Uniwersytet Zielonogórski
Wydział Informatyki, Elektrotechniki i Automatyki

Platforma .NET – Projekt
Prowadzący: dr inż. Marek Sawerwain

Tytuł raportu/sprawozdania

Wykonał: Damian Radecki, Grupa dziekańska: 33-INF-SSI-SP
Projekt realizowano razem z:
Damian Kurkiewicz

Data oddanie projektu: DD mmmm YYYY

Ocena: .....

Spis treści

Spis listingów

Motto:
Pisanie raportu przywilejem każdego studenta.

1 Wprowadzenie

1.1 Aplikacje czasu rzeczywistego

Aplikacje działające na żywo oferują wiele korzyści, które są sporym ułatwieniem dla użytkowników podczas używania takiej aplikacji. Czynności wykonywane bez odświeżania strony nie tylko skracają czas wykonywania czynności czy obsługiwania samej witryny to jeszcze znacznie ułatwiają komunikację, unikają blokowania strony i tworzą bardziej intuicyjny interface. Takie aplikacje internetowe staja się normą w dzisiejszych czasach. Każde przeładowanie strony jest nie komfortowe i stwarza pewnego rodzaju niebezpieczeństwo wykradnięcia danych. Serwisy internetowe obsługujące komunikację real-time z klientem są lepiej zabezpieczone i działają wydajnościowo lepiej. Powstawało wiele technologii do wsparcia komunikacji na żywo, które działają zarówno po stronie witryny i serwera. Są to między innymi *WebSocket, SignalR, RabbitMQ czy Apache Kafka*. Wszystkie z nich są dziś globalnie używane do wsparcia przekazu informacji.

1.2 Opis działania

Projekt chatu na żywo jest aplikacją, która wspiera komunikację między użytkownikami, aby ich konwersacje nie działały w stylu w jakim działa klasyczny serwer e-mail. Założenie projektu są takie, aby użytkownicy bez przeładowania strony mogli wymieniać miedzy sobą wiadomości. Dodatkowo wszelkie powiadomienia przychodzą również bez zbędnego odświeżania witryny. Sprawia to, że witryna jest bardziej intuicyjna, łatwiejsza i szybsza w obsłudze. Taka architektura aplikacji jest przyjazna użytkownikowi, od którego będzie wymagana minimalny wysiłek w trakcie używania strony. Celem takiej aplikacji jest też maksymalne bezpieczeństwo wspierane przez bearen token i autoryzację użytkowników z zachowaniem szyfrowania danych poufnych. Architektura zapewnia, że nie będziemy otrzymywać wiadomości od niezaakceptowanych użytkowników lecz daje możliwość uczestnictwa w czatach posiadających osoby nieznajome poprzez mechanizm grup. W grupach każdy użytkownik może zaprosić swoich znajomych co może spowodować komunikację między nieznajomymi w danym czacie.

1.3 Grupa docelowa

Grupą docelową są wszyscy użytkownicy którzy cenią sobie bezpieczeństwo i wygodę. Chcą szybko skomunikować się ze swoimi przyjaciółmi bądź grupą docelową bez żadnych opóźnień czy niepotrzebnych przeładować strony. Są pewni tego, że ich dane są przechowywane w bezpiecznym miejscu i nikt nie wkradnie się na ich konto. Mogą to być zarówno firmy, które chcą komunikować się między sobą i ewentualnie z klientami poprzez utworzenie chatu dla grupy użytkowników jak i dla szkół, uniwersytetów, grup pracowników, przyjaciół czy kolegów.

2 Użyte technologie

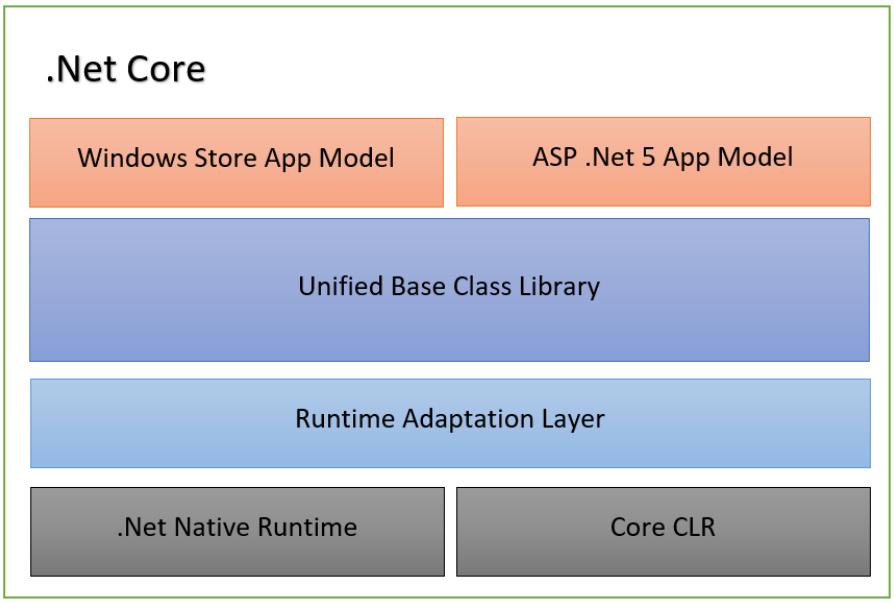
2.1 .Net Core

Popularny, nowoczesny i wydajny framework oparty o otwartoźródłowa implementację, który został wydany w 2016 roku do ogólnego przeznaczenia. Stanowi zestaw bibliotek pozwalający tworzyć wieloplatformowe aplikacje o wysokim stopniu bezpieczeństwa. Framework ten pozwala na pisanie aplikacji między innymi przeznaczonych do obliczeń chmurowych, IoT oraz jak w naszym przypadku do pisania web serwisu.

Rysunek 1: Logo .Net Core

Framework *.Net Core* został przez nas wybrany, ponieważ jest to nowa oraz dobrze prosperująca technologia wprowadzająca dużą dawkę świeżości podczas tworzenia nowego oprogramowania. Posiada wsparcie dla tworzenia web serwisów opartych o metodykę *REST*, poprzez

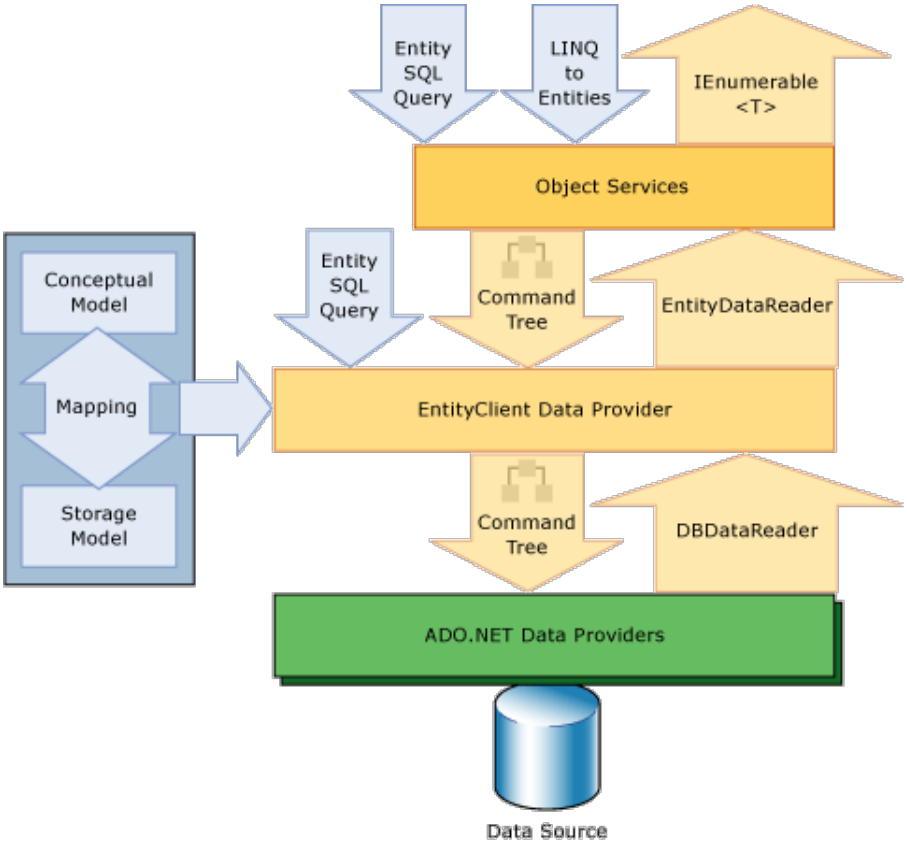
dodanie nowych i gotowych do działania bibliotek. Platforma *.Net Core* jest znacznie wydajniejsza od .Net Framework. Wprowadza znaczące usprawnienia przekładające się na szybkość działania pisanych programów. Architektura przedstawia się w sposób następujący.



Rysunek 2: Architektura .NET Core

2.2 Entity Framework

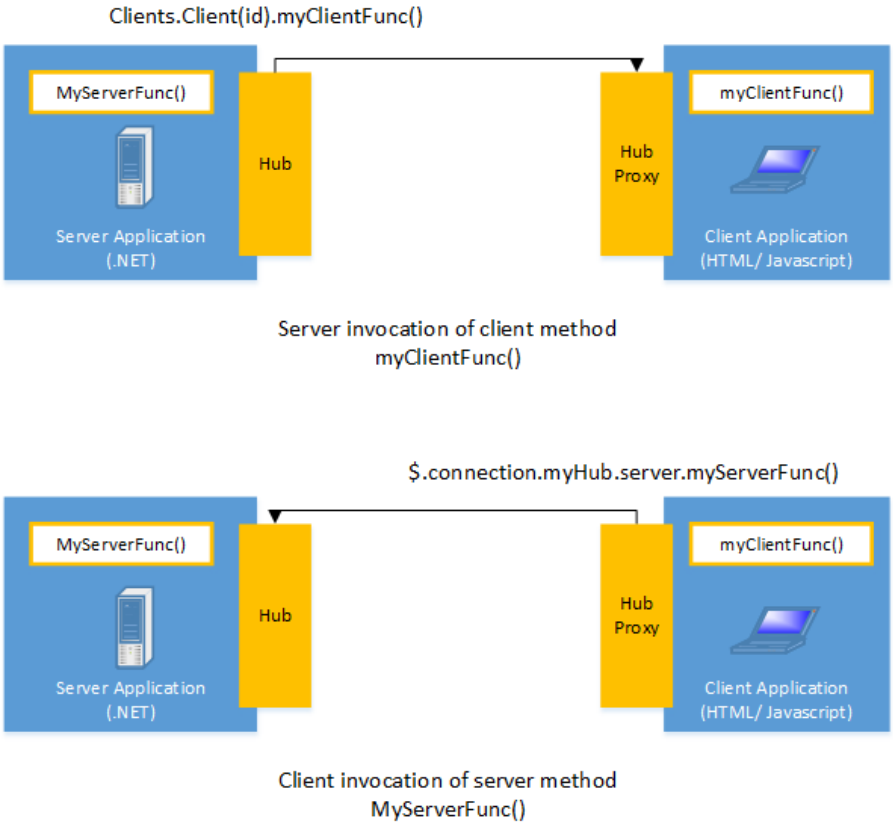
Entity Framework to zestaw technologii *ADO.NET*, obsługujące opracowywanie aplikacji zorientowanych na dane. Framework ten pozwala modelować konkretne jednostki, relację czy też logikę działania bazy danych. Obsługuję wiele systemów magazynowania danych takich jak na przykład *MySQL*, który to został przez nas wybrany do realizacji projektu. Całość jest konfigurowalna z poziomu kodu co jest bardzo wygodnym rozwiązaniem. Pozwala deweloperom na współpracę z danymi w postaci obiektów. Model fizyczny jest rafinowany przez administratorów bazy danych w celu zwiększenia wydajności, ale programiści piszący kod aplikacji przede wszystkim zawężają się do pracy z modelem logicznym, pisząc zapytania *SQL* i wywołując procedury składowane. Modele domen są zwykle używane jako narzędzie do przechwytywania i komunikowania się z wymaganiami aplikacji, często tak jak w przypadku diagramów obojętnych, które są wyświetlane i omówione w wczesnych etapach projektu, a następnie porzucone. Wiele zespołów programistycznych pomija Tworzenie modelu koncepcyjnego i rozpoczyna się od określenia tabel, kolumn i kluczy w relacyjnej bazie danych.



