

Dokumentacja projektowa  
Podstawy Teleinformatyki  
*Rozmawiator*

Sebastian Daniel Alex      Igor Patryk Guziołek  
Norbert Rafał Sala

17 czerwca 2016

# Spis treści

<b>1</b>	<b>Wstęp</b>	<b>3</b>
1.1	Opis projektu . . . . .	3
<b>2</b>	<b>Założenia realizacyjne</b>	<b>3</b>
2.1	Używane technologie . . . . .	3
<b>3</b>	<b>Podział prac</b>	<b>3</b>
3.1	Sebastian Daniel Alex . . . . .	3
3.2	Igor Patryk Guziołek . . . . .	4
3.3	Norbert Rafał Sala . . . . .	4
<b>4</b>	<b>Opis implementacji</b>	<b>4</b>
4.1	Serwer . . . . .	4
4.1.1	Call . . . . .	4
4.1.2	CallRequest . . . . .	4
4.1.3	Client . . . . .	5
4.1.4	Conversation . . . . .	5
4.1.5	Listener . . . . .	6
4.2	Baza danych . . . . .	9
4.2.1	Realizacja . . . . .	10
4.2.2	Tabela Server . . . . .	11
4.2.3	Tabela User . . . . .	11
4.2.4	Tabela Conversation . . . . .	12
4.2.5	Tabela ConversationParticipant . . . . .	12
4.2.6	Tabela TextMessage . . . . .	12
4.2.7	Tabela Call . . . . .	12
4.2.8	Tabela CallParticipant . . . . .	13
4.2.9	Tabela CallRequest . . . . .	13
4.2.10	Tabele ASP.NET Identity . . . . .	13
4.2.11	ViewModele . . . . .	13
4.3	Serwer REST . . . . .	14
4.3.1	Realizacja . . . . .	14
4.3.2	Autoryzacja w usłudze . . . . .	14
4.3.3	Filtrowanie zapytań . . . . .	15
4.3.4	Metody kontrolera Server . . . . .	16
4.3.5	Metody kontrolera User . . . . .	16
4.3.6	Metody kontrolera Conversation . . . . .	18
4.3.7	Metody kontrolera Message . . . . .	18
4.3.8	Metody kontrolera CallRequest . . . . .	19

4.3.9	Metody kontrolera Account . . . . .	19
4.3.10	Inne metody . . . . .	20
4.4	Program kliencki . . . . .	21
4.5	Klient UDP . . . . .	21
4.5.1	Klasa Call.cs . . . . .	21
4.5.2	Klasa CallRequest.cs . . . . .	22
4.5.3	Klasa Client.cs . . . . .	23
4.5.4	Klasa Conversation.cs . . . . .	24
4.6	Moduł audio . . . . .	25
4.6.1	Klasa Gsm610Codec.cs . . . . .	26
4.6.2	Klasa AcmCodec.cs . . . . .	26
4.6.3	Klasa Player.cs . . . . .	26
4.6.4	Klasa Recorder.cs . . . . .	27
4.6.5	Biblioteki wykorzystywane w projekcie: . . . . .	28
<b>5</b>	<b>Użytkowanie i testowanie systemu</b>	<b>28</b>
5.1	Serwer . . . . .	28
5.2	Klient . . . . .	30
<b>6</b>	<b>Kod źródłowy programu</b>	<b>34</b>
6.1	Baza danych . . . . .	34
6.1.1	Inicjalizacja kontekstu . . . . .	34
6.2	Serwer REST . . . . .	35
6.2.1	Filtrowanie zapytań . . . . .	35

# 1 Wstęp

## 1.1 Opis projektu

Usługa pozwalająca na rozmowy tekstowe i połączenia głosowe. Projekt składa się z aplikacji klienckiej, serwera, serwera REST i bazy danych.

## 2 Założenia realizacyjne

- Celem projektu jest utworzenie systemu umożliwiającego komunikację zarówno tekstową jak i głosową między użytkownikami podłączonymi do serwera. Użytkownicy mogą tworzyć między sobą konwersacje, w ramach których rozsyłane są wiadomości.
- Temat ten został wybrany w celu poszerzenia wiedzy o technologiach VoIP i przygotowaniu podbudowy pod system rozmów w pracy inżynierskiej.

### 2.1 Używane technologie

- C#
- REST API
- Entity Framework
- Windows Presentation Foundation
- NAudio

## 3 Podział prac

### 3.1 Sebastian Daniel Alex

- Baza danych
- REST API

### 3.2 Igor Patryk Guziołek

- Program kliencki
- Klient UDP
- Komunikacja audio

### 3.3 Norbert Rafał Sala

- Serwer UDP

## 4 Opis implementacji

### 4.1 Serwer

#### 4.1.1 Call

- HandleMessage - metoda obsługująca przychodzące typy wiadomości (Nowy użytkownik, pożegnanie, użytkownik rozłączył się, wiadomość audio)
  - Parametry:
    - \* message - wiadomość do obsłużenia
- HandleBye - metoda obsługująca wiadomość zawierającą komunikat o zakończeniu połączenia
  - Parametry:
    - \* message - wiadomość do obsłużenia
- TryClose - metoda do zamykania serwera. Wykona się tylko jeżeli podłączony jest tylko 1 użytkownik (wtedy wysyła temu użytkownikowi informację o zamknięciu serwera i odłącza go) lub gdy nie ma żadnego podłączonego użytkownika

#### 4.1.2 CallRequest

Klasa obsługująca wysyłanie zapytań między klientami.

- SendRequest - metoda ta wysyła (jako klient) zapytanie do innego klienta, ustawia status zapytania na wysłane
- ResolveRequest - metoda ustawia status zapytania na zrealizowane

- Zwraca: pierwszy bajt otrzymanej wiadomości
- Parametry:
  - \* message - wiadomość do obsłużenia

#### 4.1.3 Client

Reprezentacja klienta.

- OnTimeout - callback wywoływany w momencie gdy przez dany czas użytkownik nie wysłał żadnego pakietu KeepAlive
- KeepAlive - metoda resetuje (zatrzymuje i uruchamia) timer zliczający czas od ostatnio otrzymanej wiadomości od danego użytkownika

#### 4.1.4 Conversation

Klasa odpowiedzialna za tworzenie i obsługiwanie konwersacji po stronie serwera.

- HandleMessage - metoda obsługująca przychodzące typy wiadomości (Nowy użytkownik, pożegnanie, wiadomość tekstowa, utworzenie nowego połączenia, odpowiedź na nowe połączenie); każda otrzymana wiadomość jest broadcastowana
  - Parametry:
    - \* message - wiadomość do obsłużenia
- CreateCall - metoda tworzy nowe połączenia z klientami i wysyła pozostałym klientom prośby o dołączenie do danego połączenia
  - Parametry:
    - \* message - wiadomość zawierająca wszystkie wymagane dane nowego klienta
- SaveMessage - metoda zapisująca wiadomość danej konwersacji do bazy danych
  - Parametry:
    - \* message - wiadomość do obsłużenia

#### 4.1.5 Listener

Klasa odpowiadająca za logikę serwera.

