Kierunek: ***Informatyka***

2024/2025

Jakub Chamielec

PRACA INŻYNIERSKA

Projekt i implementacja forum dyskusyjnego

Promotor pracy:

mgr inż. Tomasz Gądek

Tarnów, *2025*

Spis treści

1. Wstęp

Ludzie od zawsze chętnie wymieniają się opiniami i doświadczeniami, szukając w nich inspiracji oraz wsparcia w podejmowaniu decyzji. Jednocześnie naturalna ciekawość sprawia, że chętnie poszerzają swoją wiedzę na różne tematy, wchodząc w interakcję z innymi osobami.

1.1. Cel pracy

Celem pracy inżynierskiej jest zaprojektowanie oraz implementacja aplikacji webowej pełniącej funkcję forum dyskusyjnego. System ma umożliwiać użytkownikom swobodną i efektywną wymianę informacji oraz opinii. W ramach projektu szczególny nacisk położono na zapewnienie responsywnego i intuicyjnego interfejsu, a także na wdrożenie kluczowych funkcjonalności wspierających interakcje pomiędzy użytkownikami, takich jak: tworzenie i zarządzanie wątkami, publikowanie odpowiedzi na posty oraz ocenianie treści. Dodatkowo administratorzy zostaną wyposażeni w narzędzia do zarządzania forum dyskusyjnym takie jak: możliwość usuwania nieodpowiednich treści, blokowania użytkowników na okres czasowy oraz trwałego usunięcia konta użytkownika. W celu zagwarantowania stabilności i poprawności działania aplikacji, projekt zostanie przetestowany za pomocą testów jednostkowych, integracyjnych oraz manualnych.

1.2. Zakres pracy

Aby efektywnie zrealizować założenia projektu, prace zostały podzielone na poszczególne etapy, obejmujące pełny cykl tworzenia aplikacji forum dyskusyjnego. W skład tych etapów wchodzą:

- **Analiza wymagań i planowanie** - przeprowadzenie szczegółowej analizy wymagań funkcjonalnych i niefunkcjonalnych aplikacji.
- **Projektowanie systemu** - opracowanie architektury systemu, stworzenie schematu bazy danych oraz prototypów interfejsu użytkownika.
- **Implementacja funkcji** - kodowanie kluczowych elementów aplikacji.
- **Zabezpieczenie aplikacji** - użycie tokenów JWT do autoryzacji użytkowników oraz zastosowanie middleware do zabezpieczenia ścieżek routingu.
- **Testowanie i optymalizacja** - przeprowadzenie testów jednostkowych, integracyjnych oraz manualnych w celu weryfikacji poprawności działania aplikacji.

2. Analiza biznesowa

Celem tego rozdziału jest określenie wymagań, które stanowią podstawę do zaprojektowania oraz wdrożenia aplikacji forum dyskusyjnego. Szczegółowa analiza tych potrzeb umożliwi opracowanie platformy, która będzie nie tylko intuicyjna i funkcjonalna, ale również spełni wysokie standardy w zakresie bezpieczeństwa, skalowalności oraz niezawodności. W kolejnych podrozdziałach opisano kluczowe funkcje i cechy systemu, które będą fundamentem w realizacji tego projektu.

2.1. Wymagania funkcjonalne

2.1.1. Rejestracja i logowanie

Aplikacja powinna zapewniać możliwość rejestracji użytkowników poprzez podanie adresu e-mail, unikalnej nazwy użytkownika, imienia, nazwiska oraz hasła. Proces rejestracji powinien być intuicyjny, aby umożliwić łatwe tworzenie kont nawet osobom, które są mniej obeznane z obsługą aplikacji internetowych. W celu zwiększenia bezpieczeństwa, system musi umożliwiać zmianę hasła oraz zapewniać walidację adresu e-mail, aby potwierdzić tożsamość użytkownika i zapobiec tworzeniu fałszywych kont. W przypadku zapomnienia hasła przez użytkownika, aplikacja powinna umożliwiać jego odzyskanie za pomocą linku, który będzie przesłany na zarejestrowany adres e-mail.

2.1.2. Tworzenie wątków i odpowiadanie na posty

Użytkownicy powinni mieć możliwość inicjowania nowych wątków dyskusyjnych poprzez podanie treści i przypisania jej do odpowiedniej kategorii tematycznej. Takie rozwiązanie ułatwi porządkowanie tematów na forum i będzie umożliwiać ich łatwe wyszukiwanie. Ponadto, aplikacja musi pozwalać użytkownikom na edytowanie oraz usuwanie wątków. W celu zapewnienia dynamicznej dyskusji użytkownicy powinni mieć również możliwość odpowiadania na istniejące publikacje, przy czym odpowiedzi muszą być przypisane do konkretnych postów. System powinien także pozwalać na edytowanie i usuwanie komentarzy przez użytkowników.

2.1.3. Moderacja treści

Administratorzy muszą dysponować narzędziami umożliwiającymi nadzorowanie treści publikowanych na forum, w tym edytowanie i usuwanie postów oraz komentarzy naruszających regulamin. System powinien oferować administratorom możliwość czasowego blokowania użytkowników łamiących zasady platformy.

2.1.4. Kategoryzacja treści

Każdy post powinien być przypisany do odpowiedniej kategorii tematycznej. Kategoryzacja ta ułatwi nawigację po treściach forum, a także wesprze ich moderowanie, co poprawi ogólną użyteczność systemu.

2.1.5. System oceniania postów

Aplikacja powinna wspierać ocenianie postów przez użytkowników, na przykład za pomocą systemu polubień. Posty, które zostały polubione przez konkretnego użytkownika, mogą wtedy zostać zapisane i można wrócić do nich w dowolnym momencie. Użytkownik powinien mieć możliwość przeglądania swojej listy ulubionych postów, co ułatwi mu odnalezienie treści, które najbardziej go interesowały.

2.2. Wymagania niefunkcjonalne

2.2.1. Bezpieczeństwo i autoryzacja

W celu zabezpieczenia dostępu do aplikacji, powinien zostać wdrożony system autoryzacji oparty na tokenie JWT (ang. *JSON Web Token*). Każdy zalogowany użytkownik będzie otrzymywał token, który dołączony do zapytań umożliwi autoryzowany dostęp do zasobów chronionych. Wdrożenie tego rozwiązania pozwoli zabezpieczyć ścieżki aplikacji przed nieupoważnionym dostępem, gwarantując, że jedynie uprawnieni użytkownicy będą mieli dostęp do funkcji systemu.

2.2.2. Zabezpieczenie ścieżek administratora

W systemie powinien zostać wdrożony middleware odpowiedzialny za kontrolę dostępu do ścieżek administratora. Middleware będzie przechwytywał żądanie, weryfikując czy użytkownik ma przypisaną rolę administratora w bazie danych. W przypadku braku autoryzacji dostęp do funkcji administratora powinien być automatycznie blokowany.

2.2.3. Usuwanie niezweryfikowanych kont

W celu ograniczenia liczby kont w bazie danych, system powinien automatycznie usuwać konta, które nie zostały zweryfikowane w określonym czasie. Po zakończeniu procesu rejestracji użytkownik będzie otrzymywać wiadomość e-mail z linkiem do weryfikacji, a brak jego aktywacji w wyznaczonym czasie skutkować będzie usunięciem konta z bazy danych.

2.2.4. Responsywność

Aplikacja powinna być zaprojektowana z myślą o responsywności, aby zapewnić optymalne doświadczenie użytkowników na różnych urządzeniach, takich jak smartfony, tablety i komputery stacjonarne. Responsywny design powinien dostosowywać układ i elementy interfejsu w zależności od rozmiaru ekranu, co umożliwi użytkownikom wygodne korzystanie z oprogramowania.

2.2.5. Wydajność i skalowalność

Aplikacja musi być zoptymalizowana pod kątem wydajności, aby umożliwić sprawne działanie przy jednoczesnej obsłudze wielu użytkowników. System powinien być także skalowalny, aby w przyszłości możliwe było łatwe rozszerzanie funkcji i obsługa rosnącej liczby użytkowników.

3. Stos technologiczny

Celem tego rozdziału jest przedstawienie komponentów technologicznych, które będą zastosowane do realizacji projektu forum dyskusyjnego. Opisane zostaną kluczowe elementy stosu technologicznego, które pozwolą na budowę aplikacji spełniającej wymagania funkcjonalne i niefunkcjonalne.

3.1. JavaScript

Język JavaScript powstał w 1995 roku jako technologia umożliwiająca dodawanie programów do stron internetowych w przeglądarce Netscape Navigator. Potem został przyjęty we wszystkich pozostałych przeglądarkach internetowych. Za pomocą tego języka możliwe stało się tworzenie aplikacji internetowych, z którymi użytkownik może bezpośrednio wchodzić w interakcje bez konieczności odświeżania całej strony po wykonaniu czynności. JavaScript używa się również na bardziej tradycyjnych stronach internetowych, w celu zapewnienia różnych rodzajów interaktywności i innych dodatków [1].

Właściwości języka JavaScript:

- **Wysoko poziomowość** - każdy język programowania wymaga zasobów komputerowych do prawidłowego funkcjonowania. W porównaniu do języków niskopoziomowych, w których programista musi samodzielnie zarządzać zasobami, JavaScript oferuje algorytm garbage-collection, który automatycznie zarządza pamięcią komputera, usuwając stare i nieużywane obiekty. Dzięki takiemu rozwiązaniu programiści mogą skupić się na implementacji aplikacji bez konieczności zarządzania pamięcią, co upraszcza proces tworzenia oprogramowania.
- **Dynamicznie typowany** - oznacza to, że zmienne w języku JavaScript nie muszą mieć z góry przypisanego typu. Typ zmiennej jest określany w czasie wykonywania programu na podstawie przypisanej jej wartości. Upraszcza to pisanie oprogramowania, ale może powodować błędy jeśli np. zmienna zmieni swój typ w sposób nieoczekiwany przez programistę.

- **Interpretowany** - oznacza to, że kod źródłowy jest analizowany i wykonywany linijka po linijce przez interpreter, a nie kompilowany do kodu maszynowego przed uruchomieniem programu. Najpopularniejszym silnikiem interpretującym kod JavaScript jest V8. Korzysta z niego wiele przeglądarek, a także środowisko uruchomieniowe Node.js. Dzięki takiemu podejściu, programista może łatwo testować i modyfikować kod w czasie rzeczywistym.
- **Wieloparadygmatowość** - w języku JavaScript można programować przy użyciu różnych podejść. Możliwe jest korzystanie zarówno z paradygmatu programowania imperatywnego, w którym programista precyzyjnie określa kroki konieczne do realizacji poszczególnych etapów programu lub paradygmatu programowania deklaratywnego, gdzie programista wskazuje co ma być wykonane, bez szczegółowego definiowania kroków operacyjnych [1].

Zdecydowano się użyć języka JavaScript do zaimplementowania aplikacji forum dyskusyjnego ze względu na jego uniwersalność oraz wszechstronne właściwości, które wspierają rozwój programów internetowych.

3.2. REST API

REST API (ang. *Representational State Transfer Application Programming Interface*) to styl architektury stosowany w tworzeniu interfejsów programistycznych dla systemów rozproszonych, który umożliwia efektywną komunikację pomiędzy aplikacjami za pomocą protokołu HTTP. Styl ten, opracowany przez Roya Fieldinga w 2000 roku, bazuje na zasadach mających na celu uproszczenie i standaryzację komunikacji między systemami, co czyni go jednym z najczęściej stosowanych standardów w aplikacjach webowych oraz mobilnych [8].

Kluczowe zasady REST API:

- **Bezstanowość** - każde żądanie przesłane od klienta do serwera jest niezależne i zawiera wszystkie niezbędne informacje potrzebne do jego przetworzenia. Serwer nie przechowuje żadnych informacji o stanie aplikacji między kolejnymi żądaniami klienta, co ułatwia skalowanie aplikacji i zwiększa jej wydajność.
- **Separacja klient-serwer** - klient i serwer są od siebie oddzielone, co oznacza, że każda ze stron może być rozwijana niezależnie. Serwer jest odpowiedzialny za przechowywanie i przetwarzanie danych oraz logiki biznesowej, natomiast klient obsługuje warstwę prezentacji [8].

3.3. Protokół HTTP

HTTP (ang. *Hypertext Transfer Protocol*) to podstawowy protokół komunikacyjny używany w sieci internetowej, umożliwiający przesyłanie danych między klientem, a serwerem. Jego głównym zadaniem jest zarządzanie wymianą informacji i zasobów, w sposób, który jest zrozumiały zarówno dla przeglądarek, jak i aplikacji sieciowych [10].

Metody protokołu HTTP:

- **GET** - służy do pobierania zasobu.
- **POST** - służy do utworzenia nowego zasobu.
- **PUT** - aktualizacja istniejącego zasobu.
- **DELETE** - usunięcie zasobu.

Kody statusu HTTP:

- **200** - żądanie zostało wykonane pomyślnie (zawiera dane zasobu).
- **201** - zasób został pomyślnie utworzony.
- **400** - niepoprawne żądanie.
- **401** - brak autoryzacji.
- **404** - serwer nie jest wstanie znaleźć zasobu pod podanym adresem URL.
- **500** - błąd po stronie serwera [10].

3.4. MERN Stack

W ramach projektu wykorzystany zostanie stos technologiczny MERN, składający się z technologii takich jak: MongoDB, Express.js, React, Node.js. Każdy z tych komponentów, w pełni opartych na JavaScripcie, wspiera zarówno frontendową, jak i backendową warstwę aplikacji, co znacząco ułatwi pracę nad spójnością i integracją kodu.

3.4.1. MongoDB

MongoDB jest jedną z najpopularniejszych baz danych NoSQL, która jest szczególnie dobrze dopasowana do aplikacji internetowych, które wymagają elastycznego modelu danych i dużej skalowalności. MongoDB przechowuje dane w formie dokumentów BSON (ang. *Binary JSON*), co pozwala na przechowywanie struktur danych w postaci, która jest zbliżona do struktury obiektów w JavaScript. Dzięki temu łatwo jest integrować bazę danych z aplikacjami napisanymi w tym języku [4].

