WYŻSZA SZKOŁA BANKOWA W GDAŃSKU

WYDZIAŁ FINANSÓW I ZARZĄDZANIA

Dmytro Vozniachuk

nr albumu 37227

**APLIKACJA INTERNETOWA „FRIENDY”**

Praca inżynierska

na kierunku Informatyka

Praca napisana pod kierunkiem

dr. Dariusza Kralewskiego

Gdańsk 2020

# Spis treści

[Spis treści 1](#_Toc29673460)

[Wstęp 2](#_Toc29673461)

[1. Aplikacje internetowe 4](#_Toc29673462)

[1.1. Historia aplikacji internetowych 4](#_Toc29673463)

[1.2. Podejścia do tworzenia aplikacji internetowych 6](#_Toc29673464)

[1.3. Architektura aplikacji internetowych 8](#_Toc29673465)

[2. Projektowanie 11](#_Toc29673466)

[2.1. Wykorzystane technologie programistyczne 11](#_Toc29673467)

[2.1.1. Angular – technologia do tworzenia SPA 11](#_Toc29673468)

[2.1.2. Tworzenie backend’u w ASP.NET Core 15](#_Toc29673469)

[2.1.3. Systemy zarządzania bazami danych 16](#_Toc29673470)

[2.1.4. System wyszukiwania Elasticsearch 18](#_Toc29673471)

[2.2. Planowanie aplikacji 19](#_Toc29673472)

[2.2.1. Wymagania pozafunkcjonalne 19](#_Toc29673473)

[2.2.2. Wymagania funkcjonalne 20](#_Toc29673474)

[2.3. Architektura systemu 23](#_Toc29673475)

[2.3.1. REST API 24](#_Toc29673476)

[2.3.2 Bezpieczeństwo aplikacji 25](#_Toc29673477)

[2.3.3. Uwierzytelnianie i autentykacja 27](#_Toc29673478)

[3.1. Projektowanie i wykorzystanie bazy danych w ASP.NET Core 30](#_Toc29673479)

[3.2. Projektowanie i tworzenie REST API 37](#_Toc29673480)

[3.3. Proces uwierzytelniania i autentykacji użytkowników 45](#_Toc29673481)

[3.4. Routing w aplikacji 48](#_Toc29673482)

[3.5. Implementacja przykładowej podstrony 49](#_Toc29673483)

[3.6. Wyszukiwanie użytkowników 50](#_Toc29673484)

[3.7. Zarządzanie stanem aplikacji 53](#_Toc29673485)

[3.8. Elementy aplikacji czasu rzeczywistego 55](#_Toc29673486)

[3.9. Implementacja algorytmu dopasowania znajomych 60](#_Toc29673487)

[3.10. Przedstawienie podstron 61](#_Toc29673488)

[Zakończenie 66](#_Toc29673489)

[Bibliografia 67](#_Toc29673490)

[Spis tabel 68](#_Toc29673491)

[Spis ilustracji 69](#_Toc29673492)

### Wstęp

Ludzie z natury są istotami społecznymi. Ten fakt był głównym powodem tego, iż Facebook jako bezprecedensowy projekt wyrósł na giganta w bardzo krótkim czasie. Mimo że na rynku internetowym istnieje ogrom serwisów zrzeszających ogromne ilości użytkowników jest bardzo mało rozwiązań względnie małych oraz prostych w użytkowaniu, także lokalnych, służących do nawiązywania faktycznych znajomości, a nie wirtualnych. Człowiek odwiecznie dąży do tworzenia sieci znajomości, do tworzenia grup. Grup, które łączą interakcje polegające na wspólnych zainteresowaniach, dążeniach. Większość istniejących rozwiązań mijają się z tym celem. Właśnie ten cel powinna spełnić aplikacja „Friendy”.

Celem niniejszej pracy jest utworzenie aplikacji internetowej – sieci społecznościowej, której głównym zadaniem będzie umożliwianie użytkownikom poznawanie nowych osób. Użytkownicy aplikacji będą mieli możliwość brać udział w różnych wydarzeniach tematycznie celowanych na poznawanie nowych osób, będą mogli komunikować się z innymi użytkownikami poprzez czat oparty o zastosowanie elementów aplikacji czasu rzeczywistego z użyciem biblioteki SignalR. Oprócz implementacji wybranego tematu celem pracy jest wybór i analiza różnych podejść do projektowania systemów informatycznych, na przykład wzorce projektowe „Repozytorium i serwisy” i CQRS. Oprócz części teoretycznej w trakcie pracy będą przedstawiane elementy praktyczne implementacji aplikacji, na przykład implementacja wcześniej wspomnianych wzorców projektowych, elementów czatu, diagramy UML i ER. Do implementacji różnych części aplikacji zostaną zastosowane różne dobrze zaadaptowane technologie programistyczne: backend tworzonego rozwiązania będzie oparty o ASP.NET Core, bazę danych Microsoft SQL Server oraz klaster rozproszonej wyszukiwarki Elasticsearch, frontend z kolei powstanie z użyciem technologii do tworzenia aplikacji jednostronicowych o nazwie Angular.

Pierwszy rozdział zostanie poświęcony części teoretycznej związanej z całokształtem aplikacji internetowych. Dla początku zostanie przedstawiona historia aplikacji internetowych od lat dziewięćdziesiątych aż do tego czasu, w szczególności zostaną omówione najważniejsze wydarzenia z historii aplikacji internetowych. Następnie zostaną przedstawione podejścia do tworzenia aplikacji internetowych, w tym podejścia MPA (od ang. Multi Page Application), SPA (od ang. Single Page Application) oraz wprowadzenie do wzorców architektonicznych. Z kolei w ostatnim podrozdziale zostaną opisane wspomniane wzorce architektoniczne jako jedne z głównych elementów architektury aplikacji internetowych.

W drugim rozdziale zostanie przedstawiona część związana z projektowaniem tworzonej aplikacji. Początek rozdziału zostanie poświęcony zastosowanym technologiom programistycznym, w tym zostanie przedstawione wprowadzenie do architektury każdej z nich. Kolejna część rozdziału będzie składała się z etapu pozyskiwania wymagań i przedstawienia przypadków użycia. Ponadto znowuż zostanie przedstawiona architektura tworzonego rozwiązania i aspekt bezpieczeństwa, w tym idea JWT (od ang. *JSON Web Token*), ogólny schemat uwierzytelniania i autentykacji oraz typowe ataki przed którymi należy zabezpieczać system informatyczny.

Kolejny i ostatni rozdział zostanie zadedykowany bezpośredniej implementacji aplikacji. Rozdział rozpocznie się od projektowania bazy danych, w tym zostaną opisane wszystkie tabele bazy danych, przedstawiony diagram ER oraz opisane pojęcie *scaffolding* w kontekście narzędzia Entity Framework. Następnym etapem będzie projektowanie i tworzenie REST API – na tym etapie zostaną przedstawione implementacje wzorców repozytoriów, serwisów i CQRS, a także zostanie przedstawiona implementacja uwierzytelniania i autentykacji w oparciu o JWT. Po czym zostaną przedstawione podstawowe fragmenty frontend’u aplikacji, w tym pojęcie i implementacja routing’u, podstrona profilu użytkownika, użycie Elasticsearch do wyszukiwania użytkowników według kryteriów, implementacja czatu z użyciem biblioteki SignalR oraz zarządzanie stanem aplikacji z użyciem NgRx. Poza tym, tuż przed końcem rozdziału zostanie przedstawiona implementacja algorytmu *Cosine Similarity* w kontekście dopasowania prawdopodobnych znajomych. Na końcu rozdziału zaś zostaną ukazane podstrony frontend’u w ramach przedstawienia wyników pracy.

Podczas opracowywania projektu głównie wykorzystywane były materiały anglojęzyczne, w tym preferowanymi materiałami były artykuły internetowe i blogi. Z kolei podczas opracowywania teoretycznej części pracy, na przykład określania wymagań lub przygotowaniu informacji o dobrych praktykach projektowania REST API, głównym źródłem wiedzy były informacje zaczerpnięte z książek i dokumentów naukowych. Jednak podczas implementacji aplikacji za główne źródło wiedzy posłużyła oficjalna dokumentacja użytych technologii programistycznych.

# 1. Aplikacje internetowe

Aplikacje internetowe to rodzaj wieloplatformowego programu komputerowego opartego o architekturę klient-serwer działającego w przeglądarce[[1]](#footnote-1). W ramach tworzenia aplikacji internetowej nie może zabraknąć historii rozwoju tej wręcz branży. Właśnie w tym rozdziale zostanie przedstawione krótkie wprowadzenie do historii i całości idei aplikacji internetowych.

# Historia aplikacji internetowych

Początki aplikacji internetowych znalazły swe korzenie w latach 90 ubiegłego wieku. Pierwszym zalążkiem interaktywnych aplikacji internetowych były statyczne strony internetowe utworzone z wykorzystaniem HTML w postaci dokumentów tekstowych dostarczające minimalistyczne efekty interaktywne wymagające ciągłego pobierania z serwera widoku przy zmianie danych wejściowych. Jednak rok 1995 zmienił wszystko, ponieważ firma Netscape Communications przedstawiła Javascript – język skryptowy zadaniem którego było urozmaicenie interakcji między użytkownikiem a przeglądarką poprzez umożliwienie tworzenia dynamicznych komponentów interfejsu użytkownika. Korzystanie z języka także usprawniło szybkość działania aplikacji, ponieważ znikała potrzeba pobierania widoku z serwera przy wprowadzaniu nowych danych. W krótkim czasie Javascript wraz z HTML i CSS stał się najczęściej używaną technologią do tworzenia aplikacji internetowych.[[2]](#footnote-2)

W roku 1996 firma Macromedia[[3]](#footnote-3) wydała pierwszą wersję platformy Adobe Flash wykorzystywanej do tworzenia animacji, interaktywnych stron internetowych oraz różnych typów aplikacji. W kontekście internetu, Flash był bardzo popularnym narzędziem w latach 2000, ponieważ wykorzystywano go praktycznie na każdej interaktywnej stronie internetowej, grze online, filmie lub audio. Obecnie Flash jest technologią odchodzącą w zapomnienie, a programiści aplikacji preferują użycie HTML5, dlatego też na końcu 2020 roku Flash ma zakończyć swój żywot.[[4]](#footnote-4)

Następnymi ważnymi etapami historii rozwoju aplikacji internetowych były lata 1999 i 2005. W 1999 po wydaniu nowej wersji serwletów Java po raz pierwszy do tego języka zostało wprowadzone pojęcie aplikacji internetowej lub *web app*, zaś w 2005 roku światu okazała się technika AJAX (od ang. *Asynchronous JavaScript + XML*), wykorzystywana do tworzenia asynchronicznych aplikacji internetowych, czyli takich, które nie wymagają ciągłego pobierania widoków z serwera po żądaniu lub wprowadzaniu nowych danych.[[5]](#footnote-5)

Mimo że już w latach 2000 programiści mogli tworzyć dość ciekawe rozwiązania korzystając z istniejących technologii programistycznych, niestety proces ten nie należał do najłatwiejszych. W czerwcu 2004 roku firmy The Mozilla Foundation[[6]](#footnote-6) i Opera Software[[7]](#footnote-7) opracowały i przedstawiły organizacji W3C[[8]](#footnote-8) dokument dotyczący aplikacji internetowych jako całości, na przykład poruszone w nim zostały tematy dotyczące tworzenia, technologii programistycznych lub informacje o kompatybilności wstecznej. Także w tym dokumencie został przedstawiony projekt specyfikacji standardu Web Forms jako kierunku w którym należy rozwijać aplikacje internetowe. Dokument ten jednak nie cieszył się zbytnim poparciem przez organizację W3C, która stawiała na rozwój rozwiązań XML. Z czasem jednak została powołana organizacja WHATWG (od ang. *Web Hypertext Application Technology Working Group*)[[9]](#footnote-9) zrzeszająca programistów z firm Opera, Apple i Mozilla, która zaproponowała całościowy standard aplikacji internetowych, który z czasem został połączony ze standardem Web Forms, a nazwę całości zmieniono w HTML5[[10]](#footnote-10). Już w 2008 roku mimo nieukończenia prac nad standardem niektóre funkcjonalności HTML5 zostały zaimplementowane do przeglądarek. Po czym w 2009 roku organizacja W3C postawiła na rozwój HTML5 zamiast XHTML 2.0. W związku z tym od tamtego czasu organizacja W3C rozwijała standard HTML5 z WHATWG, a 28 października 2014 roku HTML5 osiągnął status rekomendacji przez W3C.[[11]](#footnote-11)

Ostatni, ale nie mniej ważny etap rozwoju aplikacji internetowych przypadł na 2015 rok, kiedy to inżynier pracujący nad przeglądarką Google Chrome Alex Russel opisał progresywne aplikacje internetowe (PWA, od ang. *Progressive Web Applications*) w kontekście funkcjonalności wspieranych przez nowe wersje przeglądarek, na przykład *service workers* i *web app manifests*. Progresywne aplikacje internetowe to typ aplikacji internetowej utworzony z użyciem tradycyjnych technologii HTML, CSS i JavaScript, który poprzez zasady działania przypomina natywne aplikacje mobilne, ponieważ aplikacje te mogą działać bez połączenia z siecią internet, obsługiwać powiadomienia i otrzymywać dostęp do urządzeń w smartfonie. Po wspomnianym wcześniej wystąpieniu przez Alex’a Russel’a firma Google zaczęła aktywnie promować tworzenie aplikacji PWA dla systemu operacyjnego Android, a tuż w 2017 roku do przeglądarki Safari zostało wprowadzone wsparcie *service workers* – oznaczało to, że dwa najpopularniejsze systemy operacyjne wspierały PWA[[12]](#footnote-12). Przykładową progresywną aplikacją internetową jest Twitter Lite[[13]](#footnote-13) – ta aplikacja jest przeznaczona przede wszystkim dla użytkowników z wolnym połączeniem internetowym, a rozmiar aplikacji nie przekracza 1 MB.

# 1.2. Podejścia do tworzenia aplikacji internetowych

Charakterystykę podejść do tworzenia aplikacji internetowych warto zacząć od ziarenka, które przyczyniło się do powstania aplikacji internetowych, czyli aplikacji natywnych. Aplikacja natywna to rodzaj programu komputerowego instalowanego na jednym systemie operacyjnym, którego główną jak wadą tak i zaletą jest dostęp do zasobów komputerowych użytkownika oraz zależność od jednego systemu operacyjnego, co sprawia, że najczęściej aplikacja natywna ma być wydajniejsza od rozwiązań webowych. Właśnie po to aby w niektórych przypadkach likwidować lub minimalizować wpływ ograniczeń wywieranych przez aplikacje natywne częściej zaczęto tworzyć aplikacje internetowe.

Obecnie istnieje kilka podejść do tworzenia aplikacji internetowych. Jednym z nich są aplikacje wielostronicowe albo tak zwane MPA (od ang. *Multi Page Application*) – jest to tak zwany konwencjonalny typ aplikacji internetowych działający na zasadzie architektury *server-centeric*. Tradycyjne działanie MPA polega na wysyłaniu żądań o widoki HTML do serwera w celu dopełnienia strony o nowe dane. Właśnie to jest jedna z główniejszych wad aplikacji wielostronicowych, ponieważ nawet zatwierdzanie formy, oprócz wysłania zapytania HTTP, skutkuje odświeżeniem strony oraz ponownym pobraniem widoku z serwera. Poza tym tutaj w grę wchodzi problematyczne zarządzanie stanem aplikacji.

Zatem w celu zminimalizowania ograniczeń wynikających z aplikacji wielostronicowych postawiono na opracowanie typu aplikacji jednostronicowych lub SPA (od ang. *Single Page Application*). Aplikacje jednostronicowe, jak wskazuje nazwa, składają się z wyłącznie jednej strony, która w zasadzie jest zbiorem powiązanych komponentów. Jedną z głównych zalet SPA w porównaniu do MPA jest większy poziom interaktywności rozwiązania, ponieważ w tym wypadku serwer służy jako API lub brama dostępowa do danych, a żadne z wysłanych zapytań HTTP do API w kontekście zarządzania danymi nie wymaga przeładowania strony ponieważ te zapytania są oparte o wcześniej przedstawiony Ajax (choć Ajax może być wykorzystywany w aplikacjach MPA, na przykład używając biblioteki jQuery[[14]](#footnote-14), natomiast nawet takie podejście jest dość trudne implementacyjnie). Inną ważną zaletą jest możliwość zarządzania stanem w pamięci, ponieważ tutaj nie ma powinności ciągłego odświeżenia strony. Najczęściej w kontekście zarządzania stanem korzysta się z implementacji wzorca projektowego o nazwie *flux*, który zostanie przedstawiony później. Jednak należy podkreślić, że w wypadku obu typów aplikacji częściowe przechowywanie stanu jest możliwe w zdefiniowanym w HTML5 magazynie danych przeglądarki o nazwie *local storage*. Podsumowując, fundament każdej aplikacji jednostronicowej składa się z następujących elementów[[15]](#footnote-15):

1. Biblioteka lub framework Javascript, których typowo używa się do zbudowania aplikacji jednostronicowej
2. Routing, który definiuje nawigowanie się pomiędzy widokami w SPA, ponieważ wszystkie widoki lub komponenty egzystują w obrębie jednego elementu HTML, a natywnie zaimplementowany w bibliotekach lub frameworkach router jest przeznaczony do tego celu. Na przykład, w aplikacjach utworzonych z użyciem framework’a Angular komponenty należące do jednego modułu routing’u są tworzone w obrębie znacznika <app-route>…</app-route>
3. *Template engines*, które są używane do wyświetlania lub ponownego wyświetlania komponentów
4. HTML5, który jest używany do tworzenia komponentów aplikacji jednostronicowej
5. Backend i REST API, które są częścią serwerową konsumowaną przez aplikację typowo w celu pobierania danych z magazynów serwerowych, na przykład baz danych
6. Ajax, ponieważ wszystkie interakcje zachodzące pomiędzy serwerem a aplikacją są żądaniami asynchronicznymi, a dane pobrane z serwera są wyświetlane w widokach dopiero po przybyciu bez wpływu na inne komponenty

Każde z podejść ma pewne wady i zalety. Mimo że aplikacje wielostronicowe nie zagwarantują poziomu interaktywności aplikacji jednostronicowych należy brać pod uwagę, że tworzenie aplikacji wielostronicowych nie wymaga umiejętności posługiwania się oddzielnymi technologiami do tworzenia widoków, czyli cała aplikacja może zostać utworzona korzystając z wyłącznie jednego framework’a serwerowego. Typowo rozwiązania MPA są oparte o wzorzec architektoniczny MVC (od ang. *Model-View-Controller*), głównym zadaniem którego jest separacja modelu i widoku poprzez kontroler. Natomiast takie rozwiązania jak Angular lub React.JS podążają ścieżką rozwiązań opartych o komponenty, chociaż same z siebie zapożyczają dużo koncepcji ze wzorca MVVM (od ang. *Model-View-ViewModel*), który jest następcą wzorca MVC i MVP (od ang. *Model-View-Presenter*) wykorzystywanego do budowania interfejsów użytkownika[[16]](#footnote-16), składającego się z tegoż modelu i widoku zawartych we wzorcu MVC, jednak do tych dwóch elementów został dołączony dodatkowy składnik o nazwie ViewModel, który często porównywany jest do warstwy prezentera wzorca MVP.

# 1.3. Architektura aplikacji internetowych

Pierwszym wspomnianym wzorcem architektonicznym był *Model-View-Controller* przedstawiony przez norweskiego programistę Trygve Reenskaug w 1979 roku, który pracował nad językiem Smalltalk-79[[17]](#footnote-17) w Xerox Palo Alto Research Center (PARC)[[18]](#footnote-18). Po czym w latach 80 ubiegłego wieku wzorzec MVC został zaimplementowany jako część biblioteki klas w Smalltalk-80. Jednak dopiero w 1988 roku w magazynie The Journal of Object Technology (JOT)[[19]](#footnote-19) ten wzorzec został przedstawiony jako ogólna koncepcja[[20]](#footnote-20). Obecnie wzorzec MVC jest możliwy do wykorzystania w większości popularnych rozwiązań webowych, na przykład ASP.NET MVC[[21]](#footnote-21) lub Spring Web MVC[[22]](#footnote-22). Jak już wspomniano wzorzec MVC jest wykorzystywany do separacji modelu i widoku przez bramę zwaną kontrolerem – pomaga to osiągnąć separację odpowiedzialności. Są to trzy zasadnicze komponenty tego wzorca dzielące między sobą różne odpowiedzialności: model jest odpowiedzialny za implementację logiki biznesowej aplikacji, widok wyłącznie za wyświetlanie informacji i nie powinien posiąść żadnej logiki biznesowej, zaś kontroler jest warstwą odpowiadającą za zarządzanie tym co ma być wyświetlane i jakie zmiany powinny zachodzić w modelu. Na przykład, gdyby tworzona aplikacja była oparta o wzorzec MVC to implementacja wyświetlania profilu użytkownika, czyli jednej z funkcjonalności aplikacji mogłaby wyglądać następująco:

1. Użytkownik wprowadza adres URL do przeglądarki. Na przykład, [host]/app/profile/:id, gdzie :id jest parametrem odpowiadającym identyfikatorowi użytkownika
2. Kontroler z metodą odpowiadającą temu adresowi pobiera dane użytkownika w oparciu o podany identyfikator, na przykład z repozytorium zwracającego model użytkownika
3. Odpowiednia metoda kontrolera zwraca widok przekazując do niego otrzymane dane z repozytorium
4. Strona zostaje wyświetlona w przeglądarce

Następnym wspomnianym wzorcem architektonicznym wywodzącym się ze wzorca MVC był wzorzec MVP (od ang. *Model-View-Presenter*). Wzorzec MVC pozwala na tworzenie lepszej architektury w kontekście prezentowania danych dla użytkowników, natomiast z jednej strony możliwe zmiany jednych komponentów mogą nie wpływać na działanie innych, ale z innej istnieje prawdopodobieństwo, że zmiany wprowadzone do jednego komponentu będą wymagały zmian w innych, na przykład zmiana wprowadzona do modelu może wymagać wprowadzania zmian do kontrolera oraz widoku. Właśnie dlatego aby złagodzić tę zależność został opracowany wzorzec MVP. Zasadnicza różnica między tymi wzorcami polega na tym, że widok nasłuchuje zdarzeń i przekazuje je do prezentera, który to ma wprowadzać zmiany do modelu. Tutaj prezenter odgrywa rolę podobną do kontrolera, natomiast wprowadza on większy poziom niezależności między widokiem i modelem, ponieważ tylko warstwa prezentera zawiera dane widoku i modelu, co sprawia, że zmiany wprowadzone do modelu nie będą wymagały zmian w widoku[[23]](#footnote-23). Mimo wszystko obecnie nie jest to często wykorzystywany wzorzec, a już w 2006 roku znany autor książek z architektury oprogramowania Martin Fowler stwierdził, że ten wzorzec powinien zostać podzielony na inne i wycofany[[24]](#footnote-24).

Ostatnim wspomnianym wzorcem architektonicznym był opublikowany w 2005 roku przez John’a Gossman’a, jednego z programistów pracujących nad rozwiązaniami WPF[[25]](#footnote-25) i Silverlight[[26]](#footnote-26), wzorzec MVVM (od ang. *Model-View-ViewModel*), który początkowo był przeznaczony do tworzenia aplikacji okienkowych w technologii WPF[[27]](#footnote-27). Jak już wspomniano wzorzec ten jest następcą wzorca MVC i MVP z dodatkową warstwą zwaną ViewModel. Jednak nawet mimo podobieństwa pomiędzy wzorcami warto rozpatrzyć oddzielnie dwie pierwsze warstwy wzorca MVVM: tutaj pierwsza warstwa, czyli model, tak jak w wypadku MVC, definiuje model domenowy albo warstwę dostępu do danych (w kontekście używanego w tworzonym projekcie framework’a Angular taką warstwą może być serwis), z kolei druga warstwa to widok, którego definicja nie odbiega od definicji wzorców MVC i MVP. Trzecią warstwą jest ViewModel, zadaniem którego jest jak wskazuje nazwa łączenie modelu i widoku, aniżeli separacja narzucana przez wzorzec MVC, w istocie ta warstwa powinna dostarczać dane z modelu do widoku. Wcześnie zostało podkreślone to, że ViewModel jest często porównywany do prezentera wzorca MVP, natomiast zasadniczą różnicą pomiędzy tymi warstwami jest to, że prezenter w porównaniu do ViewModel ma referencję do widoku, jednak widoki w MVVM są powiązane z ViewModel poprzez mechanizm *data binding* lub *binder*, który jest odpowiedzialny za wiązanie widoku z danymi dostarczanymi przez ViewModel. Dlatego, gdy zajdzie zmiana jakiejkolwiek wartości widok zostanie automatycznie odświeżony.[[28]](#footnote-28) Dla porównania, w kontekście framework’a Angular koncepcja ViewModel jest porównywalna ze skryptami komponentu.