Rysunek 3: Architektura Entity Framework do uzyskania dostępu do danych

2.3 SignalR

Biblioteka dla dewelpoarów *ASP.NET*, która przyśpiesza proces dodawania funkcji sieci web w czasie rzeczywistym do aplikacji. Często wykorzystuje się tą bibliotekę w czatach internetowych, lecz może ona o wiele więcej. *SignalR* może być używany też w takich aplikacjach jak pulpity nawigacyjne, aplikacje do monitorowania czy formularze działające w czasie rzeczywistym. Biblioteka umożliwia również zupełnie nowe typy aplikacji internetowych, które wymagają aktualizacji wysokiej częstotliwości z serwera, na przykład gier w czasie rzeczywistym. Komunikacja między stroną a serwerem odbywa się poprzez tak zwane Hub'y. Bardzo dobrze ukazuje to rysunek poniżej.



Rysunek 4: SignalR przykład działania

2.4 MySql

Baza danych rozwijana przez firmę *Oracle*. *MySql* jest to otwarto źródłowy system zarządzania relacyjnymi bazami danych. System ten obsługuję język zapytań *SQL*, który służy do pisania zapytań do tej bazy. *MySql* jest relacyjną bazą danych. Model relacyjny w prosty i intuicyjny sposób przedstawia dane w tabelach. Każdy wiersz w tabeli jest rekordem z unikatowym identyfikatorem zwanym kluczem. Kolumny tabeli zawierają atrybuty danych, a każdy rekord zawiera zwykle wartość dla każdego atrybutu, co ułatwia ustalenie relacji między poszczególnymi elementami danymi.

Rysunek 5: Logo MySQL

2.5 Angular

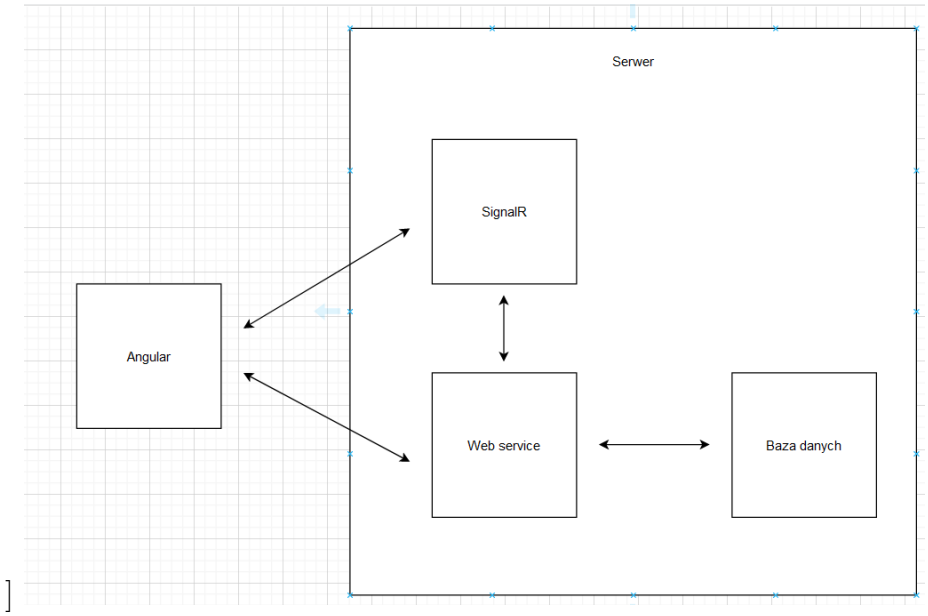
Otwarto źródłowy framework *JavaScript*, zaprojektowany i napisany przez inżynierów z *Google*. Ich celem było zrewolucjonizowanie projektowania części wizualnej stron internetowych. Szybko zyskał popularność wśród programistów JavaScript, którzy zaczęli odstawiać go na rzecz *jQuery*. Jego największą zaletą oraz najbardziej rozpoznawalną cechą jest integracja z atrybutami *HTML*. Framework ten umożliwia proste wdrożenie wzorca *MVC(Model-View-Controller)*, dzięki czemu testowanie jak i rozwój aplikacji nie sprawia wielu problemów.

Rysunek 6: Logo Angular

3 Projekt

3.1 Struktura projektu

Diagram pokązujący architekturę całego projektu opisuje główne komponenty i występujące między nimi połączenia. Strona internetowa napisana w Angularze jest podłączona bezpośrednio z *SignalR* i web serwisem. Miedzy nim występują połączenia dwukierunkowe. Z tym pierwszym strona internetowa utrzymuje stałe połączenie, aby zapewnić natychmiastową wymianę danych. Drugi wymieniony komponent odpowiada na zadane żądania. *SignalR* został wbudowany w web serwis jako integralna część, ale jest traktowany osobno, ponieważ utrzymuje z klientem innego typu połączenie. Między nimi także następuje wymiana danych. Ostatnim komponentem projektu jest baza danych, która jest podłącza tylko do web serwisu. To on stanowi most do pobierania wszelkich zawartych tam informacji.

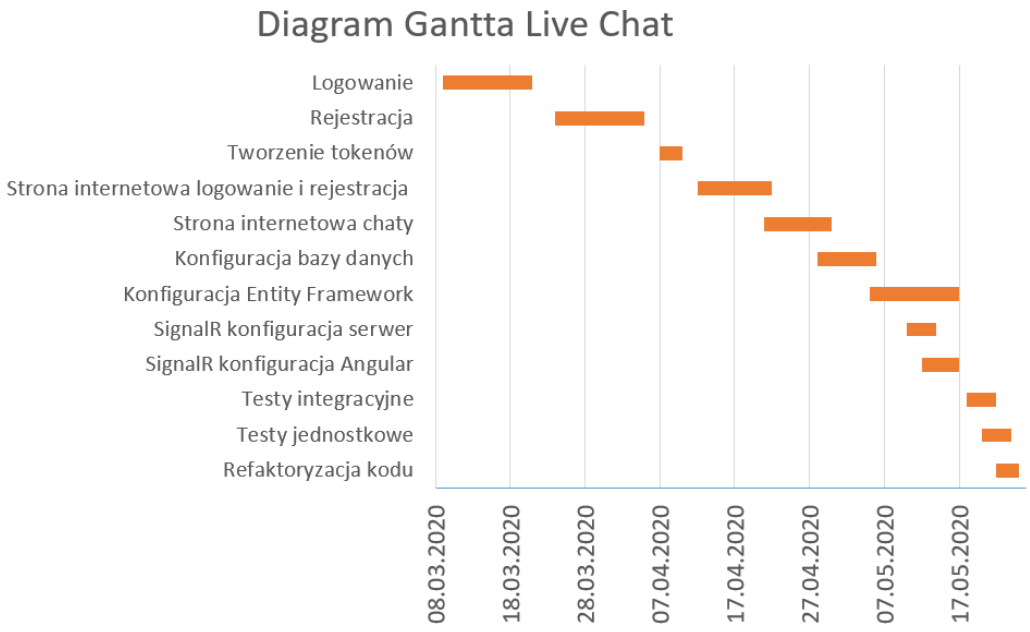


]

Rysunek 7:

3.2 Diagram Gantta

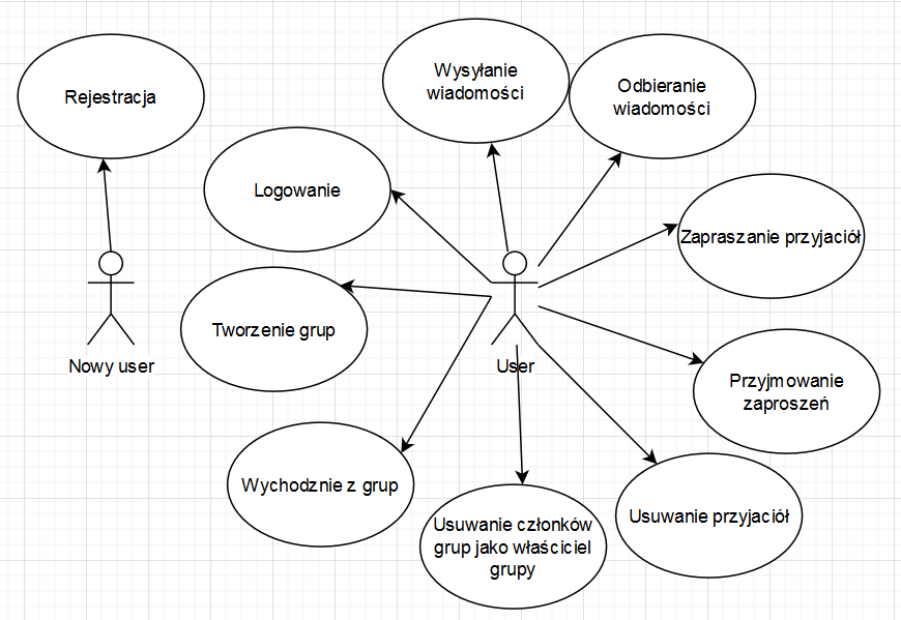
Diagram Gantta dla naszego projektu przedstawia się następująco:



Rysunek 8: Diagram Gantta Live Chat

3.3 Use Cases

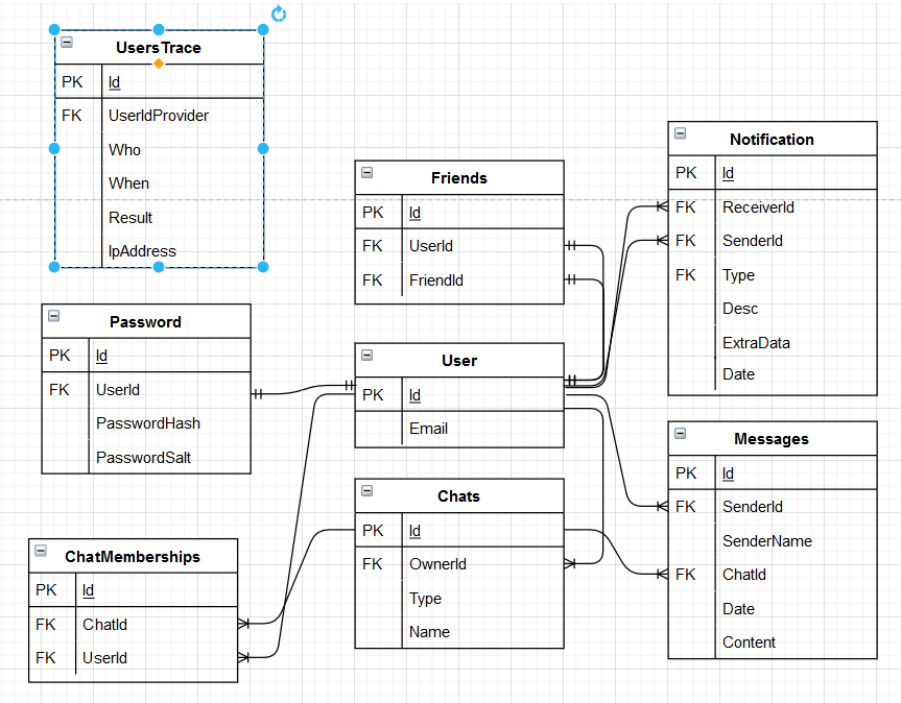
Use case pokazują fukcjonalności projektu z podziałem na rodzaje użytkownik danej aplikacji. W live chat bedziemy wyrózniać dwa rodzaje użytkowników. Będzie to nowy user, który będzie miał możliwość jedynie rejestracji i drugi to będzie już utworzony user, który będzie mógł korzystać ze wszystkich korzyści programu.



