Przejdźmy teraz do klasy, która obsługuje nam to polecenie.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.44 Klasa UpdateStatisticCommand

Konstruktor tej klasy przyjmuje nasz input model, który jest potrzebny do zaktualizowania konkretnej statystyki. Klasa Handler posiada obsługę dwóch wyjątków:

- przypadek, gdy statystyka o id równym id podanemu w parametrze wejściowym nie istnieje,

- przypadek, gdy piłkarz o id równym id podanemu w parametrze wejściowym nie istnieje.

W klasie Handler wstrzykujemy nasz IStatisticRepository wraz z IUnitOfWork, dzięki czemu będziemy mogli zaktualizować rekord oraz skorzystać z metody Complete(), aby dany rekord został zaktualizowany w bazie danych.

### 3.3.4 Teams

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.45 Klasa TeamsController

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.45 Klasa TeamsController

Kontroler TeamsController zawiera ścieżkę api/teams oraz pięć metod: GetTeam(), AddTeam(), HireCoach(), FireCoach() oraz SignPlayer().

Pierwsza z nich to zapytanie HTTP GET do bazy danych, który zwróci nam model TeamResultModel, czyli drużynę ze swoją nazwą, listą piłkarzy, imieniem i nazwiskiem trenera oraz numerem turnieju (tygodnia). Metoda ta jako parametr wejściowy przyjmuje Id drużyny.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.46 Klasa TeamResultModel

Przejdźmy teraz do klasy, która obsługuje dane zapytanie. Klasa ta zawiera dwie klasy. Pierwsza to TeamResult czyli model na jaki mają być mapowane dane oraz klasa Handler, która zawiera naszą całą logikę. W klasie Handler wstrzykujemy nasz FutStatDbContext, czyli naszą bazę, do której później się odniesiemy, aby wyciągnąć wszystkie potrzebne dane z bazy danych.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.47 Klasa GetTeamQuery

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.47 Klasa GetTeamQuery

Kolejną metodą w naszym kontrolerze jest zapytanie HTTP POST, które obsługuje polecenie CreateTeamCommand. Ta metoda przyjmuje parametr, którym jest CreateTeamInputModel, którym jest model danych potrzebnych do stworzenia rekordu drużyny w bazie danych. Modelem tym jest nazwa drużyny oraz numer turnieju (tygodnia).

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.48 Klasa CreateTeamInputModel

Przejdźmy teraz do klasy, która obsługuje nam to polecenie.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.49 Klasa CreateTeamCommand

Konstruktor tej klasy przyjmuje nasz input model, który jest potrzebny do stworzenia rekordu trenera w bazie danych. CreateTeamCommand posiada także klasę Validator, która odpowiada za walidację danych, w tym przypadku walidacja jest ustawiona w następujący sposób :

- Nazwa musi być przynajmniej trzy-literowe oraz nie przekraczać 50 znaków,

- Numer turnieju musi być większy od zera.

W klasie Handler wstrzykujemy nasz ITeamRepository wraz z IUnitOfWork, dzięki czemu będziemy mogli skorzystać z metody Add() oraz metody Complete(), aby dodać nowo utworzony rekord do bazy.

Kolejną, a w zasadzie kolejnymi metodami, ponieważ logika działania polecenia FireCoachCommand oraz HireCoachCommand jest podobna, w naszym kontrolerze jest zapytanie HTTP PUT, które obsługuje polecenie, weźmy tu jako przykład główny, HireCoachCommand. Parametrem wejściowym do tej metody jest UpdateOfCoachInputModel, którym jest model danych potrzebnych do przypisania trenera do drużyny. Modelem tym jest id drużyny, id trenera, imię oraz nazwisko trenera.

Obraz zawierający tekst, wewnątrz

Opis wygenerowany automatycznie

Rysunek 3.50 Klasa UpdateOfCoachInputModel

Przejdźmy teraz do klasy, która obsługuje nam to polecenie.

Obraz zawierający tekst, zrzut ekranu, ekran, srebrny

Opis wygenerowany automatycznie

Rysunek 3.51 Klasa HireCoachCommand

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.51 Klasa HireCoachCommand

Konstruktor tej klasy przyjmuje nasz input model, który jest potrzebny do zaktualizowania konkretnej drużyny o konkretnego trenera. Klasa Handler posiada obsługę dwóch wyjątków:

- przypadek, gdy drużyna o id równym podanemu w parametrze wejściowym nie istnieje,

- przypadek, gdy trener o id równym id podanemu w parametrze, o imieniu podanemu

w parametrze oraz nazwisku podanemu w parametrze wejściowym nie istnieje.

W klasie Handler wstrzykujemy nasz ITeamRepository, ICoachRepository wraz z IUnitOfWork, dzięki czemu będziemy mogli zaktualizować rekord oraz skorzystać z metody Complete(), aby dany rekordy zostały zaktualizowane w bazie danych.

Jak też już wcześniej wspomniałem podobnie działa polecenie FireCoachCommand,

z tą różnicą, że zamiast przypisywać do drużyny wartość trenera to przypisuję w to miejsce wartość null.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.52 Klasa FireCoachCommand

Obraz zawierający tekst, zrzut ekranu, monitor, srebrny

Opis wygenerowany automatycznie

Rysunek 3.52 Klasa FireCoachCommand

Ostatnią metodą w naszym kontrolerze jest zapytanie HTTP PUT, które obsługuje polecenie SignPlayerCommand. Parametrem wejściowym do tej metody jest UpdateOfPlayerInputModel, którym jest model danych potrzebnych do zaktualizowania statystyki. Modelem tym jest id drużyny, id piłkarza oraz imię i nazwisko piłkarza.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.53 Klasa UpdateOfPlayerInputModel

Przejdźmy teraz do klasy, która obsługuje nam to polecenie.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.54 Klasa SignPlayerCommand

Obraz zawierający tekst

Opis wygenerowany automatycznie

Rysunek 3.54 Klasa SignPlayerCommand

Konstruktor tej klasy przyjmuje nasz input model, który jest potrzebny do zaktualizowania konkretnej drużyny o piłkarza. Klasa Handler posiada obsługę dwóch wyjątków:

- przypadek, gdy drużyna o id równym id podanemu w parametrze wejściowym nie istnieje,

- przypadek, gdy piłkarz o id równym id podanemu w parametrze, imieniu oraz nazwisku równym imieniu i nazwisku podanemu w parametrze wejściowym nie istnieje.

