AKADEMIA NAUK STOSOWANYCH W TARNOWIE

Wydział Politechniczny

**Kierunek: Informatyka**

Specjalność: Informatyka stosowana

Specjalizacja: Inżynieria oprogramowania

2022/2023

Jakub Krężołek

PRACA INŻYNIERSKA

# Projekt oraz implementacja komunikatora internetowego

Promotor pracy:

mgr inż. Tomasz Gądek

Tarnów, 2023

# Spis Treści

# 1. Wstęp

Komunikacja jest jedną z kluczowych dziedzin we współczesnym życiu ludzi. Jej rozwój można uznać za jeden z ważniejszych czynników rozwoju technologii. Powstało i zapewne powstanie wiele nowych metod oraz technologii, które pozwalają na coraz to efektywniejszą wymianę informacji zarówno w szybkości, niezawodności czy zasięgu ich przesyłu. Zrozumienie, a co za tym idzie, umiejętność wdrażania tych przeróżnych form komunikacji w oprogramowanie jest fundamentalną częścią pracy programisty. Prowadzi nas to do tematu pracy, jakim jest "Projekt oraz implementacja komunikatora internetowego."

## 1.1. Cel pracy

Punktem docelowym pracy jest zaprojektowanie oraz implementacja komunikatora internetowego, który cechuje się dwoma rodzajami komunikacji:

- prywatną, której przebieg mogą śledzić tylko dwie osoby,
- publiczną, która jest dostępna dla określonej grupy użytkowników.

Komunikator powinien również pozwolić użytkownikom na odczyt uprzednio wysłanych wiadomości z poprzednich sesji użytkowania.

## 1.2. Zakres pracy

Na potrzeby pracy zbudowana została aplikacja oparta na architekturze klient-serwer. Dlatego kluczowymi komponentami, na jaki składa się praca są:

- aplikacja internetowa,
- aplikacja serwerowa,
- baza danych.

Komunikacja między aplikacją serwerową, a aplikacją internetową odbywa się za pośrednictwem dwóch metod. Pierwszą z nich są zapytania HTTP (ang. Hypertext

Transfer Protocol). Drugą natomiast jest ciągły transfer danych za pośrednictwem dwóch protokołów Websocket oraz STOMP (ang. Streaming Text Oriented Messaging Protocol).

Aplikacja została udokumentowana z użyciem platformy znanej pod nazwą “SwaggerHub”. Szczegółowy opis użytych protokołów oraz technologii można znaleźć w rozdziale drugim tego dokumentu.

Architekturę oraz sposób użycia opisanych w rozdziale pierwszym protokołów oraz technologii do zrealizowania pomniejszych kamieni milowych w trakcie implementowania aplikacji umieszczono w trzecim rozdziale. W rozdziale czwartym umieszczono opis aplikacji oraz schematy opisujące funkcjonalności, z którymi użytkownik może wchodzić w interakcję.

# 2. Opis użytych narzędzi oraz technologii

W tym rozdziale opisane zostaną zagadnienia, dotyczące użytych technologii.

## 2.1. Zagadnienia ogólne

Opis terminów używanych w pracy, które stanowią część słownika informatycznego wykorzystanego by opisać pozostałe elementy pracy.

### 2.1.1. Framework

Jest to szkielet aplikacji. Oznacza on bibliotekę lub rozszerzenie danego języka programowania. Charakteryzuje się on większym rozmiarem niż standardowa biblioteka lub rozszerzenie oraz narzuca on pewne standardy, paradygmaty programowania lub wzorce projektowe. Sprawia to, że użytkownicy danego Frameworku są poniekąd zmuszani do pisania kodu w narzucony przez twórcę sposób. Oczywiście programista może odrzucić tego rodzaju konwencję programowania i pisać kod po swojemu, ale istnieje parę dobrych powodów do stosowania tego typu konwencji:

- jednym z takich powodów jest próba wymuszenia twórcy na programistach pewnych dobrych praktyk, które mają pomóc programiście zrozumieć ideę danego Frameworku, ale także ułatwić mu z nim pracę,
- w trakcie wybierania takiego rodzaju rozszerzeń języka warto pamiętać, że powstają zazwyczaj one w celu ułatwienia pracy w konkretnych przypadkach i mają one rozwiązywać przewidziane problemy,
- wymuszona konwencja pozwala, aby programista w trakcie korzystania z naszego rozszerzenia skupił się na rozwiązaniu problemu, a nie zastanawiał się, jakie jest najlepsze do niego podejście,
- kolejnym dobrym powodem na istnienie takiej konwencji, jest jednolitość kodu. Dzięki tej jednolitości dokumentacja zawsze charakteryzuje się jednym stylem i jest prostsza w zrozumieniu. Pozwala to nowym programistom, jak i tym z większym doświadczeniem w danym Frameworku się zrozumieć. Wpływa to na czas i poziom trudności, z jakim się wiąże zarówno wdrożenie się w dane rozszerzenie, jak i jego rozwój osobom niezależnym.

### 2.1.2. ORM (ang. Object–Relational Mapping)

Jest to technika oprogramowania, którą opisujemy narzędzia programistyczne zaimplementowanych w celu mapowania elementów relacyjnej bazy danych na kolekcje obiektów [5].

### 2.1.3. POJO oraz programowanie aspektowe

Zarówno POJO, jak i programowanie aspektowe są to elementy, które propagują pisanie kodu w taki sposób, aby metody, klasy i obiekty w kodzie miały tylko jedną odpowiedzialność. Służy to zmniejszeniu powiązania elementów aplikacji między sobą, bardziej wyraźnego podziału kodu na elementy, które spełniają jedną funkcję. Poprawia to także czytelność kodu oraz pozwala to na więcej okazji dla programisty do refaktoryzacji.

### 2.1.4. Refaktoryzacja

Jest to zabieg, w którym programista zmienia częściowo lub całkowicie istniejące już funkcjonalności, aby poprawić jego cechy takie jak czytelność, efektywność itd.

## 2.2. Aplikacja Serwerowa

W podrozdziale opisano technologie, biblioteki oraz rozszerzenia wykorzystane w aplikacji serwerowej.

### 2.2.1. Java

Jest to język programowania ogólnego przeznaczenia [1]. Przykłady jego zastosowania:

- aplikacje internetowe (webowe),
- aplikacje okienkowe (desktopowe),
- aplikację mobilne,
- aplikacje oparte na chmurze itd.

Java mimo trzydziestoletniego stażu, jest dalej szeroko stosowanym językiem. Są cztery najważniejsze zalety, dzięki którym język Java zawdzięcza swój sukces:

- maszyna wirtualna - Java kompiluje się do tzw. kodu pośredniego. Kod następnie używany jest w maszynie wirtualnej, która dostosowuje działanie kodu do systemu oraz sprzętu, na jakim jest uruchomiony. Jest to ogromna zaleta, gdyż sprawia to, że aplikacja napisana w tym języku może zostać uruchomiona na różnych platformach systemowych,
- wieloparadygmatowość - Java jest językiem obiektowym. Liczba funkcjonalności sukcesywnie się rozrasta, co pozwala na wykorzystanie innych paradygmatów. Jest to cecha, którą Java nabyła. Wspomniane funkcjonalności zostały zaimplementowane zarówno przez autorów języka, jak i przez autorów niezależnych.
- biblioteki - czyli obszerne i bez przerwy rosnące zaplecze rozszerzeń. Pozwala to na poszerzenie horyzontów dla tego języka oraz jego zastosowań. Wpływa to też na przyspieszenie pracy dla każdego programisty, który ma z nim do czynienia,
- dokumentacja - język Java cechuje się bardzo łatwą w zrozumieniu szczegółową dokumentacją. Cecha ta jest obecna zarówno w funkcjach podstawowych, ale również w rozszerzeniach od twórców niezależnych.

Cechy drugiej, trzeciej oraz czwartej nie można było natomiast uświadczyć w tym oto języku w pierwszych latach jej funkcjonowania. Są to zalety, które zostały wypracowane

na przestrzeni lat w miarę rozwoju omawianej technologii. Mimo wielu zalet wszystko ma swoje wady:

- największą wadą języka Java, jest to, że uznaje się go za język stosunkowo wymagający. Wynika to z faktu, iż wymaga on większej ilości pamięci RAM (ang. Random Access Memory) oraz mocy obliczeniowej niż większość ogólnie dostępnych języków programowania,
- drugą wadą, jaką przytaczają niektórzy programiści, która jest już nieaktualna, jest to, że w przeszłości język ten był rzadko aktualizowany przez głównych twórców.