Rysunek 9:

3.4 Struktura bazy danych

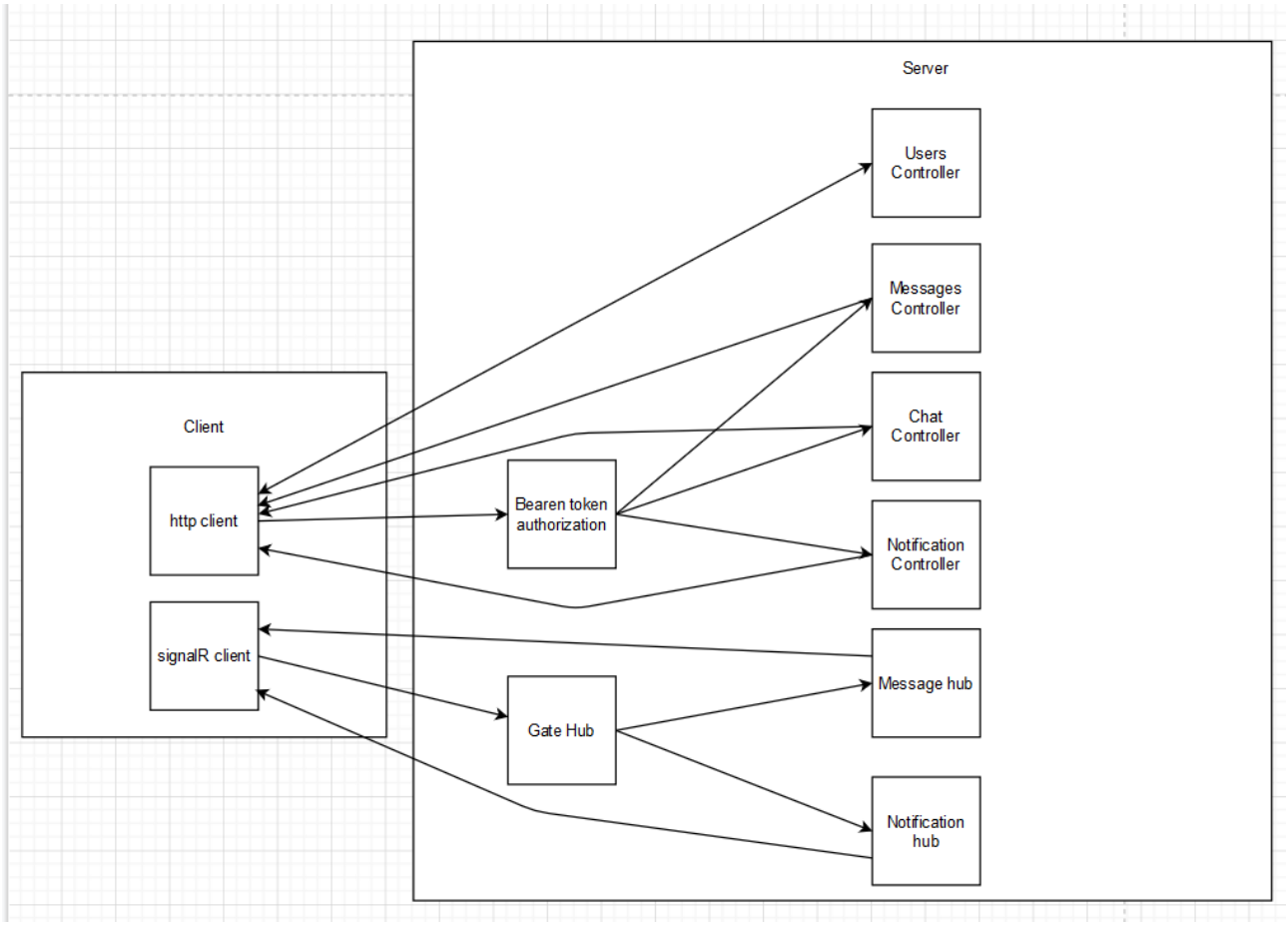
Struktura bazy danych jest prosta i czytelna. Dla takiego projektu baza danych jest fundamentem. Musimy mieć miejsce do przechowywania informacji o użytkownikach, relacjach, wiadmościach czy czatach. Tabele opieraja się na relacjach między sobą. Sama struktura w mysql została wygenerowana przez *Entity Framework Core*, który zrobił w najbardziej optymalny i wydajny sposób.



Rysunek 10:

3.5 Komunikacja z web serwisem

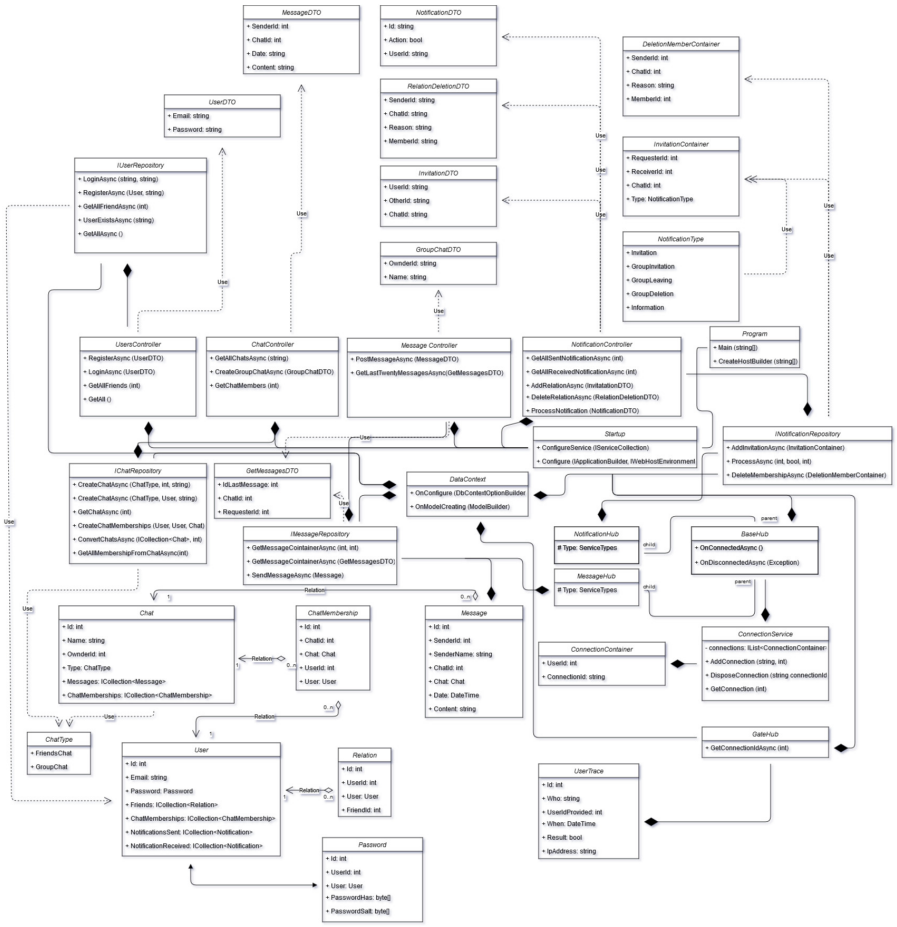
Komunikacja z web serwisem została podzielona na część dla każdego usera i część tylko dla zalogowanych. Po zalogowaniu wytwarzany jest token w *User Controller* i przesyłany z powrotem do klienta. Takowy token jest wykorzystywany do autoryzacji podczas dostępu do pozostałych kontrolerów. W przypadku *SignalR*, tylko zalogowani użytkownicy mogą się podłączyć. Wywoływana jest metoda RPC do walidacji użytkownika. Jest to druga część autoryzacji co czyni ją dwuetapową. Ta część polega na sprawdzeniu czy email zawarty w tokenie zgadza się z id usera. Wywoływana jest wówczas baza danych do pobrania emailu zawartego w rekordzie o danym id i przyrównanie tej wartości z wartością email zawartą w tokenie. Prosty diagram komunikacji z web serwisem jest następujący.



Rysunek 11:

3.6 Diagram klas

Diagram klas przedstawia od strony czysto technicznej połączenia między poszczególnymi klasami, interfejsami czy typami prostymi. Pozwala zauważyć pewne założenia, schematy, które ułatwią implementowanie całego mechanizmu, aby uniknąć powtarzania się kodu czy zminimalizować ilość użytych linijek.



Rysunek 12:

4 Implementacja

4.1 Modele bazy danych

Baza danych dla naszego projektu musiała być bardzo rozbudowana. W trakcie samego tworzenia, modeli ciągle przybywało. Jako że używaliśmy Entity Frameworka zdecydowaliśmy się na użycie adnotacji dla klas modeli. Skorzystaliśmy z bibliotek *System.ComponentModel.DataAnnotations*. Jedenym z przykładowych lecz bardziej rozbudowanych modeli użytych w naszej bazie danych znajduje się na listingu poniżej, dana klasa została przygotowana dla tabeli zawierającej wiadomości.

Listing 1: Przykładowa klasa modelu tabeli - Message

```
public class Message
{
        [Key]
        public int Id { get; set; }

        [Required]
```

```
        [ForeignKey("Sender")]
        [JsonIgnore]
        public int SenderId { get; set; }

        [JsonIgnore]
        public virtual User Sender { get; set; }

        public string SenderName { get; set; }

        [Required]
        [ForeignKey("Chat")]
        public int ChatId { get; set; }

        [JsonIgnore]
        public virtual Chat Chat { get; set; }

        public DateTime Date { get; set; }

        public string Content { get; set; }
    }
```

Jak widać na powyższym listingu. Model ten zawiera takie adnotacje:

- **Key** - określa że dane pole jest kluczem głównym;
- **Required** - informuje żę dane pole musi być wypełnione;
- **ForeignKey** - oznacza daną zmienną jako klucz obcy oraz w nawiasie przyjmuje nazwę tabeli której jest to klucz obcy;
- **JsonIgnore** - adnotacja wskazuje że dane pole bądź metodę należy zignorować podczas deserializacji danych.