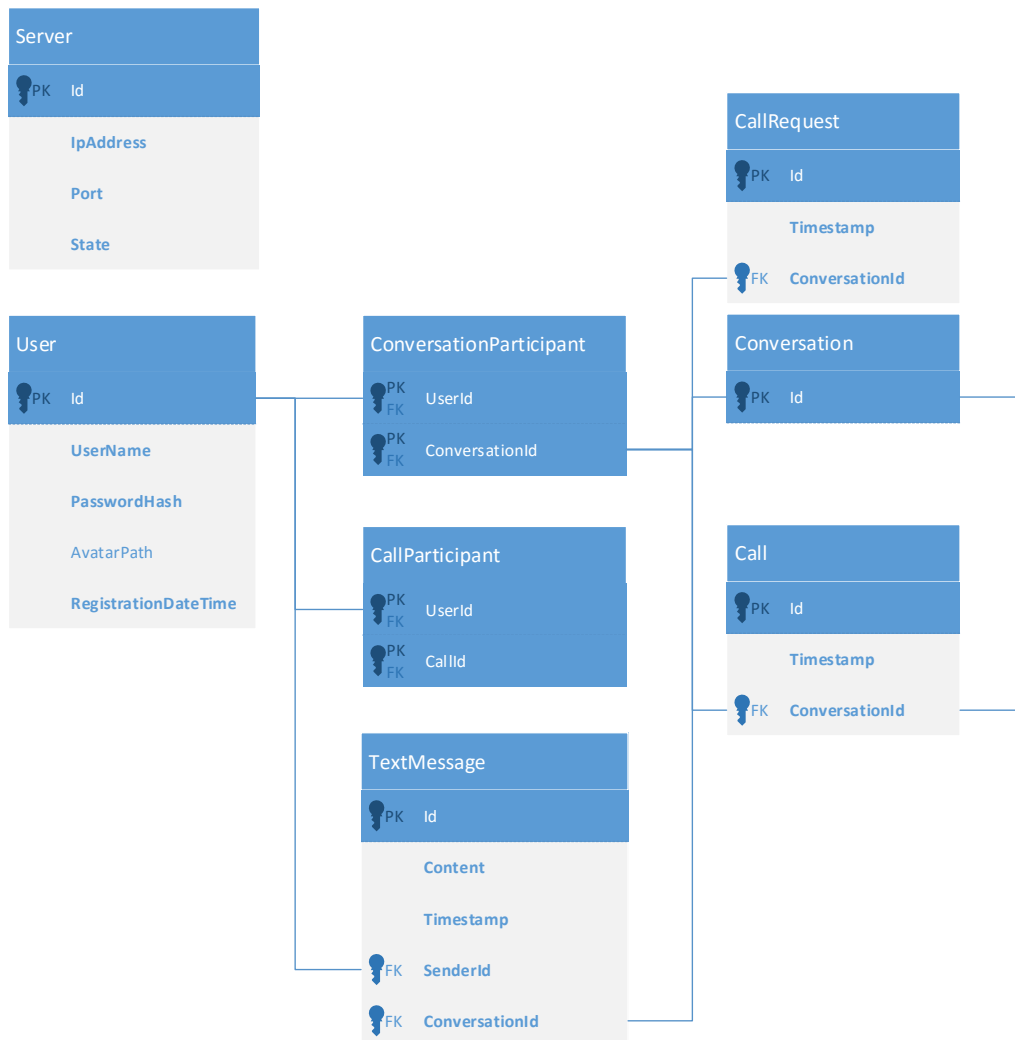
- Start - metoda odpowiedzialna za uruchomienie serwera; najpierw łączy się z bazą danych i wyszukuje tam danych serwera wykorzystując GUID (w przypadku nie odnalezienia danego GUID, tworzy nowy serwer i zapisuje go do bazy), a następnie rozpoczyna nasłuchiwanie
- Stop - metoda odpowiedzialna za kończenie pracy serwera; w bazie danych ustawia status serwera na offline, potem rozsyła wszystkim podłączonym klientom wiadomość o rozwiązywaniu połączenia i połączenie to rozwiązuje, a na koniec zamyka nasłuchiwanie
- ListenLoop - metoda typu Task służąca do nasłuchiwania nowych połączeń, w przypadku wykrycia nowego połączenia rozpoczyna całą logikę dotyczącą obsłużenia dodania nowego klienta
- HandleMessage - metoda obsługująca przychodzącą wiadomość (Serwer, konwersacja, połączenia głosowe); jej zadaniem jest wykryć typ wiadomości i na podstawie tego przekazać ją do odpowiedniej innej metody
  - Parametry:
    - \* endpoint - punkt końcowy klienta
    - \* message - wiadomość do obsłużenia
- HandleServerMessage - metoda obsługująca wiadomości serwerowe (Przywitanie, pożegnanie, pakiet wysyłany w celu utrzymania połączenia, tworzenie konwersacji)
  - Parametry:
    - \* endpoint - punkt końcowy klienta
    - \* message - wiadomość do obsłużenia
- HandleConversationMessage - metoda obsługująca wiadomości konwersacji; wyszukuje czy konwersacja istnieje, w przypadku jej braku tworzy nową konwersację, a następnie przekazuje wiadomość dalej do obsłużenia
  - Parametry:
    - \* endpoint - punkt końcowy klienta
    - \* message - wiadomość do obsłużenia

- HandleCallMessage - metoda obsługująca wiadomości dotyczące połączeń głosowych; wyszukuje czy połączenie głosowe istnieje, a następnie przekazuje wiadomość dalej do obsłużenia
  - Parametry:
    - \* endpoint - punkt końcowy klienta
    - \* message - wiadomość do obsłużenia
- HandleHello - metoda obsługująca wiadomość powitalną, rozpoczynającą tworzenie połączenia między serwerem a klientem; najpierw wyszukuje czy dany klient już istnieje w bazie danych (i przerywa działanie w przypadku znalezienia klienta), a następnie ustanawia połączenie i wysyła klientowi wiadomość potwierdzającą
  - Parametry:
    - \* endpoint - punkt końcowy klienta
    - \* message - wiadomość do obsłużenia
- HandleBye - metoda obsługująca wiadomość pożegnalną, kończąca połączenie między serwerem a klientem
  - Parametry:
    - \* sender - klient przysyłający wiadomość
    - \* message - wiadomość do obsłużenia
- HandleCreateConversation - metoda obsługująca tworzenie nowych konwersacji; wyszukuje konwersacji w bazie danych, dołącza doń podanego klienta i wysyła mu wiadomość potwierdzającą
  - Parametry:
    - \* sender - klient przysyłający wiadomość
    - \* message - wiadomość do obsłużenia
- HandleKeepAlive - metoda odpowiedzialna za resetowanie czasu przedawnienia połączenia klienta (timeout)
  - Parametry:
    - \* sender - klient przysyłający wiadomość
    - \* message - wiadomość do obsłużenia
- AddClient - metoda dodająca użytkownika do listy klientów



- Zwraca: utworzonego klienta
- Parametry:
  - \* endpoint - punkt końcowy klienta
  - \* user - użytkownik jakiego należy dodać jako klienta
- ClientOnTimeout - metoda usuwająca klienta z listy; uruchamia się jeżeli licznik przedawnienia połączenia (timeout) przekroczy ustaloną wartość
  - Parametry:
    - \* client - klient, którego połączenie jest przedawnione

## 4.2 Baza danych



Rysunek 1: Diagram bazy danych

Serwer i usługa REST wymagają postawionej bazy danych. Pozwala to na przechowywanie użytkowników zarejestrowanych w usłudze, i całkowitej historii wiadomości i połączeń.

Baza danych zawiera w sumie 8 tabel, z czego dwie są czysto powiązaniowe. Dzięki zastosowaniu biblioteki ORM Entity Framework, tabele powiązaniowe są niewidoczne w kodzie.

W tabeli znajdują się dodatkowe tabele wymagane przez system ASP.NET Identity, które nie wpływają znacząco na założenia projektowe.

#### 4.2.1 Realizacja

Baza danych jest pisana zgodnie z założeniami CodeFirst w Entity Framework. Jako klucz indeksowy używany jest typ GUID. Jego zaletą jest możliwość wygenerowania go przez kod. Powtórzenie się takiego klucza jest praktycznie niemożliwe, gdyż ilość dostępnych kluczy do wygenerowania to  $2^{128}$ .

