



POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Krzysztof Żelazek
s27598

System wersjonowania plików wspomagający zdalną współpracę nad materiałami audio „WaveScope”

Praca inżynierska pod kierunkiem

dr inż. Mariusza Trzaski

miejsce, miesiąc, rok obrony

Streszczenie

Praca odpowiada na brak ustandaryzowanych narzędzi do zdalnej współpracy w branży audio, zastępując uniwersalne rozwiązania (np. Google Drive) dedykowanym systemem. Głównym celem było zaprojektowanie i zaimplementowanie aplikacji webowej umożliwiającej zarządzanie zespołami, katalogowanie projektów, wersjonowanie utworów oraz wizualizację dźwięku z funkcją dodawania komentarzy do konkretnych fragmentów pliku. System oparto na technologiach Java, Spring i Hibernate (backend) oraz React i wavesurfer.js (frontend). Skuteczność i intuicyjność rozwiązania potwierdziły testy z udziałem profesjonalnych realizatorów dźwięku oraz muzyków.

Słowa kluczowe:

Java, React, Spring, Hibernate, RestApi, aplikacja webowa, współpraca zdalna, produkcja muzyczna.

Podziękowania

Pragnę serdecznie podziękować mojemu promotorowi, dr. inż. Mariuszowi Trzasce, za poświęcony czas, ogromną cierpliwość do sprawdzania wielu wersji tekstu oraz nieustanne nakierowywanie mnie na właściwe, inżynierskie tory – szczególnie w momentach, gdy zawodziły podstawy.

Wielkie podziękowania kieruję również do mojej nieoficjalnej ekipy QA, czyli wszystkich użytkowników, którzy zechcieli poświęcić swój czas na przeklikanie i przetestowanie prototypu.

Na koniec dziękuję moim bliskim. Zarówno za motywację do dowiezienia tej pracy do końca, jak i za to, że dzielnie znosili opowieści o moich nowych, wspaniałych pomysłach oraz frustracjach napotkanych w procesie programowania.

Spis treści

Streszczenie	2
Słowa kluczowe:	2
Podziękowania	2
1. WSTĘP	5
1.1. OMÓWIENIE PROBLEMATYKI	5
1.2. CEL PRACY	6
1.3. UZASADNIENIE WYBORU TEMATYKI ORAZ MOTYWACJA DO JEJ PODJĘCIA	6
1.4. ROZWIĄZANIA PRZYJĘTE W PRACY	7
2. PRZEGLĄD ISTNIEJĄCYCH ROZWIĄZAŃ DO ZDALNEJ WSPÓŁPRACY NAD MATERIAŁEM AUDIO	8
2.1. METODY KOMUNIKACJI I WYMIANY PLIKÓW W BRANŻY AUDIO	8
2.2. PRZEGLĄD DEDYKOWANYCH PLATFORM DO WSPÓŁPRACY NAD DŹWIĘKIEM	9
2.2.1 Avid Cloud Collaboration	9
2.2.2 Opusnix	10
2.2.3 Dropbox Replay	11
2.3. ANALIZA MECHANIZMÓW OZNACZANIA CZASU I WERSJONOWANIA	12
2.4. ZESTAWIENIE PORÓWNAWCZE OMAWIANYCH ROZWIĄZAŃ	14
2.5. PODSUMOWANIE ANALIZY I IDENTYFIKACJA LUK FUNKCJONALNYCH	16
3. PROPOZYCJA SYSTEMU DO ZDALNEJ WSPÓŁPRACY NAD MATERIAŁAMI AUDIO	17
3.1. CELE	17
3.2. ZAŁOŻENIA I ZAKRES PROJEKTU	17
3.3. OKREŚLENIE WYMAGAŃ FUNKCJONALNYCH I NIEFUNKCJONALNYCH	18
3.3.1 Wymagania funkcjonalne	18
3.3.2 Wymagania нефункционалне	19
3.4. ZDEFINIOWANIE I PROJEKT DIAGRAMU PRZYPADKÓW UŻYCIA	19
3.5. PROJEKT DIAGRAMÓW KLAS DLA SYSTEMU	22
3.5.1 Analityczny diagram klas	23
3.5.2 Projektowy diagram klas	25
3.6. WSTĘPNY PROJEKT INTERFEJSU UŻYTKOWNIKA REALIZUJĄCY KLUCZOWE FUNKCJONALNOŚCI	27
4. TECHNOLOGIE I NARZĘDZIA WYKORZYSTANE W PRACY	30
4.1. JAVA	30
4.2. REACT.JS	30
4.3. POSTGRESQL	31
4.4. SPRING BOOT	31
4.5. HIBERNATE	32
4.6. WAVESURFER.JS	33
4.7. REST API	33
4.8. DOCKER	35
4.9. JUNIT	35
4.10. MICROSOFT CLARITY	36
5. IMPLEMENTACJA SYSTEMU	37
5.1. IMPLEMENTACJA WARSTWY SERWEROWEJ – BACKEND	37
5.1.1 Struktura katalogów i wzorzec warstwowy	37
5.1.2 Modelowanie danych i komunikacja z bazą	38
5.1.3 Komunikacja REST API i separacja danych za pomocą obiektów DTO	40
5.1.4 Bezpieczeństwo i bezstanowa autoryzacja JWT	41
5.1.5 Logika biznesowa i wersjonowanie plików audio	48
5.1.6 Zarządzanie plikami i integracja z zewnętrznym API Google Drive	51
5.2. IMPLEMENTACJA CZĘŚCI KLIENCKIEJ – FRONTEND	54
5.2.1 Architektura komponentowa i struktura projektu React	54

5.2.2	<i>Zarządzanie stanem aplikacji</i>	55
5.2.3	<i>Asynchroniczna komunikacja z serwerem i obsługa zapytań</i>	58
5.2.4	<i>Integracja odtwarzacza audio i waveform</i>	60
5.2.5	<i>Nawigacja wewnątrz aplikacji i routing</i>	63
5.2.6	<i>Responsywność i adaptacyjny interfejs użytkownika</i>	64
6.	INTERFEJS UŻYTKOWNIKA	65
6.1.	INTERFEJS DESKTOPOWY	65
6.2.	INTERFEJS MOBILNY	69
7.	TESTOWANIE APLIKACJI	70
7.1.	TESTY JEDNOSTKOWE	70
7.2.	TESTY INTERFEJSU UŻYTKOWNIKA	72
8.	PODSUMOWANIE	76
8.1.	NAPOTKANE TRUDNOŚCI	76
8.2.	POTENCJAŁ ROZWOJU APLIKACJI	77
8.3.	WNIOSKI Z PROCESU POWSTAWANIA PROTOTYPU APLIKACJI	78
9.	BIBLIOGRAFIA	80
10.	SPIS LISTINGÓW	85
11.	SPIS ILUSTRACJI	87

1. Wstęp

Rozdział pierwszy stanowi ogólne wprowadzenie do tematyki niniejszej pracy. Zarysowano w nim kontekst projektu oraz zdefiniowano problemy, z jakimi mierzy się współczesna branża audio w zakresie zdalnej współpracy. W dalszej części rozdziału sformułowano główny cel pracy, opisano motywację do podjęcia wybranego tematu, a także pokrótce przedstawiono wybrane rozwiązania technologiczne i metodyczne wykorzystane podczas tworzenia prototypu aplikacji.

1.1. Omówienie problematyki

Dźwięki otaczają nas dookoła. Są czymś, bez czego nie wyobrażamy sobie naszego życia. Tworzymy różne treści audio, których słuchamy dla własnej przyjemności, od piosenek, przez audiobooki, aż po podcasty. Korzystamy z tych treści w drodze do pracy, podczas wykonywania codziennych czynności czy też wieczorami dla relaksu. Dźwięk stał się produktem, który tak samo jak inne produkty przechodzi proces wytwarzania i można go porównać do wytwarzania innych treści, które mogą mieć dystrybucję cyfrową, takich jak artykuły czy filmy.

W dzisiejszych czasach, szczególnie po pandemii, praca zdalna stała się standardem i jest ona pożądana przez wiele osób. Praca taka jest z jednej strony ułatwieniem, ponieważ umożliwia współpracę, która stacjonarnie nie miałaby szans się odbyć. Z drugiej zaś niesie wiele nowych wyzwań, które w przypadku stacjonarnej współpracy nie mają miejsca.

W branży audio głównym takim problemem jest organizacja i komunikacja. Na szczególną uwagę zasługuje precyzja wyrażanych opinii odnośnie audio, nad którym jest toczona współpraca. W przypadku audio, którego częścią jest tekst, takich jak piosenki czy podcasty, sytuacja jest o tyle łatwiejsza, że język naturalny jest zrozumiały dla każdego. Komentarz „słowa ‘Ala ma kota’ są za cicho” jest zrozumiały i precyzyjny. Komplikacja powstaje, gdy współpraca dotyczy utworu instrumentalnego. Stwierdzenie: „drugie przejście zagrane nierówno” w przypadku utworu instrumentalnego jest niejednoznaczne i nie wskazuje konkretnego miejsca w sytuacji, gdzie każdy instrument ma swoje przejścia.

Częściowym rozwiązaniem jest dodawanie odnośnika w czasie, ale narzędzia udostępniane przez domyślne programy audio są nieprecyzyjne i ograniczają się w większości jedynie do sekund. W wielu przypadkach jest to za mało dokładne i wymaga późniejszego doprecyzowywania myśli osób współpracujących nad materiałem audio.

Współczesny rynek muzyczny oraz, szerzej rzecz ujmując, audio nie wykształcił jednej wiodącej metodyki zdalnej współpracy. W czasach przed erą komputeryzacji problem ten nie istniał, ponieważ w celu odsłuchania i oceny wspólnie stworzonego dzieła, czy to muzycznego, czy słuchowiska, konieczne było spotkanie się w studiu muzycznym i odsłuchanie bezpośrednio z taśmy. Komunikacja podczas takiej współpracy była łatwa oraz bardziej bezpośrednia, ponieważ w każdej chwili można było skomentować dany fragment i wskazać dokładnie, jakie uwagi do niego mamy. Dodatkowo praca na analogowych nośnikach audio mocno ograniczała postprodukcję, która miała limit poprawek oraz ich zakres.

Czasy komputeryzacji i digitalizacji audio wprowadziły Nielimitowaną liczbę poprawek, które można wprowadzać do zrealizowanego nagrania. Doprowadziło to do tego, że klienci korzystający z usług realizatorów nagrań zgłaszają często wiele mikroskopijnych uwag i chcą, żeby było tworzone wiele wersji uwzględniających te uwagi.

Praca zdalna w środowisku, w którym brak jest ustandaryzowanych narzędzi, prowadzi do wielu niepotrzebnych błędów oraz sprawia, że współpraca z różnymi osobami może wyglądać zupełnie inaczej pod kątem technicznym i organizacyjnym. Branża audio cierpi na brak takiego ustandaryzowanego narzędzia. Główną bolączką, która dotyka szczególnie realizatorów współpracujących z klientami oczekującymi wiele poprawek, jest wersjonowanie plików audio oraz sprawienie, żeby klient na pewno odsłuchiwał najnowszej wersji. Praca nad plikami, których nazwa brzmi „piosenka.final.wersjaOstateczna.v4.mp3”, jest niestety często spotykana.

Kanały udostępniania plików audio również nie są ustandaryzowane i różni producenci muzyczni korzystają z własnych ulubionych rozwiązań. Najpopularniejszymi są przesyłanie plików przy użyciu chmury osobistej, takich jak Google Drive [1] czy Dropbox [2] lub też przesyłanie plików za pomocą narzędzi typu WeTransfer [3].

1.2. Cel pracy

Celem pracy było stworzenie uniwersalnego narzędzia, które będzie przyjazne zarówno dla realizatora dźwięku, jak i jego klientów. W ramach pracy powstał prototyp aplikacji webowej o nazwie „WaveScope”. Nazwa ma zachęcać osoby korzystające z aplikacji do skupienia się na dźwięku oraz fali audio celem krytycznego odsłuchu materiału, nad którym współpracują.

Pierwszą główną funkcjonalnością, na której opiera się prototyp, jest wersjonowanie plików audio. Funkcjonalność ta ma na celu wsparcie użytkowników w zarządzaniu plikami, nad którymi pracują. Funkcja zapewnia użytkownikowi pewność, że domyślnie odsłuchuje aktualną wersję.

Drugim głównym aspektem pracy było stworzenie przyjaznego interfejsu użytkownika dla aplikacji webowej. Interfejs, który powstał, pozwala na intuicyjne tworzenie zespołów projektowych oraz zarządzanie projektami w ramach konkretnego zespołu. Najważniejszym widokiem aplikacji jest widok konkretnego projektu. Widok ten oferuje wizualizację audio pozwalającą na precyzyjną wizualną lokalizację konkretnych fragmentów, funkcję zaznaczania fragmentów audio celem dodania komentarza oraz inne opcje wspierające współpracę nad materiałem audio i krytyczne słuchanie.

Trzecią funkcją, która została zrealizowana, jest udostępnienie użytkownikowi możliwości wyboru miejsca przechowywania jego plików audio. Użytkownik może wybrać jedno z dwóch miejsc, w których są jego pliki, czyli bezpośrednio na serwerze aplikacji lub na jego własnym Google Drive. Aby umożliwić przechowywanie audio na dysku Google, została zaimplementowana integracja z kontem Google użytkownika.

1.3. Uzasadnienie wyboru tematyki oraz motywacja do jej podjęcia

Motywacją do podjęcia realizacji tego tematu były moje osobiste doświadczenia. Jako osoba hobbystycznie zajmująca się muzyką, spotkałem się z problemami opisanymi w podrozdziale 1.1, w którym została omówiona problematyka. Docelowymi użytkownikami, do których kierowany jest prototyp, są mój zespół muzyczny oraz osoby, z którymi okazjonalnie współpracuję nad muzyką oraz innymi formami dźwięku.

Istotnym impulsem do opracowania autorskiego prototypu była praca zawodowa w sektorze postprodukcji dźwięku w filmach i serialach. Wykorzystywane tam profesjonalne narzędzia webowe do recenzji materiałów wideo wykazują ogromną skuteczność w procesie zatwierdzania kolejnych wersji montażowych. Zauważono jednak brak analogicznego, dedykowanego i przystępnego cenowo rozwiązania skupionego wyłącznie na specyfice pracy z dźwiękiem (np. na precyzyjnej pracy z falą dźwiękową).

Niniejszy projekt stanowi próbę przeniesienia najlepszych wzorców z branży filmowej do świata produkcji muzycznej. Wybór technologii webowych oraz planowana w przyszłości migracja do infrastruktury chmurowej (np. AWS) podyktowane są jest potrzebą zapewnienia wysokiej dostępności oraz łatwości współdzielenia plików audio bez konieczności instalowania zewnętrznego oprogramowania przez klienta końcowego. Takie podejście nie tylko optymalizuje proces twórczy, ale również otwiera drogę do komercjalizacji projektu w modelu subskrypcyjnym (SaaS [4]), dostosowanym do różnych potrzeb użytkowników – od niezależnych artystów po profesjonalne studia nagraniowe.

1.4. Rozwiązania przyjęte w pracy

Praca nad prototypem została podzielona na kilka kluczowych etapów. Etapy te są standardem w produkcji oprogramowania i są niezbędne w celu profesjonalnego stworzenia systemu informatycznego.

Pierwszym etapem było zdefiniowanie problematyki. Etap ten został rozszerzony o analizę wymagań użytkownika, wstępne modelowanie struktury danych oraz projekt graficzny (*mockup*) najważniejszego widoku aplikacji, czyli widoku jednego konkretnego projektu.

Drugim etapem był wybór narzędzi oraz implementacja zaprojektowanych rozwiązań. Prototyp został zbudowany w standardowym stacku technologicznym dla aplikacji webowych. Część serwerowa została zbudowana w języku Java [5] opartym na *frameworku* Spring [6] i narzędziu Hibernate [7]. Serwer dane przechowuje w bazie danych PostgreSQL [8]. Baza ta udostępnia wszystkie swoje rozwiązania na licencji *open source*. Warstwa kliencka została zbudowana przy użyciu React [9], wykorzystując TypeScript [10] jako język programowania. Główną zewnętrzną biblioteką, która została wykorzystana, jest wavesurfer.js [11], która umożliwia wizualizację audio oraz nawigację odtwarzania.

Ostatnim etapem były testy powstałego prototypu. Zostały zrealizowane testy interfejsu użytkownika, testy jednostkowe konkretnych serwisów, testy integracyjne całego systemu oraz testy manualne już gotowego prototypu. Testy interfejsu użytkownika zostały przeprowadzone na użytkownikach docelowych.

Praca opierała się na sekwencyjnym modelu wytwarzania oprogramowania, z elementami podejścia ewolucyjnego w fazie implementacji. Każdy etap był realizowany dopiero po zakończeniu poprzedzającego go etapu. Podczas fazy implementacji pojedynczej funkcji w warstwie serwerowej była również prowadzona praca równolegle nad wizualizacją tej funkcjonalności w warstwie klienckiej oraz połączeniem komunikacji między tymi warstwami. Taki sposób pracy eliminował błędy oraz umożliwiał rozwój prototypu o nowe pomysły.

2. Przegląd istniejących rozwiązań do zdalnej współpracy nad materiałem audio

Współczesny rynek oprogramowania oferuje szereg narzędzi wspierających pracę zdalną, jednak niewiele z nich zostało zaprojektowanych z myślą o specyficznych potrzebach inżynierii dźwięku i produkcji muzycznej. W niniejszym rozdziale dokonano analizy dostępnych rozwiązań, dzieląc je na systemy ogólnego przeznaczenia oraz dedykowane platformy do recenzji audio.

Analiza ta ma na celu wskazanie luk w funkcjonalnościach obecnych narzędzi, szczególnie w obszarze precyzyjnego komentowania osi czasu (waveform) oraz efektywnego zarządzania wersjami plików. Przegląd ten stanowi bezpośrednie uzasadnienie dla implementacji autorskiego rozwiązania, które łączy prostotę narzędzi chmurowych z precyzją wymaganą w profesjonalnej postprodukcji audio.

2.1. Metody komunikacji i wymiany plików w branży audio

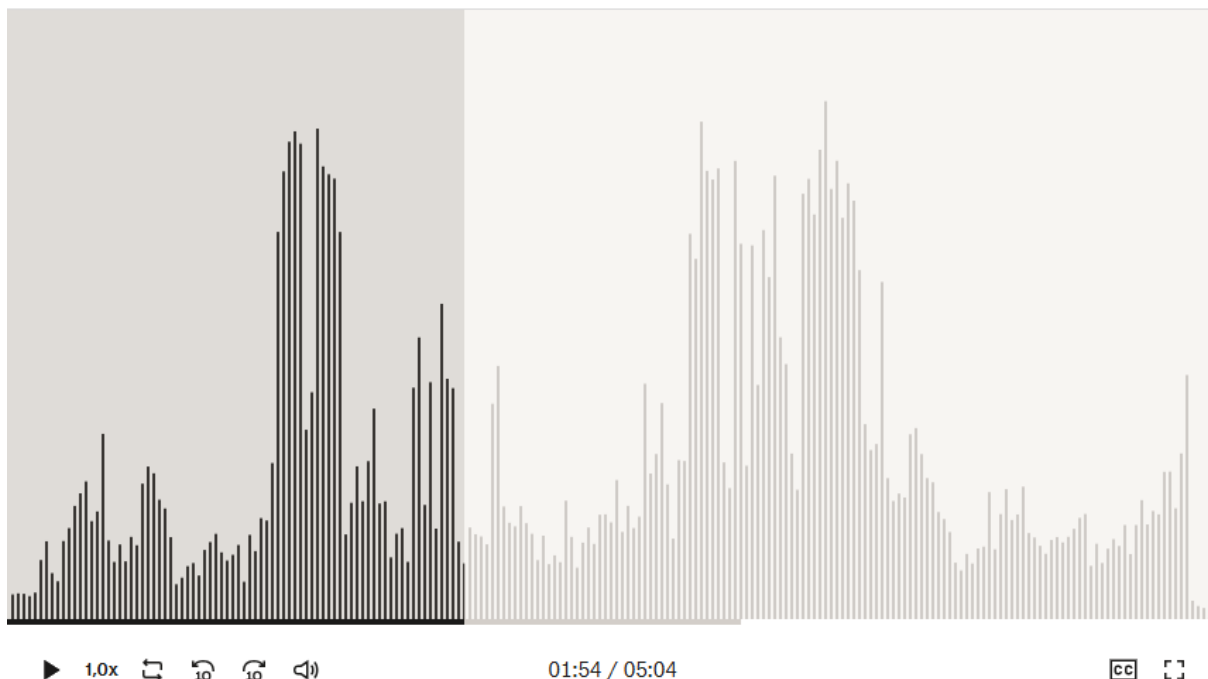
Dobór odpowiednich narzędzi ułatwiających i wspomagających wymianę informacji jest ważnym elementem podczas zdalnej współpracy. Narzędzie takie nie powinno w żadnym stopniu ograniczać komunikacji.

Standardowymi kanałami wymiany informacji stały się komunikatory tekstowe, takie jak Messenger [12], WhatsApp [13] czy Discord [14]. Ich główną zaletą jest szybkość przepływu informacji oraz powszechność użycia, co potwierdzają dane o ich adopcji w procesach zdalnej komunikacji [15]. Największą wadą natomiast są limity dotyczące przesyłania plików. Drugą wadą są nieintuicyjne oraz bardzo ograniczone wbudowane odtwarzacze, które udostępniają tylko bardzo podstawowe funkcje. Odtwarzacze te nadają się raczej do szybkiego podglądu niż do zaawansowanej analizy materiału dźwiękowego.

Rozwiązanie opierające współpracę na komunikatorach tekstowych zazwyczaj jest rozszerzane o dodatkowe narzędzia wspierające wymianę większych plików. Zazwyczaj są to dyski chmurowe, takie jak Google Drive czy Dropbox. Użytkownik sam może zarządzać przestrzenią dyskową, którą posiada i przeznaczona na współpracę. Platformy takie zazwyczaj posiadają też bardziej zaawansowane odtwarzacze dźwięku. Ich największą zaletą jest to, że użytkownik ma pełną kontrolę nad lokalizacją plików i ma poczucie ich bezpieczeństwa podczas korzystania z rozwiązań renomowanych firm takich.

Na szczególną uwagę zasługuje aplikacja Dropbox [2], którą przedstawia Rysunek 1. Udostępnia ona bowiem odtwarzacz, który posiada rozbudowany kontroler do nawigacji. Użytkownik ma możliwość kontroli głośności, zmiany prędkości odtwarzania (ratio) oraz bardzo użytecznego cofania się lub przeskakiwania do przodu o 10 sekund. Do dyspozycji jest również podstawowa wizualizacja audio, dzięki której można odróżnić głośniejsze fragmenty od cichszych. Nie jest to wizualizacja fali audio, lecz jedynie poglądowa amplituda głośności, ale do podstawowego rozróżnienia jest ona wystarczająca.

Połączenie komunikatora, gdzie użytkownicy wymieniają się opiniami o materiale audio, nad którym współpracują, z dyskiem chmurowym jest obecnie standardem w branży audio. Wykorzystuje ono narzędzia powszechnie dostępne oraz przyjazne użytkownikom. Głównym problemem jest jednak brak precyzji komentarzy spowodowany przez podstawowy licznik czasu ograniczony jedynie do sekund. Drugim dużym problemem jest konieczność manualnego wersjonowania plików i zarządzania nazwami. Takie działanie wymaga dużo uwagi przy udostępnianiu nowej wersji i prowadzenia skrupulatnej historii zmian. Niedokładność w tym działaniu i nieprecyzyjna komunikacja może prowadzić do sytuacji, w których różni członkowie zespołu słuchają różnych wersji z przeświadczeniem, że ta, której słuchają, jest tą najnowszą. Powszechnym problemem w tym sposobie pracy jest gubienie plików i informacji. Brak ustrukturyzowanej komunikacji prowadzi do rozproszenia informacji, co utrudnia odnalezienie kluczowych ustaleń oraz zwiększa ryzyko pomyłek w zarządzaniu plikami.



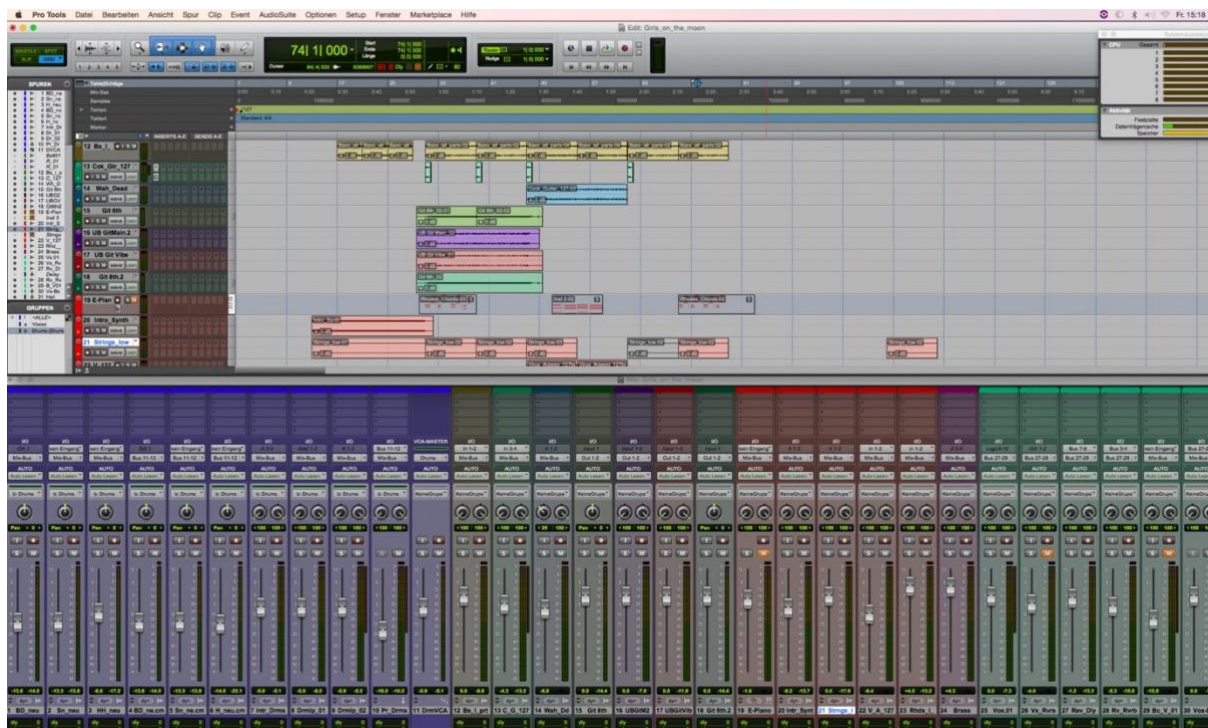
Rysunek 1 - Odtwarzacz Dropbox. Źródło: [2]

2.2. Przegląd dedykowanych platform do współpracy nad dźwiękiem

Zapotrzebowanie na narzędzia do zdalnej współpracy sprawiło, że zaczęły powstawać rozwiązania zastępujące standardowe dyski chmurowe. Narzędzia takie znacząco różnią się między sobą i oferują różne funkcjonalności. Skierowane są też do różnych grup odbiorców od profesjonalnych muzyków, aż po amatorów hobbystycznie zajmujących się dźwiękiem. Powoli zyskują one popularność, ale ze względu na różnice między nimi oraz wady każdego rozwiązania, proces ten jest powolny i wiele osób niechętnie podchodzi do takiej zmiany.

2.2.1 Avid Cloud Collaboration

Firma Avid jest producentem oprogramowania DAW o nazwie *Pro Tools* (Rysunek 2), które od lat jest standardem w branży audio [16]. W wersji 2024.10 została zintegrowana funkcja współpracy chmurowej (*Cloud Collaboration*) [17]. Umożliwia ona udostępnianie sesji nagraniowej praktycznie w czasie rzeczywistym. Z racji integracji z DAW *Pro Tools* użytkownicy mają do dyspozycji wszystkie funkcje tego oprogramowania. Kluczowymi dla zagadnienia omawianego w tej pracy są precyzyjne markery, do których użytkownik może dodać komentarz dla konkretnego punktu w czasie, oraz zaawansowana nawigacja. Szczególnie funkcja pre-roll która pozwala na ustawienie automatycznego odtwarzania materiału przed kursorem odtwarzania oraz zapętlenie zaznaczonego fragmentu.



Rysunek 2 - Odtwarzacz Pro Tools. Źródło: [18]

Mimo obecności tego rozwiązania w środowisku Avid od 2016 roku (przed wersją Pro Tools 2024.10 wymagało ręcznej integracji), nie przyjęło się ono na szeroką skalę. Jednym z głównych powodów takiego stanu rzeczy jest wysoka cena, którą musi ponieść każdy użytkownik. Mimo że funkcja *Cloud Collaboration* (Rysunek 3) jest dostępna jako standardowy element oprogramowania *Pro Tools*, to sam koszt zakupu tego oprogramowania jest barierą nie do przejścia i często zbędnym wydatkiem dla osób pracujących, na przykład, jako lektor lub muzyk sesyjny. Dodatkowo dostęp do takiej sesji użytkownik ma tylko na komputerze z racji braku mobilnej wersji oprogramowania *Pro Tools*. Biorąc pod uwagę dużą różnorodność stacji roboczych DAW oraz indywidualne preferencje użytkowników, ograniczanie procesu współpracy wyłącznie do środowiska *Pro Tools* staje się istotną barierą technologiczną.



Rysunek 3 - Panel zarządzania sesją Cloud Collaboration. Źródło: [19]

Opinie w internecie bywają krytyczne w stosunku do tego narzędzia. Na forum [20] AudioPost na Reddit.com można przeczytać, że zamysł działania tej funkcji jest idealny, ale w rzeczywistości bywa niestabilne i może powodować zawieszanie się programu.

2.2.2 Opusonix

Opusonix (Rysunek 4) jest stosunkowo nowym rozwiązaniem. Zostało zaprezentowane przez firmę NOTES 17 LLC w lipcu 2025. Aplikacja zyskała szerszy rozgłos w styczniu 2026 po publikacji artykułu [21], w którym została omówiona. Oferuje ono wiele funkcji wspierających ocenę materiału dźwiękowego. Warte uwagi jest wizualizacja audio oraz system dodawania komentarzy. Jest on intuicyjny i bardzo przypomina ten znany z oprogramowania DAW, które osoby pracujące z audio mają opanowane. Dużą zaletą w porównaniu do rozwiązania *Avid Cloud Collaboration* jest możliwość

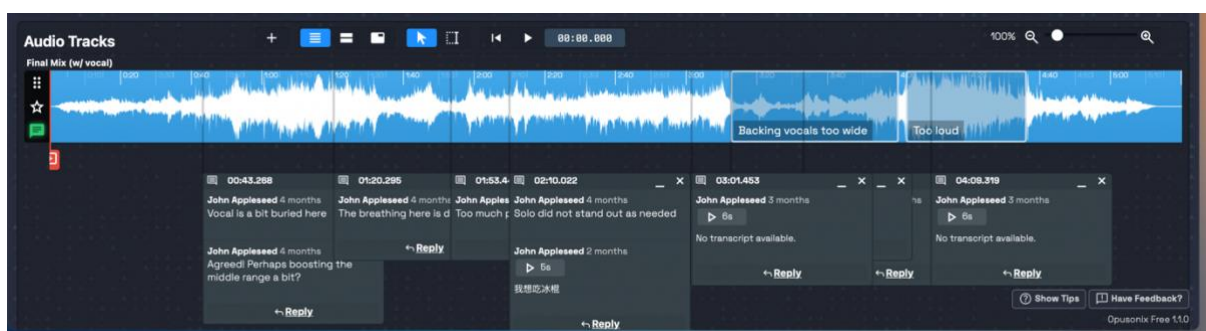
określenia fragmentu, do którego można dodać komentarz. Udostępnianie projektów odbywa się przez dodanie użytkowników do projektu. Wymaga to posiadania konta w usłudze *Opusonix* oraz zainstalowanej aplikacji przez wszystkie osoby mające pracować nad danym projektem.

Aplikacja ta, w przeciwieństwie do *Pro Tools* operuje na wygenerowanych plikach audio. Takie rozwiązanie eliminuje ograniczenie korzystania tylko z konkretnego DAW. Reżyser dźwięku przygotowuje plik w swoim ulubionym programie, a następnie udostępnia go przez *Opusonix*. Zarządzanie projektami jest intuicyjne i umożliwia użytkownikom grupowanie projektów w albumy.

Głównym ograniczeniem rozwiązania *Opusonix* jest brak integracji z zewnętrznymi dostawcami przestrzeni dyskowej. Pliki przechowywane są wyłącznie na wewnętrznych serwerach producenta, co odbiera użytkownikowi pełną kontrolę nad zarządzaniem własnymi zasobami i wymusza opłacanie wyższych planów subskrypcyjnych w przypadku projektów o dużej objętości. W przeciwieństwie do rozwiązań integrujących się np. z *Google Drive*, użytkownik jest tu uzależniony od infrastruktury jednej firmy.

Mimo charakteru *web-based*, aplikacja wykazuje wysokie zapotrzebowanie na zasoby sprzętowe, co przy rozbudowanych projektach z dużą liczbą ścieżek może prowadzić do spadków wydajności w przeglądarce. Ponadto, brak dedykowanej aplikacji mobilnej ogranicza możliwość szybkiego odsłuchu i akceptacji materiału "w biegu", co jest standardem w nowoczesnych procesach produkcyjnych. Interfejs, choć bogaty w funkcje, charakteryzuje się wysokim progiem wejścia – duża liczba komponentów sprawia, że nowi użytkownicy mogą potrzebować dodatkowego czasu na naukę obsługi systemu.

Opusonix bazuje na modelu subskrypcyjnym SaaS. Oferuje darmowy plan ograniczony do 1GB przestrzeni dyskowej i 10 projektów. Dostępny jest też plan edukacyjny za 2,99\$ gdzie do dyspozycji jest 50GB przestrzeni i 50 projektów, oraz plan pro za 9,99\$ gdzie do dyspozycji jest 200GB przestrzeni dyskowej i 500 projektów [22].