### 2.2.2. Spring

Spring to Framework. Został on opracowany w 2003 roku z prostej przyczyny. Przyczyną tą była obszerna ilość kodu oraz konfiguracji, jaka jest potrzebna, aby w czystym języku Java napisać aplikację na poziomie biznesowym. Aplikacje biznesowe to raczej kategoria aplikacji, które cechują się zazwyczaj sporym rozmiarem i szeregiem wymagań, ze strony klienta. Wiele tego typu aplikacji napisanych w języku Java charakteryzowały się powtarzalnym kodem oraz konfiguracją. Spring miał ograniczyć tę powtarzalność, dzięki czemu aplikacja mogła być napisana stosunkowo szybko. Springowi udało się to zadanie. Sukces ten osiągnął poprzez wprowadzenie bardzo szerokiego zakresu funkcjonalności, które zmniejszały ilość powtarzających się fragmentów kodu. Polityka twórców też miała w tym swój udział. Narzucony przez nich styl programowania aspektowego oraz wzorzec projektowy POJO (ang. Plain Old Java Object) też miał w tym sukcesie swój udział [9].

### 2.2.3. Spring Boot

Zanim opiszę czym jest Spring Boot i co go charakteryzuje warto wspomnieć o jego poprzedniku, czyli tzw. Springu. Mimo powstania Springa i ułatwień, które wprowadzili, programiści dalej mieli sporo kodu do napisania przed sobą podczas implementacji aplikacji. Co więcej, w Springu brakowało ułatwień zmniejszających ilość konfiguracji, jaką trzeba było wykonać, na potrzeby aplikacji oraz mimo zmniejszenia powiązania elementów aplikacji. Wspomniane elementy dalej były w pewnym stopniu ze sobą

powiązane. Dlatego na scenę wkroczył Spring Boot. Jest to następca Frameworka Spring. Aby rozwiązać dalej istniejące problemy i wzmocnić aktualne propagowany przez Spring styl pisania kodu, Spring Boot dodał kolejne elementy:

- dzięki nowym funkcjonalnościom i udogodnieniom dla programistów, rozwiązanie problemu z konfiguracją zostało w znacznym stopniu osiągnięte.
- pozbyto się potrzeby tworzenia tzw. deskryptora wdrożeniowego. Aplikację opartą zarówno na Spring, jak i Spring Boot trzeba uruchomić na serwerze lub w kontenerze. Aby to zrobić, trzeba ten powyżej wspomniany plik zdefiniować. Spring nie tworzy takiego pliku ani nie dostarcza żadnego z tych elementów potrzebnych do uruchomienia aplikacji. Natomiast Spring Boot nie dość, że zapewnia jedną instancję serwera Tomcat wraz ze standardową paczką podczas jego pobrania, to całkowicie niweluje potrzeby tworzenia deskryptora wdrożeniowego,
- poza automatycznie skonfigurowanym środowiskiem Spring Boot udostępnia szereg adnotacji, których stosowanie w kodzie zapewnia automatyczne wiązanie elementów. Połączono to ze wzorcem projektowym, jakim jest wstrzykiwanie zależności, pozbywa się jakiejkolwiek potrzeby powiązania ze sobą elementów kodu.

Kod napisany z jego wykorzystaniem można aktualizować i podmienić bez ingerencji w inne elementy kodu oraz ogranicza to potrzebę tworzenia konfiguracji.

### 2.2.4. Websockets

Oznacza protokół, który jest powszechnie używany na wielu stronach i aplikacjach internetowych. Jest równie popularny, jak protokół HTTP [2][3]. Pozwala on na utworzenie dwukierunkowego kanału komunikacyjnego pomiędzy serwerem, a inną aplikacją lub serwerem. Umożliwia to na zastosowania, do których protokół HTTP nie nadaje się najlepiej. Najczęstszym z takich zastosowań, jest szybka wymiana informacji w krótkich odstępach czasu. Protokół HTTP nie byłby w takim przypadku najlepszym rozwiązaniem, gdyż przed każdą wymianą informacji potrzebne jest żądanie, które musi wysłać aplikacja kliencka. Wynikiem tego jest szereg czynności, które aplikacja serwerowa musiałaby wykonać przed dostarczeniem zarządzanych informacji. Wadą tego podejścia jest brak możliwości wysłania informacji ze strony serwera do aplikacji klienckiej bez wspomnianego żądania. Protokół WebSockets znajduje tu swoje zastosowanie. Uruchomiony przy jego użyciu kanał komunikacyjny nie wymaga dalszych

żądań. Dzięki temu rozwiązaniu aplikacja kliencka może wymieniać dane z serwerem w trybie ciągłym. Aby taki kanał powstał, muszą być wykonane następujące kroki:

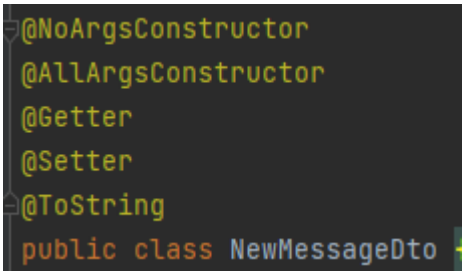
- aplikacja kliencka musi zażądać utworzenia takiego kanału za pośrednictwem protokołu HTTP,
- aplikacja serwera wysyła tzw. handshake za pośrednictwem protokołu HTTP Upgrade,
- po pomyślnym handshakeu obu stron utworzony zostaje kanał, po którym komunikacja odbywa się dzięki protokołowi TCP.

### 2.2.5. STOMP ( Simple Text Oriented Message Protocol )

Protokół tekstowy, który zapewnia tzw. pośrednika komunikacji (ang. message broker). Wspomniany pośrednik umożliwia wysyłanie wiadomości w prostym formacie tekstowym. Protokół ten jest bardzo podobny do protokołu HTTP [4][10].

### 2.2.6. Lombok

Jest jedną z szeroko wykorzystanych bibliotek języka Java. Jest to niewielka biblioteka, która udostępnia wiele różnych adnotacji. Adnotacje pozwalają uniknąć powtórzeń w kodzie źródłowym.




Rys. 2.1 - Przykładowe adnotacje biblioteki Lombok [opracowanie własne]

Na rysunku 2.1 zaprezentowano pięć przykładowych adnotacji, które możemy wykorzystać dzięki rozszerzeniu Lombok. Adnotacje te pozwalają na automatyczne generowanie kodu. Oznaczają one kolejno:

- @NoArgsConstructor - adnotacja tworząca konstruktor bezparametorwy dla klasy,

- @AllArgsConstructor - adnotacja tworząca konstruktor klasy z wymogiem podania takiej ilości parametrów jaka jest ilość atrybutów klasy,
- @Getter - tworzy ona tzw. *gettery* klasy dla każdego atrybutu klasy,
- *@Setter* - tworzy ona tzw. *settery* klasy dla każdego atrybutu klasy,
- @ToString - tworzy tzw. funkcję *toString*. Służy reprezentacji obiektu w formie tekstowej.

### 2.2.7. Hibernate

Jest to Framework ORM, który został zaimplementowany w oparciu o standardy JPA. JPA jest to zestaw reguł oraz zasad, jakimi powinien charakteryzować się ORM podczas łączenia obiektów z relacyjną bazą danych. Pomaga on połączyć serwer z relacyjną bazą danych, które są:

```
@Repository
public interface UserRepository extends CrudRepository<UserEti, String> {

    Jakub Krezolek
    List<UserEti> findAll();

    1 usage  Jakub Krezolek
    UserEti findByUserName(String userName);

}
```

Rys. 2.2 - Przykładowa klasa z wykorzystaniem Frameworku Hibernate [opracowanie własne]

- prostsze, ponieważ z jego wykorzystaniem nie ma potrzeby pisania zapytań do bazy danych w odpowiadającym jej języku. Zaprezentowano przykładową klasę z metodami, które reprezentują zapytania na rysunku 2.2. Przykładem takiego zapytania jest metoda “findAll”, którą można zinterpretować jako
“SELECT * FROM …”.

- bezpieczniejsze, gdyż Hibernate zapewnia pewne formy ochrony bazy danych przed nieautoryzowanym dostępem. Przykładem takiej ochrony jest odporność na SQL Injection.

Dodatkowo w przypadku zapytań aktualizacji oraz usunięcia danych zostały wprowadzone pewne udogodnienia. W klasycznej sytuacji, takie zapytanie w języku **SQL**, musi zawierać klauzule **WHERE**. Pozwala ono określić jaki rekord należy zaktualizować lub usunąć. Hibernate niweluje potrzebę tego typu zabiegu, ponieważ wykona go za programistę. Jest to możliwe jeśli przekażemy w ramach zapytania cały obiekt w zmienionej postaci. Hibernate w takiej sytuacji sam znajdzie rekord, który mu odpowiada i przeprowadza na nim operacje [6][7][8].

## 2.3. Aplikacja internetowa

W tym podrozdziale opisano biblioteki, rozszerzenia oraz technologie wykorzystane podczas budowy aplikacji klienckiej.