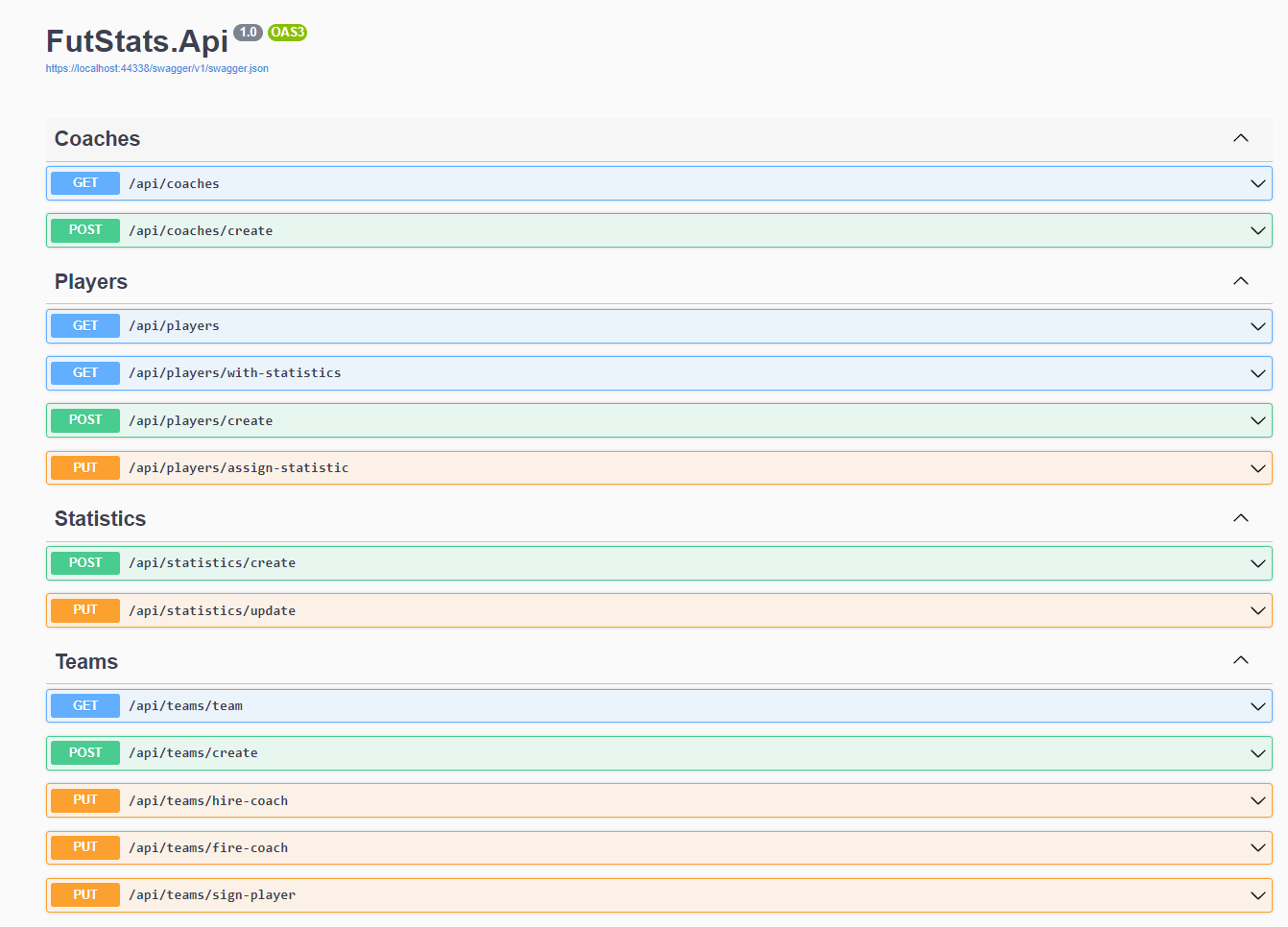
W klasie Handler wstrzykujemy nasz ITeamRepository, IPlayerRepository wraz

z IUnitOfWork, dzięki czemu będziemy mogli zaktualizować rekord oraz skorzystać z metody Complete(), aby dany rekord został zaktualizowany w bazie danych.

## **3.4 Działanie aplikacji**

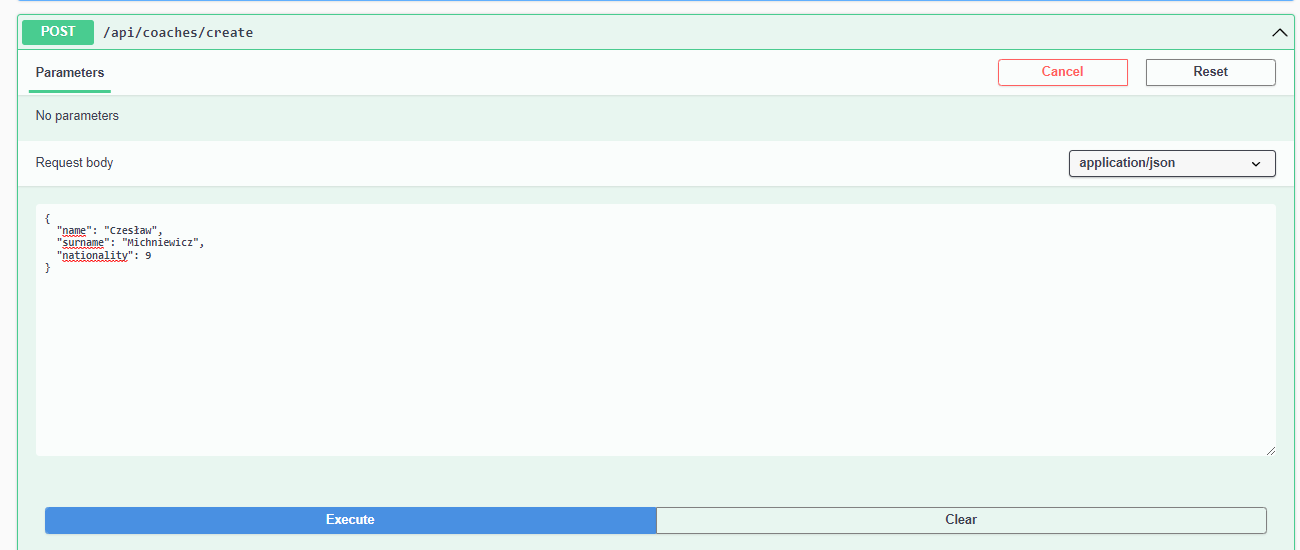
### 3.4.1 Działanie aplikacji z wykorzystaniem SwaggerUI