Rysunek 4 - Odtwarzacz Opusonix. Źródło [23]

2.2.3 Dropbox Replay

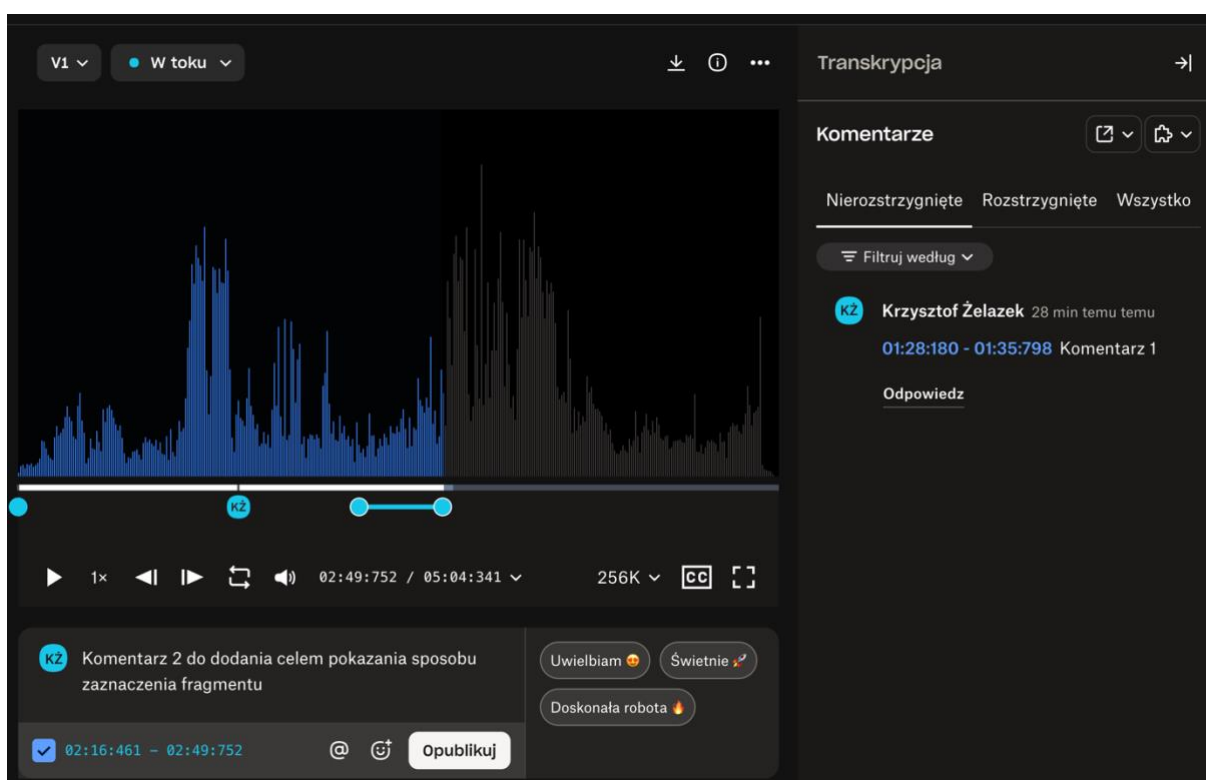
Firma Dropbox zauważyła specyficzne potrzeby użytkowników pracujących kreatywnie i stworzyła rozszerzenie swojego produktu o nazwie *Dropbox Replay* [24]. Jest to aplikacja szczególnie skierowana do użytku z wideo, ale udostępnia ona również funkcje przydatne podczas pracy z plikami audio. Narzędzie to zostało zaprezentowane w roku 2023, będąc odpowiedzią na zapotrzebowanie rynku. Dla zastosowań audio posiada wszystkie atuty udostępnione przez Dropbox, omówione w podrozdziale 2.1, rozszerzone o dodatkowe funkcje specyficzne do pracy nad multimediami. Dodatkowe funkcjonalności obrazuje Rysunek 5. Udostępnianie odbywa się przez dodanie innych użytkowników do projektu. Wymaga ono jednak posiadania przez nich konta Dropbox.

Główną zaletą tej aplikacji jest integracja z dyskiem Dropbox. Dzięki temu pliki są przetrzymywane na renomowanych serwerach zapewniających prywatność i bezpieczeństwo. Głównym rozszerzeniem w porównaniu do wbudowanego odtwarzacza jest system komentarzy, które można zamieszczać na osi czasu. Komentarz może być jednym punktem lub pokrywać zaznaczony przedział czasu. *Replay* posiada wbudowany system wersjonowania, co znacząco ułatwia organizację projektu.

Kolejną cechą wartą uwagi jest opcja zmiany statusu projektu. Użytkownicy mogą wybrać spośród dostępnych stanów: do sprawdzenia, w toku, do poprawy, zatwierdzono lub bez wybranego stanu.

Do wad tego rozwiązania należy jego główne przeznaczenie, czyli praca z wideo. Powoduje to, że narzędzie posiada narzucone pewne cechy, które są standardem w branży filmowej, ale niekoniecznie dobrze przekładają się na sferę audio. Przykładem takiej cechy jest sposób dodawania komentarza. Wymaga on zaznaczenia odpowiedniego momentu na małym pasku odtwarzania. Dodatkowo przed dodaniem zakresu komentarza trzeba już dodać tekst komentarza. Rozwiązanie takie jest nieintuicyjne i wymaga odnalezienia tej funkcji przez użytkownika. Z wad można też wymienić obecność tylko ciemnego motywu.

Replay, jako rozszerzenie do Dropbox, oferuje darmowy plan (ograniczenie do 4 plików). Chcąc korzystać z niego w pełnym zakresie, konieczne jest dokupienie tego narzędzia w modelu subskrypcyjnym. Nie jest on częścią podstawowych planów chmurowych Dropbox, a stanowi ich rozszerzenie, co wiąże się z podniesieniem kosztów użytkowania.



Rysunek 5 - Odtwarzacz Dropbox Replay. Źródło: [24].

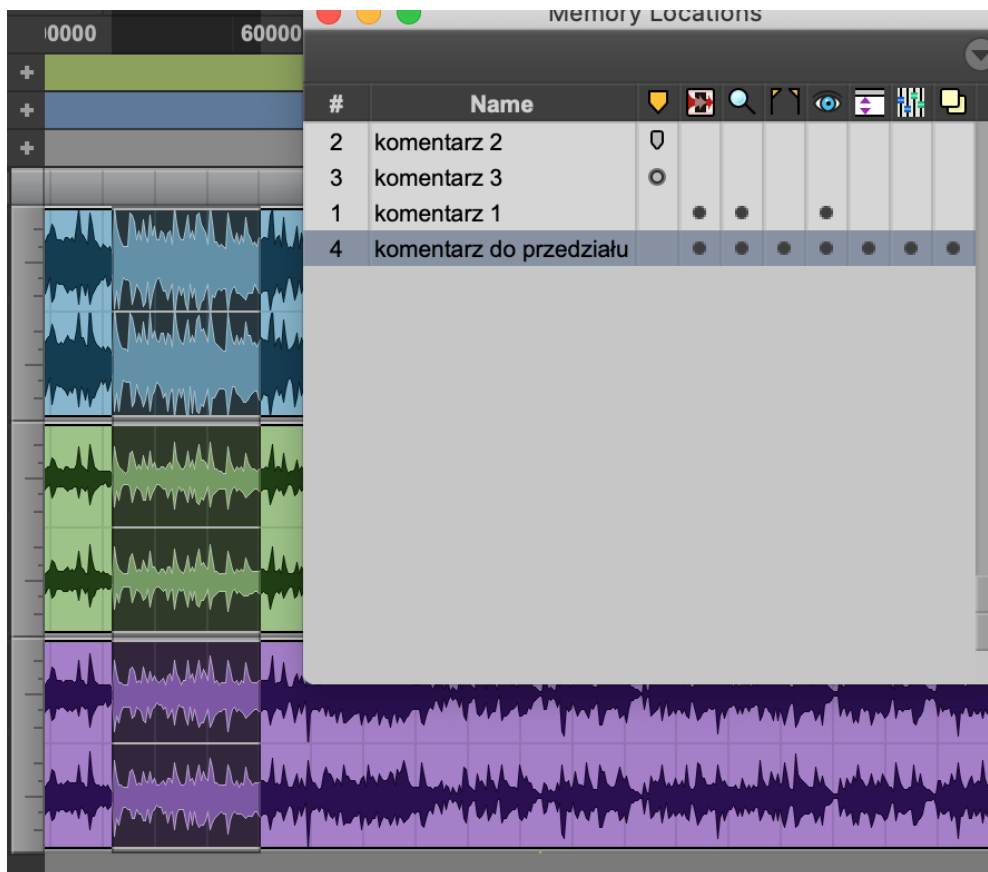
2.3. Analiza mechanizmów oznaczania czasu i wersjonowania

Każde z przedstawionych rozwiązań realizuje wymogi wersjonowania oraz dodawania komentarzy z precyzyjnym odnośnikiem do osi czasu, ale robi to w inny sposób i z różną precyzją. Użytkownik ma wybór mechanizmów realizujących to samo wymaganie funkcjonalne, ale w odmienny sposób.

Sposób podstawowy, wykorzystujący komunikatory tekstowe rozszerzone o rozwiązania chmurowe, wymaga od użytkowników dużej precyzji. Osoby korzystające z takiego modelu pracy są zmuszone do samodzielnego prowadzenia historii wersji, pilnowania odpowiednich nazw i numeracji plików oraz ręcznego zapisywania odnośników w czasie.

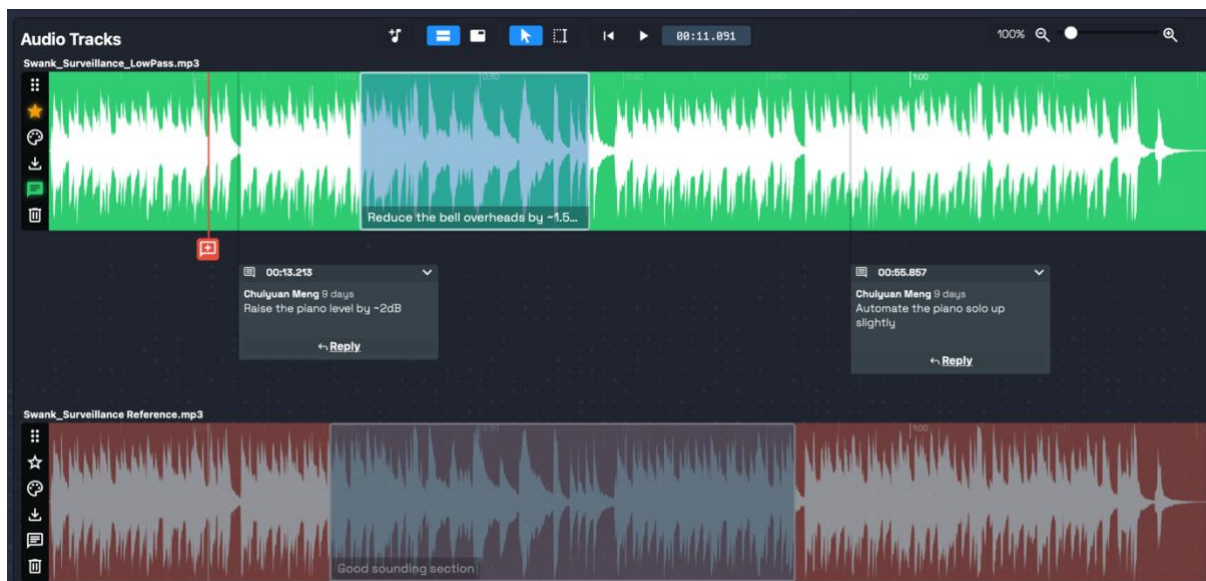
Avid Cloud Collaboration umożliwia precyzyjne dodanie komentarza do wybranego punktu na osi czasu lub do zaznaczonego fragmentu. Na osi czasu są jednak widoczne tylko punktowe komentarze.

Użytkownik ma też możliwość szybkiego przejścia do konkretnego wpisu poprzez wybranie go na liście wszystkich komentarzy (Rysunek 6). Automatyczne wersjonowanie wymaga włączenia w ustawieniach. Z racji tego, że *Pro Tools* służy do pracy z nagraniami wielośladowymi, nie jest to łatwo dostępne i intuicyjne. Działa to na zasadzie tworzenia automatycznych kopii zapasowych co ustalony czas. Plik taki zapisuje wszystkie zmiany wprowadzone w projekcie.



Rysunek 6 - Lista Markerów Pro Tools. Źródło: [16]

Precyzyjne dodawanie komentarzy i wersjonowanie plików to główne funkcje udostępnione przez *Opusonix*. Użytkownik, w celu oznaczenia punktu na osi czasu podczas odtwarzania pliku, klika przycisk celem dodania jednego punktu lub trzyma przycisk przez pewien czas, żeby stworzyć zakres, a następnie wpisuje swoje uwagi. Wpisy te są potem widoczne dla wszystkich osób pracujących nad projektem. Komentarze są wyświetlane bezpośrednio pod konkretną wersją (Rysunek 7), a przy wystąpieniu bliskiej odległości, ich okna mogą nachodzić na siebie. Wersjonowanie odbywa się automatycznie. Nowa wersja dostaje kolejny numer. Wersje są wyświetlane jedna pod drugą, a użytkownicy mogą manualnie przestawiać kolejność wersji. Oznacza to, że manualnie można zamienić wersje numerami.



Rysunek 7 - Opusnix Widok główny. Źródło: [23]

Dropbox Replay posiada funkcję dodawania komentarzy zaimplementowaną na pasku odtwarzania. Aby stworzyć wpis dla pojedynczego punktu, wystarczy wpisać informacje w polu tekstowym, a znacznik na osi zostanie automatycznie dodany na podstawie momentu, w którym odtwarzanie zostało zatrzymane. Celem oznaczenia fragmentu, należy najechać na stworzony punkt myszką, a następnie trzymając przycisk SHIFT, przesunąć wybrany punkt do końca przedziału, który chcemy stworzyć. Wersjonowanie odbywa się automatycznie poprzez tworzenie etykiet typu „V1”. Nowo dodany plik automatycznie zyskuje największy numer i wskazuje na najwyższe miejsce na liście. Dzięki takiemu rozwiązaniu domyślnie odtwarzana jest zawsze najnowsza wersja.

2.4. Zestawienie porównawcze omawianych rozwiązań

Każde z omówionych rozwiązań oferuje inne mechanizmy dodawania komentarzy oraz wersjonowania. W celu zebrania tych informacji zostały stworzone dwie tabele, które porównują przedstawione rozwiązania dostępne na rynku. Tabela 1 przedstawia porównanie mechanizmów dodawania komentarzy oraz sposób wersjonowania. Tabela 2 ma na celu porównanie aplikacji pod kątem łatwości i dostępności oraz kosztu użycia każdej z nich. Zestawienia te mają na celu pokazanie, jakie rozwiązania są najczęściej stosowane w tego typu oprogramowaniu. Pozwalają one również określić luki funkcjonalne i procesowe istniejące w dostępnych rozwiązaniach.

Tabela 1 - Porównanie realizacji funkcji komentarzy i wersjonowania. Źródło: Opracowanie własne.

lp.	Narzędzie	Komentarze	Wersjonowanie
1	Komunikator + dysk chmurowy	Manualne tworzenie komentarza w formie tekstu zawierającego odniesienie w czasie.	Manualne
2	<i>Avid Cloud Collaboration</i>	Mechanizm dodawania punktowych i przedziałowych komentarzy. Brak wizualizacji na osi czasu przedziałowych wpisów.	Automatyczne (ograniczone)
3	<i>Opusonix</i>	Mechanizm dodawania punktowych i przedziałowych komentarzy. Możliwość dodania tylko podczas odtwarzania. Brak listy komentarzy.	Automatyczne
4	<i>Dropbox Reply</i>	Mechanizm dodawania punktowych i przedziałowych komentarzy. Możliwość dodania podczas odtwarzania lub po zatrzymaniu.	Automatyczne

Tabela 2 - Porównanie dostępności, łatwości oraz kosztów użycia. Źródło: Opracowanie własne.

lp.	Narzędzie	Dostępność i łatwość użycia	Koszt użycia
1	Komunikator + dysk chmurowy	Dostęp na urządzeniach mobilnych i komputerach. Intuicyjne w użyciu, ale wymaga uwagi i dużej ilości akcji użytkownika.	Darmowy/koszt chmury
2	<i>Avid Cloud Collaboration</i>	Wymaga instalacji na komputerze. Intuicyjny interfejs	Bardzo wysoki. Wymaga zakupu licencji lub subskrypcji.
3	<i>Opusonix</i>	Dostępny z poziomu przeglądarki na komputerze. Brak wersji mobilnej. Skomplikowany interfejs użytkownika.	Średni, wszystkie funkcje w obrębie jednej subskrypcji.
4	<i>Dropbox Reply</i>	Dostępny z poziomu przeglądarki na komputerze i smartfonie. Intuicyjny interfejs.	Duży, wymaga subskrypcji <i>Dropbox</i> oraz dodatkowej subskrypcji <i>dropbox reply</i> .

2.5. Podsumowanie analizy i identyfikacja luk funkcjonalnych

Przeprowadzony przegląd rozwiązań do zdalnej współpracy nad materiałem audio pozwala na sformułowanie wniosków dotyczących obecnego stanu rynku oraz zidentyfikowanie barier, z jakimi mierzą się użytkownicy. Mimo szerokiego wachlarza narzędzi — od prostych komunikatorów po zaawansowane systemy korporacyjne — żaden z analizowanych produktów nie oferuje optymalnego balansu między precyzją, kosztem a swobodą zarządzania danymi.

Na podstawie zestawień zawartych w Tabeli 1 i Tabeli 2 zidentyfikowano następujące **luki funkcjonalne i procesowe**:

- Brak integracji z zewnętrzną infrastrukturą chmurową: Większość dedykowanych platform (np. *Opusonix*) wymusza na użytkowniku przechowywanie plików na wewnętrznych serwerach producenta. Prowadzi to do zjawiska *vendor lock-in* oraz generuje dodatkowe koszty subskrypcyjne za przestrzeń dyskową, którą użytkownik często już posiada w ramach innych usług (np. *Google Drive*).
- Niski stopień optymalizacji interfejsu pod kątem audio: Rozwiązania ogólnego przeznaczenia (jak *Dropbox Replay*) są zorientowane na materiały wideo, co skutkuje mało intuicyjnym systemem oznaczania zakresów na osi czasu. Z kolei narzędzia profesjonalne (*Avid Cloud Collaboration*) cechują się wysokim progiem wejścia i nadmiarowością funkcji, co zniechęca mniej zaawansowanych klientów lub muzyków sesyjnych.
- Problemy z dostępnością i barierą finansową: Najbardziej precyzyjne narzędzia wymagają instalacji kosztownego, specjalistycznego oprogramowania DAW na komputerach stacjonarnych, co wyklucza szybką weryfikację materiału na urządzeniach mobilnych lub komputerach bez zainstalowanego środowiska produkcyjnego.
- Chaos w wersjonowaniu plików: Standardowe metody oparte na komunikatorach i dyskach chmurowych nie oferują natywnego mechanizmu zarządzania wersjami, co w praktyce prowadzi do błędów komunikacyjnych i pracy na nieaktualnych plikach.

3. Propozycja systemu do zdalnej współpracy nad materiałami audio

Niniejszy rozdział poświęcony jest fazie projektowej systemu „WaveScope”. Na podstawie wniosków wyciągniętych z analizy istniejących rozwiązań oraz zidentyfikowanych luk funkcjonalnych opracowano kompleksową propozycję narzędzia, które stanowi próbę optymalizacji procesów komunikacyjnych w branży audio. Głównym celem projektowym było stworzenie systemu, który łączy elastyczność wyboru infrastruktury chmurowej z precyzją wymaganą przy recenzowaniu ścieżek dźwiękowych.

Proces projektowy podzielono na etapy, które pozwalają na przejście od ogólnych założeń do szczegółowej architektury technicznej. W pierwszej kolejności sformułowano cele i założenia projektu, stanowiące fundament dla specyfikacji wymagań funkcjonalnych oraz niefunkcjonalnych. Dalsza część rozdziału skupia się na modelowaniu zachowań i struktury systemu z wykorzystaniem języka UML (*Unified Modeling Language*) [25]. Przygotowano diagramy przypadków użycia, obrazujące interakcje aktorów z systemem, oraz analityczny diagram klas, który następnie został rozwinięty do poziomu projektu technicznego. Proces ten pozwolił na precyzyjne odwzorowanie logiki biznesowej oraz struktury danych niezbędnych do poprawnej implementacji. Rozdział kończy prezentacja wstępnego projektu interfejsu użytkownika, który stanowi wizualną realizację założeń dotyczących intuicyjności i ergonomii pracy z materiałem audio.

3.1. Cele

Głównym celem niniejszej pracy jest zaprojektowanie i implementacja prototypu webowego systemu „WaveScope”, służącego do zdalnej współpracy nad materiałami audio, który umożliwi precyzyjną komunikację między realizatorem a klientem oraz zautomatyzuje proces wersjonowania plików. Aby zrealizować główny cel, wyznaczono następujące cele szczegółowe:

- **Analiza merytoryczna:** Identyfikacja kluczowych problemów komunikacyjnych w branży audio oraz analiza dostępnych rozwiązań pod kątem ich wad i zalet (przedstawiono w rozdziałach 1 oraz 2).
- **Projektowanie architektury:** Opracowanie struktury systemu opartej na modelu klient - serwer, uwzględniającej integrację z zewnętrznym API (Google Drive [1]).
- **Implementacja warstwy serwerowej:** Stworzenie stabilnego zaplecza odpowiedzialnego za logikę biznesową, zarządzanie projektami oraz komunikację z bazą danych.
- **Implementacja warstwy klienckiej:** Stworzenie responsywnego i intuicyjnego interfejsu z wykorzystaniem biblioteki do wizualizacji fali dźwiękowej.
- **Weryfikacja i testy:** Przeprowadzenie testów funkcjonalnych oraz użyteczności prototypu w celu potwierdzenia poprawności założeń projektowych.

3.2. Założenia i zakres projektu

Projekt systemu WaveScope opiera się na kilku kluczowych założeniach. Głównym z nich jest zapewnienie wysokiej dostępności aplikacji poprzez udostępnienie jej z poziomu przeglądarki internetowej. Rozwiązanie to eliminuje bariery sprzętowe i sprawia, że użytkownicy są bardziej skłonni do skorzystania z nowego narzędzia. Kolejnym fundamentem projektu jest wykorzystanie infrastruktury chmurowej (Google Drive), co ma na celu zapewnienie użytkownikowi pełnego poczucia kontroli nad swoimi plikami.

Ważnym elementem realizowanym w pracy jest projekt interfejsu użytkownika. Zakłada się, że będzie on responsywny oraz intuicyjny, co powinno pozwalać na swobodną pracę oraz sprzyjać skupieniu i kreatywności podczas oceny materiałów audio. Jednocześnie system ma być w pełni niezależny od oprogramowania DAW używanego przez realizatora. Komunikacja odbywa się na

poziomie wyeksportowanych plików audio w powszechnie stosowanych formatach (np. WAV, MP3), co zapewnia swobodę doboru narzędzi pracy. Interfejs będzie opracowany w języku angielskim, co umożliwi korzystanie z systemu w środowisku międzynarodowym.

W kwestiach technicznych projekt zakłada wykorzystanie bezpiecznych protokołów autoryzacji (OAuth 2.0) do komunikacji z zewnętrznymi dostawcami chmurowymi [26]. Dane użytkownika oraz dostęp do plików mają być chronione, a system nie będzie przechowywać haseł do zewnętrznych kont użytkowników.

System opiera się na modelu współpracy asynchronicznej, w którym komunikacja odbywa się poprzez pozostawianie precyzyjnych komentarzy na osi czasu. Z założenia WaveScope nie jest jedynym kanałem komunikacji i nie zastępuje w pełni tradycyjnych komunikatorów. Jego zadaniem jest przejęcie obowiązków dotyczących zarządzania projektami oraz ustrukturyzowanie wymiany informacji wynikającej z odniesień do konkretnych momentów w nagraniu. Przedstawiony system należy zatem rozumieć jako narzędzie wspierające współpracę, a nie jako kompletny system umożliwiający pełną realizację zdalnej interakcji nad materiałem dźwiękowym.

3.3. Określenie wymagań funkcjonalnych i нефункциональных

Zdefiniowanie celów pracy oraz założeń projektowych umożliwiło podjęcie dalszych prac nad systemem. Kolejnym krytycznym etapem procesu wytwórczego było precyzyjne określenie wymagań funkcjonalnych i нефункциональных. Proces ten pozwala na przełożenie ogólnej wizji produktu na zestaw konkretnych, weryfikowalnych kryteriów technicznych, które stanowią fundament dla późniejszego modelowania architektury i implementacji kodu źródłowego.

Wymagania funkcjonalne opisują bezpośrednie zachowanie systemu — definiują one konkretne operacje, usługi i zadania, które aplikacja musi realizować w odpowiedzi na interakcje użytkownika. Z kolei wymagania нефункциональные, często określane jako atrybuty jakościowe, definiują ograniczenia systemowe oraz standardy dotyczące wydajności, bezpieczeństwa, niezawodności czy użyteczności. Poniżej przedstawiono szczegółową specyfikację obu grup wymagań, dostosowaną do specyfiki pracy z materiałem audio w środowisku rozproszonym.

3.3.1 Wymagania funkcjonalne

- System zapewnia wersjonowanie plików audio.
- System zapewnia odtwarzanie pliku dźwiękowego.
- System automatycznie nadaje etykiety wersjom plików (V1, V2).
- Użytkownik może dodać komentarz do konkretnego momentu w nagraniu.
- Użytkownik może dodać komentarz do zaznaczonego przedziału czasu.
- System zapewnia wizualną synchronizację listy komentarzy z odpowiadającymi im punktami lub zakresami na fali dźwiękowej.
- System umożliwia wizualizację fali dźwiękowej.
- System pokazuje licznik czasu odtwarzania z dokładnością do setnych sekundy w formacie MM:SS:ms.
- System umożliwia zmianę szybkości oraz głośności odtwarzania.
- System zapewnia możliwość katalogowania projektów w albumy.
- System umożliwia dodanie własnej grafiki jako okładki do albumu.
- System zapewnia możliwość wyboru statusu projektu (w trakcie pracy, zatwierdzony).
- System daje możliwość tworzenia zespołów produkcyjnych.
- System umożliwia członkom zespołu pobranie konkretnego pliku dźwiękowego.

- System udostępnia wersję demonstracyjną swoich możliwości niezalogowanemu użytkownikowi, oferuje ona możliwość zaznajomienia z interfejsem (brak możliwości wgrania swoich plików).
- System umożliwia edycję wprowadzonych danych tekstowych, takich jak nazwy i opisy.

3.3.2 Wymagania niefunkcjonalne

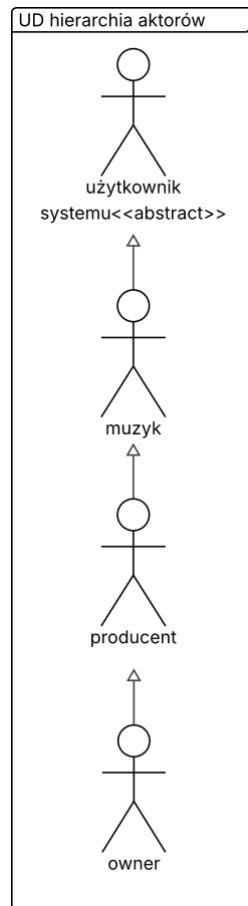
- Autoryzacja zewnętrzna realizowana przez OAuth 2.0 [26].
- Wymiana tokenów uwierzytelniających realizowana przez mechanizm HttpOnly Cookies w celu ochrony przed atakami typu XSS.
- Izolacja danych – system zapewnia dostęp do projektów wyłącznie uprawnionym użytkownikom.
- Intuicyjność nawigacji – główne funkcje dostępne za pomocą skrótów klawiszowych.
- Całość interfejsu musi być realizowana konsekwentnie w języku angielskim.
- Dostępność przeglądarkowa – pełna funkcjonalność na najnowszych wersjach Google Chrome [27].
- System zapewnia integralność relacji między projektami, ich wersjami a komentarzami.
- Modułowość – architektura systemu oparta na wzorcu warstwowym (Controller-Service-Repository), co zapewnia separację logiki biznesowej od warstwy dostępu do danych i prezentacji.
- Czas renderowania wizualizacji – system powinien wygenerować i wyświetlić wizualizację audio (*waveform*) dla pliku mp3 o długości do 5 minut w czasie nieprzekraczającym 3 sekund od momentu pobrania danych.

3.4. Zdefiniowanie i projekt diagramu przypadków użycia

Kolejnym etapem modelowania systemu WaveScope jest opracowanie diagramu przypadków użycia (ang. *Use Case Diagram*). Stanowi on graficzną reprezentację interakcji zachodzących pomiędzy użytkownikami (aktorami) a systemem, pozwalając na precyzyjne określenie granic aplikacji oraz zakresu jej funkcjonalności. Diagram ten stanowi pomost pomiędzy wymaganiami funkcjonalnymi opisanymi w podrozdziale 3.3, a faktyczną architekturą systemu.

W projekcie przyjęto hierarchiczną strukturę aktorów, co odzwierciedla realne role występujące w procesie produkcji audio. Aktor abstrakcyjny — Użytkownik systemu — definiuje podstawowe interakcje, takie jak autoryzacja, które są dziedziczone przez bardziej wyspecjalizowane role: Muzyka, Producenta oraz Właściciela (Owner). Taka struktura pozwala na czytelne przedstawienie gradacji uprawnień.

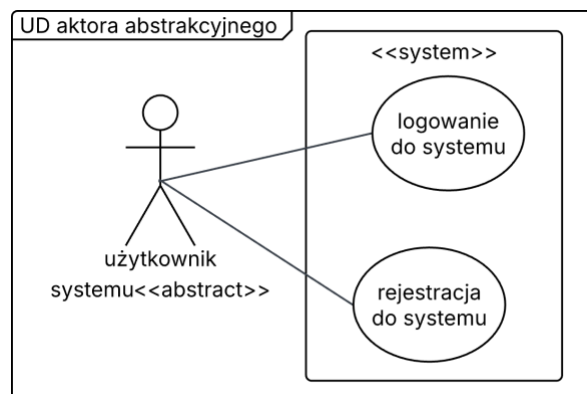
Na diagramie uwzględniono również relacje typu `<<include>>` oraz `<<extend>>`. Relacje `<<include>>` wskazują na operacje niezbędne do zakończenia danego przypadku użycia (np. weryfikacja statusu przy zatwierdzaniu utworu), natomiast relacje `<<extend>>` opisują funkcjonalności opcjonalne, które rozszerzają działanie podstawowych procesów, takie jak dodawanie pliku do konkretnego projektu.



Rysunek 8 - Hierarchia aktorów. Źródło: Opracowanie własne.

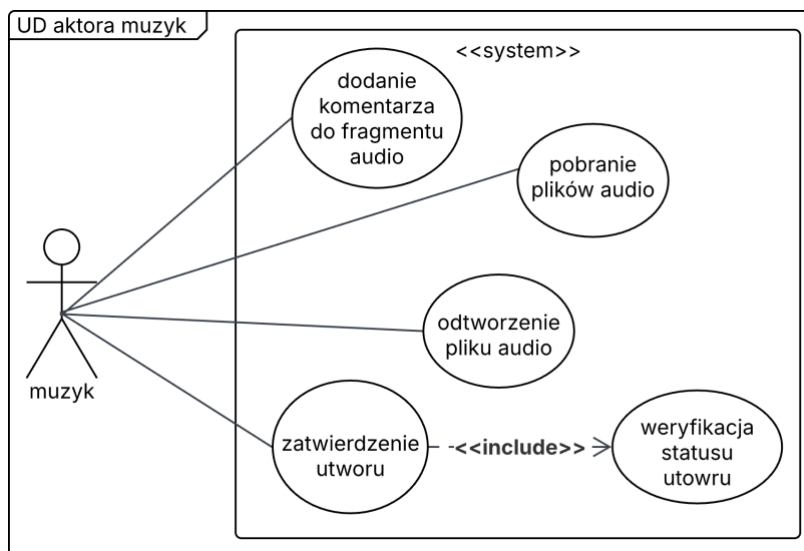
Struktura aktorów przedstawiona na Rysunek 8 - Hierarchia aktorów, została zaprojektowana w sposób hierarchiczny, co pozwala na precyzyjne zarządzanie uprawnieniami w systemie. Zastosowanie relacji generalizacji (dziedziczenia) pozwala na uniknięcie powtórzeń i czytelne wskazanie zakresu odpowiedzialności poszczególnych użytkowników:

- Użytkownik systemu <<abstract>> – jest to aktor abstrakcyjny, który nie występuje w systemie samodzielnie. Definiuje on zestaw podstawowych funkcjonalności, które przedstawia Rysunek 9 - Przypadki użycia dla aktora abstrakcyjnego, takich jak rejestracja i logowanie, które są niezbędne dla każdego zalogowanego uczestnika projektu.



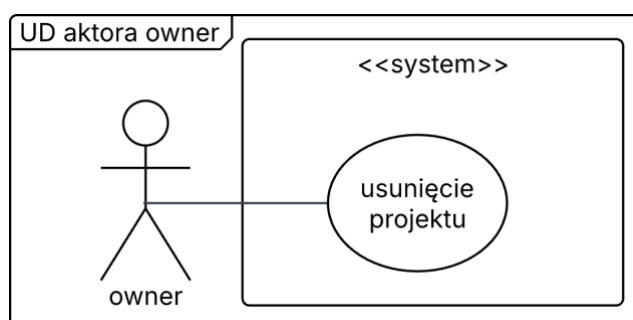
Rysunek 9 - Przypadki użycia dla aktora abstrakcyjnego. Źródło: Opracowanie własne.

- Muzyk – dziedziczy uprawnienia od użytkownika podstawowego. Jego rola skupia się na merytorycznej pracy z materiałem audio. Posiada on uprawnienia do odtwarzania plików, ich pobierania oraz kluczową z punktu widzenia systemu funkcję dodawania precyzyjnych komentarzy do osi czasu. Zbiór akcji dostępnych dla użytkownika Muzyk obrazuje Rysunek 10 - Przypadki użycia dla aktora Muzyk.