4.2 Modele DTO

Model DTO (Data Transfer Object) jest bardzo wygodnym wzorcem projektowym należącym do grupy wzorców dystrybucji. Dzięki DTO można w prosty sposób transferować dane między systemami czy aplikacjami. W naszym projekcie potrzebowaliśmy stworzyć wiele modeli DTO w celu komunikacji *Strona web - Serwer*. Dobrze skonstruowany kontener, bo tak często nazywana jest klasa będąca modelem DTO, często staje się podstawową konstrukcją w wielu rozbudowanych projektach. W naszym projekcie znajduję się aktualnie aż osiem modeli DTO. Posiadamy modele potrzebne do transportu wiadomości czy też powiadomień. Jak wygląda przykładowy model DTO użyty w naszym projekcie znajduje się poniżej:

Listing 2: Przykładowa klasa modelu DTO - Message

```
public class InvitatationDTO
{
```

```
    [Required]
    [RegularExpression(@"^[1-9]\d*$", ErrorMessage = "'0' Can not be used")]
    public string UserId { get; set; }

    [Required]
    [RegularExpression(@"^[1-9]\d*$", ErrorMessage = "'0' Can not be used")]
    public string OtherId { get; set; }

    public string ChatId { get; set; }
}
```

Powyższy listing pokazuje że w projekcie użyliśmy również wyrażeń regularnych w celu sprawdzania poprawności przesyłanych danych. W momencie gdy przesłany przez stronę *Model DTO* zawiera dane nie zgodne z wyrażeniem regularnym, serwer automatycznie zwracał błąd że jakieś dane są nie poprawne. Dzięki temu mogliśmy pominąć sprawdzanie wielu przypadków przesyłania błędnych danych.

4.3 Implementacja kontrolerów

Kontrolery w apliacjach asp.net core opartych o usługi RESTful to próg wejścia do aplikacji ze strony klienta HTTP. Każdy z nich odpowiada za obsługę jednego z modeli aplikacji. Tymi modelami są kontenery na dane, gdzie te dan są tylko przechowywane. Wszystkie operacje na danym modelu są wykonywane w kontrolerze. Od strony protokołu przesyłu danych służą one do przechwytywania i obsługiwania przychodzących żądań i odsłynia odpowiedzi. W aplikacji powstały cztery kontrollery. Są to:

1. **UserController** - jego działanie skupia się na rejestrowaniu nowych użytkowników i logowaniu juz isteniejących. W drugim przypadku zwracany jest token przypisany do zalogowanego użytkownika. Za pomocą tego kontrolera można także pobrać wszystkich użytkowników oraz przyjaciół danego użytkownika. Istnieje także opcja sprawdzenia czy dany użytkownik jest przyjaciel z innym.

   Lista metod:

   - – Metoda: Post
     – Url: /Users/login
     – Typ przyjmowany: UserDTO
     – Typ zwracany: LoginResponse
     – Status kody:
       * Sukces - 200
       * Niepowodzenie - 400, 401
   - – Metoda: Post
     – Url: /Users/register
     – Typ przyjmowany: UserDTO
     – Typ zwracany: Void
     – Status kody:
       * Sukces - 201

        * Niepowodzenie - 400
- – Metoda: Get
  – Url: /Users/friends/userId
  – Typ przyjmowany: Integer
  – Typ zwracany: Array¡User¿
  – Status kody:
    * Sukces - 200
    * Niepowodzenie - 400
- – Metoda: Get
  – Url: /Users/friends/userId/friendId
  – Typ przyjmowany: Integer, Integer
  – Typ zwracany: Bool
  – Status kody:
    * Sukces - 200
    * Niepowodzenie - 400
- – Metoda: Get
  – Url: /Users/getAll
  – Typ przyjmowany: Integer, Integer
  – Typ zwracany: Array¡Users¿
  – Status kody:
    * Sukces - 200
    * Niepowodzenie - 400

2. **MessagesController** - w tym przypadku obsługiwany model to Message, czyli klasa zawierające dane o wiadomości. Kontroler obsługuje wysyłanie wiadomości z wykorzystaniem SignalR. Znajduje się również metoda do pobrania ostatnich dwudziestu wiadmości od wiadomości wysłanej w żądaniu. Kontroler nie posiada funkcji pobrania wszystkich wiadomości, ponieważ byłoby to niewydajne w przypadku dużej ilości danych.

   Lista metod:

- – Metoda: Post
  – Url: /Messages/post
  – Typ przyjmowany: MessageDTO
  – Typ zwracany: String
  – Status kody:
    * Sukces - 200
    * Niepowodzenie - 400
- – Metoda: Post
  – Url: /Messages/getLastTwentyMessages
  – Typ przyjmowany: GetMessagesDTO
  – Typ zwracany: Array¡Message¿
  – Status kody:

        * Sukces - 200
        * Niepowodzenie - 400, 404

3. **NotificationController** - najbardziej rozbudowany kontroler w całej aplikacji. Posiada on funkcjonalności odpowiedzialne za obsługę modelu Notification, który przedstawia powiadomienia. Wykorzystuje on SignalR do wysyłania powiadomień jak i do ich odbierania. Znajdują się w nim metody zwracające wszystkie wysłane i odebrane powiadomienia użytkownika. Tutaj dodaje się zaproszenia oraz usuwa się przyjaciół. Całość zamyka metoda do obsłużenia danego powiadomienia, na przykład zaproszenia można przyjąć po czym następuje stworzenie nowej relacji z czatem dla nowych przyjaciół oraz przygotowanie nowego powiadomienia zwrotnego.

   Lista metod:

   - – Metoda: Get
     – Url: /Notification/getAllSent/userId
     – Typ przyjmowany: Integer
     – Typ zwracany: Array¡Notification¿
     – Status kody:
       * Sukces - 200
       * Niepowodzenie - 400
   - – Metoda: Get
     – Url: /Notification/getAllReceived/userId
     – Typ przyjmowany: Integer
     – Typ zwracany: Array¡Notification¿
     – Status kody:
       * Sukces - 200
       * Niepowodzenie - 400
   - – Metoda: Post
     – Url: /Notification/invitation
     – Typ przyjmowany: InvitationDTO
     – Typ zwracany: String
     – Status kody:
       * Sukces - 200
       * Niepowodzenie - 400
   - – Metoda: Post
     – Url: /Notification/deleteRelation
     – Typ przyjmowany: RelationDeletionDTO
     – Typ zwracany: Bool
     – Status kody:
       * Sukces - 200
       * Niepowodzenie - 400, 404
   - – Metoda: Post

- Url: /Notification/proccess
- Typ przyjmowany: NotificationDTO
- Typ zwracany: Void
- Status kody:
  * Sukces - 200
  * Niepowodzenie - 400

4. **ChatController** - obsługiwany model to Chat, który przechowuje informacje na temat wszystkich userów w nim uczęstniczacych jak i wiadomościach znajdujacych się w nim. Wykonywanie są tu operacje pobierania danego czatu oraz danego wszystkich czatów danego użytkownika. Tworzone są tutaj oczywiście czaty, ale tylko grupowe. Czaty przyjacielskie są tworzene w kontrolerze powiadomień. Ostatnią metodą jest pobranie wszystkich użytkowników należących do danego czatu.

   Lista metod:

   - Metoda: Post
     - Url: /Chat/getAll/id
     - Typ przyjmowany: Integer
     - Typ zwracany: Array¡ChatDTO¿
     - Status kody:
       * Sukces - 200
       * Niepowodzenie - 400
   - Metoda: Post
     - Url: /Chat/createGroupChat
     - Typ przyjmowany: GroupChatDTO
     - Typ zwracany: ChatDTO
     - Status kody:
       * Sukces - 200
       * Niepowodzenie - 400
   - Metoda: Get
     - Url: /Chat/getChatMembers/id
     - Typ przyjmowany: Integer
     - Typ zwracany: Array¡User¿
     - Status kody:
       * Sukces - 200
       * Niepowodzenie - 400

W każdym kontrolerze używana jest baza danych, ale nie bezpośrednio w nim. Repozytoria obsługują głębszą logikę obsługiwania żadań. Są to klasy do obsługi pewnych powtarzających się lub większych fukcjonalności. Posiadają dostęp do bazy danych, SiganlR oraz serwisów działających w aplikacji. Są skonfigurowane jako jedna instacja dla każdego połączenia. Bezpośrednio w kontrolerach odbywa się walidacja żadań i generowanie odpowiedzi.

4.4 Implementacja SignalR

Dużym wyzwaniem było zaimplementowanie ciągłego połączenia z klientem. Połączenie HTTP na bazie kontrolerów nie spełnia tego rozwiązania ponieważ jest bezstanowe i działa na zasadzie wysłania żądania i otrzymania odpowiedzi. Najlepszym rozwiązaniem w tym przypadku jest SignalR. Jego implementacja jak i obsługa jest bezproblemowa. Zawiera wszystkie ważne komponenty i fukcjonalności. Jego popularność zawdzięcza wysokiej wydajności i bardzo dobrej osbłudze połączenia.

SignalR jest frameworkiem ASP.NET ułatwiającym pisanie aplikacji czasu rzeczywistego, czyli takich, gdzie nie trzeba odświeżać przeglądarki. Mechanizmy takie jak WebSocket czy HTML SEE są podstawowym mechnizmem w takich aplikacjach, ale wymagają wparcia od przeglądarki. SignalR zapewnia jeden framework niezależny od przeglądarki.

SignalR zapewnia dwa typy połączenia: PersistentConnection oraz HubAPI. My skorzystaliśmy z HubAPI, jest prostsze w obsłudze, a równie wydajne jak pierwsze rozwiazanie. Dodatkowo o umożliwia wykonywanie RPC nad połączniem oraz przesyłanie *"strongly typed"* parametrów.

Po stronie serwera implementacja SignalR dzieli się na 3 Hub's. Pierwszym z nich jest GateHub, który posiada RPC, do weryfikacji połączenia. Jest to drugi stopień autoryzacji użytkownika. Kolejnymi dwoma hubami są klasy odpowiedzialne za wiadomości i notyfikację. Ze względu na wspólne ciało dziedziczą one od klasy BaseHub.

Listing 3: Implementacja Klasy BaseHub

```
[Authorize]
public abstract class BaseHub : Hub
{
        protected abstract ServiceTypes Type { get; }

        private IConnectionService _connectionService;

        public BaseHub(Func<ServiceTypes,
        IConnectionService> servicesResolver)
        {
                _connectionService = servicesResolver(Type);
        }

        public override Task OnConnectedAsync()
        {
                if(int.TryParse(Context.UserIdentifier,
                 out int userId))
                {
                        _connectionService.AddConnection
                        (Context.ConnectionId, userId);

                        return base.OnConnectedAsync();
                }

                throw new Microsoft.AspNetCore.SignalR
```

```
                .HubException("Cannot parse user id");
        }

        public override Task OnDisconnectedAsync(Exception exception)
        {
                _connectionService.
                DisposeConnection(Context.ConnectionId);

                return base.OnDisconnectedAsync(exception);
        }
}
```

Posiadaja one atrybut *[Authorize]*, ponieważ za jego pomocą mogą zwryfikować token danego użytkownika.