Kontekst bazy danych jest rozszerzony o bibliotekę ASP.NET Identity, która zapewnia metody uwierzytelnienia w usłudze HTTP. Domyślnie Identity korzysta z klucza typu *string*, przechowującego GUID. Dla zapewnienia konsystencji implementacji, typ został zmieniony na *System.Guid*, który zachowuje się praktycznie tak samo. Do zmiany typu klucza Identity wymagał użycia odpowiednich klas, które także korzystają z tego typu:

- GuidUserClaim : IdentityUserClaim<Guid>
- GuidUserLogin : IdentityUserLogin<Guid>
- GuidUserRole : IdentityUserRole<Guid>
- GuidRole : IdentityRole<Guid, GuidUserRole>
- User : IdentityUser<Guid, GuidUserLogin, GuidUserRole, GuidUserClaim>

Każda klasa zawiera także metody nawigacyjne, oferowane przez EF. Dzięki temu zamiast poruszać się po kluczach GUID, programista ma do dyspozycji relacje silnie mapowane na obiekty w kodzie:

```
[ForeignKey("ConversationId")]
public virtual Conversation Conversation { get; set; }

public virtual ICollection<User> Participants { get; set; }
```

Te metody nawigacyjne są mapowane przy utworzeniu kontekstu, i są obowiązujące tylko dla tego kontekstu. Próba użycia ich gdziekolwiek indziej zaskutkuje wyjątkami.

Entity Framework zapewnia także realizację relacji wielu-do-wielu, co w SQLu możliwe jest tylko przez użycie tabeli pośredniczącej. EF mapuje to "pod maską", i tabela pośrednicząca jest niewidoczna dla programisty. Należy ją jednak jawnie zmapować:

```

/* Conversation - User mapping */
modelBuilder
    .Entity<User>()
    .HasMany(u => u.Conversations)
    .WithMany(c => c.Participants)
    .Map(m =>
    {
        m.MapLeftKey("UserId");
        m.MapRightKey("ConversationId");
        m.ToTable("ConversationParticipants");
    });

```

Domyślnie bezpośredni dostęp do kontekstu posiada serwer i usługa REST.

#### 4.2.2 Tabela Server

Tabela przechowująca dane o serwerach dostępnych w usłudze. Dzięki temu klient, chcąc połączyć się do serwera, nie musi posiadać u siebie listy dostępnych serwerów, gdyż te automatycznie aktualizują się w tej tabeli.

- Id - unikalny identyfikator GUID.
- IpAddress - zewnętrzny adres IP serwera.
- Port - port, na którym serwer nasłuchuje.
- State - status serwera - 0 = offline, 1 = online.

#### 4.2.3 Tabela User

Podstawowy użytkownik bazy danych. Aby korzystać z usługi, klient musi być zarejestrowany. Rejestracji można dokonać w programie klienckim.

- Id - unikalny identyfikator GUID.
- UserName - unikalna nazwa użytkownika w usłudze.
- AvatarPath - opcjonalna ścieżka do pliku z awatarem po stronie serwera REST.
- RegistrationDateTime - czas rejestracji.

Pola dodawane przez ASP.NET Identity:

- PasswordHash - skrót hasła użytkownika.

- Email - adres e-mail użytkownika.

ASP.NET Identity dodaje więcej pól, ale nie są one jawnie używane w programie.

#### 4.2.4 Tabela Conversation

Konwersacje tekstowe użytkowników. Agreguje ona połączenia głosowe, i użytkowników za pomocą tabeli *ConversationParticipant*.

- Id - unikalny identyfikator GUID.

#### 4.2.5 Tabela ConversationParticipant

Przynależność użytkownika do danej konwersacji. Obsługiwana przez ORM.

- UserId - identyfikator GUID użytkownika.
- ConversationId - identyfikator GUID konwersacji.

#### 4.2.6 Tabela TextMessage

Prosta wiadomość tekstowa. Należy do danej konwersacji, i posiada zapis, który użytkownik ją wysłał. Widoczna dla wszystkich użytkowników konwersacji.

- Id - unikalny identyfikator GUID.
- Content - zawartość wiadomości.
- Timestamp - czas stworzenia (wysłania) wiadomości.
- SenderId - identyfikator GUID wysyłającego.
- ConversationId - identyfikator GUID konwersacji.

#### 4.2.7 Tabela Call

Połączenia głosowe użytkowników. Agreguje ona użytkowników uczestniczących w rozmowie za pomocą tabeli *CallParticipant*. Ta agregacja jest przechowywana tylko na czas rozmowy, w celu umożliwienia klientowi pobrania aktualnych uczestników połączenia.

- Id - unikalny identyfikator GUID.
- Timestamp - czas rozpoczęcia połączenia.
- ConversationId - identyfikator GUID konwersacji.

#### 4.2.8 Tabela CallParticipant

Przynależność użytkownika do połączenia. Obsługiwana przez ORM. Usuwana po odłączeniu się użytkownika od rozmowy.

- UserId - identyfikator GUID użytkownika.
- CallId - identyfikator GUID połączenia.

#### 4.2.9 Tabela CallRequest

Historia żądań próśb dołączenia do rozmowy.

- Id - unikalny identyfikator GUID.
- Timestamp - czas utworzenia prośby.
- ConversationId - identyfikator GUID konwersacji.

#### 4.2.10 Tabele ASP.NET Identity

Biblioteka Identity dodaje dodatkowe tabele do bazy danych, które nie są używane jawnie w programie:

- Roles
- UserClaims
- UserLogins
- UserRoles

#### 4.2.11 ViewModele

Modele zdefiniowane do użytku w kontekście CodeFirst nie nadają się do wyświetlania ich użytkownikom REST Api, ze względu na dużą ilość niepotrzebnych pól, takich jak pola nawigacyjne. Definiują także to, co może użytkownik otrzymać. Dla przykładu, przy wyświetlaniu danych o użytkowniku nie powinny być wysyłane jego dane niejawne, np. email lub skrót hasła.

Każda tabela w bazie danych zawiera swój odpowiedni *viewmodel*. Są nazwane z konwencją *<NazwaTabeli> ViewModel*.

## 4.3 Serwer REST

Serwer REST jest pośrednikiem między bazą danych a klientami usługi. Jest dostępny pod stałym adresem/domeną, wkompilem w kod klienta. Dzięki zastosowaniu ASP.NET Identity możliwa jest rejestracja i autoryzacja użytkowników w usłudze. REST udostępnia głównie metody pozwalające na odczytywanie obiektów z bazy danych, takich jak nazwy użytkowników, ich awatary, wiadomości tekstowe itd.

Domyślnie serwer REST nie pozwala na modyfikację bazy danych poza danymi użytkownika. Obiekty takie jak wiadomości czy konwersacje są dodawane przez serwer za pomocą ORM'a. Pozwala to na hermetyzację dostępu do bazy danych, nawet w przypadku próby odwrotnej inżynierii.

Do komunikacji z REST napisana jest odpowiednia biblioteka, która jest opisana w oddzielnej sekcji.

Na obecny stan projektu serwer REST nie obsługuje protokołów TLS.

### 4.3.1 Realizacja

Serwer REST jest napisany w technologii ASP.NET WebAPI. Interakcja z danymi w bazie jest realizowana poprzez kontrolery, które wystawiają odpowiednie funkcje. W większości przypadków jest to pobieranie pojedynczego obiektu lub listy, uprzednio przefiltrowanej.

Ścieżka do metod RESTowych jest pisana z przyjętą przez nas konwencją, i wygląda ona tak:

```
http://<adres>/api/<nazwa kontrolera>/<metoda>
```

Argumenty do metod mogą być podawane przez zapis ze znakiem zapytania, lub jako kolejne parametry oddzielane ukośnikiem:

```
http://<adres>/api/<nazwa kontrolera>/<metoda>?param=paramValue  
http://<adres>/api/<nazwa kontrolera>/<metoda>/paramValue
```

W przypadku metod POST, argumenty można też podawać w ciele żądania, jako obiekt w notacji JSON. Do serializacji obiektów wykorzystujemy bibliotekę *Json.NET*.

### 4.3.2 Autoryzacja w usłudze

Większość metod jest zabezpieczonych przed niepowołanym dostępem, i zwraca dane zależne od zalogowanego użytkownika. Do autoryzacji używany jest token OWIN, który jest zwracany przez metodę *token*. Przykładowa zawartość takiego tokena to:

```
{
  "access_token": "o4IuKkxELMzBHR55DxBCGVdHMAUafSP6wPq{...}",
  "token_type": "bearer",
  "expires_in": 86399,
  "userName": "user1823",
  ".issued": "Mon, 13 Jun 2016 17:17:04 GMT",
  ".expires": "Tue, 14 Jun 2016 17:17:04 GMT"
}
```

Zawartość pola "access\_token" w zapytaniu do serwera REST należy wpisać w nagłówek HTTP "Authorization", w formacie:

```
Bearer o4IuKkxELMzBHR55DxBCGVdHMAUafSP6wPq{...}
```

Tokeny są unikalne dla każdego logowania, i nie są przechowywane w bazie danych.