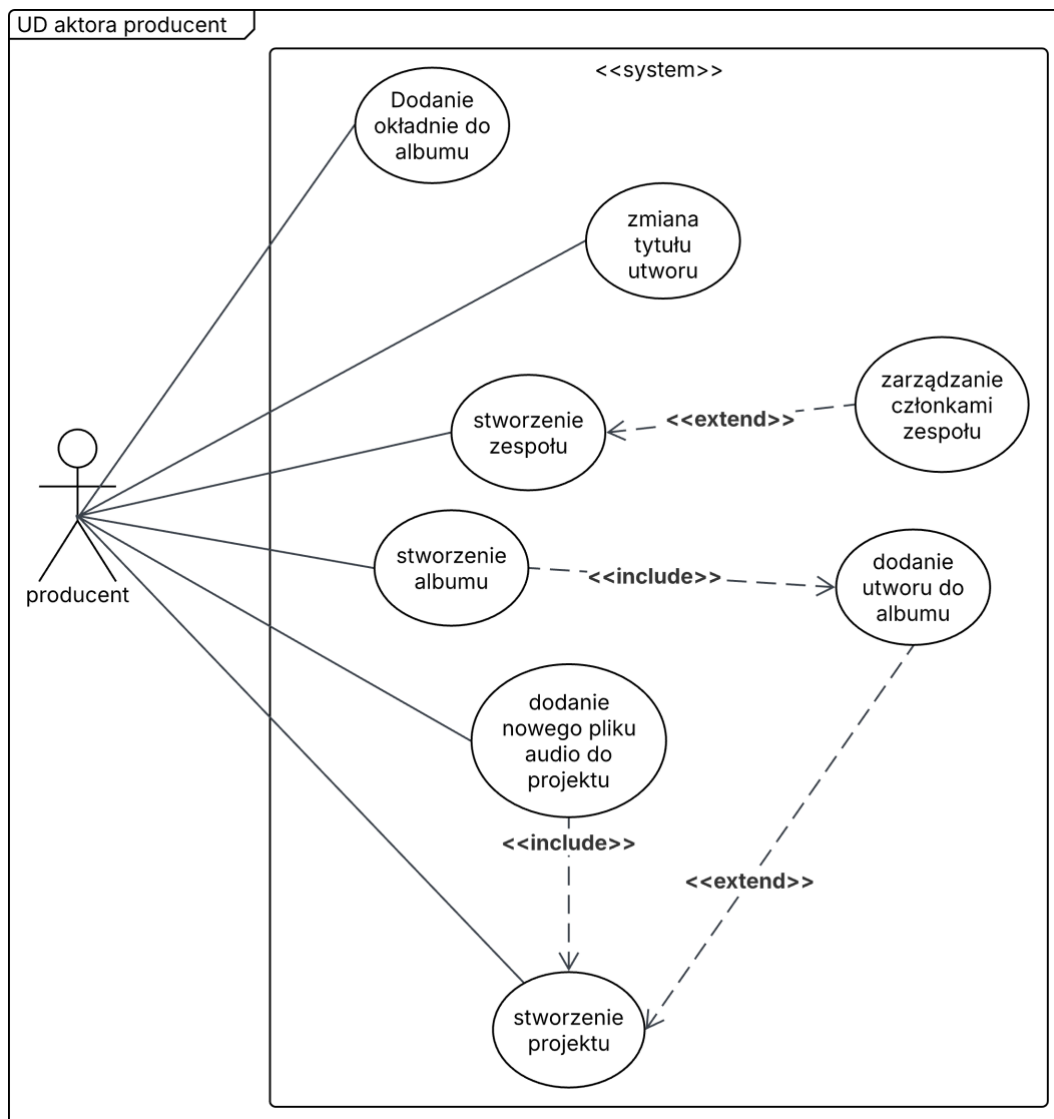


Rysunek 10 - Przypadki użycia dla aktora Muzyk. Źródło: Opracowanie własne.

- Producent – rozszerza uprawnienia Muzyka o funkcje zarządcze, które przedstawia Rysunek 12 - Przypadki użycia dla aktora producent. Producent jest odpowiedzialny za organizację pracy: tworzenie zespołów produkcyjnych, zarządzanie ich członkami, zmianę statusów utworów oraz katalogowanie projektów w albumy. Dodatkowo Producent może stworzyć nowy projekt.
- Owner (Właściciel) – znajduje się na szczycie hierarchii uprawnień. Dziedziczy on wszystkie funkcjonalności Producenta, a dodatkowo posiada wyłączone uprawnienie do usuwania projektów z systemu, które przedstawia Rysunek 10 - Przypadki użycia dla aktora Muzyk. Właścicielem staje się każdy producent, który stworzy projekt.



Rysunek 11 - Przypadek użycia dla aktora Owner. Źródło: Opracowanie własne.



Rysunek 12 - Przypadki użycia dla aktora producent. Źródło: Opracowanie własne.

Zastosowanie relacji typu `<<include>>` (np. przy zatwierdzaniu utworu) zapewnia integralność procesów biznesowych, wymuszając weryfikację statusu przed wykonaniem akcji. Z kolei relacje `<<extend>>` wskazują na opcjonalność działań, takich jak rozszerzenie procesu tworzenia projektu o natychmiastowe przyporządkowanie do albumu, co zwiększa elastyczność i ergonomię interfejsu.

3.5. Projekt diagramów klas dla systemu

Kolejnym kluczowym etapem projektowania systemu WaveScope jest opracowanie modelu klas. W procesie inżynierii oprogramowania model ten stanowi statyczny model struktury systemu, definiując klasy, ich atrybuty oraz wzajemne powiązania. Przejście od modelu przypadków użycia do diagramu klas pozwala na przełożenie wymagań funkcjonalnych na konkretną strukturę danych oraz logikę biznesową, która zostanie następnie zaimplementowana w warstwie serwerowej.

Zaprojektowana struktura odzwierciedla domenę pracy nad materiałem dźwiękowym, uwzględniając złożone relacje pomiędzy użytkownikami a procesem twórczym. Główną osią systemu jest klasa *Osoba*, która poprzez *Członkostwo* w zespole wchodzi w interakcje z konkretnymi grupami roboczymi (*Zespół*). Struktura ta została zaprojektowana tak, aby wspierać asynchroniczną współpracę nad *Projektem*, który składa się z wielu *Wersji Utworu*.

Podczas prac nad projektem systemu opracowano dwa warianty diagramu obrazujące rozkład klas oraz ich powiązania. W pierwszej kolejności powstał diagram analityczny, który pokazuje, jak z biznesowego punktu widzenia powinien zostać zbudowany system. Następnie model ten został poddany transformacji do postaci implementacyjnej (projektowej). To właśnie ta wersja stała się bezpośrednią podstawą do wytworzenia bazy danych i logiki w prototypie systemu WaveScope. Projekt analitycznego diagramu klas.

3.5.1 Analityczny diagram klas

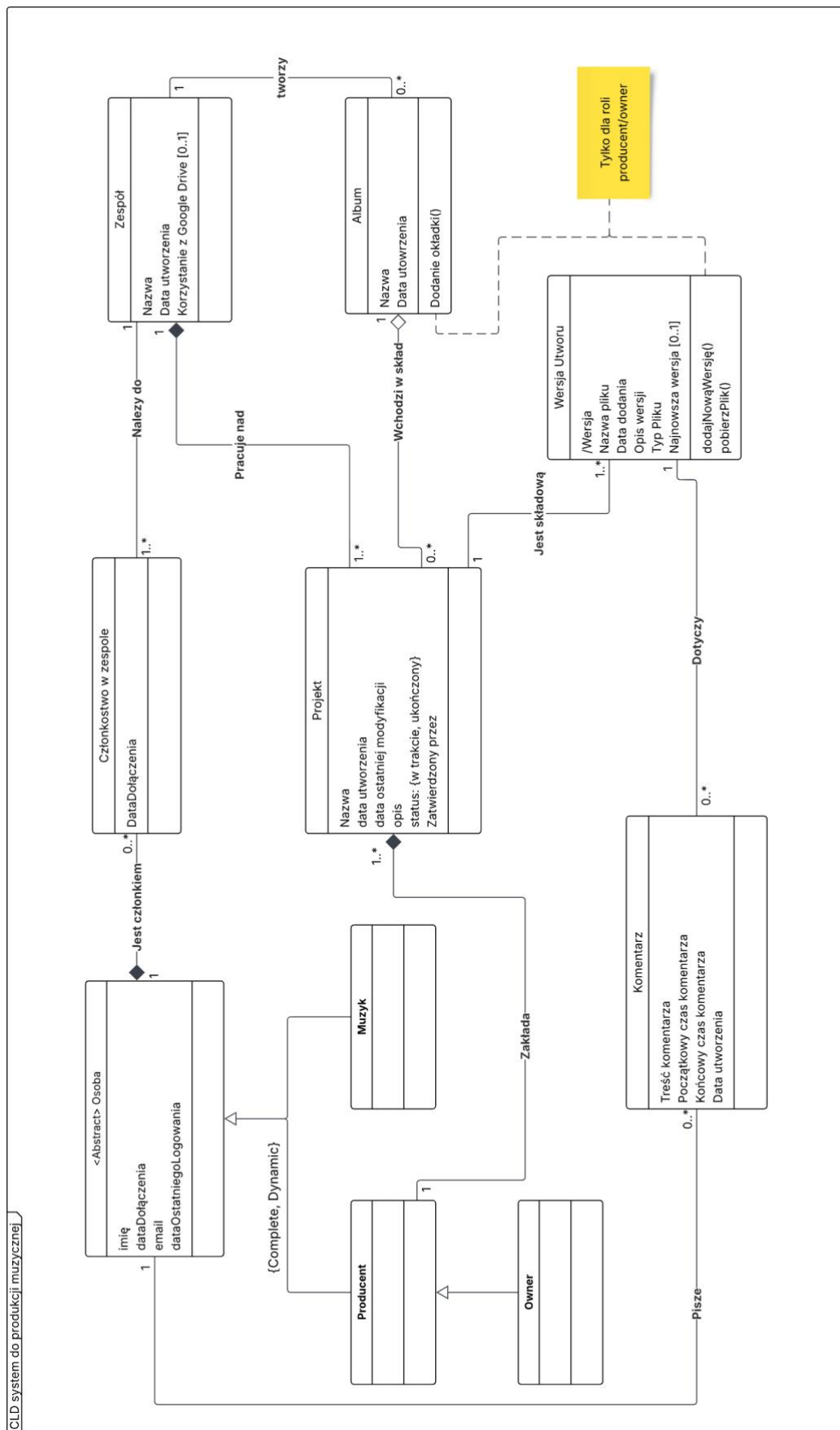
Na diagramie uwzględniono również mechanizmy wersjonowania oraz precyzyjnego komentowania osi czasu, co realizowane jest poprzez relację między wersjami a klasą *Komentarz*. Zastosowanie wzorców takich jak kompozycja (np. w relacji zespołu z projektami) oraz agregacja (w przypadku albumów) zapewnia integralność danych i logiczne uporządkowanie zasobów multimedialnych. Rysunek 13 - Analityczny diagram klas, przedstawia pełny model analityczny klas systemu WaveScope.

Analiza przedstawionego diagramu klas pozwala na wyróżnienie czterech kluczowych obszarów funkcjonalnych systemu, które odpowiadają za zarządzanie tożsamością, strukturę projektową, treściami audio oraz systemem recenzji.

Fundamentem systemu jest abstrakcyjna klasa *Osoba*, przechowująca podstawowe dane profilowe oraz informacje o aktywności użytkownika. Jest ona wzorcem dla klas: *Owner*, *Producent* i *Muzyk*, reprezentujących poszczególne role, które może pełnić użytkownik w systemie. Dziedziczenie dynamiczne wskazuje, że jeden użytkownik może płynnie przechodzić pomiędzy rolami podczas korzystania z systemu. Relacja z klasą *Członkostwo* w zespole realizuje model uprawnień oparty na rolach zdefiniowanych w dziedziczeniu po klasie abstrakcyjnej *Osoba*. Wprowadzenie klasy pośredniczącej pozwala jednej osobie na przynależność do wielu zespołów z różnymi poziomami uprawnień.

Klasa *Zespół* stanowi centralny punkt zarządzania pracą. Posiada ona atrybut logiczny *Korzystanie z Google Drive*, który determinuje sposób składowania plików audio dla danej grupy roboczej. Zespoły mogą organizować swoją pracę poprzez *Albumy*, które pełnią funkcję katalogów dla projektów. Relacja między *Albumem* a *Projektem* to agregacja. Oznacza to, że projekt może istnieć samodzielnie, ale może też wchodzić w skład większej całości (*Albumu*). Klasa *Projekt* jest ściśle powiązana z *Zespołem* i zawiera dane dotyczące postępu prac, w tym statusu projektu oraz listę osób, które zatwierdziły projekt.

Kluczowym elementem systemu jest relacja kompozycji między klasą *Projekt* a *Wersja Utworu*. Oznacza to, że wersje utworu są integralną częścią projektu i nie mogą istnieć bez niego. Klasa *Wersja Utworu* zawiera atrybut wyliczalny *Wersja*, który automatycznie inkrementuje numer kolejnej rewizji materiału. Obiekt ten przechowuje też informacje o tym, czy dana wersja jest aktualna (*Najnowsza wersja*). Dzięki zastosowaniu takiego atrybutu możliwe było dodanie funkcjonalności odtwarzającej najnowszą wersję jako domyślną. Implementacja metod *dodajNowąWersję()* oraz *pobierzPlik()* bezpośrednio realizują wymagania funkcjonalne i precyzując, w odniesieniu do jakiej klasy zostały zdefiniowane.



Rysunek 13 - Analityczny diagram klas. Źródło: Opracowanie własne.

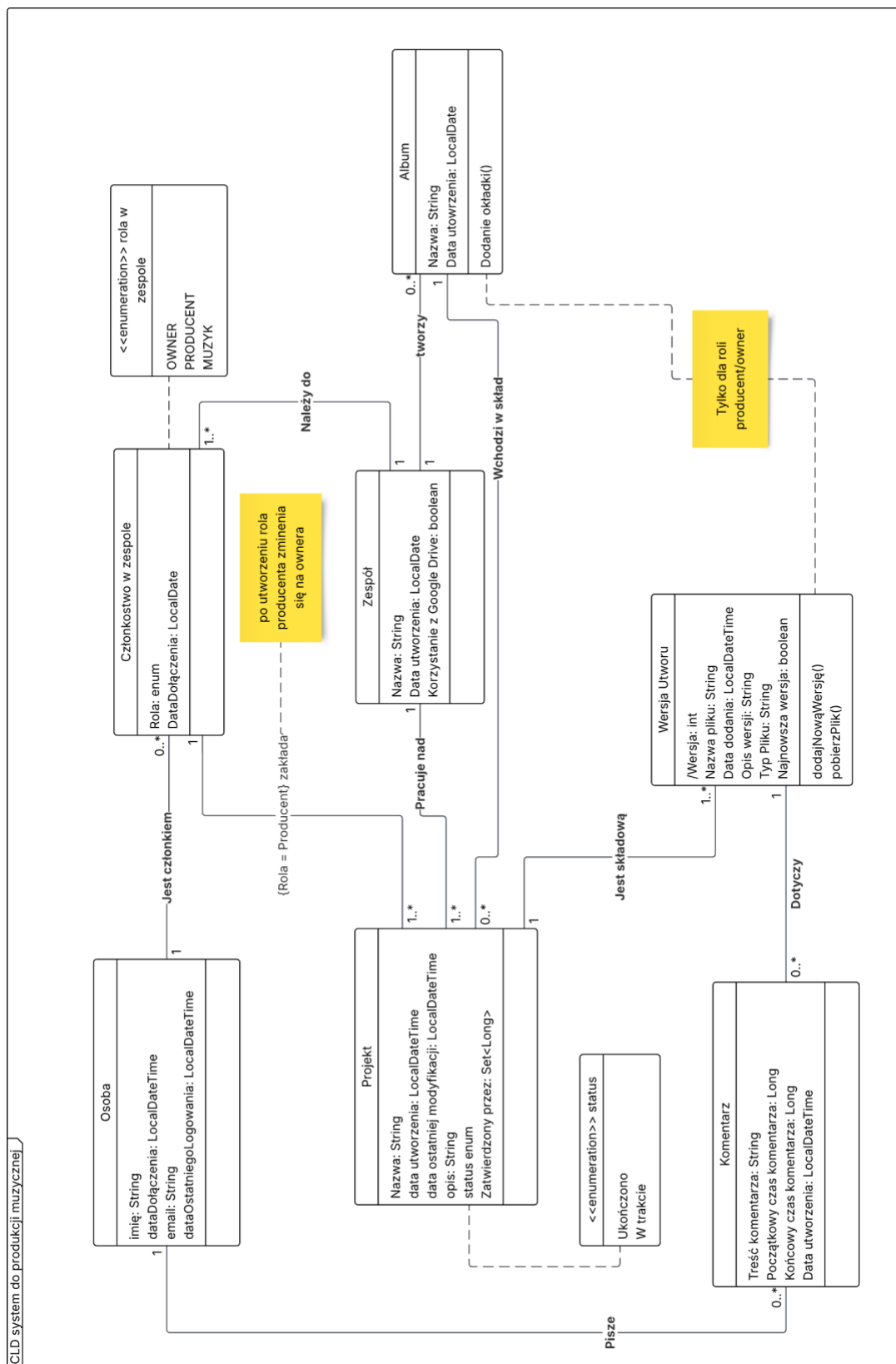
3.5.2 Projektowy diagram klas

Diagram projektowy (implementacyjny) stanowi bezpośrednie odwzorowanie założeń analitycznych na docelową architekturę bazy danych oraz kod aplikacji obiektowej. Główne zmiany względem modelu analitycznego wynikają z dobrych praktyk programistycznych oraz konieczności dostosowania struktury do możliwości technicznych frameworków mapowania relacyjno-obiektowego (ORM) użytych w prototypie (Hibernate). Rysunek 14 - Projektowy diagram klas przedstawia docelowy model, który posłużył jako bezpośrednia baza do implementacji prototypu systemu WaveScope.

Zasadniczą różnicą w stosunku do modelu analitycznego jest rezygnacja z paradygmatu dziedziczenia dla ról systemowych. Zastosowano tutaj dobrą praktykę inżynierii oprogramowania opisaną w książce „Java. Efektywne programowanie” [28], zwaną kompozycją zamiast dziedziczenia (*composition over inheritance*). Klasa `Osoba` stała się w pełni samodzielną encją, która przechowuje dane tożsamościowe. Dynamiczny charakter uprawnień, wcześniej modelowany dziedziczeniem, zrealizowano za pomocą klasy łączącej `Członkostwo` w zespole oraz nowego atrybutu opartego na typie wyliczeniowym (enum) `Rola` w zespole. Taka struktura idealnie odwzorowuje tabele w relacyjnej bazie danych i pozwala pojedynczemu użytkownikowi na pełnienie skrajnie różnych ról (np. `Ownera` w jednym zespole i `Muzyka` w drugim) bez naruszania struktury obiektu bazowego `Osoba`.

W modelu implementacyjnym zrezygnowano również z silnych relacji kompozycji oraz agregacji na rzecz standardowych asocjacji. Zmiana ta podyktowana jest faktem, że na poziomie kodu źródłowego oraz kluczy obcych w bazie danych relacje te są implementowane w niemal identyczny sposób – poprzez referencje do obiektów. Realizację kompozycji zapewnia odpowiednia walidacja wprowadzona w warstwie serwisów.

Podczas rozwijania diagramu analitycznego do postaci implementacyjnej zdefiniowano typy dla poszczególnych atrybutów. Oprócz domyślnych typów dostępnych w wielu językach programowania, takich jak: `String`, `Boolean` oraz `Integer` zostały przedstawione typy specyficzne dla języka Java. Przykładem takich typów są `LocalDate` oraz `LocalDateTime` reprezentujące kolejno datę oraz datę z czasem.



Rysunek 14 - Projektowy diagram klas. Źródło: Opracowanie własne.

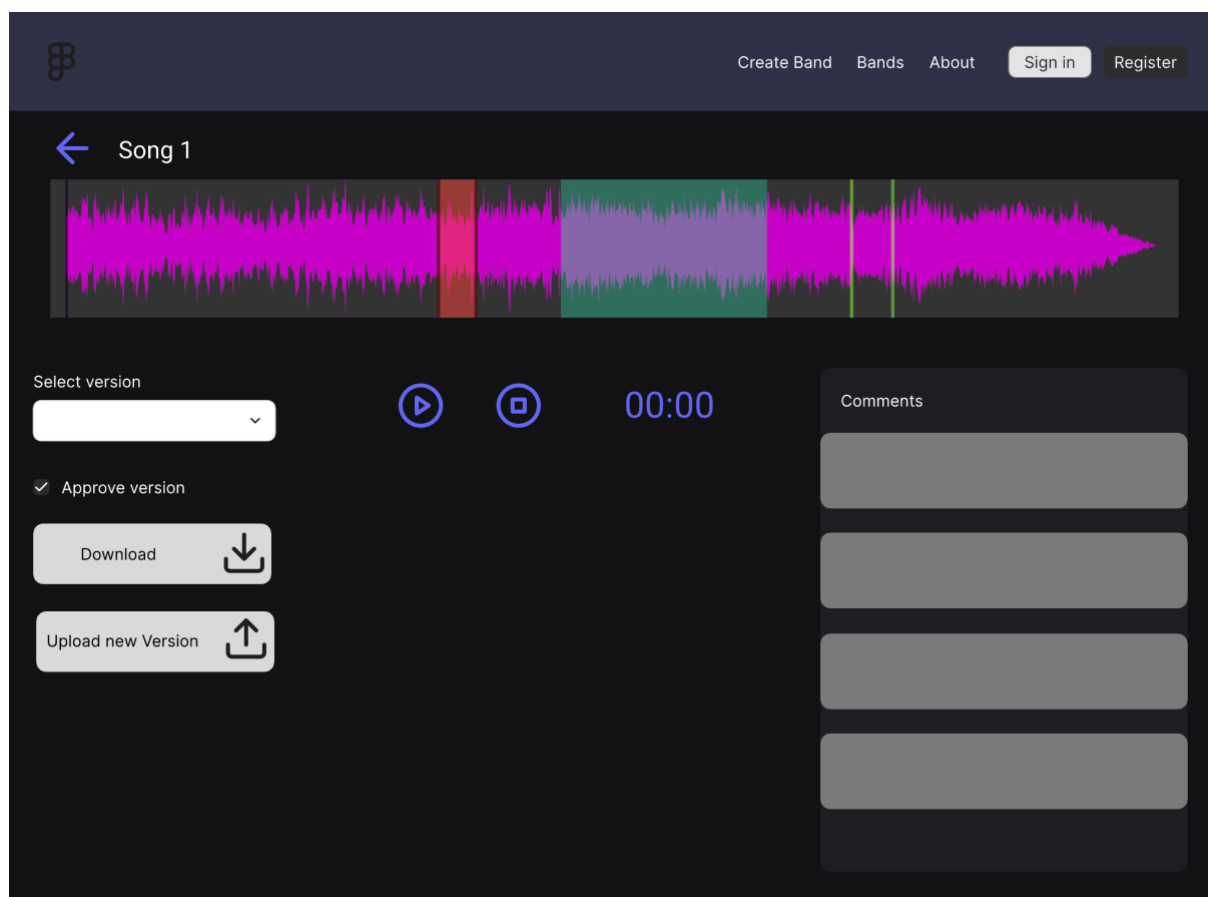
3.6. Wstępny projekt interfejsu użytkownika realizujący kluczowe funkcjonalności

Ostatnim etapem planowania i projektowania systemu WaveScope był projekt interfejsu użytkownika oraz test biblioteki wavesurfer.js pod kątem użyteczności dostarczanych funkcji. Diagram przypadków użycia przedstawiony w rozdziale 3.4 stał się podstawą do stworzenia tego projektu. Interfejs prezentuje fizyczne umiejscowienie przycisków oraz wyświetlanie informacji określonych na tym diagramie.

Podczas etapu projektowania interfejsu została definiowana warstwa kolorystyczna aplikacji. Głównym założeniem podczas doboru kolorów dla aplikacji było takie dobranie barw, które nie będą męczyły wzroku oraz będą umożliwiały komfortową pracę w ciemnym oświetleniu. W tym celu podjęta została decyzja o tym, że motywem przewodnim aplikacji będą ciemne barwy. Poniższa Tabela 3 prezentuje zbiór większości barw użytych podczas prac nad interfejsem.

Tabela 3. - Kolory aplikacji. Źródło: Opracowanie własne.

Hex	Rola w interfejsie
121214	Tło aplikacji (bardzo ciemne, ale nie czarne)
1E1E22	Tło fali dźwiękowej i komponentów (jaśniejsze niż tło aplikacji, ale nadal ciemne)
6366F1	Główny kolor aplikacji, niektóre przyciski, obwódki (fiolet)
10B981	Akcent, kolor niektórych przycisków (zielony)
E5E7EB	Napisy (łagodna biel)



Rysunek 15 - Makietę interfejsu użytkownika. Źródło: Opracowanie własne.

Do stworzenia projektu interfejsu użytkownika zostało wykorzystane oprogramowanie Figma. Jest ono standardowym oprogramowaniem wykorzystywanym do tworzenia makiet aplikacji (*mockupów*). Wynik prac nad wstępnym interfejsem użytkownika dla systemu WaveScope prezentuje Rysunek 15 - Makietę interfejsu użytkownika. Projekt uwzględnia funkcjonalności, takie jak lista komentarzy, wizualizacja fali audio wraz z odnośnikami do komentarzy na niej, przycisk do wyboru wersji, przycisk pobrania wersji, przycisk wgrania nowej wersji, przyciski nawigacji oraz licznik czasu.

Prezentowana makietę stanowi projekt koncepcyjny i punkt wyjścia dla dalszych prac programistycznych. Ostateczna wersja interfejsu, przedstawiona w rozdziale 5 dotyczącym implementacji, uwzględnia optymalizacje techniczne oraz specyfikę wykorzystanych bibliotek frontendowych.

Ważnym etapem prac nad interfejsem była analiza możliwości biblioteki *wavesurfer.js* [11], wybranej do realizacji kluczowego zadania systemu – wizualizacji fali dźwiękowej (*waveform*). Przeprowadzone testy technologiczne potwierdziły, że narzędzie to, bazując na technologii HTML5 Canvas [29], zapewnia wysoką wydajność renderowania przy niskim obciążeniu zasobów przeglądarki.

Podczas weryfikacji biblioteki ustalono, że poza podstawową funkcją graficznej reprezentacji dźwięku, oferuje ona rozbudowany ekosystem wtyczek (*plugins*) [30], które znacząco przyspieszają implementację wymagań funkcjonalnych. Do najważniejszych z nich należą:

- **Timeline plugin** – umożliwiający dodanie precyzyjnej osi czasu,
- **Regions plugin** – stanowiący fundament systemu WaveScope.

To właśnie funkcja tworzenia regionów okazała się kluczowa z punktu widzenia projektu, gdyż pozwala ona na definiowanie interaktywnych markerów – zarówno punktowych, jak i przedziałowych.

Dzięki temu możliwe stało się zbudowanie systemu komentarzy ściśle powiązanych z konkretnymi współrzędnymi czasowymi w pliku audio. Wykorzystanie gotowych komponentów nawigacyjnych i liczników czasu (z dokładnością do milisekund) pozwoliło na zachowanie wysokiej precyzji operacyjnej, co było jednym z priorytetów projektowych aplikacji.

Zwieńczeniem prac nad warstwą graficzną było opracowanie logotypu aplikacji WaveScope (Rysunek 16 - Logotyp WaveScope). Sygnet logo przedstawia przeplatające się formy falowe, co stanowi bezpośrednią metaforę kluczowej funkcjonalności systemu – procesu wersjonowania oraz dynamicznej współpracy nad materiałem dźwiękowym.

Paleta kolorystyczna wykorzystana w projekcie logotypu jest skorelowana z kolorystyką interfejsu użytkownika. Takie podejście zapewnia pełną spójność identyfikacji wizualnej systemu, budując profesjonalny i jednolity wizerunek aplikacji w oczach użytkownika końcowego.



Rysunek 16 - Logotyp WaveScope. Źródło: Opracowanie własne.

4. Technologie i narzędzia wykorzystane w pracy

Rozdział czwarty poświęcony jest szczegółowemu omówieniu stosu technologicznego oraz narzędzi programistycznych wykorzystanych w procesie tworzenia systemu WaveScope. Wybór konkretnych rozwiązań był podyktowany chęcią stworzenia nowoczesnej i stabilnej aplikacji, opartej na uznanych standardach branżowych, co gwarantuje nie tylko wysoką wydajność, ale również możliwość skalowania i późniejszego rozwoju projektu.

Z uwagi na to, że WaveScope jest aplikacją typu full-stack, proces projektowy wymagał integracji zróżnicowanych technologii odpowiadających za warstwę danych, logikę serwerową oraz interfejs użytkownika. Zaplecze systemu (backend) oparto na języku Java [5] oraz frameworku Spring [6], wykorzystując narzędzie Hibernate [7] do komunikacji z otwartoźródłową bazą danych PostgreSQL [8]. Warstwa kliencka (Front-end) została zrealizowana z wykorzystaniem biblioteki React [9], przy użyciu języka TypeScript [10], a do kluczowej funkcjonalności wizualizacji audio oraz nawigacji odtwarzania wykorzystano bibliotekę wavesurfer.js [11].

W dalszej części rozdziału opisano również założenia architektury REST API [31] jako modelu komunikacji między warstwami. Przedstawiono także narzędzia wspomagające proces wdrażania oraz weryfikacji poprawności działania systemu, takie jak platforma Docker [32] oraz framework JUnit [33] wykorzystywane do testów jednostkowych.

4.1. Java

Warstwa serwerowa systemu WaveScope została zaimplementowana w języku Java, który od lat zajmuje ugruntowaną pozycję jako jeden z najważniejszych standardów w budowie systemów klasy korporacyjnej oraz zaawansowanych aplikacji webowych. Wybór tego języka był podyktowany faktem, iż jest to technologia, w której autor posiada największe doświadczenie, co pozwoliło na sprawne i bezpieczne przygotowanie logiki biznesowej prototypu.

Historia języka Java sięga połowy lat 90. XX wieku, kiedy to firma Sun Microsystems zaprezentowała go jako rozwiązanie oparte na zasadzie „*Write Once, Run Anywhere*” (WORA) [34]. Przez dekady język ten ewoluował, stając się fundamentem dla ogromnej liczby systemów backendowych na całym świecie. Do jego głównych koncepcji należy pełna obiektowość, silne typowanie oraz automatyczne zarządzanie pamięcią poprzez mechanizm *Garbage Collector* [35], co minimalizuje ryzyko wystąpienia błędów związanych z wyciekami pamięci.

W procesie tworzenia prototypu zdecydowano się na wykorzystanie jednej z najnowszych wersji o długoterminowym wsparciu (LTS) — Javy 21. Wersja ta wprowadza szereg nowoczesnych usprawnień, które znacząco poprawiają wydajność i czytelność kodu, oferując jednocześnie najwyższy poziom stabilności. Wybór Javy 21 zapewnia również systemowi ogromny potencjał w zakresie późniejszej rozbudowy, umożliwiając na przykład łatwą migrację do architektury mikroservisowej.

Niezwykle istotnym argumentem przemawiającym za wykorzystaniem tego języka jest jego bogaty ekosystem dojrzałych bibliotek i frameworków. Java stanowi naturalne środowisko dla narzędzi, takich jak Spring oraz Hibernate, które zostały wykorzystane w niniejszej pracy do budowy architektury aplikacji oraz sprawnego zarządzania warstwą trwałości danych.

4.2. React.js

Warstwa kliencka systemu WaveScope została zrealizowana z wykorzystaniem biblioteki React.js. Jest to obecnie jedno z najpopularniejszych narzędzi do budowy dynamicznych i wysoko interaktywnych interfejsów użytkownika, rozwijane przez firmę Meta. Wybór tej technologii pozwolił na stworzenie responsywnej aplikacji typu *Single Page Application* (SPA) [36], co jest kluczowe dla zapewnienia płynności odsłuchu i recenzji materiałów audio bez konieczności przeładowywania całej strony.

Główną koncepcją React.js jest komponentowość. Aplikacja została podzielona na niezależne, reużywalne fragmenty kodu, co znacząco ułatwiło implementację złożonych elementów, takich jak odtwarzacz z wizualizacją fali dźwiękowej czy dynamiczna lista komentarzy. Dzięki takiej konstrukcji biblioteki możliwe było stworzenie w prosty sposób wersji demonstracyjnej dostępnej dla niezalogowanego użytkownika. Narzędzie to wykorzystuje mechanizm *Virtual DOM* [37], który optymalizuje proces aktualizacji interfejsu, zapewniając wysoką wydajność nawet przy częstych zmianach stanu, na przykład podczas przesuwania kursora na osi czasu audio.

W projekcie wykorzystano bibliotekę React w wersji 19.2.0, która wprowadza zaawansowane mechanizmy optymalizacji procesu renderowania. Wykorzystano wzorzec zarządzania stanem oparty na *hookach* (`useState`, `useEffect`), co pozwoliło na zachowanie spójności danych między odtwarzaczem audio a listą komentarzy. Co niezwykle istotne, cały kod frontendu został napisany w języku TypeScript w wersji 4.9.5. Wprowadzenie statycznego typowania pozwoliło na wykrywanie błędów już na etapie pisania kodu, co drastycznie podniosło niezawodność aplikacji, szczególnie w obszarze komunikacji z serwerem oraz obsługi skomplikowanych struktur danych reprezentujących wersje utworów i komentarze.

4.3. PostgreSQL

Jako system przechowywania danych dla projektu WaveScope wykorzystano PostgreSQL. Jest to jeden z najbardziej zaawansowanych, otwartoźródłowych systemów zarządzania relacyjną bazą danych (RDBMS), który od dekad uznawany jest za standard w zakresie niezawodności i wydajności. Wybór tego rozwiązania był podyktowany pełną zgodnością ze standardem SQL oraz silnym naciskiem na zachowanie integralności danych.

W prototypie wykorzystano najnowszą stabilną wersję 18.1 w architekturze 64-bitowej. Oferuje ona zaawansowane mechanizmy transakcyjne zapewniające pełną zgodność z zasadami ACID (*Atomicity, Consistency, Isolation, Durability*) [38]. W kontekście systemu WaveScope, PostgreSQL odpowiada za bezpieczne przechowywanie metadanych o użytkownikach, zespołach produkcyjnych oraz precyzyjnych współrzędnych czasowych komentarzy nanoszonych na oś czasu plików audio.