Kluczowe cechy MongoDB:

- **Elastyczność schematu** - MongoDB nie wymaga sztywnego schematu, co daje programistom dużą elastyczność w przechowywaniu danych o zmiennej strukturze. Dzięki temu łatwiej jest dostosować strukturę bazy danych do zmieniających się wymagań aplikacji.
- **Skalowalność** - MongoDB wspiera rozproszone przechowywanie danych (ang. *sharding*), co w przypadku rozbudowanych aplikacji poprawia ich wydajność oraz umożliwia dodawanie nowych serwerów do istniejącego systemu bez konieczności inwestowania w drogi sprzęt. Dzięki tej funkcjonalności, aplikacje mogą rozrastać się w miarę potrzeb, zapewniając moc obliczeniową i przestrzeń na dane, nawet przy dużym wzroście liczby użytkowników lub ilości przechowywanych informacji.
- **Wydajność** - MongoDB jest zoptymalizowane pod kątem dużych ilości danych oraz operacji odczytu i zapisu, co czyni go idealnym wyborem do aplikacji, które muszą przetwarzać dane w czasie rzeczywistym, jak w przypadku forum dyskusyjnego.
- **Wsparcie dla agregacji** - MongoDB oferuje zaawansowane operacje agregacyjne, które pozwalają na wydajne przetwarzanie i analizowanie danych w bazie. Dzięki temu można szybko uzyskać potrzebne informacje [4].

3.4.2. Express.js

Express.js to popularny framework dla Node.js, stworzony do budowy aplikacji internetowych oraz API. Ze względu na prostotę i elastyczność, Express.js jest często wykorzystywany w aplikacjach o różnej skali, od prostych stron internetowych po złożone systemy serwerowe. Jego minimalistyczna architektura pozwala na szybkie tworzenie aplikacji, a jednocześnie zapewnia dużą elastyczność w zakresie dostosowywania funkcji do specyficznych potrzeb projektu [2].

Główne zalety Express.js:

- **Prostota i elastyczność** - Express.js jest minimalistyczny, co oznacza, że dostarcza tylko podstawowy zestaw funkcji potrzebnych do obsługi zapytań HTTP, zarządzania routingiem i przetwarzania odpowiedzi. To sprawia, że framework jest łatwy do nauki i idealnie nadaje się zarówno dla początkujących, jak i zaawansowanych programistów.
- **Modularność i middleware** - Express.js pozwala na łatwe dodawanie funkcji za pomocą tzw. middleware, co umożliwia rozszerzenie możliwości aplikacji. Middleware można stosować do obsługi uwierzytelniania, logowania, walidacji danych oraz wielu innych zadań, co zapewnia elastyczność i ułatwia zarządzanie kodem.
- **Wsparcie dla REST API** - Express.js jest idealny do tworzenia RESTful API dzięki prostemu zarządzaniu trasami (ang. *routing*) oraz obsłudze metod HTTP. Umożliwia to szybkie tworzenie interfejsów API, które mogą być łatwo zintegrowane z frontendem lub innymi systemami.
- **Wydajność i optymalizacja pod kątem Node.js** - Express.js wykorzystuje asynchroniczną architekturę Node.js, co pozwala na obsługę dużej liczby jednoczesnych zapytań. Dzięki temu aplikacje oparte na Express.js mogą być skalowalne i wydajne, co jest szczególnie istotne w przypadku aplikacji wymagających natychmiastowej reakcji, jak np. aplikacje komunikacyjne [2].

3.4.3. React

React to nowoczesna biblioteka JavaScript, opracowana przez Facebooka, umożliwiająca tworzenie interaktywnych i dynamicznych interfejsów użytkownika. Dzięki architekturze opartej na komponentach, React pozwala na budowanie aplikacji internetowych o wysokiej wydajności, które są łatwe do skalowania i rozbudowy. Ze względu na swoją elastyczność React jest szeroko stosowany w aplikacjach o różnym stopniu złożoności, od prostych stron po rozbudowane aplikacje [3].

Główne zalety React:

- **Modularność i ponowne wykorzystanie kodu** - React opiera się na komponentach jako podstawowych jednostkach budowy. Komponenty mogą być wielokrotnie używane i osadzane w różnych częściach aplikacji, co sprzyja modularności kodu i ułatwia jego utrzymanie. Dzięki temu można tworzyć komponenty, które są łatwo zintegrowane z pozostałymi elementami aplikacji.
- **Reaktywność i szybka aktualizacja interfejsu** - React efektywnie zarządza aktualizacjami interfejsu za pomocą wirtualnego DOM, co minimalizuje koszt ponownego renderowania widoku. Dzięki temu aplikacje działają szybko i płynnie, co znacząco poprawia doświadczenia użytkowników, szczególnie w dynamicznych interfejsach z dużą liczbą interakcji.
- **Jednokierunkowy przepływ danych** - React stosuje jednokierunkowy przepływ danych (ang. *unidirectional data flow*), co ułatwia zarządzanie stanem aplikacji oraz debugowanie. Pozwala to na lepszą przewidywalność działania aplikacji i sprawia, że jest ona łatwiejsza w utrzymaniu, zwłaszcza w przypadku rozbudowanych interfejsów użytkownika [3].

3.4.4. Node.js

Node.js to środowisko uruchomieniowe JavaScript, które umożliwia wykonywanie kodu po stronie serwera, co czyni je popularnym wyborem do budowy aplikacji internetowych oraz API. Dzięki architekturze asynchronicznej opartej na zdarzeniach, Node.js jest wyjątkowo wydajny i doskonale nadaje się do obsługi aplikacji, które wymagają zarządzania dużą ilością jednoczesnych połączeń. Node.js wyróżnia się elastycznością i szerokim wsparciem społeczności, co sprawia, że znajduje zastosowanie zarówno w małych projektach, jak i w dużych, skalowalnych aplikacjach [2].

Główne zalety Node.js:

- **Asynchroniczna architektura i obsługa wielu połączeń** - Node.js jest oparty na architekturze asynchronicznej, co pozwala mu obsługiwać wiele zadań równocześnie bez blokowania zasobów serwera. Ta cecha sprawia, że Node.js idealnie nadaje się do aplikacji o wysokiej liczbie równoczesnych połączeń, takich jak systemy obsługujące transakcje lub komunikację w czasie rzeczywistym.
- **Bogata biblioteka modułów NPM** - Node.js jest zintegrowany z NPM (ang. *Node Package Manager*), który oferuje szeroką gamę bibliotek i modułów. Moduły te przyspieszają rozwój aplikacji i pozwalają korzystać z gotowych rozwiązań do takich zadań jak autoryzacja, obsługa sesji użytkowników, walidacja danych i operacje na bazach danych.
- **Wydajność i skalowalność** - Node.js wykorzystuje silnik V8, który tłumaczy JavaScript na kod maszynowy, co przyczynia się do jego wyjątkowej wydajności. Dzięki temu aplikacje oparte na Node.js są szybkie i łatwo skalowalne, co sprawdza się w dynamicznych projektach, które muszą obsługiwać duże ilości danych i użytkowników [2].

3.5. Biblioteki do zabezpieczeń w MERN Stack

Bezpieczeństwo jest jednym z kluczowych aspektów każdej aplikacji webowej, szczególnie tych, które przetwarzają wrażliwe dane użytkowników. Zastosowanie odpowiednich technik i narzędzi zabezpieczających jest niezbędne, aby chronić aplikację oraz informacje o użytkownikach. Poniżej przedstawiono technologie, które będą wykorzystywane do zabezpieczenia aplikacji forum dyskusyjnego.

3.5.1. Bcrypt

Bcrypt to biblioteka, której głównym celem jest ochrona poufnych danych użytkowników przed nieuprawnionym odczytem. W przeciwieństwie do prostego szyfrowania lub przechowywania haseł w formie zwykłego tekstu, Bcrypt dokonuje transformacji hasła użytkownika w złożony ciąg znaków (ang. *hash*), który jest praktycznie niemożliwy do odtworzenia do pierwotnej postaci [5].

Sposób działania Bcrypt:

- **Generowanie soli** - Bcrypt generuje losowy ciąg znaków, który jest dodawany do hasła przed jego hashowaniem (ang. *salt*).
- **Hashowanie hasła** - po dodaniu soli hasło przechodzi przez serię iteracji, które zwiększają jego bezpieczeństwo, a wynik tego procesu jest hashem.
- **Weryfikacja hasła** - podczas logowania hasło wprowadzone przez użytkownika jest hashowane przy użyciu tej samej soli i porównywane z oryginalnym hashem. Jeśli hash z hasła wprowadzonego przy logowaniu zgadza się z tym zapisanym w bazie, hasło jest uznawane za poprawne [5].

3.5.2. Cookie-parser

Cookie parser to biblioteka dla Node.js, która służy do obsługi i parsowania ciasteczek HTTP (ang. *cookies*). Ciasteczka są często wykorzystywane w aplikacjach webowych do przechowywania informacji sesyjnych, takich jak dane logowania czy preferencje użytkownika. Dzięki cookie-parser możliwe jest szybkie i bezproblemowe zarządzanie danymi w ciasteczkach, co znacząco ułatwia implementację funkcji związanych z sesjami użytkowników oraz autoryzacją [7].

Cookie parser umożliwia:

- **Automatyczne parsowanie ciasteczek** - Cookie-parser przetwarza ciasteczka z nagłówków HTTP i udostępnia je w formie łatwego do obsługi obiektu JavaScript, co pozwala na szybkie odczytywanie informacji zapisanych przez przeglądarkę.
- **Lepsza kontrola nad sesjami użytkownika** - Cookie-parser umożliwia łatwą implementację ciasteczek sesyjnych, co pozwala na przechowywanie krótkotrwałych informacji w przeglądarce użytkownika [7].

3.5.3. JSON Web Tokens

Json Web Tokens to biblioteka dla Node.js która umożliwia tworzenie i weryfikację tokenów JWT (ang. *JSON Web Tokens*). JWT to otwarty standard, który definiuje sposób bezpiecznego przesyłania informacji między stronami w formie obiektu JSON. Tokeny te są szeroko stosowane w aplikacjach webowych, szczególnie w systemach autoryzacji i autentykacji użytkowników [6].

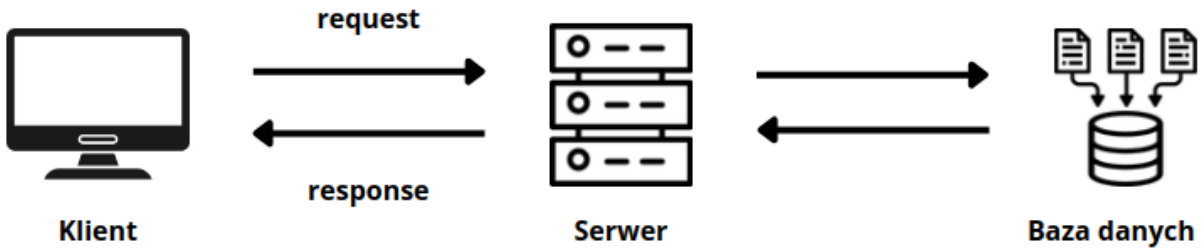
Zastosowanie Json Web Tokens:

- **Autoryzacja** - po zalogowaniu się użytkownika, serwer generuje token JWT, który jest następnie wysyłany do klienta (np. przeglądarki) i przechowywany w lokalnym magazynie (np. w pamięci lub ciasteczkach). Przy każdym kolejnym żądaniu do serwera klient dołącza omawiany token, co pozwala na identyfikację użytkownika oraz autoryzację dostępu do zasobów wymagających uprawnień.
- **Bezstanowość** - JWT to tokeny bezstanowe, co oznacza, że serwer nie musi przechowywać informacji o sesji użytkownika. Wszystkie dane, które są niezbędne do weryfikacji tożsamości i uprawnień użytkownika, są zawarte w tokenie. Dzięki temu aplikacja może skalować się w sposób bardziej efektywny, ponieważ nie wymaga przechowywania stanu na serwerze.
- **Bezpieczeństwo** - JWT mogą być podpisywane za pomocą algorytmów kryptograficznych, co zapewnia integralność danych zawartych w tokenie [6].

4. Implementacja systemu

Celem tego rozdziału jest przedstawienie wykonania kluczowych funkcji aplikacji forum dyskusyjnego, uwzględniając zarówno implementację podstawowych modułów, jak i zaawansowanych mechanizmów zarządzania treścią oraz użytkownikami. Do realizacji zadania wykorzystano diagram przypadków użycia, który ilustruje relacje pomiędzy aktorami a poszczególnymi funkcjami systemu.

4.1. Architektura aplikacji



Rys. 4.1 Architektura systemu [opracowanie własne]

W aplikacji forum dyskusyjnego architektura klient-serwer dzieli aplikację na dwie części: klienta i serwer. Klient odpowiada za interfejs użytkownika i wysyłanie żądań do serwera. Serwer przetwarza żądania, wykonuje logikę aplikacji i komunikuje się z bazą danych, a następnie odsyła odpowiedź do klienta.