Po stronie klienta zostały również zaimplementowane 3 rodzaje hubów. Ważne było zastosowanie wspólnej nazwy dla przesyłanego strumienia. Dzięki niemu połączenia miedzy konkretnymi hubami mogło się udać. Jego autonomiczność wspiera również osobny adres, który trzyba podać aby się połączyć do danego Hub. Każdy z Hub's po stronie klienta to osobny serwis działający w tle.

Listing 4: Implementacja SignalR po stronie klienta

```
@Injectable({
providedIn: 'root'
})
export class MessageSignalRService {

        private hubConnection: signalR.HubConnection;

        messageData: Subject<any> = new Subject<any>();

        constructor() { }

        startConnection(token: string) {
                const options: IHttpConnectionOptions = {
                accessTokenFactory: () => {
                        return token;
                        }
                };
        this.hubConnection = new signalR.HubConnectionBuilder()

        .withUrl('http://localhost:53064/messagechart', options)
        .build();

        this.hubConnection
        .start()
        .then(() => console.log('Connection started'))
        .catch(err => console.log('Error while starting connection: '
```

```
        + err));
      }

      getMessageData(): Observable<any> {
      return this.messageData.asObservable();
      }

      addTransferChartDataListener() {
            console.log('listen');
            this.hubConnection.on('transferchartdata', (data) => {
                  console.log('Just Received message');
                  this.messageData.next(data[0]);
            });
      }
}
```

5 Testy

5.1 Testy jednostkowe

5.2 Testy integracyjne

6 Opis wkładu własnego w realizację projektu

6.1 Stworzenie i konfiguracja bazy danych

Złożoność oraz poziom zaawansowania naszego projektu był bardzo duży co wymagało rozbudowanej bazy danych. Budowę oraz połączenie między tabelami w bazie danych ukazuje rysunek numer 10. Praktycznie każda tabela posiada klucz obcy innej tabeli. Spowodowane to jest tym iż specyfika naszego projektu wymaga od nas takiego połączenia. Każdy czat musi być przypisany do konkretnych użytkowników, każda wiadomość do konkretnego czatu. Powiadomienia w podobny sposób co wiadomości czy czaty muszą posiadać przypisanego konkretnego użytkownika, a do tego wszystkiego jeszcze dochodzą czaty grupowe co wymagało od nas dodatkowego przemyślenia działania całej infrastruktury aplikacji.

Po wielu problemach i próbach ich rozwiązania udało nam się skonfigurować *Entity Framework* do oczekiwanego przez nas stopnia. Większość tabel i połączeń między nimi bardzo dobrze się skonfigurowało poprzez adnotacje nadane polom w modelach bazy danych. Sam nasz *DbContext* wyglądał w sposób następujący:

Listing 5: DbContext Entity Framework

```
public DbSet<Password> Passwords { get; set; }
public DbSet<User> Users { get; set; }
public DbSet<Message> Messages { get; set; }
public DbSet<Relation> Friends { get; set; }
```

```
public DbSet<Notification> Notifications { get; set; }
public DbSet<Chat> Chats { get; set; }
public DbSet<ChatMembership> ChatMemberships { get; set; }
public DbSet<UserTrace> UsersTrace { get; set; }
```

Powyższa część odpowiada za utworzenie tabel *DBSet* z nazwą modelu we wnętrzu trójkątnych nawiasów. Niestety nie wszystkie tabele chciały się w ten sposób utworzyć. Jedyną tabelą, którą musieliśmy utworzyć w trochę inny sposób okazała się tabela *Notyfication* odpowiadająca za przetrzymywanie informacji na temat powiadomień. Tą jedną tabelę utworzyliśmy używając wbudowanych dodatkowych funkcji *Entity Frameworka*, nadpisując funkcję *OnModelCreating* byliśmy w stanie utworzyć tabelę wraz jej połączeniami. Poniższy listing ukazuje naszą rozbudowę funkcji:

Listing 6: OnModelCreating tabela Notyfication - Entity Framework

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<Notification>()
        .HasOne(t => t.Sender)
        .WithMany(m => m.NotificationsSent)
        .HasForeignKey(p => p.SenderId);

        modelBuilder.Entity<Notification>()
        .HasOne(t => t.Receiver)
        .WithMany(m => m.NotificationsReceived)
        .HasForeignKey(p => p.ReceiverId);
}
```

6.2 Stworzenie systemu logowania i rejestracji

Stworzenie przez nas systemu logowania i rejestracji było bardzo skoplikowanym procesem. Przemyśleć musieliśmy jakiego algorytmu szyfrowania haseł użyć by hasła użytkowników przechowywane w bazie danych były bezpieczne. Do szyfrowania haseł użyliśmy wbudowanego w język *C#* algorytmu wyznaczania wartości skrótu zwangeo *HMAC-SHA512*, który to zbudowany jest z funkcji skrótu *SHA-512* oraz oparty jest o uwierzytelnianie wiadomości opartych o skróty *HMAC*. Proces *HMAC* miesza klucz tajny z danymi komunikatów i miesza wynik. Wartość skrótu jest ponownie mieszana z kluczem tajnym, a następnie poddana skrótu po raz drugi. Wynikowy skrót ma długość 512 bitów.*HMAC* działa w sposób następujący:

$$\mathbf{HMAC}_K(m) = h\Big((K \oplus \mathbf{opad}) \| h\big((K \oplus \mathbf{ipad}) \| m\big)\Big),$$

Rysunek 13: HMAC Signature

Podczas rejestracji szyfrowane jest hasło po stronie serwera oraz zaszyfrowane zapisywane jest w bazie danych. W celu zapobiegnięcia odszyfrowania, hasło jest szyfrowane w jedną stronę. Dzięki temu nawet my jako twórcy nie jesteśmy w stanie od szyfrować dla własnych potrzeb hasła użytkownika. Logowanie przeprowadza podobny proces co rejestracja lecz nie zapisuje hasła w bazie. Hasło wpisane przez użytkownika jest szyfrowane po stronie serwera i już zaszyfrowane hasło jest porównywane z zaszyfrowanym hasłem w bazie danych. Jeśli wartości są takie same użytkownik zostaje poprawnie zarejestrowany w aplikacji i uzyskuje dostęp do wszystkich funkcji systemu.

6.3 Autoryzacja i autentykacja użytkowników

W celu autoryzacji oraz autentykacji użytkowników po stronie serwera. Tworzony jest token. Token jest to ciąg znaków identyfikujący aktualnie zalogowanego użytkownika. Tworzony jest w taki sposób by nie było możliwości powtórzenia się tokena.

Listing 7: Tworzenie tokena przy logowaniu

```
var tokenDescriptor = new SecurityTokenDescriptor
{
        Subject = new ClaimsIdentity(claims),
        Expires = DateTime.Now.AddHours(12),
        SigningCredentials = creds
};

var tokenHandler = new JwtSecurityTokenHandler();
var token = tokenHandler.CreateToken(tokenDescriptor);
```

Jak widać na powyższym listingu, do tworzenia tokena uwierzytelniającego użytkownika wykorzystujemy klasę *SecurityTokenDescriptor*. Dzięki tej klasie jesteśmy w stanie przekazać jej wszystkie potrzebne informacje do utworzenia tokena. Token jest ważny przez 12 godzin ale jeśli użytkownik zamknie kartę z otwarta naszą stroną automatycznie token jest usuwany. Natomiast gdy użytkownik przekroczy czas dwunastu godzin zostanie poproszony o ponowne zalogowanie. Po utworzeniu tokena przesyłamy w formacie *JSON* token wraz z informacjami o zalogowanym użytkowniku do aplikacji w przeglądarce.

6.4 Konfiguracja środowiska i serwera

Konfiguracja aplikacji rozrastała się w raz z powstawaniem nowych funkcji. Przed rozpoczęciem działania aplikacji odpalana jest konfiguracja, która trzeba najpierw skonfigurować. Klasa *Startup* metody takie jak *ConfigureService* oraz *Configure*. Zacznijmy od tej pierwszej. Metoda *ConfigureServices* jest metodą opcjonalną. Wywoływana jest przez hosta przed uruchomieniem metody *Configure*. W metodzie tej wszystkie opcje konfiguracyjne ustawiane są wraz z konwencją przedstawianą w dokumentacji firmy.

6.4.1 ConfigureServices

W tej metodzie jako pierwsze ustawiamy dostęp do bazy danych oraz wskazujemy do jakiej bazy chcemy się łączyć. W naszym przypadku łączymy się do bazy *MySQL*.

Listing 8: Dodanie bazy danych do konfiguracji

```
services.AddDbContext<DataContext>(option =>
{
 option.UseLazyLoadingProxies();
 option.UseMySql(Configuration.GetConnectionString("DefaultConnection"));
});
```

ConnectionString jest pobierany z pliku konfiguracyjnego. Opcja *UseLazyLoadingProxies* powoduje tak zwane Leniwe Ładowanie, które polega na każdorazowym odpytywaniu bazy o interesujące nas dane. Ten typ ładowania jest wykorzystywany gdy chcemy pobrać jakieś konkretne dane z bazy a nie chcemy zaśmiecać sobie pamięci masą innych niepotrzebnych danych.

Jako drugie ustawiamy zakres. Zakresy są potrzebne do bazy danych. Informujemy w ten sposób które interfejsy oraz klasy należą do kontroli bazy danych. Dodajemy je w następujący sposób:

Listing 9: Dodanie klas Repository

```
services.AddScoped<IUserRepository, UserRepository>();
services.AddScoped<INotificationRepository, NotificationRepository>();
services.AddScoped<IMessagesRepository, MessageRepository>();
services.AddScoped<IChatRepository, ChatRepository>();
```

Powyższy listing przedstawia w jaki sposób dodajemy klasy oraz interfejsy, które są repozytoriami dla danych tabel w bazie danych.

Następną częścią jaką dodajemy do naszej konfiguracji są wzorce projektowe *Singleton*. *.Net Core* posiada wbudowane metody pozwalające na tworzenie konkretnych klas na serwerze jako *Singletony*. W jaki sposób to się robi przedstawia poniższy listing z naszego programu:

Listing 10: Wzorzec Singleton w konfiguracji

```
services.AddSingleton<MessageConnectionService>();
services.AddSingleton<NotificationConnectionService>();
services.AddSingleton<Func<ServiceTypes, IConnectionService>>(provider => s
{
        switch(serviceType)
        {
        case ServiceTypes.MessageConnectionService:
                return provider.GetService<MessageConnectionService>();
        case ServiceTypes.NotificationConnectionService:
                return provider.GetService<NotificationConnectionService>(
        default:
                return null;
        }
});
```

Kolejną bardzo ważną częścią jak nie najważniejszą w konfiguracji jest autentykacja i autoryzacja. W naszej aplikacji użyliśmy tak zwanego *BearerTokena*. Określamy wszelkie ustawienia jego jak i to jakim kluczem jest szyfrowane oraz podajemy token potrzebny do tworzenia klucza. Tak jak w przypadku *ConnectionString* tutaj pobierana jest wartość z pliku

zawierającego dane konfiguracyjne. Poniższy listing przedstawia jak skonfigurowany jest nasz *BearerToken*.

Listing 11: BearerToken - konfiguracja

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
 .AddJwtBearer(options =>
 {
 options.TokenValidationParameters = new TokenValidationParameters
 {
  ValidateIssuerSigningKey = true,
  IssuerSigningKey = new  SymmetricSecurityKey(Encoding.ASCII.GetBytes(
  Configuration.GetSection("AppSettings:Token").Value)),
  ValidateIssuer = false,
  ValidateAudience = false,
 };
 options.Events = new JwtBearerEvents
 {
  OnMessageReceived = context =>
  {
   var accessToken = context.Request.Query["access_token"];
  if (string.IsNullOrEmpty(accessToken) == false)
  {
   context.Token = accessToken;
  }
   return Task.CompletedTask;
  }
 };
});
```