Kluczowe argumenty przemawiające za wyborem PostgreSQL w tym projekcie to:

- Optymalizacja wydajności zapytań relacyjnych: Baza wspiera zaawansowane złączenia. W warstwie dostępu do danych wykorzystano mechanizmy takie jak `@EntityGraph`, które zostały zastosowane celem uniknięcia problemu „N+1 Select”.
- Modularność i rozszerzalność: System oferuje bogatą gamę typów danych, co ułatwia ewentualny dalszy rozwój aplikacji o nowe funkcjonalności.
- Bezproblemowa współpraca z Hibernate: PostgreSQL jest natywnie wspierany przez framework Spring, co umożliwiło szybką i stabilną implementację warstwy dostępu do danych (Repository).

Zastosowanie PostgreSQL jako darmowego rozwiązania typu *open source* pozwoliło na stworzenie profesjonalnego zaplecza bazodanowego bez generowania dodatkowych kosztów licencyjnych, co wpisuje się w założenia budowy dostępnego i skalowalnego prototypu.

4.4. Spring Boot

Spring Boot w wersji 3.5.7 stanowi fundament warstwy serwerowej systemu WaveScope. Jest to nowoczesne rozszerzenie frameworka Spring, które w ogromnym stopniu zwiększa możliwości i funkcje języka Java oraz minimalizuje czas potrzebny do konfiguracji środowiska pozwalając na bezpośrednią implementację logiki serwerowej.

Kluczową cechą tego narzędzia jest zastosowanie wzorców projektowych, takich jak *Inversion of Control* (IoC) [39] oraz *Dependency Injection* (DI) [40]. Pozwalają one na luźne powiązanie komponentów systemu, co przekłada się na wysoką czytelność kodu, łatwość jego konserwacji oraz efektywne przeprowadzanie testów jednostkowych i integracyjnych. Dodatkowo mechanizmy te

zdejmują z programisty obowiązek zarządzania stanem obiektu. Wersja 3.5.7 frameworka jest w pełni kompatybilna z językiem Java 21, co umożliwia wykorzystanie najnowszych optymalizacji maszyny wirtualnej oraz nowoczesnych struktur językowych.

W architekturze prototypu wykorzystano szereg dedykowanych modułów (tzw. starterów), które automatyzują kluczowe obszary działania aplikacji:

- Spring Boot Starter Web [41] - posłużył do stworzenia punktów końcowych (endpoints) bezstanowego interfejsu REST API. Odpowiada on za mapowanie żądań HTTP na odpowiednie metody serwisowe oraz automatyczną serializację danych do formatu JSON.
- Spring Boot Starter Data JPA [42] - stanowi warstwę pośredniczącą między aplikacją a bazą danych PostgreSQL. Umożliwia on korzystanie z wysokopoziomowych repozytoriów, które automatyzują procesy trwałości danych i zarządzanie relacjami.
- Spring Boot Starter Validation [43] - realizuje wymagania dotyczące spójności danych poprzez automatyczną weryfikację pól obiektów wejściowych, co zapobiega wprowadzaniu do bazy danych niepoprawnych informacji, np. błędnych formatów nazw.
- Integracja z JWT [44] - wykorzystanie bibliotek jjwt-api oraz jjwt-impl pozwoliło na implementację mechanizmu autoryzacji opartego na tokenach JWT [45]. Rozwiązanie to zapewnia bezpieczny dostęp do zasobów systemu bez konieczności przesyłania wrażliwych danych w każdym zapytaniu.

Dodatkowo wykorzystanie narzędzi deweloperskich Spring Boot DevTools [46] pozwoliło na automatyczny restart serwera po wykryciu zmian w kodzie źródłowym, co znacząco zoptymalizowało cykl wytwórczy prototypu. Wybrany stos technologiczny zapewnia aplikacji stabilność oraz gotowość do przyszłej migracji do infrastruktury chmurowej.

4.5. Hibernate

Hibernate jest zaawansowanym narzędziem klasy ORM (*Object-Relational Mapping*) [7], które pełni w systemie WaveScope kluczową rolę w procesie wymiany danych między obiektową strukturą języka Java a relacyjną strukturą bazy danych PostgreSQL. Narzędzie to minimalizuje potrzebę ręcznego pisania zapytań SQL do bazy danych. W niniejszym prototypie Hibernate został wykorzystany jako implementacja standardu JPA (*Jakarta Persistence API*) [47], co pozwoliło na ujednolicenie i uproszczenie operacji na bazie danych przy zachowaniu wysokiego poziomu abstrakcji.

Działanie frameworka Hibernate opiera się na mapowaniu języka Java na tabele w relacyjnej bazie danych. Odbyna się to przez dodanie adnotacji `@Entity` dla klasy mającej stanowić wzór tabeli w bazie danych. Dzięki temu podczas pracy można operować na obiektach, a system automatycznie generuje zapytania SQL niezbędne do zapisu, odczytu, aktualizacji czy usunięcia danych z bazy.

W systemie WaveScope Hibernate zarządza cyklem życia obiektów takich jak *Osoba*, *Projekt* czy *Album*. Wykorzystuje on mechanizm sesji do śledzenia zmian w obiektach i synchronizacji ich z bazą danych w ramach transakcji zarządzanych przez Spring Boot. Narzędzie to w zautomatyzowany sposób obsługuje relacje zdefiniowane na projektowym diagramie klas (Rysunek 14). Przykładem jest relacja jeden do wielu, którą w kodzie oznacza się adnotacją `@OneToMany`.

Wybór frameworku Hibernate przyniósł szereg korzyści, ale wymagał również uwzględnienia specyficznych ograniczeń tej technologii. Wśród zalet tej technologii można wymienić:

- Niezależność od silnika bazy danych: Hibernate pozwala na łatwą zmianę dialektu SQL, co sprawia, że system WaveScope jest gotowy do pracy z różnymi dostawcami baz danych bez znaczących modyfikacji kodu źródłowego.
- Automatyzacja i szybkość wytwarzania: Drastyczne zmniejszenie ilości powtarzalnego kodu pozwoliło na skupienie się na implementacji logiki biznesowej, a nie na ręcznym pisaniu zapytań SQL.

- Zarządzanie integralnością: Framework automatycznie dba o spójność między encjami, co gwarantuje, że na przykład komentarz zawsze pozostanie przypisany do właściwej wersji utworu.

Hibernate, oprócz zalet wymienionych powyżej posiada też wady, o których trzeba pamiętać podczas pracy z nim. Głównymi z nich są:

- Problem N+1: Niewłaściwe skonfigurowanie pobierania danych relacyjnych może prowadzić do nadmierowej liczby zapytań w bazie. W projekcie WaveScope problem ten został zminimalizowany poprzez wykorzystanie dedykowanych zapytań JPQL w miejscach, w których problem ten wystąpiłby przy użyciu zapytania wygenerowanego domyślnie przez Hibernate.
- Narzut wydajnościowy: Warstwa abstrakcji wprowadzana przez ORM generuje pewien narzut procesowy, który przy bardzo dużej skali danych może być zauważalny w porównaniu do czystych zapytań SQL. Projekt WaveScope nie przewiduje aż tak dużej skali danych, przez co wada ta została pominięta w procesie tworzenia systemu.

Dzięki integracji Hibernate ze Spring Data JPA, warstwa dostępu do danych w systemie WaveScope jest nie tylko czytelna, ale również łatwa w utrzymaniu i odporna na błędy typowe dla manualnego zarządzania połączeniami z bazą danych.

4.6. Wavesurfer.js

Centralnym elementem interfejsu użytkownika w systemie WaveScope jest wizualizacja fali dźwiękowej, która została zrealizowana przy użyciu biblioteki wavesurfer.js w wersji 7.11.0. Jest to otwartoźródłowe narzędzie napisane w języku TypeScript, które oprócz wizualizacji dźwięku dostarcza również kilka innych funkcji przydatnych w budowaniu interaktywnego odtwarzacza audio w przeglądarce.

Biblioteka opiera swoje działanie na nowoczesnych technologiach przeglądarkowych, takich jak Web Audio API [48] oraz HTML5 Canvas [49]. Dzięki takiemu podejściu system jest w stanie wygenerować graficzną reprezentację pliku dźwiękowego z bardzo wysoką wydajnością, zachowując przy tym płynność działania interfejsu nawet przy długich nagraniach. Wavesurfer.js nie tylko dostarcza funkcję odtwarzania, ale też umożliwia nawigację po utworze z milisekundową precyzją, co było priorytetem projektowym aplikacji.

Wersja 7 biblioteki charakteryzuje się modułarną architekturą, co pozwoliło na dołączenie jedynie niezbędnych rozszerzeń, optymalizując tym samym wagę aplikacji. W systemie WaveScope kluczowe role odgrywają dwie wtyczki:

- Regions Plugin [50]: stanowi fundament autorskiego systemu recenzji. Umożliwia on definiowanie interaktywnych obszarów na osi czasu, które w projekcie służą jako wizualne odnośniki dla komentarzy. Dzięki tej wtyczce użytkownik może zaznaczyć konkretny fragment utworu (zakres) lub pojedynczy punkt, do którego przypisywane są uwagi techniczne.
- Timeline Plugin [51]: odpowiada za renderowanie precyzyjnej osi czasu nad falą dźwiękową. Pozwala to realizatorowi oraz klientowi na szybką orientację w strukturze utworu i ułatwia komunikację asynchroniczną poprzez odwołania do konkretnych wartości czasowych.

Integracja biblioteki z komponentami React pozwoliła na synchronizację stanu odtwarzacza z resztą interfejsu, dzięki czemu kliknięcie w komentarz na liście powoduje natychmiastowe przeniesienie kursora odtwarzania do odpowiadającego mu regionu na fali audio.

4.7. Rest API

Komunikacja pomiędzy warstwą kliencką, a serwerową w systemie WaveScope opiera się na architekturze REST (*Representational State Transfer*). Bazuje on na protokole HTTP jako medium transportowym, co zapewnia wysoką interoperacyjność, skalowalność oraz prostotę implementacji.

Architektura systemu została zaprojektowana zgodnie z kluczowymi zasadami REST:

- Bezstanowość: Serwer nie przechowuje informacji o stanie sesji klienta między zapytaniami. Każde żądanie wysyłane z aplikacji React zawiera komplet danych niezbędnych do jego autoryzacji i przetworzenia (wykorzystanie tokenów JWT).
- Interfejs oparty na zasobach: Operacje na elementach systemu realizowane są za pomocą standardowych metod protokołu HTTP: GET (odczyt), POST (tworzenie), PUT (aktualizacja) oraz DELETE (usuwanie).
- Reprezentacja danych: Wymiana informacji odbywa się w formacie JSON (*JavaScript Object Notation*), który ze względu na swoją lekkość i czytelność jest idealnie dopasowany do współpracy z biblioteką React.

Zaletami architektury REST są:

- Powszechność i wsparcie: REST jest standardem wspieranym przez wszystkie nowoczesne przeglądarki i frameworki, co ułatwiło integrację ze Spring Boot i React.
- Łatwość debugowania: Przejrzystość zapytań pozwala na szybką analizę ruchu sieciowego oraz zapytań za pomocą narzędzi deweloperskich, takich jak Postman.
- Niski próg wejścia: Prostota architektury pozwoliła na szybkie zbudowanie warstwy serwerowej i skupienie się na budowie interfejsu użytkownika.
- Wbudowane wsparcie transferu plików multimedialnych: Framework Spring MVC udostępnia interfejs `MultipartFile`, który służy do odbierania plików w żądaniach REST typu POST oraz PUT.

Główną wadą architektury REST API jest *over-fetching* i *under-fetching*. Każde zapytanie zwraca określone dane. Może być ich za dużo lub wymagać kilku zapytań do pobrania kompletnych informacji. W systemie WaveScope problem ten został zminimalizowany przez odpowiednie projektowanie obiektów DTO. Takie rozwiązanie pozwoliło zapobiec wysyłaniu pełnego obiektu w zapytaniu. Drugą istotną wadą tego rozwiązania jest konieczność napisania i wystawienia punktu końcowego (*endpointu*) dla każdego zapytania. Oznacza to wysoki koszt stworzenia interfejsu komunikacyjnego między warstwą kliencką, a serwerem.

Podczas fazy projektowej rozważano wykorzystanie technologii GraphQL [52] jako alternatywy dla REST API. GraphQL to język zapytań, który pozwala klientowi na precyzyjne określanie struktury zwracanych danych w ramach jednego punktu końcowego (*endpoint*).

Porównanie architektury REST API oraz GraphQL przedstawia Tabela 4.

Tabela 4. - Porównanie GraphQL i REST API. Źródło: Opracowanie własne.

Kryterium	GraphQL	REST API
Elastyczność	Wysoka, klient w zapytaniu definiuje jakie dane chce pobrać	Niska, serwer definiuje jakie dane może pobrać klient
Obsługa przesyłania plików	Złożony, wymaga dodatkowych bibliotek i obejść	Natywny i stabilny transfer plików WAV/MP3
Złożoność implementacji	Wymaga zdefiniowania schematów w warstwie serwerowej oraz zapytań w warstwie klienckiej	Wymaga zdefiniowania sztywnych ednpointów w warstwie serwerowej, w warstwie klienckiej komunikacja nie wymaga pisania złożonych zapytań

Przeprowadzona analiza porównawcza stanowiła podstawę do wyboru architektury REST API jako standardu komunikacji w relacji klient-serwer. Za przyjęciem tego rozwiązania przemawiały następujące argumenty:

- Natywna obsługa przesyłania plików: Ze względu na specyfikację systemu, stabilność standardowych zapytań HTTP w REST była kluczowym argumentem.
- Struktura danych: Modele danych w WaveScope są relatywnie proste i stabilne. Elastyczność GraphQL nie przynosiłaby w tym przypadku wymiernych korzyści, które przewyższyłyby koszt implementacji REST API.
- Wsparcie w Spring Boot: Moduł *Spring Boot Starter Web* oferuje dojrzałe i bezpieczne mechanizmy budowy REST API, co było priorytetem dla stabilności prototypu.

4.8. Docker

Aplikacja WaveScope składa się z odizolowanych elementów, działających w spójnym ekosystemie, w którym wszystkie komponenty współdzielą ze sobą jądro systemu operacyjnego. Takie rozwiązanie było możliwe dzięki platformie Docker. Umożliwia ona konteneryzację aplikacji i jest współczesnym standardem w zakresie izolacji środowisk uruchomieniowych.

Główną zaletą wykorzystania technologii jest możliwość zdefiniowania całej infrastruktury w formie kodu. W ramach prac nad prototypem przygotowano konfiguracje dla kluczowych elementów stosu technologicznego:

- Baza danych PostgreSQL: Wykorzystano oficjalny obraz bazy danych w wersji 18.1 bez konieczności jego manualnej instalacji w systemie operacyjnym. Dane przechowywane w bazie są trwałe i niezależne od cyklu życia samego kontenera dzięki mechanizmowi wolumenów (*volumes*) [54].
- Warstwa serwerowa (Spring Boot): Aplikacja backendowa została spakowana do obrazu bazującego na środowisku uruchomieniowym Java 21. Zapewnia to identyczne zachowanie systemu na etapie testów oraz docelowego wdrożenia.
- Warstwa kliencka (React): Obraz frontendu został przygotowany w oparciu o lekki serwer HTTP Nginx. Statyczne pliki aplikacji, wygenerowane podczas procesu budowania prototypu, są serwowane bezpośrednio z wnętrza kontenera.

W celu zarządzania wielokontenerową architekturą systemu WaveScope wykorzystano narzędzie Docker Compose [55]. Pozwala ono na zdefiniowanie relacji między kontenerami, konfigurację sieci wewnętrznej oraz zarządzanie zmiennymi środowiskowymi w jednym pliku konfiguracyjnym (`docker-compose.yml`).

4.9. JUnit

W celu zapewnienia poprawności działania kluczowych mechanizmów systemu WaveScope oraz zminimalizowania ryzyka wystąpienia błędów regresyjnych, w procesie wytwórczym wykorzystano framework JUnit [33]. Jest to najpopularniejsze narzędzie do automatyzacji testów w ekosystemie języka Java, które w pełni integruje się z frameworkiem Spring Boot.

W projekcie WaveScope JUnit został wykorzystany głównie do przeprowadzenia testów jednostkowych (*unit tests*) najważniejszych serwisów aplikacji. Pozwoliło to na odizolowaną weryfikację poszczególnych metod bez konieczności uruchamiania pełnego kontekstu aplikacji, co znacząco przyspieszyło proces deweloperski.

Dzięki wykorzystaniu frameworka JUnit, system WaveScope cechuje się wysoką odpornością na błędy logiczne w warstwie backendowej, co stanowiło istotny cel inżynierski niniejszej pracy. Takie

podejście wpisuje się w standardy piramidy testów [56], gdzie testy jednostkowe stanowią najszerszy i najszybszy fundament zapewnienia stabilności oprogramowania.

4.10. Microsoft Clarity

W celu obiektywnej oceny interfejsu użytkownika oraz weryfikacji założeń projektowych dotyczących ergonomii systemu WaveScope, zintegrowano aplikację kliencką z narzędziem analitycznym Microsoft Clarity [57]. Jest to zaawansowana, darmowa platforma telemetryczna służąca do śledzenia i analizy zachowań użytkowników w aplikacjach webowych, nieobciążająca przy tym znacząco wydajności warstwy frontowej.

Wykorzystanie tego narzędzia w procesie ewaluacji gotowego prototypu opierało się na dwóch głównych modułach funkcjonalnych:

- **Mapy cieplne (*Heatmaps*):** Narzędzie posłużyło do wygenerowania zagregowanych map kliknięć. Pozwoliło to na graficzną wizualizację miejsc, z którymi docelowi odbiorcy wchodzili w najczęstszą interakcję (takich jak elementy sterujące odtwarzaczem czy formularze dodawania komentarzy). Umożliwiło to również weryfikację, czy kluczowe z punktu widzenia systemu przyciski są odpowiednio wyeksponowane i zauważalne.
- **Nagrania sesji użytkowników (*Session Recordings*):** Oprogramowanie umożliwiło w pełni anonimowe rejestrowanie ścieżek kursora oraz ekranów testerów podczas wykonywania przez nich określonych scenariuszy testowych. Analiza odtworzonych nagrań pozwoliła na precyzyjne zidentyfikowanie tzw. punktów tarcia, czyli momentów zawahania użytkownika, zbędnych kliknięć czy miejsc, w których nawigacja po systemie sprawiała trudności.

Zastosowanie Microsoft Clarity dostarczyło cennych danych badawczych. Zebrana w ten sposób telemetria stanowiła twardy, analityczny fundament do zbadania użyteczności oraz intuicyjności zaproponowanego rozwiązania. Pozwoliło to na weryfikację rzeczywistych doświadczeń użytkowników w zderzeniu z zaimplementowanym interfejsem graficznym.

5. Implementacja systemu

Rozdział piąty poświęcony jest szczegółowemu omówieniu procesu implementacji systemu WaveScope. Przedstawiono w nim strukturę kodu źródłowego, kluczowe algorytmy oraz sposoby realizacji wymagań funkcjonalnych w poszczególnych warstwach aplikacji. Zgodnie z architekturą klient-serwer, opis prac programistycznych podzielono na implementację logiki biznesowej po stronie serwera (*Backend*) oraz budowę interaktywnego interfejsu użytkownika (*Frontend*).

5.1. Implementacja warstwy serwerowej – backend

Warstwa serwerowa systemu WaveScope stanowi technologiczny fundament aplikacji, odpowiadający za realizację kluczowej logiki biznesowej, bezpieczne zarządzanie danymi oraz integrację z zewnętrznymi usługami sieciowymi. Zaplecze systemu zostało zaimplementowane w języku Java (wersja 21) [5] przy użyciu frameworka Spring Boot (wersja 3.5.7) [6], co pozwoliło na zbudowanie wydajnej, bezstanowej architektury typu REST API [31].

Głównym zadaniem warstwy serwerowej jest asynchroniczne i bezpieczne procesowanie żądań pochodzących od warstwy klienckiej, automatyzacja procesu wersjonowania plików audio oraz ścisła kontrola uprawnień w ramach zespołów produkcyjnych. Zaimplementowana logika gwarantuje spójność danych oraz izolację zasobów, dzięki czemu operacje na plikach, albumach i komentarzach są wykonywane w sposób przewidywalny i odporny na błędy transakcyjne.

W poniższych podrozdziałach szczegółowo omówiono strukturę architektury, mechanizmy mapowania obiektowo-relacyjnego, zasady separacji danych za pomocą struktur DTO [53], a także techniczną realizację systemów bezpieczeństwa, automatycznego wersjonowania i integracji z chmurowym API Google Drive [58].

5.1.1 Struktura katalogów i wzorzec warstwowy

Prace nad implementacją warstwy serwerowej systemu WaveScope rozpoczęto od zdefiniowania struktury pakietów oraz implementacji wzorca warstwowego. Zastosowanie tego podejścia pozwoliło na separację odpowiedzialności poszczególnych komponentów systemu, co bezpośrednio przekłada się na czytelność kodu, jego hermetyzację oraz ułatwia niezależne testowanie elementów logiki biznesowej.

W strukturze projektu wyróżniono dedykowane pakiety, z których każdy odpowiada za odizolowany technologicznie obszar działania aplikacji:

- **api** – warstwa prezentacji i interfejsu programistycznego, w której umieszczono klasy kontrolerów (oznaczone adnotacją `@RestController`). Odpowiadają one za nasłuchiwanie żądań HTTP, mapowanie punktów końcowych (*endpoints*) oraz przekazywanie parametrów wejściowych do warstwy biznesowej.
- **service** – centralna warstwa logiki biznesowej systemu, grupująca klasy oznaczone adnotacją `@Service`. To w tym miejscu przetwarzane są kluczowe operacje aplikacji, takie jak algorytmy automatycznego wersjonowania utworów oraz koordynacja procesów wymiany danych.
- **repository** – warstwa dostępu do danych (trwałości), zawierająca interfejsy oznaczane adnotacją `@Repository`. Wykorzystują one mechanizmy Spring Data JPA oraz frameworka Hibernate do bezobsługowej lub dedykowanej komunikacji z bazą danych PostgreSQL.
- **model** – pakiet przechowujący klasy encji bazodanowych (oznaczonych adnotacją `@Entity`). Definiują one strukturę danych systemu, które są mapowane bezpośrednio na tabele relacyjne w bazie danych.
- **DTO** (*Data Transfer Object*) – katalog zawierający bezstanowe struktury danych wykorzystywane do asynchronicznej komunikacji wewnątrz REST API. Pozwalają one na odchudzenie i bezpieczne przesyłanie obiektów między klientem a serwerem.

- **security** – dedykowany pakiet odpowiedzialny za bezpieczeństwo systemu. Skupia on filtry sieciowe, konfigurację Spring Security, komponenty generujące i weryfikujące tokeny JWT oraz logikę uwierzytelniania i autoryzacji użytkowników.
- **googleDrivePlugin** – moduł izolujący logikę integracji z zewnętrznym API Google Drive. Przechowywany jest tu plik odpowiedzialny za zarządzanie plikami na dysku użytkownika.

Zaimplementowana struktura pakietów pozwoliła na zorganizowaną, modułarną pracę nad projektem, gwarantując, że modyfikacja kodu w jednej warstwie (np. zmiana wyglądu obiektu DTO) nie wpływa destrukcyjnie na stabilność pozostałych elementów systemu (np. na zapytania repozytorium).

5.1.2 Modelowanie danych i komunikacja z bazą

Trwałym fundamentem systemu WaveScope, odpowiadającym za bezpieczne i ustrukturyzowane przechowywanie metadanych aplikacji, jest relacyjna baza danych PostgreSQL. Komunikacja pomiędzy warstwą serwerową a silnikiem bazy danych została zrealizowana przy użyciu technologii Spring Data JPA oraz implementującego ją frameworka Hibernate. Pozwoliło to na zastosowanie mapowania obiektowo-relacyjnego (*ORM*), dzięki czemu struktury tabel bazodanowych są w pełni odzwierciedlane przez klasy języka Java, a operacje na danych mogą być wykonywane na poziomie obiekowym, bez konieczności manualnego zarządzania połączeniami JDBC [59].

Proces modelowania danych rozpoczęto od zdefiniowania klas encji, które reprezentują obiekty biznesowe systemu, takie jak użytkownicy, zespoły, przynależność do zespołów, projekty, albumy, wersje utworów oraz komentarze. Klasy te zostały opatrzone adnotacją `@Entity`, która informuje dostawcę JPA o konieczności powiązania danej klasy z odpowiednią tabelą w bazie danych. Klasy te zostały dodatkowo opatrzone standardowymi adnotacjami pozwalającymi wykorzystywać te dane w systemie takimi jak `@Getter` oraz `@Setter`. Kluczowe znaczenie przy projektowaniu encji miało poprawne odzwierciedlenie relacji zaimplementowanych w bazie (jeden – do – wielu), realizowanych przez adnotację `@OneToMany` oraz konfiguracja kaskadowości, która gwarantuje spójność danych – przykładowo, usunięcie projektu automatycznie usuwa powiązane z nim wersje utworów. W encjach zastosowano również adnotacje walidujące dane przed zapisem do bazy takie jak `@NotNull` oraz `@NotBlank`. Ostatnim wymogiem przy konfiguracji encji było dodanie konstruktora bezargumentowego. Jest on niezbędny dla frameworka Hibernate, który wykorzystuje mechanizm refleksji [60] do instancjonowania pustych obiektów Javy przed wypełnieniem ich danymi z bazy. Przykładową implementację encji przedstawia *Listing 1*.

Listing 1 – Przykład encji dla klasy Project. Źródło: Opracowanie własne.

```
@Entity
@Getter
@Setter
@Table(name = "projects")
public class Project {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull
    @NotBlank
    @Column(name = "name", nullable = false)
    private String name;

    @NotNull
    @Column(name = "created_date")
    private LocalDateTime createdAt;

    @NotNull
```

```

@Column(name = "modified_date")
private LocalDateTime modifiedDate;

@NotNull
@Column(name = "status")
@Enumerated(EnumType.STRING)
private ProjectStatus status = ProjectStatus.IN_PROGRESS;

private String description;

@ManyToOne(fetch = FetchType.LAZY)
@NotNull
private Band band;

@ManyToOne(fetch = FetchType.LAZY)
@NotNull
private BandMember owner;

@ManyToOne(fetch = FetchType.LAZY)
private Album album;

@OneToMany(mappedBy = "project", cascade = CascadeType.ALL,
orphanRemoval = true)
private Set<Version> versions = new HashSet<>();

@ElementCollection
@CollectionTable(
    name = "project_approvals",
    joinColumns = @JoinColumn(name = "project_id")
)
@Column(name = "member_id")
private Set<Long> approvedByMemberIds = new HashSet<>();

public Project() {this.createdDate = LocalDateTime.now();}

```

Za bezpośrednią wymianę informacji z bazą danych odpowiada warstwa repozytoriów (pakiet `repository`). Interfejsy w tej warstwie rozszerzają dostarczany przez framework interfejs `JpaRepository`, co automatycznie zapewnia dostęp do podstawowych operacji CRUD [61] bez konieczności pisania jakiegokolwiek kodu implementacyjnego. Spring Data JPA analizuje również nazwy metod deklarowanych w interfejsach i na ich podstawie generuje automatyczne zapytania.

Domyślne mechanizmy generowania zapytań przez system ORM, choć drastycznie przyspieszają proces wytwórczy, mogą w pewnych scenariuszach prowadzić do istotnych problemów wydajnościowych. Najpopularniejszym z nich jest problem N+1 zapytań [62]. Pojawia się on w sytuacjach, gdy aplikacja pobiera listę obiektów głównych (np. komentarzy), a następnie dla każdego z nich wykonuje osobne, pojedyncze zapytanie SQL w celu dociągnięcia obiektów powiązanych relacją (np. danych ich autorów). Przy dużej skali danych generuje to ogromny narzut na sieć i drastycznie obniża wydajność aplikacji.

W celu optymalizacji komunikacji i eliminacji tego problemu, w systemie WaveScope zastosowano mechanizm grafów encji (*Entity Graph*). Pozwala on na deklaratywne wskazanie, które powiązane relacje powinny zostać dociągnięte zachłannie (*eager fetching*) w ramach jednego, zoptymalizowanego zapytania złączeniowego (JOIN). Podejście to jest znacznie bezpieczniejsze od tradycyjnych, tekstowych zapytań JPQL umieszczanych w adnotacji `@Query` wraz z klauzulą JOIN FETCH, ponieważ pozwala zachować pełną spójność danych (zapobiega powstawaniu tzw. *partial objects*) oraz zapewnia bezpieczeństwo typów na etapie kompilacji aplikacji.

Dodatkowo, o wydajność i poprawność operacji filtrujących na poziomie bazy danych dba mechanizm metod pochodnych (*Derived Query Methods*), na podstawie których framework

automatycznie generuje bezpieczne zapytania SQL. W kodzie wykorzystano nie tylko podstawowe konstrukcje składniowe (łączące prefiks z kryteriami filtrowania), ale zastosowano również specjalny znak podkreślenia (`_`), który sygnalizuje frameworkowi przejście przez zagnieżdżoną kolekcję powiązanych obiektów. Przykładową strukturę zoptymalizowanego repozytorium, łączącą dociąganie relacji za pomocą adnotacji `@EntityGraph` z precyzyjnym filtrowaniem opartym na nazewnictwie metody, przedstawia Listing 2.

Listing 2 – Przykład interfejsu rozszerzającego `JpaRepository` dla klasy `Band`. Źródło: Opracowanie własne

```
public interface BandRepository extends JpaRepository<Band, Long> {  
  
    @EntityGraph(attributePaths = {"bandMembers", "bandMembers.user"})  
    List<Band> findByBandMembers_UserId(Long userId);  
}
```

5.1.3 Komunikacja REST API i separacja danych za pomocą obiektów DTO

Komunikacja pomiędzy warstwą kliencką, a serwerową w systemie WaveScope opiera się na asynchronicznej wymianie komunikatów w formacie JSON [64] za pośrednictwem bezstanowego interfejsu REST API [31]. W architekturze systemów dobrą praktyką inżynierską jest unikanie bezpośredniego ekspozowania encji bazodanowych (klas z pakietu `model`) w punktach końcowych (*endpoints*). Bezpośrednie przesyłanie obiektów mapowanych przez Hibernate do warstwy prezentacji niesie za sobą poważne ryzyka wydajnościowe oraz bezpieczeństwa, takie jak ujawnienie wrażliwych danych (np. skrótów haseł użytkowników) czy wywołanie błędów cyklicznych powiązań podczas serializacji obiektów relacyjnych do formatu JSON.

W celu wyeliminowania tych zagrożeń oraz wprowadzenia ścisłej separacji odpowiedzialności, w projekcie zaimplementowano wzorzec projektowy DTO (*Data Transfer Object*) [53]. Obiekty DTO to proste, bezstanowe struktury danych, które nie zawierają żadnej logiki biznesowej, a ich jedynym zadaniem jest przetransportowanie dokładnie takiego zestawu informacji, jakiego w danym momencie wymaga interfejs użytkownika. W systemie obiekty te zostały zaimplementowane przy użyciu specjalnego rodzaju klasy jakim jest Rekord [65] cechujący się niezmiennością oraz zwięzłością.

Zastosowanie obiektów DTO w systemie WaveScope przyniosło dwie kluczowe korzyści inżynierskie:

- **Odchudzenie pakietów sieciowych:** Pakiet zawiera jedynie dane wykorzystywane przez konkretny endpoint w warstwie klienckiej. Minimalizuje to narzut sieciowy.
- **Ochrona przed cyklami:** Relacje dwukierunkowe w JPA podczas domyślnej próby zamiany na format JSON przez bibliotekę Jackson [66] powodują pętlę nieskończoną. Klasy DTO „płaszczą” tę strukturę, przesyłając na przykład tylko identyfikator ID obiektu nadrzędnego.

Proces transformacji danych zachodzi w warstwie usługowej (*service*). Serwis pobiera encję z repozytorium, a następnie przepisuje jej pola na odpowiadający jej obiekt DTO przed przekazaniem wyniku do kontrolera. Przykładową strukturę klasy typu DTO, wykorzystywaną do przesyłania informacji o projekcie, przedstawia Listing 3.

Listing 3 – przykład obiektu DTO dla klasy `Projekt`. Opracowanie własne.

```
public record ProjectDTO(  
    String name,  
    Long projectId,  
    ProjectStatus status,  
    String description,  
    List<ApprovedUserDTO> approvedUsers ({}  
) {  
}
```

Z kolei za samo wystawienie punktu końcowego, odebranie żądania i zwrócenie przygotowanego obiektu DTO w ciele odpowiedzi HTTP odpowiada klasa kontrolera z pakietu `api`. Wykorzystuje ona adnotacje Springa, takie jak `@GetMapping` czy `@PostMapping`, do obsługi odpowiednich metod protokołu HTTP. Przykład kontrolera realizującego komunikację z wykorzystaniem separacji danych prezentuje *Listing 4*.

Listing 4 – przykład kontrolera REST API przetwarzającego obiekty DTO dla klasy Zespół. Opracowanie własne.

```
@PostMapping
public ResponseEntity<BandDTO> createBand(@RequestBody CreateBandDTO
bandDTO, @AuthenticationPrincipal User user) {
    Band createdBand = bandService.createBand(bandDTO.name(),
bandDTO.useGoogleDrive(), user);
    return ResponseEntity.status(201).body(new BandDTO(createdBand.getId(),
createdBand.getName()));
}
```

Dzięki tak zaprojektowanemu interfejsowi komunikacyjnemu, warstwy frontendowa i backendowa systemu WaveScope są od siebie niezależne strukturalnie. Modyfikacje w bazie danych (np. zmiana typu kolumny) wymagają jedynie poprawy mapowania wewnątrz serwera, pozostając całkowicie przezroczyste dla aplikacji klienckiej, o ile struktura obiektu DTO pozostaje niezmienna.