Gdy po raz pierwszy odpalimy naszą aplikację włączy nam się przeglądarka z wpisanym URL - **https://localhost:44338**. By włączyć interfejs swaggera należy do URL dopisać - **/swagger** i wtedy naszym oczom ukaże się interfejs umożliwiający korzystanie z naszych poleceń oraz zapytań.



Rysunek 3.55 Interfejs użytkownika SwaggerUi

Przejdźmy przez wszystkie możliwe akcją zaczynając od samej góry od trenera. Na początku spróbujmy wykonać akcję dodania rekordu do bazy. Klikamy zatem w zapytanie Post po czym w przycisk *Try it out* a ukaże nam się JSON, po którego poprawnym uzupełnieniu dodamy rekord trenera do bazy danych. Spróbujmy dodać trenera z Polski o imieniu Czesław oraz nazwisku Michniewicz.

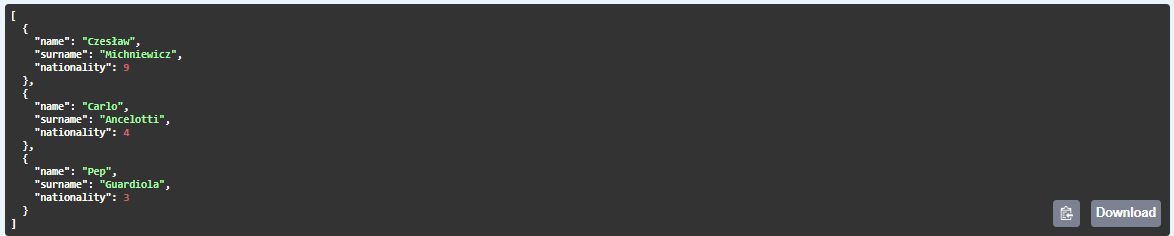


Rysunek 3.56 Dodanie trenera do bazy danych



Rysunek 3.57 Odpowiedź serwera

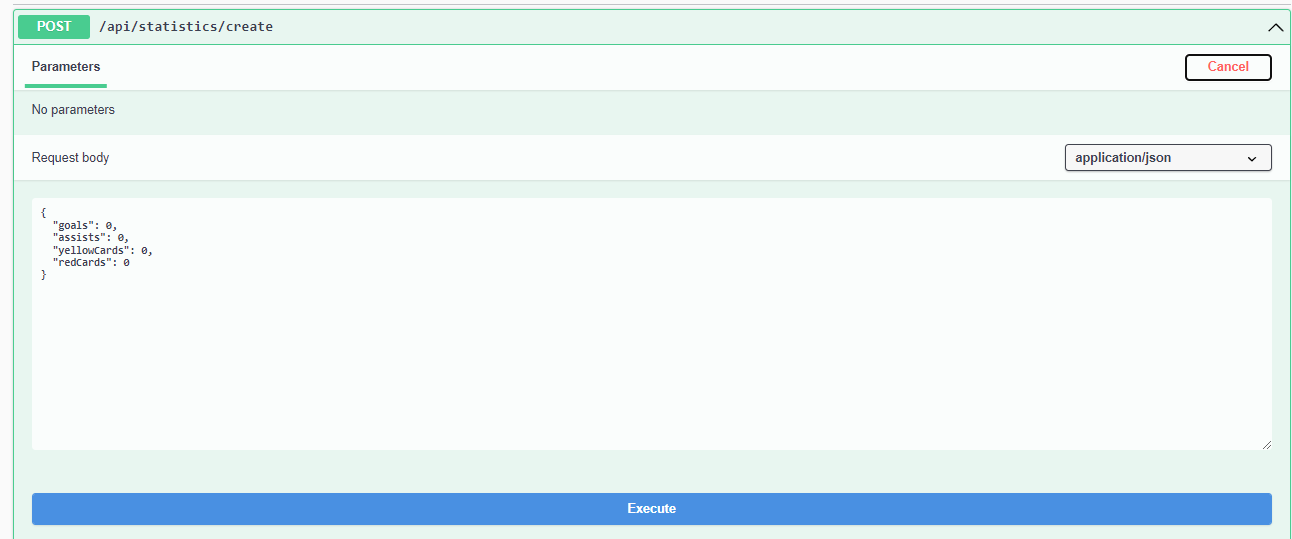
Jak widzimy po statusie – 200 Ok, nasz rekord trenera Czesława Michniewicza z Polski został dodany do bazy danych. Sprawdźmy teraz kolejnym przyciskiem akcyjnym GET, czy faktycznie taki rekord w bazie istnieje. Klikamy zatem w zapytanie GET po czym w przycisk *Try it out* a ukaże nam się lista wszystkich trenerów istniejących w bazie danych.



Rysunek 3.58 Lista trenerów istniejąca w bazie danych

Jak widzimy nasz dodany Czesław istnieje w bazie danych.