#### 4.3.3 Filtrowanie zapytań

Aby uniknąć nieskończonej ilości metod filtrujących zapytania, udostępniona została możliwość filtrowania zapytań poprzez obiekt w notacji JSON. Przy dowolnej metodzie pobierającej dane, należy przesłać obiekt w notacji JSON z filtrami w postaci:

```
{
  "nazwaPola" : "oczekiwanaWartosc"
}
```

Należy wspomnieć, że to nie jest wyszukiwanie - zostaną zwrócone tylko te obiekty, których pola o odpowiednich nazwach mają dokładnie takie same wartości, jak przedstawione w JSONie. Dla przykładu metoda zwracająca wiadomości tekstowe zwróci nam tylko wiadomości tekstowe od nadawcy o GUIDzie *09b5d376-fbd6-4425-9239-92250fe43f8d*:

```
{
  "SenderId" : "09b5d376-fbd6-4425-9239-92250fe43f8d"
}
```

W filtrowaniu wielkość znaków nie ma znaczenia.

Należy także wspomnieć, że filtrowanie zapytań nie odbywa się na serwerze REST. Dzięki zastosowaniu wyrażeń drzewiastych, można dopisać parametry dynamicznie do budowanego przez Entity Framework drzewa zapytania:



```

var condition = Expression.Lambda<Func<TModel, bool>>(
    Expression.Equal(
        Expression.Property(param, filter.Key),
        Expression.Constant(filterValue, filterValue.GetType())
    ), param);

return query.Where(condition);

```

#### 4.3.4 Metody kontrolera Server

- GetServer - zwraca serwer o podanym identyfikatorze
  - Metoda: GET
  - Adres: /api/Servers/{id}
  - Zwraca: ServerViewModel
  - Parametry:
    - \* id - identyfikator GUID serwera
- GetOnlineServer - zwraca pierwszy serwer, który jest online
  - Metoda: GET
  - Adres: /api/Servers/Online
  - Zwraca: ServerViewModel
- GetServers - zwraca listę serwerów, opcjonalnie przefiltrowaną
  - Metoda: POST
  - Adres: /api/Servers/List
  - Zwraca: lista ServerViewModel
  - Parametry:
    - \* filters - filtry w notacji JSON

#### 4.3.5 Metody kontrolera User

- GetUser - zwraca dane użytkownika o podanym identyfikatorze, lub zalogowanego użytkownika
  - Metoda: GET
  - Adres: /api/Users/id?

- Zwraca: UserModel
  - Parametry:
    - \* id - identyfikator GUID użytkownika - opcjonalny
- GetUser - zwraca dane użytkownika o podanej nazwie użytkownika, lub zalogowanego użytkownika
  - Metoda: GET
  - Adres: /api/Users/Info/username?
  - Zwraca: UserModel
  - Parametry:
    - \* username - nazwa użytkownika - opcjonalny
- GetUsers - zwraca listę użytkowników, opcjonalnie przefiltrowaną
  - Metoda: POST
  - Adres: /api/Users/List
  - Zwraca: lista UserModel
  - Parametry:
    - \* filters - filtry w notacji JSON
- SearchUsers - zwraca listę użytkowników, zawierających podaną nazwę użytkownika
  - Metoda: GET
  - Adres: /api/Users/Search/query
  - Zwraca: lista UserModel
  - Parametry:
    - \* query - nazwa użytkownika do wyszukania
- GetAvatar - zwraca awatar podanego użytkownika w postaci binarnej
  - Metoda: GET
  - Adres: /api/Users/Avatar/username
  - Zwraca: HttpResponseMessage, z zawartością z MIME image/png
  - Parametry:
    - \* username - nazwa użytkownika

- SetAvatar - ustawia awatar zalogowanego użytkownika na przesłany w formie binarnej
  - Metoda: POST
  - Adres: /api/Users/Avatar
  - Zwraca: kod HTTP
  - Parametry:
    - \* obrazek w formie binarnej

#### 4.3.6 Metody kontrolera Conversation

- GetConversation - zwraca konwersację o podanym identyfikatorze
  - Metoda: GET
  - Adres: /api/Conversations/id
  - Zwraca: ConversationViewModel
  - Parametry:
    - \* id - identyfikator GUID konwersacji
- GetConversations - zwraca listę użytkowników, opcjonalnie przefiltrowaną
  - Metoda: POST
  - Adres: /api/Conversations/List
  - Zwraca: lista ConversationViewModel
  - Parametry:
    - \* filters - filtry w notacji JSON

#### 4.3.7 Metody kontrolera Message

- GetMessage - zwraca wiadomość o podanym identyfikatorze
  - Metoda: GET
  - Adres: /api/Messages/id
  - Zwraca: MessageViewModel
  - Parametry:
    - \* id - identyfikator GUID wiadomości

- GetMessages - zwraca listę wiadomości, opcjonalnie przefiltrowaną
  - Metoda: POST
  - Adres: /api/Messages/List?page=0&count=100
  - Zwraca: lista MessageViewModel
  - Parametry:
    - \* filters - filtry w notacji JSON
    - \* page - strona wyników, domyślnie 0
    - \* count - ilość wyników na stronę, domyślnie 100
    - \* priorTo - zwraca wiadomości starsze od wiadomości, której identyfikator podano w tym parametrze - opcjonalny

#### 4.3.8 Metody kontrolera CallRequest

- GetCallRequest - zwraca prośbę połączenia o podanym identyfikatorze
  - Metoda: GET
  - Adres: /api/CallRequests/id
  - Zwraca: CallRequestViewModel
  - Parametry:
    - \* id - identyfikator GUID prośby
- GetCallRequests - zwraca listę prośb połączenia, opcjonalnie przefiltrowaną
  - Metoda: POST
  - Adres: /api/CallRequests/List
  - Zwraca: lista CallRequestViewModel
  - Parametry:
    - \* filters - filtry w notacji JSON

#### 4.3.9 Metody kontrolera Account

Pominięte zostały metody jawnie nieużywane.

- ChangePassword - zmienia hasło aktualnie zalogowanego użytkownika
  - Metoda: POST
  - Adres: /api/Account/ChangePassword

- Zwraca: kod HTTP
- Parametry (JSON):
  - \* OldPassword - obecne hasło
  - \* NewPassword - nowe hasło
  - \* ConfirmPassword - powtórzone nowe hasło
- Register - rejestruje nowego użytkownika w bazie danych
  - Metoda: POST
  - Adres: /api/Account/Register
  - Zwraca: kod HTTP
  - Parametry (JSON):
    - \* UserName - nazwa użytkownika
    - \* Email - adres email
    - \* Password - hasło
    - \* ConfirmPassword - powtórzone hasło

#### 4.3.10 Inne metody

- token - zwraca token OWIN do autoryzacji w usłudze
  - Metoda: POST
  - Adres: /token
  - Zwraca: token, czas ważności - JSON
  - Parametry:
    - \* username - nazwa użytkownika
    - \* password - hasło
    - \* grant\_type - typ autoryzacji, powinien być "password"
  - Uwagi:
    - \* Content-Type musi być ustawiony na "application/x-www-form-urlencoded"
    - \* parametry muszą być podane jako klucze w x-www-form-urlencoded

## 4.4 Program kliencki

### 4.5 Klient UDP

Moduł projektu, którego głównym zadaniem jest komunikacja z serwerem UDP, w celu informowania o rozmowach oraz konwersacjach z poziomu aplikacji klienckiej.