Ważnym aspektem dla zastosowania komunikacji REST API w systemie WaveScope było natywne wsparcie dla przesyłania multimediów, a w szczególności plików audio. Zostało to zrealizowane przy użyciu interfejsu `MultipartFile` [67]. Wbudowany w Springa `MultipartResolver` wykrywa takie żądanie, a następnie parsuje jego zawartość i udostępnia plik jako obiekt `MultipartFile`. Plik taki do momentu zapisu w docelowym miejscu jest przetrzymywany w pamięci RAM. Aby prawidłowo obsłużyć takie zapytanie należy zapisać taki plik przed zakończeniem żądania HTTP, w przeciwnym wypadku zostanie bezpowrotnie usunięty. Przykład kontrolera obsługującego komunikację z wymianą plików prezentuje *Listing 5*.

Listing 5 – przykład kontrolera REST API przetwarzającego plik przez MultipartFile. Opracowanie własne

```
@PostMapping(consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
public ResponseEntity<VersionDTO> uploadVersion(@RequestPart("file")
MultipartFile file, @PathVariable Long bandId,
                                           @PathVariable Long
projectId, @RequestPart("version") VersionDescriptionDTO dto,
                                           @AuthenticationPrincipal
User user) throws IOException {
    Version version = versionService.createNewVersion(projectId, bandId,
dto.description(), file, user);
    VersionDTO response = new VersionDTO(version.getId(),
version.getVersion(), version.isCurrentVersion(),
version.getOriginalFileName(), version.getDescription(),
version.getDate(), version.getOwner().getName());

    return ResponseEntity.status(201).body(response);
}
```

5.1.4 Bezpieczeństwo i bezstanowa autoryzacja JWT

Jednym z najważniejszych niefunkcjonalnych wymagań stawianych systemowi WaveScope było zapewnienie bezpiecznego dostępu do zasobów przy jednoczesnej rezygnacji z tradycyjnego przechowywania stanu sesji w pamięci serwera (`HttpSession`), co znacząco ograniczałoby skalowalność poziomą aplikacji. Aby wyeliminować te ograniczenia, architekturę bezpieczeństwa

systemu oparto na podejściu hybrydowym, łączącym standard OAuth 2.0 [26] / OpenID Connect (OIDC) [68] (realizowanego przez zewnętrznego dostawcę tożsamości Google) oraz mechanizmu bezstanowych tokenów JWT (*JSON Web Token*) [45]. Architektura ta eliminuje klasyczną sesję serwerową, jednak w celu zapewnienia pełnej kontroli nad uprawnieniami użytkowników w czasie rzeczywistym, każde żądanie podlega weryfikacji kryptograficznej oraz szybkiej walidacji bazodanowej. Realizacja łączności realizowanej przed dostawcą tożsamości Google była możliwa do realizacji po konfiguracji projektu Google Cloud [69].

Proces uwierzytelniania w aplikacji przebiega w sposób dwuetapowy, co pozwala na pełną izolację logiki biznesowej od bezpośrednich danych uwierzytelniających użytkownika:

- **Uwierzytelnienie zewnętrzne (OAuth2/OIDC):** Użytkownik inicjuje logowanie, które jest przekierowywane do serwerów Google. Po pomyślnej weryfikacji tożsamości, aplikacja backendowa odbiera profil użytkownika i tokeny dostępowe.
- **Generowanie sesji bezstanowej (JWT):** Na podstawie unikalnego identyfikatora Google, backend tworzy własny, kryptograficznie podpisany token JWT, który staje się jedynym dowodem tożsamości użytkownika w relacji z systemem WaveScope.

Centralnym punktem konfiguracji tego procesu jest klasa `SecurityConfig` (*Listing 6*), która definiuje łańcuch filtrów bezpieczeństwa (*SecurityFilterChain*). Konfiguracja ta wprost wyłącza mechanizm CSRF. W kontekście uruchomieniowym, gdzie aplikacja kliencka i serwerowa mogą działać pod różnymi adresami (np. w środowisku deweloperskim na osobnych portach), ochrona przed nieautoryzowanymi żadaniami międzywitrynowymi została powierzona restrykcyjnie skonfigurowanemu komponentowi CORS (*Cross-Origin Resource Sharing*) [70] oraz atrybutom ciasteczek. Dodatkowo narzucono politykę `SessionCreationPolicy.STATELESS` oraz zarejestrowano dedykowane komponenty odpowiedzialne za obsługę protokołu OAuth2 i weryfikację tokenów.

Listing 6 – Konfiguracja łańcucha filtrów bezpieczeństwa w klasie SecurityConfig. Opracowanie własne.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    private final CustomOAuth2UserService customOAuth2UserService;
    private final OAuth2LoginSuccessHandler oAuth2LoginSuccessHandler;
    private final JwtAuthenticationFilter jwtAuthenticationFilter;

    public SecurityConfig(CustomOAuth2UserService customOAuth2UserService,
        OAuth2LoginSuccessHandler oAuth2LoginSuccessHandler,
        JwtAuthenticationFilter jwtAuthenticationFilter) {
        this.customOAuth2UserService = customOAuth2UserService;
        this.oAuth2LoginSuccessHandler = oAuth2LoginSuccessHandler;
        this.jwtAuthenticationFilter = jwtAuthenticationFilter;
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws
    Exception {
        http
            .cors().and()
            .csrf(csrf -> csrf.disable())
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/",
                    "/error",
                    "/login**",
                    "/oauth2/**",
                    "/login/success").permitAll()
                .anyRequest().authenticated()
            )
            .oauth2Login(oauth2 -> oauth2
```

```

        .userInfoEndpoint(userInfo -> userInfo
            .oidcUserService(customOAuth2UserService)
        )
        .successHandler(oAuth2LoginSuccessHandler)
    )
    .sessionManagement(session ->
session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .addFilterBefore(jwtAuthenticationFilter,
UsernamePasswordAuthenticationFilter.class);

    return http.build();
}

@Bean
public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();

configuration.setAllowedOrigins(Arrays.asList("http://localhost:3000",
"http://localhost:8080"));
    configuration.setAllowedMethods(Arrays.asList("GET", "POST", "PUT",
"DELETE", "OPTIONS"));
    configuration.setAllowedHeaders(Arrays.asList("*"));
    configuration.setAllowCredentials(true);
    configuration.setExposedHeaders(List.of("Set-Cookie"));
    UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", configuration);
    return source;
}
}

```

W momencie, gdy serwer Google potwierdzi tożsamość użytkownika, kontrolę przejmuje klasa CustomOAuth2UserService rozszerzająca OidcUserService (Listing 7). Jej zadaniem jest załadowanie profilu użytkownika i ekstrakcja kluczowych atrybutów OIDC, takich jak unikalny identyfikator dostawcy (sub), adres e-mail, nazwa użytkownika oraz adres URL zdjęcia profilowego.

Komponent realizuje podejście polegające na automatycznym tworzeniu kont w locie (*Just-In-Time Provisioning*) – sprawdza, czy użytkownik o danym identyfikatorze istnieje w bazie danych PostgreSQL. Jeśli nie, tworzy nową encję User. Dodatkowo przy każdym logowaniu dane profilowe są aktualizowane (na wypadek zmiany nazwy lub zdjęcia w usłudze Google).

Listing 7 – Implementacja serwisu CustomOAuth2UserService przetwarzającego profil OIDC. Opracowanie własne.

```

@Service
public class CustomOAuth2UserService extends OidcUserService {

    private final UserService userService;

    public CustomOAuth2UserService(UserService userService) {
        this.userService = userService;
    }

    @Override
    @Transactional
    public OidcUser loadUser(OidcUserRequest userRequest) throws
OAuth2AuthenticationException {
        OAuth2User oAuth2User = super.loadUser(userRequest);

        String googleId = oAuth2User.getAttribute("sub");
    }
}

```

```

        String email = oAuth2User.getAttribute("email");
        String name = oAuth2User.getAttribute("name");
        String pictureUrl = oAuth2User.getAttribute("picture");

        User user = userService.getUserByGoogleId(googleId)
            .orElseGet(() -> new User(googleId, email, name,
pictureUrl));

        user.setEmail(email);
        user.setName(name);
        user.setPictureUrl(pictureUrl);
        user.setLastLogin();
        userService.saveUser(user);

        return new DefaultOidcUser(
            oAuth2User.getAuthorities(),
            userRequest.getIdToken(),
            "sub"
        );
    }
}

```

Po pomyślnym przetworzeniu profilu, Spring Security uruchamia komponent `OAuth2LoginSuccessHandler` (Listing 8). Klasa ta przechwytuje długotrwały token odświeżania (*Google Refresh Token*) – niezbędny do integracji z API Google Drive – i trwale zapisuje go w encji użytkownika. Następnie generuje bezpieczny token JWT za pomocą dedykowanego serwisu `JwtService` i osadza go w odpowiedzi HTTP, implementując kluczowy mechanizm obrony przed atakami typu XSS (Cross-Site Scripting) [71].

Token JWT nie jest zwracany w ciele odpowiedzi ani w parametrze URL (co narażałoby go na przechwycenie przez złośliwe skrypty JavaScript), lecz zostaje wysłany jako ciasteczko o nazwie `jwt`. Zastosowanie obiektu `ResponseCookie` pozwala na precyzyjną konfigurację polityki bezpieczeństwa:

- Flaga `HttpOnly=true` całkowicie blokuje dostęp do zawartości ciasteczka z poziomu kodu JavaScript w przeglądarce.
- Flagi `Secure=true` oraz `SameSite="None"` są niezbędne do prawidłowej i bezpiecznej obsługi żądań typu CORS (*Cross-Origin*), umożliwiając przesyłanie tokenu z zewnętrznej domeny frontendu, przy jednoczesnym wymuszeniu szyfrowania.

Po konfiguracji ciasteczka użytkownik jest przekierowywany na docelowy adres aplikacji klienckiej.

Listing 8 – Obsługa sukcesu uwierzytelniania i osadzenia tokenu JWT w ciasteczku `HttpOnly`. Opracowanie własne.

```

@Component
public class OAuth2LoginSuccessHandler extends
SimpleUrlAuthenticationSuccessHandler {

    private final UserService userService;

    private final JwtService jwtService;

    private final OAuth2AuthorizedClientService authorizedClientService;

    @Value("${app.frontend.url:http://localhost:3000}")
    private String frontendUrl;
}

```

```

    public OAuth2LoginSuccessHandler(UserService userService, JwtService
jwtService, OAuth2AuthorizedClientService authorizedClientService) {
        this.userService = userService;
        this.jwtService = jwtService;
        this.authorizedClientService = authorizedClientService;
    }

    @Override
    public void onAuthenticationSuccess(HttpServletRequest request,
                                       HttpServletResponse response,
                                       Authentication authentication)
throws IOException {
        OAuth2AuthenticationToken oauthToken = (OAuth2AuthenticationToken)
authentication;
        OAuth2User oauth2User = (OAuth2User) authentication.getPrincipal();

        String googleId = oauth2User.getAttribute("sub");
        User user = userService.getUserByGoogleId(googleId)
            .orElseThrow(() -> new RuntimeException("User not found
after OAuth2 login"));

        OAuth2AuthorizedClient client =
authorizedClientService.loadAuthorizedClient(
            oauthToken.getAuthorizedClientRegistrationId(),
            oauthToken.getName());

        if (client != null && client.getRefreshToken() != null) {
user.setGoogleRefreshToken(client.getRefreshToken().getTokenValue());
            userService.saveUser(user);
        }

        String token = jwtService.generateToken(user);

        ResponseCookie jwtCookie = ResponseCookie.from("jwt", token)
            .httpOnly(true)
            .secure(true)
            .path("/")
            .maxAge(3600)
            .sameSite("None")
            .build();

        response.addHeader(HttpHeaders.SET_COOKIE, jwtCookie.toString());
        response.sendRedirect(frontendUrl + "/auth/callback");
    }
}

```

Za kryptograficzne operacje na tokenach odpowiada klasa `JwtService` (Listing 9). Do podpisu tokenów wykorzystywany jest silny algorytm HMAC-SHA256 (HS256), bazujący na tajnym kluczu (*Secret Key*) wstrzykiwanym ze zmiennych środowiskowych aplikacji. W strukturze tokenu (w sekcji *Claims*) jako główny podmiot (*Subject*) umieszczany jest identyfikator `googleId` użytkownika, co pozwala na bezstanową identyfikację. Serwis odpowiada również za walidację kryptograficzną tokenu oraz weryfikację czasu jego wygaśnięcia (*Expiration Time*).

Listing 9 – klasa `JwtService` odpowiedzialna za operacje kryptograficzne na tokenach. Opracowanie własne

```

@Service
public class JwtService {

    @Value("${jwt.secret}")

```

```

private String SECRET_KEY;

@Value("${jwt.expiration}")
private long EXPIRATION_TIME;

public String generateToken(User user) {
    Date now = new Date();
    Date expiration = new Date(now.getTime() + EXPIRATION_TIME);

    return Jwts.builder()
        .setSubject(user.getGoogleId())
        .setIssuedAt(now)
        .setExpiration(expiration)
        .signWith(getSigningKey(), SignatureAlgorithm.HS256)
        .compact();
}

private Key getSigningKey() {
    byte[] keyBytes = SECRET_KEY.getBytes(StandardCharsets.UTF_8);
    return Keys.hmacShaKeyFor(keyBytes);
}

public String extractUserId(String token) {
    return extractAllClaims(token).getSubject();
}

private Claims extractAllClaims(String token) {
    return Jwts.parserBuilder()
        .setSigningKey(getSigningKey())
        .build()
        .parseClaimsJws(token)
        .getBody();
}
}

```

Każde kolejne żądanie wysyłane z aplikacji klienckiej do chronionych zasobów API przechodzi przez dedykowany filtr `JwtAuthenticationFilter` (Listing 10), który rozszerza klasę `OncePerRequestFilter`, gwarantując jednokrotne wykonanie filtru na jedno żądanie HTTP. Filtr ten implementuje elastyczną logikę poszukiwania tokenu:

- W pierwszej kolejności przeszukuje nadesłane ciasteczka w poszukiwaniu klucza `jwt` (standardowa komunikacja z aplikacji frontendowej).
- Jeśli ciasteczko nie istnieje, filtr próbuje wyekstrahować token z nagłówka `Authorization` przy użyciu schematu `Bearer` (przydatne przy testowaniu punktów końcowych za pomocą zewnętrznych narzędzi, na przykład Postman).

Listing 10 – implementacja filtru `JwtAuthenticationFilter` autoryzującego żądania. Opracowanie własne.

```

@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private final JwtService jwtService;

    private final UserService userService;

    public JwtAuthenticationFilter(JwtService jwtService, UserService
userService) {
        this.jwtService = jwtService;
    }
}

```

```

        this.userService = userService;
    }
    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain) throws
ServletException, IOException {

        String token = null;

        if (request.getCookies() != null) {
            for (Cookie cookie : request.getCookies()) {
                if ("jwt".equals(cookie.getName())) {
                    token = cookie.getValue();
                }
            }
        }

        if (token == null) {
            String authHeader = request.getHeader("Authorization");
            if (authHeader != null && authHeader.startsWith("Bearer ")) {
                token = authHeader.substring(7);
            }
        }

        if (token == null) {
            filterChain.doFilter(request, response);
            return;
        }

        try {
            String googleId = jwtService.extractUserId(token);

            User user =
userService.getUserByGoogleId(googleId).orElse(null);

            if (user != null &&
SecurityContextHolder.getContext().getAuthentication() == null) {

                UsernamePasswordAuthenticationToken auth =
                    new UsernamePasswordAuthenticationToken(
                        user,
                        null,
                        List.of(new SimpleGrantedAuthority("USER"))
                    );

                auth.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));
                SecurityContextHolder.getContext().setAuthentication(auth);
            }

        } catch (Exception e) {
            logger.error("Błąd weryfikacji tokenu JWT: " + e.getMessage());
        }

        filterChain.doFilter(request, response);
    }
}

```

Jeżeli token zostanie odnaleziony i pomyślnie zweryfikowany przez `JwtService`, filtr pobiera dane użytkownika z bazy i tworzy obiekt `UsernamePasswordAuthenticationToken` z przypisaną rolą `USER`. Mimo że krok ten generuje dodatkowe zapytanie bazodanowe, jest to świadomy kompromis architektoniczny – pozwala on na natychmiastową weryfikację statusu konta użytkownika w systemie i odświeżenie jego danych (istotne z punktu widzenia kontrolerów API), bez konieczności czekania na wygaśnięcie tokenu JWT. Dodatkowo, w przypadku wykrycia tokenu uszkodzonego lub wygasłego, wyjątek jest rzetelnie logowany na potrzeby audytowe (`logger.error`), a żądanie zostaje przekazane dalej jako nieautoryzowane, pozostawiając ostateczną decyzję o zablokowaniu dostępu nadrzędnej konfiguracji Spring Security. Obiekt uwierzytelnienia wstrzykiwany jest do kontekstu bezpieczeństwa (`SecurityContextHolder`), dzięki czemu w dalszych warstwach aplikacji zapytanie traktowane jest jako w pełni autoryzowane.

5.1.5 Logika biznesowa i wersjonowanie plików audio

Główną rolą warstwy serwerowej jest udostępnianie logiki biznesowej realizującej założenia projektu. System `WaveScope` składa się zarówno z prostej logiki służącej do zarządzania użytkownikami, jak i bardziej zaawansowanej obsługującej zarządzanie plikami na serwerze. Zgodnie z architekturą systemu, logika została zaimplementowana wewnątrz pakietu `service`.

Przykładem prostej logiki składającej się z zapisu danych do bazy oraz podstawowej walidacji jest funkcja `CreateComment` (*Listing 11*) z serwisu `CommentService`. Posiada ona adnotację `@Transactional`, która zapewnia spójność danych między systemem, a bazą. Jeżeli funkcja ta zostanie przerwana przed zakończeniem swoich zadań, to wszystkie operacje na bazie danych zostaną przywrócone do stanu poprzedzającego wywołanie. Pierwszym elementem logicznym funkcji jest sprawdzenie, czy czas rozpoczęcia komentarza jest mniejszy niż jego koniec. Jeżeli czas rozpoczęcia jest późniejszy niż zakończenia, są one zamieniane między sobą. Może to mieć miejsce, gdy użytkownik będzie zaznaczał przedział komentarza niezgodnie z kierunkiem osi czasu. System sam zapewni poprawność logiczną danych. W następnej kolejności są pobierane z bazy obiekty z którymi komentarz będzie powiązany, czyli wersja utworu i użytkownik. Dodatkowo sprawdzane jest czy użytkownik jest członkiem zespołu. Następnie obiekt komentarza zostaje stworzony, powiązany z wersją, zapisany do bazy, a na koniec zwrócony jako obiekt DTO do kontrolera.

Listing 11 – funkcja tworzenia komentarza. Opracowanie własne.

```
@Transactional
public CommentGetDTO createComment(String text, Long tcStart, Long tcEnd,
Long versionId, User userContext) {
    if (tcStart > tcEnd) {
        Long temp = tcStart;
        tcStart = tcEnd;
        tcEnd = temp;
    }
    Version version = versionService.getVersionById(versionId);
    isMemberOfBand(userContext, version);
    User user =
userService.getUserByGoogleId(userContext.getGoogleId()).orElseThrow();

    Comments comments = new Comments(text, tcStart, tcEnd, version, user);
    version.addComment(comments);
    commentRepository.save(comments);

    return new CommentGetDTO(
        comments.getId(),
        comments.getVersion().getId(),
        comments.getText(),
        comments.getTcStart(),
        comments.getTcEnd(),
```

```

        user.getId(),
        user.getName(),
        user.getPictureUrl(),
        comments.getCreateDate());
    }

```

Z biznesowego punktu widzenia, ważne jest zarządzanie rolami. W systemie WaveScope funkcjonują trzy role użytkowników: MUSICIAN, PRODUCER, OWNER. Każda z tych ról z założenia ma dostęp do innych funkcji. Takim przykładem jest funkcja `updateProjectNameAndDesc` (Listing 12). Zawiera ona weryfikację czy użytkownik wywołujący ten endpoint posiada uprawnienia producenta lub właściciela (Listing 13) i dopiero gdy dostęp zostanie potwierdzony, nazwa i opis zostają zmienione.

Listing 12 – funkcja aktualizująca nazwę i opis projektu dostępna tylko dla producenta i właściciela. Opracowanie własne.

```

@Transactional
public ProjectDTO updateProjectNameAndDesc(Long projectId, Long bandId,
String newName, String newDesc, User userContext) {
    Project project = getProjectById(projectId);
    Band band = bandRepository.findById(bandId).orElseThrow();
    User user =
userService.getUserByGoogleId(userContext.getGoogleId()).orElseThrow(() ->
new RuntimeException("User not found"));

    List<ApprovedUserDTO> approvedUsers = mapApprovalsToDTO(project);

    if (bandMembersService.isProducerOrOwner(user, band)) {
        project.setName(newName);
        project.setDescription(newDesc);
        return new ProjectDTO(project.getName(), project.getId(),
project.getStatus(), project.getDescription(), approvedUsers);
    } else {
        throw new IllegalArgumentException("You are not authorized to
update this project");
    }
}

```

Listing 13 – funkcja sprawdzająca uprawnienia. Opracowanie własne.

```

public Boolean isProducerOrOwner(User user, Band band) {
    return bandMembersRepository.isProducerOrOwner(band.getId(),
user.getId());
}

```

Ciekawą częścią logiki biznesowej jest moduł odpowiedzialny za zarządzanie statusem utworu. Funkcja `toggleApproval` (Listing 14) jest wywoływana, gdy tylko jakiś użytkownik kliknie przycisk do zatwierdzania utworu. Sprawdza ona, czy użytkownik zatwierdza utwór, czy oznacza jako niezatwierdzony (cofa kliknięcie zatwierdzenia). Następnie odpowiednio ustawia status na przeciwny od aktualnego. Po tym sprawdzeniu następuje wywołanie funkcji `checkAndSetCompleteStatus` (Listing 15) sprawdzającej czy wszyscy członkowie zespołu zatwierdzili dany utwór. Jeżeli ten warunek zostanie spełniony, status takiego utworu zmienia się na zatwierdzony.

Listing 14 – funkcja odpowiedzialna za obsługę zatwierdzania projektu. Opracowanie własne

```

@Transactional
public Project toggleApproval(Long projectId, Long bandId, User

```

```

userContext){
    Project project = getProjectById(projectId);
    User user =
userService.getUserByGoogleId(userContext.getGoogleId()).orElseThrow(() ->
new RuntimeException("User not found"));
    BandMember member =
bandMembersService.getBandMemberByUserIdAndBandId(user.getId(), bandId);

    if (project.isMemberApproved(member.getId())){
        project.removeApproval(member.getId());
        if (project.getStatus() == ProjectStatus.COMPLETED){
            project.setStatus(ProjectStatus.IN_PROGRESS);
        }
    } else {
        project.addApproval(member.getId());
    }
    checkAndSetCompleteStatus(project);

    return projectRepository.save(project);
}

```

Listing 15 – funkcja zarządzająca statusem projektu. Opracowanie własne.

```

private void checkAndSetCompleteStatus(Project project){
    int totalBandMembers = project.getBand().getBandMembers().size();
    int totalApprovals = project.getApprovedByMemberIds().size();
    if (totalApprovals >= totalBandMembers){
        project.setStatus(ProjectStatus.COMPLETED);
    } else {
        project.setStatus(ProjectStatus.IN_PROGRESS);
    }
}

```

Jedną z kluczowych funkcjonalności zdefiniowanych w wymaganiach funkcjonalnych (rozdział 3.3.1) jest wersjonowanie plików. Zostało ono zrealizowane przez funkcję `createNewVersion` (Listing 16), która jest wywoływana podczas dodawania nowej wersji. Na początku funkcja ta standardowo pobiera i weryfikuje powiązane encje oraz uprawnienia użytkownika (`user`, `band`, `isProducerOrOwner()`, `project`), a następnie resetuje status utworu (funkcja `resetApprovals`). Następnie wyznaczany jest numer nowej wersji w oparciu o stan bazy. Odpowiednio przygotowany plik jest zapisywany na dysku Google lub na serwerze, w zależności od tego, jak został skonfigurowany zespół. Jeżeli wszystkie dotychczasowe kroki zakończą się powodzeniem, to zostaną zaktualizowane flagi (`currentVersion`) dla starych wersji oraz zostanie utworzona i utrwalona nowa wersja w bazie danych.

Listing 16 – funkcja dodawania nowej wersji utworu odpowiedzialna za wersjonowanie. Opracowanie własne.

```

@Transactional
public Version createNewVersion(Long projectId, Long bandId, String
description, MultipartFile file, User userContext) throws IOException {
    User user =
userService.getUserByGoogleId(userContext.getGoogleId()).orElseThrow();
    Band band = bandRepository.findById(bandId).orElseThrow();

    if (!bandMembersService.isProducerOrOwner(user, band)){throw new
IllegalArgumentException("You are not authorized to create versions for
this band");}

    Project project = projectRepository.findById(projectId).orElseThrow();
}

```

```

        project.resetApprovals();

        Integer maxVersion =
versionRepository.findMaxVersionByProjectId(projectId);

        int newVersionNumber = (maxVersion == null) ? 1 : maxVersion + 1;

        String savedFileName;
        if (band.isUsingGoogleDrive()){
            savedFileName =
googleDriveService.uploadAudioFile(user.getGoogleRefreshToken(), file);
        } else {
            savedFileName = storageService.storeProjectFile(
                file,
                bandId,
                projectId,
                ProjectSubFolder.VERSIONS
            );
        }

        if (newVersionNumber > 1){
            List<Version> oldVersions =
versionRepository.findAllByProjectId(projectId);
            for (Version v : oldVersions) {
                v.setCurrentVersion(false);
            }
            versionRepository.saveAll(oldVersions);
        }
        Version version = new Version(
            project,
            newVersionNumber,
            description,
            savedFileName,
            file.getOriginalFilename(),
            file.getContentType(),
            user
        );

        version.setCurrentVersion(true);

        return versionRepository.save(version);

```

5.1.6 Zarządzanie plikami i integracja z zewnętrznym API Google Drive

Z uwagi na specyfikę systemu WaveScope, pracującego z plikami multimedialnymi, krytycznym elementem warstwy serwerowej jest optymalne i bezpieczne zarządzanie zasobami binarnymi. Wychodząc naprzeciw różnorodnym potrzebom zespołów produkcyjnych, system wprowadza hybrydowy model przechowywania danych. W zależności od konfiguracji zespołu, pliki mogą być utrwalane w lokalnym systemie plików serwera lub strumieniowane bezpośrednio do prywatnej przestrzeni dyskowej użytkownika za pośrednictwem chmury Google Drive.

Zarządzanie przestrzenią lokalną zostało zaimplementowane w klasie `StorageService`. W celu zapewnienia skalowalności i hermetyzacji danych pomiędzy poszczególnymi podmiotami korzystającymi z systemu, zaprogramowano hierarchiczną strukturę katalogów. Każdy zespół, projekt oraz typ pliku posiadają dedykowaną ścieżkę w systemie operacyjnym (np. `/band_{id}/project_{id}/VERSIONS`).

Odbierane w żądaniach HTTP pliki, reprezentowane przez interfejs `MultipartFile`, są zapisywane na dysku twardym przy użyciu klasy `java.nio.file.Files` [72]. Z inżynierskiego

punktu widzenia kluczowe jest tu zastosowanie algorytmu UUID (*Universally Unique Identifier*) [73] do zmiany nazw plików wejściowych. Eliminuje to powszechny problem nadpisywania się danych w sytuacji, gdy różni użytkownicy przesyłają pliki o takich samych, trywialnych nazwach (np. „v1.wav” lub „mix.mp3”). Do udostępniania plików warstwie frontendowej wykorzystano obiekt `FileSystemResource`, co zapewnia bezpieczny dostęp do zasobów dyskowych. Implementację zapisu pliku na serwerze przedstawia *Listing 17*.

*Listing 17 – implementacja zapisu pliku na serwerze z wykorzystaniem unikalnych identyfikatorów UUID.
Opracowanie własne.*

```
public String storeProjectFile(MultipartFile file, Long bandId, Long
projectId, ProjectSubFolder subFolder) {
    String originalFilename = file.getOriginalFilename();
    String extensions = originalFilename != null &&
originalFilename.contains(".")
        ? originalFilename.substring(originalFilename.lastIndexOf("."))
        : "";
    String uniqueFileName = UUID.randomUUID().toString()+extensions;

    try {
        Path targetFolder = fileStorageLocation
            .resolve("band_"+bandId)
            .resolve("project_"+projectId)
            .resolve(subFolder.folderName);

        if (!Files.exists(targetFolder)) {
            Files.createDirectories(targetFolder);
        }

        Path targetPath = targetFolder.resolve(uniqueFileName);
        Files.copy(file.getInputStream(), targetPath,
StandardCopyOption.REPLACE_EXISTING);

        return uniqueFileName;
    } catch (IOException e) {
        throw new RuntimeException("Could not store file " +
originalFilename + " for band " + bandId);
    }
}
```

Alternatywnym, docelowym dla wielu profesjonalnych środowisk rozwiązaniem, jest wykorzystanie zewnętrznej infrastruktury firmy Google. Moduł `GoogleDriveService` został zaprojektowany w oparciu o oficjalną bibliotekę *Google API Client Library* [74] i wykorzystuje autoryzację typu bezstanowego. Dla każdego żądania konstruowany jest nowy obiekt dostępowy `Drive`, uwierzytelniany tokenem odświeżania (`userRefreshToken`), powiązany z kontem danego użytkownika. Dzięki przechowywaniu tego tokenu w bazie danych, system `WaveScope` działa jako zaufany pośrednik. Gdy uprawniona osoba chce odtworzyć plik, serwer używa tokenu właściciela, aby wylegitymować się przed API Google w jego imieniu. Pozwala to na pobranie i przekazanie strumienia audio bezpośrednio do słuchacza, całkowicie z pominięciem konieczności publicznego udostępniania plików na dysku chmurowym.

Na szczególną uwagę zasługuje inżynierskie podejście do separacji plików w usłudze chmurowej (*Listing 18*). Zamiast umieszczać utwory w głównym, widocznym dla użytkownika katalogu `Google Drive`, aplikacja `WaveScope` wykorzystuje specjalną, ukrytą przestrzeń dedykowaną aplikacjom – `appDataFolder`. Gwarantuje to, że pliki systemowe nie wprowadzają chaosu w prywatnych dokumentach klienta, a jednocześnie pozostają w pełni zarządzalne przez warstwę serwerową `WaveScope`.

Listing 18 – funkcja do przesyłania pliku do przestrzeni appDataFolder. Opracowanie własne.

```
public String uploadAudioFile(String userRefreshToken, MultipartFile
multipartFile) throws IOException {
    Drive driveService = getDriveService(userRefreshToken);

    String fileName = multipartFile.getOriginalFilename();
    String mimeType = multipartFile.getContentType();
    InputStream fileStream = multipartFile.getInputStream();

    File fileMetadata = new File();
    fileMetadata.setName(fileName);

    fileMetadata.setParents(Collections.singletonList("appDataFolder"));

    // Treść pliku przygotowana dla Google API
    InputStreamContent mediaContent = new InputStreamContent(mimeType,
fileStream);

    // Wykonanie uploadu na serwery Google
    File file = driveService.files().create(fileMetadata, mediaContent)
        .setFields("id")
        .execute();

    return file.getId();
}
```

Zastosowanie API chmurowego wymusiło również odpowiednią optymalizację procesu odczytu. W metodzie `downloadFile` (Listing 19) zastosowano technikę strumieniowania danych (*Data Streaming*). Plik pobierany z Google Drive nie jest w całości ładowany do pamięci operacyjnej (RAM) serwera WaveScope. Pobrany z chmury `InputStream` jest natychmiast przekazywany do odpowiedzi HTTP poprzez opakowanie go w obiekt `InputStreamResource`. Dzięki temu rozwiązaniu, system zachowuje wysoką wydajność operacyjną nawet przy obsłudze wielu jednoczesnych żądań pobierania dużych plików w formacie WAV bez ryzyka wystąpienia błędu `OutOfMemoryError`. Podczas korzystania z danych przechowywanych na dysku Google warstwa serwerowa systemu WaveScope działa jako proxy [75], będąc pośrednikiem między klientem, a dyskiem.

Listing 19 – funkcja pobierająca strumień danych z Google Drive i przysyłająca go do punktu końcowego REST API. Opracowanie własne.

```
public Resource downloadFile(String userRefreshToken, String fileId) throws
IOException {
    Drive driveService = getDriveService(userRefreshToken);

    // Pobranie strumienia danych z Google Drive
    InputStream inputStream =
driveService.files().get(fileId).executeMediaAsInputStream();
    // Opakowanie strumienia w obiekt Resource ze Springa
    return new InputStreamResource(inputStream);
}
```

Istotnym aspektem integracji z systemami rozproszonymi jest obsługa wyjątków i nieprzewidywalnych zachowań (Listing 20). W procesie usuwania plików (`deleteFile`) zaimplementowano dedykowany mechanizm obronny. Jeżeli plik został usunięty z dysku manualnie (z pominięciem systemu WaveScope), API Google zwraca błąd 404 (*Not Found*). Kod ten jest celowo przechwytywany (`GoogleJsonResponseException`) i ignorowany. Takie podejście zapobiega

całkowitemu zablokowaniu transakcji w bazie PostgreSQL, pozwalając na pomyślne usunięcie rekordu o projekcie z systemu WaveScope pomimo braku pliku źródłowego na zdalnym serwerze.

Listing 20 – implementacja bezpiecznego usuwania. Opracowanie własne.