# 2. Projektowanie

W tym rozdziale zostanie przedstawiony aspekt projektowania tworzonej aplikacji. Przede wszystkim zostaną opisane wykorzystywane technologie programistyczne, w tym frameworki do tworzenia SPA, frameworki używane do tworzenia backend’u oraz systemy zarządzania bazami danych. Do modelowania systemu zostaną wykorzystane diagramy UML w zasadzie opisujące stawiane systemowi wymagania. Ponadto zostanie rozpatrzony aspekt bezpieczeństwa systemów informatycznych.

# 2.1. Wykorzystane technologie programistyczne

Ten podrozdział zostanie poświęcony wykorzystanym technologiom programistycznym w trakcie opracowywania aplikacji „Friendy”. Na początku zostanie przedstawiony dodatkowy opis architektury SPA oraz zostaną przytoczone przykłady technologii używanych do tworzenia tego typu aplikacji, jednak najgłówniejszym tematem podrozdziału będzie szczegółowe przestawienie architektury używanej do opracowania frontend’u tworzonej aplikacji. Następnie zostanie przedstawiona platforma ASP.NET Core, w tym przykłady alternatywnych platform do tworzenia backend’u, architektura ASP.NET Core i framework SignalR. Z kolei w następnym podrozdziale zostanie przedstawiony system zarządzania bazą danych Microsoft SQL Server. Tutaj też zostaną przedstawione różnice pomiędzy podejściami SQL a NoSQL, przytoczone przykłady technologii tych typów baz danych, natomiast głównym aspektem podrozdziału będzie przedstawienie podstaw architektury MSSQL. W ostatnim podrozdziale zostanie przedstawiony silnik wyszukiwania o nazwie Elasticsearch.

# 2.1.1. Angular – technologia do tworzenia SPA

Obecnie programista ma do wyboru dość szeroki wachlarz sprawdzonych i renomowanych technologii programistycznych do tworzenia aplikacji jednostronicowych. Najbardziej popularnymi oraz przyjaznymi dla programisty technologiami opartymi o język JavaScript pozwalające utworzyć *Single Page Application* są wspomniane wcześniej Angular, React.JS oraz Vue.JS. Typowa aplikacja internetowa składa się z dwóch warstw: wspomniany *backend*, którego częścią ma być REST API oraz *frontend* definiujący aplikację jednostronicową utworzoną w oparciu o jedną z powyższych technologii. Alternatywnie *frontend* jest nazywany warstwą z którą użytkownik wchodzi w bezpośrednią interakcję w celu zarządzania danymi poprzez podłączony *backend*.

Jedną z najbardziej popularnych bibliotek do tworzenia aplikacji internetowych w przeciągu ostatniego dziesięciolecia była biblioteka o nazwie Angular.js. Tak jak każda inna biblioteka lub frameworksłużyła ona do ułatwiania tworzenia małych i dużych aplikacji internetowych, a także pozwalała budować dynamiczne widoki zamiast statycznych stron internetowych. Jednak w maju 2016 roku została wydana ostateczna wersja zupełnie nowego framework’a o nazwie Angular, wspieranego i rozbudowywanego przez zespół Google oraz społeczność i zazwyczaj aktualizowanego co sześć miesięcy[[29]](#footnote-29).

Celem tej technologii jest umożliwienie łatwego i efektywnego budowania aplikacji jednostronicowych w oparciu o język TypeScript[[30]](#footnote-30) i HTML5. Każda aplikacja utworzona w Angularze powinna składać się z przynajmniej jednej klasy oznaczonej dekoratorem @NgModule oraz modułu routing’u, jednak zazwyczaj, w zależności od rozmiaru, aplikacja składa się z dużo większej ilości modułów, na przykład definiujących podstrony oraz podzielone moduły routing’u. Ten moduł, podobnie jak inne, przyjmuje obiekt metadanych definiujący początkowe parametry działania aplikacji i typowo składa się z tych właściwości:

1. declarations to właściwość przyjmująca tablicę komponentów zarejestrowanych w aplikacji i widocznych w granicach modułu
2. imports to właściwość przyjmująca tablicę modułów zarejestrowanych w aplikacji i widocznych w granicach modułu
3. providers to właściwość przyjmująca tablicę serwisów zarejestrowanych w aplikacji i widocznych w granicach modułu
4. bootstrap to właściwość najczęściej wykorzystywana tylko w pierwszym modułu aplikacji, przyjmująca drzewo komponentów, które powinny zostać przeniesione do początkowego stanu DOM

Opisane wyżej podejście dzielenia modułu na podmoduły sprawia, że całe moduły mogą zostać w łatwy sposób przeniesione z jednego projektu do innego, a utrzymywanie i rozwój aplikacji staje się dużo łatwiejszy. Oprócz tego stosując to podejście programista może korzystać z tak zwanego mechanizmu *lazy loading*, które pozwala ładować całe moduły na żądanie, sprawiając, że czas uruchamiania aplikacji skraca się, ponieważ zmniejsza się rozmiar początkowo uruchomianego bundle’a.

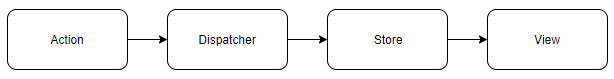
Drugim najważniejszym składnikiem każdej aplikacji są wspomniane wcześniej komponenty. Tak samo jak w wypadku modułów każda aplikacja powinna mieć przynajmniej jeden główny komponent, który łączy inne komponenty w całość. Właśnie ten komponent jest elementem przekazywanym do właściwości bootstrap z modułu głównego. Zazwyczaj każdy komponent w Angularze składa się z przynajmniej trzech plików[[31]](#footnote-31):

1. Skrypty komponentu. Jest to plik skryptów TypeScript, który ma zawrzeć definicję i implementację klasy oznaczonej dekoratorem @Component przyjmującego obiekt metadanych definiujących ów komponent. Zazwyczaj ten obiekt przyjmuje trzy właściwości. Pierwsza z nich to selector, która ma definiować nazwę komponentu do wykorzystania w widokach, na przykład główny komponent zazwyczaj jest definiowany przez selektor app-root, a dostęp do niego w widoku uzyskuje się poprzez kod HTML <app-root></app-root>. Zazwyczaj drugą właściwością jest template albo tempalteUrl – pierwsza może przyjąć kod HTML komponentu, ale druga przyjmuje ciąg znaków definiujących ścieżkę do pliku z kodem HTML komponentu. Trzecią i ostatnią tutaj opisywaną właściwością jest styles lub styleUrls – pierwsza przyjmuje tablicę ciągu znaków definiującą style komponentu, a druga ma przyjmować tablicę ścieżek do plików zawierających style komponentu. Oprócz tego obiektu wspomniana klasa może składać się z danych definiowanych przez zmienne, które mogą być używane przez widok lub metod reagujących na wydarzenia wewnątrz komponentu, na przykład wciśnięcie przycisku
2. *Template syntax*. Jest to plik, który oprócz kodu HTML zawiera elementy dostarczane przez framework. Jednym z takich elementów mogą być dyrektywy. Na przykład, \*ngIf to dyrektywa której używa się do warunkowego wyświetlania pewnych bloków kodu HTML. Drugim elementem jest mechanizm *data binding*, łączący widok ze skryptami komponentu. Ten mechanizm składa się z trzech innych mechanizmów. Pierwszy to *property binding*, który pozwala na przekazanie i łączenie właściwości komponentu rodzica z komponentami dziećmi. Drugi to mechanizm interpolacji wykorzystywanej do wyświetlania danych pobranych ze skryptów w widoku oraz wykorzystywany do wiązania wydarzeń ze zdefiniowanymi metodami skryptu komponentu *event binding*. Trzecim elementem jest *pipe*, który może być używany do transformacji wyświetlanych wartości. Czwartym i ostatnim elementem są dyrektywy atrybutów lub *attribute directives*, zazwyczaj używane do zmiany wyglądu lub zachowań elementów DOM
3. Style. Jest to jeden lub kilka plików przechowujących style elementów komponentu. Do tworzenia stylów może być używany zwykły CSS lub rozszerzenia typu SCSS, SASS

Trzecim najważniejszym składnikiem aplikacji są serwisy. Serwisy to klasy zawierające logikę i dane, które mogą być wstrzykiwane i wykorzystywane przez różne komponenty. To właśnie z poziomu serwisów powinna odbywać się komunikacja z backend’em. Każdy serwis podobnie jak komponent lub moduł powinien być oznaczony dekoratorem @Injectable, który sprawia, że serwis jest dostępny do wstrzykiwania w komponentach lub innych serwisach. Jak wspomniano wcześniej każdy serwis ma zakres widoczności zdefiniowany przez moduł, jednak alternatywnie zakres widoczności serwisu może być zdefiniowany przez tenże dekorator @Injectable i właściwość providedIn obiektu metadanych[[32]](#footnote-32).

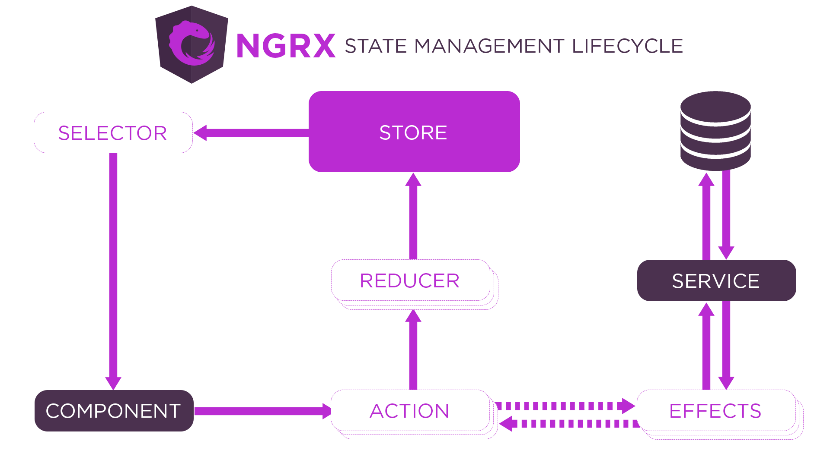
Inną wspomnianą zaletą SPA jest możliwość zarządzania stanem. W gruncie rzeczy stan aplikacji określa przepływ danych w aplikacji. Zarządzanie stanem ułatwia wzorzec o nazwie *flux*[[33]](#footnote-33), który jest wzorcem architektonicznym opisującym przepływ danych w oparciu o trzy główne części:

* + 1. *Dyspozytor*, który jest odpowiedzialny za przesyłanie akcji do magazynu
    2. *Magazyn* lub *stan*, który jest centralną jednostką systemu przechowującą dane
    3. *Widok*, który w zasadzie jest komponentem korzystającym z danych znajdujących się w magazynie



Rysunek 1 Flux - diagram przedstawiający podstawowy schemat przepływu danych

Jak widać na diagramie oprócz powyższych trzech części ten wzorzec zawiera również akcje, które składają się z metod przypisujących typy akcjom i dostarczających je do dyspozytora. Natywnie Angular zamiast programisty rozwiązuje większość problemów związanych z zarządzaniem stanem, ale z rozszerzeniem się aplikacji zachodzi konieczność skorzystania z dodatkowych narzędzi, z czego w kontekście aplikacji tworzonych w Angularze najpopularniejszym rozwiązaniem jest framework NgRx[[34]](#footnote-34).



Rysunek 2 Diagram przedstawiający ogólny schemat przepływu danych przy zarządzaniu stanem. Źródło: <https://ngrx.io/guide/store#diagram>

# 2.1.2. Tworzenie backend’u w ASP.NET Core

Tak samo jak w wypadku technologii przeznaczonych do tworzenia frontend’u programiści tworzący backend mają do wyboru dość dużo ciekawych rozwiązań. Praktycznie każdy popularny język programowania ma framework dedykowany do tworzenia aplikacji internetowych lub API konsumowanych przez frontend. Sprawia to, że programista tworzący rozwiązania niezwiązane z aplikacjami internetowymi, ale oparte o język programowania mający dedykowane frameworki do tworzenia aplikacji internetowych jest w stanie zacząć je tworzyć w bardzo krótkim czasie. Na przykład, programiści Java mogą korzystać z framework’a Spring[[35]](#footnote-35), programiści C# z ASP.NET, natomiast nawet programiści JavaScript mogą korzystać z wielu rozwiązań opartych o Node.js[[36]](#footnote-36), choć początkowo JavaScript był językiem używanym stricte do elementu interaktywności na stronach internetowych.

Obecnie jedną z najbardziej popularnych technologii do tworzenia małych aplikacji lub rozbudowanych systemów korporacyjnych jest framework ASP.NET Core. Jest to framework oparty o platformę .NET Core, który jest następcą framework’a ASP.NET opartego o poprzednika .NET Framework. Pierwsza wersja ASP.NET Core została wydana w czerwcu 2016 roku przez firmę Microsoft i jest aktywnie rozwijana przez twórców oraz społeczność. Głównymi zaletami ASP.NET Core są[[37]](#footnote-37):

1. *Rozszerzalność*, ponieważ ASP.NET Core jest zbudowany jako seria niezależnych komponentów mających dobrze zdefiniowane cechy, a funkcjonalności są oparte o interfejsy .NET lub klasy abstrakcyjne. To sprawia, że programista może zmieniać lub rozszerzać działanie kluczowych komponentów
2. *Testowalność*, ponieważ architektura ASP.NET Core sprawia, że całość aplikacji składa się z osobnych komponentów testowanie jest bardzo łatwe. Ponadto jest możliwość odizolowania i przetestowania natywnych komponentów platformy ASP.NET Core
3. *Współczesne API*, ponieważ ASP.NET Core jest oparte o .NET Core programista może korzystać ze wszystkich funkcjonalności dostarczanych przez język programowania C#
4. *Wieloplatformowość*, ponieważ ASP.NET Core w porównaniu do poprzednika umożliwia tworzenie rozwiązań opartych o różne systemy operacyjne
5. *Open source*,ponieważ kod źródłowy jest dostępny do pobrania przez każdego. Ponadto podczas testowania własnych rozwiązań opartych o ASP.NET Core jest możliwość sprawdzenia kodu źródłowego. Tym bardziej bardzo często funkcjonalność może być zaopatrzona w komentarze programisty, który ją opracował

Jedną z „cegieł” ASP.NET jest framework SignalR, który jest używany do tworzenia interaktywnych aplikacji czasu rzeczywistego. Ogólnie rzecz biorąc, SignalR odizolowuje użytkownika od szczegółów implementacji niskopoziomowych funkcjonalności umożliwiając tworzenie wirtualnych połączeń pomiędzy klientami w oparciu o jedną z dostarczanych natywnie technik transportowych: *WebSockets*, *Server-sent Events*, *Forever frame* lub *Long polling*[[38]](#footnote-38). Jak podkreślono wcześniej tworzona aplikacja będzie oferowała możliwość skorzystania z czatu stworzonego z użyciem tego rozwiązania.

# 2.1.3. Systemy zarządzania bazami danych

Obecnie systemy bazodanowe są dzielone na systemy relacyjne i nierelacyjne lub tak zwane systemy NoSQL. Główna różnica pomiędzy tymi systemami polega na tym, iż w porównaniu z systemami NoSQL obiekty w systemach SQL są związane relacjami, co oznacza, że każda tabela w bazie danych jest związana z innymi tabelami kluczami głównymi i obcymi. Przykładowymi bazami NoSQL mogą być MongoDB lub Redis. Ta ostania jest często wykorzystywana do zarządzania pamięcią podręczną, ponieważ dane są przechowywane w pamięci RAM, co czyni ją niezwykle wydajną, a zasada działania oparta jest o schemat klucz-wartość. W obecnej chwili najczęściej wykorzystywanymi bazami relacyjnymi są Oracle[[39]](#footnote-39), MySQL[[40]](#footnote-40), SQL Server i PostgreSQL[[41]](#footnote-41).

W tworzonym rozwiązaniu jak podkreślono we wstępie zostanie użyta relacyjna baza danych SQL Server, która jest komercyjnym systemem bazodanowym stworzonym i rozwijanym przez firmę Microsoft. W tej chwili jest możliwe użycie różnych edycji bazy danych skierowanych na mniejsze lub większe grona użytkowników i różnej wielkości rozwiązania. SQL Server wspiera wszystkie podstawowe typy danych, umożliwia tworzenie widoków i indeksów, a jedną z zalet SQL Server jest rozszerzenie języka SQL o nazwie T-SQL umożliwiające tworzenie procedur, funkcji i zmiennych lokalnych.[[42]](#footnote-42)

W znacznym stopniu architektura SQL Server jest zbudowana w taki sposób aby optymalizować operacje wejścia-wyjścia. Każda baza danych SQL Server składa się przynajmniej z dwóch plików: plik danych i plik logów. Jak wskazuje nazwa pierwszy plik jest przeznaczony do przechowywania danych, na przykład tabel, indeksów, procedur i widoków. Drugi jednak jest przeznaczony do przechowywania logów, które mogą być użyte do śledzenia transakcji[[43]](#footnote-43). Przechowywanie lokalnych plików odbywa się w oparciu o fundamentalną jednostkę pamięci zwaną stroną (od ang. *page*). Rozmiar strony wynosi 8 KB, a każda z nich posiada nagłówek o rozmiarze 96 bajtów używany do przechowywania metadanych strony: numer strony, typ strony, ilość wolnej przestrzeni w obrębie strony oraz identyfikator alokacji obiektu przechowującego stronę. Zasadniczo typ strony definiuje dane przechowywane przez stronę. Dla przykładu takimi typami mogą być wiersze danych, indeksy, tekst lub obraz. Maksymalny rozmiar wiersza na stronie nie może przekraczać 8 KB, jednak jeżeli dana wiersza odpowiada typowi varchar i pochodnym to część przekraczająca granicę 8 KB może zostać przeniesiona do strony lub stron należących do jednostki alokacji (od ang. *allocation unit*). Inną jednostką pamięci przechowującą tabele i indeksy są zakresy (od ang. *extents*), z czego każdy zakres przechowuje 8 sąsiednich stron, czyli wykorzystuje 64 KB pamięci. Istnieje dwa typy zakresów: jednolite (od ang. *uniform*) i mieszane (od ang. *mixed*). Jednolite zakresy przechowują obiekty należące do jednego typu, na przykład strony odnoszące się do tej samej tabeli, lecz zakresy mieszane przechowują strony należące do maksymalnie ośmiu różnych obiektów[[44]](#footnote-44).

Jednym z najważniejszym komponentów SQL Server sprawiającym, że operacje wejścia-wyjścia są jeszcze wydajniejsze jest zarządzanie buforem (od ang. Buffer Management). Całość zarządzania buforem jest podzielona na dwa mechanizmy:

* + - 1. Menedżer buforu, używany do uzyskiwania dostępu i aktualizacji stron
      2. Pula buforu, używana do optymalizacji operacji wejścia-wyjścia poprzez przechowywanie strony w pamięci RAM

Menedżer buforu jest wykorzystywany wyłącznie do odczytywania i zapisywania danych do bazy danych. Te operacje są wykonywane asynchronicznie, co skutkuje tym, że wątek wywołujący operację może przetwarzać następne czynności, gdy wówczas w tle ma miejsce operacja wejścia-wyjścia.[[45]](#footnote-45)

Pobieranie danych z bazy SQL Server odbywa się poprzez zapytania, a w kontekście pobierania danych podstawowym zapytaniem jest SELECT. Jest to typ zapytania nieproceduralnego, które nie mówi jednoznacznie jak powinno się ono wykonać, dlatego to serwer bazy danych, mianowicie komponent o nazwie optymalizator zapytań (od ang. *query optimizer*) powinien je przetworzyć i zdecydować w jaki najbardziej efektywny sposób należy je wykonać. Wynikiem tej operacji jest tak zwany plan wykonania zapytania (od ang. *query execution plan*) w oparciu o który zostają pobierane dane przez silnik relacyjny. Ponadto należy też podkreślić, że plan ten zostaje dodany do pamięci podręcznej w celu szybszego wykonania tego samego zapytania w przyszłości.[[46]](#footnote-46)

# 2.1.4. System wyszukiwania Elasticsearch

Elasticsearch to rozproszony system wyszukiwania i silnik analityczny służący do przechowywania dokumentów w formacie JSON[[47]](#footnote-47) zbudowany w oparciu o Apache Lucene[[48]](#footnote-48), które mogą składać się z pól o większości istniejących typów danych. Typowo system informatyczny korzystający z tego rozwiązania składa się z powiązanych indeksów przechowujących typowe dla siebie dane zwane dokumentami. Głównymi zaletami tego systemu jest wydajność i skalowalność ponieważ indeksy mogą być podzielone na odłamki przechowywane w różnych klastrach.

Działanie systemów wyszukiwania zasadniczo różni się od działania zwykłych systemów bazodanowych, ponieważ wyszukiwanie tekstu opiera się o indeksy odwrócone. Nawiązując do SQL indeks może być porównywany do tabeli, dokument do wiersza, a pola przechowywane przez dokument do kolumn. Przykładem działania indeksów może być blog internetowy na którym każdy artykuł odnoszony jest do kategorii. Zamiast przechowywać dane o kategoriach wpisu w postaci surowej są one przechowywane w odniesieniu do indeksu, co skutkuje dużo szybszym tempem wyszukiwaniem artykułów odnoszących się do pewnej kategorii ponieważ Lucene może przypisywać listę artykułów do każdej kategorii[[49]](#footnote-49).

Standardowo każde pole jest indeksowane, ale struktura danych w której jest ono przechowywane zależy od jego typu. Na przykład, zwykłe pola tekstowe są przechowywane w indeksach odwróconych, ale dane geograficzne i liczby w drzewach BKD. Jednak Elasticsearch zezwala na ręczne ustawienie mapowań typów i schemat indeksowania pola.[[50]](#footnote-50)

Zarządzanie danymi indeksu jest możliwe na różne sposoby. Jednym z nich jest Search API i interfejs webowy HTTP oparte o mechanizm Query DSL (od ang. *Domain Specific Language*)[[51]](#footnote-51), który umożliwia tworzenie ciał zapytań w formacie JSON. Poza tym jest możliwość skorzystania z klienta ES w językach programowania. W tym projekcie oprócz Search API zostanie użyty klient o nazwie NEST przeznaczony do języka C#.

# 2.2. Planowanie aplikacji

Na początku tego podrozdziału zostaną przedstawione listy wymagań pozafunkcjonalnych i funkcjonalnych. W pierwszym podrozdziale zostanie przedstawiona lista wymagań pozafunkcjonalnych. Następnie zostaną zidentyfikowani aktorzy i przedstawiona lista wymagań funkcjonalnych. Z kolei w oparciu o wymagania funkcjonalne zostaną sporządzone diagramy przypadków użycia UML.