Ostatnią częścią tej metody jest dodanie tak zwanej *Policy*. Ta część pozwala na ustawienie metod, nagłówków oraz ustawienie jaki adres jest tym domyślnym, bazowym. W naszym przypadku jako że to projekt, adres bazowy posiada u nas wartość *http://localhost:4200*. Na sam koniec dodajemy obsługę bibliotek *SingalR*. Poniższy listing przedstawia jak to robimy w kodzie:

Listing 12: Polityka oraz dodanie SingalR w konfiguracji

```
services.AddCors(o => o.AddPolicy("CorsPolicy", builder =>
{
        builder
        .AllowAnyMethod()
        .AllowAnyHeader()
        .WithOrigins("http://localhost:4200")
        .AllowCredentials();
}));

services.AddSignalR();
```

6.4.2 Configure

W tej metodzie w przeciwieństwie do metody wyżej opisanej, bardzo ważną role gra kolejność wywoływanych metod. Przepływ zwany również *Pipeline*, w tej części programu wszystko musi być ustawione w odpowiedniej kolejności, by zapewnić bezawaryjne działanie. Poniższy listing przedstawia jak wygląda nasza metoda *Configure*:

Listing 13: Metoda Configure

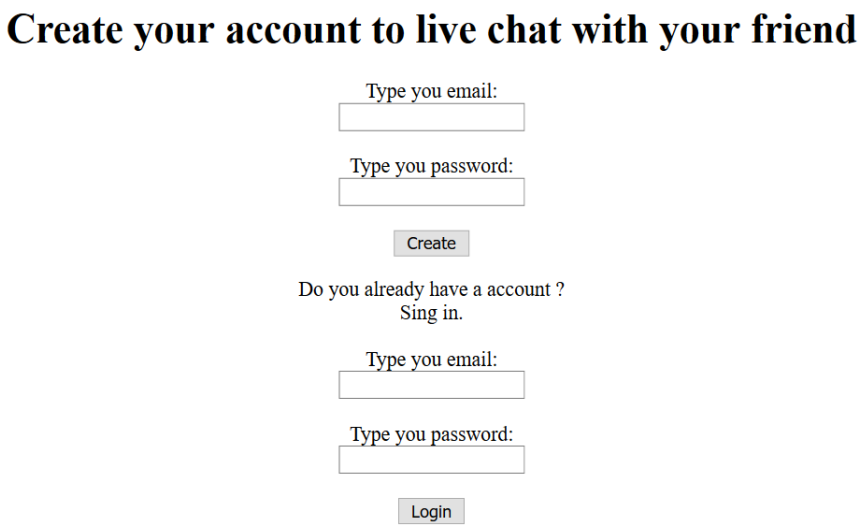
```
if (env.IsDevelopment())
{
        app.UseDeveloperExceptionPage();
}

app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();
app.UseCors("CorsPolicy");
app.UseForwardedHeaders(new ForwardedHeadersOptions
{
        ForwardedHeaders = ForwardedHeaders.XForwardedFor |
        ForwardedHeaders.XForwardedProto
});

app.UseEndpoints(endpoints =>
{
        endpoints.MapControllers();
        endpoints.MapHub<MessagesHub>("/messagechart");
        endpoints.MapHub<NotificationHub>("/notificationchart");
        endpoints.MapHub<GateHub>("/connection");
});
```

Na samym początku informujemy serwer że jeśli aplikacja jest włączana w trybie deweloperskim to aktywujemy *DeveloperExceptionPage*, stronę odpowiedzialną za pokazywanie szczegółowych informacji w momencie wystąpienia poważnego błędu po stronie serwera. Następnie włączamy możliwość *Routingu*, autoryzację oraz autetykację. Kolejnym krokiem jest aktywowanie polityk, które w metodzie *ConfigureServices* ustawialiśmy szczegółowo. Następnie informujemy serwer że zezwalamy na przekierowania informacji takich jak na przykład adresy IP czy porty. Wykorzystywane przy przełączaniu serwerów proxy. Ostatnią częścią można powiedzieć że nawet najważniejszą są tak zwane *EndPoints*. W tym miejscu mapujemy wszystkie kontrolery działające w aplikacji oraz ścieżki do nich.

6.5 Strona internetowa opracowana w Angular

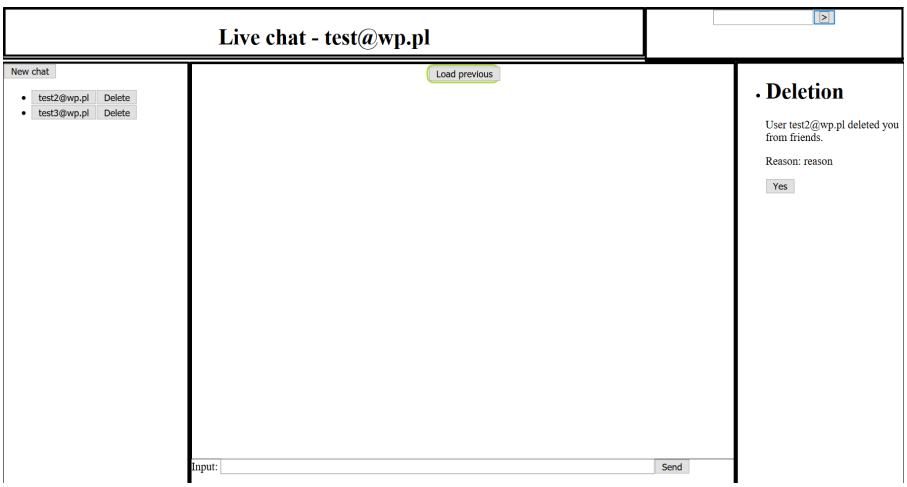
Strona internetowa powstała w popularnym frameworku Angular. Bazuje on na stworzynej przez Microsoft nakładkę na JavaScript TypeScript. Angular dostarcza deweloperom szeroki zakres możliwości manipulowania zachowaniem elementów wizualnej warstwy aplikacji webowych, które udało się wykorzystać w naszej aplikacji. Aplikację Angular podzieliliśmy na

sześć komponentów, które odpowiadają za przechowywanie i obsługę innego rodzaju danych. Wchodzą na stronę wyświetla się nam komponent **RegisterComponent** oraz **LoginComponent**. Służa one do wpisania kompletu danych kolejno do rejestracji lub logowania i wysłania informacji na serwer.



Rysunek 14: Strona logowania i rejestacji

Po zalogowaniu otrzymuję interface użytkownika. Po lewej stronie widoczne są wszytkie czaty. Jest to miejsce obsługiwane przez komponent **ListChatsComponent**. Na środku widać komponent **ChatComponent** przechowujący wiadomości aktualnie wybranego czatu. Prawy górny róg to komponent **SearchingUserComboBoxComponent**. Za jego pomocą możemy zaprosić danego użytkowniak do przyjaciół. Prawa strona to komponent **NotificationComponent** do wyświetlania i obsługiwania powiadomień.



Rysunek 15: Interfejs użytkownika

Całą obsługa klienta na stronie zaczyna się w klasie **ControlService**, która przechowuje wszystkie informacje na temat zalogowanego użytkownika. Rozsyła też te dane po wszystkie komponenty za pomocą frameworku **rxjs**. Słuzy do stworzenia wzorca obserwator na danej

zmiennej. Wystarczy wtedy tylko zasubsrykować tą zmienną i można odczytać każdą nową jej wartość podczas nowego przypisania.

Listing 14: Obsługa wzorca obserwator po stronie serwisu

```
chats: Subject<Array<Chat>> = new Subject();

public getChats(): Observable<Array<Chat>> {
        return this.chats;
}

public setChats(chats: Array<Chat>) {
        this.chats.next(chats);
}
```

Listing 15: Obsługa wzorca obserwator po stronie klienta

```
chats: Array<Chat>;

constructor(private controlService: ControlService) {
this.controlService.getLogged().subscribe( isLogged => {
        if (isLogged) {
                        this.controlService.getChats()
                        .subscribe(chats => {
                        this.chats = chats;
                });
        }
});
```

Do wysyłania danych do kontrolerow na serwerze służy serwis **HttpService**. Jest on podłączony bezpośrednio tylko do ControlSerivce.

Listing 16: Implementacja pobierania wiadmości

```
getLastMessages(lastMessagesController: LastMessagesController):
Observable<Array<Message>> {

        if (lastMessagesController.chatId === 0) {
                return EMPTY;
        }

        const headersLivechat = new HttpHeaders({
                'Authorization': 'Bearer ' + this.token,
        });


        return this.httpClient.post<Array<Message>>(this.url +
        'Messages/getLastTwentyMessages/', lastMessagesController,
```

```
        {headers: headersLivechat});
}
```

Listing 17: Obsługa otrzymanej z web serwisu odpowiedzi

```
getLastMessages() {
        let idLastMessage: number;
        if (this.messages.value == null ||
        this.messages.value.length === 0) {
                idLastMessage = 0;
        } else {
                idLastMessage = this.messages.value[0].id;
        }

        const lastMessageController: LastMessagesController
        = {idLastMessage,
        chatId: this.currentChat.value.id,
        requesterId: this.user.value.id};
        this.httpService.getLastMessages(lastMessageController)
        .subscribe(messages => {
                if (this.messages.value != null) {
                        this.messages.value.forEach(message => {
                                messages.push(message);
                                });
                        }
                this.messages.next(messages);
        });
}
```

Obsługa SignalR została zawarta w serwisie **HubsControllerService**. Jest on odpowiedzialny za uruchomienia połączenia i odczytywania danych. Została tu też zaimplmenetowana obsługą RPC z serwera do podwójnej autoryzacji w **ConnectionSignalRService**.

Listing 18: Uruchomienie RPC po stronie klienta do podwójnej autoryzacji.

```
this.hubConnection
        .start()
        .then(() => this.hubConnection
        .invoke<boolean>('GetConnectionId', userId)
        .then(value => {
                if (!value.valueOf()) {
                        this.isSuccessLogged.next(false);
                } else {
                        this.isSuccessLogged.next(true);
                }
        })
        .catch(err => {
                console.log('Error while starting connection: ' + err);
```

```
                this.isSuccessLogged.next(false);
        }));
        this.isSuccessLogged.next(false);
}
```

Za jej pomocą wystarczy odczytać nową wartość zmiennej **isSuccessLogged**, która dostarczy nam informacji na temat pomyślnego zalogowania. Jeśli wartość jest na true następuję uruchomienia Hub's odpowiedzialnych za nasłuchiwanie nadchodzących wiadmości i powiadomień.

Listing 19: Uruchomienie nasłuchiwania Hub's do wiadomości i powiadomień

```
this.connectionSignalRSerivce.getIsSuccessLoggedStatus()
.subscribe(x => {
        this.isSuccessLogged.next(x);

        if (x) {
                this.messageSignalRService
                .startConnection(this.token);
                this.messageSignalRService
                .addTransferChartDataListener();

                this.notificationSignalRService
                .startConnection(this.token);
                this.notificationSignalRService
                .addTransferChartDataListener();
        }
});
```

Przychodzące dane są odczytywane w ControlSerwis i rozsyłane też za pomoca wzorca obserwator.