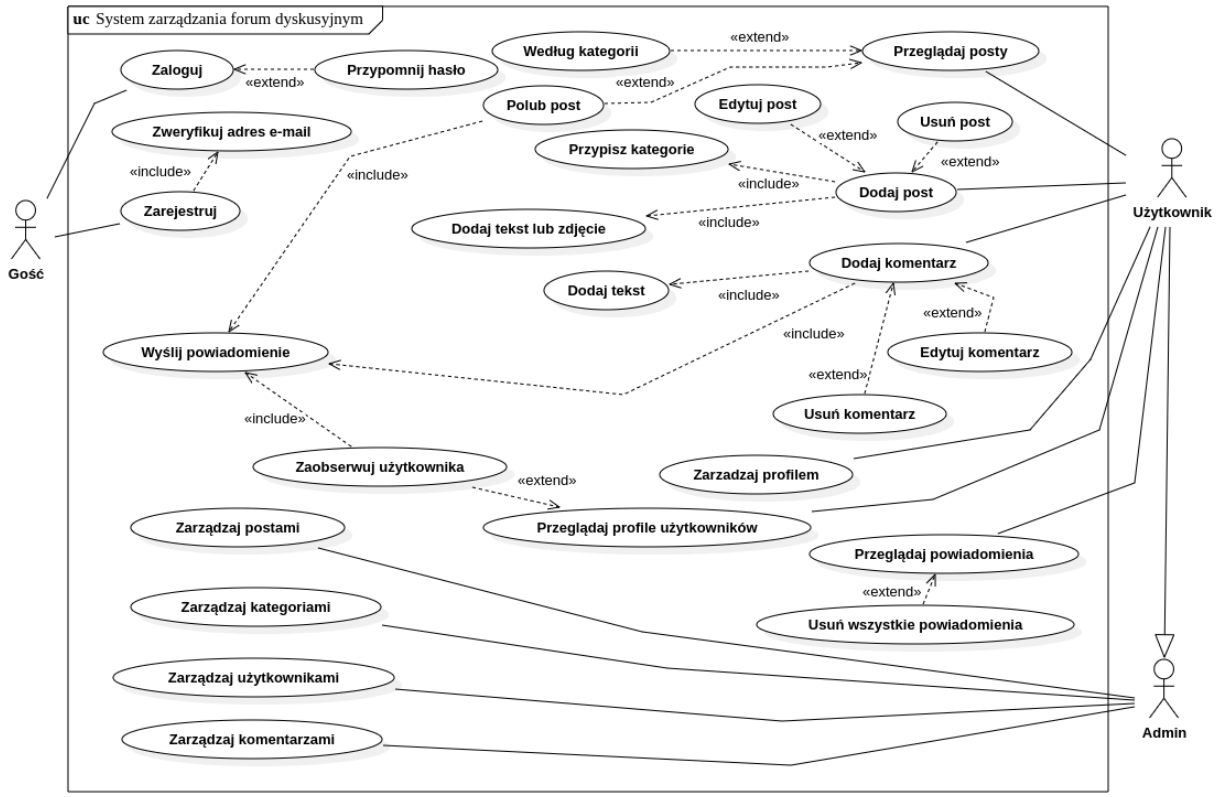
Zalety:

- **Centralne zarządzanie treścią** - serwer przechowuje dane o użytkownikach, wątkach i postach.
- **Elastyczność** - użytkownicy mogą korzystać z forum na różnych urządzeniach.
- **Bezpieczeństwo** - dane są przetwarzane i przechowywane na serwerze, a nie na urządzeniu użytkownika.
- **Rozdzielenie odpowiedzialności** - klient wyświetla dane, a serwer zarządza logiką aplikacji.

Wady:

- **Przeciążenie serwera** - duża liczba użytkowników może spowolnić działanie forum.
- **Awaria serwera** - brak dostępu do forum w przypadku awarii serwera.

4.2. Diagram przypadków użycia



Rys. 4.2 Diagram przypadków użycia [opracowanie własne]

Gość - osoba odwiedzająca forum dyskusyjne bez autoryzacji. Może:

- Zarejestrować się, aby utworzyć konto.
- Zalogować się, jeśli posiada już konto.
- Przypomnieć hasło w przypadku jego utraty.
- Zweryfikować adres e-mail w procesie rejestracji.

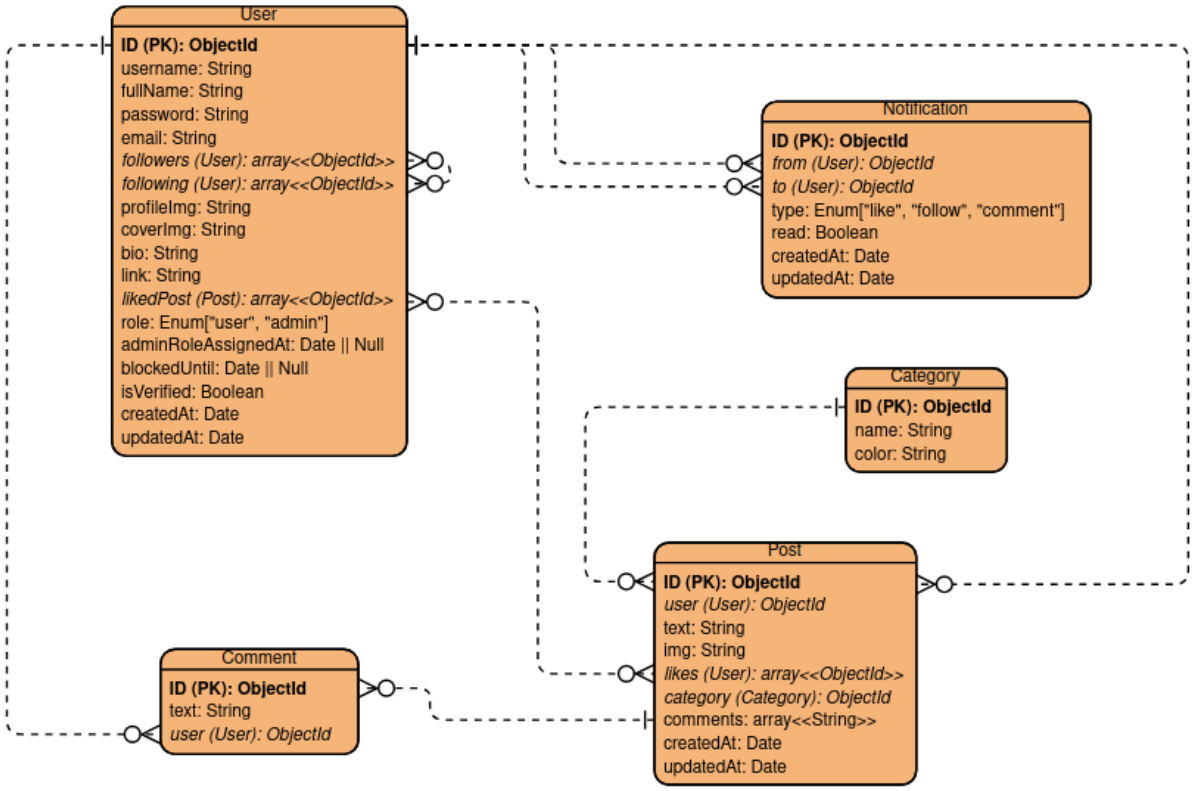
Użytkownik - osoba zalogowana na swoje konto. Ma dostęp do podstawowych funkcjonalności forum takich jak:

- Tworzenie postów.
- Przeglądanie postów według kategorii.
- Edytowanie lub usuwanie swoich postów.
- Polubienie postów innych użytkowników.
- Dodawanie komentarzy, ich usuwanie oraz edycja.
- Obserwowanie innych użytkowników.
- Zarządzanie własnym profilem.
- Przeglądanie powiadomień oraz ich usuwanie.

Admin - użytkownik o największych uprawnieniach, który oprócz standardowych funkcji dostępnych dla zwykłego użytkownika, ma dodatkowe możliwości zarządzania forum takie jak:

- Zarządzanie postami - administrator może edytować i usuwać dowolny post użytkownika.
- Zarządzanie kategoriami - obejmuje dodawanie nowych kategorii oraz ich edycję i usuwanie.
- Zarządzanie użytkownikami - obejmuje blokowanie kont, przypisywanie ról administratora oraz usuwanie użytkowników.
- Zarządzanie komentarzami - administrator może edytować i usuwać dowolny komentarz użytkownika.

4.3. Diagram ERD



Rys. 4.3 Diagram relacji i encji [opracowanie własne]

User (Użytkownik):

- Reprezentuje kolekcję użytkowników. Przechowuje informacje o danych osobowych użytkownika (**username, fullName**, **email)** oraz o profilu (**profileImg, bio, link, coverImg**).

● Użytkownik może obserwować innych użytkowników i być obserwowany. Relacje te są zapisane w polach **followers** i **following**, które przechowują referencje do innych użytkowników.

● Użytkownik może mieć przypisaną rolę, być zweryfikowany lub zablokowany do określonej daty.

Post:

● Reprezentuje kolekcję postów tworzonych przez użytkowników. Każdy post zawiera treść, opcjonalnie dodany obraz, kategorię oraz referencje do użytkownika, który go stworzył.

● Posty mogą być polubione przez innych użytkowników. Informacje te są przechowywane w polu **likes** jako tablica referencji do użytkowników.

● Każdy post ma powiązane komentarze, które są przechowywane w osobnej kolekcji **Comment.**

Comment (Komentarz):

● Kolekcja przechowuje komentarze przypisane do postów. Każdy komentarz zawiera treść, referencje do użytkownika, który go stworzył oraz logiczne powiązanie z postem, do którego został dodany.

Category (Kategoria):

● Reprezentuje kolekcję, kategorii przypisanych do postów. Przechowuje nazwę i kolor kategorii.

Notifications (Powiadomienia):

● Kolekcja przechowuje informacje o powiadomieniach generowanych w systemie. Każde powiadomienie zawiera referencje do nadawcy, odbiorcy oraz typ powiadomienia (**like**, **comment** lub **follow)**.

● Powiadomienia mają również pole **read**, które wskazuje, czy zostały one odczytane.

Rysunek 4.3 przedstawia logiczny model danych dla bazy MongoDB, która jest nierelacyjna. Relacje zaprezentowane na schemacie są relacjami logicznymi, a nie fizycznymi, co oznacza, że są realizowane za pomocą referencji (**ObjectId**) między dokumentami.

4.4. Ścieżki routingu

W tym podrozdziale zostaną zaprezentowane ścieżki routingu, które pełnią kluczową rolę w działaniu aplikacji.

4.4.1. Ścieżka autoryzacji

```
router.post("/signup", signup);
router.post("/login", login);
router.post("/logout", logout);
router.get("/me", protectRoute, getMe);
router.post("/verify-email", verifyEmail);
router.post("/request-password-reset", requestPasswordReset);
router.post("/reset-password/:token", resetPassword);
```

Rys. 4.4 Ścieżka autoryzacji [opracowanie własne]

Tabela 4.1 Ścieżka autoryzacji [opracowanie własne]

Metoda	Ścieżka API	Opis
POST	/signup	Obsługuje rejestrację nowych użytkowników.
POST	/login	Obsługuje logowanie użytkowników.
POST	/logout	Obsługuje wylogowywanie użytkowników.
GET	/me	Zwraca informacje o zalogowanym użytkowniku.
POST	/verify-email	Obsługuje weryfikację adresu e-mail użytkownika.
POST	/request-password-reset	Obsługuje żądania resetowania hasła.
POST	/reset-password/:token	Sprawdza czy token do resetowania hasła jest poprawny.

4.4.2. Ścieżka kategorii

```
router.get("/", protectRoute, getCategories);
router.post("/createCategory", protectRoute, isAdmin, createCategory);
router.put("/editCategory/:categoryId", protectRoute, isAdmin, editCategory);
```

Rys. 4.5 Ścieżka kategorii [opracowanie własne]

Tabela 4.2 Ścieżka kategorii [opracowanie własne]

Metoda	Ścieżka API	Opis
GET	/categories	Zwraca listę kategorii.
POST	/createCategory	Obsługuje tworzenie nowej kategorii, ścieżka dostępna tylko dla zalogowanych użytkowników z uprawnieniami administratora.
PUT	/editCategory/:categoryId	Obsługuje edycję istniejących kategorii, ścieżka dostępna tylko dla zalogowanych użytkowników z uprawnieniami administratora.

4.4.3. Ścieżka administratora

```
router.put("/assignOrRemoveRole/:id", protectRoute, isAdmin, assignOrRemoveRole);
router.put("/blockOrUnblockUser/:userId", protectRoute, isAdmin, blockOrUnblockUser);
router.delete("/users/:userId", protectRoute, isAdmin, deleteUser);
```

Rys. 4.6 Ścieżka administratora [opracowanie własne]

Tabela 4.3 Ścieżka administratora [opracowanie własne]

Metoda	Ścieżka API	Opis
PUT	/assignOrRemoveRole/:id	Obsługuje przypisywanie lub usuwanie roli admina. Admin z wcześniej nadanymi uprawnieniami nie może mieć zmienionej roli przez admina z później nadanymi uprawnieniami.
PUT	/blockOrUnblockUser/:userId	Obsługuje blokowanie użytkowników na określony czas.
DELETE	/users/:userId	Obsługuje usuwanie użytkowników.

4.4.4. Ścieżka powiadomień

```
router.get("/", protectRoute, getNotifications);
router.delete("/", protectRoute, deleteNotifications);
```

Rys. 4.7 Ścieżka powiadomień [opracowanie własne]

Tabela 4.4 Ścieżka powiadomień [opracowanie własne]

Metoda	Ścieżka API	Opis
GET	/notification	Zwraca listę powiadomień.
DELETE	/notification	Usuwa wszystkie powiadomienia.

4.4.5. Ścieżka użytkownika

```
router.get("/profile/:username", protectRoute, getUserProfile);
router.get("/suggested", protectRoute, getSuggestedUsers);
router.post("/follow/:id", protectRoute, followUnfollowUser);
router.post("/update", protectRoute, updateUser);
router.get("/search", protectRoute, searchUsers);
```

Rys. 4.8 Ścieżka użytkownika [opracowanie własne]

Tabela 4.5 Ścieżka użytkownika [opracowanie własne]

Metoda	Ścieżka API	Opis
GET	/profile/:username	Zwraca profil użytkownika o podanej nazwie.
GET	/suggested	Zwraca listę losowych użytkowników z bazy danych.
POST	/follow/:id	Obsługuje obserwowanie lub zaprzestanie obserwowania użytkownika o podanym identyfikatorze.
POST	/update	Obsługuje aktualizacje danych zalogowanego użytkownika.
GET	/search	Umożliwia wyszukiwanie użytkowników.