```
public void deleteFile(String userRefreshToken, String fileId) throws
IOException {
    Drive driveService = getDriveService(userRefreshToken);

    try {
        driveService.files().delete(fileId).execute();
    } catch (GoogleJsonResponseException e) {
        if (e.getStatusCode() == 404) {
            System.out.println("Plik o ID " + fileId + " nie istnieje na
Google Drive. Pomijam usuwanie z dysku.");
        } else {
            throw e;
        }
    }
}
```

5.2. Implementacja części klienckiej – frontend

Warstwa kliencka systemu WaveScope odpowiada za prezentację danych, asynchroniczną interakcję z interfejsem programistycznym REST API oraz realizację najbardziej unikalnej funkcjonalności systemu – precyzyjnej, graficznej wizualizacji fali dźwiękowej. Interfejs użytkownika został zaimplementowany w oparciu o bibliotekę React (w wersji 19.2.0) [9] przy użyciu języka TypeScript (w wersji 4.9.5) [10]. Pozwoliło to na stworzenie responsywnej aplikacji typu SPA (*Single Page Application*) [36], która gwarantuje płynność działania oraz brak konieczności przeładowywania witryny podczas odsłuchu i recenzowania materiałów audio.

Głównym wyzwaniem na etapie prac nad frontendem było zaprojektowanie wydajnej struktury komponentów, która bez opóźnień synchronizuje globalny stan aplikacji z dynamicznymi zdarzeniami generowanymi przez odtwarzacz muzyczny. Warstwa wizualna musi w czasie rzeczywistym przetwarzać współrzędne czasowe, renderować markery komentarzy bezpośrednio na osi czasu oraz zapewniać intuicyjną nawigację niezależnie od rozdzielczości ekranu urządzenia docelowego.

W poniższych podpodrozdziałach szczegółowo opisano architekturę komponentową projektu, mechanizmy zarządzania stanem globalnym i lokalnym, asynchroniczną obsługę zapytań sieciowych, a także integrację z zaawansowanym silnikiem odtwarzacza biblioteki wavesurfer.js [11] oraz konfigurację routingu.

5.2.1 Architektura komponentowa i struktura projektu React

Implementacja warstwy klienckiej systemu WaveScope została oparta na architekturze zorientowanej na komponenty autonomiczne (*Smart Components*) [76]. W podejściu tym poszczególne elementy interfejsu użytkownika integrują w sobie nie tylko warstwę prezentacji (szablony TSX) i style CSS, ale również własną logikę komunikacji z interfejsem REST API.

Decyzja o wyborze takiej architektury była podyktowana chęcią uzyskania wysokiej modularności oraz reużywalności kodu. Dzięki temu, że komponenty takie jak `Members.tsx` czy `Albums.tsx` samodzielnie zarządzają swoim stanem i bezpośrednio odpytują punkty końcowe serwera, są one całkowicie niezależnymi jednostkami biznesowymi. Pozwala to na ich łatwe osadzanie w dowolnym miejscu aplikacji oraz upraszcza strukturę stron nadrzędnych (*pages*), które pełnią w systemie wyłącznie rolę bezstanowych kontenerów odpowiedzialnych za kompozycję widoków i układ graficzny (*layout*).

W celu zachowania przejrzystości tak zaprojektowanego systemu, struktura katalogów wewnątrz folderu źródłowego `src` została ustrukturyzowana tematycznie:

- **components** – centralne repozytorium reużywalnych, autonomicznych komponentów aplikacji (wraz z odpowiadającymi im plikami stylów `.css`). Katalog zawiera również podfolder `projectComponents` zawierający elementy dedykowane wyłącznie obsłudze głównego widoku odtwarzacza.
- **pages** – warstwa odpowiedzialna za definicję widoków aplikacji (np. pulpit nawigacyjny wewnątrz zespołu, widok projektu). Strony w tej warstwie agregują komponenty z pakietu `components` i organizują je w spójne ekrany.
- **context** – pakiet przechowujący konteksty aplikacji, służące do globalnego współdzielenia danych.
- **api** – katalog zawierający konfigurację bazową klienta HTTP (`Axios`).
- **types** – dedykowany folder zawierający interfejsy i typy języka TypeScript, odzwierciedlające struktury obiektów DTO przesyłanych z backendu.
- **assets** – katalog przechowujący statyczne zasoby multimedialne, takie jak logotypy, pliki graficzne oraz pliki audio służące do demonstracji możliwości aplikacji.
- **Mobile** – odizolowana przestrzeń przeznaczona na komponenty dedykowane specyfice urządzeń mobilnych.

Kluczowym elementem pakietu `components` są również pliki `AuthCallback.tsx` – odpowiedzialny za przechwycenie sesji po przekierowaniu z protokołu OAuth2 [26], oraz `ProjectPlayerDemo.tsx` – stanowiący odizolowaną, demonstracyjną wersję odtwarzacza dla użytkowników niezalogowanych. Prezentowany podział strukturalny zapewnił pełną kontrolę nad cyklem życia każdego elementu interfejsu oraz pozwolił na sprawne rozwijanie warstwy wizualnej WaveScope.

5.2.2 Zarządzanie stanem aplikacji

Do przechowywania danych współdzielonych pomiędzy wieloma komponentami, na różnych poziomach drzewa aplikacji wykorzystano wbudowany w React Context API [77]. Został on zaimplementowany w pliku `AuthContext.tsx` i jest odpowiedzialny za autoryzację i sesję użytkownika. Kontekst (*Listing 21*) przechowuje informacje o bieżącym użytkowniku (`user`), statusie ładowania sesji (`loading`), a także udostępnia globalne funkcje takie jak `logout` czy `refreshUser`.

Listing 21 – kontekst odpowiedzialny za autoryzację. Źródło: Opracowanie własne.

```
export const AuthProvider: React.FC<AuthProviderProps> = ({ children }) =>
{
  const [user, setUser] = useState<User | null>(null);
  const [loading, setLoading] = useState(true);

  const refreshUser = async () => {
    console.log("refreshUser called");
    try {
      const res = await api.get<User>("/api/users/me");
      console.log("USER FROM BACKEND:", res.data);
      setUser(res.data);
    } catch(err) {
      console.log("refreshUser error:", err);
      setUser(null);
    }
  };

  useEffect(() => {
```

```

        refreshUser().finally(() => setLoading(false));
    }, []);

    const logout = async () => {
        try {
            await api.post("/api/users/logout");
            console.log("User logged out");
            setUser(null);
        } finally {
            window.location.href = "/";
        }
    };

    return (
        <AuthContext.Provider value={{ user, loading, logout, refreshUser }}>
            {children}
        </AuthContext.Provider>
    );
};

```

Kontekst ten jest wstrzykiwany na najwyższym poziomie aplikacji (w App.tsx) dzięki czemu każdy komponent może łatwo uzyskać do niego dostęp za pomocą niestandardowego hooka `useAuth` (Listing 22). Eliminuje to problem drążenia właściwości (*prop drilling*) [78] w przypadku danych sesyjnych.

Listing 22 – hook dający dostęp do stanu aplikacji. Źródło: Opracowanie własne

```

export const useAuth = () => {
    const context = useContext(AuthContext);
    if (!context) throw new Error("useAuth must be used inside AuthProvider");
    return context;
};

```

Stan lokalny, specyficzny dla konkretnego widoku lub logiki biznesowej danego komponentu, jest kontrolowany przy użyciu natywnego hooka `useState`. Zgodnie z inżynierską zasadą rozdziału odpowiedzialności, poszczególne komponenty systemu są świadome wyłącznie tych danych, które są im bezpośrednio niezbędne do prawidłowego wyrenderowania interfejsu użytkownika lub obsłużenia akcji w danym bloku kodu.

Doskonałym przykładem zaawansowanego zarządzania złożonym stanem lokalnym oraz procesami asynchronicznymi jest autonomiczny komponent Projects (Listing 23). Analiza jego implementacji pozwala wyróżnić dwie główne grupy stanów lokalnych, które odpowiadają za płynne zachowanie interfejsu:

- **Stany danych biznesowych:** Stan `projects` (przechowujący tablicę obiektów `ProjectDTO[]`) oraz `albums` (tablica `AlbumDTO[]`) reprezentują aktualne dane pobrane z bazy poprzez API. Dodatkowo tworzona jest flaga `loading`, która blokuje interakcję z komponentami przed pełnym załadowaniem zasobów.
- **Stany formularzy i interakcji użytkownika:** Do obsługi operacji CRUD (tworzenie nowego projektu wraz z asynchronicznym wysyłaniem binarnego pliku audio) komponent bezstanowo śledzi dane wejściowe formularza za pomocą zmiennych takich jak `newName`, `newDescription`, `versionDescription` oraz `selectedFile` (przechowujący natywny obiekt pliku `File`).

Listing 23 – fragment inicjalizacji stanów w komponencie Projects. Źródło: Opracowanie własne.

```
const Projects: React.FC<ProjectsProps> = ({
  refreshTrigger, selectedAlbumId, selectedAlbumName, onDataChange, userRole,
  bandId,
}) => {
  // --- STANY DANYCH BIZNESOWYCH ---
  const [projects, setProjects] = useState<ProjectDTO[]>([]);
  const [albums, setAlbums] = useState<AlbumDTO[]>([]);
  const [loading, setLoading] = useState(true);

  // --- STANY TWORZENIA ---
  const [isModalOpen, setIsModalOpen] = useState(false);
  const [newName, setNewName] = useState("");
  const [newDescription, setNewDescription] = useState("");
  const [versionDescription, setVersionDescription] = useState("");
  const [selectedFile, setSelectedFile] = useState<File | null>(null);
  const [modalSelectedAlbumId, setModalSelectedAlbumId] =
    useState<string>("");

  // --- STAN PRZYPISYWANIA ---
  const [isAssignModalOpen, setIsAssignModalOpen] = useState(false);
  const [projectToAssignId, setProjectToAssignId] = useState<number |
null>(null);
  const [assignTargetAlbumId, setAssignTargetAlbumId] =
    useState<string>("");

  // --- STAN ZMIANY NAZWY---
  const [isRenameModalOpen, setIsRenameModalOpen] = useState(false);
  const [renameTargetId, setRenameTargetId] = useState<number |
null>(null);
  const [renameNewName, setRenameNewName] = useState("");
  const [renameNewDescription, setRenameNewDescription] = useState("");

  const [actionLoading, setActionLoading] = useState(false);
  const fileInputRef = useRef<HTMLInputElement>(null);

  // --- STAN MENU ---
  const [openMenuId, setOpenMenuId] = useState<number | null>(null);
```

Przełączanie stanów odbywa się w sposób reaktywny. Przykładowo, po pomyślnym wykonaniu asynchronicznego żądania sieciowego POST w metodzie `handleAddProject` (Listing 24), system nie odświeża całej strony. Zamiast tego, przy użyciu metody `setProjects`, nowo utworzony obiekt DTO jest dynamicznie dopisywany do istniejącej tablicy w pamięci przeglądarki, co wywołuje natychmiastowe i płynne przerysowanie (*re-render*) wyłącznie listy utworów.

Listing 24 – dopisanie danych do istniejącej tablicy. Źródło: Opracowanie własne.

```
const handleAddProject = async () => {
  if (!newName.trim()) return;
  if (!selectedFile) { alert("Proszę wybrać plik audio."); return; }
  try {
    setActionLoading(true);
    // ... pobranie danych z REST API

    const addedToCurrentView = (!targetAlbumId && (!selectedAlbumId ||
selectedAlbumId === -1)) || (targetAlbumId === selectedAlbumId);
    if (addedToCurrentView) setProjects((prev) => [...prev, res.data]);
    if (onDataChange) onDataChange();
  }
```

```

    // ... czyszczenie stanu
  } catch (err: any) {
    console.error(err);
    alert("Błąd podczas dodawania projektu");
  } finally {
    setActionLoading(false);
  }
};

```

Takie podejście do zarządzania stanem aplikacji gwarantuje, że interfejs użytkownika systemu WaveScope działa w sposób przewidywalny, wydajny i natychmiastowo reaguje na akcje klienta.

5.2.3 Asynchroniczna komunikacja z serwerem i obsługa zapytań

Komunikacja sieciowa pomiędzy warstwą kliencką a serwerową systemu WaveScope odbywa się w sposób w pełni asynchroniczny, co pozwala na nieprzerwane działanie interfejsu użytkownika podczas pobierania lub wysyłania danych. Do realizacji żądań HTTP wykorzystano bibliotekę `Axios` [79], na bazie, której stworzono dedykowaną, scentralizowaną instancję konfiguracji (obiekt `api`). Instancja ta automatycznie zarządza konfiguracją adresu bazowego (Base URL) serwera oraz, co kluczowe dla bezpieczeństwa, obsługuje nagłówek `withCredentials: true`, umożliwiając automatyczne dołączanie ciasteczek `HttpOnly` zawierających token JWT do każdego zapytania sieciowego.

W architekturze autonomicznych komponentów, proces komunikacji z API i cykl przetwarzania żądań realizowany jest według ściśle zdefiniowanego inżynierskiego schematu, co doskonale obrazuje metoda pobierania danych w komponencie `Members` (Listing 25):

- **Inicjalizacja i sygnalizacja (blok try):** Przed uruchomieniem zapytania asynchronicznego stan `loading` ustawiany jest na wartość prawdziwą (`true`), co pozwala obsłużyć w interfejsie moment pobierania danych, np. poprzez pokazanie wskaźnika ładowania (*spinnera*). Następnie przy użyciu składni `await` wywoływana jest odpowiednia metoda `http` (w tym przypadku `api.get`).
- **Aktualizacja reaktywna:** Po pomyślnym odebraniu odpowiedzi struktura danych w formacie JSON, dzięki zastosowaniu DTO i zgodności typów, jest przekazywana do funkcji modyfikującej stan lokalny, co natychmiastowo aktualizuje widok.
- **Obsługa błędów (blok catch):** Ewentualne błędy sieciowe lub usterki po stronie serwera są przechwytywane i logowane w konsoli, chroniąc aplikację przed nagłym zatrzymaniem działania.
- **Finalizacja stanu (blok finally):** Niezależnie od wyniku operacji, na samym końcu flaga `loading` jest resetowana do wartości fałszywej (`false`), co przywraca pełną interaktywność interfejsu.

Listing 25 – funkcja pobierania danych z API. Źródło: Opracowanie własne.

```

const fetchMembers = async () => {
  try {
    setLoading(true);
    const res = await
api.get<BandMemberDTO[]>(`/api/bands/${bandId}/members`);
    setMembers(res.data);
  } catch (err) {
    console.error(err);
  } finally {
    setLoading(false);
  }
};

```

Wyjątkowym wyzwaniem implementacyjnym w systemie WaveScope, była obsługa żądań modyfikujących zasoby. Ze względu na specyfikę aplikacji, przykładowo funkcja dodawania nowego projektu wymaga jednoczesnego przesłania danych ustrukturyzowanych (metadane projektu w formacie JSON) oraz surowego pliku binarnego audio (obiekt `File`).

Aby zrealizować to zadanie w ramach jednego asynchronicznego zapytania http, zaimplementowano mechanizm oparty na obiekcie `FormData` oraz kodowaniu typu `multipart/form-data` (*Listing 26*). Ponieważ standardowe przesyłanie wieloczęściowe natywnie obsługuje pliki, ale bywa problematyczne przy złożonych obiektach, zastosowano zaawansowane podejście inżynierskie: dane tekstowe projektu (struktura `CreateProjectDTO`) zostały spakowane do obiektu typu `Blob` z jawnie zdefiniowanym typem zawartości `application/json`, a następnie dołączone do formularza obok strumienia pliku audio.

Listing 26 – funkcja wysyłania danych z zasobami do API. Źródło: Opracowanie własne.

```
const handleAddProject = async () => {
  // ... Walidacja danych wejściowych
  try {
    setActionLoading(true);
    const formData = new FormData();
    formData.append("file", selectedFile);

    let targetAlbumId: number | null = null;
    if (modalSelectedAlbumId === "-1") targetAlbumId = null;
    else if (modalSelectedAlbumId) targetAlbumId =
Number(modalSelectedAlbumId);
    else if (selectedAlbumId && selectedAlbumId !== -1) targetAlbumId =
selectedAlbumId;

    const projectDto: CreateProjectDTO = {
      name: newName,
      projectDescription: newDescription,
      versionDescription: versionDescription,
      albumId: targetAlbumId
    };

    const jsonBlob = new Blob([JSON.stringify(projectDto)], { type:
'application/json' });
    formData.append("project", jsonBlob , "project.json");

    const res = await
api.post<ProjectDTO>(`/api/bands/${bandId}/projects`, formData, {
      headers: { "Content-Type": "multipart/form-data" }
    });
    // ... Czyszczenie stanu i złapanie błędów
  }
};
```

Wartym uwagi jest również sposób pobierania treści audio do klienta. Odbywa się to na podobnej zasadzie co wysyłanie pliku dźwiękowego na serwer, czyli odbiera się dane w formacie binarnym (`blob`). Wywołanie `await api.get` zawiesza wykonywanie kodu do momentu, aż cały plik audio zostanie przesłany przez sieć do przeglądarki. W tym czasie użytkownikowi prezentowane jest kółko ładowania (*spinner*) informujące, że trwa ładowanie. Po zakończeniu pobierania, plik audio jest przekazywany do odtwarzacza wykorzystującego `wavesurfer.js`. Świadomie zastosowano podejście wymuszające pobranie całego pliku do pamięci przeglądarki z uwagi na chęć pokazania użytkownikowi całej fali audio w momencie, kiedy zyskuje do niej dostęp. Przykład pobierania pliku audio prezentuje *Listing 27*.

Listing 27 – fragment funkcji pobierającej plik i informacje o projekcie. Źródło: Opracowanie własne.

```
if (targetVersionId) {
  setAudioLoading(true);
  try {
    const audioRes = await
api.get(`\api/bands/${bandId}/projects/${projectId}`, {
  responseType: 'blob',
  params: {
    t: new Date().getTime(),
    download: false
  }
});

    const blobUrl = URL.createObjectURL(audioRes.data);
    setAudioUrl(blobUrl);
  } catch (audioErr) {
    console.error("Błąd pobierania audio:", audioErr);
  } finally {
    setAudioLoading(false);
  }
}
```

5.2.4 Integracja odtwarzacza audio i waveform

Sercem warstwy wizualnej systemu WaveScope, odróżniającym go od klasycznych rozwiązań chmurowych, jest interaktywny interfejs analizy materiałów dźwiękowych. Za wygenerowanie, renderowanie i obsługę zdarzeń związanych z falą dźwiękową (*waveform*) odpowiada dedykowany komponent `WaveformPanel.tsx`. Został on zbudowany w oparciu o zaawansowaną bibliotekę `wavesurfer.js`, która do prawidłowego działania w ekosystemie React wymagała zastosowania specyficznych technik zarządzania cyklem życia komponentu oraz drzewem DOM [37].

Proces tworzenia graficznej reprezentacji dźwięku rozpoczyna się od wywołania metody fabrykującej `WaveSurfer.create` (Listing 28), do której wstrzykiwane są dane konfiguracyjne. Z punktu widzenia integracji biblioteki `wavesurfer.js` i frameworka React, głównym wyzwaniem jest zapewnienie bezpośredniego dostępu do fizycznych elementów HTML. W systemie zrealizowano to przy użyciu hooka `useRef`. Przechwycona referencja jest przekazywana do konfiguracji jako parametr, co pozwala bibliotece na wstrzyknięcie zoptymalizowanego komponentu `<canvas>` bezpośrednio do drzewa DOM bez zaburzania procesów renderowania React.

Podczas konfiguracji instancji `wavesurfer.js` należy zdefiniować podstawowe parametry wizualne. W przypadku aplikacji WaveScope wysokość obszaru roboczego została ustalona na 160 pikseli. Interakcja użytkownika jest wspierana przez kodowanie kolorystyczne, gdzie kontrastujące barwy odróżniają bazową falę całego nagrania od fragmentu (`waveColor`), który już został odtworzony (`progressColor`).

Listing 28 – inicjalizacja komponentu `wavesurfer.js`. Źródło: Opracowanie własne.

```
const ws = WaveSurfer.create({
  container: containerRef.current,
  waveColor: "#8a21e3",
  progressColor: "#007bff",
  height: 160,
  autoScroll: true,
  autoCenter: true,
  minPxPerSec: 0,

  backend: "MediaElement",
  plugins: [
```

```

    regionsPlugin,
    TimelinePlugin.create({
      container: timelineRef.current,
      height: 20,
      style: { fontSize: '10px', color: '#6b7280' },
    })
  ],
});

```

Kluczową inżynierską decyzją projektową na etapie konfiguracji było wymuszenie użycia alternatywnego silnika odtwarzania poprzez ustawienie parametru `backend` na wartość `"MediaElement"`. W takiej architekturze interfejs Web Audio API wykorzystywany jest przez bibliotekę wyłącznie w fazie inicjalizacji do wyliczenia wartości szczytowych (*peaks*), niezbędnych do wygenerowania geometrycznego wykresu fali. Samo dekodowanie i odtwarzanie strumienia audio zostało natomiast oddelegowane do natywnego tagu HTML5 `<audio>`. Rozwiązanie to drastycznie zmniejsza zużycie pamięci operacyjnej (RAM) przeglądarki i optymalizuje zarządzanie buforem przy długich plikach.

Wizualizacja fali dźwiękowej jest funkcjonalnie uzupełniana przez miarkę czasu, zrealizowaną za pomocą oficjalnego rozszerzenia `TimelinePlugin`. W celu zachowania modularności interfejsu i rozdzielenia akcji wywoływanych przez kliknięcie, oś czasu została przesunięta nad falę dźwięku oraz odizolowana od przedziałów komentarzy przez nadanie odpowiednich stylów. Z racji domyślnego zamknięcia osi czasu w jednym kontenerze razem z falą audio i komentarzami, zaimplementowano filtr, który pozwala odróżnić te przestrzenie od siebie i inne różne akcje (Listing 29). Oś czasu nie pozwala na zaznaczanie fragmentu, ale za to umożliwia nawigację wewnątrz komentarza.

Listing 29 – Filtr zdarzeń separujący logikę osi czasu od fali dźwiękowej. Źródło: Opracowanie własne.

```

const handleContainerMouseDown = (e: React.MouseEvent<HTMLDivElement>) => {
  if (!containerRef.current) return;
  const rect = containerRef.current.getBoundingClientRect();
  const y = e.clientY - rect.top;
  if (y <= 21) {
    return;
  }
  onMouseDown(e);
};

```

Najbardziej złożonym elementem implementacji warstwy klienckiej, a przy okazji jednym z głównych wymagań funkcjonalnych, jest system asynchronicznych recenzji nanoszonych bezpośrednio na obszar widma dźwiękowego. Do wizualnej reprezentacji czasu trwania poszczególnych uwag wykorzystano dodatek `RegionsPlugin`, który rejestrowany jest w cyklu życia obiektu `wavesurfer` na etapie jego inicjalizacji (Listing 28).

Aby zapobiec niekontrolowanemu nakładaniu się zdarzeń, domyślna funkcja wbudowanego przeciągania wtyczki została dezaktywowana (`dragSelection: false`). Zamiast tego zaimplementowano pełną, programową synchronizację stanu regionów z danymi pochodzącymi z API backendu. Po pobraniu listy uwag do danej rewizji utworu uruchamiana jest procedura `syncRegions()`. Czyści ona obszar roboczy, a następnie iteruje po tablicy obiektów DTO. Dla każdego rekordu wywoływana jest metoda `addRegion()` (Listing 30), wstrzykująca dokładne współrzędne czasowe wyciągnięte z bazy danych (`tcStart`, `tcEnd`).

Listing 30 – funkcja synchronizacji komentarzy z regionami biblioteki wavesurfer.js. Źródło: Opracowanie własne.

```
const syncRegions = (currentComments: CommentGetDTO[]) => {
  if (!regionsRef.current) return;
  try { regionsRef.current.clearRegions(); } catch (e) {}
  currentComments.forEach(c => {
    const region = regionsRef.current.addRegion({
      id: String(c.commentId),
      start: c.tcStart,
      end: c.tcEnd,
      color: "rgba(255, 215, 0, 0.2)",
      drag: false,
      resize: false,
      data: { id: c.commentId }
    });
    // ... kod podświetlający region i aktywujący akcję pod kliknięciem
  });
};
```

Wartym uwagi rozwiązaniem architektonicznym w obrębie odtwarzacza jest autorski proces dodawania nowych obszarów recenzji. Zamiast polegać na ograniczeniach natywnego mechanizmu, wprowadzono niestandardową logikę przechwytywania zdarzeń kursora (`onMouseDown`, `onMouseMove`, `onMouseUp`).

Proces dodawania rozpoczyna się od przytrzymania przez użytkownika modyfikatora (`Cmd/Ctrl`). Narzędzie renderuje wówczas w warstwie wierzchniej React tymczasowy element HTML (`dragOverlay`), śledzący ruch myszy na ekranie (Listing 31). Zdarzenie zwolnienia przycisku myszy (Listing 32) uruchamia algorytm, który przelicza relatywne pikselowe współrzędne ekranowe na dokładne wartości milisekund w skali całego utworu audio (z wykorzystaniem metody `ws.getDuration()` i uwzględnieniem aktualnego przesunięcia przewijania fali). Wyliczone wartości stanowią podstawę do stworzenia obiektu transferowego i asynchronicznego wysłania nowej uwagi do serwera (metoda HTTP POST). Dopiero po odebraniu statusu sukcesu z API, interfejs nakazuje wtyczce `RegionsPlugin` nałożenie nowego, trwałego bloku na fali dźwiękowej, co chroni system przed tworzeniem tzw. martwych regionów w przypadku błędu połączenia z siecią.

Listing 31 – funkcja odpowiedzialna za obliczenia podczas zaznaczania. Źródło: Opracowanie własne.

```
const handleMouseMove = (e: React.MouseEvent) => {
  const container = e.currentTarget as HTMLElement;
  const rect = container.getBoundingClientRect();
  const visualX = e.clientX - rect.left;

  if (wsRef.current) {
    const { scrollLeft, scrollWidth } = getWaveInfo(container);
    const absoluteX = visualX + scrollLeft;
    const t = (absoluteX / scrollWidth) * wsRef.current.getDuration();

    const m = Math.floor(t/60);
    const s = Math.floor(t%60);
    setTooltip({pos: visualX,
time: `${m}:${s.toString().padStart(2, '0')}`});
  }

  if (dragStart !== null) {
    setDragWidth(visualX - dragStart);
  }
};
```

Listing 32 - funkcja odpowiedzialna za obliczenie całości zaznaczenia. Źródło: Opracowanie własne.

```
const handleMouseUp = (e: React.MouseEvent) => {
  if (dragStartTime === null || !wsRef.current) {
    setDragStart(null);
    setDragWidth(0);
    setDragStartTime(null);
    return;
  }

  const container = e.currentTarget as HTMLElement;
  const rect = container.getBoundingClientRect();
  const visualX = e.clientX - rect.left;

  // Obliczenie faktycznego czasu dla momentu puszczenia przycisku myszy
  const { scrollLeft, scrollWidth } = getWaveInfo(container);
  const absoluteX = visualX + scrollLeft;
  const endTime = (absoluteX / scrollWidth) *
wsRef.current.getDuration();
  // Czyszczenie zmiennych i przekazanie wyliczonych koordynatów
  czasowych do funkcji dodawania komentarza.
```

5.2.5 Nawigacja wewnątrz aplikacji i routing

Za obsługę nawigacji oraz zarządzanie historią przeglądarki w warstwie klienckiej odpowiada biblioteka `react-router-dom`. Główny silnik nawigacji oparto na komponencie `BrowserRouter`, który w warstwie natywnej wykorzystuje mechanizm HTML5 History API. Takie rozwiązanie stanowi fundament architektury SPA (*Single Page Application*) [36], ponieważ pozwala na płynne przełączanie widoków wyłącznie poprzez manipulację drzewem DOM [37] i zmianę ścieżki w pasku adresu, całkowicie eliminując kosztowne operacje przeładowywania całej witryny z serwera.

Struktura adresacji systemu została zdefiniowana w sposób deklaratywny przy użyciu komponentu `<Routes>`. Ze względu na konieczność obsługi wielu zmiennych zasobów (różne zespoły, projekty), zrezygnowano z adresacji statycznej na rzecz mechanizmu ścieżek dynamicznych (*Dynamic Routing*). Przykład głównej konfiguracji tras aplikacji, uwzględniający parametryzację adresu URL oraz dodatkową warstwę autoryzacji w postaci komponentu `ProtectedRoute` (Listing 33).

Listing 33 – Konfiguracja ścieżek routingu. Źródło: Opracowanie własne.

```
import {BrowserRouter as Router, Routes, Route, Navigate} from "react-
router-dom";
//... inne importy i wrapper do autoryzacji
function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/bands/:bandId" element={<ProtectedRoute><Band
/></ProtectedRoute>} />
        //... Definicje pozostałych widoków
      </div>
    </div>
  </Router>
);
}
```

Ekstrakcja parametrów z powyższych adresów URL (takich jak `:bandId`) wewnątrz komponentów docelowych realizowana jest przy użyciu wbudowanego hooka `useParams`. Pozwala to poszczególnym widokom na precyzyjną identyfikację żadanego zasobu, co stanowi punkt wejścia do zainicjowania procesu asynchronicznego dociągania danych dla konkretnego projektu z API.

5.2.6 Responsywność i adaptacyjny interfejs użytkownika

Zapewnienie ergonomii i czytelności interfejsu na urządzeniach mobilnych stanowiło jedno z wyzwań projektowych. W celu rozwiązania tego problemu, w systemie WaveScope zastosowano architekturę hybrydową, będącą połączeniem podejścia responsywnego (*Responsive Web Design*) [81] dla prostych komponentów, oraz koncepcji projektowania adaptacyjnego (*Adaptive Design*) [82] dla elementów wymagających własnej logiki.

Decyzja ta była podyktowana chęcią poprawy doświadczeń użytkownika z korzystania z aplikacji na mniejszych urządzeniach (np. smartfon). Zdecydowano o ograniczeniu niektórych funkcji takich jak dodawanie komentarzy i wizualizacji audio dla urządzeń mobilnych. Wynikało to z chęci stworzenia z telefonu odtwarzacza materiału opracowanego na komputerze, a nie środowiska, w którym można w pełni pracować z systemem WaveScope.

Techniczna realizacja opiera się na bibliotece `react-responsive` oraz wykorzystaniu hooka `useMediaQuery`. Definiuje on w logice komponentu dynamiczny punkt załamania o wartości 768 pikseli, co stanowi rynkowy standard oddzielający ekrany urządzeń mobilnych od środowisk desktopowych. Na podstawie tej weryfikacji, framework React wykorzystuje mechanizm renderowania warunkowego (*Conditional Rendering*) do wstrzyknięcia do drzewa DOM całkowicie odrębnej, zoptymalizowanej gałęzi komponentów (Listing 34).

Listing 34 – implementacja renderowania warunkowego. Źródło: Opracowanie własne.

