AKADEMIA TARNOWSKA

Wydział Politechniczny

Kierunek: ***Informatyka***

2024/2025

Jakub Piekielniak

PRACA INŻYNIERSKA

Projekt i implementacja aplikacji webowej do zarządzania pojazdami

Promotor pracy:

mgr inż. Tomasz Gądek

Tarnów, *2025*

Spis treści

1. Wstęp

W czasach dynamicznego rozwoju technologii informatycznych aplikacje webowe stają się nieodzownym narzędziem w życiu codziennym. Niniejsza praca dyplomowa skupia się na projekcie oraz implementacji aplikacji webowej, która ma na celu usprawnienie procesu zarządzania pojazdami, oferując intuicyjne i kompleksowe rozwiązanie.

1.1. Motywacja

Motywacją aby zaprojektować aplikację do zarządzania pojazdami jest własna obserwacja z powszechnych problemów związanych z tradycyjnymi metodami rejestrowania informacji o przeglądach, ubezpieczeniach i naprawach. Wielu właścicieli pojazdów polega na prowizorycznych rozwiązaniach w celu przechowywania informacji. Często są to zapiski w zeszytach, na luźnych kartkach, notatki w telefonie lub po prostu poleganie na własnej pamięci. Takie metody łatwo zawodzą. Notatki można zgubić czy też przypadkowo usunąć, a informacje przechowywane w pamięci są podatne na zapomnienie. Te niezorganizowane podejścia mogą prowadzić do przeoczenia ważnych terminów związanych z pojazdem.

Aplikacja webowa ma być rozwiązaniem tych problemów poprzez zapewnienie centralnego, cyfrowego miejsca do przechowywania i zarządzania wszystkimi istotnymi informacjami dotyczącymi pojazdu. Dzięki temu użytkownicy mogą uzyskać łatwy dostęp do historii serwisowej, terminów przeglądów oraz innych kluczowych danych, co przyczynia się do lepszego, kompleksowego utrzymania pojazdu i potencjalnego przedłużenia jego żywotności.

1.2. Cel pracy

Celem niniejszej pracy jest zaprojektowanie i implementacja aplikacji webowej do efektywnego zarządzania pojazdami. Aplikacja zaimplementowana przy użyciu technologii .NET i Angular, ma na celu rozwiązanie problemów związanych z powszechnymi metodami przechowywania informacji o pojazdach. Oferuje użytkownikom odpowiednio dostosowane narzędzie z szerokim zakresem możliwości, przedstawionych w czytelnym i przejrzystym interfejsie graficznym.

1.3. Zakres pracy

Zakres pracy obejmuje kompleksowy proces tworzenia aplikacji webowej do zarządzania pojazdami, składający się z następujących etapów:

1. **Analiza wymagań użytkownika:**
   Identyfikacja kluczowych potrzeb potencjalnych użytkowników aplikacji oraz określenie priorytetów w zakresie użyteczności i interfejsu. Celem jest opracowanie aplikacji, która będzie w pełni odpowiadać oczekiwaniom i potrzebom użytkowników.
2. **Dobór odpowiednich technologii:**
   Wybór odpowiednich narzędzi i frameworków do realizacji, ze szczególnym uwzględnieniem technologii .NET dla aplikacji serwerowej, Angular dla aplikacji klienckiej oraz PostgreSQL jako baza danych wraz z Azure Blob Storage jako chmurowy magazyn przechowywania zdjęć. Wybór ten jest podyktowany możliwościami tych technologii w tworzeniu wydajnych i skalowalnych aplikacji webowych.
3. **Implementacja systemu:**
   Ten etap skupia się na praktycznej implementacji aplikacji. Obejmuje on dwa główne obszary: rozwój serwera w .NET, w tym tworzenie logiki biznesowej i zarządzanie bazą danych, oraz budowę interfejsu użytkownika w Angular. Istotnym elementem jest również integracja obu części systemu zapewniając płynne działanie całej aplikacji.
4. **Testowanie możliwości systemu:**
   Ostatni etap obejmuje kompleksowe sprawdzenie możliwości aplikacji. Przeprowadzenie testów jednostkowych poszczególnych komponentów, testów integracyjnych sprawdzających współdziałanie różnych części systemu. Celem tego etapu jest zapewnienie niezawodności, efektywności i intuicyjności tworzonego rozwiązania.

Każdy z tych etapów odgrywa kluczową rolę dla zaprojektowania efektywnego i niezawodnego rozwiązania, które spełnia potrzeby współczesnych użytkowników.