4.4.6. Ścieżka postu

```
router.get("/user/:username/:categoryId?", protectRoute, getPostsByUser);
router.get("/likes/:userId/:categoryId?", protectRoute, getLikedPosts);
router.get("/following/:categoryId?", protectRoute, getFollowingPosts);
router.get("/all/:categoryId?", protectRoute, getAllPosts);
router.get("/:categoryId", protectRoute, getPostsByCategory);
router.post("/create", protectRoute, createPost);
router.post("/like/:id", protectRoute, likeUnlikePost);
router.post("/comment/:id", protectRoute, commentOnPost);
router.delete("/:id", protectRoute, deletePost);
router.delete("/comment/:commentId", protectRoute, deleteComment);
router.put("/:id", protectRoute, updatePost);
router.put("/comment/:commentId", protectRoute, updateComment);
```

Rys. 4.9 Ścieżka postu [opracowanie własne]

Tabela 4.6 Ścieżka postu [opracowanie własne]

Metoda	Ścieżka API	Opis
GET	/user/:username/:categoryId?	Zwraca posty użytkownika według nazwy, opcjonalne filtrowanie według kategorii.
GET	/likes/:userId/:categoryId?	Zwraca posty polubione przez użytkownika o podanym identyfikatorze, opcjonalne filtrowanie według kategorii.
GET	/following/:categoryId?	Zwraca posty obserwowanych użytkowników, opcjonalne filtrowanie według kategorii.
GET	/all/:categoryId?	Zwraca wszystkie posty, opcjonalne filtrowanie postów według kategorii.
GET	/:categoryId	Zwraca posty z określonej kategorii.
POST	/create	Obsługuje tworzenie nowego wątku.
POST	/like/:id	Obsługuje polubienie lub usunięcie polubienia posta.

POST	/comment/:id	Obsługuje dodawanie komentarza o podanym identyfikatorze.
DELETE	/:id	Obsługuje usuwanie wątku o podanym identyfikatorze.
DELETE	/comment/:commentId	Obsługuje usuwanie komentarza o podanym identyfikatorze.
PUT	/:id	Obsługuje aktualizacje postu o podanym identyfikatorze.
PUT	/comment/:commentId	Obsługuje aktualizacje komentarza o podanym identyfikatorze.

4.5. Analiza wybranych fragmentów kodu

W niniejszym podrozdziale dokonano szczegółowej analizy wybranych fragmentów kodu źródłowego, które ilustrują kluczowe aspekty funkcjonalności omawianego systemu.

4.5.1. Sortowanie postów według kategorii

```
export const getPostsByCategory = async (req, res) => {
  try {
    const { categoryId } = req.params;
    const posts = await Post.find({ category: categoryId })
      .sort({ createdAt: -1 })
      .populate({ path: "user", select: "-password" })
      .populate({ path: "comments.user", select: "-password" })
      .populate({ path: "category", select: "name color" });

    if (posts.length === 0) {
      return res.status(200).json([]);
    }

    res.status(200).json(posts);
  } catch (error) {
    console.log("Error fetching posts by category: ", error);
    res.status(500).json({ error: "Internal Server Error" });
  }
};
```

Rys. 4.10 Funkcja sortująca posty [opracowanie własne]

Funkcja **getPostByCategory** to asynchroniczna metoda obsługująca sortowanie postów według kategorii. Pobiera **categoryId** z parametrów żądania, a następnie wyszukuje w bazie danych posty przypisane do tej kategorii, sortując je malejąco według daty utworzenia. Przy użyciu metody **population** dołącza dodatkowe dane o użytkownikach, komentarzach i kategorii. Jeśli w wybranej kategorii nie znaleziono żadnych postów, zwracana jest pusta tablica z kodem 200. W przypadku wystąpienia błędu funkcja zwraca odpowiedź z kodem 500 i komunikatem "Internal Server Error".

4.5.2. Dodawanie postu

```
export const createPost = async (req, res) => {
  try {
    const { text, categoryId } = req.body;
    let { img } = req.body;
    const userId = req.user._id.toString();

    const user = await User.findById(userId);
    if (!user) {
      return res.status(404).json({ error: "User not found" });
    }
    if (!text && !img) {
      return res.status(400).json({ error: "Text or image is required" });
    }

    if (!categoryId) {
      return res.status(400).json({ error: "Category is required" });
    }

    if (categoryId) {
      const category = await Category.findById(categoryId);
      if (!category) {
        return res.status(404).json({ error: "Category not found" });
      }
    }

    if (img) {
      const uploadedResponse = await cloudinary.uploader.upload(img);
      img = uploadedResponse.secure_url;
    }

    const newPost = new Post({
      user: userId,
      text,
      img,
      category: categoryId,
    });
```

Rys. 4.11 Funkcja do dodawania postu [opracowanie własne]

Funkcja **createPost** jest asynchroniczną metodą obsługującą tworzenie nowych postów. Pobiera **text**, **categoryId** oraz opcjonalnie **img** z ciała żądania, a następnie weryfikuje, czy użytkownik istnieje w bazie danych. Jeśli użytkownik nie zostanie znaleziony, zwracana jest odpowiedź z kodem statusu 404 i komunikatem o błędzie. W przypadku braku zarówno tekstu, jak i obrazu lub niepodania **categoryId**, serwer odpowiada kodem 400 z odpowiednim komunikatem. Jeżeli w żądaniu znajduje się obraz to jest on przesyłany do Cloudinary, a URL obrazu zostaje zaktualizowany. Na końcu tworzony jest nowy obiekt posta zawierający dane użytkownika, tekst, obraz (jeśli podano) oraz kategorię, który jest zapisywany w bazie danych. W odpowiedzi zwracany jest kod 201 wraz z nowo utworzonym postem. W przypadku błędu, zwracana jest odpowiedź z kodem statusu 500 i komunikatem "Internal Server Error".

4.5.3. Rejestracja użytkownika

```
export const signup = async (req, res) => {
  try {
    const { fullName, username, email, password } = req.body;

    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    if (!emailRegex.test(email)) {
      return res.status(400).json({ error: "Invalid email format" });
    }

    const existingUser = await User.findOne({ username });
    if (existingUser) {
      return res.status(400).json({ error: "Username is already taken" });
    }

    const existingEmail = await User.findOne({ email });
    if (existingEmail) {
      return res.status(400).json({ error: "Email is already taken" });
    }

    if (password.length < 6) {
      return res
        .status(400)
        .json({ error: "Password must be at least 6 characters long" });
    }

    const salt = await bcrypt.genSalt(10);
    const hashedPassword = await bcrypt.hash(password, salt);

    const verificationCode = crypto.randomBytes(20).toString("hex");

    const newUser = new User({
      fullName,
      username,
      email,
      password: hashedPassword,
      role: "user",
      verificationCode,
    });

    await sendVerificationEmail(email, verificationCode);

    await newUser.save();

    res.status(201).json({
      message:
        "User registered successfully. Please check your email to verify your account.",
    });
  } catch (error) {
    console.error("Error in signup:", error);
    res.status(500).json({ error: "Internal Server Error" });
  }
};
```

Rys. 4.12 Funkcja do rejestracji użytkowników [opracowanie własne]

Funkcja **signup** jest asynchroniczną metodą obsługującą rejestrację nowych użytkowników. Pobiera **fullName, username**, **email** i **password** z ciała żądania, a następnie sprawdza, czy podany adres email jest w prawidłowym formacie za pomocą wyrażenia regularnego. Jeśli format jest nieprawidłowy, zwraca odpowiedź z kodem 400 i komunikatem o błędzie. Funkcja weryfikuje, czy nazwa użytkownika lub adres email są już zarejestrowane w bazie danych. W przypadku ich duplikacji zwraca odpowiedź z kodem 400 i odpowiednim komunikatem o błędzie. Następnie sprawdza, czy **password** ma co najmniej 6 znaków. Jeśli hasło jest za krótkie, zwraca odpowiedź z kodem 400 i komunikatem o błędzie. Jeśli wszystkie dane są poprawne, funkcja generuje sól za pomocą funkcji **bycrypt.genSalt** i hashuje **password** za pomocą **bycrypt.hash**. Generuje również losowy kod weryfikacyjny za pomocą **crypto.randomBytes**. Następnie tworzy nowy obiekt użytkownika z podanymi danymi, zaszyfrowanym hasłem i kodem weryfikacyjnym. Następnie wysyła email z kodem weryfikacyjnym do użytkownika za pomocą funkcji **sendVerificationEmail** i zapisuje nowego użytkownika w bazie danych. Jeśli operacja zakończy się sukcesem, funkcja zwraca odpowiedź z kodem 201 i komunikatem o sukcesie, informującym, że użytkownik został zarejestrowany i musi zweryfikować swój email. W przypadku błędu zwraca odpowiedź z kodem 500 oraz komunikatem "Internal Server Error".

4.5.4. Wylogowywanie użytkownika

```
export const logout = async (req, res) => {
  try {
    res.clearCookie("jwt").json({ message: "Logged out" });
  } catch (error) {
    console.error("Error in logout:", error);
    res.status(500).json({ error: "Internal Server Error" });
  }
};
```

Rys. 4.13 Funkcja do wylogowywania użytkowników [opracowanie własne]

Funkcja **logout** jest asynchroniczną metodą obsługującą proces wylogowania użytkownika. W ramach bloku **try** usuwa ciasteczko o nazwie **jwt** przy użyciu metody **clearCookie**, a następnie zwraca odpowiedź z komunikatem “Logged out”. W przypadku wystąpienia błędu, jest on przechwytywany w bloku **catch**, a funkcja zwraca odpowiedź z kodem statusu 500 oraz komunikatem "Internal Server Error".

5. Interfejsy użytkownika

W rozdziale zaprezentowano interfejsy użytkownika opracowanej aplikacji wraz z ich strukturą i najważniejszymi elementami składowymi. Omówiono również sposób zaprojektowania interfejsu, mający na celu zapewnienie intuicyjności obsługi oraz wygody użytkowania.

5.1. Struktura interfejsu

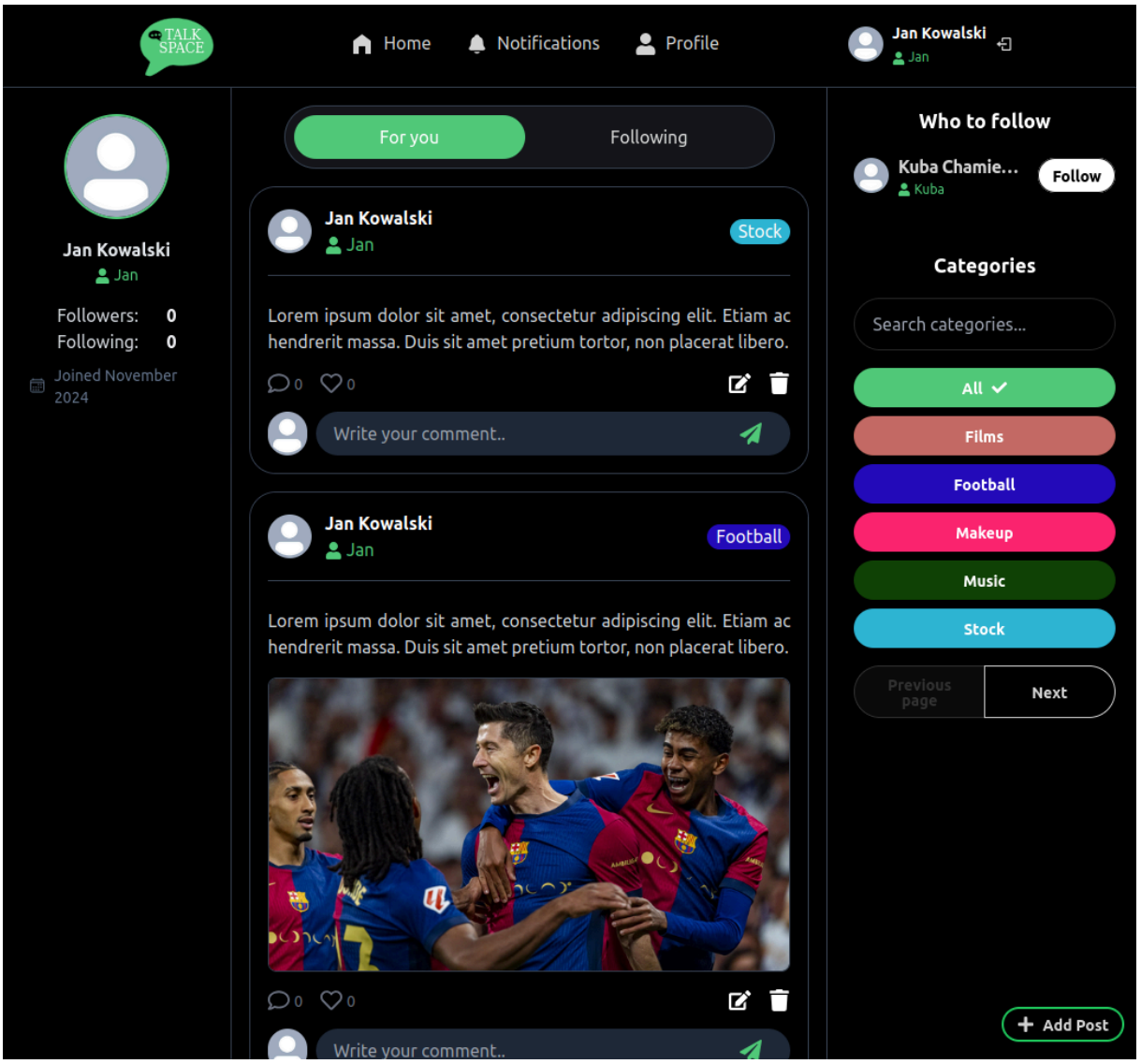
Interfejs użytkownika aplikacji został podzielony na dwie główne części - wersję przeznaczoną dla komputerów oraz wersję zoptymalizowaną dla urządzeń mobilnych. Celem takiego rozwiązania było zapewnienie jak najwyższego komfortu korzystania z aplikacji, niezależnie od rodzaju urządzenia.

- **Wersja desktopowa** - została zaprojektowana z myślą o większych ekranach, co pozwala na wyświetlanie dużej ilości informacji jednocześnie.
- **Wersja mobilna** - została uproszczona, aby interfejs był przejrzysty i łatwy w obsłudze na mniejszych ekranach.