#### 4.5.1 Klasa Call.cs

Klasa odpowiadająca za modelowanie oraz obsługę rozmów głosowych. Poniżej znajduje się opis używanych przez tą klasę metod oraz pól:

- Id - zmienna typu GUID (Globally Unique Identifier) przechowująca id rozmowy głosowej
- Conversation - obiekt przechowujący informacje o konwersacji bezpośrednio związanej z rozmową audio
- Participants - kolekcja przechowująca dane o uczestnikach rozmowy
- NewParticipant - event mający za zadanie poinformować o dołączeniu nowego uczestnika rozmowy
- ParticipantDeclined - event mający za zadanie poinformować o odmowie dołączenia do rozmowy użytkownika zaproszonego do udziału w rozmowie
- ParticipantLeft - event mający za zadanie zasygnalizować sytuację gdy uczestnik opuści rozmowę audio.
- NewAudio - event sygnalizujący podłączenie do rozmowy nowego strumienia audio
- CallEnded - event sygnalizujący zakończenie rozmowy głosowej
- IsMember - metoda określająca czy użytkownik o GUIDzie podanym w parametrze metody jest uczestnikiem rozmowy
- AddUser - metoda pozwalająca na dodanie uczestnika o podanym GUIDzie do rozmowy
- Disconnect - metoda wysyłająca do serwera UDP informację o zakończeniu rozmowy

- `HandleCallMessage` - metoda pełniąca funkcję pomocniczą do obsługi wiadomości serwera UDP
- `Send` - metoda, której zadaniem jest wysyłanie wiadomości sterujących rozmową do serwera UDP
- `HandleNewUser` - metoda pełniąca funkcję pomocniczą do obsługi wiadomości serwera UDP na wypadek dołączenia nowego uczestnika do rozmowy audio
- `HandleUserDeclined` - metoda pełniąca funkcję pomocniczą do obsługi wiadomości serwera UDP na wypadek odrzucenia zaproszenia przez użytkownika zaproszonego do rozmowy
- `HandleBye` - metoda pełniąca funkcję pomocniczą do obsługi wiadomości serwera UDP na wypadek zakończenia rozmowy audio
- `HandleUserLeft` - metoda pełniąca funkcję pomocniczą do obsługi wiadomości serwera UDP na wypadek gdy jakiś użytkownik odejdzie z rozmowy audio
- `HandleAudio` - metoda pełniąca funkcję pomocniczą do obsługi wiadomości serwera UDP na wypadek nowego, przychodzącego strumienia audio

#### 4.5.2 Klasa `CallRequest.cs`

Klasa będąca modelem zapytań związanych z rozmowami audio. Poniżej znajduje się opis używanych przez tą klasę metod oraz pól:

- `Conversation` - obiekt przechowujący informacje o konwersacji bezpośrednio związanej z rozmową audio
- `CallId` - zmienna typu GUID (Globally Unique Identifier) przechowująca id rozmowy głosowej
- `Response` - obiekt typu `CallResponseType`, może przyjmować wartość `null`, będący reprezentacją kodu odpowiedzi dotyczącej zapytania o operację dotyczącą rozmowy audio
- `Accept` - metoda zarządzająca zatwierdzaniem rozmów audio
- `Decline` - metoda zarządzająca odrzucaniem rozmów audio
- `Ignore` - metoda zarządzająca ignorowaniem odbierania rozmów audio

#### 4.5.3 Klasa Client.cs

Klasa będąca modelem stanu klienta oraz aplikacji klienta. Poniżej znajduje się opis używanych przez tą klasę metod oraz pól:

- Id - zmienna typu GUID (Globally Unique Identifier) przechowująca id klienta
- State - wartość enumeracyjna reprezentująca stan klienta. Domyślnie jest to Disconnected
- ServerEndPoint - obiekt przechowujący informacje typu IPEndPoint, na którym pracuje usługa klienta
- Conversations - lista konwersacji klienta
- Connected - event sygnalizujący zmianę stanu na Connected
- DisconnectedByServer - event sygnalizujący wystąpienie zdarzenia gdy klient zostanie odłączony od serwera
- NewConversation - event informujący o nadejściu nowej konwersacji
- Connect - metoda służąca do łączenia aplikacji klienckiej z serwerem aplikacji
- Disconnect - metoda służąca do rozłączania aplikacji z serwerem
- ForceSend - metoda wymuszająca wysłanie wiadomości niezależnie od stanu połączenia z serwerem
- Send - metoda służąca do wysyłania wiadomości do serwera
- SendAsync - rozszerzenie metody Send działające asynchronicznie
- CreateConversation - metoda służąca do wysyłania do serwera informacji o utworzeniu nowej konwersacji
- LoadConversation - metoda służąca do pobierania informacji o konwersacji
- ReceiveLoop - metoda, której zadaniem jest nasłuchiwanie wiadomości od serwera w zapętleniu
- HandleMessage - metoda pomocnicza do obsługi i interpretacji wiadomości odebranych od serwera



- `HandleServerMessage` - metoda pomocnicza do obsługi i interpretacji wiadomości odebranych od serwera
- `HandleConversationMessage` - metoda pomocnicza do obsługi wiadomości tekstowych konwersacji
- `HandleCallMessage` - metoda pomocnicza do obsługi i interpretacji wiadomości o połączeniach głosowych
- `HandleBye` - metoda służąca do obsługi wiadomości z serwera o rozłączaniu połączeń głosowych
- `SendHeartbeat` - metoda, która informuje serwer o aktualnym stanie działania aplikacji klienckiej

#### 4.5.4 Klasa `Conversation.cs`

Klasa będąca modelem konwersacji. Poniżej znajduje się opis używanych przez tę klasę metod oraz pól:

- `Id` - zmienna typu `GUID` (Globally Unique Identifier) przechowująca id konwersacji
- `Client` - obiekt przechowujący informacje o kliencie
- `Call` - obiekt przechowujący informacje o rozmowie audio
- `Participants` - lista przechowująca `GUIDy` uczestników konwersacji
- `TextMessages` - lista przechowująca obiekty klasy `ConversationMessage` powiązanych z konwersacją
- `ParticipantConnected` - event informujący o dołączeniu nowego klienta do konwersacji
- `ParticipantDisconnected` - event informujący o odłączeniu klienta od konwersacji
- `NewCallRequest` - event informujący o nowym nadchodzącym połączeniu głosowym
- `CallRequestRevoked` - event informujący o odrzuceniu przychodzącego połączenia głosowego
- `NewTextMessage` - event informujący o odebraniu nowej wiadomości tekstowej

- NewCall - event informujący o nowym połączeniu
- HandleConversationMessage - metoda pomocnicza do obsługi wiadomości tekstowych konwersacji
- HandleNewUser - metoda pomocnicza do obsługi i interpretacji wiadomości o dołączeniu nowego użytkownika do konwersacji
- HandleUserLeft - metoda pomocnicza do obsługi i interpretacji wiadomości o odejściu użytkownika z konwersacji
- HandleTextMessage - metoda pomocnicza do obsługi i interpretacji wiadomości tekstowych
- HandleCallRequest - metoda pomocnicza do obsługi i interpretacji wiadomości o obsłudze nadchodzących połączeń
- HandleCallRequestRevoked - metoda pomocnicza do obsługi i interpretacji wiadomości odrzuceniu połączenia głosowego
- AddUser - metoda służąca do dodawania nowych użytkowników do konwersacji
- CreateCall - metoda służąca do tworzenia połączenia głosowego dla konwersacji
- RespondToRequest - metoda odpowiadająca za odpowiadanie na zapytania o rozmowę głosową
- DisconnectCall - metoda służąca do odłączania się od połączenia głosowego
- Send - metoda służąca do wysyłania wiadomości do serwera
- Disconnect - metoda służąca do odłączania się od konwersacji

## 4.6 Moduł audio

Projekt współdzielony dla całej solucji udostępniający programowi Rozmawiator kluczowe funkcje dla prowadzenia rozmów głosowych. Projekt Rozmawiator.Audio jest typem projektu "Class Library", którego postacią wynikową po skompilowaniu jest biblioteka dll. Kodekiem wykorzystywanym w projekcie jest kodek G.711 znany także pod nazwą GSM 6.10. Całość dostępna jest dzięki wykorzystaniu biblioteki NAudio. W projekcie wykorzystano następujące klasy:

- `Gsm610Codec.cs`
- `AcmCodec.cs`
- `Player.cs`
- `Recorder.cs`

#### 4.6.1 Klasa `Gsm610Codec.cs`

Klasa będąca wrapperem na `AcmCodec` z biblioteki `NAudio` umożliwiająca kodowanie dźwięku metodą PCM do formatu Wave o charakterystyce: 8000 próbek na sekundę, 16 bitów rozdzielczości próbki oraz jednym kanałem (mono).