Przejdźmy teraz do statystyk. Na początku spróbujmy wykonać akcję dodania rekordu do bazy. Klikamy zatem w zapytanie Post po czym w przycisk Try it out a ukaże nam się JSON, po którego poprawnym uzupełnieniu dodamy rekord trenera do bazy danych. Spróbujmy dodać rekord, który ma wszystkie statystyki jako 0.

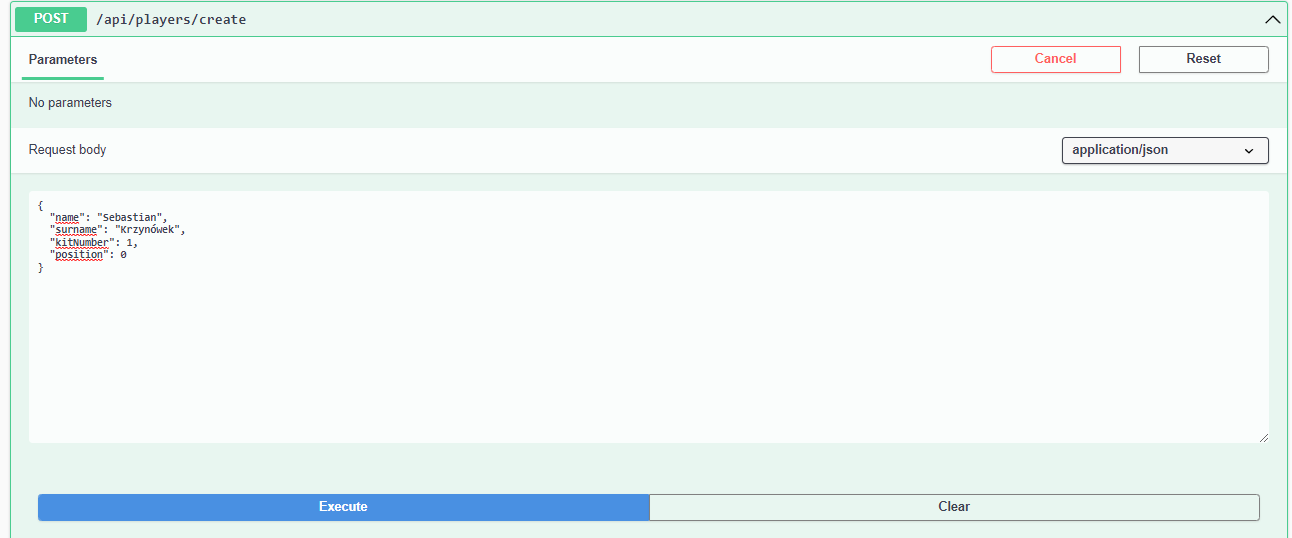


Rysunek 3.59 Dodawanie rekordu statystyki do bazy danych



Rysunek 3.60 Odpowiedź serwera

Zanim przejdziemy do zaktualizowania naszej stworzonej statystyki zahaczmy teraz o akcje piłkarzy. Spróbujmy stworzyć piłkarza o imieniu Sebastian, nazwisku Krzynówek, numerze koszulki 1 oraz pozycji na boisku – bramkarz.



Rysunek 3.61 Dodawanie rekordu piłkarza do bazy danych



Rysunek 3.62 Odpowiedź serwera

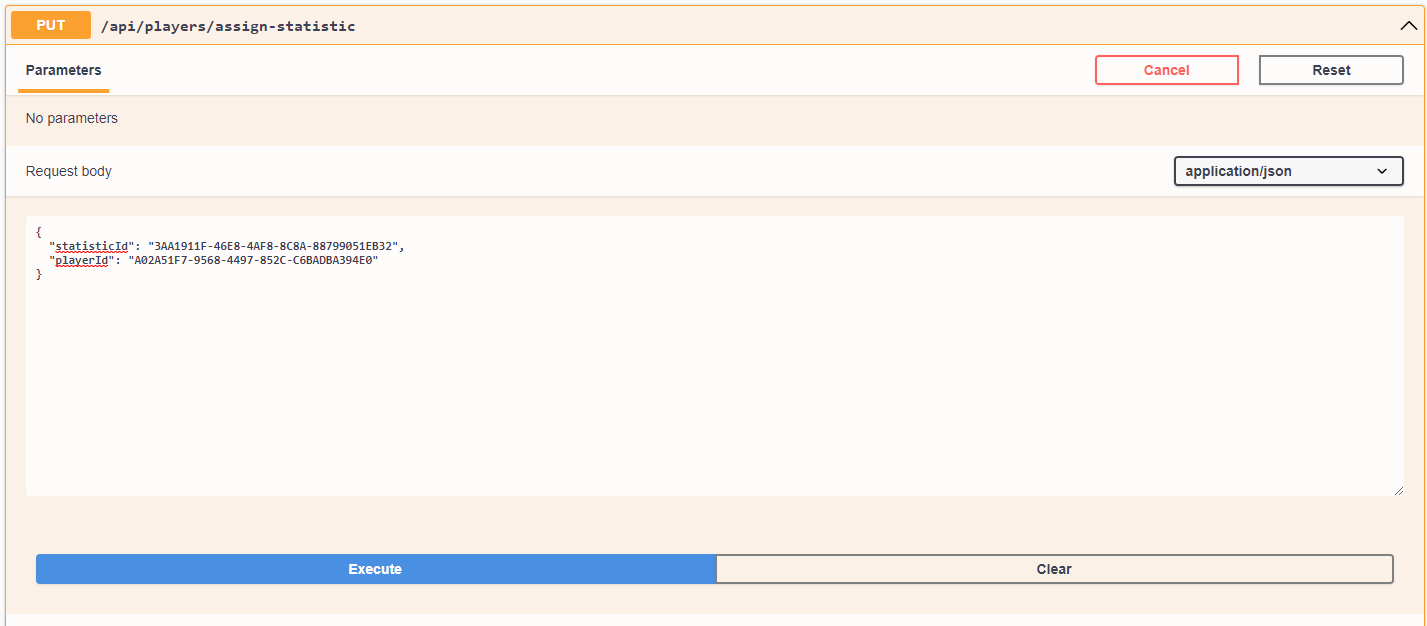
Jak widzimy po statusie 200 OK, nasz rekord piłkarza został dodany do bazy, spróbujmy teraz sprawdzić przycisk akcyjny GET, który zwraca wszystkich istniejących w bazie piłkarzy i sprawdźmy, czy piłkarz Sebastian Krzynówek istnieje w bazie danych.