2. Analiza biznesowa

W rozdziale zostaną przedstawione wymagania aplikacji, podzielone na dwie dwie kategorie wymagań: funkcjonalne i niefunkcjonalne. Spełnienie obu grup wymagań jest kluczowe dla zapewnienia efektywności i jakości zaimplementowanego oprogramowania.

2.1. Wymagania funkcjonalne

Głównym zadaniem aplikacji jest umożliwienie kompleksowego zarządzania pojazdami, przeglądami technicznymi, historią napraw oraz ubezpieczeniami. W związku z tym użytkownik powinien mieć możliwość:

1. **Rejestracji i logowania** do systemu przy użyciu adresu e-mail oraz hasła, zapewniając bezpieczny i spersonalizowany dostęp do aplikacji.
2. **Dodawania nowych pojazdów** do systemu oraz edytowania ich danych, takich jak numer rejestracyjny, marka, model, rok produkcji i inne kluczowe informacje techniczne.
3. **Wprowadzania i aktualizowania informacji** dotyczących przeglądów technicznych i serwisów, pozwalając na bieżące śledzenie stanu technicznego pojazdu.
4. **Rejestrowania danych związanych z ubezpieczeniami pojazdów**, takich jak numer polisy, termin ważności, nazwę ubezpieczyciela.
5. **Przeglądania listy wszystkich pojazdów** wraz ze szczegółowymi informacjami o każdym z nich.
6. **Usuwania wprowadzonych danych** z systemu, w tym szczegółów dotyczących pojazdów, przeglądów oraz ubezpieczeń.
7. **Otrzymywania automatycznych powiadomień** o zbliżającym się terminie wygaśnięcia ubezpieczenia, przeglądu pojazdu, co umożliwia zaplanowanie odpowiednich działań, aby uniknąć przerw w ochronie.
8. **Trwałego usunięcia konta** z systemu wraz z wszystkimi danymi, zapewniając pełną kontrolę nad swoją obecnością w aplikacji i prywatnością danych.

2.2. Wymagania niefunkcjonalne

Wymagania niefunkcjonalne określają standardy i kryteria, które nie dotyczą bezpośrednio funkcji systemu, ale mają kluczowy wpływ na jego bezpieczeństwo, wydajność i przejrzystość.

2.2.1. Bezpieczeństwo

Zapewnienie bezpieczeństwa aplikacji jest kluczowym aspektem, szczególnie w kontekście ochrony wrażliwych danych użytkowników, w tym szczegółowych informacji o pojazdach, historii serwisowej i ubezpieczeniowej. W tym celu aplikacja powinna korzystać z tokenów JWT (ang. *JSON Web Token*), które zapewniają bezpieczną weryfikację tożsamości i przesyłanie danych w sposób zaszyfrowany. Wprowadzając ten mechanizm użytkownik nie ma konieczności podawania danych uwierzytelniających przy każdym zapytaniu do systemu. Konieczność ponownego logowania pojawia się jedynie po upływie ważności tokenu.

2.2.2. Wydajność

Aplikacja powinna zapewniać płynną komunikację z serwerem poprzez optymalizację wydajności. Użytkownik, odwołując się do odpowiedniego zasobu, powinien uzyskać dostęp w jak najkrótszym czasie. Zoptymalizowane zapytania i efektywne zarządzanie zasobami powinny umożliwiać szybsze przetwarzanie informacji, co pozytywnie wpłynie na komfort oraz satysfakcję użytkowników. W rezultacie aplikacja powinna być bardziej elastyczna, przyjazna, zwiększając przy tym ogólne zadowolenie.

2.2.3. Przejrzysty interfejs

Kluczowym elementem niefunkcjonalnych wymagań aplikacji, wpływającym na jej efektywność i zadowolenie użytkowników, jest przejrzysty interfejs. Powinien on spełniać takie kryteria jak intuicyjność, czytelność, minimalizm, dzięki czemu ułatwi nawigację i zwiększy efektywność korzystania z systemu. Taki interfejs minimalizuje ryzyko błędów wynikających z nieprawidłowego zrozumienia danych. Stosując się do tych praktyk aplikacja staje się bardziej przyjazna i łatwiejsza w codziennym użytkowaniu.

3. Stos technologiczny

W tym rozdziale zostanie omówiony stos technologiczny, który dzięki zastosowaniu nowoczesnych narzędzi oraz standardów, pozwala na wykonanie zaawansowanych rozwiązań i odpowiada na potrzeby współczesnych użytkowników w aplikacjach internetowych.