```
const Project: React.FC = () => {
  const isMobile = useMediaQuery({ query: '(max-width: 768px)' });
  return (
    isMobile ? (
      <ProjectPlayerMobile/>
    ) : (
      <ProjectPlayer/>
    )));
};
```

Dzięki takiemu podejściu, gdy użytkownik uruchamia aplikację na smartfonie, zmienna `isMobile` przyjmuje wartość prawdziwą (`true`). Wówczas React pobiera i montuje wyłącznie komponent `<ProjectPlayerMobile />`, posiadający odrębną strukturę (brak użycia `wavesurfer.js`). Tak zaplanowana architektura frontendu pozwala na stworzenie interfejsu skrojonego na miarę specyfiki danego urządzenia.

6. Interfejs użytkownika

Niniejszy rozdział poświęcony jest prezentacji końcowego interfejsu graficznego (UI) oraz omówieniu warstwy wizualnej zrealizowanego prototypu systemu WaveScope. Po szczegółowym przeanalizowaniu architektury technicznej oraz procesu implementacji logiki serwerowej i klienckiej w poprzednim rozdziale, ta część pracy skupia się na gotowym produkcie z perspektywy jego docelowego odbiorcy – realizatora dźwięku oraz muzyka.

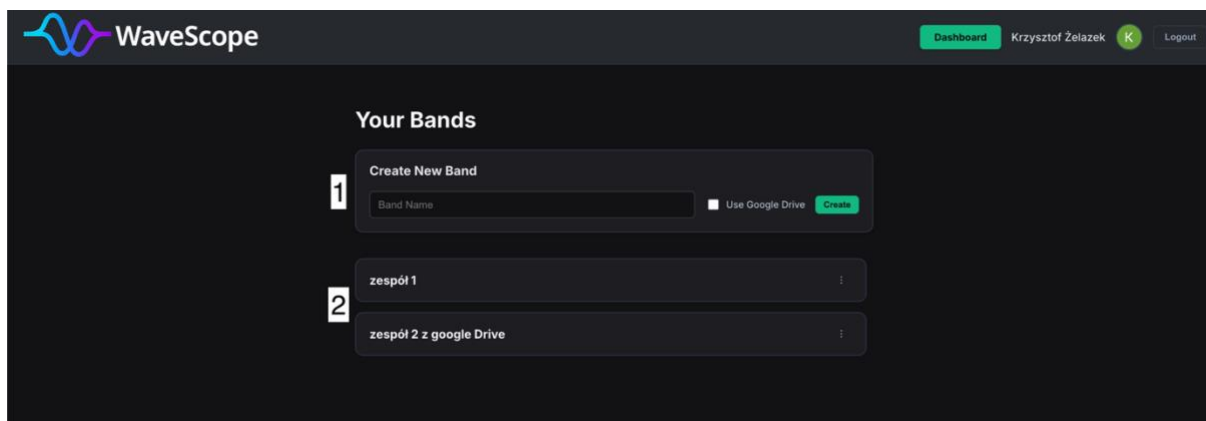
Nadrzędnym założeniem projektowym podczas budowy interfejsu było zapewnienie wysokiej ergonomii pracy. W branży produkcji audio uwaga użytkownika powinna w pełni skupiać się na odsłuchiwanym materiale dźwiękowym, dlatego nawigacja oraz proces nanoszenia komentarzy musiały zostać zaprojektowane w sposób maksymalnie intuicyjny, niewymagający korzystania z dodatkowych instrukcji obsługi. Ponadto, dzięki zastosowaniu opisanej wcześniej koncepcji adaptacyjnej, zaprezentowany interfejs cechuje się wysoką responsywnością, płynnie dostosowując swój układ do urządzeń desktopowych oraz mobilnych, co zostanie zaprezentowane w poniższych rozdziałach.

6.1. Interfejs desktopowy

Pierwszym ważnym widokiem dla użytkownika w systemie WaveScope jest Dashboard. Służy on do tworzenia edytowania i usuwania zespołów z systemu. Widok ten prezentuje Rysunek 17, posiada on dodatkowo oznaczone kluczowe dla użytkownika elementy tego interfejsu:

- 1 – Panel tworzenia zespołu z możliwością wyboru miejsca przechowywania plików audio.
- 2 – Lista zespołów zalogowanego użytkownika z otwieranym panelem modyfikacji i usuwania.

Interfejs tego widoku jest maksymalnie prosty i przejrzysty, co zapewnia intuicyjność użytkowania.

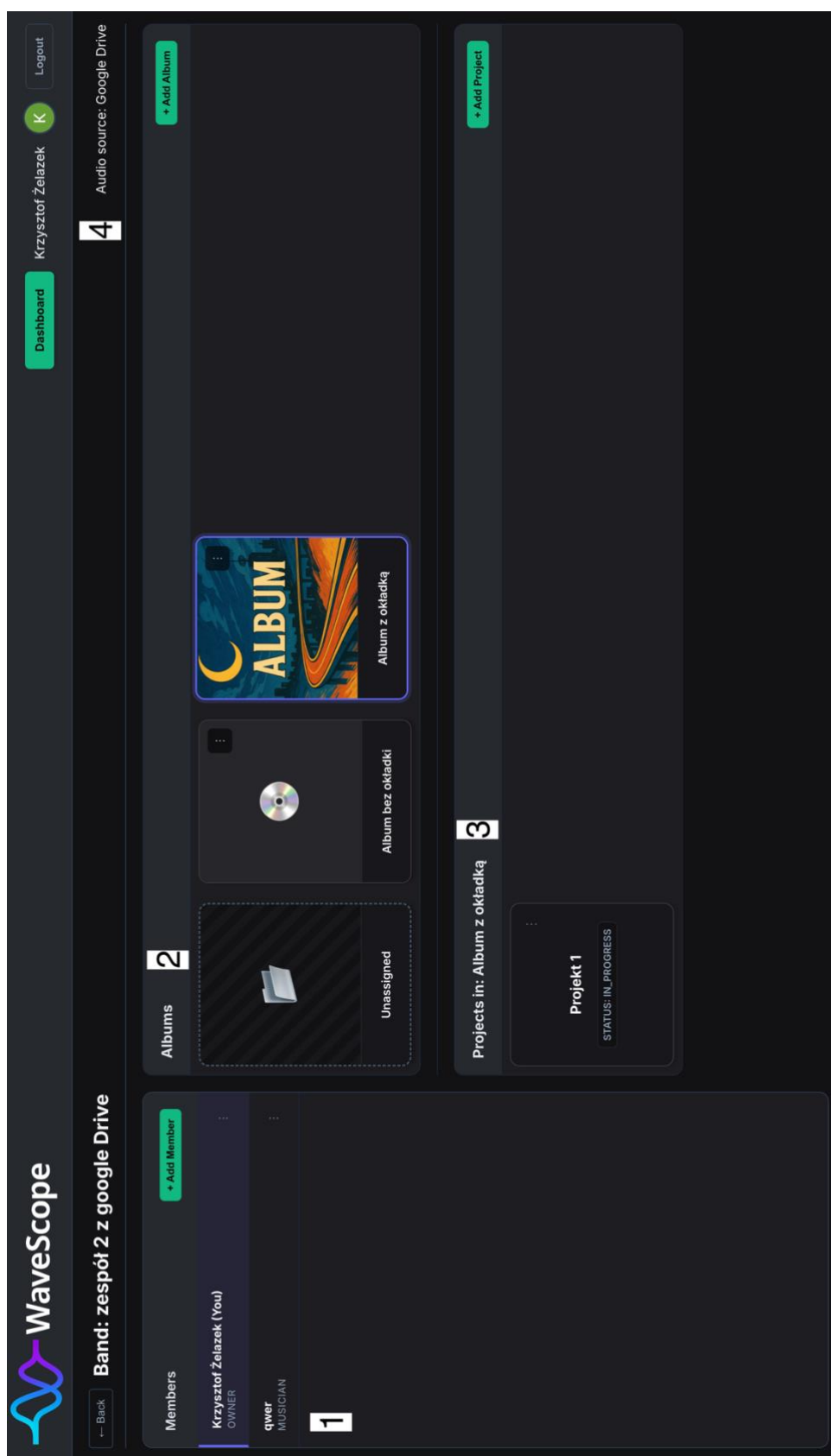


Rysunek 17 - Dashboard systemu WaveScope. Źródło: Opracowanie własne

Centrum zarządzania Systemem WaveScope jest widok konkretnego zespołu. Dostęp do niego następuje po kliknięciu w konkretny zespół w widoku Dashboard. Interfejs pojedynczego zespołu przedstawia Rysunek 18. Wyszczególnić w tym interfejsie należy:

- 1 – Panel członków zespołu, wyświetlający ich nazwy pobrane z konta Google oraz rolę w zespole. Uprawnione osoby mają możliwość dodawania nowych członków, usuwania aktywnych oraz zmiany roli.
- 2 – Panel albumów z poziomu, którego można dodać nowe albumy lub wybierać spośród już stworzonych. Dodatkowo skrajnie lewym albumem zawsze będzie Unassigned do którego będą trafiały projekty nieprzypisane do żadnego albumu.
- 3 – Panel projektów przypisanych do wybranego albumu w tym albumie nieprzypisane.

- 4 – Nagłówek widoku zespołu, w którym po lewej stronie znajduje się nazwa zespołu, a po prawej źródło plików na którym operuje.



Rysunek 18 – Widok zespołu w systemie WaveScope. Źródło: Opracowanie własne

Sercem całego systemu jest widok konkretnego projektu (Rysunek 19). Na interfejsie odtwarzacza można wyróżnić następujące elementy:

- 1 – Panel komentarzy na którym widoczny jest autor komentarza, data i godzina utworzenia, przedział wpisu i treść komentarza. Do dyspozycji są przycisk do usuwania komentarza, przycisk odtwarzania zaznaczonego fragmentu (Segment) oraz przycisk do odtwarzania od początku komentarza do końca utworu (Start).
- 2 – Przedział wybranego komentarza. Każdy przedział na fali audio pełni też rolę przycisku. W celu identyfikacji wybranego przedziału z przypisaną treścią podświetlany jest zarówno przedział jak i komentarz na liście. Użytkownik ma też możliwość kliknięcia komentarza na liście i w takim przypadku również zostaną podświetlone.
- 3 – Przedział komentarza. Wszystkie przedziały komentarzy są widoczne na fali odtwarzacza.
- 4 – Panel nawigacji. Daje użytkownikowi możliwości cofnięcia do początku, odtwarzania/wstrzymywania, podglądu czasu odtwarzania, zmiany szybkości (*ratio*), przybliżenia fali, zmiany głośności. Są to klasyczne funkcje nawigacyjne dla odtwarzaczy dźwiękowych.
- 5 – Panel zarządzania projektem, gdzie można wybrać i pobrać aktualnie odtwarzaną wersję, zatwierdzić oraz wgrać nową wersję.
- 6 – Nagłówek widoku odtwarzania, na którym jest tytuł i opis projektu oraz jego status.

Głównym wyzwaniem z punktu implementacji interfejsu odtwarzacza było stworzenie intuicyjnego sposobu zaznaczania fragmentu i dodawania komentarzy. W celu odróżnienia zwykłego kliknięcia na fali dźwiękowej, która powoduje przesunięcie odtwarzania do momentu kliknięcia, od zaznaczania, został sformułowany skrót (*cmd/ctrl + click*) wywołujący funkcję zaznaczania. Drugą akcją, którą użytkownik może wykonać za pomocą skrótu jest przybliżanie fali odtwarzacza (*option/alt + click*). Informacje o tych funkcjach zostały pokazane na środku całego interfejsu (nad panelem nawigacji i pod falą audio). Dzięki takiemu rozwiązaniu użytkownik momentalnie dowiaduje się o istnieniu tych funkcji.

Dashboard

Krzysztof Żelazek

Logout

← Back

COMPLETED

Projekt 1

Utwór inspirowany muzyką Pata Methenego

6

1

Comments

K

Krzysztof Żelazek

0:08 – 0:11

intro

► Segment

ID Start

K

Krzysztof Żelazek

0:18 – 0:26

temat

► Segment

ID Start

K

Krzysztof Żelazek

0:36 – 0:47

bridge

► Segment

ID Start

K

Krzysztof Żelazek

0:59 – 1:05

temat 2

► Segment

ID Start

4

0:00:00

SPEED

X1.0

⌕ ZOOM

VOLUME

80%

4

Upload New Version

Approved by You

APPROVED BY:

K Krzysztof Żelazek

Download Audio

5

Playing: v2

Version 2

PLAYING

2026-05-28T10:44:49.538584

metheny12_lgotowy...

KRZYSZTOF ŻELAZEK

Version 1

2026-05-29T10:41:45.701615

metheny11.mp3

KRZYSZTOF ŻELAZEK

VERSION DESCRIPTION

Przygotowane części do aranżacji

HOLD CMD/CTRL AND CLICK OR DRAG ON THE WAVEFORM TO ADD A COMMENT.

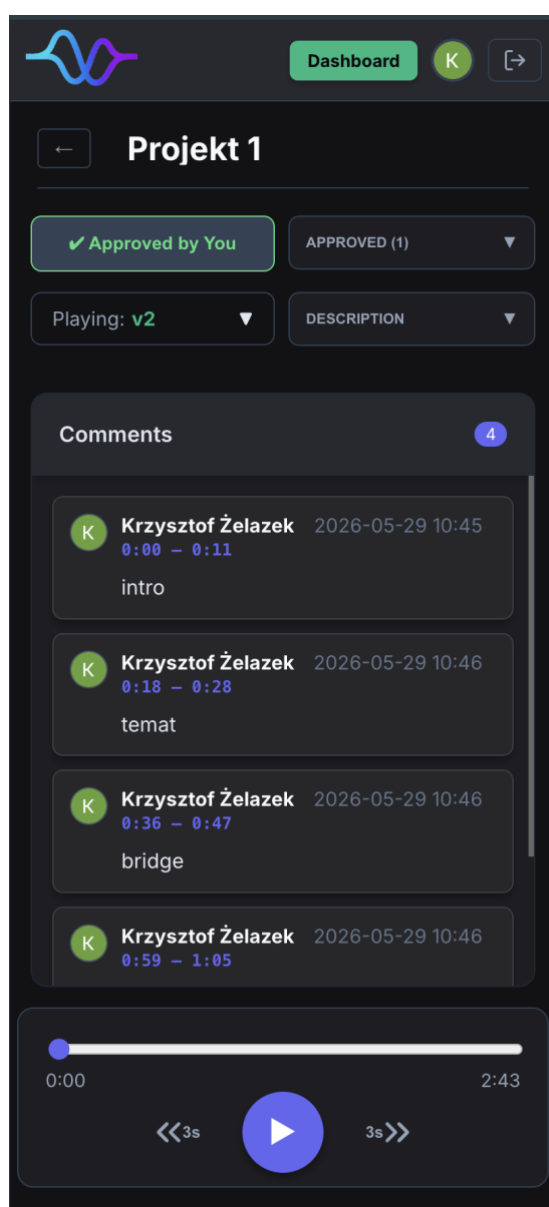
HOLD OPTION/ALT AND SCROLL ON WAVEFORM TO ZOOM IN/OUT.

Rysunek 19 – Odtwarzacz systemu WaveScope. Źródło: Opracowanie własne.

68

6.2. Interfejs mobilny

Interfejs mobilny w większości bazuje na tych samych komponentach co widok desktopowy. Największą różnicą jest interfejs odtwarzacza, który został przystosowany do specyficznych cech urządzeń mobilnych. W porównaniu do wersji desktopowej użyto domyślnego odtwarzacza, zamiast wizualizacji audio. Decyzja ta wynikała z tego, że na wąskim ekranie smartfona zatracone zostałyby wszystkie zalety tego rozwiązania jak precyzyjna nawigacja bazująca na wizualnym odniesieniu do dźwięku. Został też ograniczony dostęp do funkcji dodawania komentarzy ściśle powiązany z wykorzystaniem wavesurfer.js [11] w odtwarzaczu w wersji desktopowej. Drugą różnicą w porównaniu do wersji desktopowej jest ograniczenie nawigacji odtwarzania tylko do funkcji odtwarzania/wstrzymania oraz szybkiego przesuwania odtwarzania o 3 sekundy do przodu i do tyłu. Panel komentarzy tak samo jak w wersji desktopowej jest interaktywny, czyli kliknięcie w pojedynczy wpis przenosi odtwarzanie w miejsce wybranego komentarza. Interfejs Mobilny odtwarzacza prezentuje Rysunek 20.



Rysunek 20 – Mobilna wersja odtwarzacza WaveScope. Źródło: Opracowanie własne.

7. Testowanie aplikacji

Nieodłącznym elementem inżynierii oprogramowania oraz cyklu życia każdego projektu informatycznego jest faza weryfikacji i testowania. Po zaprojektowaniu architektury oraz zaimplementowaniu warstwy serwerowej i klienckiej systemu WaveScope, konieczne było potwierdzenie, że gotowy produkt spełnia wszystkie założone wymagania i działa poprawnie. Głównym celem tego etapu była identyfikacja ewentualnych defektów oraz walidacja stabilności rozwiązania przed jego potencjalnym wdrożeniem w rzeczywistym środowisku produkcyjnym.

Z uwagi na złożony charakter aplikacji, łączącej skomplikowaną logikę biznesową (np. wersjonowanie plików) z wysoce interaktywnym interfejsem graficznym, proces weryfikacji podzielono na dwa niezależne, uzupełniające się filary. Pierwszym z nich jest weryfikacja niskopoziomowa, skupiająca się na poprawności działania samego kodu źródłowego, drugim zaś – ewaluacja wysokopoziomowa, badająca użyteczność systemu z perspektywy jego docelowych odbiorców.

W niniejszym rozdziale szczegółowo omówiono przeprowadzone procedury testowe. W pierwszej kolejności zaprezentowano implementację testów jednostkowych dla przykładowego serwisu. Następnie przedstawiono metodykę i wyniki testów z udziałem użytkowników, pozwalające na ocenę ergonomii interfejsu (*UX*) oraz płynności obsługi procesów kolaboracji przy pracy nad materiałami audio.

7.1. Testy jednostkowe

Testy jednostkowe (*unit tests*) stanowią najniższy, a zarazem najszerszy poziom piramidy testów oprogramowania [56]. Ich głównym zadaniem jest weryfikacja poprawności działania wyizolowanych fragmentów kodu – najczęściej pojedynczych metod lub klas – całkowicie niezależnie od pozostałych modułów systemu. W architekturze WaveScope zrezygnowano z testowania prostych komponentów typu DTO czy encji, a główny nacisk położono na przetestowanie klas z warstwy usługowej (*service*), w których zaimplementowana została krytyczna logika biznesowa oraz weryfikacja uprawnień użytkowników.

Do implementacji testów wykorzystano framework JUnit 5 [33], będący standardem w ekosystemie języka Java. Aby zapewnić pełną hermetyzację testowanych serwisów i uniezależnić je od warstwy trwałości (bazy danych PostgreSQL) oraz zewnętrznych systemów, zintegrowano go z biblioteką Mockito [83]. Narzędzie to pozwala na tworzenie tzw. atrap obiektów (*mocks*), które symulują zachowanie rzeczywistych zależności.

Inżynierskie podejście do weryfikacji logiki biznesowej doskonale obrazuje zestaw testów dla serwisu `CommentService` (Listing 35). Środowisko testowe jest inicjalizowane przed uruchomieniem każdego przypadku testowego (adnotacja `@BeforeEach`), co gwarantuje pełen determinizm i powtarzalność wyników. Testowana klasa oznaczona jest adnotacją `@InjectMocks`, natomiast wszystkie jej zależności, takie jak `CommentRepository` czy `UserService`, wstrzykiwane są jako kontrolowane atrapy (`@Mock`).

Listing 35 – Struktura testu jednostkowego. Źródło: Opracowanie własne.

```
@Mock
private CommentRepository commentRepository;

@Mock
private BandMembersService bandMembersService;

@InjectMocks
private CommentService commentService;

@Test
```

```

void createComment_shouldSaveAndReturnDTO() {
    when(versionService.getVersionById(100L)).thenReturn(version);

    when(userService.getUserByGoogleId("google123")).thenReturn(Optional.of(user));
    when(bandMembersService.existsByBandIdAndUserId(10L,
1L)).thenReturn(true);
    when(commentRepository.save(any(Comments.class))).thenReturn(comment);

    CommentGetDTO result = commentService.createComment("Test text", 0L,
10L, 100L, user);

    assertNotNull(result);
    assertEquals("Test text", result.text());
    verify(commentRepository).save(any(Comments.class));
}

```

Zgodnie z dobrymi praktykami inżynierii oprogramowania, zaprezentowany test został zorganizowany według klasycznego wzorca AAA (*Arrange-Act-Assert*):

- **Arrange (Przygotowanie):** Za pomocą metody `when().thenReturn()` zdefiniowano przewidywane zachowanie atrap. Symulują one między innymi pozytywną weryfikację przynależności użytkownika do zespołu bez konieczności wykonywania rzeczywistych zapytań SQL.
- **Act (Działanie):** Wywoływana jest docelowa metoda dodawania komentarza z precyzyjnie przygotowanym zestawem danych wejściowych.
- **Assert (Weryfikacja):** Za pomocą asercji `assertEquals` sprawdzana jest struktura zwróconego obiektu. Ponadto, użycie funkcji `verify()` pozwala potwierdzić, że w trakcie wykonywania metody serwis faktycznie wywołał metodę `save()` na repozytorium, co gwarantuje próbę utrwalenia danych.

Oprócz weryfikacji tzw. ścieżek pozytywnych (*happy paths*), testy jednostkowe w systemie WaveScope pokrywają również ścieżki negatywne i obsługę przypadków brzegowych. Przykładowo, zaimplementowano testy weryfikujące, czy przy próbie usunięcia komentarza przez osobę niebędącą jego autorem (Listing 36), metoda `deleteComment` poprawnie rzuca wyjątek `IllegalArgumentException` (weryfikowane asercją `assertThrows`). Dzięki takiemu podejściu aplikacja cechuje się wysoką stabilnością oraz odpornością na nieautoryzowane manipulacje danymi ze strony użytkowników.

Listing 36 – struktura testu z negatywnym scenariuszem. Źródło: Opracowanie własne.

```

@Test
void deleteComment_shouldThrowWhenNotAuthor() {
    User otherUser = new User();
    otherUser.setId(2L);
    otherUser.setGoogleId("other123");

    comment.setAuthor(user);

    when(userService.getUserByGoogleId("other123")).thenReturn(Optional.of(otherUser));
    when(commentRepository.findById(50L)).thenReturn(Optional.of(comment));