Obraz zawierający tekst

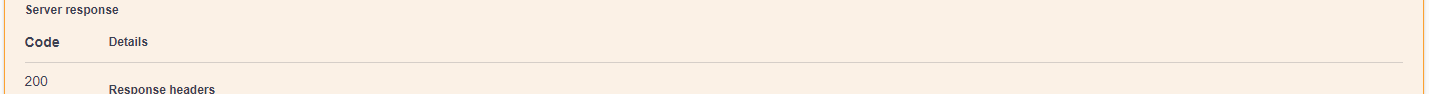
Opis wygenerowany automatycznie

Rysunek 3.63 Piłkarz istniejący w bazie danych

Spróbujmy teraz przypisać wcześniej stworzone statystyki do naszego piłkarza Sebastiana Krzynówek. Przechodzimy do przycisku akcyjnego PUT, klikamy przycisk *Try it out* a ukaże nam się JSON, po którego poprawnym uzupełnieniu przypiszemy statystyki do naszego piłkarza.

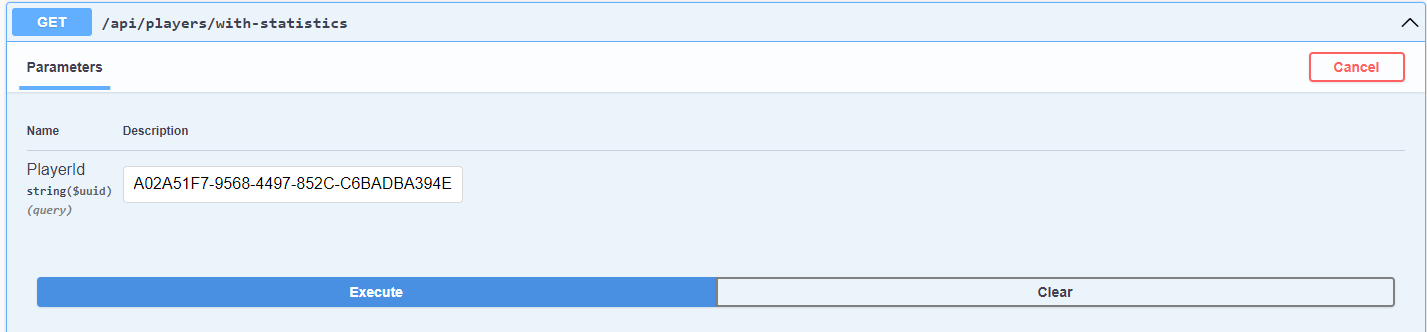


Rysunek 3.64 Przypisanie statystyk do piłkarza

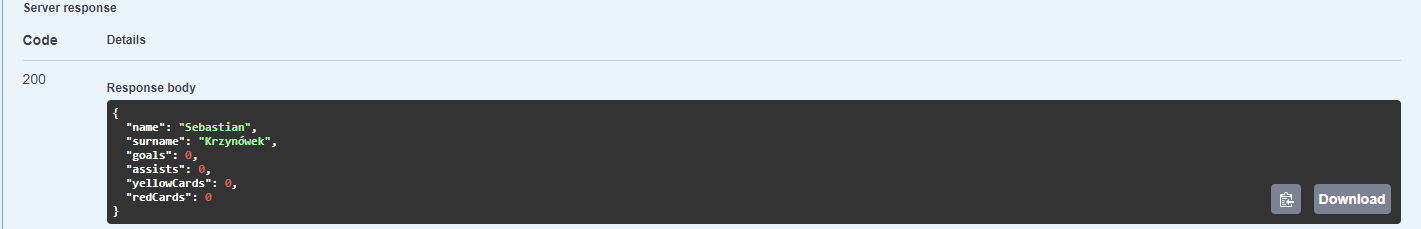


Rysunek 3.65 Odpowiedź serwera

Po odpowiedzi serwera wnioskujemy, że statystyki zostały przypisane do naszego piłkarza. Sprawdźmy to. Przechodzimy do drugiego przycisku akcyjnego GET, klikamy przycisk *Try it out* oraz podajemy Id piłkarza, którego chcemy aby zapytanie nam zwróciło.