### 2.3.1. JavaScript

JavaScript został zbudowany na podstawie specyfikacji ECMAScript. Najważniejszym faktem, jaki trzeba wiedzieć na temat tego języka, jest to, że jest on bardzo popularny. Jest to język wieloparadygmatowy z wieloma funkcjonalnościami. W rezultacie otrzymujemy bardzo dynamicznie rozwijający się język.

### 2.3.2. React

Jest to biblioteka języka JavaScript. Została zaprojektowana, w celu ułatwienia implementacji dynamicznych interfejsów użytkownika. Najważniejszymi elementami tego rozszerzenia, które wyróżniają je na tle innych tego typu bibliotek lub Frameworków to:

- JSX jest to format plików, w którym można pisać kod łączący ze sobą składnie języków JavaScript, HTML oraz CSS,
- huki (ang. hooks), czyli elementy, dzięki którym aplikacja może w dynamiczny sposób zarządzać stanem. Jest ich wiele rodzajów oraz pomagają one również w zarządzaniu przepływem danych przez aplikację,
- wirtualny DOM (ang. Document Object Model) - to struktura, która odzwierciedla drzewo elementów w aplikacjach napisanych w React. W przeciwieństwie do DOMu przeglądarki przy wprowadzaniu zmian na stronie internetowej, znajduje on elementy w wspomnianym drzewie, których modyfikację dotyczą. Dzięki czemu adaptacja drzewa DOM przeglądarki może być ograniczona do minimum.

### 2.3.3. Axios

Kolejna biblioteka języka JavaScript, która ułatwia programiście pisanie zapytań potrzebnych do komunikacji aplikacji klienckiej z serwerową. Wykorzystuje ona tzw. Obietnice (ang. Promise). Obietnice są to elementy języka JavaScript, które rozwiązują problemy asynchroniczności tego języka. Kod pisany w tym języku może zostać otoczony taką obietnicą. Spowoduje to, że zamiast zwrócić konkretną wartość, zostanie zwrócona obietnica, której rezultat zależy od rezultatu wykonanego kodu. Otoczony w obietnice kod ma do dyspozycji dwie dodatkowe funkcje:

- Settled - wywołanie tej funkcji informuje obietnicę, że kod wykonał się poprawnie i zakończy ona jego wykonanie z takim rezultatem.
- Rejected - wywołanie tej funkcji informuje obietnicę, że kod w niej wykonany natrafił na problem.

Axios z użyciem obietnic udostępnia programiście funkcjonalność, która po przeprowadzeniu zapytania może zostać obsłużona na trzy sposoby przez programistę:

- zapytanie powiodło się,
- wystąpił problem związany z zapytaniem,
- wystąpił błąd kodu przed lub po zapytaniu.

### 2.3.4. stomp js

Biblioteka, która udostępnia programiście funkcjonalności do operowania pośrednikiem komunikacji wykorzystującym protokół STOMP oraz formę komunikacji protokołu WebSocket [4].

### 2.3.5. MUI

Jest to biblioteka, która udostępnia gotowe do użytku komponenty, których można użyć do budowania aplikacji. Opiera się ona na licencji open-core, ale dla firm dostępne są także inne licencje, w ramach których dostępna jest dodatkowa zawartość oraz wsparcie ekspertów.

# 3. Architektura oraz Implementacja

W tym rozdziale zostaną opisane funkcjonalności, jakimi cechuje się system zaimplementowany na potrzeby pracy dyplomowej.

## 3.1. Spektrum funkcjonalności

W napisanej aplikacji dostępnych jest dwóch aktorów:

- użytkownik niezalogowany,
- użytkownik zalogowany.

Użytkownik niezalogowany ma możliwość zarejestrowania się. Podczas rejestracji użytkownik może wybrać swoją nazwę użytkownika, hasło oraz zdjęcie profilowe. Próba wejścia na inne podstrony aplikacji bez zalogowania się nie wyłoni żadnych funkcji oraz żadne dane nie zostaną pobrane z serwera.

Użytkownik zalogowany ma do dostęp do wszystkich funkcji aplikacji. Po zalogowaniu dostaje do dyspozycji następujące opcje:

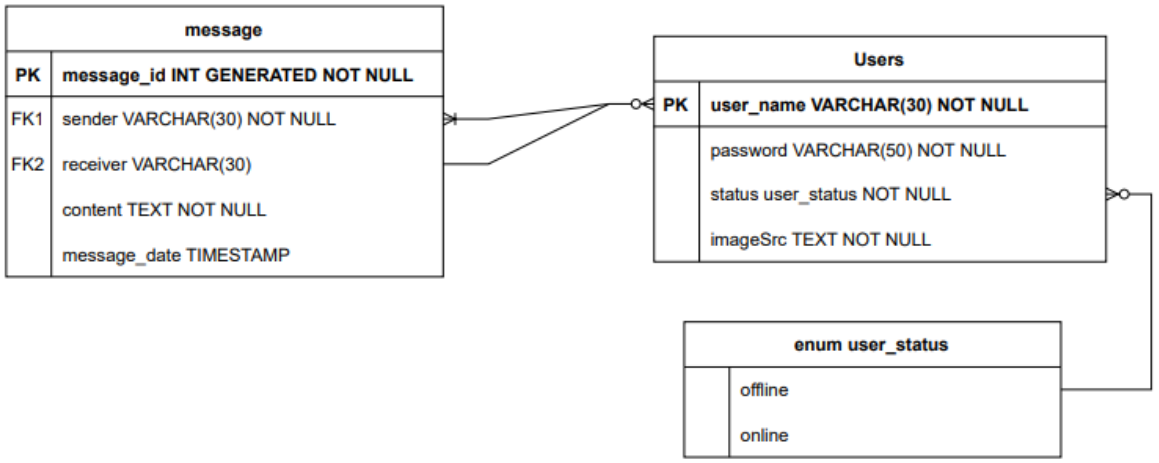
- wylogowania,
- jednego czatu publicznego, do którego wgląd mają wszyscy użytkownicy,
- czatu prywatnego z każdym zalogowanym użytkownikiem.

Poza wymienionymi funkcjonalnościami aplikacja zapewnia:

- system statusu - pozwala nam zweryfikować kto aktualnie jest zalogowany,
- wiadomości historyczne - po powrocie użytkownika do aplikacji lub po wylogowaniu zostaną przedstawione mu zaległe rozmowy oraz wiadomości, które otrzymałby w czasie jego nieobecności.

## 3.2. Diagram ERD

ERD (ang. Entity Relationship Diagram), jest to graficzna prezentacją strkutury relacyjnej bazy danych.

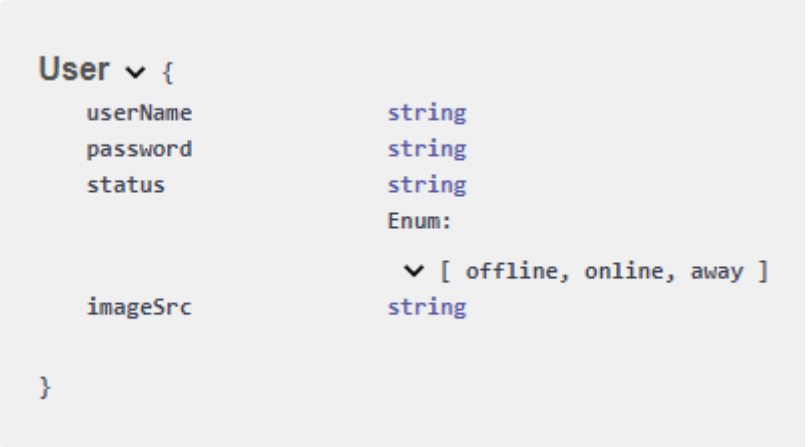



Rys. 3.1 - Diagram ERD bazy danych [opracowanie własne]

Baza danych (rys. 3.1) składa się z dwóch encji oraz typu wyliczeniowego **user_status**. Typ wyliczeniowy służy za flagę oznaczającą status użytkownika w serwisie. W zależności od stanu, użytkownik może być **online** lub **offline**. Encja **users** służy do przechowywania danych o użytkownikach. Natomiast encja **message** przechowuje informację o wiadomościach. Dane użytkowników używane są w encji **message** do identyfikowania nadawcy oraz odbiorcy wiadomości. W przypadku, gdyby wiadomośc została skierowana do czatu publicznego, w pole odbiorcy należy wstawić wartość **null**.

## 3.3. Dokumentacja

Dokumentacja została sporządzona przy pomocy narzędzia o nazwie “SwaggerHub”. Pozwala ono szczegółowo udokumentować tzw. endpointy. Udokumentowane endpointy można podzielić na dwa rodzaje. Pierwszym z nich są to endpointy, które umożliwiają utworzenie kanału komunikacyjnego z użyciem protokołu WebSocket. Natomiast druga grupa endpointów, pozwala na wykonywanie do nich zapytań HTTP.