#### 4.6.2 Klasa `AcmCodec.cs`

Klasa odpowiadająca za operacje na strumieniu danych (dźwięku). Poniżej znajduje się opis wykorzystywanych metod:

- `IsAvailable` - metoda pozwalająca określić czy kodek ACM jest zainstalowany na komputerze, na którym działa aplikacja Rozmawiator.
- `Encode` - metoda będąca wrapperem metody `Convert` służąca do kodowania strumienia.
- `Decode` - metoda będąca wrapperem metody `Convert` służąca do odkodowania strumienia.
- `Convert` - metoda pozwalająca na konwersję dwustronną strumienia bajtów. Pozwala zakodować strumień bajtów do postaci odpowiadającej kodekowi ACM oraz odkodować strumień zakodowany kodekiem ACM do tablicy bajtów (strumienia).
- `Dispose` - metoda służąca do zamykania strumienia enkodera oraz dekodera, pozwalająca na bezpieczne wyłączenie usług oferowanych przez kodek.

#### 4.6.3 Klasa `Player.cs`

Klasa odpowiadająca za odtwarzanie dźwięku w aplikacji. Jej pola i metody mogą informować o stanie odtwarzania dźwięku, tj. zwracać informacje o tym czy dźwięk jest odtwarzany, zapauzowany oraz czy usługa odtwarzania dźwięku jest zatrzymana. Ponadto klasa ta odpowiada za sterowanie

poziomem głośności w aplikacji. Poniżej znajduje się opis wykorzystywanych metod oraz pól:

- Volume - pole (property) podpowiadające za zwracanie aktualnego poziomu głośności odtwarzania oraz ustawiania poziomu głośności wybranego przez użytkownika końcowego aplikacji.
- SetMute - metoda służąca do wyciszania poziomu odtwarzanego dźwięku lub jego wznowienia.
- AddSamples - metoda będąca wrapperem na typ `BufferedWaveProvider` wbudowany w bibliotekę `NAudio`. Metoda ta dostarcza do metody, którą obudowuje dane potrzebne do zbudowania próbek odtwarzanego dźwięku.
- Start - metoda służąca do uruchomienia odtwarzania dźwięku
- Stop - metoda służąca do zatrzymania odtwarzania dźwięku

#### 4.6.4 Klasa `Recorder.cs`

Klasa, której głównym zadaniem jest obsługa rejestrowania dźwięku z domyślnego urządzenia nagrywającego. Podobnie jak w przypadku klasy `Player`, klasa `Recorder` zawiera metody pozwalające na określenie stanu rejestrowania dźwięku. Ponadto klasa zawiera event, którego zadaniem jest informowanie o dostępności nowych danych z strumienia urządzenia nagrywającego. Poniżej znajduje się lista metod oraz pól:

- State - pole (property) informujące o aktualnym stanie działania modułu rejestrującego dźwięk oraz pozwalająca na przechowywaniu informacji o zmianie wyżej wspomnianego stanu.
- Start - metoda, której głównym celem jest sprawdzenie aktualnego stanu działania modułu nagrywania oraz w zależności od tego stanu inicjalizacja urządzenia nagrywającego oraz strumienia wyjściowego
- Stop - metoda, której głównym celem jest sprawdzenie aktualnego stanu działania modułu nagrywania oraz w zależności od tego stanu zatrzymanie pracy urządzenia nagrywającego
- OnNewData - event, którego zadaniem jest informowanie o dostępności nowych bajtów w buforze strumienia wejściowego

```
private void OnNewData(object sender, WaveInEventArgs waveInEventArgs)
{
    DataAvailable?.Invoke(this, waveInEventArgs.Buffer);
}
```

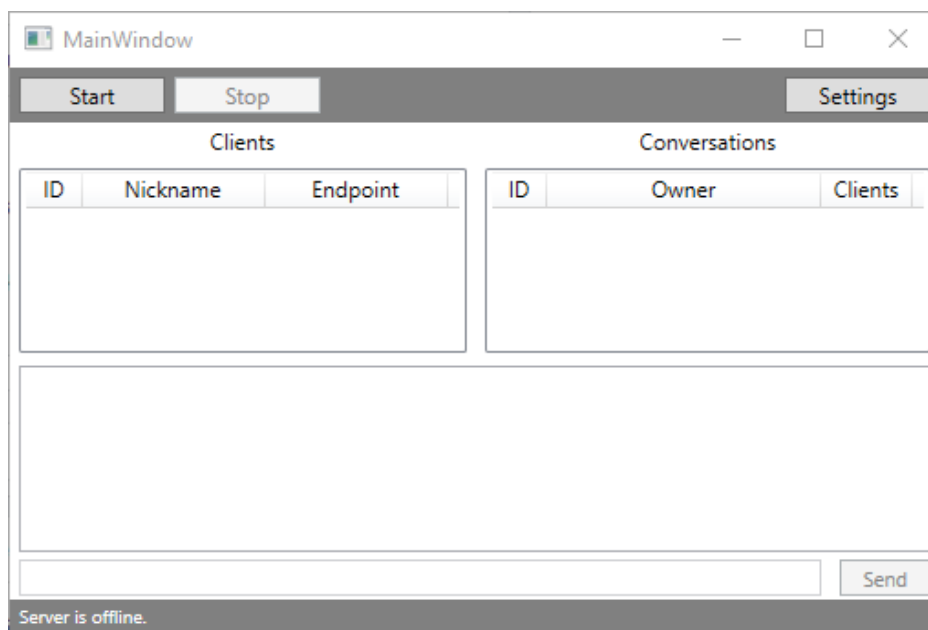
#### 4.6.5 Biblioteki wykorzystywane w projekcie:

- NAudio.NET

## 5 Użytkowanie i testowanie systemu

### 5.1 Serwer

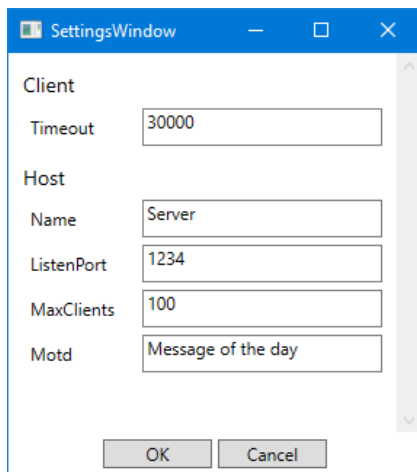
Po uruchomieniu serwera pojawi się okno przedstawione poniżej.