# 2.2.1. Wymagania pozafunkcjonalne

1. Użyte w aplikacji technologie powinny zapewniać optymalne działanie względem urządzeń na których działają
2. Wszystkie lub większość funkcjonalności użytych w aplikacji powinny być kompatybilne z większością przeglądarek
3. Funkcjonalności zaimplementowane w aplikacji powinny działać w prosty i jasny sposób
4. Frontend aplikacji powinien powstać z użyciem technologii Angular
5. Zarządzanie stanem frontend’u powinno opierać się o technologię NgRx
6. Backend powinien być aplikacją monolityczną utworzoną z użyciem technologii ASP.NET Core
7. Wszystkie funkcjonalności systemowe oparte o protokół WebSocket powinny być częścią monolitycznej aplikacji oraz powstać z użyciem biblioteki SignalR
8. Każdy przesłany plik nie może przekraczać rozmiaru 1MB
9. Każde zdjęcie przesłane przez użytkownika powinno zostać skompresowane przed zapisaniem na serwerze
10. Wszystkie przesyłane pliki powinny być zapisywane na serwerze

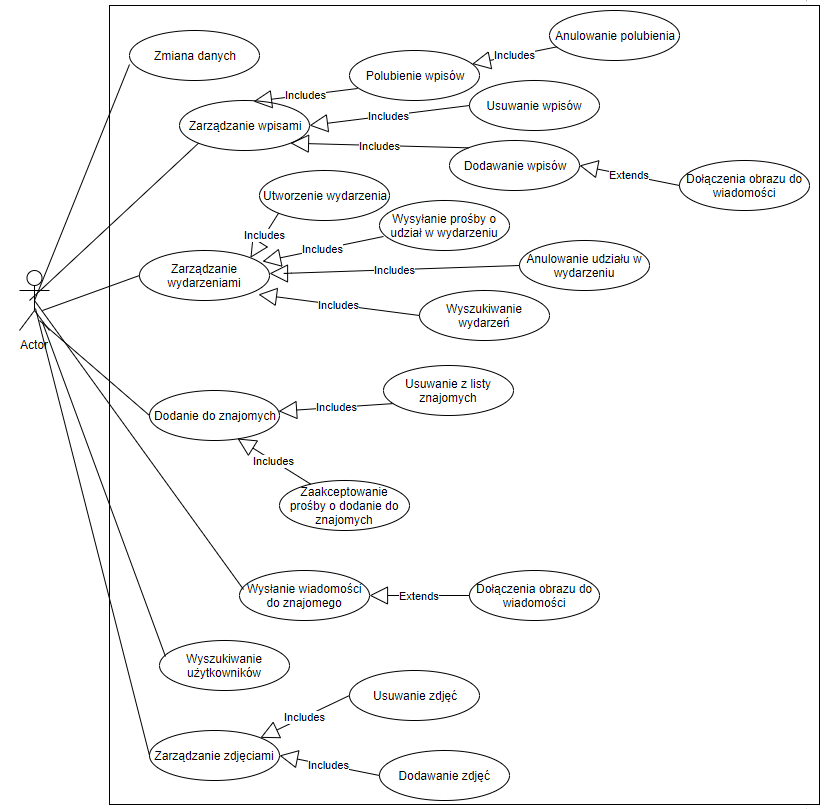
# 2.2.2. Wymagania funkcjonalne

Na tym etapie należy zidentyfikować aktorów w systemie, ponieważ następnie przedstawiona lista wymagań funkcjonalnych w pierwszej kolejności będzie bazowała na aktorach i przypadkach użycia. W celu zidentyfikowania aktorów w systemie należy odnieść się do przykładu typowego portalu internetowego. Głównym aktorem takiego zasobu jest zwykły użytkownik, który powinien móc założyć konto i korzystać z serwisu. Ponadto do grona aktorów należy zaliczyć administratora wydarzenia, który powinien mieć dostęp do ważnych funkcjonalności systemowych pozwalających na monitorowanie i podtrzymywanie porządku na stronie zaplanowanego wydarzenia. Poza tym aktorami będą zatwierdzony uczestnik i założyciel wydarzenia nadzielony dodatkowymi uprawnieniami.

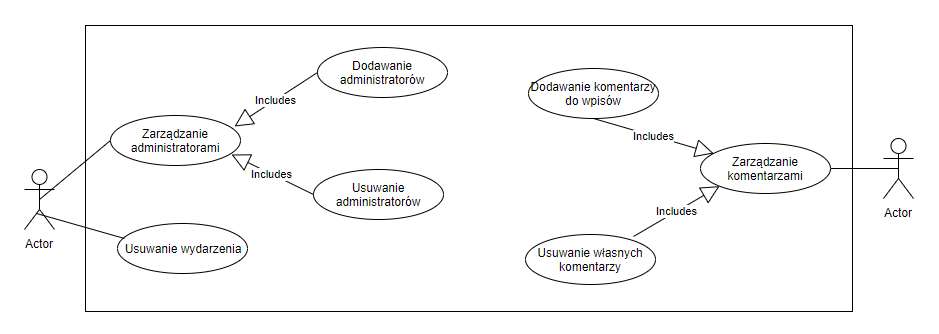
Lista wymagań funkcjonalnych wygląda następująco:

1. Użytkownik powinien mieć możliwość zarejestrowania się oraz logowania poprzez podanie emaila oraz hasła
2. Użytkownik powinien móc zmieniać swoje dane, hasło oraz email
3. Użytkownik ma mieć możliwość wyszukiwania wydarzeń i dołączenia do nich
4. Użytkownik powinien mieć możliwość napisania wiadomości do znajomych
5. Użytkownik ma mieć możliwość dołączenia jednego zdjęcia do wiadomości
6. Użytkownik powinien mieć możliwość wyszukiwania zarejestrowanych użytkowników i nawiązania z nimi kontaktu po dodaniu potencjalnego znajomego
7. Użytkownik powinien zaakceptować prośbę o dodanie do znajomych innego użytkownika zanim będą oni mogli nawiązać kontakt
8. Każdy użytkownik powinien mieć możliwość utworzenia wydarzenia oraz zarządzania nim
9. Użytkownik powinien mieć możliwość usuwać znajomych z listy znajomych
10. Użytkownik ma mieć możliwość dodawać oraz usuwać swoje zdjęcia
11. Użytkownik powinien mieć możliwość umieszczać wpisy na swojej stronie profilowej
12. Do każdego wpisu użytkownik ma móc podpiąć tylko jedno zdjęcie
13. Użytkownik ma móc składać wniosek o uczestnictwo w wydarzeniu
14. Administrator wydarzenia ma móc powinien móc zatwierdzać uczestnictwo chętnych użytkowników w wydarzeniu
15. Administrator wydarzenia powinien być uprawniony do zmieniania podstawowych danych wydarzenia
16. Administrator wydarzenia powinien móc dodawać nowe wpisy
17. Administrator wydarzenia powinien mieć możliwość usuwania wpisów
18. Członkowie wydarzenia powinni mieć możliwość dodawania komentarzy do wpisów
19. Administrator wydarzenia powinien mieć możliwość usuwania komentarzy
20. Administrator wydarzenia powinien mieć możliwość aktualizować wszystkie dane wydarzenia
21. Administrator wydarzenia powinien mieć możliwość usuwać oraz blokować dostęp do wydarzenia użytkownikom
22. Administrator wydarzenia powinien móc dodawać i usuwać nowe zdjęcia na stronie wydarzenia
23. Użytkownik jako właściciel profilu powinien mieć możliwość dodawania wpisów na stronie profilowej
24. Tylko założyciel wydarzenia ma móc dodawać i usuwać administratorów wydarzenia

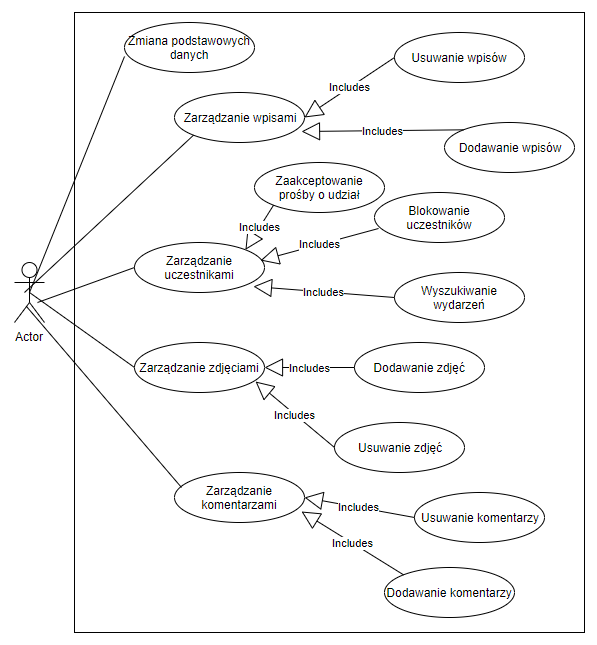
Ponieważ lista powyższych wymagań w miarę rozwoju aplikacji może rosnąć prędzej czy później wyniknie konieczność usystematyzowania tej listy do krótszej i bardziej czytelnej formy. Istnieje kilka metod usystematyzowania wymagań – jedną z nich jest przedstawienie graficzne funkcjonalności w formie diagramu przypadków użycia. Diagram przypadków użycia jest jednym z pierwszych diagramów UML (zunifikowany język modelowania), który służy do modelowania systemu informatycznego w jak najbardziej przejrzysty i zrozumiały dla czytelnika sposób poprzez wyselekcjonowanie aktorów i przypisywanie im różnorakich funkcjonalności systemowych. Aplikacja ma czterech aktorów: użytkownik, administrator wydarzenia, członek wydarzenia oraz założyciel wydarzenia. Do sporządzenia diagramów przypadków użycia zostanie użyte narzędzie draw.io



Rysunek 3 Diagram przypadków użycia przedstawiający uprawnienia użytkownika



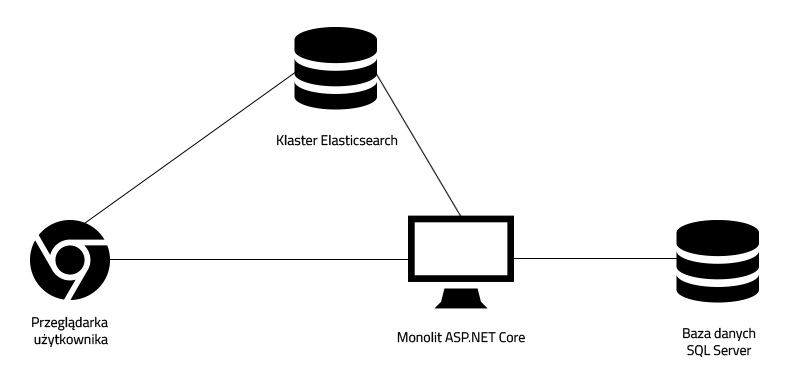
Rysunek 4 Diagram przypadków użycia przedstawiający dodatkowe uprawnienia założyciela wydarzenia (oprócz podstawowych uprawnień administratora) oraz uczestnika wydarzenia



Rysunek 5 Diagram przypadków użycia przedstawiający uprawnienia administratora wydarzenia

# 2.3. Architektura systemu

Po pozyskaniu i analizie wymagań należy zastanowić się nad architekturą systemu informatycznego. Jednymi z głównych wymagań stawianym każdemu systemowi są wydajność, niezawodność i bezpieczeństwo. Spełnienie takich oczekiwań od systemu mogą zagwarantować tylko sprawdzone wydajnościowo technologie programistyczne i dostosowanie się do zalecanych praktyk projektowania systemów informatycznych. W tym podrozdziale zostaną omówione aspekty związane z architekturą tworzonej aplikacji, w tym zostanie przedstawiona koncepcja REST API, aspekt bezpieczeństwa oraz aspekt autentykacji i autoryzacji.



Rysunek 6 Diagram przedstawiający architekturę tworzonego systemu

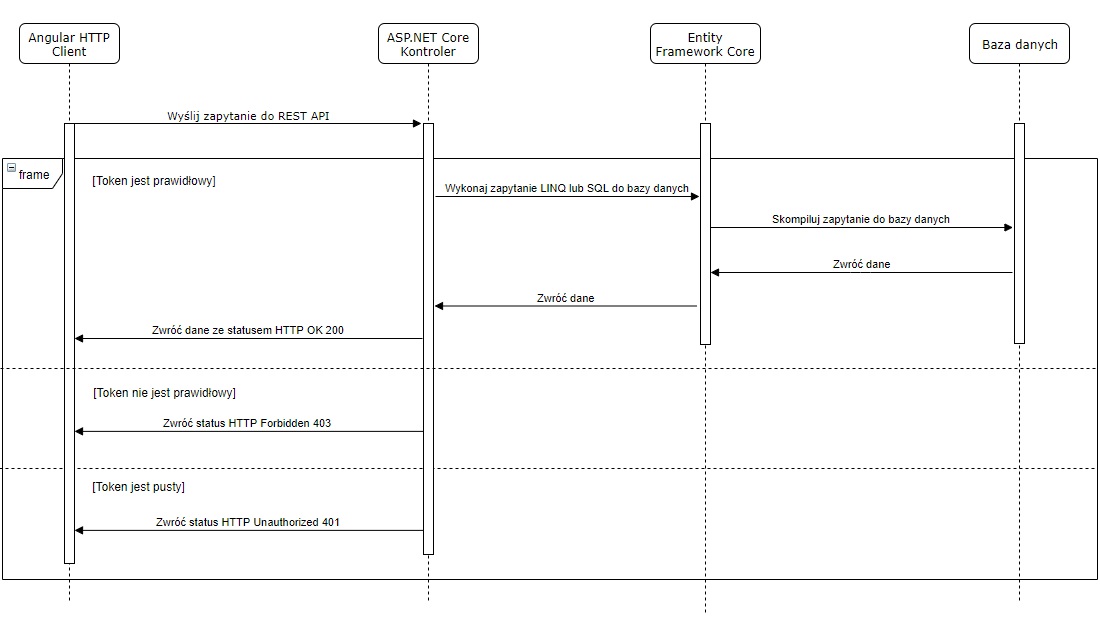
# 2.3.1. REST API

Komunikacja między serwisami Web API odbywa się poprzez użycie protokołów SOAP, JSON-RPC, XML-RPC lub metodologii REST. Obecnie to właśnie REST jest najbardziej popularnym podejściem do tworzenia prostych lub rozbudowanych Web API. *Representational state transfer* jest metodologią obejmującą zbiór zasad, którymi powinno cechować się Web API.[[52]](#footnote-52)

|  |  |
| --- | --- |
| Nazwa | Opis |
| Klient-serwer | Wymuszanie separacji obowiązków między frontend’em a backend’em, który z kolei powinien zawrzeć reguły i logikę biznesową oraz implementację zarządzania magazynem danych |
| Bezstanowość | Komunikacja między serwerem a klientem powinna być bezstanowa, co oznacza, iż to klient jest odpowiedzialny za wysyłanie danych sesji do serwera z każdym zapytaniem. Ewentualnie serwer może przesyłać te dane do innych serwisów, na przykład bazy danych w celu przechowywania danych aktualnej sesji |
| Jednolity interfejs | Jest to fundamentalna część opisująca cztery główne aspekty lub zasady REST[[53]](#footnote-53):   * 1. Identyfikowanie zasobów – każdy zasób powinien być identyfikowalny poprzez unikatowy adres URL   2. Zarządzanie zasobami – klient powinien móc zarządzać zasobami w przypadku gdy jest do tego uprawniony   3. Samoopisująca się odpowiedź – każda wiadomość powinna zawierać dostateczną ilość informacji o sobie, na przykład prawidłowej metodzie przetworzenia (Typ MIME)   4. HATEOAS lub *Hypermedia as the engine of application state* – ograniczenie mówiące, że wysyłając zapytanie pod stały adres URL klient w odpowiedzi ma otrzymać listę następnych kroków do podjęcia |
| Pamięć podręczna | Każda odpowiedź serwera powinna zawrzeć nagłówek podkreślający to czy odpowiedź może zostać zapisana do pamięci podręcznej w celu przyspieszenia działania tychże zapytań w przyszłości |
| Kod na żądanie | Jest to opcjonalne ograniczenie oznaczające możliwość wysyłania wykonywalnego kodu do klienta, na przykład komponentów napisanych w języku Javascript |

Tabela 1 Pryncypia którymi musi cechować się każdy system REST

Najczęściej proces wysyłania zapytań do API można opisać poprzez poniższy diagram sekwencji.



Rysunek 10 Diagram sekwencji przedstawiający proces wysyłania zapytań HTTP GET do API z użyciem Angular HTTP Client

# 2.3.2 Bezpieczeństwo aplikacji

Jednym z najważniejszych aspektów systemów informatycznych jest bezpieczeństwo. Obecnie istnieje dość dużo rodzajów ataków przed którymi należy zabezpieczać system informatyczny. W 2017 roku organizacja OWASP wydała nową wersję artykułu o 10 najważniejszych atakach lub usterkach aplikacji internetowych[[54]](#footnote-54).

|  |  |  |
| --- | --- | --- |
| Nazwa | Krótki opis | Możliwy przykład podatności w aplikacji |
| *SQL Injection* | Atak polega na próbie wykonania zapytań T-SQL z poziomu dynamicznie tworzonych zapytań SQL[[55]](#footnote-55) | Próba wpisania zapytania SQL do pola tekstowego na stronie „Profil” podczas tworzenia nowego wpisu, które może zwrócić wrażliwe dane z bazy danych lub usunąć tabelę |
| Nieprawidłowo zaimplementowane uwierzytelnianie i zarządzanie sesją | Atak polega na istnieniu którejkolwiek podatności umożliwiającej kradzież hasła, tokenu, danych sesji lub wykonywaniu poleceń od imienia użytkownika | Umożliwienie podania użytkownikowi podatnego na skradzenie hasła |
| *XML External Entities* | Atak polega na przesyłaniu tekstu XML zawierającego encje mogące odczytywać dane z serwera | Przesyłanie i parsowanie pliku xml, który zawiera encję mogącą wyciągać i zwracać wewnętrzne dane serwera |
| *Broken Access Control* | Usterka polega na nieprawidłowym ustawieniu kontroli dostępu do danych | Umożliwienie dostępu do strony ustawień wydarzenia zwykłym użytkownikom lub uczestnikom |
| Nieprawidłowa konfiguracja bezpieczeństwa | Usterka polega na nieprawidłowym ustawieniu konfiguracji serwera skutkującym podatnością na różnego rodzaju ataki | Umożliwienie dostępu do plików znajdujących się na serwerze |
| *XSS (Cross-Site-Scripting)* | Atak polega na wykonywaniu złośliwych skryptów od imienia użytkownika | Przechowywanie wrażliwych danych, na przykład tokenu JWT, w lokalnych magazynach dostępnych z poziomu Javascript, na przykład local storage lub session storage lub zwykłe ciasteczka |
| Niezabezpieczona deserializacja | Atak polega na przesyłaniu i deserializacji obiektów bez dodatkowej walidacji przesłanej zawartości | Przesłanie danej o typie liczby całkowitej zamiast oczekiwanego łańcucha tekstowego |
| Ujawnianie wrażliwych danych | Usterka polega na narażeniu takich danych jak numer karty kredytowej, hasło, token poprzez niezaszyfrowanie lub niedostateczną protekcję | Przechowywanie niezaszyfrowanych haseł użytkowników w bazie danych |
| Korzystanie z komponentów ze znanymi podatnościami | Usterka polega na braku aktualizacji komponentów systemu, które mogą zawierać inne podatności | Nieaktualizowanie części systemu zawierającej znaną podatność na atak |
| Niedostateczne logowanie i monitorowanie | Podatność polega na braku lub niedostatecznym poziomie logowania zachowań aplikacji | Niedostateczne monitorowanie zachowań aplikacji może skutkować awarią systemu po ataku |

Tabela 2 10 najważniejszych ataków lub usterek w aplikacjach internetowych

# 2.3.3. Uwierzytelnianie i autentykacja

Takie aplikacje jak sklepy internetowe, fora internetowe, sieci społecznościowe działają na zasadzie autoryzacji i możliwości identyfikowania każdego użytkownika: danych użytkownika, zasobów do których użytkownik może mieć dostęp. Ten proces przebiega głównie poprzez podanie unikatowego identyfikatora, na przykład loginu lub adresu email oraz hasła. Hasło jest najbardziej wrażliwą daną i wymaga szczególnej uwagi w procesie autoryzacji.

Obecnie bardzo popularnym rozwiązaniem jest użycie metody autoryzacji opartej o JWT. Według dokumentu RFC7519[[56]](#footnote-56) JSON Web Token jest standardem przeznaczonym do przechowywania i przekazywania praw użytkownika do zasobów informatycznych.

****

Rysunek 7 Przykładowy zaszyfrowany JWT

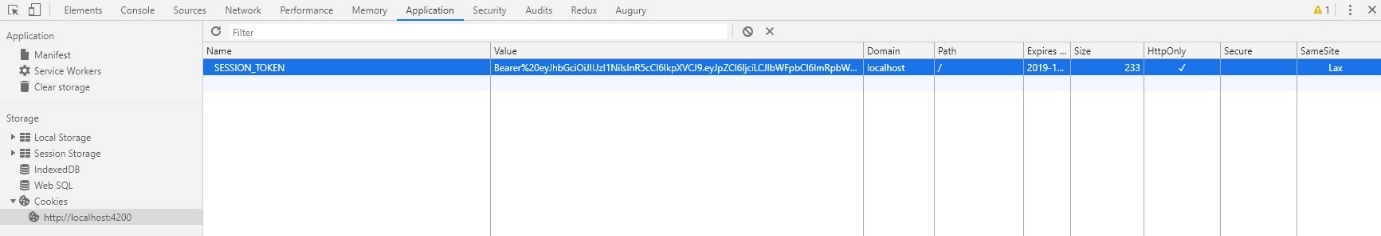
Powyższy rysunek przedstawia przykładowy zaszyfrowany token. Każdy JWT składa się z trzech strukturalnych części zwanych nagłówkiem, zawartością oraz sygnaturą. Na przykładowej ilustracji JWT czerwonym kolorem jest oznaczony nagłówek, fioletowym zawartość. Odkodowując przykładowy token do postaci czytelnej uzyskuje się dwa obiekty JSON przedstawione na rysunku poniżej.



Rysunek 8 Odkodowany JWT

Na podstawie powyższego rysunku można łatwo przeanalizować przykładowy token. Nagłówek, odpowiadający kolorowi czerwonemu, składa się z dwóch właściwości „typ” oraz „alg”, z czego „typ” oznacza typ tokenu odpowiadający JWT i właściwość „alg” oznacza algorytm używany do zaszyfrowania tokenu. Zawartość odpowiadająca kolorowi fioletowemu składa się z dwóch właściwości, których można użyć do identyfikacji użytkownika. Oprócz powyższych dwóch elementów pozostaje nagłówek, który powinien zawrzeć tajny klucz używany do haszowania sygnatury.

Po wygenerowaniu powinno się znaleźć miejsce do przechowywania JWT. W kontekście przeglądarek programista może skorzystać z local storage, session storage oraz ciasteczek. Jak już wspomniano żadne z powyższych nie jest bezpiecznym miejscem do przechowywania danych sesji ponieważ może to skutkować podatnością na atak XSS. Taka podatność może doprowadzać do sytuacji kradzieży tokenu i podawania się za innych użytkowników. Dlatego jednym z bardziej bezpiecznych rozwiązań jest przechowywanie tokenu w ciasteczku o fladze HTTPOnly, które może odczytać wyłącznie serwer. Użytkownik może dowiedzieć się o istnieniu takiego ciasteczka tylko przez narzędzia programisty w przeglądarce.



Rysunek 9 Przykład ciasteczka HTTPOnly o nazwie SESSION\_TOKEN zawierającego JWT

**3. Implementacja**

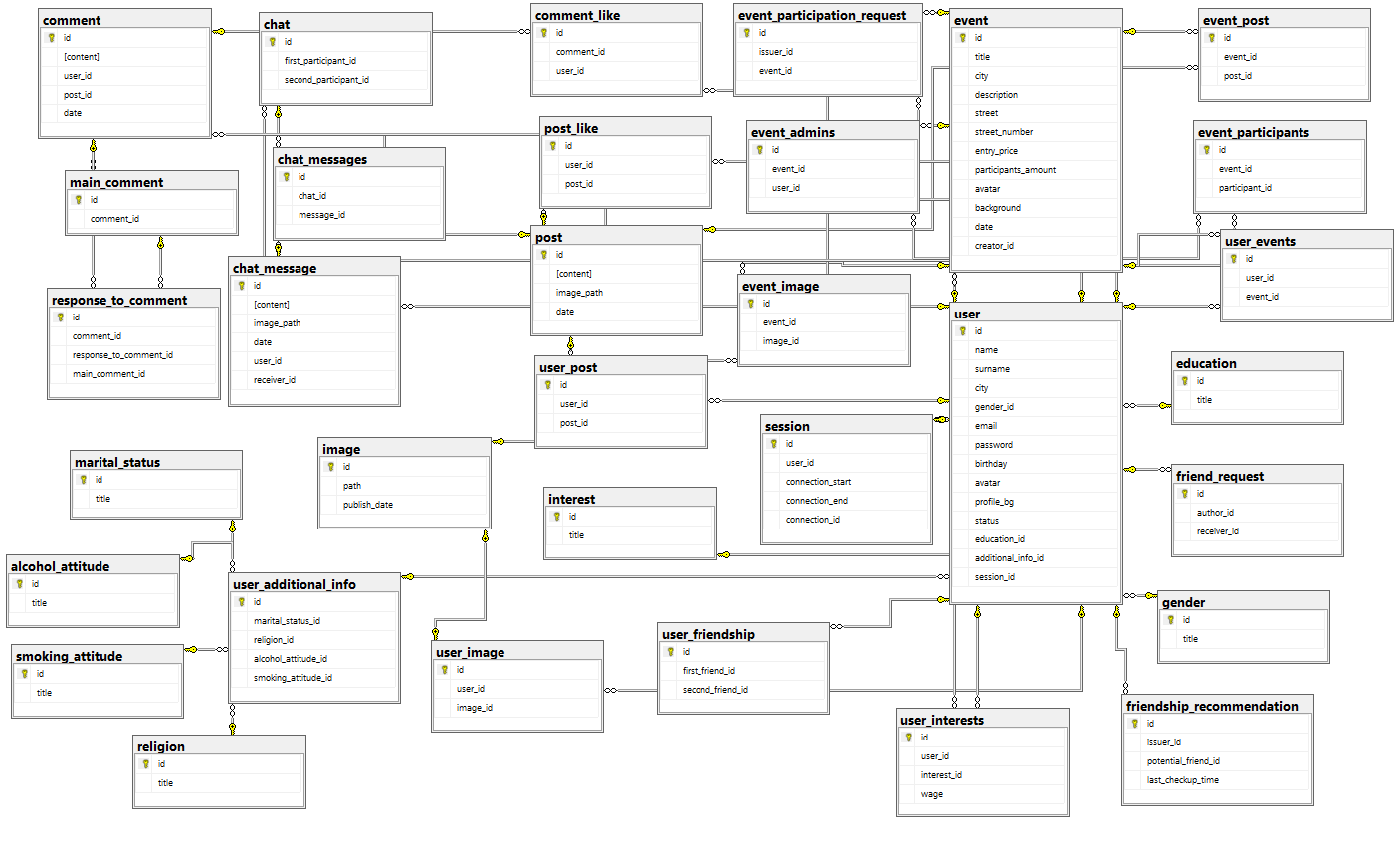
W tym rozdziale zostanie opisany proces tworzenia aplikacji, w tym zostaną przedstawione szczegóły implementacji najważniejszych elementów aplikacji.