Tabela 5.1 Porównanie wersji mobilnej i desktopowej aplikacji [opracowanie własne]

Różnice i podobieństwa pomiędzy wersją mobilną oraz desktopową aplikacji		
Rodzaj urządzenia	Desktopowe	Mobilne
Panel nawigacji	zawiera	
Panel postów	zawiera	
Panel proponowanych użytkowników	zawiera	nie zawiera
Panel kategorii	Zawiera jako sekcja boczna po prawej stronie	Zawiera jako przycisk otwierający okno kontekstowe
Panel wyszukiwania użytkowników	Zawiera jako sekcja boczna po lewej stronie na ścieżce profilu użytkownika	Zawiera jako przycisk otwierający okno kontekstowe
Panel o podstawowych informacjach użytkownika	Zawiera jako sekcja boczna po lewej stronie	Zawiera jako sekcja pod panelem nawigacji
Przycisk dodawania postu	zawiera	

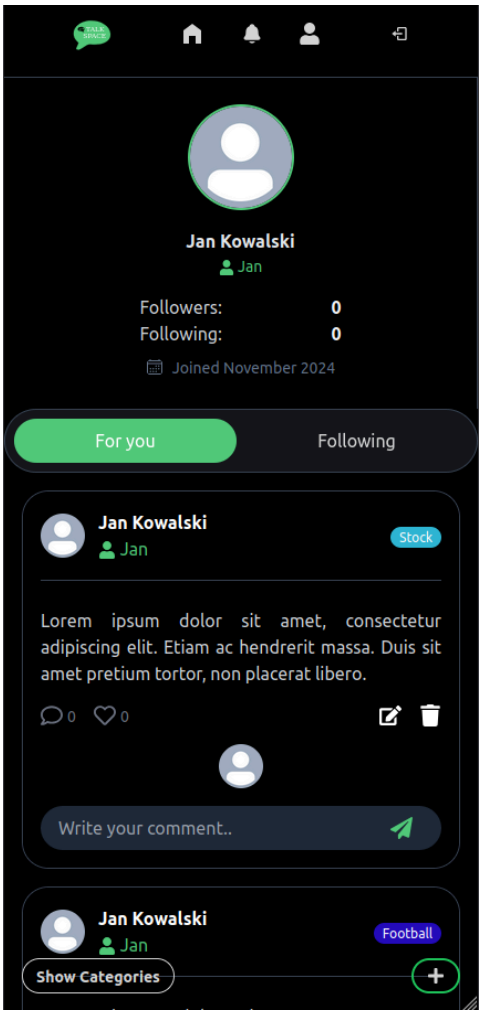
5.2. Interfejs strony głównej



Rys. 5.1 Interfejs głównej strony aplikacji - wersja desktopowa [opracowanie własne]

Rysunek 5.1 przedstawia wersję desktopową strony głównej aplikacji, która składa się z trzech sekcji. Lewy panel boczny zawiera informacje o profilu użytkownika, takie jak zdjęcie profilowe, imię i nazwisko, nazwa użytkownika, liczba obserwujących (ang. *Followers*), liczba obserwowanych (ang. *Following*) oraz data dołączenia do aplikacji. Centralna sekcja jest główną częścią interfejsu, w której wyświetlane są posty użytkowników. Znajdują się w niej zakładki **For You** (wszystkie posty) oraz **Following** (posty użytkowników obserwowanych). Każdy post zawiera: imię i nazwisko autora, nazwę użytkownika, zdjęcie profilowe, treść tekstową, opcjonalny załącznik w postaci pliku graficznego oraz kategorię tematyczną oznaczoną kolorową etykietą. Pod postami umieszczono ikony umożliwiające wyświetlanie komentarzy, dodawanie polubień, usuwanie

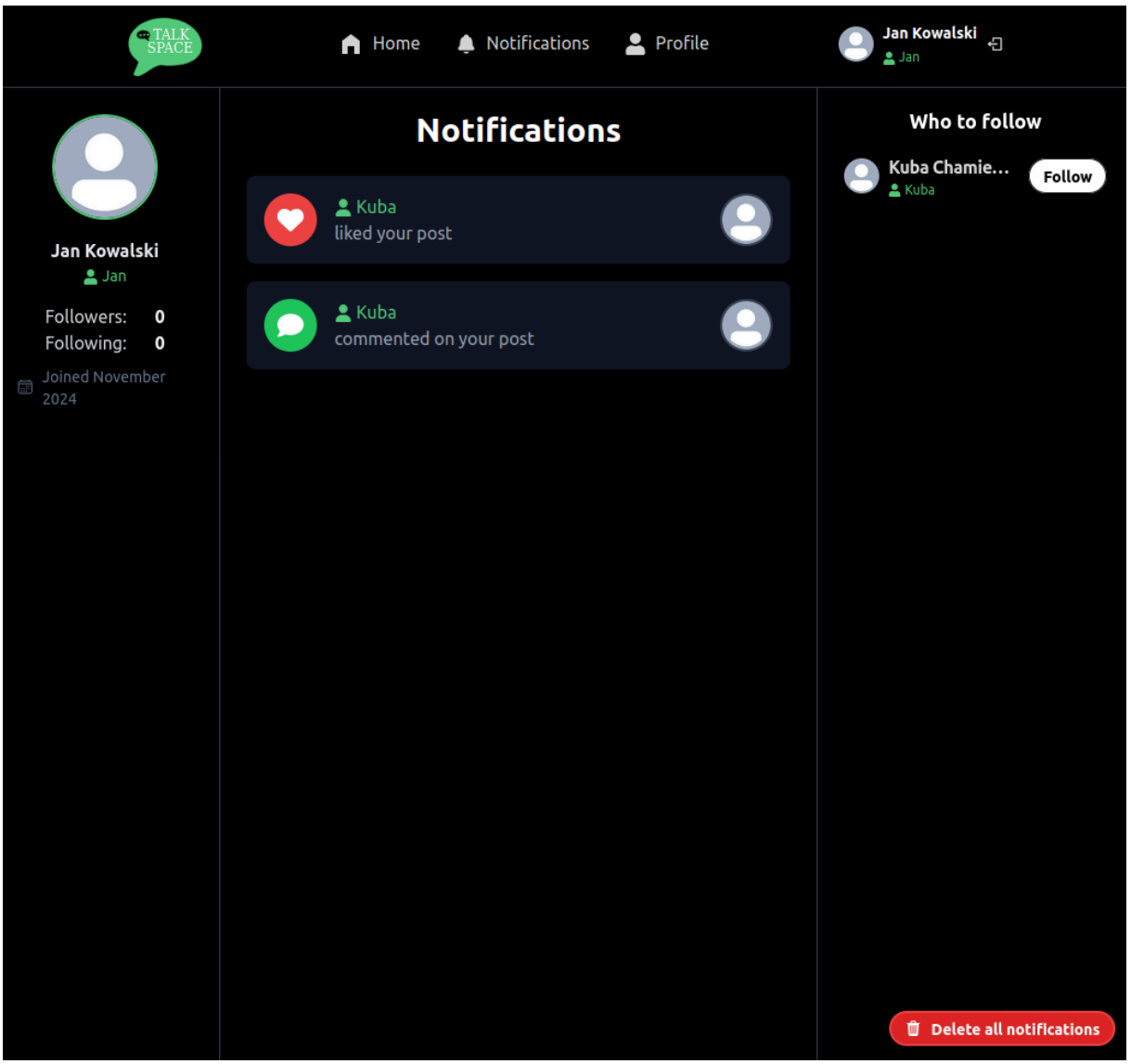
i edytowanie treści, a także pole tekstowe do wprowadzania komentarzy. Prawy panel boczny pełni funkcję nawigacyjną i wspiera interakcję użytkownika z aplikacją. Zawiera sekcję **Who to follow** z sugestiami użytkowników do zaobserwowania, a także listę kategorii tematycznych, pozwalającą filtrować posty według zainteresowań. Znajduje się tam również pole wyszukiwania kategorii oraz przyciski **Previous Page** i **Next** umożliwiające przechodzenie między stronami wyników. Na dole panelu umieszczono przycisk **Add Post**, umożliwiający dodanie nowego posta.



Rys. 5.2 Interfejs głównej strony aplikacji - wersja mobilna [opracowanie własne]

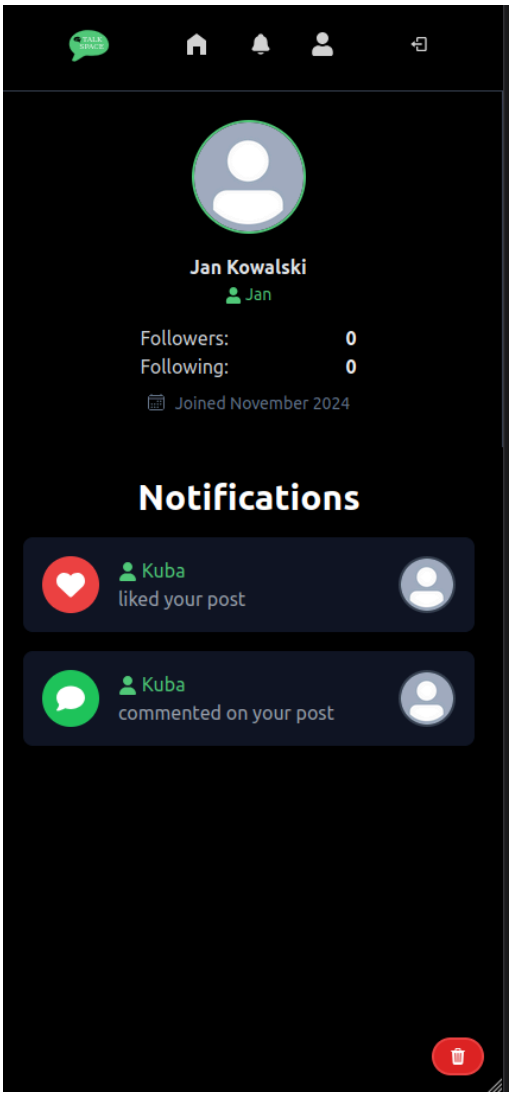
Rysunek 5.2 przedstawia wersję mobilną strony głównej aplikacji. W tej wersji nie ma sekcji **Who to follow** a lista kategorii tematycznych nie jest stale widoczna – kategorie są wyświetlane dopiero po naciśnięciu dedykowanego przycisku. Pozostałe funkcje aplikacji zostały dostosowane do mniejszych rozmiarów ekranu, zachowując jednocześnie czytelność i intuicyjność interfejsu.

5.3. Interfejs powiadomień



Rys. 5.3 Interfejs powiadomień - wersja desktopowa [opracowanie własne]

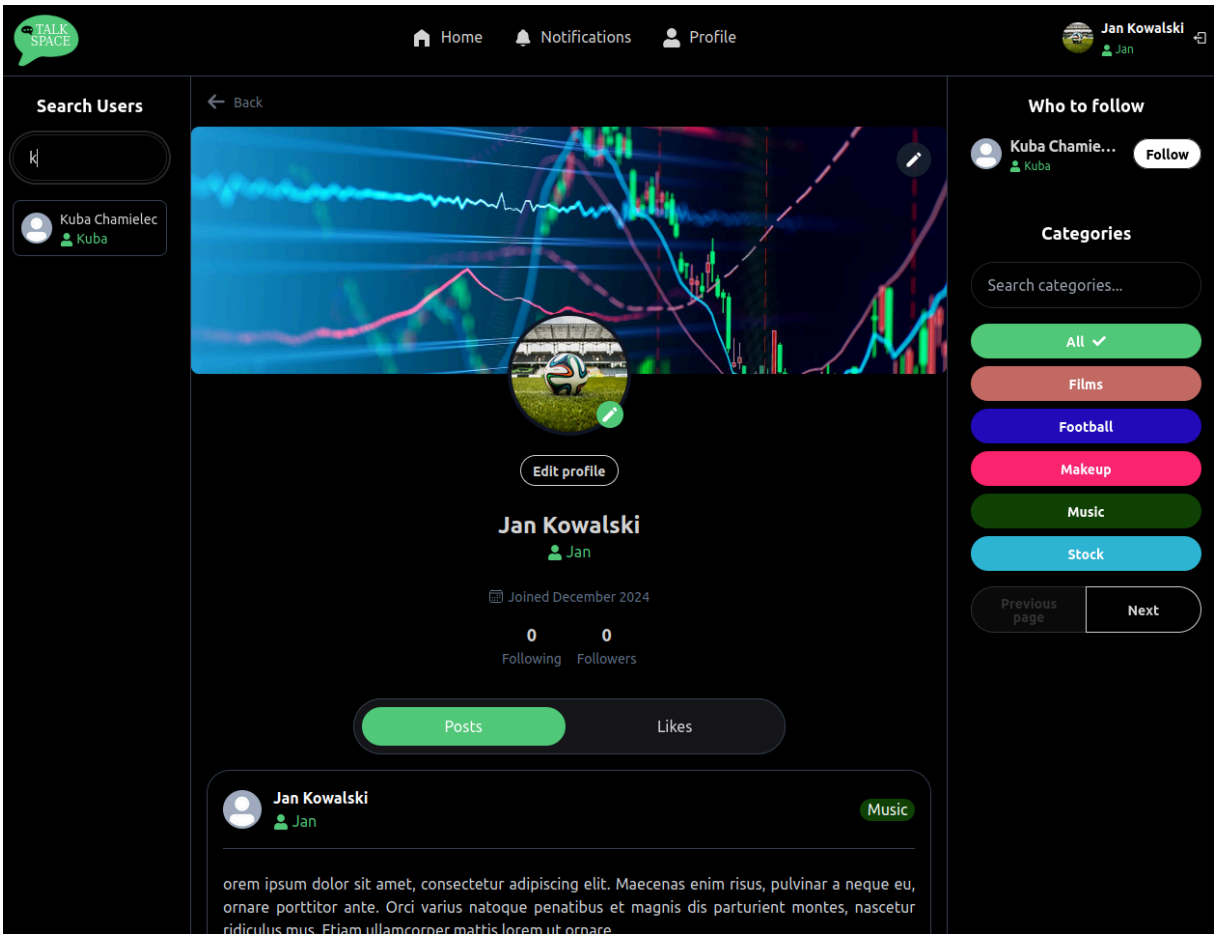
Rysunek 5.3 przedstawia sekcję powiadomień w wersji desktopowej. Interfejs, podobnie jak na **rysunku 5.1**, zawiera lewy panel boczny z informacjami o profilu użytkownika, prawy panel z sekcją **Who to follow** oraz centralny panel, w którym wyświetlane są powiadomienia, takie jak polubienia czy komentarze. Na dole ekranu znajduje się przycisk **Delete all notifications**, umożliwiający szybkie usunięcie wszystkich powiadomień.