3.1. Aplikacja serwerowa

Warstwa serwerowa (ang. *backend*) została opracowana w formie interfejsu webowego (ang. *Web Application Programming Interfaces)* do którego dostęp można uzyskać za pomocą podstawowym metod i nagłówków HTTP (ang. *Hypertext Transfer Protocol*). Pozwala to na wymianę danych między aplikacjami uruchomionymi na różnych urządzeniach [1]. Aplikacja w głównej mierze oferuje operację CRUD (ang. *create, read, update, delete*) czyli jedne z podstawowych operacji do interakcji z danymi przechowywanymi w bazie [8].

3.1.1. C#

C# jest jednym z nowoczesnych języków programowania zorientowanych obiektowo, którego prace nad powstaniem rozpoczęły się w 1998 roku. Umożliwia tworzenie aplikacji wieloplatformowych takich jak aplikacje mobilne, internetowe, systemy IoT (ang. *Internet of things*). Powstał z inspiracji starszymi językami - C, C++, Java. Jedną z zalet języka jest LINQ (ang. *Language Integrated Query*), narzędzie umożliwiające wykonywanie operacji podobnych składniowo do zapytań SQL na zbiorach danych bezpośrednio w kodzie [5]. Język udostępnia również mechanizm automatycznego zarządzania pamięcią programu w środowisku uruchomieniowym. Tym samym programiści zwolnieni są z zarządzania zasobami gdyż jest to wykonywane dynamiczne, w trakcie działania kodu.

3.1.2. ASP.NET Core

ASP.NET Core to otwarto źródłowy framework do tworzenia nowoczesnych aplikacji połączonych z Internetem dodatkowo obsługując rozwiązania chmurowe [7]. Jest jednym z komponentów wchodzących w skład .NET czyli platformy programistycznej opracowanej przez firmę Microsoft umożliwiającej tworzenie aplikacji w różnych językach. Charakteryzuje się modułową architekturą, która pozwala budować wydajne aplikacje i uruchamiać w różnych środowiskach - Windows, Linux, MacOS [17].

3.1.3. Entity Framework Core

Entity Framework Core to lekka, rozszerzalna biblioteka umożliwiająca dostęp do danych o otwartym kodzie źródłowym opracowana przez firmę Microsoft. Działa jako narzędzie przekształcające obiektowo-relacyjne (O/RM) umożliwiając sprawne zarządzanie danymi przechowywanymi w relacyjnych bazach danych np. PostgreSQL, Microsoft SQL Server. Wykorzystując bibliotekę programista ma możliwość odwzorować obiekt w kodzie C# na struktury bazodanowe bez ręcznych modyfikacji przy zastosowaniu mechanizmu migracji oraz wykonywanie różnych operacji pozbawionych potrzeby pisania zaawansowanych zapytań SQL [10].

3.1.4. JSON Web Token

JSON Web Token to standard, który definiuje bezpieczny przesył informacji w formacie obiektów JSON (ang. *JavaScript Object Notation*) pomiędzy żądaniami w aplikacjach internetowych. Dane przesyłane wewnątrz są podpisane cyfrowo dlatego za każdym razem można je zweryfikować czy pochodzą z odpowiedniego źródła. Standard ten używany jest najczęściej do autoryzacji użytkowników w systemie lub przepływu informacji. Struktura tokenu składa się z trzech części oddzielonych kropkami:

- **Nagłówek** - algorytm użyty do podpisania
- **Ładunek** - oświadczenia dotyczące podmiotu
- **Podpis** - umożliwia weryfikację czy token nie uległ modyfikacjom [11]

Przykładowy token z opisem każdej części został przedstawiony na rysunku 3.1.

Rys. 3.1 - *Token JWT [opracowanie własne]*

3.1.5. xUnit

xUnit jest darmowym frameworkiem do pisania testów w projektach na platformie .NET. Został napisany jako alternatywna, uproszczona opcja względem pozostałych bibliotek takich jak NUnit, MSTest [16]. Aby napisać najprostszy test należy nad metodą zastosować atrybut *[Fact]* lub *[Theory]*, który informuje środowisko testowe, że dana metoda jest testem i zostanie automatycznie uruchomiona przez framework. Należy pamiętać, że stosując pierwszy atrybut metoda nie może posiadać żadnych danych wejściowych. Z kolei odwołując się do drugiego atrybutu test będzie uruchamiany dla różnych przypadków testowych dostarczonych przed uruchomieniem. Parametry testowe mogą zostać przekazane za pomocą atrybutów *[InlineData]* lub *[ClassData]*.