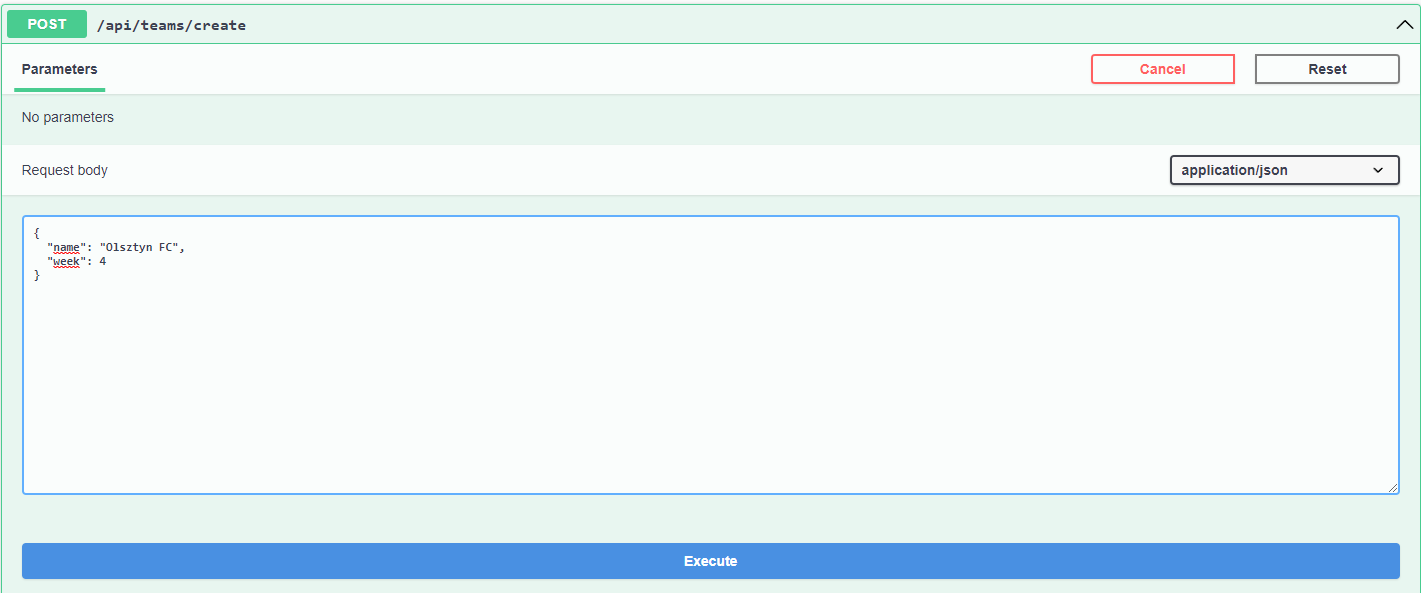


Rysunek 3.66 Wyszukanie konkretnego piłkarza wraz ze statystykami



Rysunek 3.67 Odpowiedź serwera

Przejdźmy teraz do przycisków akcyjnych związanych z drużyną i rozpocznijmy od stworzenia drużyny Olsztyn FC, która bierze udział w turnieju nr 4.



Rysunek 3.68 Dodawanie rekordu drużyny do bazy danych



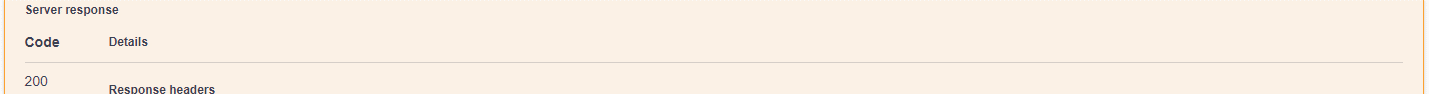
Rysunek 3.69 Odpowiedź serwera

Po odpowiedzi serwera wnioskujemy, że drużyna została dodana do bazy danych. Spróbujmy zatem zakontraktować trenera Czesława Michniewicza do naszej drużyny. Przechodzimy do przycisku akcyjnego PUT o nazwie hire-coach oraz po naciśnięciu przycisku *Try it out* uzupełniamy JSON’a o potrzebne dane.

Obraz zawierający stół

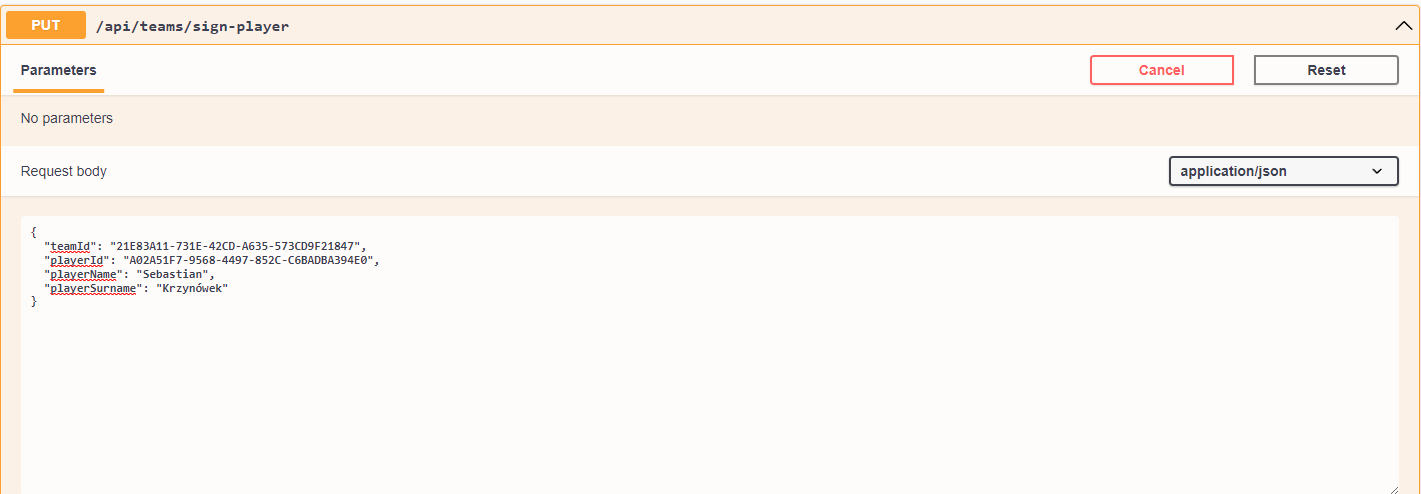
Opis wygenerowany automatycznie

Rysunek 3.70 Przypisanie trenera do drużyny

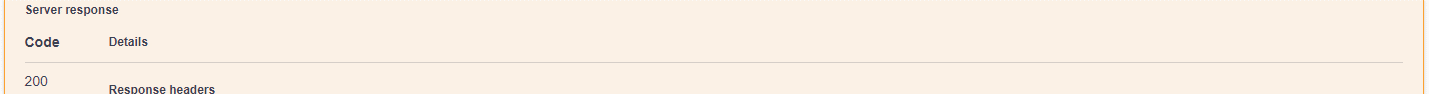


Rysunek 3.71 Odpowiedź serwera

Z odpowiedzi serwera wiemy, że nasza akcja została wykonana poprawnie. Spróbujmy teraz zakontraktować do naszej drużyny piłkarza, którego stworzyliśmy wcześniej, czyli Sebastiana Krzynówka.