Rys. 5.4 Interfejs powiadomień - wersja mobilna [opracowanie własne]

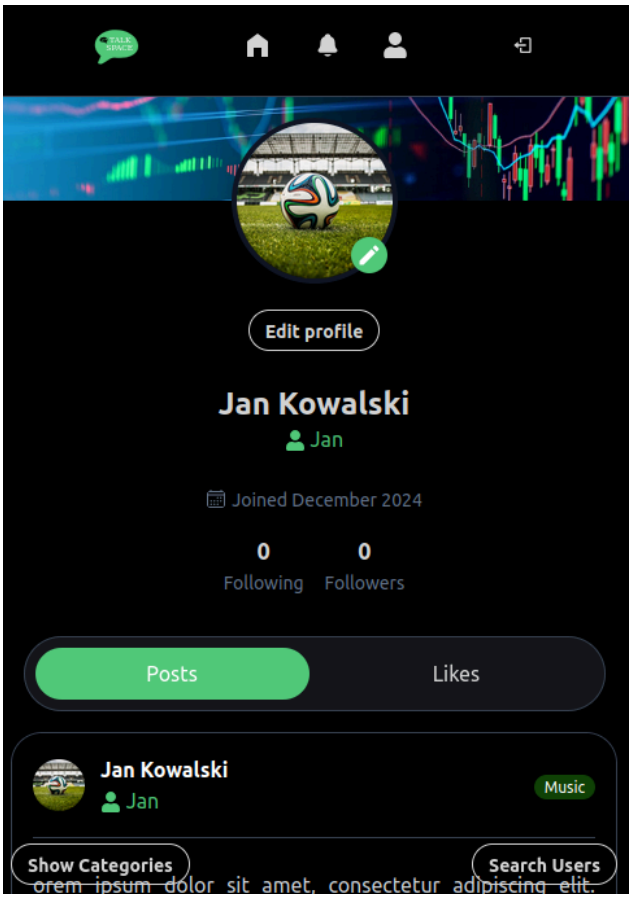
Rysunek 5.4 przedstawia wersję mobilną interfejsu powiadomień. Podobnie jak na **rysunku 5.2** interfejs mobilny nie zawiera panelu **Who to follow**, a lewy panel boczny z informacjami o profilu został przesunięty pod pasek nawigacyjny. Pozostałe funkcje aplikacji zostały dostosowane do mniejszych rozmiarów ekranu, zachowując jednocześnie czytelność i intuicyjność interfejsu.

5.4. Interfejs profilu użytkownika



Rys. 5.5 Interfejs profilu użytkownika - wersja desktopowa [opracowanie własne]

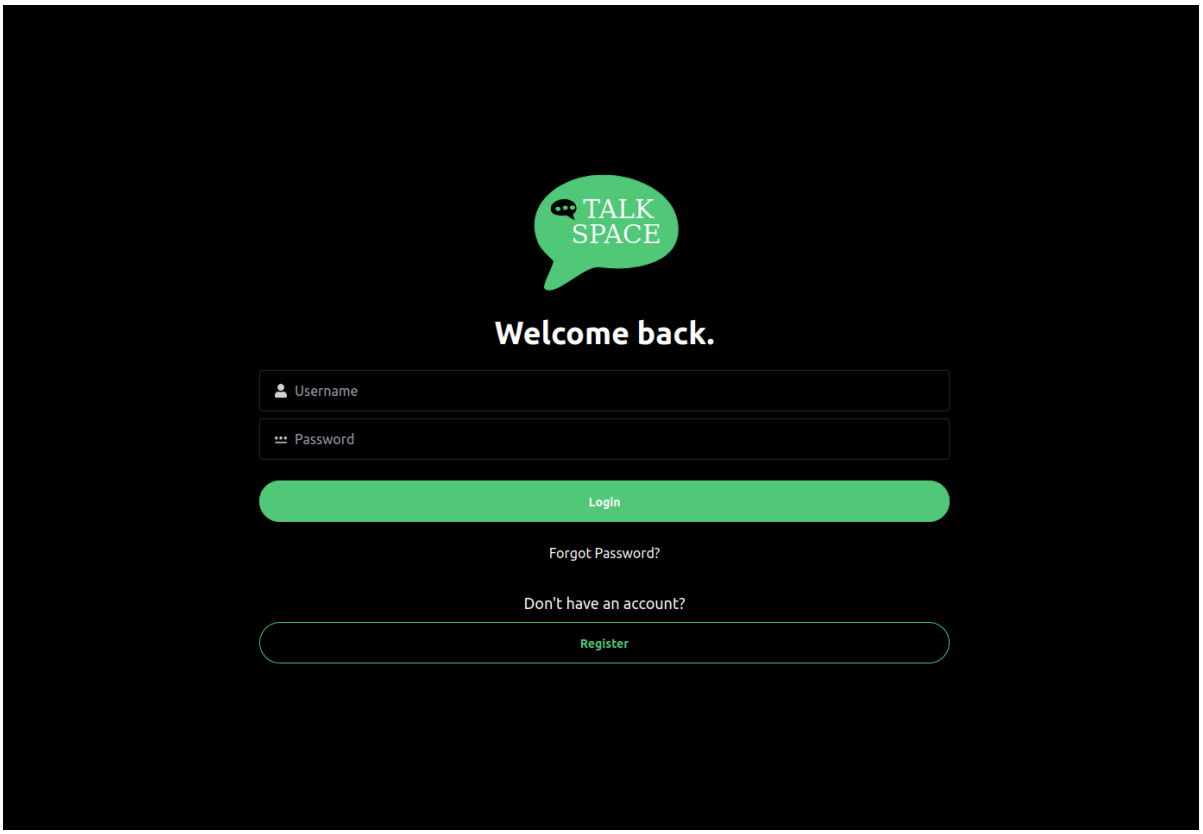
Rysunek 5.5 przedstawia wersję desktopową interfejsu profilu użytkownika. Po lewej stronie interfejsu znajduje się pole do wyszukiwania użytkowników oraz lista wyników wyszukiwania. Centralna sekcja zawiera kartę profilu danego użytkownika, na której znajduje się zdjęcie w tle, zdjęcie profilowe, przycisk do edycji profilu, podstawowe informacje o profilu oraz przyciski **Posts** (opublikowane posty) i **Likes** (polubione posty). Po prawej stronie interfejsu znajduje się sekcja **Who to follow** z sugestiami użytkowników do obserwowania oraz lista kategorii tematycznych z opcją filtrowania treści.



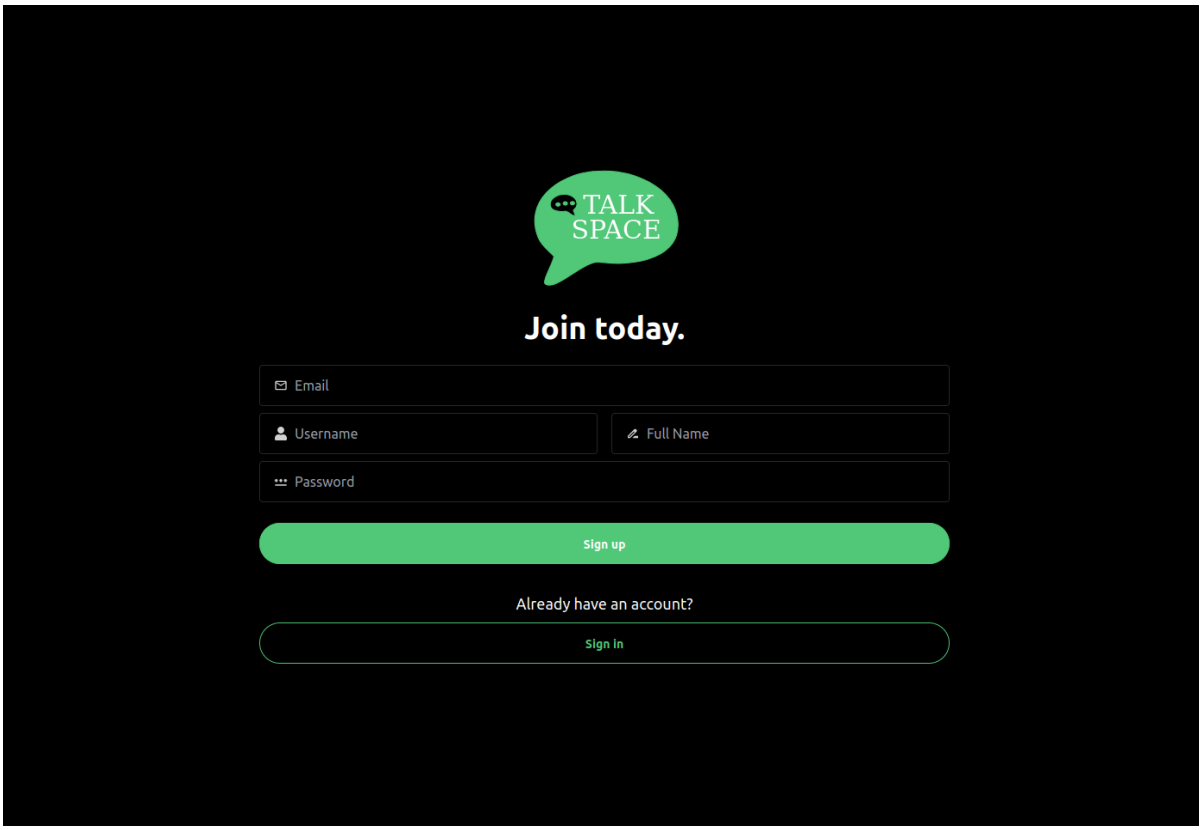
Rys. 5.6 Interfejs profilu użytkownika - wersja mobilna [opracowanie własne]

Rysunek 5.6 przedstawia wersję mobilną interfejsu profilu użytkownika, w której wprowadzono kilka zmian w porównaniu do wersji desktopowej. Interfejs mobilny nie zawiera sekcji **Who to Follow**, a panele wyszukiwania użytkowników oraz filtrowania postów za pomocą kategorii zostały zastąpione przez przyciski, które po naciśnięciu wyświetlają okna kontekstowe.

5.5. Interfejsy logowania i rejestracji



Rys. 5.7 Interfejs logowania [opracowanie własne]



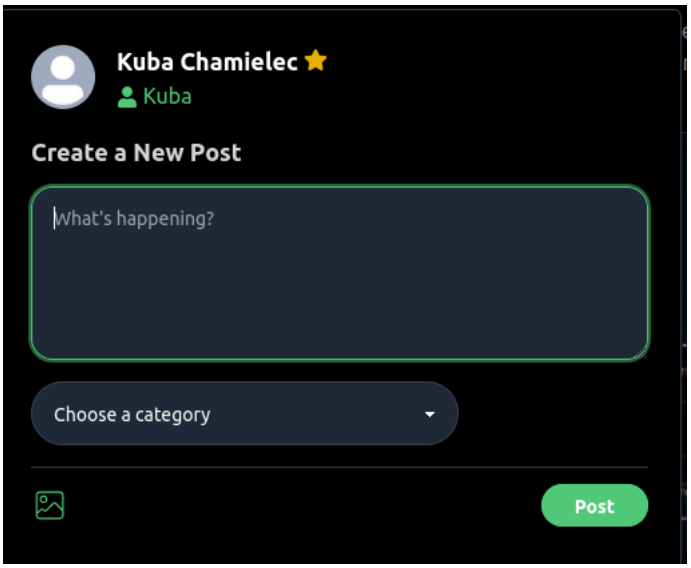
Rys. 5.8 Interfejs rejestracji [opracowanie własne]

Rysunek 5.7 przedstawia ekran logowania aplikacji forum dyskusyjnego. W tym interfejsie znajdują się pola do wpisania nazwy użytkownika oraz hasła, a także przyciski **Forgot password** do przypomnienia hasła, **Login** do zalogowania się na konto oraz **Sign in**, przenoszące użytkownika do interfejsu rejestracji. **Rysunek 5.8** przedstawia ekran rejestracji użytkownika do forum dyskusyjnego. Podczas rejestracji, użytkownik jest zobowiązany podać adres e-mail, nazwę użytkownika, imię i nazwisko oraz hasło. Interfejs zawiera także przyciski **Sign up** do finalizacji rejestracji oraz **Sign in** w przypadku, gdy użytkownik posiada już konto w aplikacji.

5.6. Ważne elementy interfejsu

W tym podrozdziale zaprezentowano istotne dla działania elementy interfejsu, które stanowią rozszerzenie funkcjonalności pozostałych ekranów aplikacji lub uzupełniają je o dodatkowe możliwości.