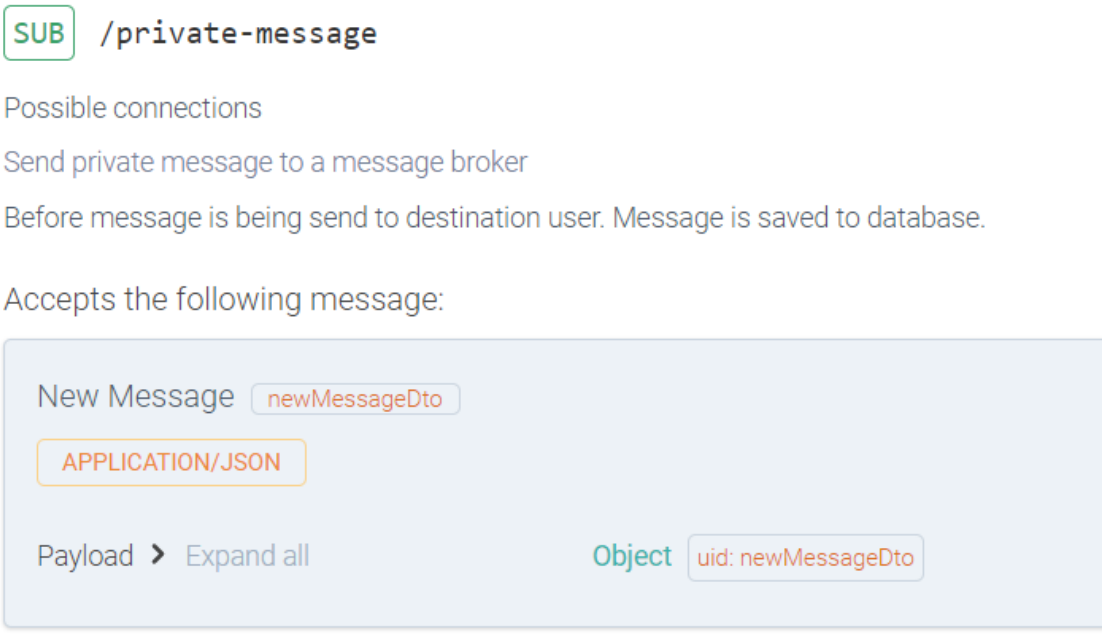

Rys. 3.2 - Struktura formatu metadanych o użytkownikach [opracowanie własne]


Rys. 3.3 - Format danych przedstawiający wiadomość [opracowanie własne]

Na rysunku 3.2 przedstawiono format danych wykorzystywany w aplikacji do przesyłu danych o użytkownikach. Rysunek 3.3 przedstawia format danych, który był używany do przesyłu wiadomości.

Aby ułatwić zrozumienie pewnych założeń projektowych wiadomości podzielono na dwa rodzaje:

- wiadomość klasyczna, jest to wiadomość wysyłana między użytkownikami zawierająca treść czatu.
- wiadomość funkcjonalna, której wykorzystanie służy do zmiany stanu statusu użytkownika.



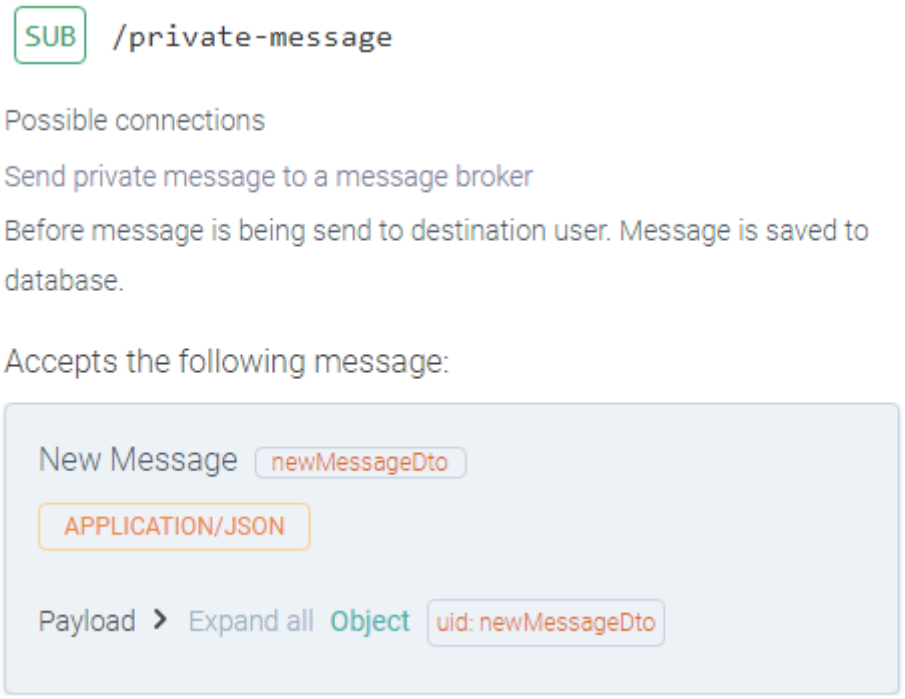

Rys. 3.4 - Endpoint obsługujący wiadomości prywatne [opracowanie własne]

Przedstawiony na rysunku 3.4 endpoint wykorzystuje protokół websocket. Połączyć się z nim możemy wykorzystując adres:

```
/user/{nazwa użytkownika}private
```

Nazwa użytkownika obecna w tym adresie odpowiada nazwie aktualnie zalogowanej osoby. Służy to do identyfikacji połączonego użytkownika.

Endpoint służy do obsługi prywatnych wiadomości klasycznych. Przesłane do niego wiadomości zostają zapisane do bazy danych oraz wysłane do odbiorcy.



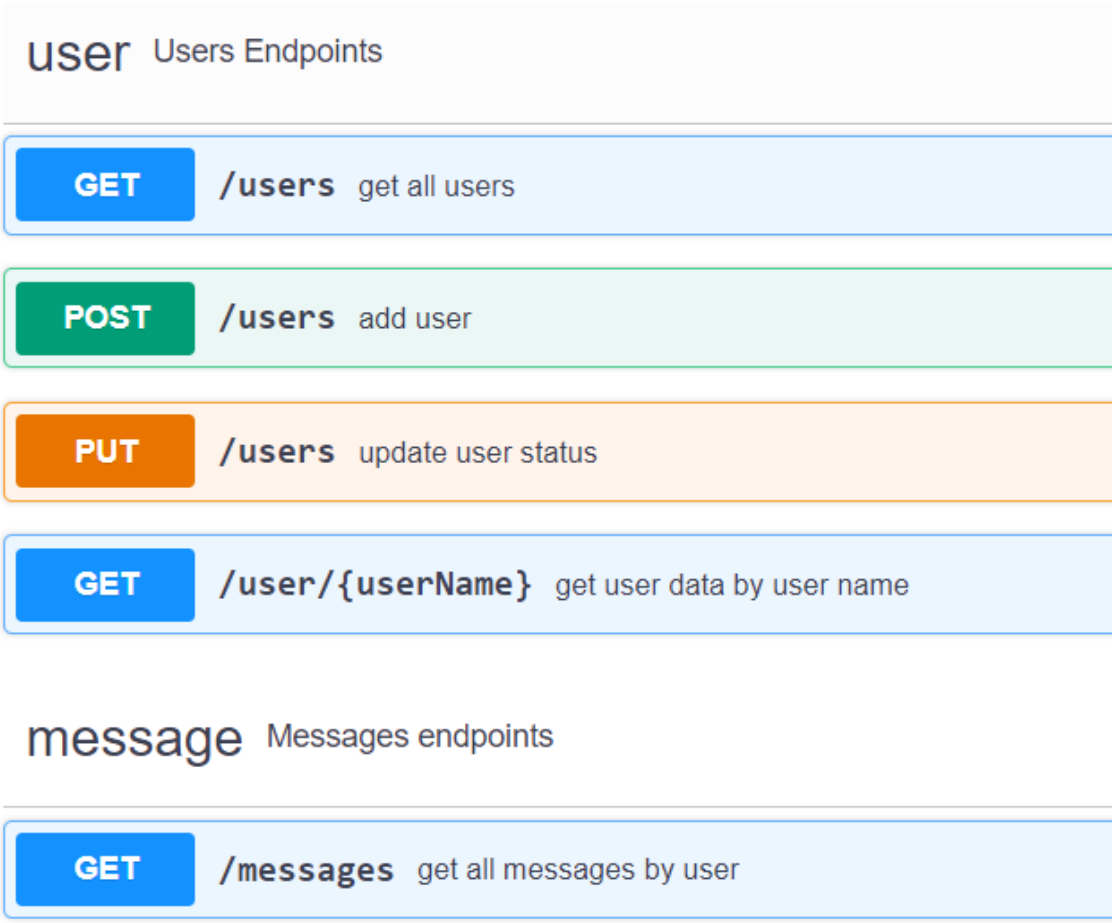

Rys. 3.5 - Endpoint obsługujący wiadomości publiczne [opracowanie własne]

Endpoint zaprezentowany na rysunku 3.6 jest używany do dwóch celów:

- pierwszym z nich są to wiadomości klasyczne dostępne do odczytu w publicznym oknie czatu,
- drugim są to wiadomości funkcjonalne wysłane w celu aktualizacji statusu użytkowników.