6.6 Implementacja zarządzania wiadomościami

Obsługę wiadomości zaczyna się od kontrolera **MessagesController**. Tu są wiadomości wysyłane, a następnie zapisywane w bazie danych oraz odbierane. Do zapisania wiadomości służy metoda **PostMessage**, która zapisana jest pod adresem **/Messages/post**. Przyjmuje ona obiekt klasy **MessageDTO**.

Listing 20: Implementacja klasy MessageDTO

```
public class MessageDTO
{
        [Required]
        [MinLength(1, ErrorMessage = "Min 1")]
        [RegularExpression(@"^[1-9]\d*$",
         ErrorMessage = "'0' Can not be used")]
        public string SenderId { get; set; }
```

```
        [Required]
        [MinLength(1, ErrorMessage = "Min 1")]
        [RegularExpression(@"^[1-9]\d*$",
         ErrorMessage = "'0' Can not be used")]
        public string ChatId { get; set; }

        [Required]
        [DataType(DataType.DateTime,
         ErrorMessage = "Wrong DateTime format")]
        public string Date { get; set; }

        [Required]
        [MinLength(1, ErrorMessage = "Min 1")]
        public string Content { get; set; }
}
```

Cała operacja zapisania danych w bazie danych odbywa się w klasie **MessageRepository**. Tam obiekt EntityFramework jest zdefiniony.
Przebieg całej operacji polega na tym, że w kontrolerze sprawdzany jest obiekt DTO pod względem jego poprawności, a następnie z repozytorium wyciągany jest obiekt struktury **MessageContainer**.

Listing 21: Implementacja struktury MessageContainer

```
public struct MessageContainer
{
        public Chat Chat { get; set; }
        public User Sender { get; set; }

        internal void Deconstruct(out Chat chat, out User user)
        {
                chat = Chat;
                user = Sender;
        }
}
```

Następuje walidacja połączenia danego czatu z użytkownikiem, który wysłał wiadomość. Metoda z repozytorium zwraca null do obiektów *Chat* i *Sender* w przypadku, jeśli dany użytkowników nie należy do podanego czatu.

Listing 22: Walidacje wysyłania wiadomości w kontrolerze

```
if (int.TryParse(message.SenderId, out int senderId) &&
 int.TryParse(message.ChatId, out int chatid))
{
        (chat, user) = await _repository
        .GetMessageCointainer(senderId, chatid)
        .ConfigureAwait(true);
```

```
}
else
{
        return BadRequest("SenderId or receiverId has a wrong format.");
}

if(chat == null || user == null)
{
        return BadRequest("Cannot find chat or user.");
}
```

Po wykonaniu walidacji następuje zapisanie wiadomości w bazie danych. SignalR otrzymuje tą wiadomość i natychmiast wysyła ją do wszystkich odbiorców.

Listing 23: Wysłanie wiadomosci za pomocą SignalR

```
var messageChartModel = new MessageChartModel
{
        ChatId = chat.Id,
        SenderName = user.Email,
        Content = message.Content,
        Date = messageDate
};

object[] param = { messageChartModel };
IEnumerable<int> userIds = chat.ChatMemberships
.Select(x => x.User.Id);

foreach(int userId in userIds)
{
        string connectionId = _service.GetConnectionId(userId);
        if(connectionId != null && connectionId.Trim().Length != 0)
        {
                _ = _hub.Clients.Client(connectionId)
                .SendAsync("transferchartdata", param)
                .ConfigureAwait(true);
        }
}
```

Kolejną fukcjonalnością jest pobieranie dwudziestu ostatnich wiadomości. Aby to zrobić potrzebujemy informację, która wiadomość aktualnie jest ostatnia. Do metody **GetLastTwentyMessages** znajdującej się pod ad/**Messages/getLastTwentyMessages** wysyłany jest obiekt klasy **GetMessagesDTO**.

Listing 24: Implementacja klasy GetMessagesDTO

```
public class GetMessagesDTO
{
```

```
        public int IdLastMessage { get; set; }

        public int ChatId { get; set; }

        public int RequesterId { get; set; }
}
```

Po sprawdzeniu poprawności obiektu wywoływana jest metoda z repozytorium, która sprawdza relacje użytkownika z podanym czatem, następnie pobiera dwadziesta ostatnich wiadomości zaczynając od zawartej w powyższej klasie. Jeśli zmienna **IdLastMessage** jest równa 0, pobierane jest dwadzieścia najnowszych wiadomości w danym czacie.

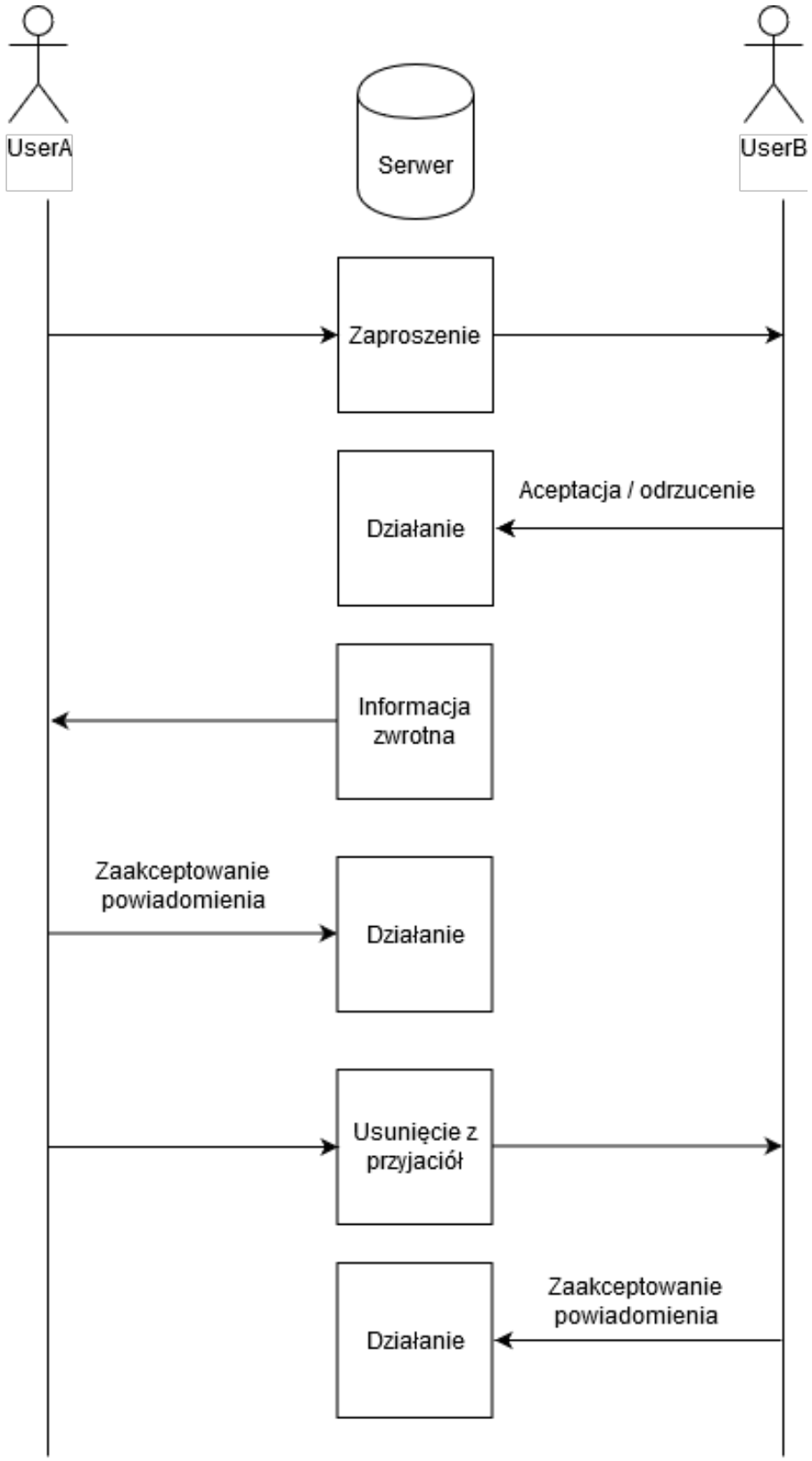
Listing 25: Implementacja pobierania ostatnich 20 wiadomości

```
int iterator = 0;
var messageWithCorrectedId = messagesToProccess.Where(x => x.Id <= getMessa

for(int i = messageWithCorrectedId.Count - 1; i >= 0; i--)
{
        if(messageWithCorrectedId[i] == null)
        {
                break;
        }

        messages.Add(messageWithCorrectedId[i]);

        iterator++;

        if (iterator >= 20)
        {
                break;
        }
}

return messages.Reverse();
```

I ostatecznie kontroler zwraca tablicę wiadomości.

6.7 Implementacja zarządzania powiadomieniami

Powiadomienia to najbardziej rozbudowana część całej aplikacji. Jest to wieloetapowy proces, który zawiera specyficzną kolejność.



Rysunek 16: Proces powiadomień w relacjach użytkowników

Kiedy *UserA* zaprasza *UseraB* do przyjaciół, na serwer wysyłane jest powiadomienie o zaproszeniu, który zpreparowane powiadomienie wysyła do danego użytkownika. Przyjmowany

jest obiekt klasy **InvitatationDTO** w metodzie **AddRelation**, znajdującej się pod adresem **/Notification/invitation**, który w kontrolerze jest walidowany i wysyłany do repozytorium powiadomień w celu jego przeprocesowania i zapisania wyniku na bazie danych.

Listing 26: Implementacja klasy InvitatationDTO

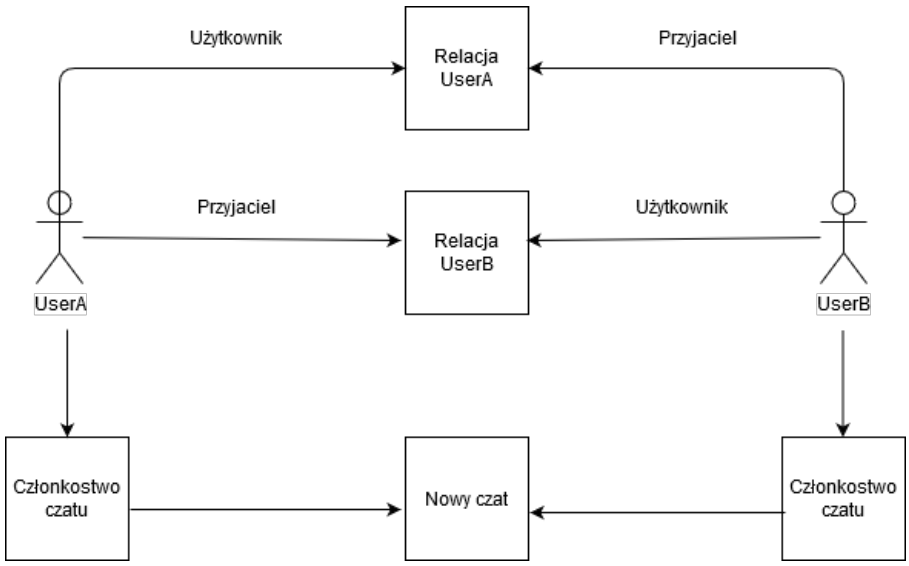
```
public class InvitatationDTO
{
        [Required]
        [RegularExpression(@"^[1-9]\d*$",
        ErrorMessage = "'0' Can not be used")]
        public string UserId { get; set; }

        [Required]
        [RegularExpression(@"^[1-9]\d*$",
         ErrorMessage = "'0' Can not be used")]
        public string OtherId { get; set; }

        public string ChatId { get; set; }
}
```

Ten użytkownik, moze przyjąć zaproszenie lub nie. Wysyła swoją decyzję za pomocą kolejnego powiadomienia do metody **ProcessNotification** znajdującej się pod adresem **Notification/proccess**. Przyjmuje ona obiekt klasy **NotificationDTO**.