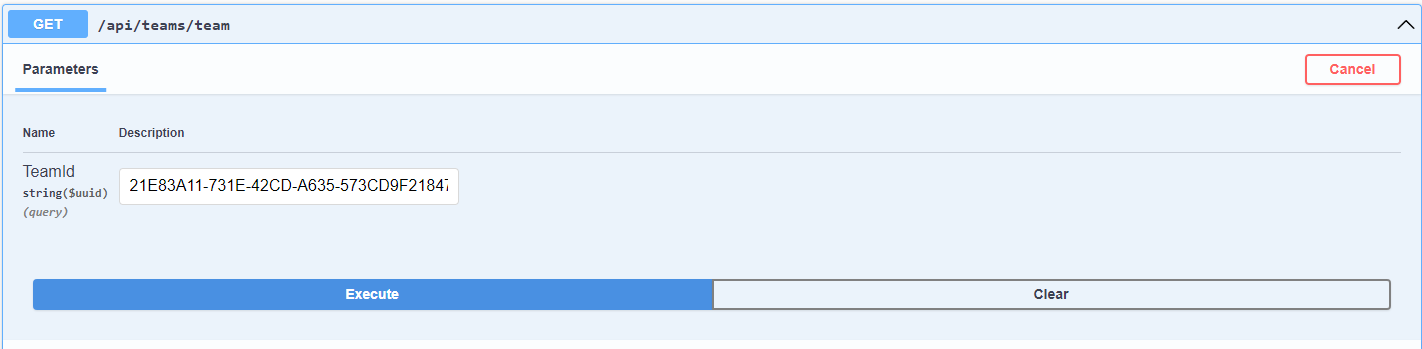


Rysunek 3.72 Przypisanie piłkarza do drużyny

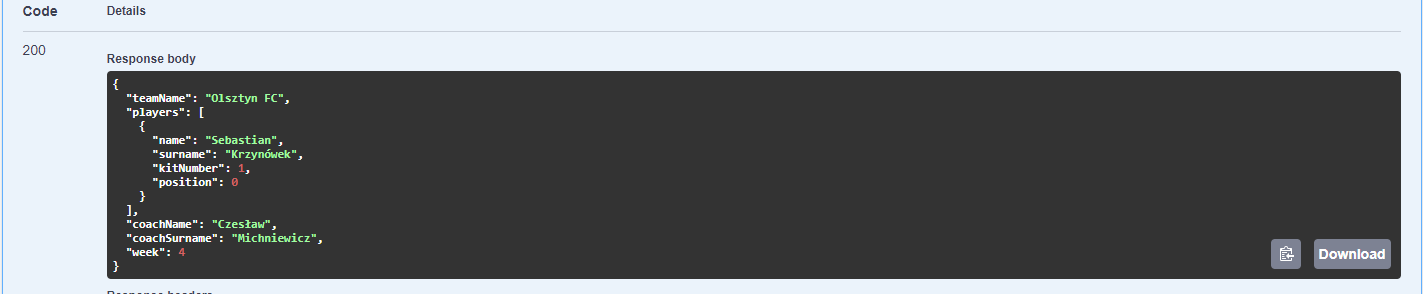


Rysunek 3.73 Odpowiedź serwera

Po odpowiedzi serwera widać, że po raz kolejny powiodło nam się wykonanie tej akcji. Spróbujmy w takim razie pobrać z bazy danych informacje o naszej dotychczasowej drużynie Olsztyn FC. Przechodzimy w takim razie do przycisku akcyjnego GET oraz po naciśnięciu przycisku *Try it out* podajemy w podane miejsce wymagane Id drużyny.



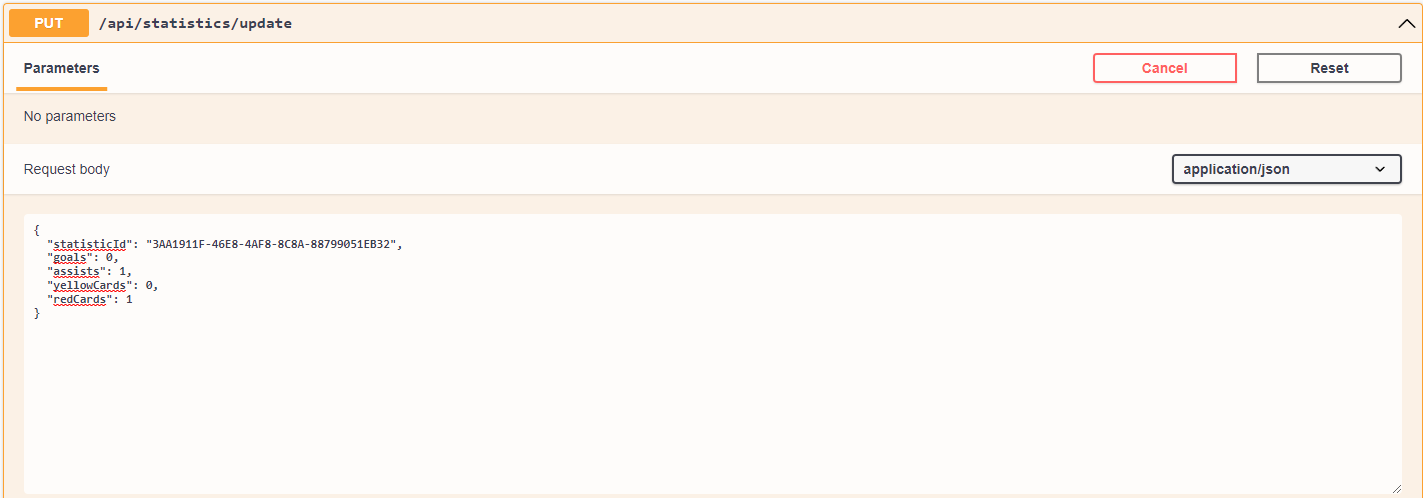
Rysunek 3.74 Pobieranie informacji odnośnie drużyny Olsztyn Fc