    assertThrows(IllegalArgumentException.class, () ->
commentService.deleteComment(50L, otherUser));
}

```

7.2. Testy interfejsu użytkownika

Ostatnim etapem prac nad prototypem systemu WaveScope było przeprowadzenie testów na użytkownikach końcowych. W tym celu aplikacja została przygotowana do zahostowania i udostępniona wybranym osobom. Dodatkowo użytkownicy dostali plik tekstowy zawierający wymagania systemowe, link do aplikacji, listę pięciu zadań oraz link do ankiety.

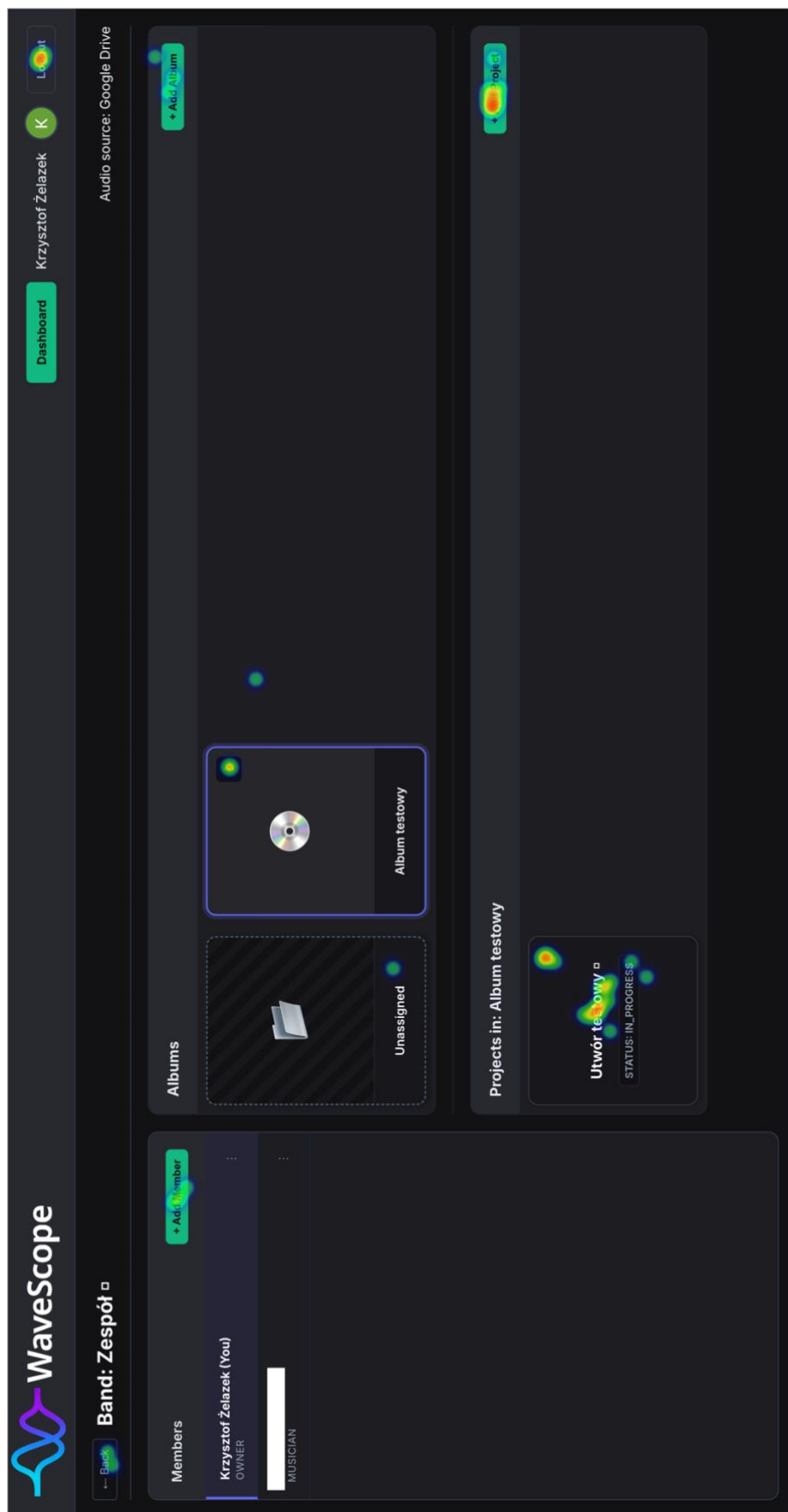
Jako technikę badawczą zdecydowano się na zastosowanie modelu hybrydowego łączącego testy moderowane i ankietę SUS [84]. Pierwsza część badania polegała na zbudowaniu mapy cieplnej (*heatmap*). Jej celem było sprawdzenie, z którymi elementami systemu WaveScope użytkownicy mają największe problemy, a jakie funkcje są zbudowane w sposób intuicyjny. W tym celu do wersji testowej zostało dołączone oprogramowanie Microsoft Clarity [57]. Rejestruje ono zachowanie użytkowników i na tej podstawie tworzy wspomniane wyżej mapy.

Do przeprowadzenia tych testów wybrano cztery osoby, stanowiące grupę docelową systemu WaveScope. Liczba osób zaangażowanych w testy bazuje na badaniu Jakoba Nielsena [85] *„Skomplikowane testy użyteczności to marnotrawstwo zasobów. Najlepsze rezultaty uzyskuje się, testując nie więcej niż 5 użytkowników i przeprowadzając tyle małych testów, na ile Cię stać.”*

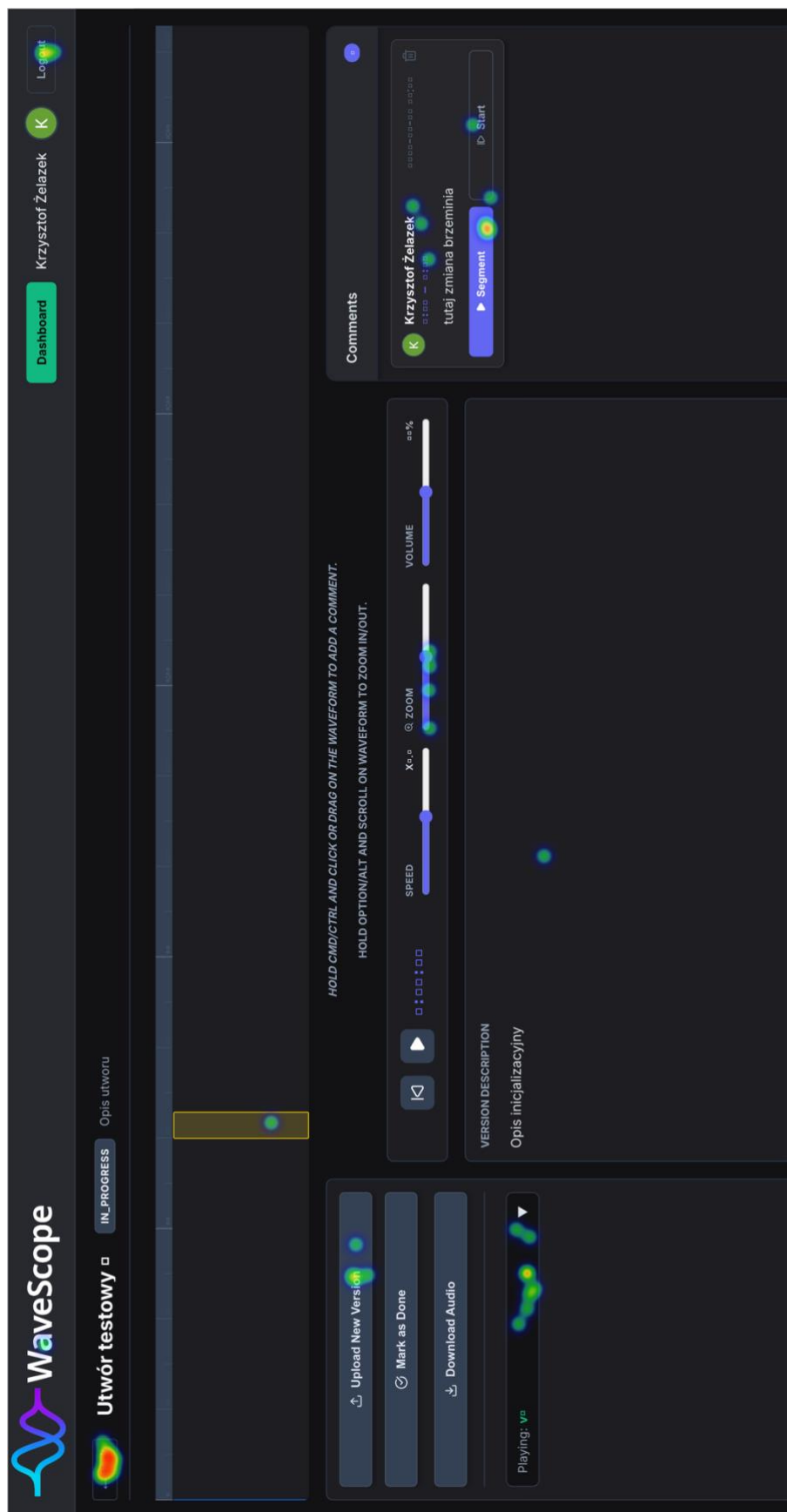
Wybrani użytkownicy dostali za zadanie przejście przez 6 scenariuszy typowych zachowań w systemie WaveScope. Listę tą stanowiły następujące zadania:

- Zaloguj się do systemu WaveScope a następnie utwórz zespół nadając mu dowolną nazwę i ustaw ten zespół tak, aby korzystał Google Drive.
- Do utworzonego zespołu dodaj użytkownika o przesłanym w wiadomości prywatnej adresie email i nadaj mu uprawnienia producenta.
- Utwórz projekt z plikiem „plik1.mp3” dołączonym do listy zadań, nadaj mu nazwę „Pierwszy projekt w wavescope”, dodaj do niego dowolny opis i dowolny opis wersji inicjalizacyjnej.
- Otwórz projekt „Pierwszy projekt w wavescope”, sprawdź możliwości odtwarzania dźwięku, dodaj dwa komentarze, spróbuj odtworzyć jeden z komentarzy.
- Do projektu „Pierwszy projekt w wavescope” dodaj nową wersję zawierającą „plik2.mp3”.
- Utwórz nowy album nadając mu dowolną nazwę, a następnie dodaj do niego projekt „Pierwszy projekt w wavescope”

Po dokonaniu testów przez użytkowników przeanalizowano ich rezultaty. Trzy z Czterech osób w pełni wykonały wszystkie zdania. Po analizie nagrań zachowań użytkowników w aplikacji WaveScope utworzonych przez Microsoft Clarity można wnioskować, że zbudowany interfejs jest intuicyjny i łatwy w obsłudze. Potwierdzają to mapy cieplne dla widoków zespołu (Rysunek 21) oraz projektu (Rysunek 22). Drobnym minusem oprogramowania Clarity jest cenzurowanie niektórych treści, gdy uzna je za poufne. W przypadku testów widoku odtwarzacza ukrył on wizualizację fali dźwiękowej, ale zostawił przedział komentarza, co pozwala zidentyfikować czy użytkownik we właściwy sposób zaznaczy utworzony komentarz. W celu ukrycia tożsamości osób biorących udział w testach mapa została zbudowana na widoku autora systemu WaveScope.



Rysunek 21 – mapa kliknięć w widoku zespołu. Źródło: Opracowanie własne.



Rysunek 22 – Mapa kliknięć w widoku odtwarzacza. Źródło: Opracowanie własne.

Niezwyczajnie wartościowych danych analitycznych dostarczyła sesja z jednym z użytkowników, który jako jedyny napotkał krytyczny błąd blokujący ukończenie zaplanowanego scenariusza testowego. Podczas próby inicjalizacji nowego projektu system odmówił współpracy, zwracając komunikat o błędzie dodawania pliku audio. W pierwszej kolejności zweryfikowano spełnienie podstawowych wymagań środowiskowych, potwierdzając, że test był przeprowadzany w zalecanej przeglądarce Google Chrome w standardowym, nieprywatnym trybie.

Szczegółowa analiza kodu odpowiedzialnego za asynchroniczne przesyłanie formularzy wieloczęściowych z warstwy klienckiej do serwera wykazała, że źródłem problemu była specyfika używanego systemu operacyjnego. Użytkownik ten, jako jedyny w grupie badawczej, korzystał ze środowiska Microsoft Windows, podczas gdy pozostałe testy oraz cały proces deweloperski odbywały się w systemie macOS. Bezpośrednią przyczyną anomalii okazał się brak jawnego określenia rozszerzenia pliku dla pakowanego w locie obiektu `jsonBlob`. Architektura sieciowa w systemie macOS poprawnie interpretowała zawartość na podstawie zadeklarowanych w kodzie nagłówków typu. Z kolei system Windows traktował ten obiekt jako plik o niezidentyfikowanym formacie, przez co przeglądarka przed wysłaniem żądania automatycznie nadpisywała jego typ MIME [86] domyślną wartością `application/octet-stream` (surowy strumień bajtów). W rezultacie mechanizmy bezpieczeństwa warstwy serwerowej WaveScope nie potrafiły zdeserializować metadanych projektu i kategorycznie odrzucały takie żądanie HTTP.

Aby skutecznie wyeliminować opisaną anomalię, wdrożono prostą, lecz kluczową poprawkę w kodzie warstwy klienckiej. Polegała ona na jawnej deklaracji nazwy przesyłanego pliku podczas dołączania go do obiektu `FormData`. Poprzez przypisanie strumieniowi danych (`Blob`) stałej nazwy `project.json` – zawierającej jednoznaczne rozszerzenie formatu – wymuszono na przeglądarce, niezależnie od używanego systemu operacyjnego, poprawne rozpoznanie zawartości i nadanie właściwego nagłówka MIME jako `application/json`. Ten drobny zabieg architektoniczny przywrócił pełną kompatybilność wieloplatformową systemu i trwale rozwiązał problem z odrzucaniem żądań przez serwer. Analogiczna zmiana została wprowadzona w komponencie odpowiedzialnym za wgrywanie nowej wersji dzięki czemu problem ten nie powinien już wystąpić.

Po zrobieniu zadań użytkownicy byli poproszeni o wypełnienie 10 punktowej ankiety SUS. Z racji tego, że metoda ta jest ilościowa, a próbka do badania była niska nie może ona stanowić głównego źródła oceny systemu. Po zsumowaniu wyników pojedynczych ankietowanych i wyliczeniu średniej osiągnięto wynik 83,75. Bazując na artykule „*Journal of Usability Studies*” [87] można wywnioskować, że osiągnięto doskonałe rezultaty.

Łącząc obie techniki badawcze potwierdzono, że zbudowany interfejs jest intuicyjny i przyjazny użytkownikom. Dodatkowo w trakcie testów uzyskano wskazówki do dalszego rozwoju systemu WaveScope.

8. Podsumowanie

Niniejszy rozdział stanowi kompleksowe podsumowanie całego procesu inżynierskiego, którego celem było zaprojektowanie, zaimplementowanie oraz przetestowanie systemu WaveScope. Realizacja zaawansowanej aplikacji typu *Single Page Application* [36], integrującej bezstanową architekturę backendową z dynamicznym przetwarzaniem multimediów w warstwie klienckiej, pozwoliła na weryfikację początkowych założeń teoretycznych w rzeczywistym środowisku programistycznym.

Jak każdy złożony projekt informatyczny, cykl wytwórczy systemu nie był pozbawiony wyzwań. Zderzenie pierwotnych koncepcji architektonicznych z fizycznymi ograniczeniami technologicznymi, asynchroniczną komunikacją sieciową czy specyfiką przeglądarek internetowych dostarczyło niezwykle cennych doświadczeń. Zostały one szczegółowo zidentyfikowane i opisane w części poświęconej napotkanym trudnościom implementacyjnym. Refleksja nad sposobami rozwiązania tych problemów stała się z kolei podstawą do sformułowania kluczowych wniosków płynących z całego procesu powstawania prototypu aplikacji, co pozwala na krytyczną ocenę podjętych decyzji projektowych.

Zwieńczeniem pracy jest ewaluacja potencjału rozwoju systemu. Zgodnie z dobrymi praktykami inżynierii oprogramowania, zaprezentowany prototyp traktowany jest jako solidny i skalowalny fundament, na bazie którego nakreślono ścieżki dalszej ewolucji platformy w kierunku pełnoprawnego, komercyjnego narzędzia wspierającego produkcję audio.

8.1. Napotkane trudności

Proces wytwarzania prototypu oprogramowania WaveScope postawił przed autorem szereg trudności, z którymi musiał się zmierzyć, celem dostarczenia jak najlepszych wrażeń użytkownika przy zachowaniu realizacji wymagań funkcjonalnych i niefunkcjonalnych.

Jednym z pierwszych wyzwań architektonicznych w projekcie był wybór oraz implementacja odpowiedniego mechanizmu uwierzytelniania. Integracja zewnętrznego dostawcy tożsamości (Google OAuth2) z bezstanową architekturą serwera (JWT) wymagała nieszablonowej konfiguracji łańcucha filtrów bezpieczeństwa w frameworku Spring. Choć na rynku istnieje wiele alternatywnych podejść do zarządzania sesją (np. klasyczne sesje serwerowe lub przechowywanie tokenów po stronie klienta w `localStorage`), zdecydowano się na wdrożenie rygorystycznego modelu opartego na ciasteczkach z flagą `HttpOnly`.

Z perspektywy czasu oraz przeprowadzonych testów można stwierdzić, że przyjęte rozwiązanie, choć optymalne pod kątem ochrony przed atakami typu XSS (*Cross-Site Scripting*), posiada istotne ograniczenia natury użytkowej. Wymuszenie opierania autoryzacji na ciasteczkach sprawia, że system WaveScope może napotykać problemy podczas działania w trybie prywatnym (tzw. tryb Incognito) nowoczesnych przeglądarek internetowych. Przeglądarki te, w ramach rygorystycznych polityk prywatności, często domyślnie blokują pliki cookie (zwłaszcza w środowisku cross-origin, gdy frontend i backend działają na różnych portach), co uniemożliwia poprawne przekazanie i odczytanie tokenu JWT, skutkując błędami autoryzacji. Jest to klasyczny przykład kompromisu inżynierskiego, w którym wyższy standard bezpieczeństwa kryptograficznego został osiągnięty kosztem obniżonej kompatybilności w restrykcyjnych środowiskach przeglądania.

Kolejnym wyzwaniem podczas prac nad warstwą serwerową była optymalizacja zapytań do bazy danych. W pierwszej iteracji projektu autor całkowicie polegał na domyślnych mechanizmach generowania zapytań przez system ORM Hibernate. Podczas ewaluacji powstałego systemu zauważono jednak, że niektóre operacje w sposób niewydajny pobierają dane, co objawiało się klasycznym problemem N+1 zapytań. Próba jego wyeliminowania poprzez tworzenie ręcznych zapytań tekstowych JPQL z klauzulą `JOIN FETCH` połączoną z instrukcją warunkową `WHERE` okazała się niezgodna z dobrymi praktykami specyfikacji JPA. Filtrowanie po dociąganych zachłannie relacjach prowadziło bowiem do naruszenia spójności modelu pamięciowego (powstawania tzw. *partial objects*)

Po konsultacji merytorycznej oraz głębszej analizie problematyki dostępu do danych, podejście to zostało całkowicie przeprojektowane. Zrezygnowano z podatnego na błędy, ręcznego pisanie zapytań

tekstowych (w tym używania adnotacji `@Query`) na rzecz mechanizmu metod pochodnych (*Derived Query Methods*). Zapewniło to pełne bezpieczeństwo typów na etapie kompilacji podczas precyzyjnego filtrowania danych. Aby jednocześnie zlikwidować problem N+1 zapytań, mechanizm ten zintegrowano z deklaratywną adnotacją `@EntityGraph`, która wymusza bezpieczne, zachłanne dociąganie powiązanych encji w ramach jednego, zoptymalizowanego złączenia SQL na poziomie silnika bazy danych. Tym samym udowodniono w praktyce, że inteligentne nazewnictwo metod filtrujących oraz świadome zarządzanie grafami encji doskonale się uzupełniają, stanowiąc nowoczesny, bezpieczny i wydajny fundament architektury opartej na Spring Data JPA.

Implementacja warstwy klienckiej również niosła ze sobą trudności implementacyjne. Największą z nich było dostosowanie biblioteki `wavesurfer.js` [11] do realizacji wymagań funkcjonalnych. Głównym problemem, jaki został zauważony podczas prac nad wdrożeniem funkcjonalności komentarzy, był brak synchronizacji zaznaczania podczas gdy fala dźwięku jest przybliżona. Problem ten został rozwiązany przez dodanie autorskiego systemu zaznaczania, który w prawidłowy sposób przelicza przedział czasu który użytkownik chce zaznaczyć. Oryginalnie rozszerzenie `Regions` [50] nie działało poprawnie w momencie, gdy fala dźwięku była przybliżona i nie skalowało zaznaczenia w odpowiedni sposób.

Wyzwaniem podczas testów z użytkownikami było samo udostępnienie aplikacji. Przez cały proces pracy nad kodem systemu `WaveScope` był on skonfigurowany do pracy na lokalnym serwerze `localhost`. W celu udostępnienia jej konieczne było przerobienie prototypu w sposób wspierający zmienne środowiskowe zmieniając dotąd na sztywno zapisane w kodzie adresy. Dodatkowo darmowy hosting użyty do udostępnienia prototypu po 15 minutach beczynności wchodził w stan uśpienia. Do rozwiązania tego problemu oraz zapewnienia użytkownikom dobrego odbioru aplikacji napisano punkt końcowy w API aplikacji służący do wywoływania aktywności aplikacji i bazy danych co zapobiega ich usypianiu.

Istotną trudnością, tym razem o charakterze organizacyjnym i procesowym, było początkowe zaniechanie wykorzystania systemu kontroli wersji (`Git`) we wczesnych fazach projektu. Z uwagi na jednoosobowy charakter prac programistycznych oraz ograniczenie środowiska deweloperskiego do jednej stacji roboczej, błędnie założono, że utrzymywanie formalnego repozytorium nie jest na tym etapie niezbędne. Brak ścisłego śledzenia zmian w kodzie źródłowym szybko okazał się jednak wąskim gardłem całego cyklu wytwórczego.

Szczególnie problematyczne stały się sytuacje wymagające implementacji eksperymentalnych funkcjonalności lub głębokiej refaktoryzacji na stabilnej wersji systemu. W przypadku niepowodzenia modyfikacji, brak mechanizmów swobodnego przywracania stanu oraz brak izolacji nowego kodu drastycznie utrudniał, a niekiedy uniemożliwiał szybki powrót do ostatniej działającej wersji aplikacji. Zderzenie z tym problemem udowodniło, jak kluczowe są standardy inżynierii oprogramowania niezależnie od skali zespołu. Wymusiło to szybkie wdrożenie repozytorium opartego na systemie `Git`, co natychmiastowo ustabilizowało proces wytwórczy i wyeliminowało ryzyko bezpowrotnej utraty spójności kodu.

8.2. Potencjał rozwoju aplikacji

Proces inżynierii oprogramowania ma z założenia charakter iteracyjny. Zrealizowany system `WaveScope`, choć w pełni odpowiada na zdefiniowane na początku pracy wymagania funkcjonalne i niefunkcjonalne, stanowi w swojej obecnej formie wysoce skalowalny prototyp. W trakcie całego cyklu wytwórczego – zarówno na etapie projektowania architektury przez autora, jak i podczas ewaluacji systemu w ramach testów z docelowymi użytkownikami – zidentyfikowano szereg obszarów o wysokim potencjale rozwoju. Planowane modyfikacje i rozszerzenia obejmują nie tylko optymalizację obecnych procesów, ale również implementację zupełnie nowych modułów, które mogłyby znacząco podnieść wartość biznesową aplikacji i jej konkurencyjność na rynku narzędzi wspierających produkcję audio.

Pierwszą znaczącą funkcjonalnością, która może zostać dodana do systemu jest pełen streaming utworów i wsparcie dla plików o większych rozmiarach. Wprowadzenie tej funkcjonalności

poprawiłoby wrażenia z użytkowania systemu oraz poprawiło komfortową pracę z plikami w formacie .wav. Implementacja tej funkcjonalności byłaby wymagająca ze względu na zastosowanie biblioteki wavesurfer.js [11]. Wymaga ona bowiem przeskanowania całego pliku celem zbudowania wizualizacji fali dźwiękowej. Jednym z możliwych rozwiązań byłoby przetrzymywanie odpowiedniego pliku tekstowego, zawierającego liczbową reprezentację fali dźwiękowej, a następnie przesłaniu tego pliku w całości przed rozpoczęciem pobierania audio.

Drugą dużą, a zarazem interesującą funkcjonalnością mogłoby być wprowadzenie zaawansowanego analizatora parametrów dźwięku. Mógłby on wykorzystywać przestrzeń pod panelem nawigacji odtwarzania. Informowałby on użytkownika jakie parametry głośności w czasie rzeczywistym osiąga odsłuchiwany plik (dB) oraz jakie na całym zakresie (LUFS). Funkcjonalność ta mogłaby być dodatkowo rozszerzona o analizator widma pasma częstotliwościowego. Wprowadzenie takiego rozszerzenia wymagałoby użycia lub stworzenia specjalnej biblioteki, a w połączeniu z wspomnianym wprowadzeniem streamingu, dane operujące na całym zakresie musiałyby być wyliczane przez serwer i przetrzymywane w bazie, co wymagałoby dodatkowych nakładów pracy nad wprowadzeniem tych zmian.

Trzecią możliwością rozwoju mogłoby być dodanie innych dostawców przestrzeni chmurowych. Użytkownik mógłby wybierać spośród przetrzymywania plików na serwerze WaveScope, na własnym Google Drive [1] oraz na innych wprowadzonych dyskach. Implementacja wymagałaby szerszych zmian w strukturze serwerowej, ponieważ aktualnie współdziałanie z Google Drive jest ściśle powiązane z autoryzacją w systemie WaveScope wymagającym zalogowania się przez konto Google. Dodatkowo należałoby przerobić warstwę kliencką, żeby w elegancki sposób dawała możliwość wyboru tych opcji.

Wartą rozważenia dodatkową funkcjonalnością jest dodanie dynamicznego porównywania wersji. Mogłoby to polegać na wyborze dwóch wersji oraz dodaniu przycisku A|B, który byłby odpowiedzialny za zmianę wersji podczas odtwarzania. Szczególnie przydatne byłoby to przy porównywaniu brzmienia dwóch różnych wersji. Implementacja tej funkcjonalności niosłaby za sobą trudność wynikającą z synchronizacji odtwarzania dwóch plików jednocześnie.

Jedną z propozycji funkcji jakie można dodać do systemu WaveScope otrzymaną od użytkowników, na których przeprowadzane były testy, jest dodanie metronomu. Mogłoby to rozszerzyć zastosowania systemu o ćwiczenie i przygotowywanie się do koncertów. W takim wypadku różne wersje mogłyby być przygotowane dla konkretnych osób jako ich pliki ćwiczeniowe, a metronom miałby być odpalany bezpośrednio w aplikacji. Mógłby on posiadać kontrolę głośności oraz zmianę brzmienia. Implementacja tej funkcji wymagałaby jedynie synchronizacji pliku dźwiękowego z zegarem aplikacji oraz dodaniu modułu metronomu, który można zrealizować tylko w warstwie klienckiej. Implementacja tej funkcji charakteryzuje się bardzo korzystnym stosunkiem nakładu pracy deweloperskiej do potencjalnych zysków biznesowych. Jej wdrożenie mogłoby znacząco poszerzyć rynek docelowy i zainteresowanie systemem, otwierając aplikację na zupełnie nowe scenariusze użycia.

Kolejnym, uzyskanym od użytkowników kierunkiem rozwoju interfejsu jest implementacja mechanizmu przeciągnij i upuść (*Drag and Drop*), umożliwiającego intuicyjne przypisywanie projektów do konkretnych albumów. W obecnej wersji systemu operacja ta wymaga od użytkownika przejścia przez dedykowany formularz modyfikacji, co niepotrzebnie wydłuża ścieżkę interakcji. Z perspektywy inżynierskiej, realizacja tej funkcjonalności ograniczałaby się wyłącznie do rozbudowy warstwy klienckiej, bez konieczności ingerencji w logikę backendową. Wdrożenie takiego rozwiązania zauważalnie usprawniłoby zarządzanie strukturą plików i znacząco podniosło ogólną ergonomię pracy oraz pozytywne wrażenia z użytkowania aplikacji.

8.3. Wnioski z procesu powstawania prototypu aplikacji

Proces projektowania, implementacji oraz ewaluacji systemu WaveScope pozwolił na sformułowanie kilku kluczowych wniosków, zarówno w kontekście użytych technologii, jak i samej metodyki wytwarzania oprogramowania.

Pierwszym i najważniejszym wnioskiem jest potwierdzenie trafności doboru stosu technologicznego. Połączenie frameworka Spring Boot [6] w warstwie serwerowej z biblioteką React [9] w warstwie klienckiej udowodniło swoją wysoką skuteczność w budowie nowoczesnych aplikacji typu *Single Page Application* [36]. Ścisła separacja obu środowisk za pomocą bezstanowego interfejsu REST API [31] oraz zastosowanie obiektów DTO [53] nie tylko uprościły proces programowania, ale również zagwarantowały wysoką elastyczność i skalowalność rozwiązania.

Drugim istotnym wnioskiem jest potwierdzenie, że współczesne przeglądarki internetowe posiadają wystarczającą moc obliczeniową oraz odpowiednie interfejsy programistyczne, aby stanowić pełnoprawne środowisko do pracy z materiałami audio. Zbudowany prototyp udowadnia, że do efektywnej kolaboracji i recenzowania miksów nie jest wymagane posiadanie ciężkich, specjalistycznych programów typu DAW [16] czy używania rozwiązań niekoniecznie przeznaczonych do tego typu zadań. Przeniesienie tego procesu do sieci w postaci aplikacji przeglądarkowej znacząco obniża próg wejścia oraz poprawia doświadczenia użytkownika zapewniając potrzebne do procesu recenzji funkcje.

Trzecia refleksja dotyczy samego podejścia do inżynierii oprogramowania i wartości, jaką niesie wczesne zderzenie prototypu z docelowym odbiorcą. Przeprowadzone testy użyteczności jednoznacznie pokazały, że nawet najlepiej zaprojektowana architektura wymaga walidacji w rzeczywistych scenariuszach użytkowych. Zidentyfikowanie problemów z odpowiednim przesyłaniem plików czy zasugerowanie implementacji metronomu dowodzą, że ciągła pętla sprzężenia zwrotnego (*feedback loop*) pomiędzy twórcą a użytkownikiem jest kluczowa dla budowania wartościowych i ergonomicznych produktów.

Podsumowując, główny cel pracy inżynierskiej – polegający na stworzeniu aplikacji webowej ułatwiającej komunikację oraz wymianę plików między muzykami a realizatorami dźwięku – został w pełni osiągnięty. Zbudowany prototyp systemu WaveScope jest stabilny, bezpieczny i realizuje wszystkie zdefiniowane w fazie projektowej wymagania funkcjonalne. Zebrane w trakcie pracy doświadczenia, zidentyfikowane trudności oraz nakreślone perspektywy rozwoju stanowią solidny fundament do ewentualnej, dalszej komercjalizacji projektu w przyszłości.

9. Bibliografia

- [1]. Google Drive – strona oficjalna <https://drive.google.com/> (stan na: 10.05.2026).
- [2]. Dropbox – strona oficjalna, <https://www.dropbox.com> (stan na: 10.05.2026).
- [3]. WeTransfer – strona oficjalna, <https://wetransfer.com> (stan na: 10.05.2026).
- [4]. SaaS – artykuł firmy Oracle, <https://www.oracle.com/pl/applications/what-is-saas> (stan na: 10.05.2026).
- [5]. Java – strona oficjalna, <https://www.java.com/> (stan na: 10.05.2026).
- [6]. Spring – strona oficjalna, <https://spring.io> (stan na: 10.05.2026).
- [7]. Hibernate – opis funkcji ORM, <https://hibernate.org/orm> (stan na 10.05.2026).
- [8]. PostgreSQL – strona oficjalna, <https://www.postgresql.org> (stan na 10.05.2026).
- [9]. React – strona oficjalna, <https://react.dev> (stan na: 10.05.2026).
- [10]. TypeScript – strona oficjalna, <https://www.typescriptlang.org/> (stan na: 10.05.2026).
- [11]. wavesurfer.js – strona oficjalna, <https://wavesurfer.xyz/> (stan na: 10.05.2026).
- [12]. Messenger – strona oficjalna, <https://www.messenger.com/> (stan na: 10.05.2026).
- [13]. WhatsApp – strona oficjalna, <https://www.whatsapp.com/> (stan na: 10.05.2026).
- [14]. Discord – strona oficjalna, <https://discord.com> (stan na: 10.05.2026).
- [15]. Statista, Most popular global mobile messenger apps as of October 2025, based on number of monthly active users, <https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps> (stan na: 10.05.2026).
- [16]. Avid Technology Inc., Pro Tools – strona oficjalna <https://www.avid.com/pro-tools> (stan na: 10.05.2026).
- [17]. Avid Technology Inc., *Pro Tools Cloud Collaboration Quick Start*, https://kb.avid.com/pkb/articles/how_to/Pro-Tools-Cloud-Collaboration-Quick-Start (stan na: 10.05.2026).
- [18]. Pro Tools – artykuł ze zdjęciem aplikacji <https://www.bonedo.de/artikel/pro-tools-125-hd-test/> (stan na: 24.05.2026).
- [19]. Pro Tools Cloud Collaboration – artykuł opisujący działanie ze zdjęciami. <https://www.soundonsound.com/techniques/part-2-cloud-collaboration-pro-tools-125> (stan na 24.05.2026).
- [20]. Pro Tools Cloud Collaboration – opinie użytkowników na forum Reddit.com https://www.reddit.com/r/AudioPost/comments/15t6o6f/cloud_collaboration_for_pro_tools/ (stan na: 10.05.2026).
- [21]. Opusonix – artykuł <https://bedroomproducersblog.com/2026/01/21/opusonix-review/> (stan na: 10.05.2026)
- [22]. Opusonix – strona oficjalna z pakietami subskrypcji <https://opusonix.com/pricing/> (stan na: 10.04.2026).
- [23]. Opusonix- strona oficjalna <https://opusonix.com/music-collaboration-workspace/> (stan na: 24.05.2026).

- [24]. Dropbox Replay – strona oficjalna <https://www.dropbox.com/replay> (stan na: 10.05.2026).
- [25]. UML – artykuł z opisem <https://www.lucidchart.com/pages/pl/czym-jest-uml-unified-modeling-language> (stan na: 10.05.2026).
- [26]. OAuth2 – dokumentacja <https://oauth.net/2/> (stan na: 10.05.2026).
- [27]. Google chrome – strona oficjalna <https://www.google.com/chrome/> (stan na: 10.05.2026).
- [28]. Bloch J., *Java. Efektywne programowanie. Wydanie III*, rozdział 4, Temat 18.
- [29]. HTML5 Canvas – dokumentacja techniczna https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API (stan na: 13.05.2026).
- [30]. wavesurfer.js – dokumentacja rozszerzenia plugins https://wavesurfer.xyz/docs/modules/plugins_regions (stan na: 13.05.2026).
- [31]. REST API – film firmy IBM <https://www.ibm.com/think/topics/rest-apis#:~:text=and%20Ian%20Smalley-.REST%20APIs%20defined,to%20connect%20distributed%20hypermedia%20systems>. (stan na: 13.05.2026).
- [32]. Docker – strona oficjalna <https://www.docker.com/> (stan na 13.05.2026).
- [33]. JUnit – strona oficjalna <https://junit.org/> (stan na 13.05.2026).
- [34]. Write Once, Run Anywhere (WORA) – artykuł opicujacy <https://softwareskill.pl/jak-dziala-garbage-collector-zarzadzanie-pamiecia-jvm> (stan na 20.05.2026).
- [35]. Garbage Collector – artykuł opisujacy <https://softwareskill.pl/jak-dziala-garbage-collector-zarzadzanie-pamiecia-jvm> (stan na: 20.05.2026).
- [36]. Single Page Application (SPA) – artykuł opisujacy i porównujacy z Multi Page Application <https://www.outsystems.com/application-development/spa-single-page-app-defined-with-examples/#benefits> (stan na: 20.05.2026).
- [37]. Virtual DOM – artykuł opisujacy <https://www.geeksforgeeks.org/reactjs/reactjs-virtual-dom/> (stan na: 20.05.2026).
- [38]. ACID – dokumentacja PostgreSQL <https://www.postgresql.org/docs/current/glossary.html#GLOSSARY-CONSISTENCY> (stan na: 20.05.2026).
- [39]. Inversion of Control (IoC) – dokumentacja techniczna Spring <https://docs.spring.io/spring-framework/reference/core/beans/introduction.html> (stan na: 20.05.2026).
- [40]. Dependency Injection (DI) – dokumentacja techniczna Spring <https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html> (stan na: 20.05.2026).
- [41]. Spring Boot Starter Web – dokumentacja techniczna <https://docs.spring.io/spring-boot/reference/web/index.html> (stan na: 13.05.2026).
- [42]. Spring Boot Data JPA – dokumentacja techniczna <https://docs.spring.io/spring-data/jpa/reference/index.html> (stan na: 13.05.2026).

- [43]. Spring Boot Starter Validation – artykuł opisujący możliwości i zastosowanie <https://dev.to/gianfcop98/spring-boot-and-validation-a-complete-guide-with-valid-and-validated-471p> (stan na: 13.05.2026).
- [44]. JWT – dokumentacja techniczna <https://javadoc.io/doc/io.jsonwebtoken/jjwt-api/latest/index.html> (stan na: 13.05.2026).
- [45]. JWT – dokumentacja techniczna <https://www.jwt.io/introduction#what-is-json-web-token> (stan na: 25.05.2026).
- [46]. Spring Boot Dev Tools – dokumentacja techniczna <https://docs.spring.io/spring-boot/reference/using/devtools.html> (stan na: 13.05.2026).
- [47]. JPA (Jakarta Persistence API) – dokumentacja techniczna <https://docs.hibernate.org/orm/7.3/javadocs/> (stan na: 20.05.2026).
- [48]. Web Audio API – dokumentacja techniczna https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API (stan na: 21.05.2026).
- [49]. HTML5 Canvas – dokumentacja techniczna https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API (stan na: 21.05.2026).
- [50]. wavesurfer.js Regions Plugin – dokumentacja techniczna https://wavesurfer.xyz/docs/modules/plugins_regions (stan na: 21.05.2026).
- [51]. wavesurfer.js Timeline Plugin – dokumentacja techniczna https://wavesurfer.xyz/docs/modules/plugins_timeline (stan na: 21.05.2026).
- [52]. GraphQL – strona oficjalna <https://graphql.org/> (stan na: 13.05.2026).
- [53]. DTO – artykuł opisujący założenia <https://medium.com/@mariorodriguezgalicia/what-is-a-dto-in-spring-boot-and-why-should-you-use-it-97651506e516> (stan na: 19.05.2026).
- [54]. Docker Volumes – dokumentacja techniczna <https://docs.docker.com/engine/storage/volumes/> (stan na: 21.05.2026).
- [55]. Docker Compose – dokumentacja techniczna <https://docs.docker.com/compose/> (stan na: 21.05.2026).
- [56]. Piramida testów – artykuł opisujący <https://medium.com/@kchaves001/the-test-pyramid-insights-from-15-years-of-experience-4adbdaf300f5> (stan na: 21.05.2026).
- [57]. Microsoft Clarity – strona oficjalna <https://clarity.microsoft.com/> (stan na 04.06.2026).
- [58]. Google Drive API – dokumentacja techniczna <https://developers.google.com/workspace/drive/api/reference/rest/v3> (stan na: 10.05.2026).
- [59]. JDBC – artykuł opisujący podstawy pracy z bazą danych i Javą <https://javastart.pl/baza-wiedzy/java-ee/jdbc-podstawy-pracy-z-baza-danych> (stan na: 25.05.2026).
- [60]. Mechanizm refleksji Java – artykuł opisujący <https://javastart.pl/baza-wiedzy/sownik/refleksja> (stan na: 25.05.2026).
- [61]. CRUD – artykuł opisujący założenia <https://www.codecademy.com/article/what-is-crud-explained> (stan na: 25.05.2026).
- [62]. N+1 problem – artykuł wyjaśniający <https://medium.com/databases-in-simple-words/the-n-1-database-query-problem-a-simple-explanation-and-solutions-ef11751aef8a> (stan na: 25.05.2026).

- [63]. JPQL – dokumentacja techniczna <https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html> (stan na: 25.05.2026).
- [64]. JSON – dokumentacja techniczna <https://www.json.org/json-en.html> (stan na: 25.05.2026).
- [65]. Java record – dokumentacja techniczna <https://docs.oracle.com/en/java/javase/17/language/records.html> (stan na: 25.05.2026).
- [66]. Jackson JSON – dokumentacja techniczna w frameworku Springu <https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-methods/jackson.html> (stan na: 25.05.2026).
- [67]. MultipartFile – dokumentacja techniczna <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org.springframework.web.multipart/MultipartFile.html> (stan na: 25.05.2026).
- [68]. OIDC – artykuł <https://www.microsoft.com/pl-pl/security/business/security-101/what-is-openid-connect-oidc> (stan na: 25.05.2026).
- [69]. Google Cloud – dokumentacja techniczna <https://docs.cloud.google.com/docs> (stan na: 25.05.2026).
- [70]. CORS – artykuł opisujący działanie <https://www.baeldung.com/spring-cors> (stan na: 25.05.2026).
- [71]. XSS – artykuł opisujący <https://owasp.org/www-community/attacks/xss/> (stan na: 25.05.2026).
- [72]. java.nio.files – dokumentacja techniczna <https://docs.oracle.com/javase/8/docs/api/java/nio/file/package-summary.html> (stan na: 25.05.2026).
- [73]. UUID – dokumentacja techniczna <https://docs.oracle.com/javase/8/docs/api/java/util/UUID.html> (stan na: 25.05.2026).
- [74]. Google API Client Library – dokumentacja techniczna <https://developers.google.com/api-client-library/java> (stan na: 25.05.2026).
- [75]. Proxy – artykuł opisujący <https://www.fortinet.com/resources/cyberglossary/proxy-server> (stan na: 25.05.2026).
- [76]. Smart Component – artykuł porównujący rodzaje komponentów <https://medium.com/@abubakarmemon/dumb-smart-components-in-react-0d19e2653398> (stan na: 25.05.2026).
- [77]. Context API – dokumentacja techniczna <https://react.dev/reference/react/createContext> (stan na: 25.05.2026).
- [78]. Prop drilling – artykuł <https://www.geeksforgeeks.org/reactjs/what-is-prop-drilling-and-how-to-avoid-it/> (stan na: 25.05.2026).
- [79]. Axios – dokumentacja techniczna <https://www.npmjs.com/package/axios> (stan na: 25.05.2026).
- [80]. Drzewo DOM – artykuł opisujący <https://uniquedevs.com/blog/react-js-virtual-dom/> (stan na: 26.05.2026).

- [81]. Responsive Web Design (RWD) – artykuł https://pl.wikipedia.org/wiki/Responsywne_projektowanie_stron_internetowych (stan na: 26.05.2026).
- [82]. Adaptive Design – artykuł opisujący <https://semcore.pl/czym-jest-design-adaptacyjny-adaptive-design/> (stan na: 26.05.2026).
- [83]. Mockito – dokumentacja techniczna <https://javadoc.io/doc/org.mockito/mockito-core/latest/org.mockito/org/mockito/Mockito.html> (stan na 31.05.2026).
- [84]. SUS – artykuł opisujący <https://yourcx.io/pl/blog/2024/09/system-usability-scale-sus-w-badaniach-ux-kluczowe-narzedzie-oceny-uzytecznosci/> (stan na: 04.06.2026).
- [85]. Artykuł Jakoba Nielsena badający potrzebną liczbę testerów aplikacji <https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/> (stan na: 04.06.2026).
- [86]. Typ MIME – artykuł https://pl.wikipedia.org/wiki/Typ_MIME (stan na: 07.06.2026).
- [87]. Journal of Usability Studies – artykuł https://uxpajournal.org/wp-content/uploads/sites/7/pdf/JUS_Bangor_May2009.pdf (stan na: 07.06.2026).

10. Spis Listingów

Listing 1 – Przykład encji dla klasy Project. Źródło: Opracowanie własne.....	38
Listing 2 – Przykład interfejsu rozszerzającego JpaRepository dla klasy Band. Źródło: Opracowanie własne.....	40
Listing 3 – przykład obiektu DTO dla klasy Projekt. Opracowanie własne.	40
Listing 4 – przykład kontrolera REST API przetwarzającego obiekty DTO dla klasy Zespół. Opracowanie własne.....	41
Listing 5 – przykład kontrolera REST API przetwarzającego plik przez MultipartFile. Opracowanie własne.....	41
Listing 6 – Konfiguracja łańcucha filtrów bezpieczeństwa w klasie SecurityConfig. Opracowanie własne.....	42
Listing 7 – Implementacja serwisu CustomOAuth2UserService przetwarzającego profil OIDC. Opracowanie własne.....	43
Listing 8 – Obsługa sukcesu uwierzytelniania i osadzenia tokenu JWT w ciasteczku HttpOnly. Opracowanie własne.....	44
Listing 9 – klasa JwtService odpowiedzialna za operacje kryptograficzne na tokenach. Opracowanie własne.....	45
Listing 10 – implementacja filtra JwtAuthenticationFilter autoryzującego żądania. Opracowanie własne.....	46
Listing 11 – funkcja tworzenia komentarza. Opracowanie własne.....	48
Listing 12 – funkcja aktualizująca nazwę i opis projektu dostępna tylko dla producenta i właściciela. Opracowanie własne.	49
Listing 13 – funkcja sprawdzająca uprawnienia. Opracowanie własne.	49
Listing 14 – funkcja odpowiedzialna za obsługę zatwierdzania projektu. Opracowanie własne	49
Listing 15 – funkcja zarządzająca statusem projektu. Opracowanie własne.	50
Listing 16 – funkcja dodawania nowej wersji utworu odpowiedzialna za wersjonowanie. Opracowanie własne.....	50
Listing 17 – implementacja zapisu pliku na serwerze z wykorzystaniem unikalnych identyfikatorów UUID. Opracowanie własne.	52
Listing 18 – funkcja do przesyłania pliku do przestrzeni appDataFolder. Opracowanie własne.	53
Listing 19 – funkcja pobierająca strumień danych z Google Drive i przysyłająca go do punktu końcowego REST API. Opracowanie własne.	53
Listing 20 – implementacja bezpiecznego usuwania. Opracowanie własne.....	54
Listing 21 – kontekst odpowiedzialny za autoryzację. Źródło: Opracowanie własne.	55
Listing 22 – hook dający dostęp do stanu aplikacji. Źródło: Opracowanie własne	56
Listing 23 – fragment inicjalizacji stanów w komponencie Projects. Źródło: Opracowanie własne.....	57
Listing 24 – dopisanie danych do istniejącej tablicy. Źródło: Opracowanie własne.	57
Listing 25 – funkcja pobierania danych z API. Źródło: Opracowanie własne.....	58
Listing 26 – funkcja wysyłania danych z zasobami do API. Źródło: Opracowanie własne. ...	59
Listing 27 – fragment funkcji pobierającej plik i informacje o projekcie. Źródło: Opracowanie własne.....	60
Listing 28 – inicjalizacja komponentu wavesurfer.js. Źródło: Opracowanie własne.	60
Listing 29 – Filtr zdarzeń separujący logikę osi czasu od fali dźwiękowej. Źródło: Opracowanie własne.....	61
Listing 30 – funkcja synchronizacji komentarzy z regionami biblioteki wavesurfer.js. Źródło: Opracowanie własne.....	62

Listing 31 – funkcja odpowiedzialna za obliczenia podczas zaznaczania. Źródło: Opracowanie własne.....	62
Listing 32 - funkcja odpowiedzialna za obliczenie całości zaznaczenia. Źródło: Opracowanie własne.....	63
Listing 33 – Konfiguracja ścieżek routingu. Źródło: Opracowanie własne.	63
Listing 34 – implementacja renderowania warunkowego. Źródło: Opracowanie własne.	64
Listing 35 – Struktura testu jednostkowego. Źródło: Opracowanie własne.....	70
Listing 36 – struktura testu z negatywnym scenariuszem. Źródło: Opracowanie własne.	71

11. Spis ilustracji

Rysunek 1 - Odtwarzacz Dropbox. Źródło: [2].....	9
Rysunek 2 - Odtwarzacz Pro Tools. Źródło: [18]	10
Rysunek 3 - Panel zarządzania sesją Cloud Collaboration. Źródło: [19].....	10
Rysunek 4 - Odtwarzacz Opusnix. Źródło [23]	11
Rysunek 5 - Odtwarzacz Dropbox Replay. Źródło: [24].	12
Rysunek 6 - Lista Markerów Pro Tools. Źródło: [16].....	13
Rysunek 7 - Opusnix Widok główny. Źródło: [23].....	14
Rysunek 8 - Hierarchia aktorów. Źródło: Opracowanie własne.	20
Rysunek 9 - Przypadki użycia dla aktora abstrakcyjnego. Źródło: Opracowanie własne.	20
Rysunek 10 - Przypadki użycia dla aktora Muzyk. Źródło: Opracowanie własne.	21
Rysunek 11 - Przypadek użycia dla aktora Owner. Źródło: Opracowanie własne.	21
Rysunek 12 - Przypadki użycia dla aktora producent. Źródło: Opracowanie własne.....	22
Rysunek 13 - Analityczny diagram klas. Źródło: Opracowanie własne.	24
Rysunek 14 - Projektowy diagram klas. Źródło: Opracowanie własne.	26
Rysunek 15 - Makieta interfejsu użytkownika. Źródło: Opracowanie własne.	28
Rysunek 16 - Logotyp WaveScope. Źródło: Opracowanie własne.....	29
Rysunek 17 - Dashboard systemu WaveScope. Źródło: Opracowanie własne.....	65
Rysunek 18 – Widok zespołu w systemie WaveScope. Źródło: Opracowanie własne	66
Rysunek 19 – Odtwarzacz systemu WaveScope. Źródło: Opracowanie własne.	68
Rysunek 20 – Mobilna wersja odtwarzacza WaveScope. Źródło: Opracowanie własne.	69
Rysunek 21 – mapa kliknięć w widoku zespołu. Źródło: Opracowanie własne.	73
Rysunek 22 – Mapa kliknięć w widoku odtwarzacza. Źródło: Opracowanie własne.	74