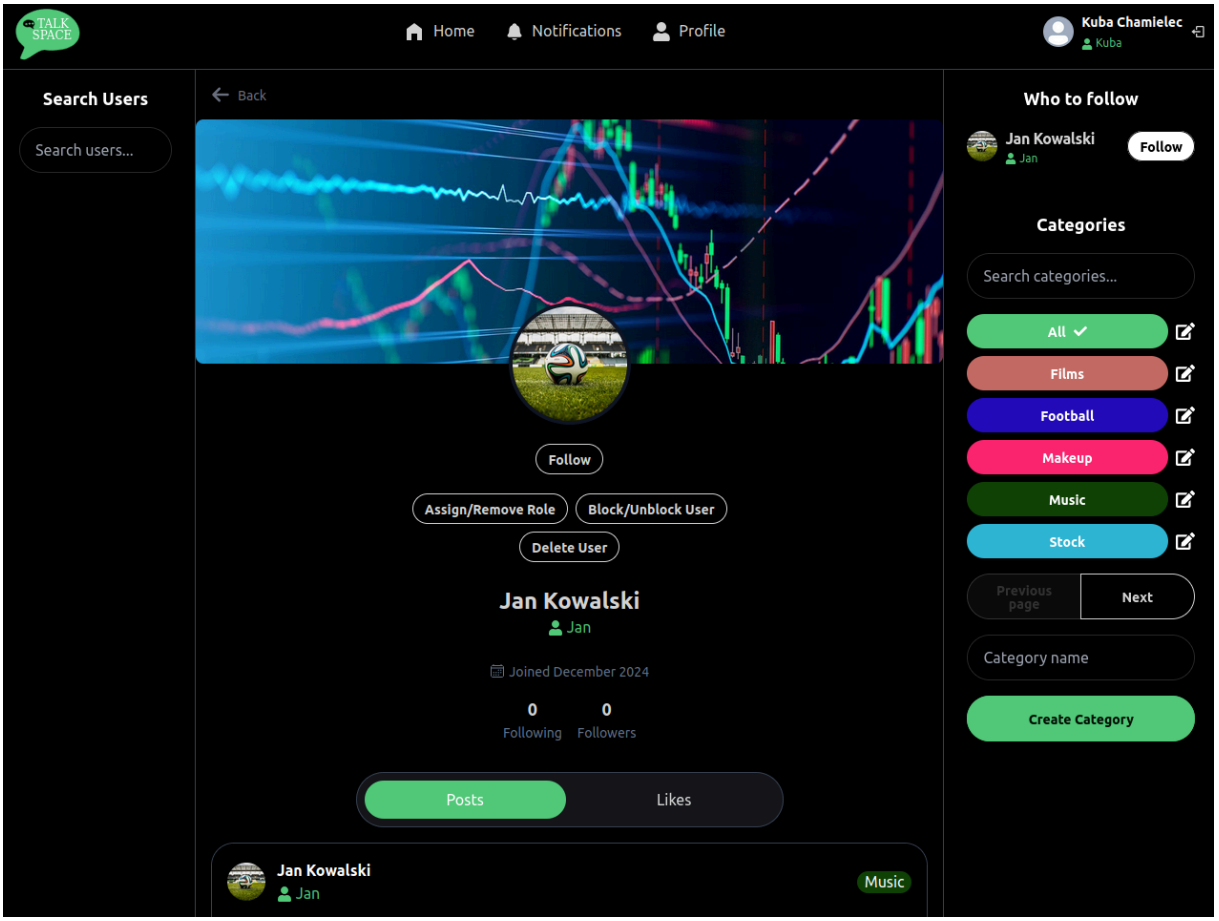
5.6.1. Okno kontekstowe tworzenia postu



Rys. 5.9 Okno kontekstowe tworzenia postu [opracowanie własne]

Rysunek 5.9 przedstawia ekran tworzenia nowego wątku w aplikacji forum dyskusyjnego. W górnej części interfejsu znajduje się nazwa użytkownika oraz avatar z przypisanym imieniem i nazwiskiem. Poniżej umieszczono pole tekstowe, które służy do wpisania treści nowego postu i rozwijane menu służące do przypisania kategorii postowi. W lewym dolnym rogu widoczna jest ikona dodawania załącznika graficznego, natomiast w prawym dolnym rogu znajduje się przycisk **Post**, służący do publikacji postu na forum.

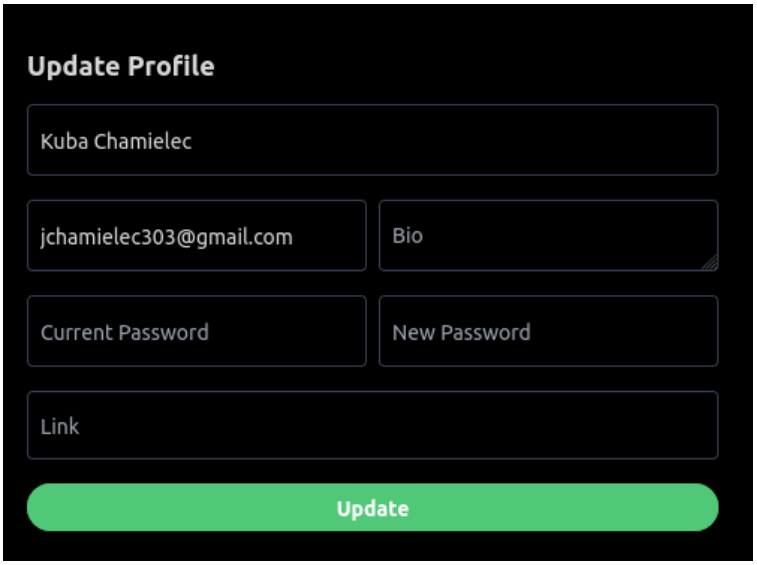
5.6.2. Rozszerzenie interfejsu o funkcje administratora



Rys. 5.10 Rozszerzenie interfejsu o funkcję administratora [opracowanie własne]

Rysunek 5.10 przedstawia rozszerzenie interfejsu z **rysunku 5.5** o funkcje przeznaczone dla użytkowników z rolą administratora. Pod zdjęciem profilowym użytkownika znajdują się dodatkowe przyciski takie jak: **Assign/Remove Role** służący do nadania lub odebrania roli administratora, **Block/Unblock User** służący do blokowania użytkowników forum na okres czasowy lub ich odblokowania, **Delete User** służący do usuwania użytkownika z forum i całej jego aktywności. W panelu **Categories** dostępna jest możliwość edycji istniejącej kategorii oraz stworzenia nowej kategorii.

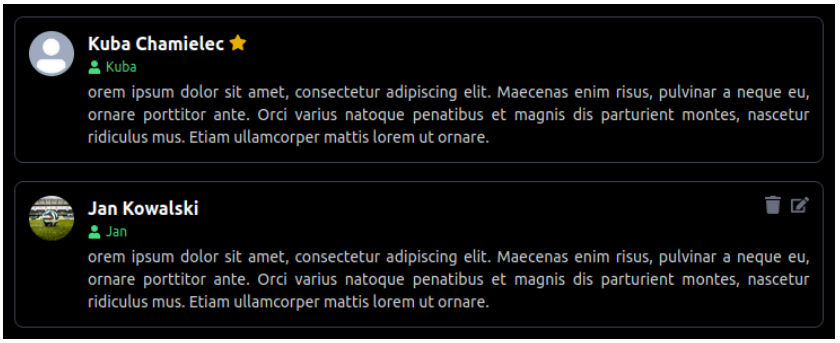
5.6.3. Okno kontekstowe edycji danych użytkownika



Rys. 5.11 Okno kontekstowe edycji danych użytkownika [opracowanie własne]

Rysunek 5.11 przedstawia formularz aktualizacji profilu, który wyświetla się po naciśnięciu przycisku **Edit Profile** w interfejsie profilu użytkownika (**Rysunek 5.5**). Formularz umożliwia wprowadzenie danych, takich jak imię i nazwisko, adres e-mail, biografia oraz link do zewnętrznej strony. Dodatkowo, użytkownik ma możliwość zmiany hasła konta, pod warunkiem poprawnego wprowadzenia dotychczasowego hasła. Na dole formularza znajduje się przycisk **Update**, służący do zapisania wszystkich wprowadzonych zmian w ustawieniach profilu.

5.6.4. Sekcja komentarzy pod postem



Rys. 5.12 Sekcja komentarzy pod postem [opracowanie własne]

Rysunek 5.12 przedstawia sekcje komentarzy pod postem. Pojedynczy komentarz zawiera zdjęcie profilowe użytkownika, jego imię i nazwisko, nazwę użytkownika oraz treść komentarza. Dodatkowo użytkownik ma możliwość edycji lub usunięcia swojego komentarza za pomocą ikon znajdujących się w prawym górnym rogu wpisu.

6. Testy

W procesie tworzenia aplikacji webowej niezbędne jest przeprowadzenie różnorodnych testów w celu zapewnienia poprawnego działania systemu oraz zgodności z wymaganiami funkcjonalnymi. W projekcie zastosowano podejście oparte na zasadzie **Given-When-Then**, która polega na opisywaniu scenariuszy testowych w trzech etapach:

- **Given** - opisuje warunki, które muszą być spełnione przed wykonaniem testu.
- **When** - opisuje konkretne działanie, które jest wykonywane w ramach testu.
- **Then** - opisuje zachowanie aplikacji po wykonaniu testu [12].

W ramach tego podejścia przetestowano aplikacje przy użyciu trzech rodzajów testów:

- **Testów manualnych** - w celu zweryfikowania aplikacji z perspektywy użytkownika oraz sprawdzenia interfejsu i funkcji takich jak: rejestracja, logowanie czy publikowanie postów.
- **Testów integracyjnych** - aby upewnić się, że poszczególne moduły aplikacji współpracują ze sobą poprawnie, szczególnie w obszarach związanych z bazą danych i logiką biznesową.
- **Testów jednostkowych** - dla sprawdzenia poprawności działania poszczególnych funkcji oraz komponentów w izolacji od reszty systemu [13].

6.1. Testy integracyjne

W tym podrozdziale przedstawiono testy integracyjne dla kontrolera odpowiedzialnego za publikowanie postów oraz rejestrację użytkowników w aplikacji.

6.1.1. Test przypisania postu do nieistniejącej kategorii

```
it("should return 404 if category is not found", async () => {
  // Given
  const postData = { text: "Test post", categoryId: "60d21b4667d0d8992e610c85" };

  // When
  const res = await request(app)
    .post("/api/post/create")
    .send(postData)
    .set("Cookie", `jwt=${token}`);

  // Then
  expect(res.status).toBe(404);
  expect(res.body.error).toBe("Category not found");
}, 30000);
});
```

Rys. 6.1 Test przypisania postu do nieistniejącej kategorii [opracowanie własne]

Test zaprezentowany na **rysunku 6.1** sprawdza czy endpoint tworzenia posta **/api/post/create** zwraca odpowiedni błąd, gdy podana kategoria nie istnieje. W bloku **Given** znajduje się obiekt **postData**, który zawiera treść posta oraz nieistniejący identyfikator kategorii. W bloku **When** wysyłane jest żądanie POST do endpointu **/api/post/create** z obiektem **postData** oraz ustawiany zostaje nagłówek **Cookie** z tokenem autoryzacyjnym **jwt=${token}**. W bloku **Then** sprawdzane jest czy status odpowiedzi wynosi 404 (Not Found) oraz czy treścią zwróconego błędu jest "Category not found".

6.1.2. Test tworzenia nowego postu z tekstem i zdjęciem

```
it("should create a new post with text and image", async () => {
  // Given
  const imagePath = path.join(__dirname, "test-image.jpg");
  const imageContent = Buffer.from(
    "iVBORw0KGgoAAAANSUhEUgAAAAEAAAABCAQAAAC1HAwCAAAAC0lEQVR42mP8/wcAAwAB/gnlZAAAAABJRU5ErkJggg==",
    "base64"
  );
  fs.writeFileSync(imagePath, imageContent);

  const uploadRes = await cloudinary.uploader.upload(imagePath);

  const postData = {
    text: "Test post",
    img: uploadRes.secure_url,
    categoryId: createdCategoryId,
  };

  // When
  const res = await request(app)
    .post("/api/post/create")
    .send(postData)
    .set("Cookie", `jwt=${token}`);

  // Then
  expect(res.status).toBe(201);
  expect(res.body.text).toBe("Test post");
  expect(res.body.img).toContain("https://res.cloudinary.com/");
  expect(res.body.category).toBe(createdCategoryId.toString());

  // Cleanup
  fs.unlinkSync(imagePath);
}, 30000);
```

Rys. 6.2 Test tworzenia nowego postu z tekstem i zdjęciem [opracowanie własne]

Test przedstawiony na **rysunku 6.2** weryfikuje działanie endpointu **/api/post/create** pod kątem poprawnego tworzenia nowego posta zawierającego tekst oraz obraz. W bloku **Given** tworzony jest plik graficzny, który później jest przesyłany do bazy obrazów Cloudinary. Tworzony jest także obiekt **postData**, zawierający treść posta, URL przesyłanego obrazu oraz identyfikator kategorii. W bloku **When** wysyłane jest żądanie POST do endpointu **/api/post/create** z obiektem **postData** oraz nagłówkiem **Cookie** zawierającym token autoryzacyjny **jwt=${token}.** W bloku **Then** sprawdzane jest czy odpowiedź ma status

201 (Created), co oznacza pomyślne utworzenie wątku. Weryfikowana jest również treść posta oraz adres URL obrazu, aby upewnić się, że obraz został poprawnie przesłany do Cloudinary. W bloku **Cleanup** usuwany jest plik zdjęciowy utworzony na potrzeby testu.

6.1.3. Test walidacji unikalności nazwy użytkownika

```
it("should return 400 if username is already taken", async () => {
  // Given
  const existingUser = {
    fullName: "Existing User",
    username: "existinguser",
    email: "existinguser@example.com",
    password: "password123",
  };
  const createdUser = await User.create(existingUser);
  createdUserIds.push(createdUser._id);

  const newUser = {
    fullName: "Test User",
    username: "existinguser",
    email: "testuser@example.com",
    password: "password123",
  };

  // When
  const res = await request(app).post("/api/auth/signup").send(newUser);

  // Then
  expect(res.status).toBe(400);
  expect(res.body.error).toBe("Username is already taken");
}, 30000);
```

Rys. 6.3 Test walidacji unikalności nazwy użytkownika [opracowanie własne]

Test zaprezentowany na **rysunku 6.3** weryfikuje działanie endpointu **/api/atuh/signup** pod kątem obsługi sytuacji, gdy nazwa użytkownika jest już zajęta. W bloku **Given** tworzony jest obiekt **existingUser**, reprezentujący użytkownika, który został wcześniej zapisany w bazie danych. Następnie przygotowywany jest obiekt **newUser** z danymi nowego użytkownika, który próbuje zarejestrować się z już zajętą nazwą użytkownika. W bloku **When** wysyłane jest żądanie POST do endpointu **/api/atuh/signup** z obiektem **newUser**. W bloku **Then** weryfikowany jest status odpowiedzi, który powinien wynosić 400 (Bad Request) oraz sprawdzane jest czy wiadomość błędu to “Username is already taken”.