Rysunek 2: Serwer - Widok głównego okna

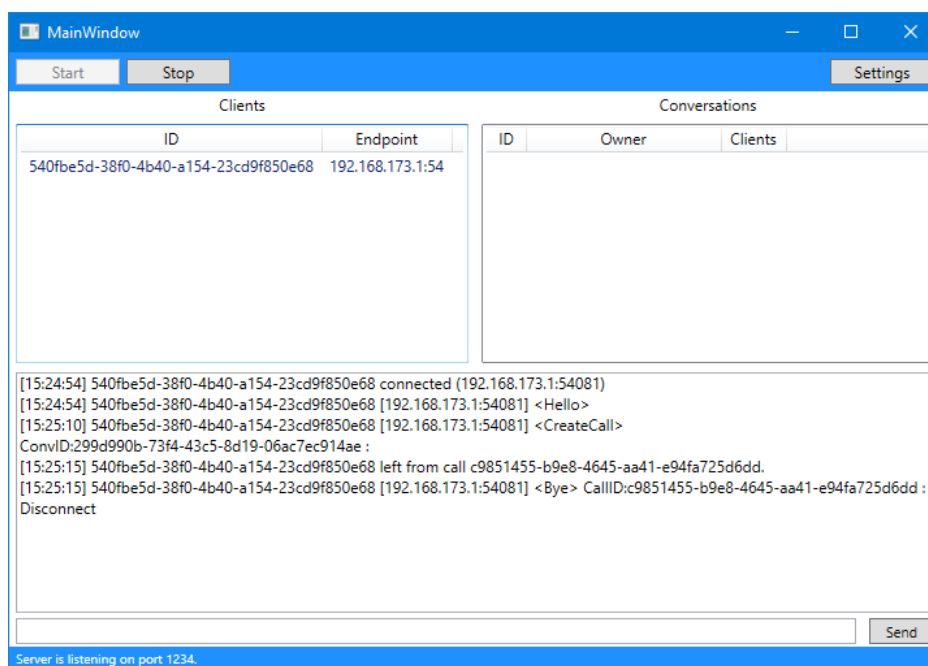
- Widok zatytułowany "Clients" pokazuje informacje o obecnie podłączonych klientach: ich ID, nazwę oraz punkt końcowy.
- Widok zatytułowany "Conversations" pokazuje informacje o obecnie utworzonych konwersacjach: ich ID, właściciela oraz podłączonych doń klientów.

- Pod tymi widokami znajduje się widok przeznaczony na dziennik zdarzeń serwera.
- Jeszcze niżej jest pole, gdzie można wpisać wiadomość, jaką serwer ma rozesłać do wszystkich podłączonych klientów.
- Na samym dole okna jest kontrolka wyświetlająca obecny status serwera.
- przycisk Start - uruchamia serwer
- przycisk Stop - zatrzymuje serwer
- przycisk Settings - otwiera nowe okno widoczne w rysunku 3.



Rysunek 3: Serwer - Widok okna opcji

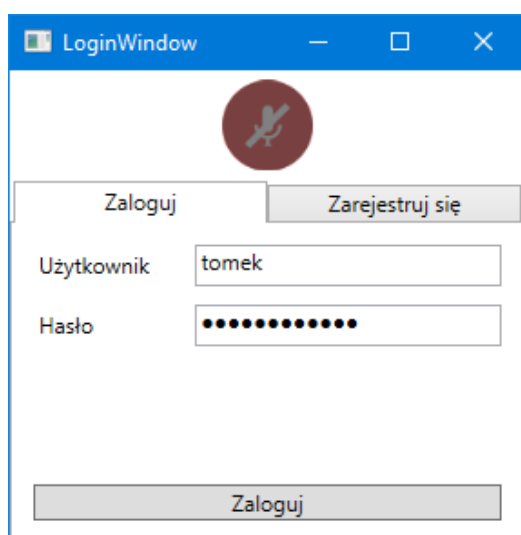
Poniżej pokazana (w widoku dziennika zdarzeń) jest sytuacja, w której klient połączył się z serwerem, utworzył nową konwersację, opuścił ją, a następnie rozłączył się.



Rysunek 4: Serwer - Widok dziennika zdarzeń

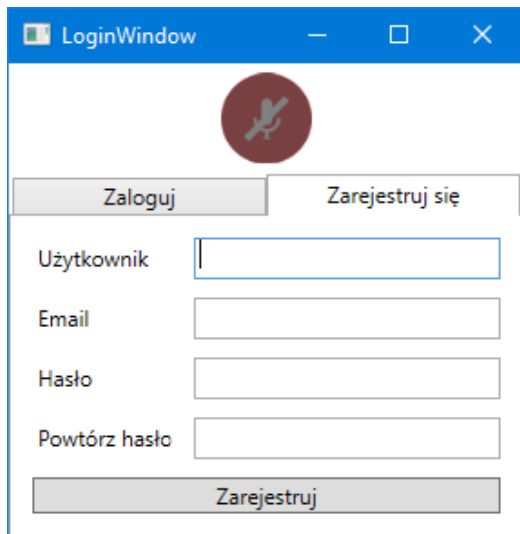
## 5.2 Klient

Po uruchomieniu osobnej aplikacji klienckiej pojawi się następujące okno:



Rysunek 5: Klient - Widok okna logowania

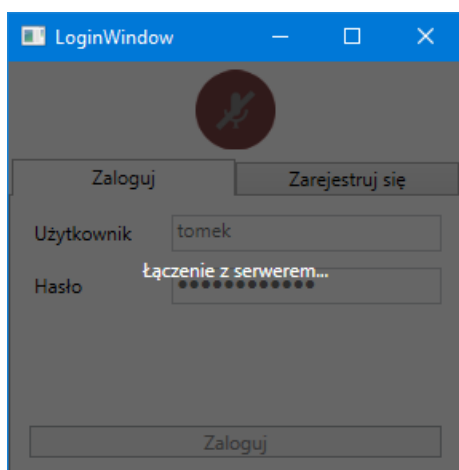
Są tutaj dwie zakładki: "Zaloguj" oraz "Zarejestruj się". W przypadku tej pierwszej pojawiają się dwa pola do wypełnienia: Użytkownik i Hasło, wymagane do połączenia się z serwerem, co jest inicjowane (Po wpisaniu poprawnych danych) poprzez przycisk "Zaloguj". Po kliknięciu w drugą zakładkę można zobaczyć okno pokazane w rysunku 6.



The screenshot shows a window titled "LoginWindow" with a blue header bar. Below the header is a circular logo with a red bird. There are two tabs: "Zaloguj" (selected) and "Zarejestruj się". Under the "Zarejestruj się" tab, there are four input fields labeled "Użytkownik", "Email", "Hasło", and "Powtórz hasło". At the bottom, there is a "Zarejestruj" button.

Rysunek 6: Klient - Widok rejestracji

W przypadku wybrania opcji logowania, operacja potwierdzona zostanie odpowiednim overlayem:

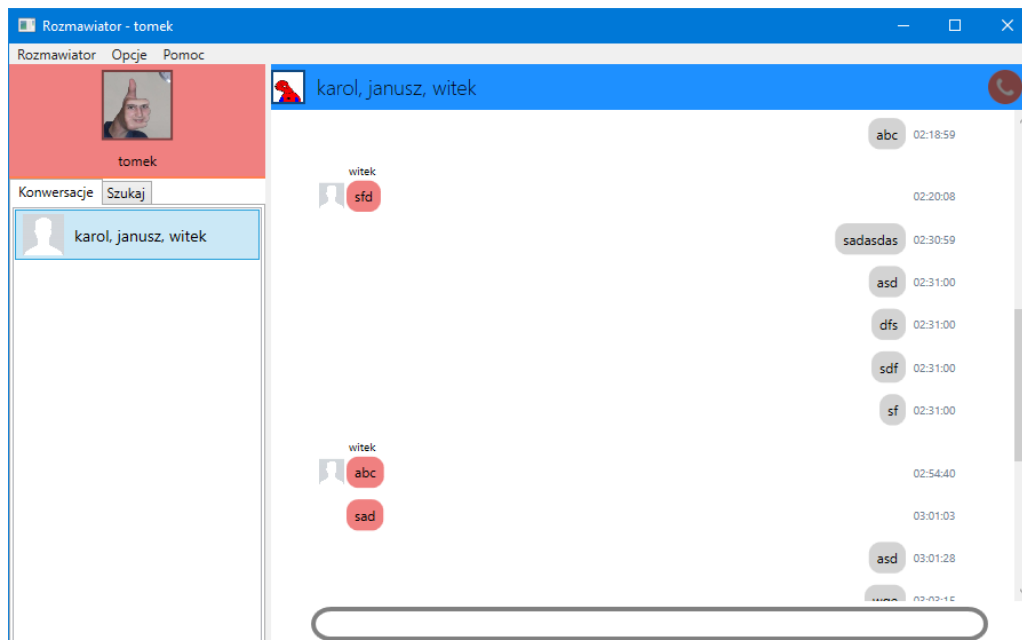


The screenshot shows the same "LoginWindow" application, but now the "Zaloguj" tab is selected. The "Użytkownik" field contains the text "tomek". The "Hasło" field is obscured by a dark overlay with the text "łączenie z serwerem..." and a series of dots. The "Zaloguj" button is visible at the bottom.