# 3.1. Projektowanie i wykorzystanie bazy danych w ASP.NET Core

W oparciu o przedstawione wymagania strukturę bazy danych przedstawia następująca tabela i diagram ER.

|  |  |  |  |
| --- | --- | --- | --- |
| Tabela | Atrybut | Typ | Opis tabeli |
| user | id | int | Przechowuje podstawowe dane użytkownika |
| name | varchar(50) |
| surname | varchar(50) |
| city | varchar(50) |
| gender\_id | int |
| email | varchar(250) |
| password | varchar(250) |
| birthday | date |
| avatar | varchar(max) |
| profile\_bg | varchar(max) |
| status | varchar(100) |
| education\_id | int |
| additional\_info\_id | int |
| session\_id | int |
| chat | id | int | Przechowuje dane dwóch użytkowników jednego dialogu |
| first\_participant\_id | int |
| second\_participant\_id | int |
| chat\_message | id | int | Przechowuje dane jednej wiadomości w dialogu |
| content | varchar(500) |
| image\_path | varchar(1000) |
| date | datetime2(7) |
| user\_id | int |
| receiver\_id | int |
| chat\_messages | id | int | Łączy wiadomości z konkretnym dialogiem poprzez chat\_id |
| chat\_id | int |
| message\_id | int |
| comment | id | int | Przechowuje dane o komentarzach pod wpisami |
| content | varchar(500) |
| user\_id | int |
| post\_id | int |
| date | datetime2(7) |
| comment\_like | id | int | Przechowuje dane o polubionych przez użytkowników wpisach |
| comment\_id | int |
| user\_id | int |
| comment\_respond | id | int | Przechowuje dane o odpowiedziach na komentarze |
| content | varchar(500) |
| comment\_id | int |
| user\_id | int |
| date | datetime2(7) |
| education | id | int | Przechowuje możliwy poziom edukacji użytkownika |
| title | varchar(150) |
| event | id | int | Przechowuje podstawowe dane wydarzenia |
| title | varchar(250) |
| city | varchar(250) |
| description | varchar(1000) |
| street | varchar(250) |
| street\_number | varchar(20) |
| entry\_price | decimal(10,2) |
| participants\_amount | int |
| avatar | varchar(MAX) |
| background | varchar(MAX) |
| date | datetime2(7) |
| creator\_id | int |
| event\_admins | id | int | Przechowuje listę administratorów wydarzenia |
| event\_id | int |
| user\_id | int |
| event\_image | id | int | Przechowuje listę obrazów wydarzeń |
| event\_id | int |
| image\_id | int |
| event\_participants | id | int | Przechowuje listę uczestników wydarzeń |
| event\_id | int |
| participant\_id | int |
| event\_post | id | int | Przechowuje listę wpisów na stronie wydarzenia |
| event\_id | int |
| post\_id | int |
| friend\_request | id | int | Przechowuje dane próśb o dodanie do znajomych |
| author\_id | int |
| receiver\_id | int |
| gender | id | int | Przechowuje dane płci do wybrania przez użytkownika |
| title | varchar(50) |
| image | id | int | Przechowuje dane o obrazie znajdującym się na serwerze |
| path | varchar(MAX) |
| publish\_date | datetime2(7) |
| interest | id | int | Przechowuje listę zainteresowań do wybrania przez użytkowników |
| title | varchar(150) |
| marital\_status | id | int | Przechowuje listę stanów cywilnych do wybrania przez użytkownika |
| title | varchar(150) |
| post | id | int | Przechowuje podstawowe dane wpisu |
| content | varchar(1000) |
| image\_path | varchar(1500) |
| date | datetime |
| post\_like | id | int | Przechowuje dane o polubionych wpisach przez użytkowników |
| user\_id | int |
| post\_id | int |
| religion | id |  | Przechowuje listę wierzeń religijnych do wybrania przez użytkownika |
| title |  |
| session | id | int | Przechowuje dane sesji użytkownika |
| user\_id | int |
| connection\_start | datetime2(7) |
| connection\_end | datetime2(7) |
| smoking\_attitude | id | int | Przechowuje listę możliwych nastawień do palenia do wybrania przez użytkownika |
| title | varchar(150) |
| user\_additional\_info | id | int | Przechowuje dodatkowe dane najczęściej wykorzystywane podczas wyszukiwania |
| education\_id | int |
| martial\_status\_id | int |
| religion\_id | int |
| alcohol\_attitude\_id | int |
| smoking\_attitude\_id | int |
| user\_events | id | int | Przechowuje listę wydarzeń w których użytkowników bierze udział |
| user\_id | int |
| event\_id | int |
| user\_friendship | id | int | Przechowuje dane o potwierdzonych znajomościach w serwisie |
| first\_friend\_id | int |
| second\_friend\_id | int |
| user\_image | id | int | Przechowuje listę obrazów dodanych przez użytkownika |
| user\_id | int |
| image\_id | int |
| user\_interests | id | int | Przechowuje listę zainteresowań użytkownika |
| additional\_info\_id | int |
| interest\_id | int |
| user\_post | id | int | Przechowuje listę wpisów użytkownika |
| user\_id | int |
| post\_id | int |
| response\_to\_comment | id | int | Przechowuje dane o odpowiedzi na komentarz |
| comment\_id | int |
| response\_to\_comment\_id | int |
| main\_comment\_id | int |
| main\_comment | id | int | Przechowuje główne komentarze do wpisów |
| comment\_id | int |
| user\_id | int |
| visited\_profile\_id | int |
| friendship\_recommendation | id | int | Przechowuje polecenia dodania do znajomych dla poszczególnych użytkowników |
| issuer\_id | int |
| potential\_friend\_id | int |
| last\_checkup\_time | datetime |

Tabela 3 Lista tabel projektowej bazy danych



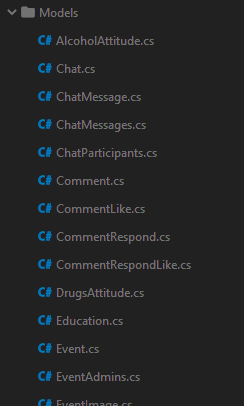
Rysunek 11 Diagram ER zaprojektowanej bazy danych

Po utworzeniu bazy danych można przejść do etapu tworzenia API. Twórcy .NET Framework zaopatrzyli programistów w narzędzie Entity Framework (po aktualizacji z ostatniej wersji .NET Framework do .NET Core zmieniono nazwę tego narzędzia na Entity Framework Core). Głównym zadaniem Entity Framework’a jest uproszczenie implementacji nawiązania komunikacji między aplikacją .NET a bazą danych. Jedną z funkcjonalności rozwiązania jest podejście zwane *Database First*, które umożliwia wygenerowanie POCO[[57]](#footnote-57) z istniejącej bazy danych. Ponieważ backend tworzonej aplikacji opiera się o ASP.NET Core należy wówczas korzystać z Entity Framework Core, które umożliwia wygenerowanie modeli domenowych używając poniższej komendy.



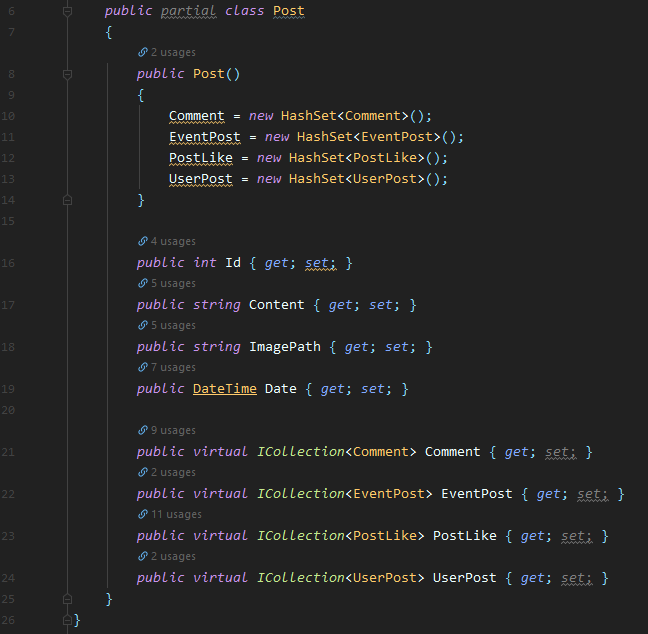
Rysunek 12 Komenda umożliwiająca wygenerowanie modeli domenowych

Po wygenerowaniu modeli domenowych do projektu zostanie dodany folder o nazwie Models, który będzie mieścił wszystkie modele oraz konfigurację kontekstu bazy danych wygenerowanego przez Entity Framework Core.



Rysunek 13 Ilustracja zawartości folderu Models, który mieści wygenerowane modele przez Entity Framework Core

Zgodnie z przedstawionym modelem na poniższym rysunku model jest klasą składającą się z właściwości odpowiadającym atrybutom w bazie danych – Entity Framework Core sprawia, że nazwy zmiennych i typy są zachowane lub przybliżone do tych właściwych językowi programowania, na przykład typ varchar jest zmieniany w typ C# string. Ponadto poprzez typ ICollection tworzą się wiązania z innymi tabelami na podstawie relacji poprzez klucze główne i obce w bazie danych.



Rysunek 14 Model wpisu jako przykładowy model wygenerowany przez EF Core

# 3.2. Projektowanie i tworzenie REST API

Każde API ma swoisty most zwany kontrolerem, który składa się z metod odpowiedzialnych za wczytywanie żądań HTTP i wysyłanie odpowiedzi do klienta[[58]](#footnote-58). Najczęściej wykorzystywanymi metodami HTTP w tworzonym REST API są GET, POST, PUT i DELETE. REST API bezpośrednio współdziała z warstwą dostępu do danych, która jest odpowiedzialna za implementację funkcjonalności dostępu do danych i przekazywanie obiektów biznesowych. Dobrą praktyką podczas projektowania API jest separacja odpowiedzialności zaletami której jest ułatwienie testowania oraz obniżenie skomplikowania implementacji komponentów systemu. Zgodnie z tym podejściem kontrolery powinny wiedzieć wyłącznie o zawartości warstwy serwisów lub repozytoriów.

|  |  |
| --- | --- |
| Nazwa | Opis |
| AuthController | Składa się z metod do uwierzytelniania i autoryzacji użytkowników |
| ChatController | Składa się z metod do zarządzania wiadomościami dialogów |
| EventAdminsController | Składa się z metod do zarządzania administratorami wydarzeń |
| EventController | Składa się z metod do zarządzania wydarzeniami |
| EventDataController | Składa się z metod do zarządzania danymi wydarzenia |
| EventParticipantController | Składa się z metod do zarządzania uczestnikami wydarzenia |
| EventParticipationRequestController | Składa się z metod do zarządzania wnioskami o udział w wydarzeniu |
| EventPhotoController | Składa się z metod do zarządzania obrazami wydarzenia |
| EventSearchController | Składa się z metod do wyszukiwania wydarzeń |
| FriendController | Składa się z metod do zarządzania znajomymi |
| FriendshipRecommendationController | Składa się z metod do zarządzania wnioskami o dodanie do znajomych |
| PostCommentController | Składa się z metod do zarządzania komentarzami do wpisów |
| PostController | Składa się z metod do zarządzania wpisami |
| UserController | Składa się z metod do zarządzania użytkownikami |
| UserDataUpdateController | Składa się z metod do zarządzania danymi użytkowników |
| UserPhotoController | Składa się z metod do zarządzania obrazami użytkowników |
| UserPostController | Składa się z metod do zarządzania wpisami użytkowników |

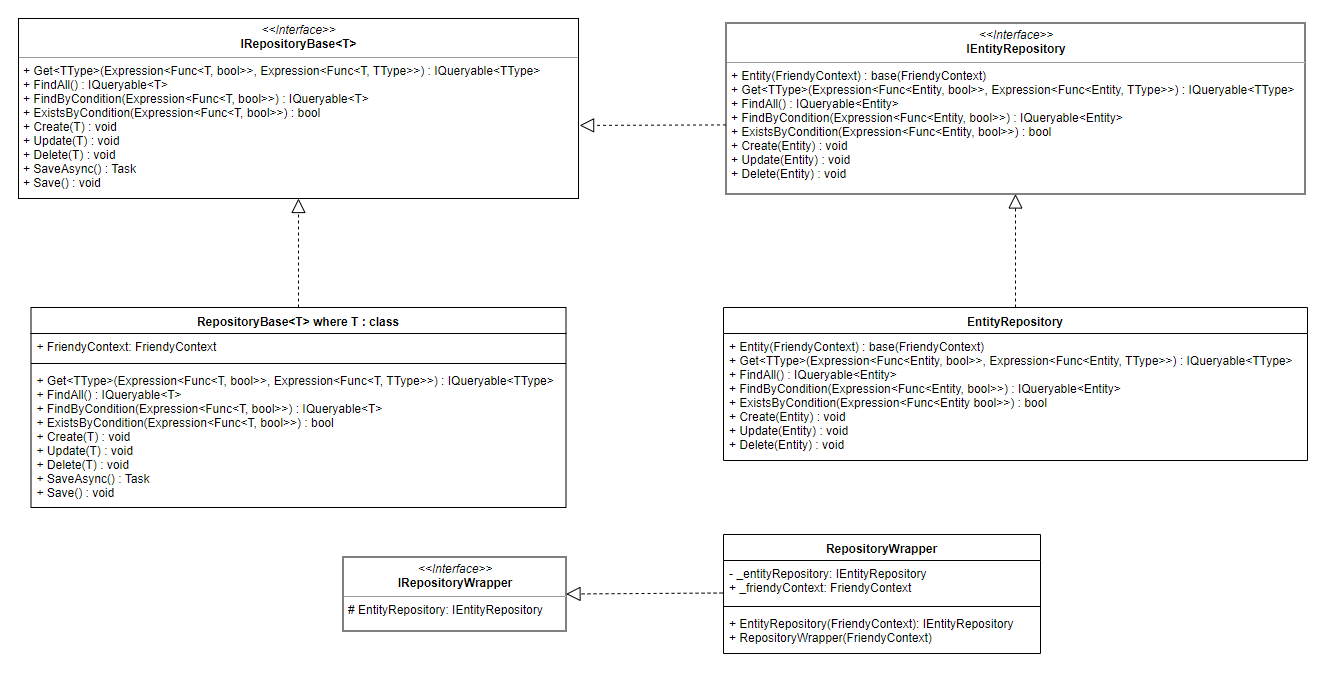
Tabela 4 Lista kontrolerów wykorzystanych w projekcie

Implementacja każdego programu komputerowego powinna przestrzegać zasad czystej architektury. Jednym z aspektów wspomagających twórców przestrzegać tych zasad są wzorce projektowe. W kontekście tworzenia REST API jednym z najbardziej pomocnych wzorców jest repozytorium. Repozytorium to wzorzec projektowy bazujący na wzorcu fasady oraz zasadę działania mechanizmu mapowania danych, który polega na oddzielaniu obiektów domenowych od implementacji bazy danych[[59]](#footnote-59). Zastosowanie wzorca repozytoriów znacznie ulepsza strukturę źródłową programu gdy składa się on z dużej ilości obiektów domenowych, a także w znacznej ilości ogranicza powtarzanie się kodu, co jest zgodne z zasadą DRY (od ang. *Don’t Repeat Yourself*). Ponadto zaletami wykorzystania repozytoriów są[[60]](#footnote-60):

* + 1. Izolacja kodu. Izolacja kodu dostępu do bazy danych sprawia, że debugowanie jest łatwiejsze
    2. Agregacja. Według zasady agregacji w DDD lub Domain Driven Design operacje usuwania i dodawania encji związanych z encjami rzędu wyższego powinny odbywać się poprzez encje rzędu wyższego – repozytorium pozwala na takie podejście
    3. Hermetyzacja skomplikowanych komend T-SQL. Przypadki wykorzystania skomplikowanych komend mogą być hermetyzowane w repozytoriach
    4. Łatwość testowania. Repozytorium jako jednostka sprawia, że testowanie jest łatwiejsze

Jedną z wad implementacji wzorca jest istnienie wcześniej wykorzystanego ORM Entity Framework Core lub NHibernate[[61]](#footnote-61), które mogą sprawić, że wykorzystanie repozytoriów zacznie wprowadzać komplikacje do kodu zamiast je likwidować. Repozytorium to wzorzec pochodzący z czasów nieistnienia ORM, kiedy to programiści musieli samodzielnie implementować szczegóły pracy z bazą danych, co skutkowało ogromnymi ilościami powtarzającego się kodu, a to stwarzało powinność opracowania warstwy hermetyzującej. Ponieważ dzisiaj programista może korzystać z rozwiązań ORM, dość często stosowanie repozytoriów może być nieracjonalne. Ale zawsze warto mieć na uwadze, iż repozytoria wprowadzają dodatkową warstwę abstrakcji między logiką domenową a bazą danych, co jest bardzo ważne z punktu widzenia architektury.

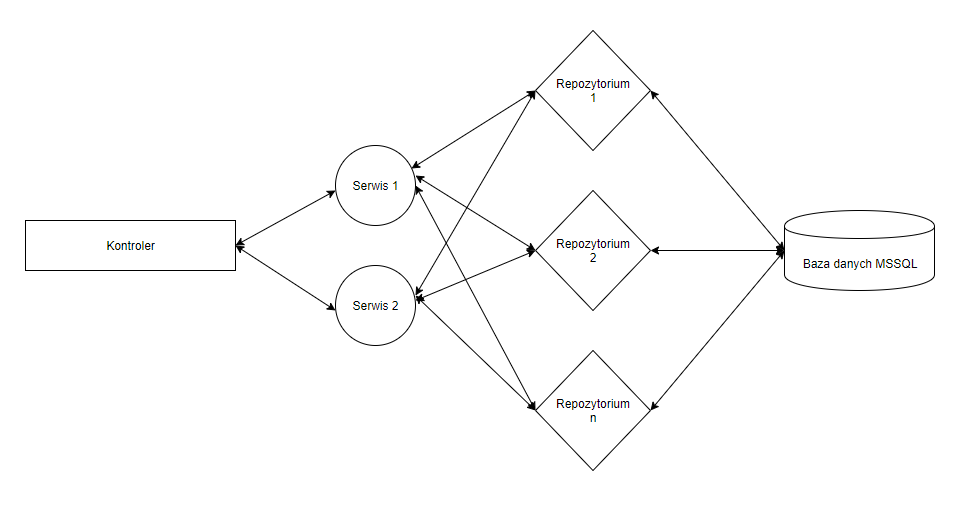
Wzorzec repozytoriów można zaimplementować na różne sposoby – w tworzonym projekcie jednak zostanie zaimplementowany wzorzec abstrakcyjnego generycznego repozytorium[[62]](#footnote-62). Implementację repozytorium w tworzonej aplikacji można przedstawić poprzez następujący diagram klas.



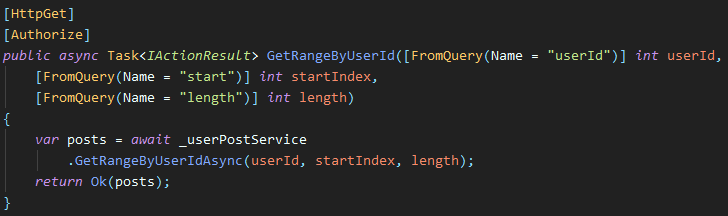
Rysunek 15 Diagram klas przedstawiający implementację wzorca repozytorium w tworzonej aplikacji

Tutaj IRepositoryBase<T> jest interfejsem generycznym skupiającym deklaracje metod, które powinny zostać zaimplementowane przez abstrakcyjną generyczną klasę RepositoryBase<T> której możliwa wartość parametru T jest ograniczona do typu klasa. IEntityRepository definiuje interfejs implementujący IRepositoryBase<T> z perspektywy konkretnego obiektu domenowego. EntityRepository jest bezpośrednią implementacją metod interfejsu IEntityRepository. Dwoma ostatnimi ważnymi składnikami wzorca jest klasa RepositoryWrapper, która jest odpowiedzialna za deklarację poszczególnych interfejsów i przypisywanie im wartości oraz interfejs IRepositoryWrapper, który jest swoistą warstwą dostępu do repozytoriów dla serwisów, zawierającą deklaracje wszystkich repozytoriów w systemie.

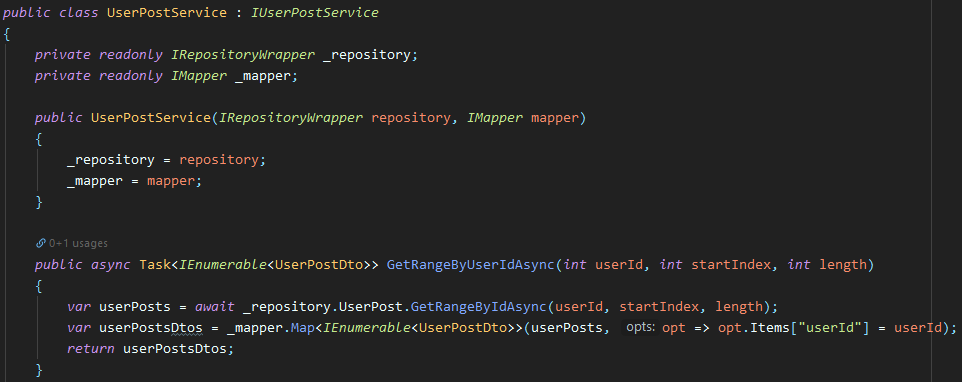
Dopełnieniem do repozytoriów są serwisy, których zadaniem powinna być implementacja logiki biznesowej oraz separacja szczegółów implementacji od kontrolerów. Komunikację przy zastosowaniu takiego podejścia można przedstawić poprzez następujący diagram oraz wycinki kodu z każdego komponentu.



Rysunek 16 Komunikacja pomiędzy komponentami przy zastosowaniu wzorca repozytoriów i serwisów. Źródło: opracowanie własne

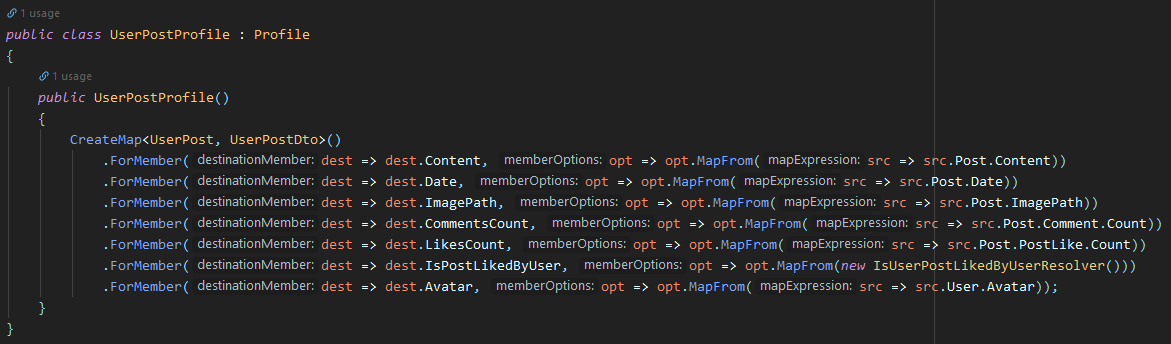


Rysunek 17 Metoda w kontrolerze UserPostController obsługująca zapytanie o konkretną liczbę wpisów użytkownika



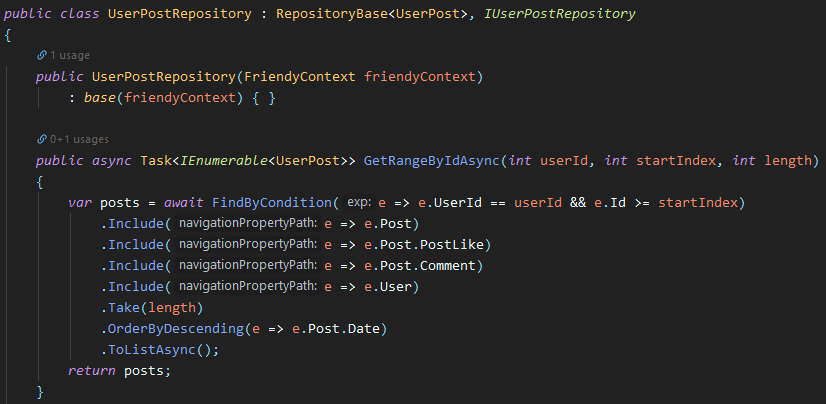
Rysunek 18 Wycinek serwisu UserPostService obsługujący logikę biznesową związaną z pobieraniem danych z repozytorium oraz przekształcaniem wyniku do postaci klasy UserPostDto

W serwisie UserPostService została użyta biblioteka Automapper[[63]](#footnote-63), służąca do mapowania obiektów jednego typu do innego. Użycie tej biblioteki sprawia, że obowiązek jawnego mapowania elementów jednej klasy do innej znika dopóki przestrzega się ustalonej konwencji mapowania przez bibliotekę, na przykład nazwy zmiennych jednej klasy powinny być takie same jak nazwy zmiennych innej. Jednak w tym wypadku należy utworzyć specjalny profil mapowania i zarejestrować go w projekcie.