Aplikacja rozróżnia wiadomości poprzez wartość wysłaną w elemencie pod nazwą status, obecną w metadanych wiadomości (rys. 3.3). Element **status** może przybrać jedną z trzech wartości:

- **MESSAGE** - informuje ona endpoint o tym, że jest to wiadomość klasyczna,
- **JOIN** - wiadomość funkcjonalna, służy do poinformowania aplikacji, że użytkownik o nazwie widniejącej w polu **senderName**, zalogował się i należy zmienić jego status,
- **LEAVE** - jest to wiadomość funkcjonalna, która daje aplikacji znać, że użytkownik się wylogował.




Rys 3.6 - Pozostałe endpointy [opracowanie własne]

Na rysunku 3.6 zostały zaprezentowane endpointy, które wykorzystują protokół HTTP. Cztery z nich zostały utworzone na potrzeby pobierania lub manipulacji danymi o użytkownikach. Ostatni endpoint natomiast służy pozyskiwaniu wiadomości klasycznych na podstawie nazwy użytkownika. Endpointy pod kategorią “user” przedstawione na rysunku (rys. 3.6) mają następujące działanie:

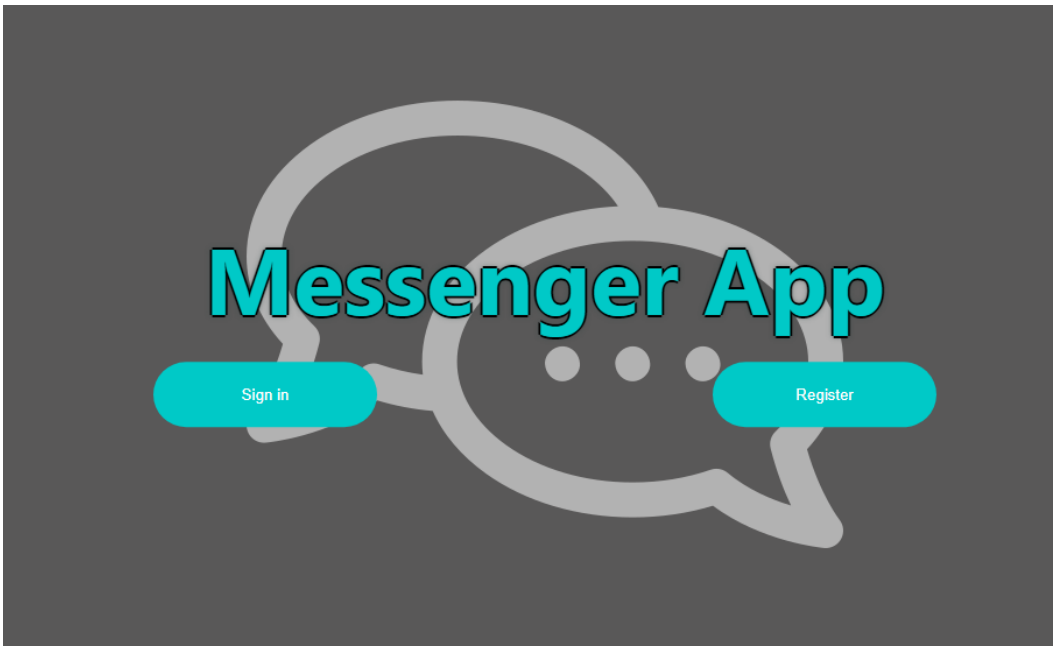
- GET /users - pobranie danych o wszystkich użytkownikach,
- POST /users - dodawanie użytkowników,
- PUT /users - edycja statusu użytkownika,
- GET /users/{userName} - pobranie informacji na temat konkretnego użytkownika o nazwie określonej w parametrze **userName**,

# 4. Interfejs użytkownika

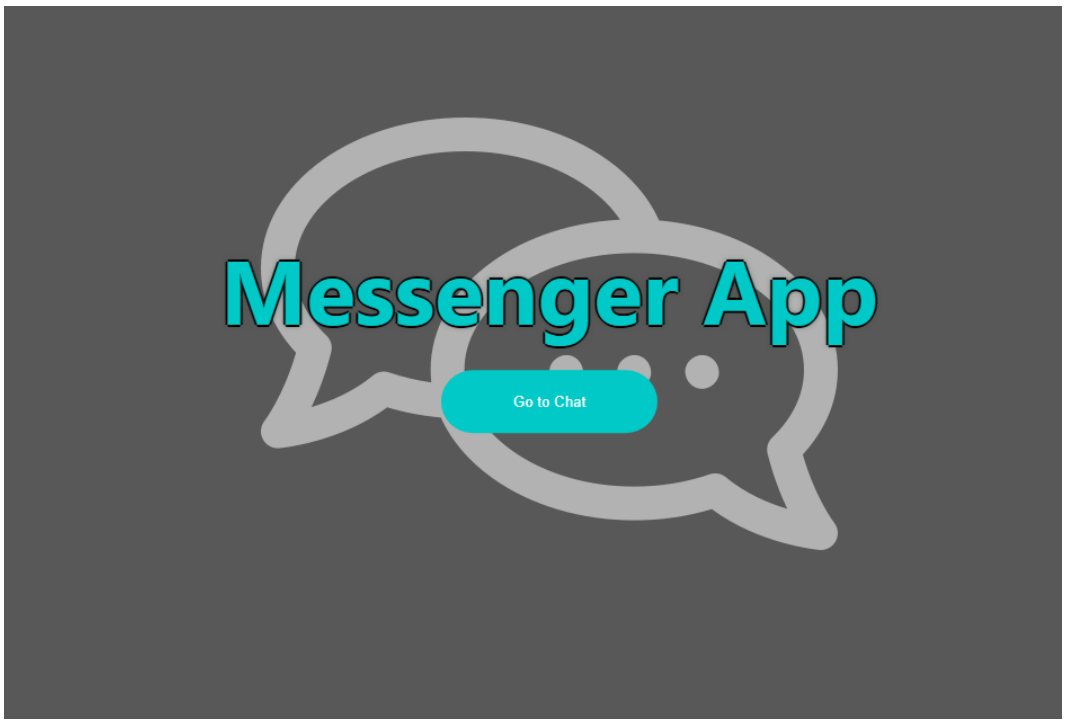
Strona wizualna aplikacji została wykonana z wykorzystaniem biblioteki React. Jej treść mieści się na 4 ekranach:

- ekran główny,
- ekran logowania,
- ekran rejestracji,
- ekran czatu.

## 4.1. Ekran główny



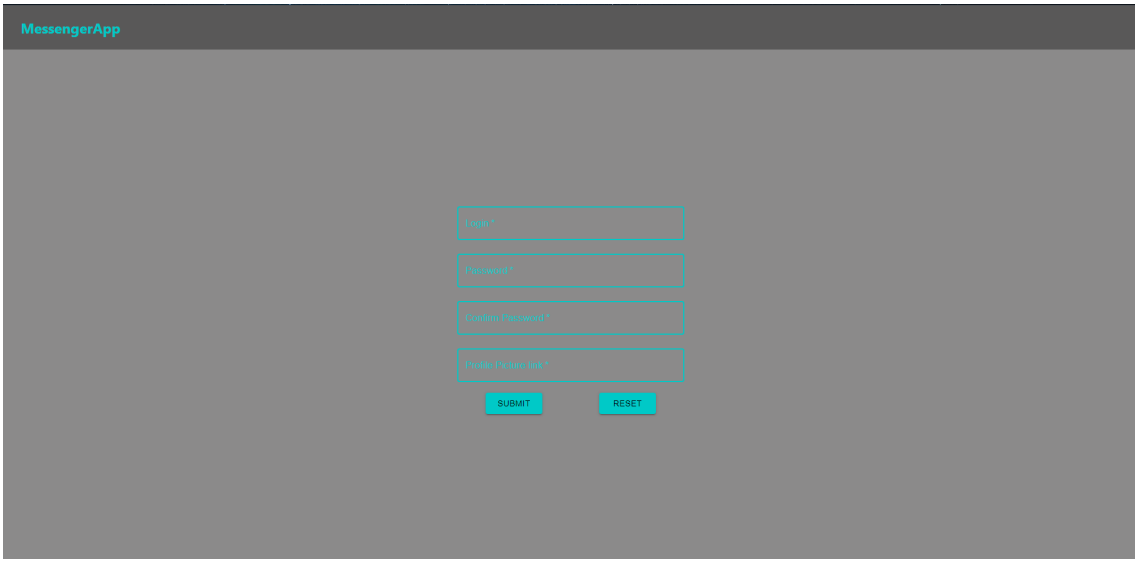

Rys. 4.1 - Ekran główny przed zalogowaniem [opracowanie własne]