Rysunek 7: Klient - Łączenie z serwerem



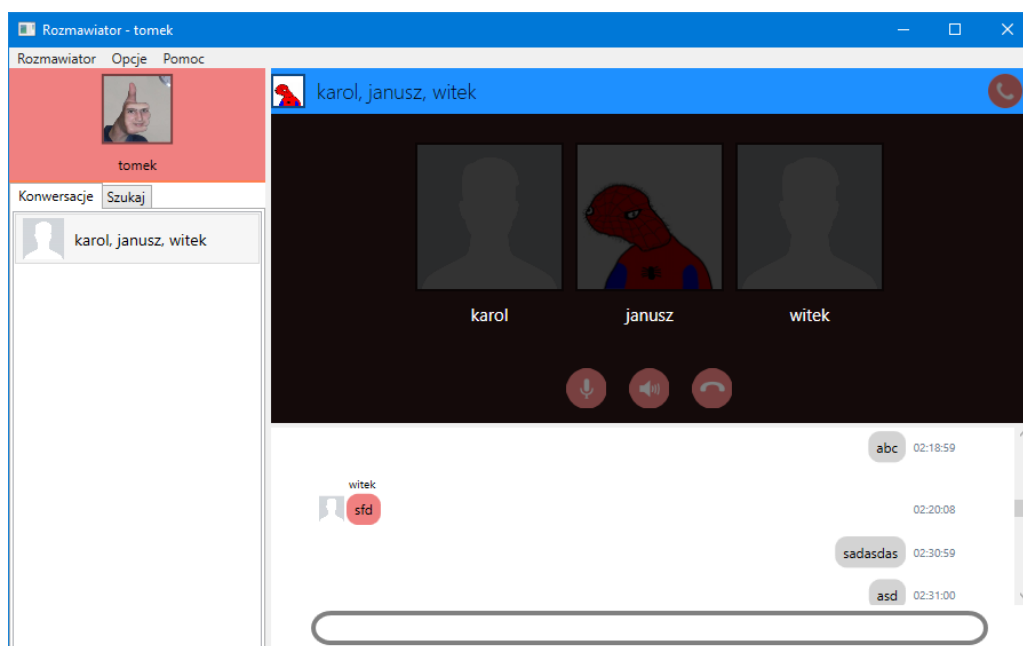
A następnie otworzone zostanie nowe okno:



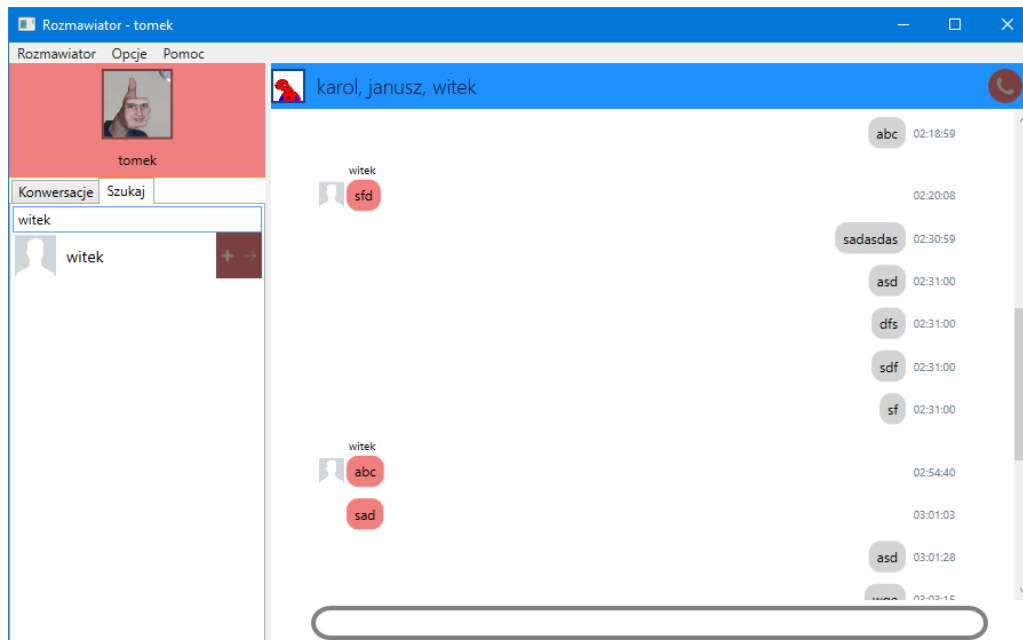
Rysunek 8: Klient - Widok okna po połączeniu z serwerem

Po lewej mamy widok avatara naszego użytkownika oraz jego nazwę. Niżej w zakładce "Konwersacje" mamy listę konwersacji, które możemy wybrać i komunikować się z obecnymi tam użytkownikami. Jest tam również zakładka "Szukaj" przedstawiona w rysunku 10., w której możemy wyszukać konkretnego użytkownika

Po prawej stronie jest widok obecnie zaznaczonej konwersacji, gdzie wyświetlane są wszystkie wiadomości wysłane przez użytkowników danej konwersacji, a w prawym górnym rogu tego widoku jest przycisk pozwalający rozpocząć komunikację głosową, co przedstawione jest w rysunku 9.



Rysunek 9: Klient - Widok rozmowy głosowej



Rysunek 10: Klient - Widok szukania użytkowników

## 6 Kod źródłowy programu

### 6.1 Baza danych

#### 6.1.1 Inicjalizacja kontekstu

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    /* Conversation - User mapping */
    modelBuilder
        .Entity<User>()
        .HasMany(u => u.Conversations)
        .WithMany(c => c.Participants)
        .Map(m =>
        {
            m.MapLeftKey("UserId");
            m.MapRightKey("ConversationId");
            m.ToTable("ConversationParticipants");
        });

    /* Call - User mapping */
    modelBuilder
        .Entity<User>()
        .HasMany(u => u.Calls)
        .WithMany(u => u.Participants)
        .Map(m =>
        {
            m.MapLeftKey("UserId");
            m.MapRightKey("ConversationId");
            m.ToTable("CallParticipants");
        });
    base.OnModelCreating(modelBuilder);

    /* Rename Identity tables */
    modelBuilder.Entity<User>().ToTable("Users", "dbo");
    modelBuilder.Entity<GuidRole>().ToTable("Roles", "dbo");
    modelBuilder.Entity<GuidUserRole>().ToTable("UserRoles", "dbo");
    modelBuilder.Entity<GuidUserClaim>().ToTable("UserClaims", "dbo");
    modelBuilder.Entity<GuidUserLogin>().ToTable("UserLogins", "dbo");
}
```

## 6.2 Serwer REST

### 6.2.1 Filtrowanie zapytań

```
public IQueryable<TModel> FilterQuery<TModel>(IQueryable<TModel> query)
{
    var type = query.ElementType;
    var properties = type.GetRuntimeProperties().ToArray();

    foreach (var filter in Filters)
    {
        var property = properties.FirstOrDefault(p => p.Name == filter.Key);
        if (property == null)
        {
            continue;
        }

        var filterValue = filter.Value;
        filterValue = TryParseToGuid(filterValue) ?? filterValue;

        var param = Expression.Parameter(type);
        Expression<Func<TModel, bool>> condition;

        try
        {
            condition =
                Expression.Lambda<Func<TModel, bool>>(
                    Expression.Equal(
                        Expression.Property(param, filter.Key),
                        Expression.Constant(filterValue, filterValue.GetType())
                    ), param);
        }
        catch (InvalidOperationException)
        {
            return Enumerable.Empty<TModel>().AsQueryable();
        }

        query = query.Where(condition);
    }

    return query;
}
```