Rysunek 19 Profil mapowania obiektu klasy UserPost do UserPostDto

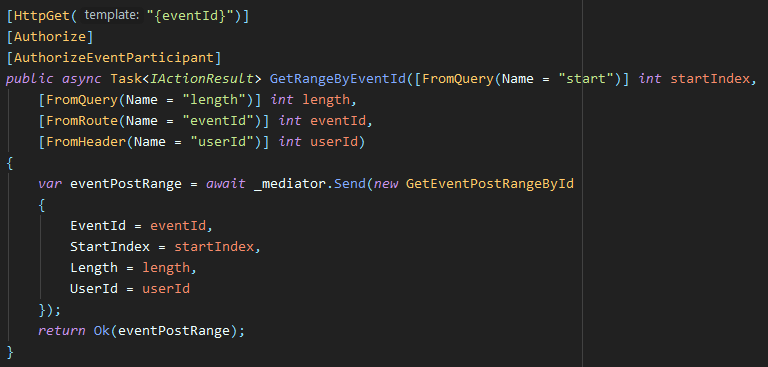
Następujący rysunek przedstawia fragment repozytorium UserPostRepozytory z metodą GetRangeByIdAsync, która jest metodą asynchroniczną bezpośrednio zwracającą wcześniej odpytane dane. Tutaj warto zwrócić uwagę na obecność wstrzykniętego przez konstruktor do klasy nadrzędnej kontekstu bazy danych – to właśnie poprzez ten kontekst odbywa się odczytywanie danych z bazy danych.



Rysunek 20 Fragment repozytorium UserPostRepository wraz z metodą pobierającą niezbędne dane o wpisie z bazy danych

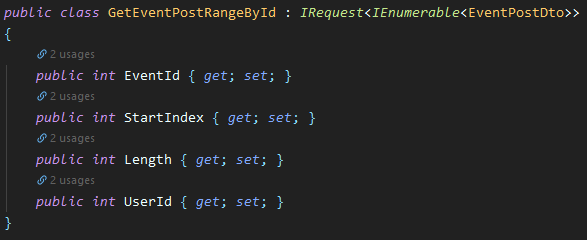
Innym podejściem do organizacji warstwy domenowej jest używanie wzorca CQRS. Ten wzorzec opiera się o wzorzec wynaleziony przez Bertanrd’a Meyer’a o nazwie *Command Query Responsibilty*[[64]](#footnote-64), którego działanie opiera się na stwierdzeniu, że każda metoda powinna być komendą wykonującą czynność lub zapytaniem zwracającym dane do wywołującego. Fundamentalną różnicą pomiędzy tymi wzorcami jest powinność dzielenia obiektów na interfejsy, z czego pierwszy służy do przechowywania zapytań, a drugi do przechowywania komend. Uważa się, że ten wzorzec jest jedną z alternatyw wzorca repozytorium, chociaż nie jest to zbytnio racjonalne podejście ponieważ likwiduje czynnik enkapsulacji, na przykład w kontekście pochodzenia danych. Architekturę CQRS można zaimplementować na różne sposoby, na przykład ręcznie lub z użyciem biblioteki MediatR[[65]](#footnote-65) implementującej wzorzec projektowy Mediator[[66]](#footnote-66), który służy do zdefiniowania obiektu hermetyzującego interakcję w grupie obiektów, co jest bardziej racjonalnym rozwiązaniem ponieważ likwiduje duże ilości powtarzającego się kodu.

Wykorzystywanie wzorca w kontrolerze odbywa się poprzez wstrzykiwanie interfejsu IMediator. Przykład wykorzystania wzorca przedstawia poniższy rysunek.



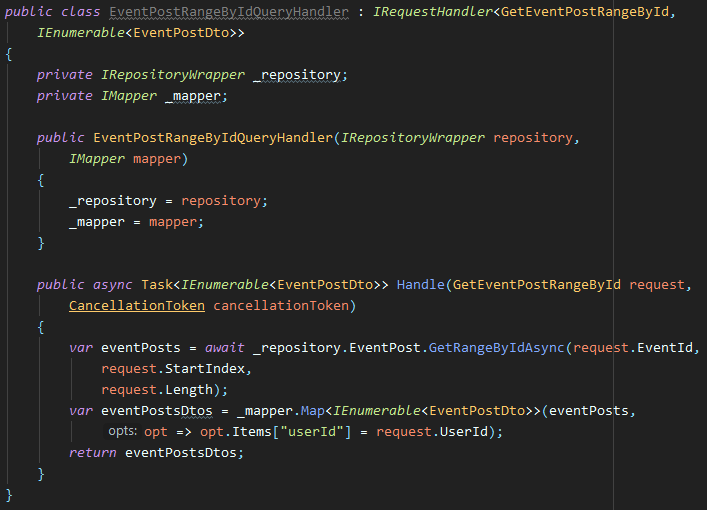
Rysunek 21 Wycinek kodu przedstawiający metodę odpowiadającą za przetwarzanie zapytania o wyznaczoną listę wpisów na podstronie wydarzenia

W tym wypadku poprzez mediator zostanie wykonane zapytanie o nazwie GetEventPostRangeById, które opisuje model do odczytu przedstawiony na poniższym rysunku.



Rysunek 22 Model GetEventPostRangeById

Każdy tworzony model do odczytu powinien implementować interfejs generyczny IRequest przyjmujący jako parametr typ, który powinien zostać zwrócony przez Handler, czyli klasę zwracającą odpytane dane do kontrolera przedstawioną na poniższym rysunku.



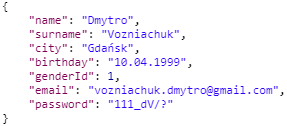
Rysunek 23 Handler obsługujący odczytywanie wpisów na stronie wydarzenia w oparciu o podany model do odczytu

Dostęp do bazy danych z poziomu handlera może odbywać się poprzez repozytorium lub wstrzykiwanie kontekstu bazy danych. W tym wypadku jednak racjonalniejszym podejściem jest wykorzystanie już zaimplementowanej logiki odpytywania danych wpisów w repozytorium.

# 3.3. Proces uwierzytelniania i autentykacji użytkowników

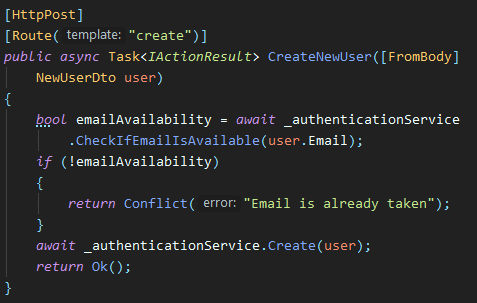
Implementację procesu rejestracji użytkowników należy rozpocząć od ustalenia danych, które powinno się pozyskiwać od użytkowników w czasie rejestracji. Głównymi składnikami procesu rejestracji są hasło oraz email. Ponieważ tworzonym rozwiązaniem jest sieć społecznościowa oprócz powyższych dwóch ważną częścią procesu rejestracji powinny być także inne podstawowe dane użytkownika: imię, nazwisko, wiek, miasto zamieszkania oraz płeć.

Rejestracja użytkownika będzie przebiegała z poziomu osobnej podstrony na której zostanie umieszczony formularz składający się z pól tekstowych oraz przycisków radiowych (Rysunek 27). Po wypełnieniu oraz zatwierdzaniu formy podane dane zostaną wysłane do API pod adres api/auth/create w postaci poniższego obiektu JSON.



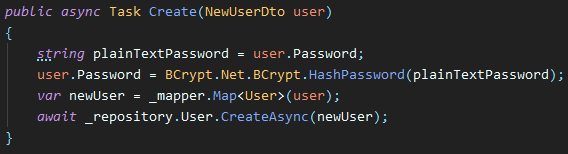
Rysunek 24 Obiekt JSON zawierający podstawowe dane nowego użytkownika

W tworzonym projekcie AuthController odpowiada za przetwarzanie zapytań związanych z uwierzytelnianiem, autoryzacją i rejestracją nowego użytkownika. Poniższy wycinek kodu przedstawia metodę kontrolera, która przetwarza rejestrację nowego użytkownika.



Rysunek 25 Metoda w kontrolerze AuthController przetwarzająca rejestrację nowego użytkownika

Po aktywacji metody zostanie użyty osobny serwis, który zawiera metodę sprawdzającą dostępność przesłanego adresu email. Po tym etapie cały obiekt z danymi tworzonego użytkownika zostanie przesłany do innej metody tego samego serwisu, która bezpośrednio odpowiada za tworzenie użytkownika. Implementacja tej metody jest przedstawiona na rysunku poniżej.

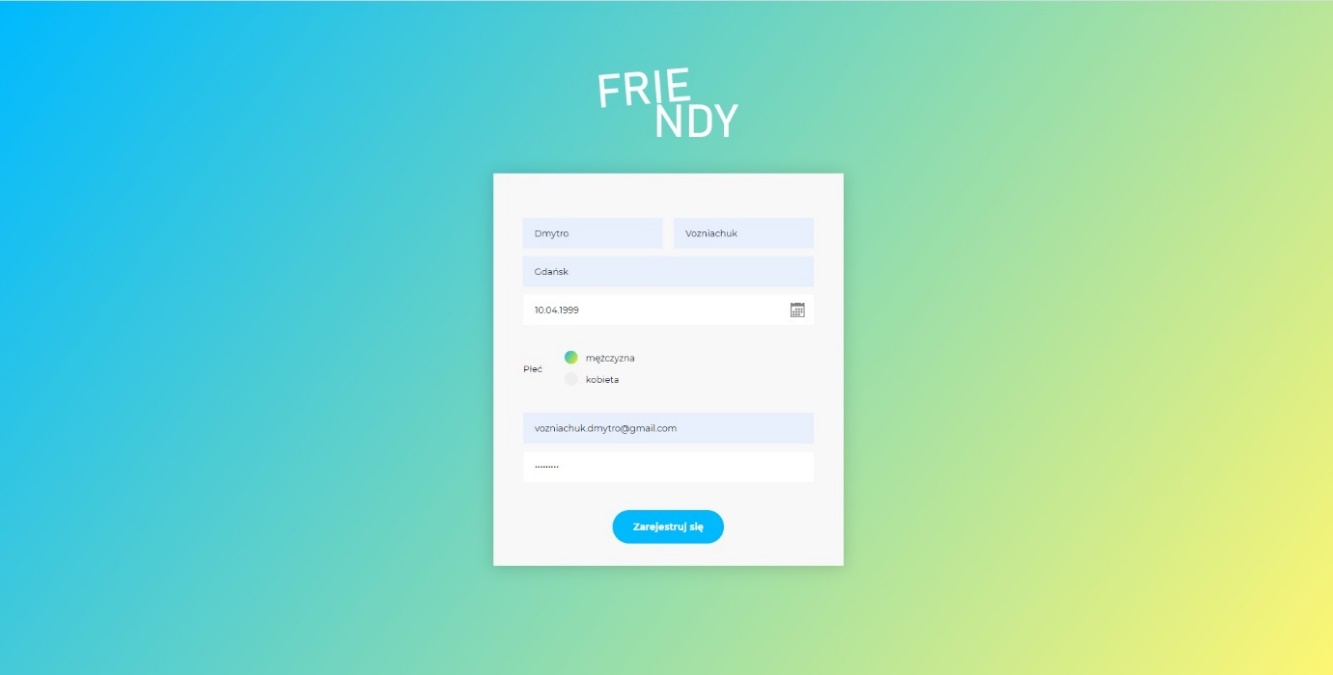


Rysunek 26 Metoda w serwisie AuthenticationService odpowiadająca za rejestrację nowego użytkownika

W przedstawionej powyżej metodzie została wykorzystana implementacja algorytmu BCrypt z biblioteki BCrypt.Net[[67]](#footnote-67). Algorytm ten jest oparty o algorytm Blowfish, a działa w oparciu o zastosowaną „sól” oraz kosztowne wydajnościowo generowanie kluczy algorytmu Blowfish, które zostało zmodyfikowane na tyle aby być jeszcze bardziej kosztowne, z czego ma wynikać bardzo mała szansa złamania hasła[[68]](#footnote-68). W tym wypadku biblioteka automatycznie wygeneruje „sól”, chociaż biblioteka pozwala na podanie własnej „soli” jako następny parametr wywołanej funkcji HashPassword(…). Wygenerowany hasz wygląda następująco:

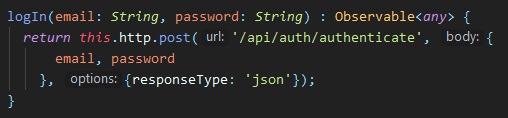
*$2b$10$49ajmnLwVC1wcOVJbOtmBuKk5FnxubGb.4U7eVLj1sP33g1Hx3vA6*

Po rejestracji użytkownik ma możliwość zalogowania się o czym zostaje on powiadomiony poprzez wiadomość w pojawiającym się oknie na stronie rejestracji.



Rysunek 27 Strona rejestracji nowego użytkownika

Jak wspomniano wcześniej schemat uwierzytelniania w aplikacji będzie oparty o JSON Web Token*.* W kontekście ASP.NET Core operować na takich ciągach można poprzez własną implementację algorytmu tworzenia i przetwarzania JWT lub korzystając z gotowego rozwiązania poprzez dodanie do projektu przestrzeni nazw System.IdentityModel.Tokens.Jwt*.* Każdy token ma własny termin ważności, czyli czas przez który użytkownik będzie mógł korzystać z zasobów używając tego samego tokenu. W ramach tej pracy standardowy termin ważności tokenu będzie ustawiony na 7 dni. Autentykacja użytkownika w tworzonej aplikacji będzie się odbywała z poziomu podstrony o adresie /loginna której użytkownik będzie winien wprowadzić swój email oraz hasło. Poniższa ilustracja przedstawia implementację tego zapytania w Angularze.

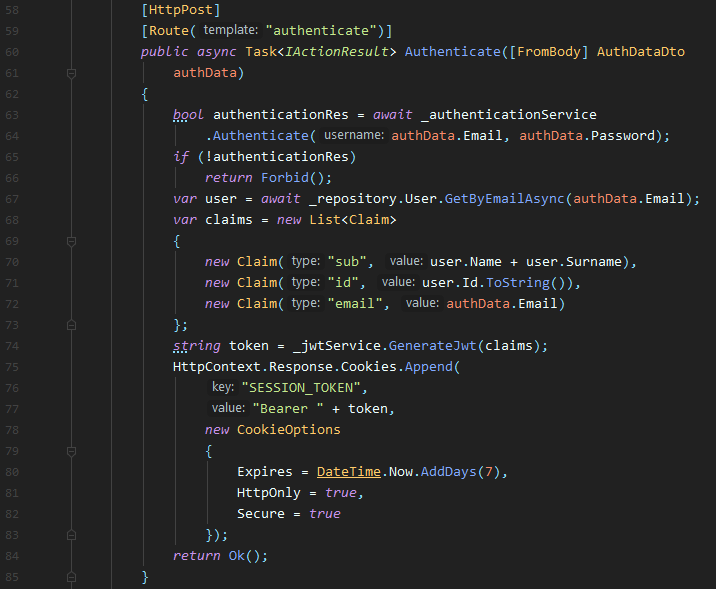


Rysunek 28 Implementacja zapytania POST do REST API o poprawność wprowadzonych danych logowania

Po wprowadzeniu tych danych do API pod adres /api/auth zostanie wysłane zapytanie POST z ciałem zawierającym obiekt JSON o poniższej postaci

{  
 login: [wprowadzony login],  
 password: [wprowadzone hasło]  
}

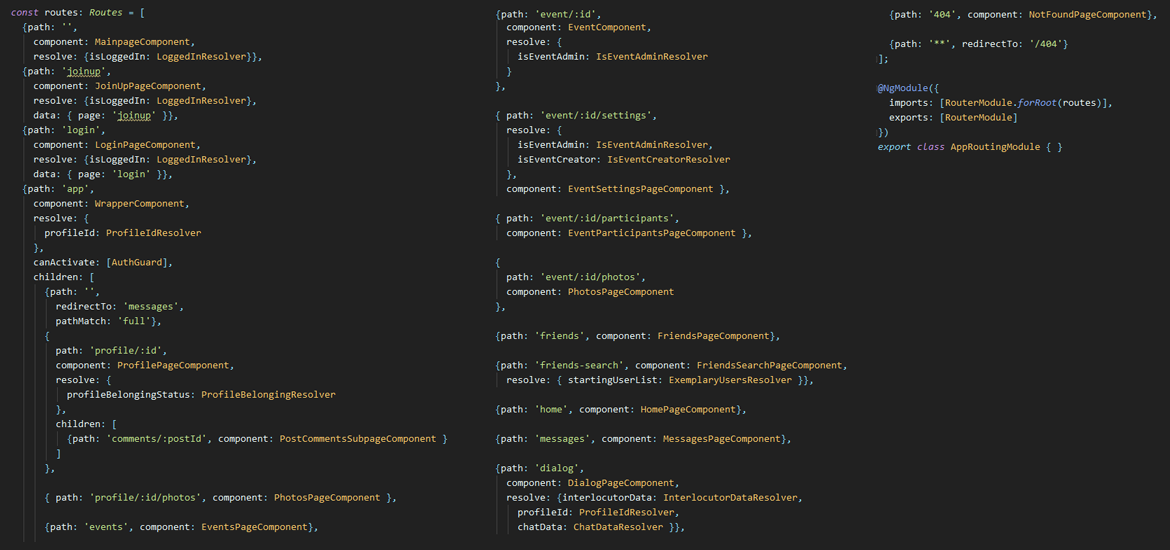
Przetwarzanie tego zapytania przez odpowiednią metodę kontrolera reprezentuje poniższy fragment kodu. W tym wypadku została użyta metoda Authenticate z odrębnego serwisu AuthenticationService sprawdzająca wprowadzone hasło i email przez użytkownika – jeżeli owa metoda zwróci wartość false to oznacza to, iż dane nie są prawidłowe lub w przypadku zwrócenia wartości true jest generowany JWT w oparciu o przekazaną zawartość, tutaj zawartością jest lista obiektów klasy Claim. Po wygenerowaniu JWT do odpowiedzi zostaje dodane wcześniej wspomniane bezpieczniejsze ciasteczko o fladze HTTPOnly zawierające wcześniej wygenerowany token i zwracana jest odpowiedź HTTP 200.



Rysunek 29 Ilustracja fragmentu kodu odpowiadającego za rejestrację nowego użytkownika

# 3.4. Routing w aplikacji

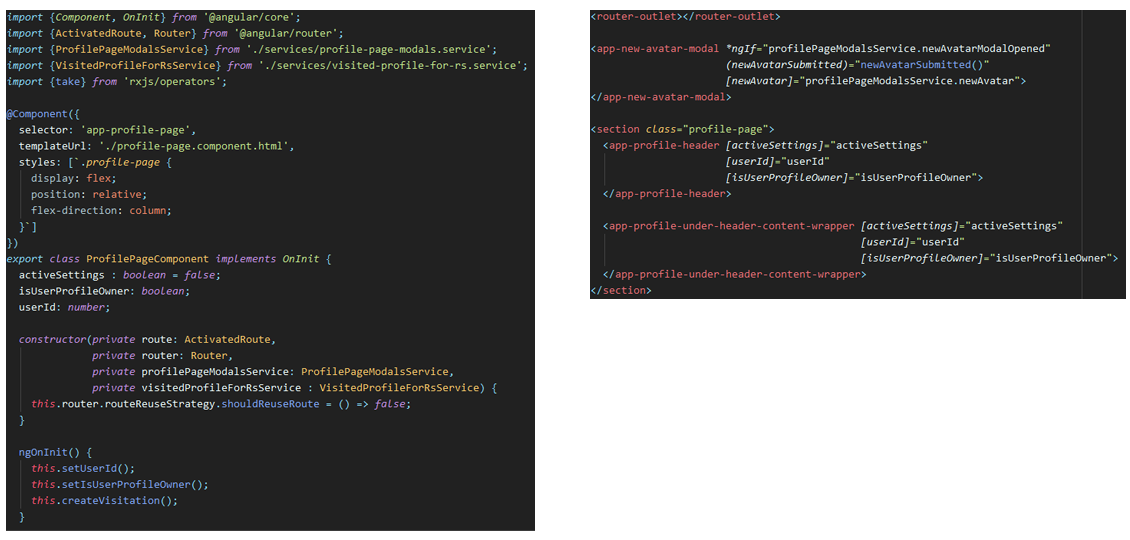
Po zalogowaniu się użytkownik zostaje przekierowany do głównej zabezpieczonej części aplikacji pod adresem /app składającej się z kilku podstron. Routing w Angularze działa podobnie do tradycyjnego routingu, ale zamiast odświeżenia całej strony odświeża się tylko komponent odpowiadający wybranemu adresowi podstrony. Angular, podobnie jak inne frameworki i biblioteki aplikacji jednostronicowych, korzysta z natywnego History API[[69]](#footnote-69) w celu dostarczenia funkcjonalności nawigacji tak jak w tradycyjnych aplikacjach. Routing w aplikacji jest definiowany przez moduły – tutaj definiowanie routingu składa się z jednego modułu, choć mogą one zostać podzielone na mniejsze moduły. Każdy moduł definiujący routing powinien importować moduł RouterModule udostępniający metodę forRoot lub forChild przyjmującą tablicę typu Routes zawartością której powinny być obiekty definiujące routing.



Rysunek 30 Moduł konfiguracji routingu

# 3.5. Implementacja przykładowej podstrony

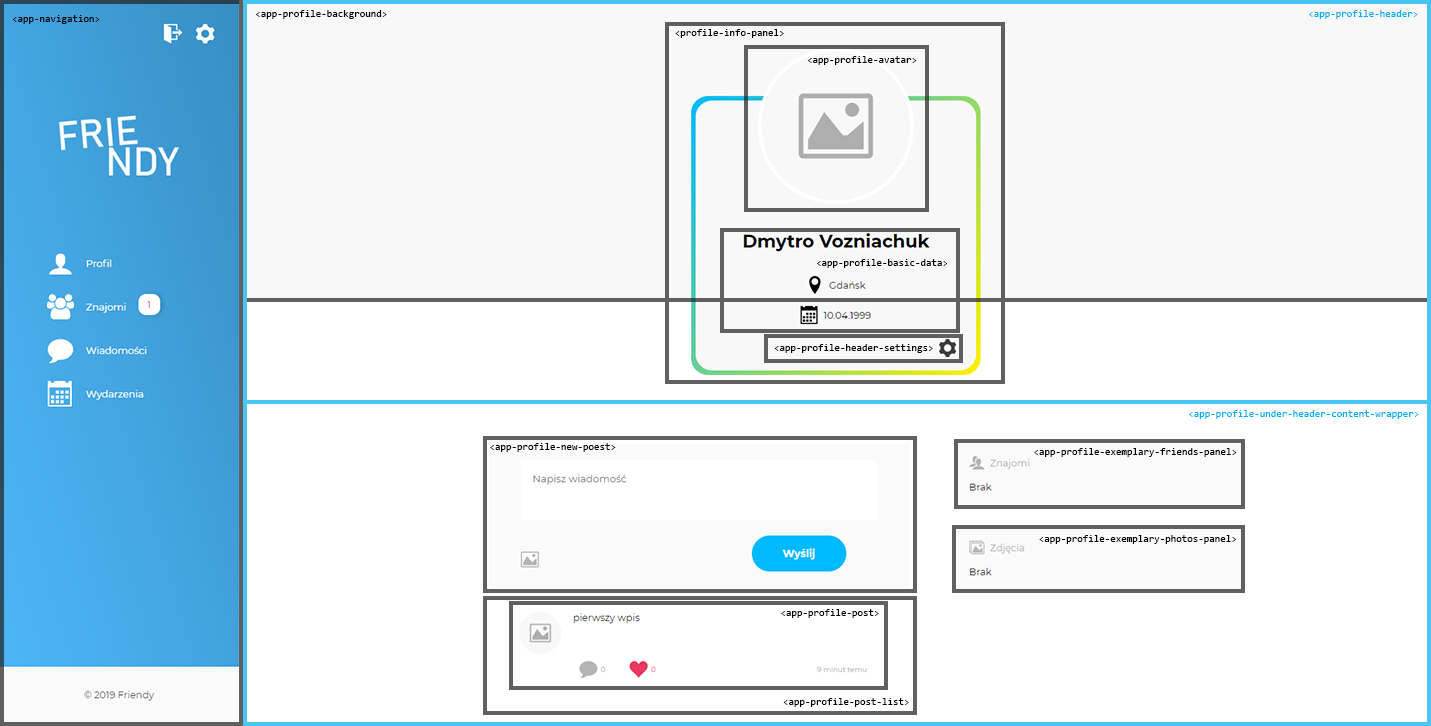
Tak jak wspomniano narzędzia do tworzenia aplikacji jednostronicowych opierają się o komponenty. Każdy komponent powinien składać się z przynajmniej trzech części: widoku, skryptów oraz styli.