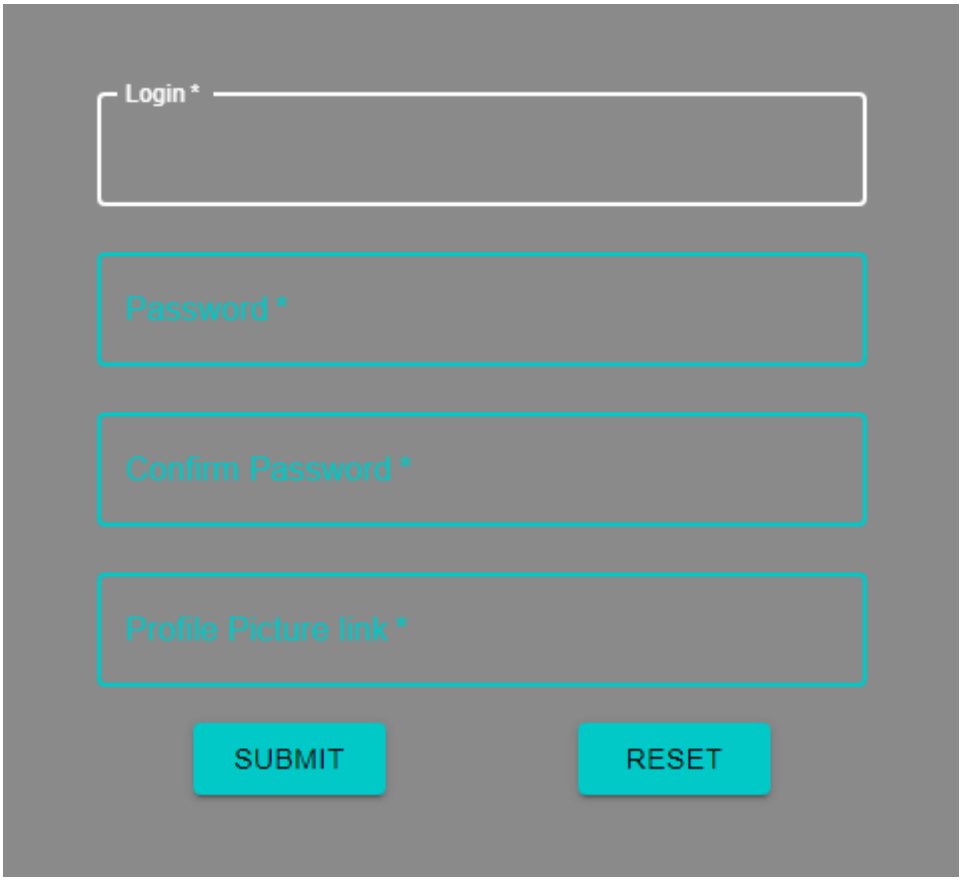

Rys. 4.2 - Ekran główny po zalogowaniu [opracowanie własne]

Ekran główny został zaprezentowany na rysunku 4.1. Składa się z tytułu aplikacji oraz dwóch przycisków. Przycisk z podpisem “Sign in” przenosi użytkownika do ekranu logowania, natomiast przycisk z napisem “Register” do ekranu rejestracji. Po zalogowaniu i ponownym powrocie do ekranu głównego użytkownikowi zostanie zaprezentowany przycisk zastępujący dwa poprzednie. Uwidoczniono to na rysunku 4.2. Przycisk ten, po kliknięciu przenosi użytkownika do ekranu czatu.

## 4.2. Ekran rejestracji




Rys. 4.3 - Ekran rejestracji [opracowanie własne]

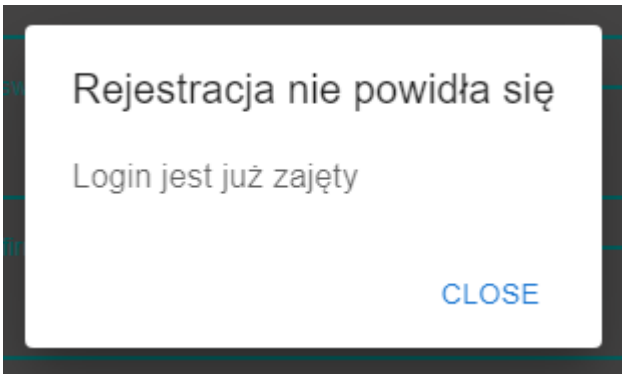



Rys. 4.4 - Formularz z ekranu rejestracji [opracowanie własne]

Na rysunkach został zaprezentowany ekran rejestracji (rys. 4.3) oraz znajdujący się w nim formularz (rys. 4.4). Aby dokonać rejestracji, użytkownik musi wypełnić wszystkie pola. Pola te oznaczają kolejno:

- login - nazwa użytkownika,
- password - hasło użytkownika,
- confrim password - pole edycji, które powinno zawierać potwierdzone hasło użytkownika,
- profile picture link - hiperłącze do zdjęcia profilowego.

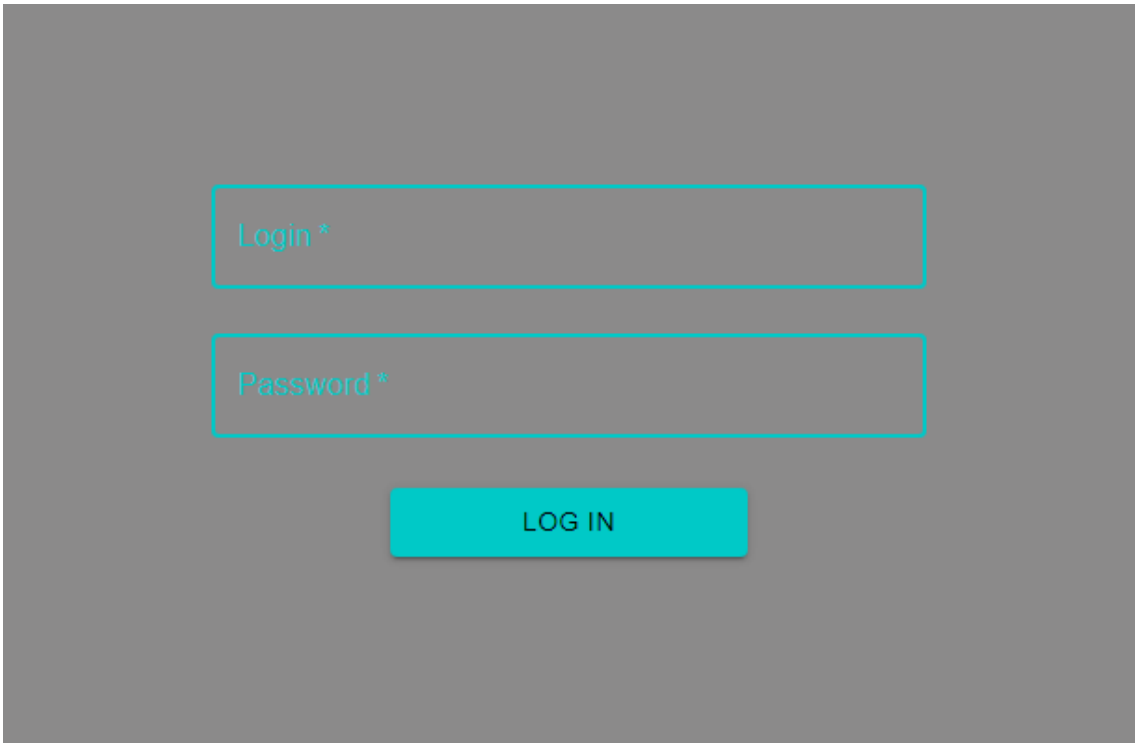
Dodatkowo formularz zawiera dwa przyciski. “Submit” do wysłania formularza oraz “Reset” do przywrócenia wszystkich pól do postaci domyślnej, niewypełnionej.




Rys. 4.5 - Dymek informacyjny [opracowanie własne]

W trakcie interakcji użytkownika z ekranem rejestracji, w zależności od rezultatu rejestracji zostanie wyświetlona wiadomość. Wiadomość zostanie zaprezentowana w postaci dymku (rys. 4.5).