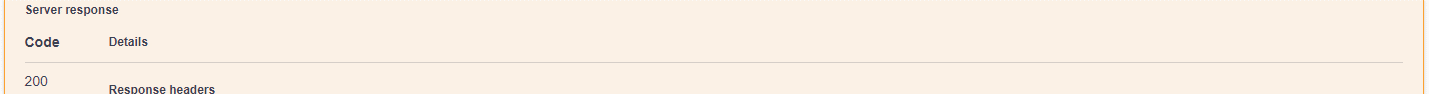
Rysunek 3.75 Odpowiedź serwera

Jak widzimy serwer zwrócił nam odpowiedź na nasze zapytanie, którym jest nazwa drużyny, lista graczy, w których na razie jest tylko Sebastian Krzynówek, imie i nazwisko trenera, czyli Czesława Michniewicza oraz numer turnieju (tygodnia) drużyny.

Wy obraźmy sobie sytuacje, w której po pierwszym meczu w nowej drużynie Sebastian zaliczył jako bramkarz asystę, jednakże dostał też czerwoną kartkę. Chcemy zaktualizować statystyki naszego piłkarza, w takim razie przechodzimy do przycisku akcyjnego PUT, wciskamy przycisk *Try it out*  oraz podajemy odpowiednie wartości w naszym wejściowym JSON’ie.

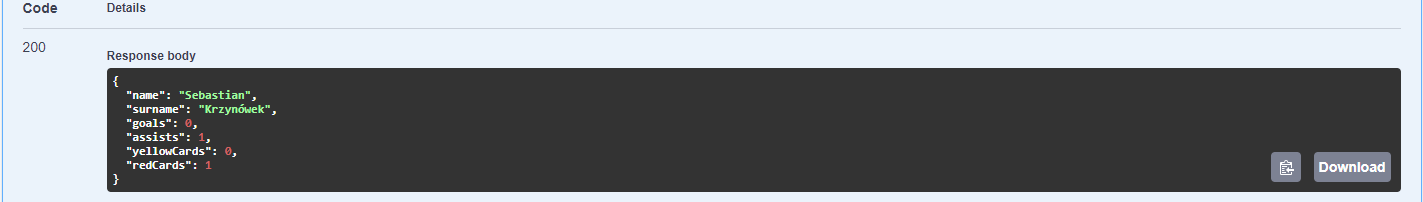


Rysunek 3.76 Aktualizowanie statystyk piłkarza



Rysunek 3.77 Odpowiedź serwera

Status pokazuje nam, że akcja powinna wykonać się bez żadnych błędów. Sprawdźmy teraz, czy statystyki naszego piłkarza faktycznie zostały zaktualizowane poprzez nasz przycisk akcyjny GET.



Rysunek 3.78 Odpowiedź serwera

Faktycznie, z odpowiedzi serwera widać jasno, że statystyki naszego piłkarza zostały zaktualizowane.

# **Podsumowanie**

Praca zawiera niezbędne informacje z dziedziny informatyki. Dostarcza podstawowych informacji o programowaniu w języku programowania C# jak też i odnośnie ASP .NET Core. Rozszerza wiedzę programistyczną o bardzo potrzebne do poznania wzorce Clean Architecture, CQRS oraz Mediator. Praca pokazuje też jak powinna wyglądać aplikacja webowa. W rozdziale poświęconym aplikacji przedstawione zostało jak krok po kroku można zbudować taką aplikację webową. Najważniejszym, przy rozpoczynaniu pracy przy takim projekcie, jest znalezienie odpowiednie go wzorca architektury aplikacji, późniejsze go zaimplementowanie oraz utrzymywanie. Po wykonaniu tego zadania pomysł na aplikacje oraz na poszczególne polecania wraz z zapytaniami rośnie w miarę jedzenia, a raczej w miarę zagłębiania się w pisanie kodu. W efekcie pracy powstała kompletna aplikacja webowa, która jest w 100% przygotowana na ewentualne rozszerzenia kodu lub rozszerzenie w postaci podpięcia framework’u fronted’owego co uczyni z naszej aplikacji webowej pełnoprawną aplikację z interfejsem graficznym.

# **Bibliografia**

1. <https://pl.wikipedia.org/wiki/J%C4%99zyk_programowania>
2. <https://progromcy.pl/zastosowania-jezykow-programowania/>
3. <https://nexttechnology.io/top-10-programming-languages-for-2022/>
4. <https://docs.microsoft.com/pl-pl/dotnet/csharp/tour-of-csharp/>
5. <https://programujzklasa.pl/co-to-ide/>
6. <https://visualstudio.microsoft.com/pl/vs/>
7. <https://www.guru99.com/best-csharp-ide.html>
8. <https://docs.microsoft.com/pl-pl/dotnet/csharp/language-reference/language-specification/types>
9. <https://www.plukasiewicz.net/CSharp_dla_poczatkujacych/Typy_danych>
10. <https://codecool.com/pl/blog/czym-jest-programowanie-obiektowe/>
11. <https://pl.wikipedia.org/wiki/Programowanie_obiektowe>
12. <https://docs.microsoft.com/pl-pl/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-6.0>
13. <https://www.plukasiewicz.net/Artykuly/AspNetCoreFeatures>
14. <https://www.tutorialsteacher.com/webapi/what-is-web-api>
15. <https://cezarywalenciuk.pl/blog/programing/aspnet-web-api-podstawy-prostego-api-i-co-na-to-wujek-wcf>
16. <https://docs.microsoft.com/pl-pl/aspnet/web-api/overview/getting-started-with-aspnet-web-api/tutorial-your-first-web-api>
17. <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>
18. <https://www.jcommerce.pl/jpro/artykuly/clean-architecture>
19. <https://code-maze.com/cqrs-mediatr-in-aspnet-core/>
20. <https://stackify.com/solid-design-principles/>
21. <https://cezarywalenciuk.pl/blog/programing/mediatr-cqrs-i-wzorzec-projektowy-mediator-w-aspnet-core>