Rysunek 31 Skrypty i kod HTML komponentu ProfilePageComponent

Każda strona lub podstrona lub moduł w aplikacji jednostronicowej sama z siebie jest komponentem obejmującym inne komponenty w obrębie tego samego modułu. Jedną z najważniejszych podstron jest profil użytkownika, który może posłużyć dobrym przykładem do przedstawienia struktury podstrony. Następny rysunek przedstawia główne składniki tego komponentu. Jak widać komponent jest opisywany klasą implementującą interfejs OnInit – jest to jeden z interfejsów lub tak zwanych hooków umożliwiających korzystanie z cyklu życia komponentu. Najczęściej wykorzystywanymi hookami w Angularze są[[70]](#footnote-70):

* 1. ngOnChanges() jest wywoływane gdy zmienia się wartość referencji do właściwości in-bound oraz przed ngOnInit()
  2. ngOnInit() jest wywoływane raz po inicjalizacji komponentu
  3. ngDoCheck() jest wywoływane po zmianach których nie może obsłużyć Angular
  4. ngOnDestroy() jest wywoływane przed zniszczeniem komponentu lub dyrektywy



Rysunek 32 Podstrona profilu użytkownika podzielona na komponenty

# 3.6. Wyszukiwanie użytkowników

Jedną z funkcjonalności tworzonej aplikacji jest wyszukiwarka zarejestrowanych użytkowników według kryteriów. Jednym z najlepszych rozwiązań do przeszukiwania danych zgodnie z ustalonymi parametrami jest przedstawiony wcześniej Elasticsearch. Dokumenty zawierające dane użytkowników do przeszukiwania będą przechowywane w indeksie o nazwie users. Przykładowy dokument w formacie JSON może wyglądać następująco.



Rysunek 33 Przykładowy dokument zawierający dane przeznaczone do wyszukiwania użytkowników

Po ekstrakcji i przeniesieniu rzeczywistych danych użytkowników z głównej bazy danych można przetestować działanie klastra. Do tego celu można użyć narzędzia Postman, które pozwala wysyłać zapytania HTTP o różnych konfiguracjach. Do opracowania i prawidłowego działania tego zapytania można użyć filtra bool, który jest używany do łączenia wielu klauzul filtracyjnych z użyciem logiki boolowskiej[[71]](#footnote-71).

|  |  |
| --- | --- |
| Nazwa | Opis działania |
| must | Wszystkie klauzule powinny odpowiadać parametrom wyjściowym |
| must\_not | Wszystkie klauzule nie powinny odpowiadać parametrom wyjściowym |
| should | Przynajmniej jedna klauzula powinna odpowiadać parametrom wyjściowym |

Tabela 5 Opis parametrów akceptowanych przez filtr boolowski

Przykładowe ciało zapytania może wyglądać następująco:

{

  "query" : {

        "bool": {

            "must": [

               {

                   "match\_phrase": {

                      "name": "Dmytro"

                   }

               },

               {

                   "match\_phrase": {

                      "surname": "Vozniachuk"

                   }

               },

               {

                   "match\_phrase": {

                      "city": "Gdańsk"

                   }

               },

               {

                    "range":{

                        "birthday": {

                            "format": "dd.mm.yyyy",

                            "gt": "10.12.1998",

                            "lt": "11.12.2000"

                        }

                    }

               },

               {

                "match": {

                    "userInterests.id": 1

                }

                },

                {

                    "match": {

                        "userInterests.id": 2

                    }

                }

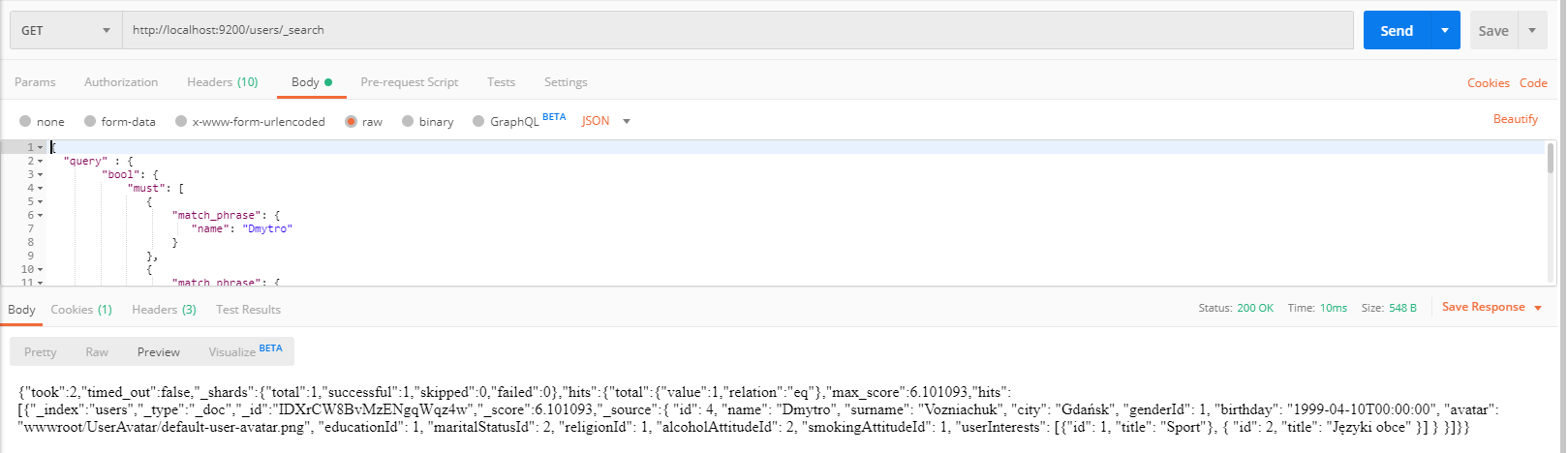
            ]

        }

    }

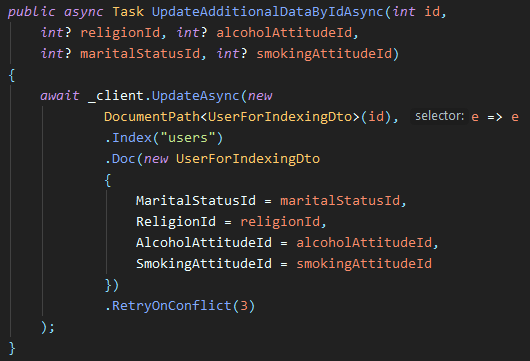
}

Odpowiedź serwera jest przedstawiona w sekcji o nazwie ciało na poniższej ilustracji z programu Postman.



Rysunek 34 Odpowiedź serwera ES w programie Postman

Takie podejście do projektu wyszukiwarki rodzi jednak problemy polegające na tym, że w razie zmiany danych przez użytkowników oprócz wprowadzania zmian do bazy danych SQL Server te dane powinny zostać zmienione w indeksie Elasticsearch. Właśnie tutaj zachodzi konieczność skorzystania z wspomnianego wcześniej klienta NEST. Przykładową aktualizację dodatkowych danych użytkownika z użyciem tego klienta przedstawia następujący rysunek.



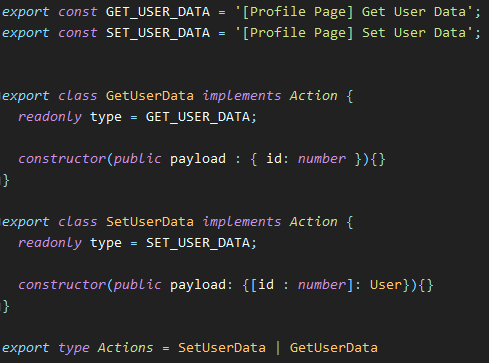
Rysunek 35 Fragment kodu przedstawiający aktualizację danych w klastrze Elasticsearch z użyciem klienta NEST

# 3.7. Zarządzanie stanem aplikacji

NgRx oferuje 7 pakietów do zarządzania stanem. Najczęściej wykorzystywanymi pakietami w tworzonej aplikacji będą:

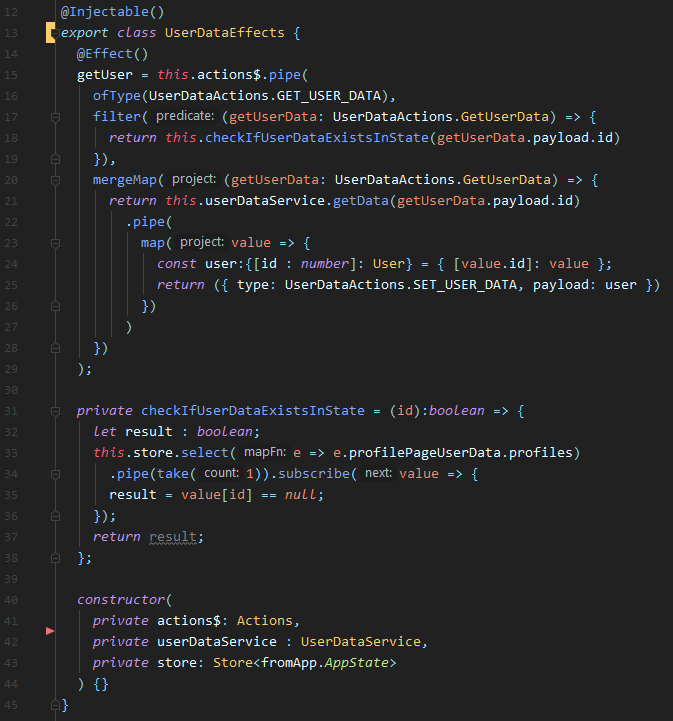
1. @ngrx/store jest przeznaczony do zarządzania stanem
2. @ngrx/effects dostarcza funkcjonalności zwane efektami, które są meta reduktorami obsługującymi wywołane przez dyspozytor akcje

Przykładem implementacji NgRx może posłużyć komponent, który wyświetla podstawowe dane użytkownika na stronie profilowej. Pierwszym i najważniejszym składnikiem są reduktory, które są odpowiedzialne za przeprowadzenie zmian w stanie. Reduktor składa się z funkcji reduktora, które odpowiadają za określenie zdefiniowanej funkcji przypisanej do jednego ze zdefiniowanych wcześniej typów akcji[[72]](#footnote-72).



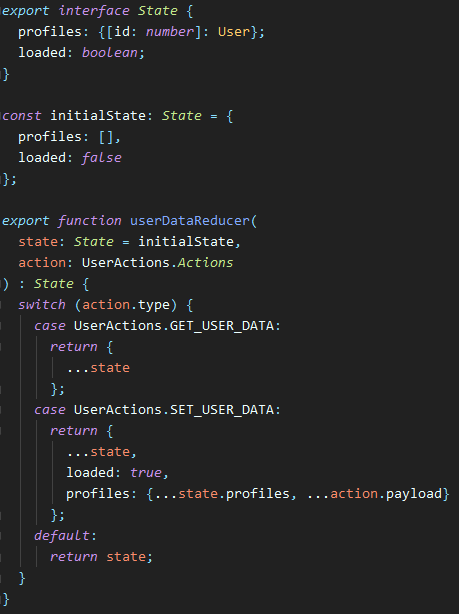
Rysunek 36 Typy akcji dla reduktora danych użytkownika

Powyższy fragment kodu przedstawia dwa typy akcji dla reduktora danych użytkownika. Każdy z typów jest osobną klasą implementującą interfejs Action zawierającą zmienną, której wartość wyznacza typ akcji. Ponadto każda klasa zawiera konstruktor, którego parametrem jest zmienna payload poprzez którą są przekazywane parametry niezbędne do dalszego zarządzania stanem. Klasa GetUserData odpowiada za aktywacje efektu wysyłającego zapytanie do API o dane użytkownika w oparciu o przekazany identyfikator, zaś klasa SetUserData oznacza akcję dodawania pobranych danych do stanu. Przedstawione typy akcji są związane ze sobą w efektach.



Rysunek 37 Efekty dla stanu danych użytkownika

Praca z efektami opiera się o bibliotekę rxjs. Do tworzenia efektów służy wstrzyknięta poprzez konstruktor prywatna zmienna actions$ typu Actions na której można nasłuchiwać wywołania poszczególnych efektów. Całość przedstawianej implementacji składa się tylko z jednego efektu. Tutaj funkcja ofType oznacza typ akcji na który powinien reagować efekt, funkcja filter poprzez użycie osobnej funkcji sprawdza czy dane konkretnego użytkownika są już zawarte w stanie zanim wykona się funkcja mergeMap, której zadaniem jest wykonanie zapytania o dane użytkownika do API oraz dodanie tych danych w wywołaniu zwrotnym do stanu. Warto zaznaczyć, że podchodzić do problemu przechowywania obiektów o identycznym typie należy opierając się o typ zawierający dodatkowy identyfikator i obiekt, co skutkuje złożonością obliczeniową O(1) zamiast O(N) gdyby się przeszukiwało po kolei każdy obiekt.



Rysunek 38 Stan i reduktor danych użytkownika

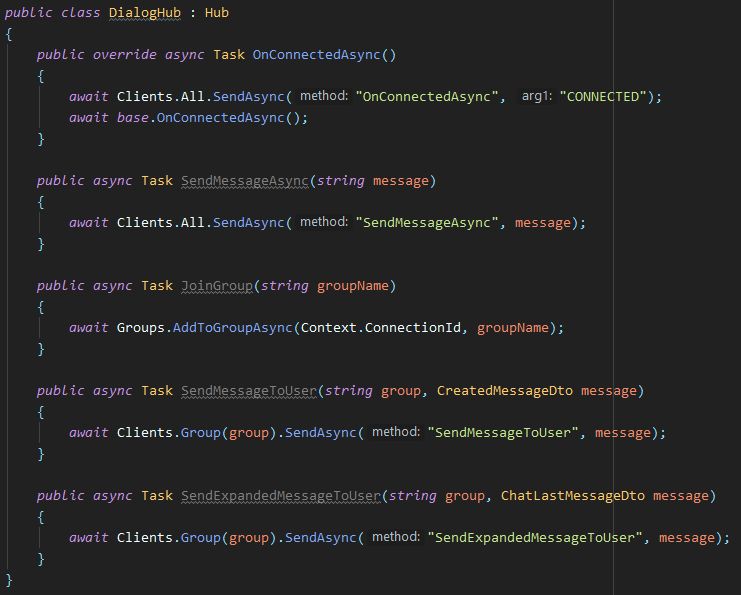
Każdy reduktor jest funkcją przyjmującą dwa parametry: stan i akcja o określonym typie. W przedstawionym przykładzie instrukcja switch obsługuje typy akcji z poprzednio przedstawionego fragmentu kodu. Jak widać każdy przypadek zwraca jakiś wynik jako wynik funkcji reduktora – oznacza to aktualizowanie stanu do obiektu zwróconego przez tę funkcję.

Później w komponencie lub serwisie dane ze stanu mogą być pobierane i wyświetlane w widoku.

# 3.8. Elementy aplikacji czasu rzeczywistego

Dotychczas działanie funkcjonalności aplikacji opierające się o protokół HTTP nie było dynamiczne: potrzebne dane były pobierane z ręcznym wysyłaniem żądań HTTP do API. Jedną z ważniejszych części tworzonej aplikacji jest czat działający w trybie czasu rzeczywistego. Do rozwiązania takich zadań ASP.NET oferuje wspomnianą bibliotekę SignalR.

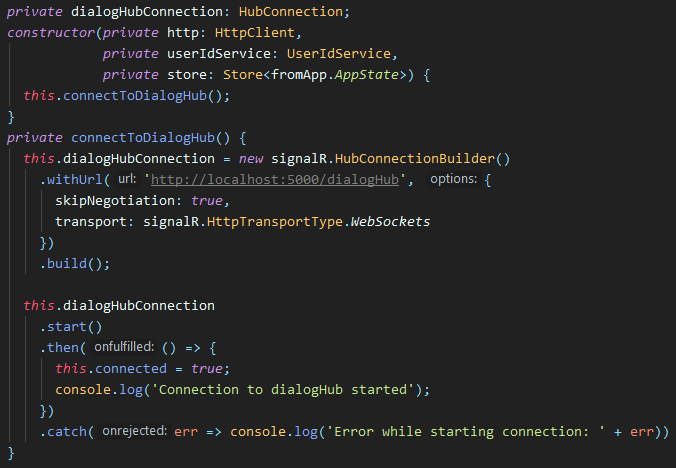
Przede wszystkim należy utworzyć oddzielny hub, który będzie określał czat i jego funkcjonalności czasu rzeczywistego. Takim hub’em może być oddzielna klasa dziedzicząca po klasie Hub pochodzącej z pakietu biblioteki.



Rysunek 39 Hub obsługujący pracę z wiadomościami

Możliwe będzie wysyłanie wiadomości do jednego znajomego naraz – do tego w hubie posłużą dwie metody SendMessageToUser oraz SendExpandedMessageToUser. Pierwsza metoda będzie przesyłała do klienta dane nowej wiadomości o krótszej zawartości wykorzystywanej bezpośrednio na podstronie czatu bazując na identyfikatorze grupy do której lub których użytkownik jest dodany, a druga będzie przesyłała je w postaci innego większego obiektu, a wyświetlać się one będą na podstronie nowych wiadomości.

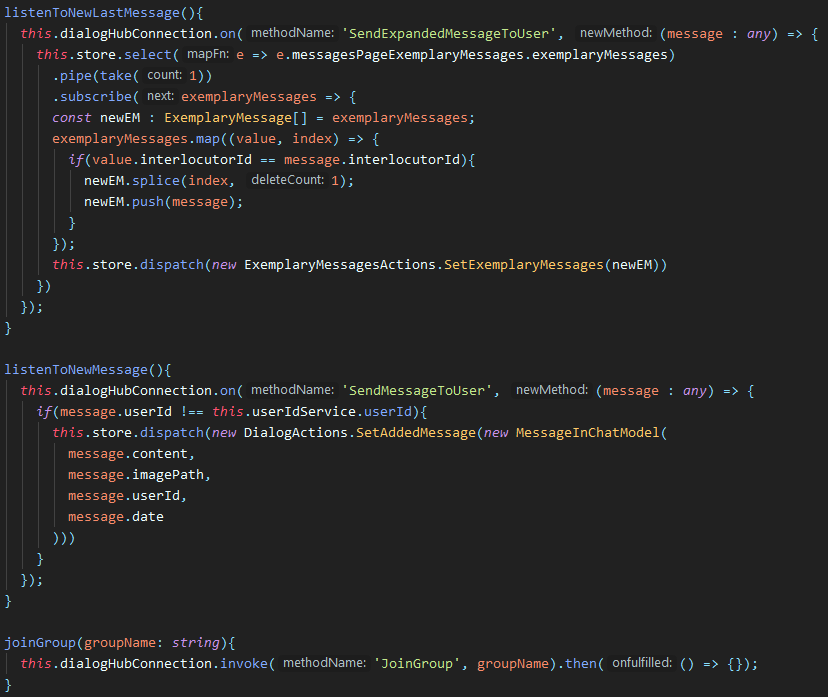
Po implementacji hubu pojawia się możliwość nawiązania połączenia przez klienta. Opracowując rozwiązanie oparte o SignalR w Javascript lub Typescript należy skorzystać z biblioteki @aspnet/signalr.



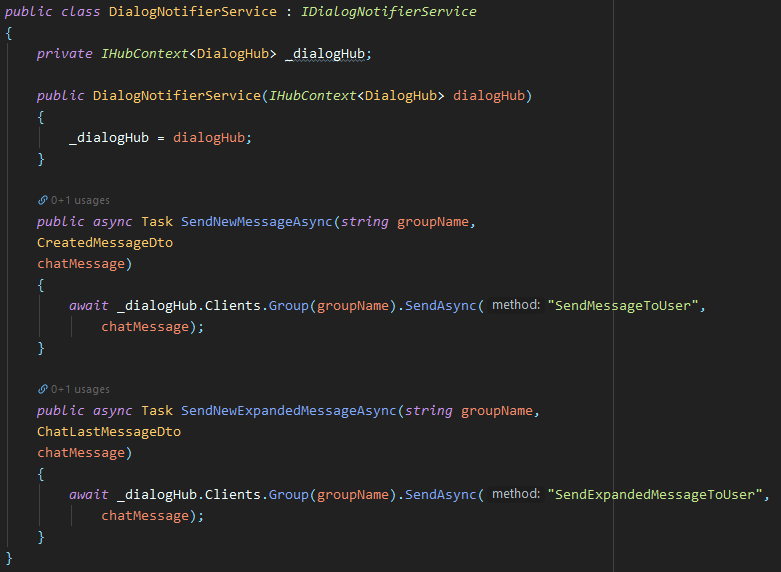
Rysunek 40 Fragment kodu przedstawiający implementację nawiązania połączenia z serwerem

Jak widać połączenie z hubem na serwerze opisuje zmienna typu HubConnection, której wartość jest inicjalizowana poprzez budowniczego w konstruktorze. Metoda withUrl tutaj przyjmuje dwa parametry: URL serwera i parametry połączenia. Negocjacja w kontekście SignalR oznacza automatyczne wybranie metody transportu, także tym wypadku parametry połączenia mogłyby zostać pominięte ponieważ SignalR zawsze automatycznie wybiera połączenie z użyciem protokołu WebSocket[[73]](#footnote-73), ponieważ jest on nowszym i właściwie obecnie najczęściej używanym kanałem komunikacji, wspieranym przez wszystkie nowsze wersje przeglądarek oraz jest to jedyny protokół komunikacji wśród dostępnych działający w trybie full-duplex przez jedno połączenie TCP (od ang. *Transmission Control Protocol*)[[74]](#footnote-74), choć zawsze jest możliwość wyboru innego protokołu.

Po nawiązaniu połączenia można nasłuchiwać dane pochodzące z metod utworzonych w hubie. Następujący rysunek przedstawia implementacje trzech metod z kontekstu czatu odpowiadające za pobieranie i przetwarzanie nowych wiadomości lub wywołanie metody na serwerze: pierwsze dwie metody otrzymują, przetwarzają i dodają dane do stanu, a trzecia metoda odpowiada za wywołanie metody na serwerze z podaniem nazwy grupy jako parametr, która doda to połączenie do określonej grupy odbiorców wiadomości (w tym wypadku maksymalnie dwojga). Trzecia metoda jest wywoływana przez klienta gdy użytkownik wejdzie na podstronę czatu.