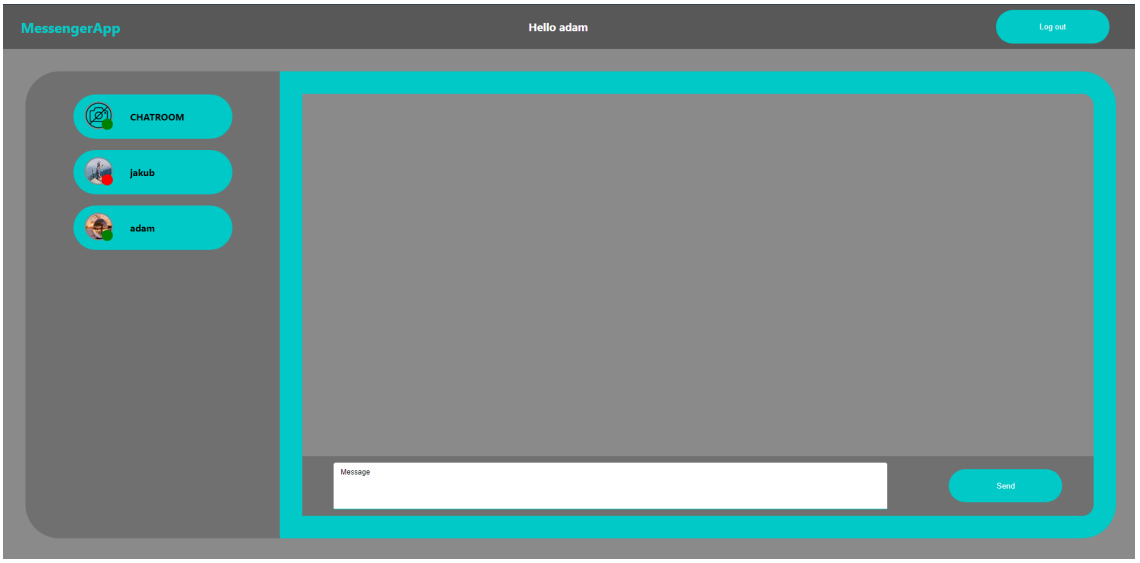
## 4.3. Ekran logowania




Rys. 4.6 - Formularz ekranu logowania [opracowanie własne]

Ekran logowania jest prawie identyczny, jak ekran rejestracji. Różni się on tylko funkcją oraz formularzem. W formularzu logowania występują dwa pola oraz przycisk. Pole “login” oznacza miejsce na wpisanie nazwy użytkownika, natomiast pole pod nazwą “password” na hasło. Przycisk służy do wysyłania formularza. Jeśli logowanie się powiedzie to użytkownik zostanie przeniesiony do ekranu czatu. Niepowodzenie operacji logowania spowoduje zaprezentowanie użytkownikowi dymku z wiadomością (rys. 4.5), jak w przypadku ekranu rejestracji.

## 4.4. Ekran czatu




Rys. 4.7 - Ekran czatu [opracowanie własne]

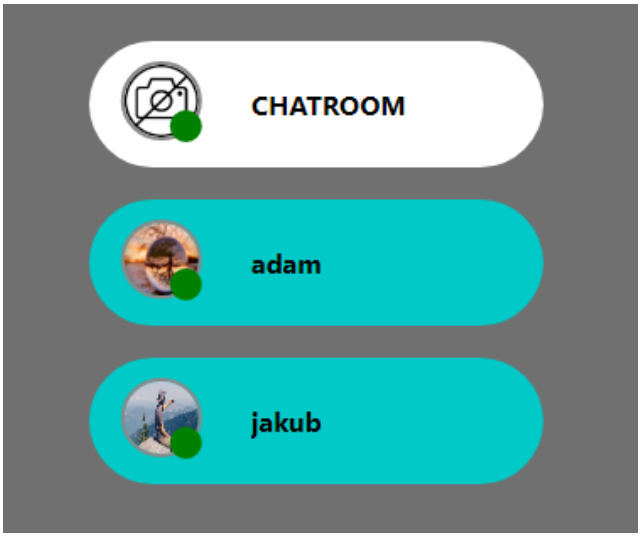
Ekran czatu składa się z kliku elementów (rys. 4.7). Na samej górze znajduję się pasek nagłówka. Nagłówek składa się z trzech elementów:

- nazwa aplikacji,
- napis informujący nas o tym, jaki użytkownik jest zalogowany,
- przycisk wylogowania.


Rys. 4.8 - Lista użytkowników znajdująca się na ekranie czatu [opracowanie własne]

W centrum ekranu znajduje się czat podzielony na dwie części. Po lewej stronie umiejscowiona została kolumna z listą zarejestrowanych użytkowników oraz czatem publicznym (rys. 4.8). Elementy przedstawiające użytkowników cechują się trzema elementami:

- obrazem profilowym podanym podczas rejestracji w postaci hiperłącza,
- statusem użytkownika, który przybiera barwę zieloną, gdy użytkownik jest zalogowany lub czerwoną gdy jest wylogowany,
- nazwy użytkownika.




Rys. 4.9 - Wizualna reprezentacja wybranego pokoju czatu [opracowanie własne]

Aby użytkownik mógł rozpocząć rozmowę na jednym z pokojów czatu dostępnych na liście użytkowników, musi on kliknąć na jeden z nich. Wybrana pozycja wyróżnia się na liście białym kolorem tła (rys. 4.9).


Rys. 4.10 - Pokój czatu [opracowanie własne]

Po wybraniu pokoju użytkownik może napisać wiadomość. Pokój czatu (rys. 4.10) charakteryzuje się trzema elementami:

- listą wiadomości,
- polem na nową wiadomość,
- przyciskiem do wysyłania wiadomości.

Wiadomości po wysłaniu zostaną zaprezentowane użytkownikowi na liście. Wiadomości umieszczone z lewej strony listy są to wiadomości wysłane przez innych użytkowników, natomiast te z prawej strony listy przez aktualnie zalogowanego użytkownika.


Rys. 4.11 - Stopka [opracowanie własne]

Na rysunku 4.11 zaprezentowano stopkę obecną na każdym ekranie aplikacji.

# 5. Testy

W ramach pracy zostały zrealizowanych sześć testów jednostkowych oraz jeden test integracyjny. Wykonanie testów zostało zaimplementowane z użyciem biblioteki JUnit oraz Mockito. Testy ograniczają się do funkcjonalności związanych z danymi o użytkownikach. Rezultaty testów jednostkowych zostaną umieszczone w formie ilustracji (rys 5.7) na końcu ich opisu.

## 5.1. Testy zapisu użytkownika

```
@Tag("User")
@Test
void canUserBeSaved() {

    //given
    String userName = "testUser";
    String password = "123";
    String testImageSrc = "testImageSrc";
    NewUserDto newUserDto = NewUserDto.builder()
            .userName(userName)
            .password(password)
            .imageSrc(testImageSrc)
            .build();
    UserEti userEti = UserEti.builder()
            .userName(userName)
            .status(UserEti.UserStatus.offline)
            .password(password)
            .imageSrc(testImageSrc)
            .build();

    //when
    Mockito.when(newUserMapper.convert(newUserDto)).thenReturn(userEti);
    Mockito.when(userRepository.findByUserName(newUserDto.getUserName())).thenReturn(userEti);
    Boolean result = chatService.saveUser(newUserDto);

    //then
    assertTrue(result);

}
```

Rys. 5.1 - Test zapisu użytkownika z prawidłowymi danymi [opracowanie własne]

Na rysunku 5.1 zaprezentowano test jednostkowy, który testuje, czy użytkownik został prawidłowo dodany do bazy danych. Metoda zapisująca dane użytkownika kończy się sukcesem po wypełnieniu trzech warunków:

- wybrana nazwa użytkownika nie jest zajęta,
- użytkownik został pomyślnie zapisany do bazy,
- użytkownik zapisany do bazy zgadza się z podanymi danymi.