3.1.6. Swagger

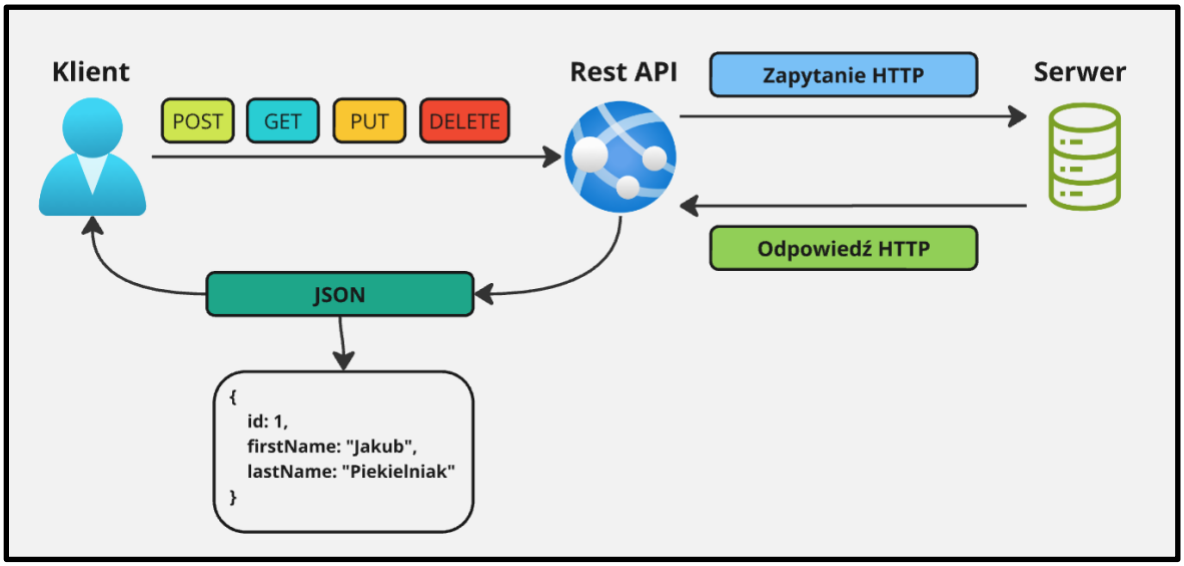
Swagger lub też inaczej specyfikacja OpenAPI to zestaw narzędzi, służący do tworzenia dokumentacji interfejsów REST API. Dokumentacja jest generowana automatycznie na podstawie kontrolerów, punktów końcowych zawartych w kodzie stosując przy tym odpowiednie atrybuty i konfigurację. W frameworku .NET chcąc skorzystać z narzędzia należy zainstalować pakiet NuGet pochodzący z *Swashbuckle.AspNetCore*. Używając *Swagger UI* programista ma dostęp do wizualnej i interaktywnej wersji dokumentacji w przeglądarce, która przedstawia możliwe odwołania do zasobów. Zarówno programiści jak i testerzy mogą w prosty, wydajny sposób testować odpowiednie kontrolery i metody [14].

3.1.7. Azure

Microsoft Azure to platforma udostępniająca rozwiązania chmurowe opracowana przez firmę Microsoft w 2010 roku jako model PaaS (ang. *Platform as a Service*, tłum. Platforma jako usługa). Udostępnia szereg usług służących między innymi do przetwarzania danych, magazynowania czy wgrywania oprogramowania na wersję produkcyjną [12]. W głównej mierze platforma rekomenduje technologię .NET aczkolwiek możliwe jest korzystanie z innych. Wymogiem jest tworzenie oprogramowań, które możliwe są do uruchomienia na środowisku Windows. Zaletą Microsoft Azure jest to, że może być wykorzystywana do współpracy z aplikacjami działającymi zarówno w chmurze jak i uruchomionymi lokalnie na komputerach użytkowników.