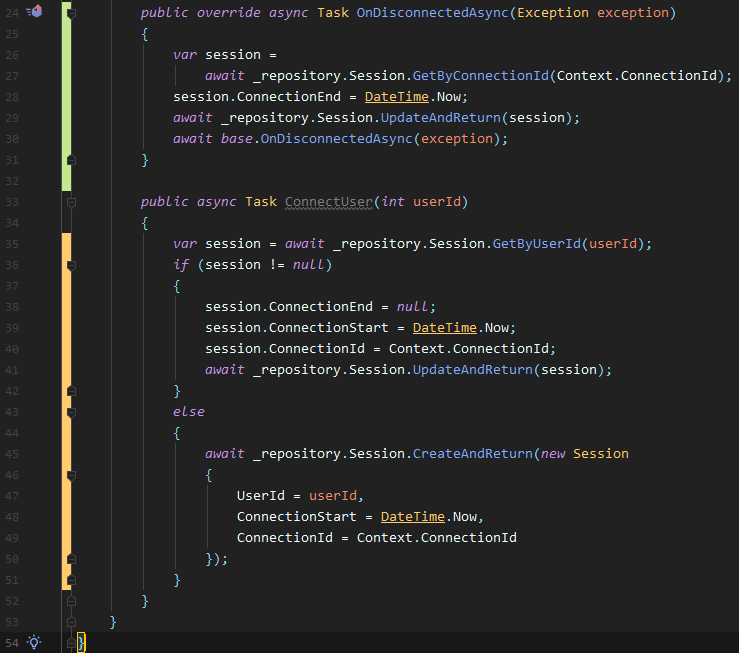
Rysunek 41 Fragment kodu przedstawiający metody nasłuchiwania i wywołania SignalR w aplikacji



Rysunek 42 Implementacja serwisu wysyłającego wiadomości do klienta

Wysyłanie wiadomości z serwera będzie odbywało się z poziomu serwisu DialogNotifierService do którego została dodana zmienna określająca Hub poprzez generyczny interfejs IHubContext przyjmujący jako parametr klasę DialogHub. Zostało tutaj zaimplementowane wysyłanie dwóch typów wiadomości do klienta poprzez dwie metody wykorzystujące wcześniej zdefiniowane metody w hubie. Te metody mogą być używane w każdym miejscu aplikacji, lecz oczywiście są one w tym wypadku przeznaczone do kontrolera ChatController w którym zostaną one użyte w metodzie przetwarzającej dodawanie nowej wiadomości.

Innym zastosowaniem SignalR w aplikacji jest wykorzystanie stałości połączenia przez WebSocket do identyfikacji dostępności użytkowników w serwisie. Jak wspomniano wcześniej baza danych zawiera tabelę Session przechowującą dane sesji poszczególnych użytkowników (dla jednego użytkownika będzie przeznaczony tylko jeden rekord): data zalogowania się, data przerwania sesji oraz identyfikator użytkownika i połączenia z hub’em (każdy hub posiada własny kontekst poprzez który można taki identyfikator pozyskać – pozwala to w bardzo prosty sposób identyfikować użytkowników korzystających z hub’u).



Rysunek 43 Fragment kodu przedstawiający hub do oznaczania dostępności użytkowników

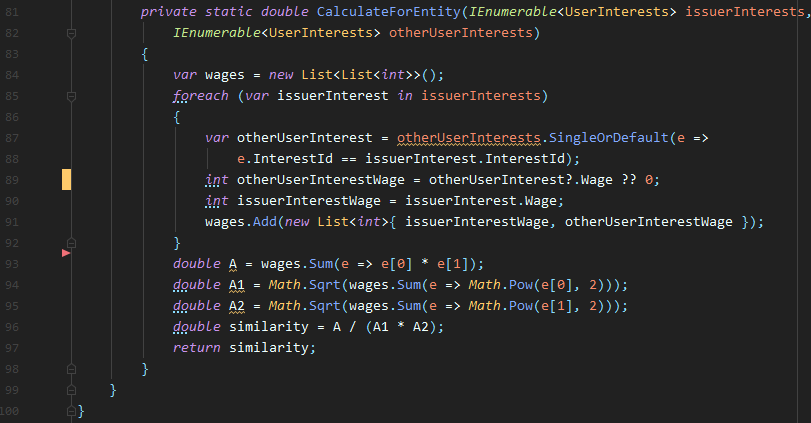
W tym wypadku klient będzie korzystał z metody ConnectUser przekazując identyfikator użytkownika, po czym w wierszu 35 odbywa się wyszukiwanie rekordu sesji użytkownika w oparciu o identyfikator. Jeżeli użytkownik logował się wcześniej taki rekord zostanie odnaleziony i zmieniony, dzięki czemu będzie możliwość stwierdzenia, że użytkownik jest aktywny lub w przypadku gdy taki użytkownik loguje się po raz pierwszy zostanie mu przydzielony nowy rekord. Do oznaczenia użytkownika zmiany daty przerwania sesji w tym wypadku można skorzystać z metody cyklu życia kontekstu OnDisconnectedAsync – wykonuje się ona tuż po rozłączeniu się z serwerem, czyli nie jest w żaden sposób zależna od kodu klienta. Tutaj implementacja wygląda podobnie za wyjątkiem tego, że wyciąganie sesji z bazy odbywa się w oparciu o identyfikator kontekstu, po czym wartość zmiennej określającej datę ukończenia sesji zostaje zmieniona na bieżącą datę i cały rekord jest aktualizowany w repozytorium.

# 3.9. Implementacja algorytmu dopasowania znajomych

Do implementacji dopasowania znajomych na podstawie zainteresowań zostanie użyta metoda *Cosine Similarity*, tradycyjnie często wykorzystywana podczas tworzenia systemów rekomendacji w oparciu o algorytm *User Based Collaboration Filtering* lub *Item Based Collaboration Filtering* korzystających z czynnika podobieństwa w celu dobrania dopasowania. Mając dwa wektory atrybutów x oraz y działanie tej metody można oprzeć o następujący wzór[[75]](#footnote-75)

Tutaj parametry x i y oznaczają wagi lub stopień zainteresowania tą samą aktywnością przez dwóch porównywanych użytkowników. Początkowo praca nad implementacją opiera się o sprowadzeniu istniejących danych do postaci macierzy wag ai,j, gdzie i = {1, 2, 3…} oznacza wiersz odpowiadający identyfikatorowi każdego użytkownika, a j = {1, 2, 3…} oznacza kolumnę odpowiadającą identyfikatorowi każdego zainteresowania. Liczenie zawsze odbywa się względem jednego użytkownika.

Poniższy rysunek przedstawia implementację powyższego algorytmu. Za obliczanie jest odpowiedzialna tylko jedna metoda przyjmująca jako parametry listę zainteresowań użytkownika dla którego są dobierani potencjalni znajomi oraz innego użytkownika. Warto zaznaczyć, że działanie algorytmu będzie opierało się o pętlę, ponieważ przestawiana metoda będzie wywoływana dla każdego użytkownika w bazie danych. W wierszu 84 jest deklarowana lista składająca się z list liczb całkowitych – tutaj będą przechowywane kombinacje wag zainteresowań, co ułatwi dalsze obliczenia. Wiersze 85 i 92 przedstawiają pętlę zadaniem której jest iteracja po każdym zainteresowaniu użytkownika wywołującego metodę i możliwe odnajdywanie odpowiadającego zainteresowania na liście zainteresowań innego użytkownika, po czym następuje dodanie wag do powyższej listy wag. Wiersze 93 i 97 przedstawiają bezpośrednią implementację rozpatrywanego algorytmu, w tym wypadku zmienna A przedstawia licznik, a zmienne A1 i A2 wartości pod mianownikiem.



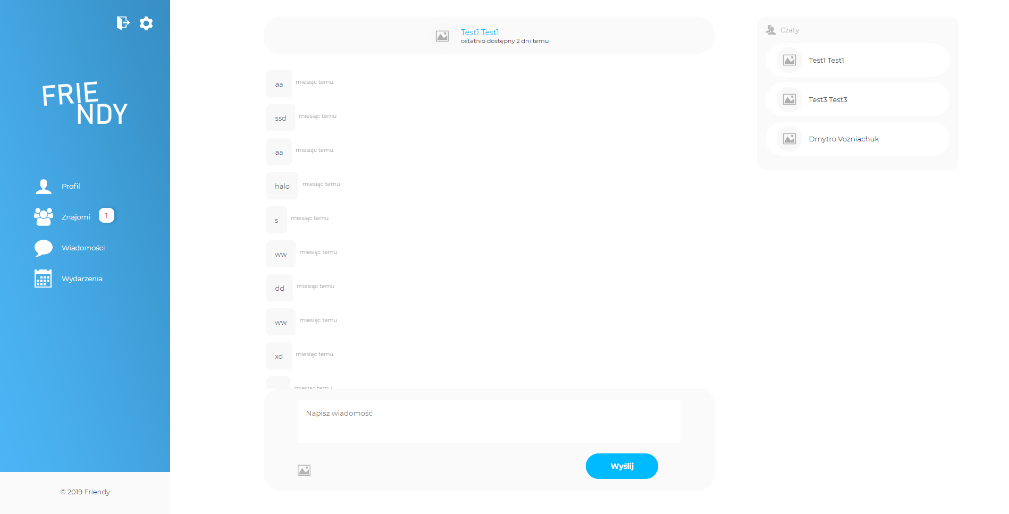
Rysunek 44 Fragment kodu przedstawiający implementację algorytmu CosSim

Po zakończeniu obliczeń użytkownik otrzymuje listę użytkowników najbardziej do siebie podobnych.

Jak widać nawet w tak prosty sposób można zaimplementować dosyć skuteczny system rekomendacji. Aby polepszyć rekomendacje w tworzonym projekcie dodatkowo można utworzyć system uwzględniający, na przykład, czynnik wzajemnych odwiedzin profili przez użytkowników, ilość wydarzeń w których użytkownicy brali lub biorą udział, lokalizację.

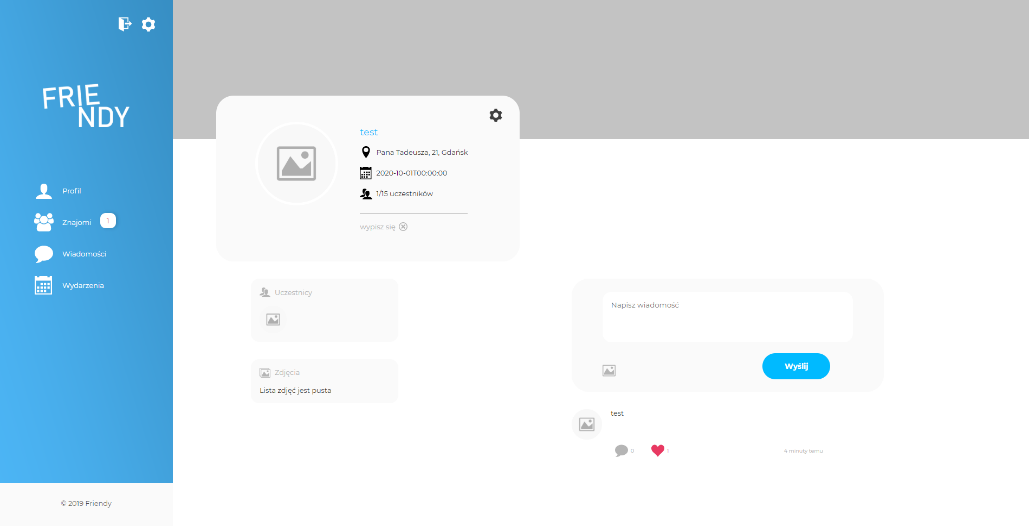
# 3.10. Przedstawienie podstron

Jest to ostatni podrozdział związany z implementacją. Tutaj zostaną przestawione wersje końcowe większości podstron w aplikacji.



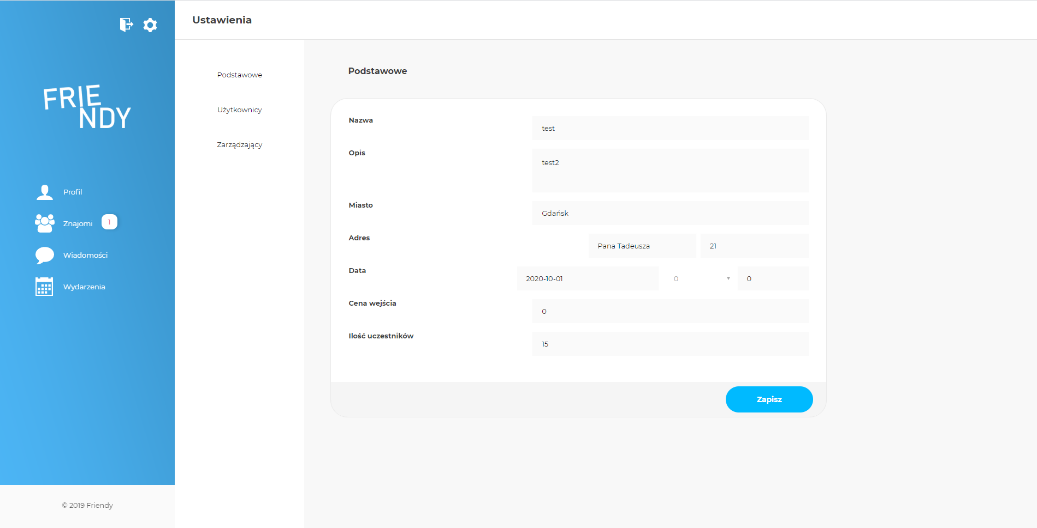
Rysunek 45 Podstrona wymiany wiadomości

Powyższy rysunek przedstawia podstronę na której odbywa się wymiana wiadomości pomiędzy użytkownikami. To właśnie tutaj została zastosowana biblioteka SignalR. Ponadto użytkownicy mają możliwość poruszania się w prosty sposób po innych czatach używając panelu „Czaty”.



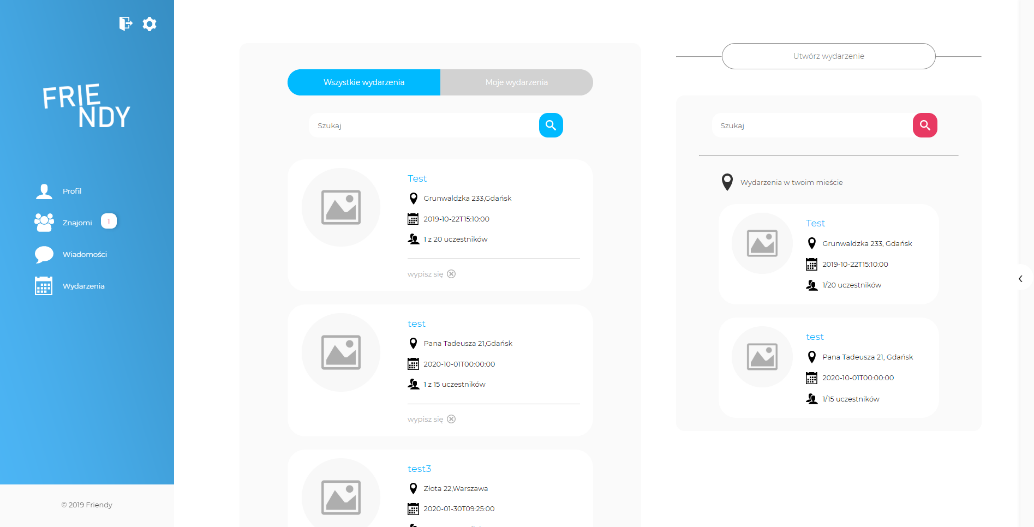
Rysunek 46 Podstrona wydarzenia

Powyższy rysunek przedstawia podstronę wydarzenia na której znalazły się podstawowe informacje o wydarzeniu, przykładowa lista uczestników i zdjęć. Oprócz tego, ponieważ zrzut ekranu został wykonany z perspektywy administratora jest możliwość dodawania nowych wpisów.



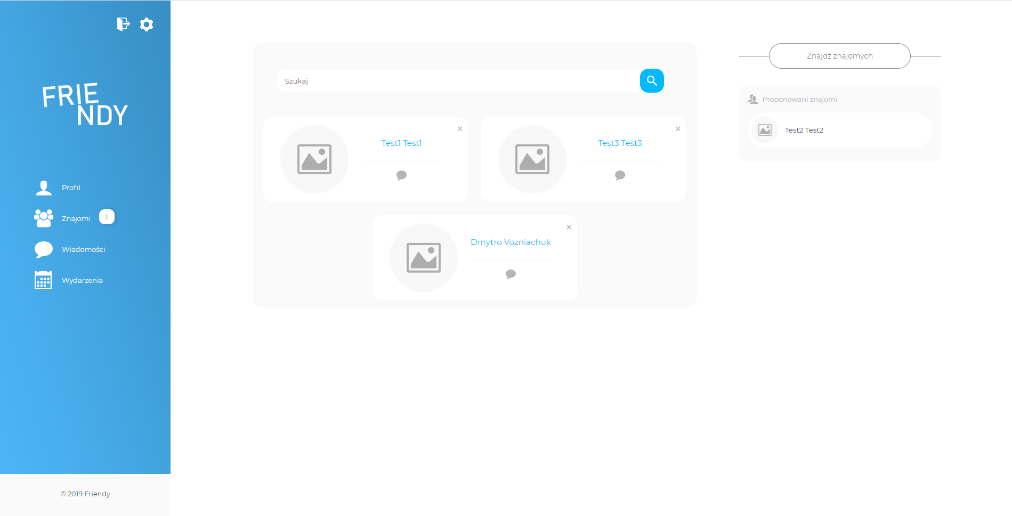
Rysunek 47 Podstrona ustawień wydarzenia

Powyższy rysunek przedstawia podstronę ustawień wydarzenia na której administrator może zmienić podstawowe dane wydarzenia.



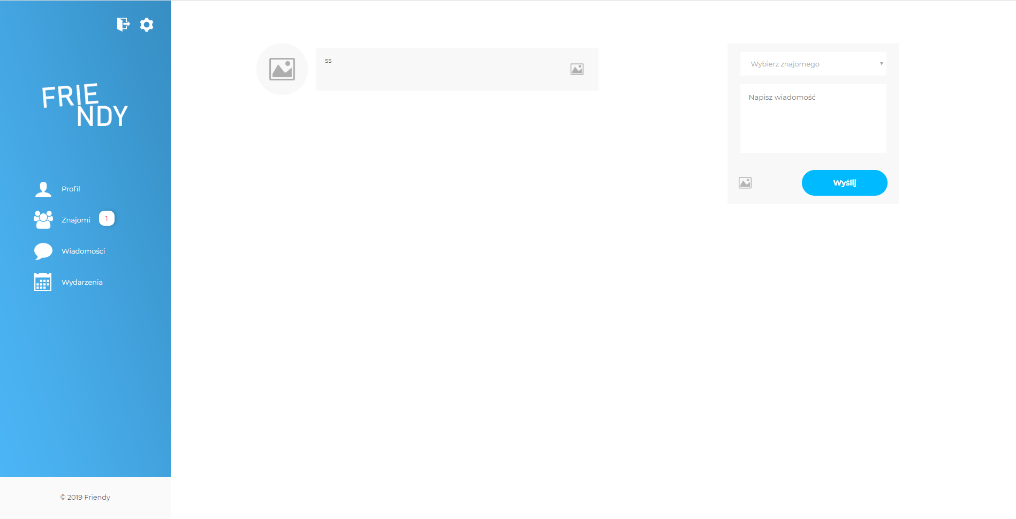
Rysunek 48 Podstrona wydarzeń

Powyższy rysunek przedstawia podstronę na której znalazły się lista wydarzeń w których uczestniczy użytkownik oraz tych zarządzanych przez użytkownika. Poza tym w prawym panelu jest lista wydarzeń które mają się odbyć w mieście zamieszkanym przez użytkownika. Poprzez kliknięcie w przycisk „Utwórz wydarzenie” otworzy się okno w którym użytkownik może utworzyć nowe wydarzenie. Po prawej stronie poprzez wciśnięcie w przycisk oznaczony strzałką wysuwa się okno zaawansowanego wyszukiwania wydarzeń według różnych kryteriów, ponieważ pola tekstowe przedstawione na zrzucie przeszukują listę wydarzeń wyłącznie w oparciu o nazwę.



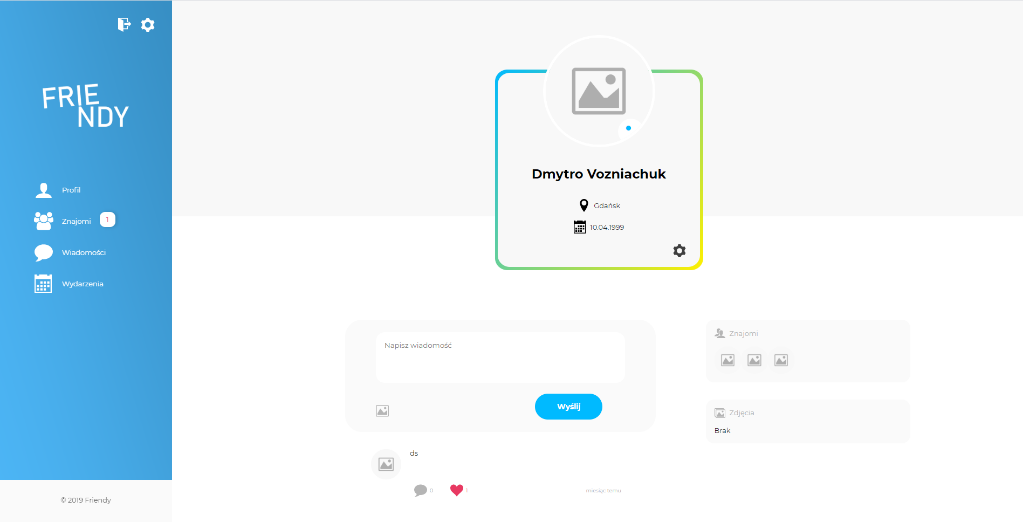
Rysunek 49 Podstrona znajomych

Powyższy rysunek przedstawia podstronę na której znalazła się lista znajomych użytkownika. Ponadto w lewym panelu została umieszczona lista proponowanych znajomych oraz przycisk przekierowujący do podstrony wyszukiwania użytkowników w oparciu o Elasticsearch.



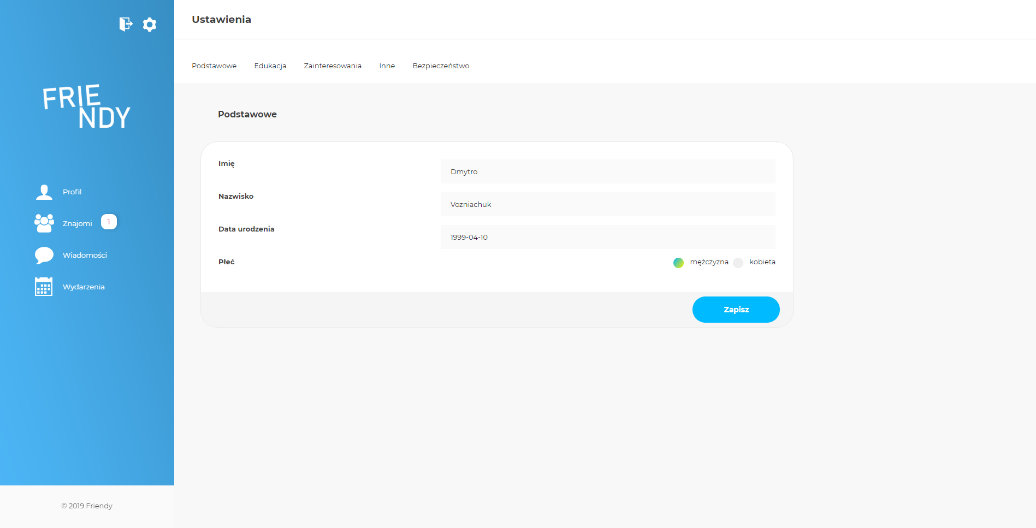
Rysunek 50 Podstrona wiadomości

Powyższy rysunek przedstawia podstronę na której znalazła się lista wszystkich wiadomości użytkownika. Oprócz tego poprzez dodatkowy panel jest możliwość szybkiego wysłania wiadomości do wybranego z listy znajomego.



Rysunek 51 Podstrona profilu użytkownika

Powyższy rysunek przedstawia wcześniej wspomnianą podstronę profilu użytkownika na której znalazły się podstawowe informacje o użytkowniku (imię, nazwisko, miasto zamieszkania, data urodzenia), przykładowa lista zdjęć i znajomych użytkownika. Niebieskie kółko położone nad zdjęciem profilowym użytkownika oznacza, że jest on aktualnie dostępny, czyli ma nawiązane połączenie z serwerem poprzez WebSocket. Ponadto, w związku z tym, że zrzut został wykonany bezpośrednio na profilu zalogowanego użytkownika jest widoczny formularz do utworzenia wpisu.



Rysunek 52 Podstrona ustawień użytkownika

Powyższy rysunek przedstawia podstronę na której zalogowany użytkownik może zmienić podstawowe informacje o sobie.

# Zakończenie

Obecnie programista ma szerokie możliwości w kontekście konstruowania wachlarzu technologii z użyciem których ma powstać system informatyczny. Do implementacji różnych funkcjonalności utworzonej aplikacji zostały wykorzystane przede wszystkim rozwiązania mające renomę i bardzo często wykorzystywane w innych projektach.

Celem tej pracy była implementacja lokalnej sieci społecznościowej umożliwiającej poznawanie nowych osób. W trakcie pracy oprócz etapów implementacji aplikacji zostały omówione zagadnienia związane z inżynierią oprogramowania wykorzystane bezpośrednio w kontekście implementacji aplikacji: zostały określone wymagania, sporządzone diagramy UML, zaimplementowane wzorce repozytorium i CQRS.