Listing 27: Implementacja klasy NotificationDTO

```
public class NotificationDTO
{
        [Required]
        [RegularExpression(@"^[1-9]\d*$",
        ErrorMessage = "'0' Can not be used")]
        public string Id { get; set; }

        [Required]
        public bool Action { get; set; }

        [Required]
        public string UserId { get; set; }
}
```

Informacje zwrotna jako odpowiedz z rezultatem jest wysyłana do *UserA*. W dalszej części w razie przyjecia następuje stworzenie relacji między użytkownikami i odpowiedniego czatu oraz członkostwa do tego czatu.



Rysunek 17: Proces tworzenia nowej relacji

Relacja to tabela, która jest pomostem w relacji wiele do wielu. Jest zdefiniowany tam id danego użytkownika i id jego przyjaciela. W kostce Członkostwo czatu zawarta jest tablea ChatMemberships, która również jest pomostem w relacji wiele do wielu, czyli w relacji między tabelą Users, a Chats.
Te wpisy na bazie danych są tworzene w klasie pomocniczej **NotificationProcess**, która tworzy się i uruchamia w repozytorium powiadomień w metodzie **Process** przyjmującej id powiadomienia, akcje związane z akceptacja lub odrzuceniem powiadomiania oraz id uzytkownika, który przyjął powiadomienie. Informacja zwrotna jest natuchmiast wysyłana do wszystkich użytkowników, który interesuje dane powiadomienie za pomocą SignalR.

Listing 28: Implementacja procesowania powiadomienia

```
NotificationProcess process = new NotificationProcess
(_chatRepository, notification, isNotificationAccepted);

await process.Process().ConfigureAwait(true);

_context.Notifications.Remove(notification);

if(process.InformationNotification != null)
{
        await _context.Notifications
        .AddAsync(process.InformationNotification)
        .ConfigureAwait(true);
}

await _context.SaveChangesAsync().ConfigureAwait(true);

if (process.InformationNotification != null)
{
```

```
        await SendData(process.InformationNotification)
        .ConfigureAwait(true);
}
```

Wysyłanie powiadomień do użytkowników za pomocą SignalR zostało zaimplementowane w metodzie **SendeData** w repozytorium powiadomień, do którego trafia **Notification**.

Listing 29: Implementacja wysyłania powiadomień do użytkownika za pomocą SignalR

```
private async Task SendData(Notification notification)
{
        if(notification != null)
        {
                object[] param = { new NotificationChartModel
                {
                        Id = notification.Id,
                        SenderEmail = notification.Sender.Email,
                        ReceiverId = notification.ReceiverId,
                        Type = notification.Type,
                        Desc = notification.Desc,
                        ExtraData = notification.ExtraData,
                        Date = notification.Date
                } };

                var connectionId = _serivce.
                GetConnectionId(notification.ReceiverId);
                if(connectionId != null)
                {
                        _ = _hub.Clients.Client(connectionId)
                        .SendAsync("transfernotifications", param)
                        .ConfigureAwait(true);
                }
        }
}
```

Każdy użytkownik może usunąć danego w każdej chwili. Służy do tego metoda **Delete-Relation** osadzona pod linkiem **Notificaiton/deleteRelation**, która przyjmuje obiekt klasy **RelationDeletionDTO**.

Listing 30: Implementacja klasy RelationDeletionDTO

```
public class RelationDeletionDTO
{
        [Required]
        public string SenderId { get; set; }

        [Required]
        public string ChatId { get; set; }
```

```
        public string Reason { get; set; }

        public string MemberId { get; set; }
}
```

Po sprawdzeniu walidacji w kontrolerze, proces przechodzi do repozytorium notifikacji, a dokładniej do metody **DeleteMembership**, w której odbywa się usuwanie relacji oraz członkostwa czatu. Co ważne, sam czat nie jest usuwany. Stosowane powiadomienie zwrotne jest wysyłane do użytkownika jako odpowiedz.

Listing 31: Implementacja usuwania relacji

```
var relations = _context.Friends.Where(x => (x.UserId.Equals(idEjected) &&

if (relations == null)
{
        return false;
}

_context.Friends.RemoveRange(relations);

var chatMemberships = _context.ChatMemberships.Where(x => x.ChatId == conta
_context.ChatMemberships.RemoveRange(chatMemberships);
```

Cały proces powiadomień został dostosowany do grup w ten sposób, że dodawanie użytkownika odbywa się w tej samej metodzie co dodawanie przyjaciela. Tak samo w przypadku usuwania kogos z grupy, co może zrobić tylko właściciel danego czatu.

6.8 Implementacja zarządzania chatami

Czaty w naszej aplikacji to jedne z najważniejszych modeli. Posiadają informację na temat wszystkich członków oraz wiadomości. Większość fukcjonalności związanych z czatami jest obsługiwana w kontrolerze powiadomień. W kontrolerze czatu podstawową metodą jest stworzenie czatu grupowego za pomocą **CreateGroupChat**, znajdujacego się pod adresem **Chats/createGroupChat**, który przyjmuje obiekt klasy **GroupChatDTO**.

Listing 32: Implementacja klasy GroupChatDTO

```
public class GroupChatDTO
{
        public string OwnerId { get; set; }
        public string Name { get; set; }
}
```

W kontrolerze walidowana jest spójność i poprawność danych, a proces tworzenia czatu przeprowadzany jest w repozytorium czatu. Każdy czat może posiadać wielu użytkowników oraz wiele wiadomości. Została zachowana abstrakcja i czat przyjacielski nie rożni się budową niczym od czatu grupowego, przez sama implementacja przebiegła w prostszy sposób.

6.9 Obsługa SignalR

SignalR został do prawidłowego działania potrzebuje deklacji adresu w klasie **Startup** w metodzie **Configure**.

Listing 33: Konfiguracja SignalR

```
app.UseEndpoints(endpoints =>
{
        endpoints.MapControllers();
        endpoints.MapHub<MessagesHub>("/messagechart");
        endpoints.MapHub<NotificationHub>("/notificationchart");
        endpoints.MapHub<GateHub>("/connection");
});
```

W apliakcji zostały użyte trzy Hub's.
Pierwszy z nich to **GateHub**, który posiada metodę **GetConnectionId**. Służy on do autoryzacji połączenia. Użytkownik musi wysłać swoje id oraz token, w którym znajduję się email. Jeśli dany id jest zgodny z tokenem następuje pomyślne zalogowanie. Taka aktywność zapisywana jest w tabeli **UsersTrace**. Ta metoda jest wywoływana zdalnie z poziomu klienta.

Listing 34: Implementacja metody GetConnectionId z klasy GateHub

```
public async Task<bool> GetConnectionId(int userId)
{
        var user = await _context.Users
        .FirstOrDefaultAsync(x => x.Id == userId)
        .ConfigureAwait(true);

        var requesterMail = Context.User.Identity.Name;
        var ipAddress = Context.Features.Get<IHttpConnectionFeature>()
        ?.ConnectionId;
        bool isSuccessLogged;

        if (requesterMail.Equals
        (requesterMail, System.StringComparison.CurrentCulture))
        {
                isSuccessLogged = true;
        }
        else
        {
                isSuccessLogged = false;
        }

        await _context.UsersTrace.AddAsync(new Models.UserTrace
        {
                Who = requesterMail,
                UserIdProvided = userId,
                When = DateTime.Now,
```

```
                Result = isSuccessLogged,
                IpAddress = ipAddress
        });

        await _context.SaveChangesAsync().ConfigureAwait(true);

        return isSuccessLogged;
}
```

Kolejnymi Hub tego samego typu są **MessageHub** oraz **NotificationHub**. Dziedziczą one po **BaseHub**. Różnicą jest definiowane serwisu odpowiedzialnego za połączenia.

Listing 35: Implementacja klasy MessageHub

```
public class MessagesHub : BaseHub
{
        protected override ServiceTypes Type =>
        ServiceTypes.MessageConnectionService;

        public MessagesHub(Func<ServiceTypes,
         IConnectionService> servicesResolver) : base(servicesResolver)
        {

        }
}
```

Klasy **MessageConnectionService** oraz **NotificationConnectionService** wspólnie dziedziczą po **ConnectionService**. Nie mają żadnych różnic, a zostały zaimplementowane w sposób osobny w celu wskrzynięcia osobnej klasy, z osobymi danymi do odpodniego Hub'a. Tak w tym przypadku MessagesHub posiada MessageConnectionService. W klasie Startup w metodzie ConfigureServices takie działanie na wstrzykiwaniu serwisów tego samego typu odbywa się za pomocą Delegatu Func.

Listing 36: Implementacja DI na dwóch klasach tego samego typu

```
services.AddSingleton<Func<ServiceTypes,
IConnectionService>>(provider => serviceType =>
{
        switch(serviceType)
        {
                case ServiceTypes.MessageConnectionService:
                        return provider.
                        GetService<MessageConnectionService>();
                case ServiceTypes.NotificationConnectionService:
                        return provider
                        .GetService<NotificationConnectionService>();
                default:
                        return null;
        }
```

```
});
```

Klasa ConnectionService przypisuje connectionId do id danego użytkownika.

Listing 37: Dodanie wpisu do słownika na temat id połączenia i użytkownika

```
public void AddConnection(string connectionId, int userId)
{
        _connections.Add(new ConnectionContainer
        {
                ConnectionId = connectionId,
                UserId = userId
        });
}
```

Wywoływane jest to w klasie BaseHub podczas nawiązania połączenia.

Listing 38: Dodanie wpisu do słownika na temat id połączenia i użytkownika

```
public override Task OnConnectedAsync()
{
        if(int.TryParse(Context.UserIdentifier, out int userId))
        {
                _connectionService
                .AddConnection(Context.ConnectionId, userId);

                return base.OnConnectedAsync();
        }

        throw new Microsoft.AspNetCore.SignalR.HubException("Cannot parse u
}
```

Taki mechanizm pozwala wysłać wiadomość do konkretnych użytkowników. Wystarczy podać id użytkownika, do którego chcemy wysłać wiadomość, a serwis za pomocą metody **GetConnectionId** zwróci id połączenia.

Listing 39: Przykład wysyłania wiadomości do danego użytkownika.

```
string connectionId = _service.GetConnectionId(userId);
if(connectionId != null && connectionId.Trim().Length != 0)
{
        _ = _hub.Clients.Client(connectionId)
        .SendAsync("transferchartdata", param)
        .ConfigureAwait(true);
}
```

7 Podsumowanie

7.1 Wnioski

7.2 Do zrealizowania przy dalszym rozwoju