3.1.8. REST

REST (ang. *Representational State Transfer*) został opracowany przez Roya Fieldinga jako styl architektury. Opisuje w jaki sposób powinny być tworzone zasoby oraz w jaki sposób uzyskiwać do nich dostęp. Stosowanie tego stylu zapewnia, że system będzie przygotowany na dalszy rozwój, otwarty na różnego rodzaju klientów, a wprowadzanie zmiany nie będą zakłócać działania aplikacji. Mówiąc o API typu REST powinno ono spełniać wymagania takie jak: klient - serwer, bezstanowość, możliwość stosowania mechanizmu cache (buforowania), jednolity interfejs do komunikacji, system oparty na warstwach czy też odpowiednie komunikaty i informacje zwracane w odpowiedzi. Spełnienie wszystkich narzuconych ograniczeń nie jest proste, aczkolwiek prowadzi do ujednolicenia i lepszej pracy z system wraz z rozwojem nowych funkcjonalności [1]. Przykład przepływu komunikacji w REST API z wykorzystaniem formatu JSON został przedstawiony na rysunku 3.2.



Rys. 3.2 - *REST API [opracowanie własne]*

3.2. Aplikacja kliencka

Aplikacja kliencka powinna prezentować przyjazny i wygodny w użyciu interfejs graficzny dla użytkownika końcowego. Do implementacji tych wymagań zastosowano nowoczesne technologie z zakresu tworzenia stron internetowych (ang. *frontend*).

3.2.1. Angular

Angular to otwartoźródłowy framework do tworzenia aplikacji internetowych typu SPA (ang. *Single Page Application*). Opracowany został w języku TypeScript przez firmę Google. Udostępnia szereg narzędzi, bibliotek, które upraszczają i usprawniają pracę programistom. Dodatkowo zapewnia solidną platformę umożliwiając przy tym tworzenie szybkich i niezawodnych aplikacji, które mogą być skalowane zarówno w zależności od wielkości zespołu, jak i rozmiaru kodu [6].

3.2.2. CSS

CSS (ang. *Cascading Style Sheets*) czyli kaskadowe arkusze stylów umożliwiające opisywanie formy prezentowania stron WWW napisanych z wykorzystaniem języku znaczników np. HTML (ang. *HyperText Markup Language*). Jest to zbiór reguł, który ubogaca wygląd odpowiedniego elementu. Głównym celem stosowania w projektach jest oddzielenie struktury dokumentu od formy w jakiej ma być wyświetlany. Przyczynia się dzięki temu do tworzenia stron elastycznych, responsywnych [4].

3.3. Baza danych

Baza danych jest miejscem do przechowywania i zarządzania danymi. Kluczowym aspektem podczas wyboru tego typu dostawcy powinna być niezawodność i wydajność ze względu na przetwarzaną ilość informacji.

3.3.1. PostgreSQL

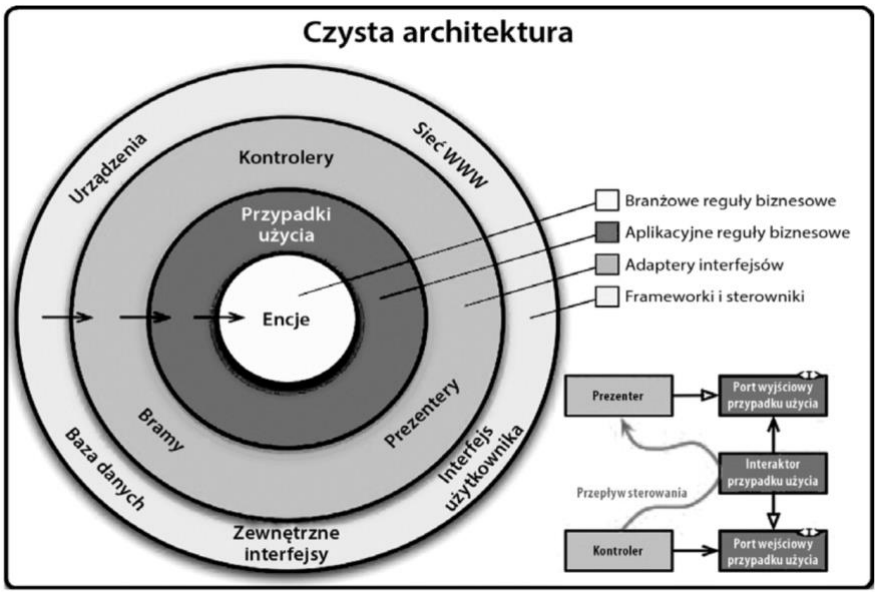
PostgreSQL jest wysokowydajną, relacyjną bazą danych z otwartym kodem źródłowym. Początki sięgają 1986 roku jako część projektu POSTGRES na Wydziale Informatyki Uniwersytetu Kalifornijskiego w Berkeley. Wywodzi się z oryginalnego, ogólnodostępnego kodu Berkeley i obsługuje większość standardów SQL, oferując różnorodne funkcje. Elastyczna licencja PostgreSQL umożliwia bezpłatne używanie, modyfikowanie i dystrybucję w dowolnym celu w tym celach osobistych, komercyjnych lub akademickich [13]. W ostatnich latach zyskuje na popularności zajmując czołowe miejsce w rankingu popularności baz danych [9].