Przyszłość utworzonego rozwiązania nie jest jednoznaczna ponieważ rozwijać je można w kontekście przeróżnych aspektów. Jednym z nich jest wykorzystanie sztucznej inteligencji do implementacji skuteczniejszego systemu rekomendacji znajomych, na przykład poprzez użycie sieci neuronowych. Ponadto przyszłość platformy .NET jest bardzo świetlana ponieważ już w roku 2020 zostanie wydane .NET 5[[76]](#footnote-76) – następca .NET Core i .NET Framework. Może to skutkować próbą migracji projektu do nowszego rozwiązania. Także powoli na rynek wchodzą rozwiązania oparte o WebAssembly[[77]](#footnote-77), które jest standardem zapisu instrukcji binarnych dla maszyn wirtualnych opartych na stosie, głównym zadaniem którego jest umożliwienie uruchamiania aplikacji internetowych utworzonych w wysokopoziomowych językach programowania. Wychodząc z kontekstu .NET uruchomienie aplikacji internetowej utworzonej w języku C# umożliwia narzędzie o nazwie Blazor[[78]](#footnote-78). Choć jest to narzędzie ciągle rozwijane w przyszłości może pojawić się możliwość przeniesienia i rozwijania utworzonego rozwiązania w oparciu o tę platformę.

# Bibliografia

Angular <https://pl.wikipedia.org/wiki/Angular_(framework)>

Adam Freeman, *Pro ASP.NET Core MVC 2*, 7 edycja, APRESS, s.6

SQL Server <https://en.wikipedia.org/wiki/Microsoft_SQL_Server>

JavaScript Object Notation <https://pl.wikipedia.org/wiki/JSON>

Apache Lucene <https://lucene.apache.org/>

Radu Gheorghe, Matthew Lee Hinman, Roy Russo, *Elasticsearch in Action*, Manning Publications, Nowy Jork 2016, s. 5

Flux <https://facebook.github.io/flux/docs/in-depth-overview>

NgRx <https://ngrx.io/docs>

José M. Aguilar, *SignalR Programming In Microsoft ASP.NET*, Microsoft Press, Redmond, Washington 98052-6399 2014, s. 18

Fenie Reynders, przedmowa Scott Hanselman, *Modern API Design with ASP.NET Core 2: Building Cross-Platform Back-End Systems*, APRESS, Odijk, Holandia 2018, s. 6

REST Uniform Interface [https://en.wikipedia.org/wiki/Representational\_state\_transfer#Uniform\_interface](https://en.wikipedia.org/wiki/Representational_state_transfer%23Uniform_interface)

10 najważniejszych ataków lub usterek w aplikacjach internetowych <https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf>

Edward Pollack, *Dynamic SQL: Applications, Performance, and Security in Microsoft SQL Server*, APRESS, Albany, NY, USA 2019, s. 31

RFC7519, JSON Web Token (JWT) <https://tools.ietf.org/html/rfc7519>

Plain Old CLR Object <https://pl.wikipedia.org/wiki/Plain_Old_CLR_Object>

Kontroler w ASP.NET [https://docs.microsoft.com/pl-pl/aspnet/web-api/overview/getting-started-with-aspnet-web-api/tutorial-your-first-web-api#adding-a-controller](https://docs.microsoft.com/pl-pl/aspnet/web-api/overview/getting-started-with-aspnet-web-api/tutorial-your-first-web-api%23adding-a-controller)

Martin Fowler, *Patterns of Enterprise Application Architecture*, Pearson Education, 2003, s. 322

Is the repository pattern useful with Entity Framework Core? <http://www.nichesoftware.co.nz/2011/05/09/reconsidering-the-repository-pattern.html>

Reconsidering the Repository Pattern <http://www.nichesoftware.co.nz/2011/05/09/reconsidering-the-repository-pattern.html>

Implementing Asynchronous Generic Repository in ASP.NET Core <https://code-maze.com/async-generic-repository-pattern>

Entity Framework <https://en.wikipedia.org/wiki/Entity_Framework>

NHibernate <https://en.wikipedia.org/wiki/NHibernate>

Automapper <http://docs.automapper.org/en/stable/Getting-started.html>

Greg Young, *CQRS Documents*,s.17

1. MediatR <https://github.com/jbogard/MediatR>

John Vlissides, Ralph Johnson, Richard Helm, Erich Gamma, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, USA 1994, s.305

1. Bcrypt.Net <https://github.com/BcryptNet/bcrypt.net>
2. Wojciech Z. Chmielowski, Dorota Wilk-Kołodziejczyk, *Metody analizy i oceny bezpieczeństwa oraz jakości informacji*, Kraków 2012, s.16
3. History API [https://developer.mozilla.org/en-US/docs/Web/API/History\_API](%20https:/developer.mozilla.org/en-US/docs/Web/API/History_API)
4. Angular hooks [https://angular.io/guide/lifecycle-hooks#lifecycle-sequence](https://angular.io/guide/lifecycle-hooks%23lifecycle-sequence)
5. Clinton Gormley, Zachary Tong, *Elasticsearch: The Definitive Guide*, O’Reilly Media, styczeń 2015, s. 103
6. NgRx Reducers <https://ngrx.io/guide/store/reducers>
7. WebSocket <https://en.wikipedia.org/wiki/WebSocket>
8. Cosine Similarity <https://en.wikipedia.org/wiki/Cosine_similarity>
9. TCP <https://pl.wikipedia.org/wiki/Protok%C3%B3%C5%82_sterowania_transmisj%C4%85>
10. Introducing .NET 5 <https://devblogs.microsoft.com/dotnet/introducing-net-5/>
11. WebAssembly <https://webassembly.org/>
12. Blazor <https://docs.microsoft.com/pl-pl/aspnet/core/blazor/?view=aspnetcore-3.1>

# Spis tabel

[Tabela 1 Pryncypia którymi musi cechować się każdy system REST 13](#_Toc29377009)

[Tabela 2 10 najważniejszych ataków lub usterek w aplikacjach internetowych 15](#_Toc29377010)

[Tabela 3 Lista tabel projektowej bazy danych 23](#_Toc29377011)

[Tabela 4 Lista kontrolerów wykorzystanych w projekcie 27](#_Toc29377012)

[Tabela 5 Opis parametrów akceptowanych przez filtr boolowski 39](#_Toc29377013)

# Spis ilustracji

[Rysunek 1 Flux - diagram przedstawiający podstawowy schemat przepływu danych. Źródło: opracowanie własne 6](#_Toc29377054)

[Rysunek 2 Diagram przedstawiający ogólny schemat przepływu danych przy zarządzaniu stanem. Źródło: https://ngrx.io/guide/store#diagram 7](#_Toc29377055)

[Rysunek 3 Diagram przypadków użycia przedstawiający uprawnienia użytkownika. Źródło: opracowanie własne 10](#_Toc29377056)

[Rysunek 4 Diagram przypadków użycia przedstawiający dodatkowe uprawnienia założyciela wydarzenia (oprócz podstawowych uprawnień administratora) oraz uczestnika wydarzenia. Źródło: opracowanie własne 10](#_Toc29377057)

[Rysunek 5 Diagram przypadków użycia przedstawiający uprawnienia administratora wydarzenia. Źródło: opracowanie własne 11](#_Toc29377058)

[Rysunek 6 Architektura tworzonego systemu. Źródło: opracowanie własne 11](#_Toc29377059)

[Rysunek 7 Przykładowy zaszyfrowany JWT. Źródło: opracowanie własne 15](#_Toc29377060)

[Rysunek 8 Odkodowany JWT. Źródło: opracowanie własne 16](#_Toc29377061)

[Rysunek 9 Przykład ciasteczka HTTPOnly o nazwie SESSION\_TOKEN zawierającego JWT. Źródło: opracowanie własne 16](#_Toc29377062)

[Rysunek 10 Diagram sekwencji opisujący proces wysyłania zapytań GET do API z poziomu przeglądarki z użyciem Angular HTTP Client. Źródło: opracowanie własne 17](#_Toc29377063)

[Rysunek 11 Diagram ER zaprojektowanej bazy danych. Źródło: opracowanie własne 24](#_Toc29377064)

[Rysunek 12 Komenda umożliwiająca wygenerowanie modeli domenowych. Źródło: opracowanie własne 24](#_Toc29377065)

[Rysunek 13 Ilustracja zawartości folderu Models, który mieści wygenerowane modele przez Entity Framework Core. Źródło: opracowanie własne 25](#_Toc29377066)

[Rysunek 14 Model wpisu jako przykładowy model wygenerowany przez EF Core. Źródło: opracowanie własne 25](#_Toc29377067)

[Rysunek 15 Diagram klas przedstawiający implementację wzorca repozytorium w tworzonej aplikacji. Źródło: opracowanie własne 28](#_Toc29377068)

[Rysunek 16 Komunikacja pomiędzy komponentami przy zastosowaniu wzorca repozytoriów i serwisów. Źródło: opracowanie własne 29](#_Toc29377069)

[Rysunek 17 Metoda w kontrolerze UserPostController obsługująca zapytanie o konkretną liczbę wpisów użytkownika. Źródło: opracowanie własne 29](#_Toc29377070)

[Rysunek 18 Wycinek serwisu UserPostService obsługujący logikę biznesową związaną z pobieraniem danych z repozytorium oraz przekształcaniem wyniku do postaci klasy UserPostDto. Źródło: opracowanie własne 30](#_Toc29377071)

[Rysunek 19 Profil mapowania obiektu klasy UserPost do UserPostDto. Źródło: opracowanie własne 30](#_Toc29377072)

[Rysunek 20 Fragment repozytorium UserPostRepository wraz z metodą pobierającą niezbędne dane o wpisie z bazy danych. Źródło: opracowanie własne 31](#_Toc29377073)

[Rysunek 21 Wycinek kodu przedstawiający metodę odpowiadającą za przetwarzanie zapytania o wyznaczoną listę wpisów na stronie wydarzenia. Źródło: opracowanie własne 32](#_Toc29377074)

[Rysunek 22 Model GetEventPostRangeById. Źródło: opracowanie własne 32](#_Toc29377075)

[Rysunek 23 Handler obsługujący odczytywanie wpisów na stronie wydarzenia w oparciu o podany model do odczytu. Źródło: opracowanie własne 32](#_Toc29377076)

[Rysunek 24 Obiekt JSON zawierający podstawowe dane nowego użytkownika. Źródło: opracowanie własne 33](#_Toc29377077)

[Rysunek 25 Metoda w kontrolerze AuthController przetwarzająca rejestrację nowego użytkownika. Źródło: opracowanie własne 33](#_Toc29377078)

[Rysunek 26 Metoda w serwisie AuthenticationService odpowiadająca za rejestrację nowego użytkownika. Źródło: opracowanie własne 34](#_Toc29377079)

[Rysunek 27 Strona rejestracji nowego użytkownika. Źródło: opracowanie własne 34](#_Toc29377080)

[Rysunek 28 Implementacja zapytania POST do REST API o poprawność wprowadzonych danych logowania. Źródło: opracowanie własne 35](#_Toc29377081)

[Rysunek 29 Ilustracja fragmentu kodu odpowiadającego za rejestrację nowego użytkownika. Źródło: opracowanie własne 36](#_Toc29377082)

[Rysunek 30 Moduł konfiguracji routingu. Źródło: opracowanie własne 37](#_Toc29377083)

[Rysunek 31 Skrypty i kod HTML komponentu ProfilePageComponent. Źródło: opracowanie własne 37](#_Toc29377084)

[Rysunek 32 Podstrona profilu użytkownika podzielona na komponenty. Źródło: opracowanie własne 38](#_Toc29377085)

[Rysunek 33 Przykładowy dokument zawierający dane przeznaczone do wyszukiwania użytkowników. Źródło: opracowanie własne 39](#_Toc29377086)

[Rysunek 34 Odpowiedź serwera ES w programie Postman. Źródło: opracowanie własne 40](#_Toc29377087)

[Rysunek 35 Fragment kodu przedstawiający aktualizację danych w klastrze Elasticsearch z użyciem klienta NEST. Źródło: opracowanie własne 40](#_Toc29377088)

[Rysunek 36 Typy akcji dla reduktora danych użytkownika. Źródło: opracowanie własne 41](#_Toc29377089)

[Rysunek 37 Efekty dla stanu danych użytkownika. Źródło: opracowanie własne 42](#_Toc29377090)

[Rysunek 38 Stan i reduktor danych użytkownika. Źródło: opracowanie własne 43](#_Toc29377091)

[Rysunek 39 Hub obsługujący pracę z wiadomościami. Źródło: opracowanie własne 44](#_Toc29377092)

[Rysunek 40 Fragment kodu przedstawiający implementację nawiązania połączenia z serwerem. Źródło: opracowanie własne 44](#_Toc29377093)

[Rysunek 41 Fragment kodu przedstawiający metody nasłuchiwania i wywołania SignalR w aplikacji. Źródło: opracowanie własne 46](#_Toc29377094)

[Rysunek 42 Implementacja serwisu wysyłającego wiadomości do klienta. Źródło: opracowanie własne 46](#_Toc29377095)

[Rysunek 43 Fragment kodu przedstawiający hub do oznaczania dostępności użytkowników. Źródło: opracowanie własne 47](#_Toc29377096)

[Rysunek 44 Fragment kodu przedstawiający implementację algorytmu CosSim. Źródło: opracowanie własne 49](#_Toc29377097)

[Rysunek 45 Podstrona wymiany wiadomości. Źródło: opracowanie własne 49](#_Toc29377098)

[Rysunek 46 Podstrona wydarzenia. Źródło: opracowanie własne 50](#_Toc29377099)

[Rysunek 47 Podstrona ustawień wydarzenia. Źródło: opracowanie własne 50](#_Toc29377100)

[Rysunek 48 Podstrona wydarzeń. Źródło: opracowanie własne 51](#_Toc29377101)

[Rysunek 49 Podstrona znajomych. Źródło: opracowanie własne 51](#_Toc29377102)

[Rysunek 50 Podstrona wiadomości. Źródło: opracowanie własne 52](#_Toc29377103)

[Rysunek 51 Podstrona profilu użytkownika. Źródło: opracowanie własne 52](#_Toc29377104)

[Rysunek 52 Podstrona ustawień użytkownika. Źródło: opracowanie własne 53](#_Toc29377105)

1. Web application <https://en.wikipedia.org/wiki/Web_application> [↑](#footnote-ref-1)
2. From History of Web Application Development <https://www.devsaran.com/blog/history-web-application-development> [↑](#footnote-ref-2)
3. Macromedia <https://pl.wikipedia.org/wiki/Macromedia> [↑](#footnote-ref-3)
4. Adobe Flash <https://en.wikipedia.org/wiki/Adobe_Flash> [↑](#footnote-ref-4)
5. Ajax <https://pl.wikipedia.org/wiki/AJAX> [↑](#footnote-ref-5)
6. Fundacja Mozilla <https://pl.wikipedia.org/wiki/Fundacja_Mozilla> [↑](#footnote-ref-6)
7. Opera Software <https://pl.wikipedia.org/wiki/Opera_Software> [↑](#footnote-ref-7)
8. World Wide Web Consortium <https://pl.wikipedia.org/wiki/World_Wide_Web_Consortium> [↑](#footnote-ref-8)
9. WHATWG <https://pl.wikipedia.org/wiki/Web_Hypertext_Application_Technology_Working_Group> [↑](#footnote-ref-9)
10. Jonathan Reid, *HTML5 Programmer's Reference*, APRESS, S.11 [↑](#footnote-ref-10)
11. HTML5 <https://pl.wikipedia.org/wiki/HTML5> [↑](#footnote-ref-11)
12. PWA <https://en.wikipedia.org/wiki/Progressive_web_application> [↑](#footnote-ref-12)
13. Twitter Mobile <https://en.wikipedia.org/wiki/Twitter#Mobile> [↑](#footnote-ref-13)
14. jQuery <https://jquery.com/> [↑](#footnote-ref-14)
15. Gil Fink, Ido Flatow, *Pro Single Page Application Development Using Backbone.js and ASP.NET*, APRESS, s. 12 [↑](#footnote-ref-15)
16. MVP <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter> [↑](#footnote-ref-16)
17. Smalltalk <https://en.wikipedia.org/wiki/Smalltalk> [↑](#footnote-ref-17)
18. Xerox PARC <https://pl.wikipedia.org/wiki/Xerox_PARC> [↑](#footnote-ref-18)
19. Journal of Object Technology <https://pl.wikipedia.org/wiki/Journal_of_Object_Technology> [↑](#footnote-ref-19)
20. History of MVC <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller#History> [↑](#footnote-ref-20)
21. ASP.NET MVC <https://pl.wikipedia.org/wiki/ASP.NET_MVC> [↑](#footnote-ref-21)
22. Spring Web MVC <https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc> [↑](#footnote-ref-22)
23. Andrea Chiarelli, *Mastering JavaScript Object-Oriented Programming*, Packt Publishing, Wielka Brytania 2016,s. 140 [↑](#footnote-ref-23)
24. Retirement note for Model View Presenter Pattern <https://martinfowler.com/eaaDev/ModelViewPresenter.html> [↑](#footnote-ref-24)
25. WPF <https://docs.microsoft.com/pl-pl/visualstudio/designers/getting-started-with-wpf?view=vs-2019> [↑](#footnote-ref-25)
26. Silverlight <https://pl.wikipedia.org/wiki/Microsoft_Silverlight> [↑](#footnote-ref-26)
27. Introduction to Model/View/ViewModel pattern for building WPF apps <https://docs.microsoft.com/pl-pl/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps> [↑](#footnote-ref-27)
28. Components of MVVM pattern <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel#Components_of_MVVM_pattern> [↑](#footnote-ref-28)
29. Angular <https://pl.wikipedia.org/wiki/Angular_(framework)> [↑](#footnote-ref-29)
30. TypeScript <https://en.wikipedia.org/wiki/TypeScript> [↑](#footnote-ref-30)
31. Intoduction to components <https://angular.io/guide/architecture-components> [↑](#footnote-ref-31)
32. Angular - architecture overview <https://angular.io/guide/architecture> [↑](#footnote-ref-32)
33. Flux <https://facebook.github.io/flux/docs/in-depth-overview> [↑](#footnote-ref-33)
34. NgRx <https://ngrx.io/docs> [↑](#footnote-ref-34)
35. Spring <https://spring.io/> [↑](#footnote-ref-35)
36. Node.js <https://pl.wikipedia.org/wiki/Node.js> [↑](#footnote-ref-36)
37. Adam Freeman, *Pro ASP.NET Core MVC 2*, 7 edycja, APRESS, s. 6 [↑](#footnote-ref-37)
38. José M. Aguilar, *SignalR Programming In Microsoft ASP.NET*, Microsoft Press, Redmond, Washington 98052-6399 2014, s. 18 [↑](#footnote-ref-38)
39. Baza danych Oracle <https://pl.wikipedia.org/wiki/Oracle_Database> [↑](#footnote-ref-39)
40. MySQL <https://pl.wikipedia.org/wiki/MySQL> [↑](#footnote-ref-40)
41. PostgreSQL <https://pl.wikipedia.org/wiki/PostgreSQL> [↑](#footnote-ref-41)
42. SQL Server <https://en.wikipedia.org/wiki/Microsoft_SQL_Server#firstHeading> [↑](#footnote-ref-42)
43. SQL Server. Database Files and Filegroups <https://docs.microsoft.com/en-us/sql/relational-databases/databases/database-files-and-filegroups?view=sql-server-ver15> [↑](#footnote-ref-43)
44. SQL Server. Pages and Extents Architecture Guide <https://docs.microsoft.com/en-us/sql/relational-databases/pages-and-extents-architecture-guide?view=sql-server-ver15> [↑](#footnote-ref-44)
45. SQL Server. Buffer Management <https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/aa337525(v=sql.105)> [↑](#footnote-ref-45)
46. SQL Server. SQL Statement Processing <https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms190623(v=sql.105)> [↑](#footnote-ref-46)
47. JavaScript Object Notation <https://pl.wikipedia.org/wiki/JSON> [↑](#footnote-ref-47)
48. Apache Lucene <https://lucene.apache.org/> [↑](#footnote-ref-48)
49. Radu Gheorghe, Matthew Lee Hinman, Roy Russo, *Elasticsearch in Action*, Manning Publications, Nowy Jork 2016, s. 5 [↑](#footnote-ref-49)
50. <https://www.elastic.co/guide/en/elasticsearch/reference/current/documents-indices.html> [↑](#footnote-ref-50)
51. Elasticsearch. Query DSL <https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html> [↑](#footnote-ref-51)
52. Fenie Reynders, przedmowa Scott Hanselman, *Modern API Design with ASP.NET Core 2: Building Cross-Platform Back-End Systems*, APRESS, Odijk, Holandia 2018, s. 6 [↑](#footnote-ref-52)
53. REST Uniform Interface [https://en.wikipedia.org/wiki/Representational\_state\_transfer#Uniform\_interface](https://en.wikipedia.org/wiki/Representational_state_transfer%23Uniform_interface) [↑](#footnote-ref-53)
54. 10 najważniejszych ataków lub usterek w aplikacjach internetowych <https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf> [↑](#footnote-ref-54)
55. Edward Pollack, *Dynamic SQL: Applications, Performance, and Security in Microsoft SQL Server*, APRESS, Albany, NY, USA 2019, s. 31 [↑](#footnote-ref-55)
56. RFC7519, JSON Web Token (JWT) <https://tools.ietf.org/html/rfc7519> [↑](#footnote-ref-56)
57. Plain Old CLR Object <https://pl.wikipedia.org/wiki/Plain_Old_CLR_Object> [↑](#footnote-ref-57)
58. Kontroler w ASP.NET [https://docs.microsoft.com/pl-pl/aspnet/web-api/overview/getting-started-with-aspnet-web-api/tutorial-your-first-web-api#adding-a-controller](https://docs.microsoft.com/pl-pl/aspnet/web-api/overview/getting-started-with-aspnet-web-api/tutorial-your-first-web-api%23adding-a-controller) [↑](#footnote-ref-58)
59. Martin Fowler, *Patterns of Enterprise Application Architecture*, Pearson Education, 2003, s. 322 [↑](#footnote-ref-59)
60. *Is the repository pattern useful with Entity Framework Core?* <http://www.nichesoftware.co.nz/2011/05/09/reconsidering-the-repository-pattern.html> [↑](#footnote-ref-60)
61. NHibernate <https://en.wikipedia.org/wiki/NHibernate> [↑](#footnote-ref-61)
62. Implementing Asynchronous Generic Repository in ASP.NET Core <https://code-maze.com/async-generic-repository-pattern> [↑](#footnote-ref-62)
63. Automapper <http://docs.automapper.org/en/stable/Getting-started.html> [↑](#footnote-ref-63)
64. Greg Young, *CQRS Documents*,s. 17 [↑](#footnote-ref-64)
65. MediatR <https://github.com/jbogard/MediatR> [↑](#footnote-ref-65)
66. John Vlissides, Ralph Johnson, Richard Helm, Erich Gamma, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, USA 1994, s. 305 [↑](#footnote-ref-66)
67. Bcrypt.Net <https://github.com/BcryptNet/bcrypt.net> [↑](#footnote-ref-67)
68. Wojciech Z. Chmielowski, Dorota Wilk-Kołodziejczyk, *Metody analizy i oceny bezpieczeństwa oraz jakości informacji*, Kraków 2012, s. 16 [↑](#footnote-ref-68)
69. History API <https://developer.mozilla.org/en-US/docs/Web/API/History_API> [↑](#footnote-ref-69)
70. Angular hooks [https://angular.io/guide/lifecycle-hooks#lifecycle-sequence](https://angular.io/guide/lifecycle-hooks%23lifecycle-sequence) [↑](#footnote-ref-70)
71. Clinton Gormley, Zachary Tong, *Elasticsearch: The Definitive Guide*, O’Reilly Media, styczeń 2015, s. 103 [↑](#footnote-ref-71)
72. NgRx Reducers <https://ngrx.io/guide/store/reducers> [↑](#footnote-ref-72)
73. WebSocket <https://en.wikipedia.org/wiki/WebSocket> [↑](#footnote-ref-73)
74. TCP <https://pl.wikipedia.org/wiki/Protok%C3%B3%C5%82_sterowania_transmisj%C4%85> [↑](#footnote-ref-74)
75. Cosine Similarity <https://en.wikipedia.org/wiki/Cosine_similarity> [↑](#footnote-ref-75)
76. *Introducing .NET 5* <https://devblogs.microsoft.com/dotnet/introducing-net-5/> [↑](#footnote-ref-76)
77. WebAssembly <https://webassembly.org/> [↑](#footnote-ref-77)
78. Blazor <https://docs.microsoft.com/pl-pl/aspnet/core/blazor/?view=aspnetcore-3.1> [↑](#footnote-ref-78)