Po spełnieniu wszystkich warunków zwracana jest wartość "**true**".

```
Jakub Krezolek
@Tag("User")
@Test
void canUserWithInvalidUserNameBeSaved() {

    //given
    String userName = null;
    String password = "123";
    String testImageSrc = "testImageSrc";
    NewUserDto newUserDto = NewUserDto.builder()
            .userName(userName)
            .password(password)
            .imageSrc(testImageSrc)
            .build();
    UserEti userEti = UserEti.builder()
            .userName(userName)
            .status(UserEti.UserStatus.offline)
            .password(password)
            .imageSrc(testImageSrc)
            .build();

    //when
    Mockito.when(newUserMapper.convert(newUserDto)).thenReturn(userEti);
    Mockito.when(userRepository.findByUserName(newUserDto.getUserName())).thenReturn( t: null);
    Boolean result = chatService.saveUser(newUserDto);

    //then
    assertFalse(result);

}
```

Rys. 5.2 - Test zapisu użytkownika z nieprawidłowymi danymi [opracowanie własne]

```java
@Tag("User")
@Test
void canExistingUserBeSaved() {

    //given
    String userName = "testUser";
    String password = "123";
    String testImageSrc = "testImageSrc";
    NewUserDto newUserDto = NewUserDto.builder()
            .userName(userName)
            .password(password)
            .imageSrc(testImageSrc)
            .build();
    UserEti userEti = UserEti.builder()
            .userName(userName)
            .status(UserEti.UserStatus.offline)
            .password(password)
            .imageSrc(testImageSrc)
            .build();
    UserDto userDto = UserDto.builder()
            .userName(userName)
            .status(UserEti.UserStatus.offline)
            .password(password)
            .imageSrc(testImageSrc)
            .build();

    //when
    Mockito.when(userMapper.convert(userEti)).thenReturn(userDto);
    Mockito.when(userRepository.findByUserName(newUserDto.getUserName())).thenReturn(userEti);
    Boolean result = chatService.saveUser(newUserDto);

    //then
    assertFalse(result);

}
```

Rys. 5.3 - Test zapisu użytkownika z zajętą nazwą użytkownika [opracowanie własne]

Na rysunkach 5.2 oraz 5.3 zilustrowano testy, które sprawdzają działanie zapisu użytkowników w przypadku, gdyby dane zostały wypełnione nieprawidłowo lub nazwa użytkownika była już zajęta.

## 5.2. Testy zmiany statusu użytkownika

```
Jakub Krezolek
@Tag("User")
@Test
void canUserStatusBeChanged() {

    //given
    String userName = "testUser";
    String password = "123";
    String testImageSrc = "testImageSrc";
    UserEti userEti = UserEti.builder()
            .userName(userName)
            .status(UserEti.UserStatus.offline)
            .password(password)
            .imageSrc(testImageSrc)
            .build();

    //when
    Mockito.when(userRepository.findByUserName(userName)).thenReturn(userEti);
    Boolean result = chatService.changeUserStatus(userName, status: "online");

    //then
    assertTrue(result);

}
```

Rys. 5.4 - Test zmiany statusu użytkownika [opracowanie własne]

Rysunek 5.4 prezentuje test weryfikujący poprawność metody, która zmienia status użytkownika. Zanim status użytkownika zostanie zmieniony, wspomniana metoda sprawdza, czy podany użytkownik istnieje w bazie danych. W przypadku powodzenia zwraca ona wartość "**true**".

```
Jakub Krezolek
@Tag("User")
@Test
void canUserStatusBeChangedForInvalidUser() {

    //given
    String userName = "testUser";

    //when
    Mockito.when(userRepository.findByUserName(userName)).thenReturn( t: null);
    Boolean result = chatService.changeUserStatus(userName,  status: "online");

    //then
    assertFalse(result);

}
```

Rys. 5.5 - Test zmian statusu nieistniejącego użytkownika [opracowanie własne]

Na rysunku 5.4 zilustrowano test, którego zadaniem jest sprawdzenie, czy można zmienić status nieistniejącego użytkownika.

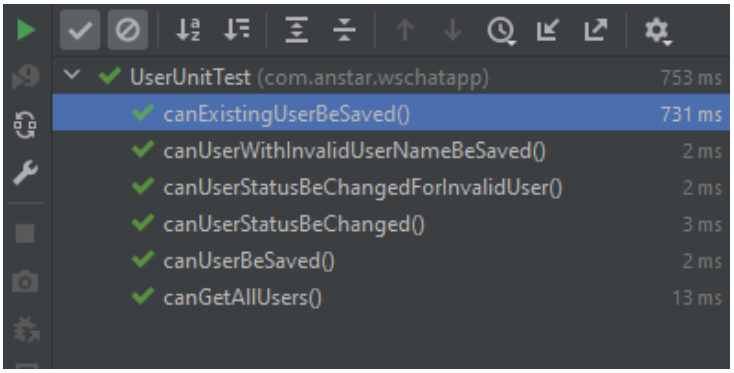
## 5.3. Test pobrania użytkowników

```
Jakub Krezolek
@Tag("User")
@Test
void canGetAllUsers() {

    //when
    chatService.findAllUsers();

    //then
    verify(userRepository).findAll();

}
```

Rys. 5.6 - Test sprawdzający poprawność pobierania danych o użytkownikach [opracowanie własne]

Rysunek 5.6 prezentuje test weryfikujący, czy możliwe jest pobranie danych o wszystkich użytkownikach.

## 5.4. Rezultaty testów jednostkowych




Rys. 5.7 - Rezultaty testów jednostkowych [opracowanie własne]

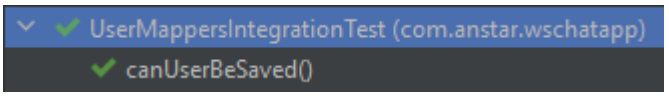
## 5.5. Test integracyjny

```
@Tag("User")
@Test
void canUserBeSaved() {

    //given
    String userName = "testUser";
    String password = "123";
    String testImageSrc = "testImageSrc";
    NewUserDto newUserDto = NewUserDto.builder()
            .userName(userName)
            .password(password)
            .imageSrc(testImageSrc)
            .build();
    UserEti userEti = UserEti.builder()
            .userName(userName)
            .status(UserEti.UserStatus.offline)
            .password(password)
            .imageSrc(testImageSrc)
            .build();

    ChatServiceImpl spyChatService = Mockito.spy(chatService);

    //when
    Mockito.doReturn( toBeReturned: Optional.empty()).when(spyChatService).findOneUserByUserName(userName);
    Mockito.when(userRepository.findByUserName(newUserDto.getUserName())).thenReturn(userEti);
    Boolean result = spyChatService.saveUser(newUserDto);

    //then
    assertTrue(result);

}
```

Rys. 5.8 - Test integracyjny [opracowanie własne]

Na rysunku 5.8 zaprezentowano test weryfikujący działanie metody zapisującej użytkowników do bazy danych wraz z mapperem. Różni się on tym od poprzednich testów, że w przypadku testów jednostkowych działanie mapera zostało symulowane. W

przypadku tego testu sprawdzana jest jego właściwa implementacja oraz integracja z metodą.




Rys. 5.9 - Rezultat testu integracyjnego [opracowanie własne]

# 6. Wnioski

Celem pracy było zaprojektowanie oraz zaimplementowanie komunikatora internetowego. Wszystkie założenia oraz cele pracy obecne w rozdziale pierwszym zostały zrealizowane. Aplikacja została zaprojektowana zgodnie z architekturą REST. Aplikacja serwerowa została zaimplementowana z użyciem języka Java oraz Frameworku Spring Boot. Posiada ona wszystkie konieczne funkcjonalności do poprawnej obsługi aplikacji internetowej oraz bazy danych. Aplikacja internetowa została zbudowana w oparciu o bibliotekę React. Jej implementacja pozwoliła mi na udoskonalenie umiejętności oraz naukę nowych technologii takich jak redux toolkit. Realizacja bazy danych została zaprojektowana z użyciem systemu do zarządzania relacyjnymi bazami danych PostgreSQL. Najważniejsze elementy aplikacji serwerowej zostały przetestowane z użyciem biblioteki JUnit. Podczas tworzenia aplikacji mogłem zgłębić wymagające fragmenty implementacji, co pozwoliło mi w pełni zrozumieć praktyczne zastosowanie protokołu WebSocket. Wiedza zdobyta w trakcie pracy oraz działania poznanych technologii pozwoliła mi na zrozumienie ich cech charakterystycznych oraz obszaru zastosowań.

Warto dodać, że aplikacja może zostać rozbudowana o dodatkowe funkcjonalności. Przykładem takich funkcjonalności są:

- system uwierzytelnienia i autoryzacji użytkowników z użyciem takich technologii, jak Spring Security,
- możliwość tworzenia czatów grupowych z dowolnej grupy użytkowników,
- możliwość dodawania załączników,
- panel edycji użytkownika oraz jego ustawień.

# 7. Literatura


1. https://pl.wikipedia.org/wiki/Java
2. https://pl.wikipedia.org/wiki/WebSocket
3. https://bulldogjob.pl/readme/prosto-o-websocket
4. https://stomp.github.io/
5. https://en.wikipedia.org/wiki/Object%E2%80%93relational_mapping
6. https://www.javatpoint.com/hibernate-tutorial
7. https://en.wikipedia.org/wiki/Hibernate_(framework)
8. https://www.javatpoint.com/jpa-introduction
9. Jon Brisbin, Mark Pollack, Michael Hunger, Oliver Gierke, Thomas Risberg. Spring Data. Wydawnictwa O'Reilly Media, Inc., Październik 2012.
10. Andrew Lombardi, WebSocket. Lightweight Client-Server Communications, Wydawnictwa O'Reilly Media, 2015.

# 8. Spis Rysunków