6.1.4. Test poprawności hasła podczas rejestracji

```
it("should return 400 if password is less than 6 characters", async () => {
  // Given
  const userData = {
    fullName: "Test User",
    username: "testuser",
    email: "testuser@example.com",
    password: "123",
  };

  // When
  const res = await request(app).post("/api/auth/signup").send(userData);

  // Then
  expect(res.status).toBe(400);
  expect(res.body.error).toBe("Password must be at least 6 characters long");
}, 30000);
```

Rys. 6.4 Test poprawności hasła podczas rejestracji [opracowanie własne]

Test przedstawiony na **rysunku 6.4** weryfikuje działanie endpointu **/api/atuh/signup** pod kątem obsługi sytuacji, gdy hasło użytkownika ma mniej niż 6 znaków. W bloku **Given** znajduje się obiekt **userData** z danymi użytkownika. W bloku **When** wysyłane jest żądanie POST do endpointu **/api/atuh/signup** z obiektem **userData.** W bloku **Then** sprawdzany jest status odpowiedzi, który powinien wynosić 400 (Bad Request) oraz treść błędu, która powinna zawierać tekst “Password must be at least 6 characters long”.

6.1.5. Weryfikacja poprawności testów integracyjnych

```
PASS backend/tests/auth.controller.test.js
  Auth Controller - Signup
    ✓ should return 400 if email format is invalid (457 ms)
    ✓ should return 400 if username is already taken (847 ms)
    ✓ should return 400 if email is already taken (378 ms)
    ✓ should return 400 if password is less than 6 characters (232 ms)
    ✓ should create a new user and send verification email (673 ms)
```

Rys. 6.5 Wykonane testy integracyjne dla auth.controller [opracowanie własne]

```
PASS backend/tests/post.controller.test.js (6.516 s)
  Post Controller - Create Post
    ✓ should create a new post with text and image (2574 ms)
    ✓ should return 400 if text and image are missing (116 ms)
    ✓ should return 400 if category is missing (141 ms)
    ✓ should return 404 if category is not found (276 ms)
```

Rys. 6.6 Wykonane testy integracyjne dla post.controller [opracowanie własne]

6.2. Testy jednostkowe

W tym podrozdziale przedstawiono testy jednostkowe dla kontrolera odpowiedzialnego za publikowanie postów oraz zarządzania kategoriami w aplikacji. W testach wykorzystano "mocki", które pozwalają na symulowanie działania zależności, takich jak bazy danych czy zewnętrzne usługi, w celu izolacji testowanego kodu i zapewnienia szybkiego oraz przewidywanego przebiegu testów [11].

6.2.1. Test dodawania komentarza do postu

```
it("should add a comment and return 201", async () => {
  // Given
  const post = {
    comments: [],
    save: jest.fn(),
    user: { _id: new mongoose.Types.ObjectId() },
  };
  const mockPopulate = jest.fn().mockResolvedValue(post);
  Post.findById.mockReturnValue({ populate: mockPopulate });
  Notification.prototype.save = jest.fn();
  req.body.text = "Test comment";

  // When
  await commentOnPost(req, res);

  // Then
  expect(post.comments).toHaveLength(1);
  expect(post.save).toHaveBeenCalled();
  expect(Notification.prototype.save).toHaveBeenCalled();
  expect(res.status).toHaveBeenCalledWith(201);
  expect(res.json).toHaveBeenCalledWith(post);
});
```

Rys. 6.7 Test dodawania komentarza do postu [opracowanie własne]

Test zaprezentowany na **rysunku 6.7** weryfikuje działanie funkcji odpowiedzialnej za dodanie komentarza do posta oraz zwrócenie odpowiedniego statusu HTTP. W bloku **Given** znajduje się obiekt **post**, który zawiera pustą tablicę **comments**, metodę **save**, utworzoną przy użyciu **jest.fn()**, która tworzy mock funkcji oraz właściwość **user** z unikalnym identyfikatorem. Blok **When** wywołuje funkcję **commentOnPost** z obiektami **req** oraz **res**. W bloku **Then** sprawdzane jest czy komentarz został dodany do tablicy **comments**. Następnie weryfikowane jest, czy metoda **save** prototypu **Notification** została wywołana, co oznacza utworzenie powiadomienia o dodaniu komentarza. Dodatkowo test sprawdza czy metoda **status** obiektu **res** została wywołana z argumentem 201, co świadczy o pomyślnym utworzeniu zasobu. Na koniec weryfikowane jest czy metoda **json** obiektu **res** została

wywołana z obiektem **post**, co oznacza, że odpowiedź zawiera zaktualizowany post z dodanym komentarzem.

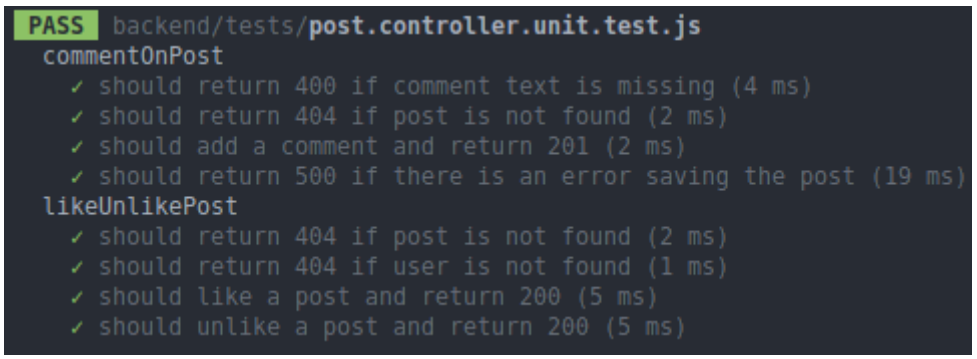
6.2.2. Test sortowania postów według kategorii

```
it("should return posts for a valid categoryId", async () => {
  // Given
  const mockPosts = [
    {
      _id: "postId1",
      text: "This is a test post",
      user: { _id: "userId1", name: "Test User" },
      comments: [
        { user: { _id: "userId2", name: "Commenter" }, text: "Test comment" },
      ],
      category: { _id: "testCategoryId", name: "Tech", color: "#0000FF" },
    },
  ];

  Post.find.mockReturnValue({
    sort: jest.fn().mockReturnValue({
      populate: jest.fn().mockReturnValue({
        populate: jest.fn().mockReturnValue({
          populate: jest.fn().mockResolvedValue(mockPosts),
        }),
      }),
    }),
  });

  // When
  await getPostsByCategory(req, res);

  // Then
  expect(res.status).toHaveBeenCalledWith(200);
  expect(res.json).toHaveBeenCalledWith(mockPosts);
});
```

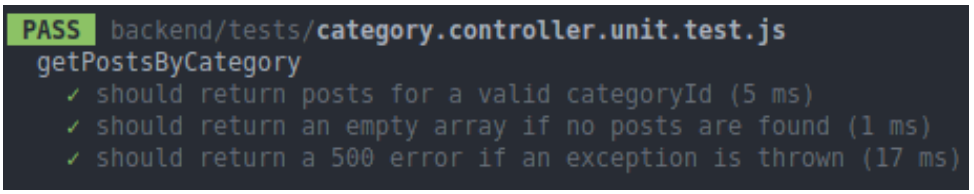
Rys. 6.8 Test sortowania postów według kategorii [opracowanie własne]

Test przedstawiony na **rysunku 6.8** sprawdza, czy funkcja **getPostsByCategory** poprawnie zwraca posty dla danego **categoryId**. W bloku **Given** tworzony jest mock danych **mockPosts**, który reprezentuje przykładowy post z przypisaną kategorią, a metoda **Post.find** jest zamockowana, aby zwracała te dane po łańcuchowym wywołaniu metod **sort** i **populate**. W bloku **When** wywoływana jest funkcja **getPostsByCategory** z obiektami **req** i **res**. W bloku **Then** sprawdzane jest, czy funkcja zwraca odpowiedź z kodem statusu 200 oraz czy metoda json została wywołana z wartością **mockPosts**.

6.2.3. Weryfikacja poprawności testów jednostkowych



Rys. 6.9 Wykonane testy jednostkowe dla post.controller [opracowanie własne]



Rys. 6.10 Wykonane testy jednostkowe dla category.controller [opracowanie własne]

6.3. Testy manualne

W tym podrozdziale przedstawiono wybrane testy manualne. Omówiono scenariusze testowe, które mają na celu sprawdzenie poprawności współdziałania komponentów systemu.

6.3.1. Test usuwania posta

Tabela 6.1 Test usuwania postu [opracowanie własne]

ID	TC07
Tytuł	Usuwanie posta.
Warunki początkowe	Użytkownik jest zalogowany na konto i znajduje się na stronie głównej aplikacji **(Home Page).**
Kroki testowe	1. Wybierz post, który chcesz usunąć. 2. Jeśli jesteś autorem posta lub administratorem, naciśnij ikonę symbolizującą kosz, która znajduje się w prawym dolnym rogu posta.
Oczekiwany rezultat	Post znika ze strony głównej, a system wyświetla komunikat potwierdzający usunięcie.

6.3.2. Test zmiany hasła użytkownika

Tabela 6.2 Test zmiany hasła użytkownika [opracowanie własne]

ID	TC010
Tytuł	Zmiana hasła użytkownika.
Warunki początkowe	Użytkownik znajduje się na stronie logowania **(Login Page).**
Kroki testowe	1. Naciśnij link z napisem **Forgot Password?** 2. W polu input z napisem “Email” wpisz swój adres mailowy. 3. Naciśnij przycisk **Send reset link.** 4. Przejdź do adresu email, który podałeś przy rejestracji konta i kliknij w link do **resetowania hasła.** 5. W polu input z napisem “New Password” wpisz nowe hasło.
Oczekiwany rezultat	Na ekranie pojawia się komunikat potwierdzający zmianę hasła.

6.3.3. Test edycji posta

Tabela 6.3 Test edycji posta [opracowanie własne]

ID	TC006
Tytuł	Edycja posta.
Warunki początkowe	Użytkownik jest zalogowany na konto i znajduje się na stronie głównej aplikacji (**Home Page**).
Kroki testowe	1. Wybierz post, który chcesz edytować. 2. Jeśli jesteś autorem posta lub administratorem, naciśnij ikonę symbolizującą edycję, która znajduje się w prawym dolnym rogu posta. 3. *(Opcjonalnie)* Zmień treść posta. 4. *(Opcjonalnie)* Zmień kategorię przypisaną do posta. 5. Naciśnij przycisk **Update.**
Oczekiwany rezultat	Zaktualizowana treść i kategoria posta są widoczne na stronie głównej aplikacji.

7. Podsumowanie i wnioski

W ramach pracy inżynierskiej opracowano aplikację forum dyskusyjnego, która stanowi nowoczesne narzędzie wspierające wymianę poglądów oraz informacji pomiędzy użytkownikami. Celem projektu było wykonanie intuicyjnego systemu do dzielenia się opiniami, wdrożenie funkcji sprzyjających interakcjom między użytkownikami oraz zapewnienie bezpieczeństwa i możliwości moderacji treści. Zaprojektowane forum realizuje wszystkie wymagania funkcjonalne i niefunkcjonalne, opisane w rozdziale drugim takie jak:

1. Funkcjonalne:
   - Rejestracja i logowanie użytkowników.
   - Tworzenie postów i odpowiadanie na wątki.
   - Moderacja treści.
   - Kategoryzacja treści.
   - System oceniania postów.
2. Niefunkcjonalne:
   - Bezpieczeństwo i autoryzacja.
   - Zabezpieczenie ścieżek administratora.
   - Usuwanie niezweryfikowanych kont.
   - Responsywność.
   - Wydajność i skalowalność.

Aplikacja została zaprojektowana w sposób umożliwiający łatwą rozbudowę i dostosowanie do zmieniających się potrzeb użytkowników. Możliwości rozwoju obejmują między innymi:

- Pozycjonowanie treści względem ocen użytkowników.
- Możliwość proponowania nowych kategorii przez użytkowników.
- Dodawanie filmów do postów.

Realizacja tego projektu pozwoliła mi poszerzyć wiedzę zdobytą podczas studiów, szczególnie w zakresie projektowania i implementacji aplikacji webowych, zarządzania bazami danych oraz zapewniania bezpieczeństwa systemów. Dodatkowo praca nad aplikacją umożliwiła mi rozwój umiejętności praktycznych, takich jak zarządzanie projektem, analiza wymagań oraz stosowanie nowoczesnych technologii w praktyce.

Bibliografia

[1] Haverbeke, M. *Zrozumieć JavaScript*, wydanie III, tłum. Piotr Gociek, Helion, Gliwice, 2018

[2] Brown, E. *Web Development with Node and Express*, 2nd Edition, O'Reilly Media, Sebastopol, 2019

[3] Subramanian, V. *Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React, and Node*. Apress, New York, 2019

[4] Chodorow, K. *MongoDB: The Definitive Guide*. 2nd Edition, O'Reilly Media, Sebastopol, 2013

[5] Bcrypt [Online] *https://en.wikipedia.org/wiki/Bcrypt*

[6] JWT [Online] *https://jwt.io/*

[7] Cookie-parser [Online] *https://www.npmjs.com/package/cookie-parser*

[8] Masse, M. *REST API Design Rulebook*, O'Reilly Media, Sebastopol, 2011

[9] React.js [Online] *https://react.dev/*

[10] What is HTTP? [Online] *https://www.freecodecamp.org/news/what-is-http/*

[11] Mocking in Unit Tests [Online] *https://microsoft.github.io/code-with-engineering-playbook/automated-testing/unit-testing/mocking/*

[12] Given-When-Then [Online] *https://en.wikipedia.org/wiki/Given-When-Then*

[13] Automated Testing [Online] *https://microsoft.github.io/code-with-engineering-playbook/automated-testing/*

Spis rysunków

Spis tabel