4. Implementacja systemu

System obejmuje kluczowe elementy, które łączone ze sobą stanowią fundament jego możliwości. W tym rozdziale zostaną przedstawione poszczególne części składające się na działanie aplikacji. Architektura, diagramy, dokumentacja czyli podstawowe człony podczas tworzenia dedykowanego systemu.

4.1. Architektura systemu

Tworząc system od podstaw należy zastanowić się nad wybraniem odpowiedniej architektury według której będzie on implementowany. W tym przypadku oprogramowanie zostało wykonane wykorzystując czystą architekturę (ang. *clean architecture*). Celem tej koncepcji jest podział aplikacji na odpowiednie warstwy, gdzie każda z nich ma przewidzianą własną odpowiedzialność biznesową. Komunikacja pomiędzy nimi przepływa od warstw najbardziej wysuniętych na zewnątrz do wewnętrznych. Sprawia to, że elementy znajdujące się w warstwach niższych nie mają dostępu do elementów z warstw wyższych. Stosując się do wszelkich wytycznych to znaczy - podział na warstwy, zasadę zależności system przygotowany jest na łatwe testowanie oraz wydajny rozwój. Umożliwia również bez większego nakładu pracy zmianę elementów zewnętrznych - baza danych, biblioteki. Na rysunku 4.1 został przedstawiony kręg obrazujący czystą architekturę wraz ze szczegółowym opisem za co odpowiedzialna jest każda z warstw [3].



Rys. 4.1 - *Czysta architektura [3]*

Implementując system użyty został również wzorzec projektowy CQRS (ang. *Command Query Responsibility Segregation*). Podejście to zakłada podział operacji wykonywanych w bazie danych na komendy (ang. *command*) oraz kwerendy (ang. *query*). Komendy czyli operację, które zmieniają stan w bazie danych to znaczy wszelkie akcje dodawania, edycji danych czy usuwania. Natomiast kwerendy to operację, które jedynie pobierają odpowiednie dane nie zmieniając przy tym stanu w bazie danych. Wzorzec ten sprawia, że system staje się wydajny, skalowalny oraz zwiększa swoje zabezpieczenia [15]. Aby CQRS można było stosować w projekcie zalecane jest używanie wzorca Mediator, którego celem jest oddelegowanie wykonania danej komendy lub kwerendy na podstawie interfejsu *IRequest* do odpowiedniej klasy implementującej interfejs *IRequestHandler*. W .NET programista po odpowiedniej konfiguracji musi jedynie wywołać metodę *Send()* na obiekcie *Mediator* przekazując argumenty do metody i na tej podstawie będzie wiedział, który obiekt ma wywołać do realizacji zapytania przychodzącego na serwer. Aby lepiej zrozumieć wykorzystanie tego wzorca, na rysunku 4.2 przedstawiono przykład kwerendy zwracającej szczegóły danego serwisu. Natomiast rysunek 4.3 przedstawia przykład komendy, która przypisuje zdjęcie do wskazanego pojazdu i wysyła je do magazynu chmurowego Blob Storage w portalu Azure.

```
internal sealed class GetServiceQueryHandler(
    IServiceBookRepository serviceBookRepository
) : IRequestHandler<GetServiceQuery, ServiceDetailsDto>
{
    public async Task<ServiceDetailsDto> Handle(GetServiceQuery query,
        CancellationToken cancellationToken)
    {
        var serviceBook = await serviceBookRepository
                                .GetAsync(query.ServiceBookId,
                                cancellationToken,
                                asNoTracking: true
                            ) ?? throw new ServiceBookNotFoundException(query.ServiceBookId);

        var service = serviceBook.Services
                            .FirstOrDefault(s => s.Id == query.ServiceId)
                        ?? throw new ServiceNotFoundException(query.ServiceId);

        return service.AsDetailsDto();
    }
}
```

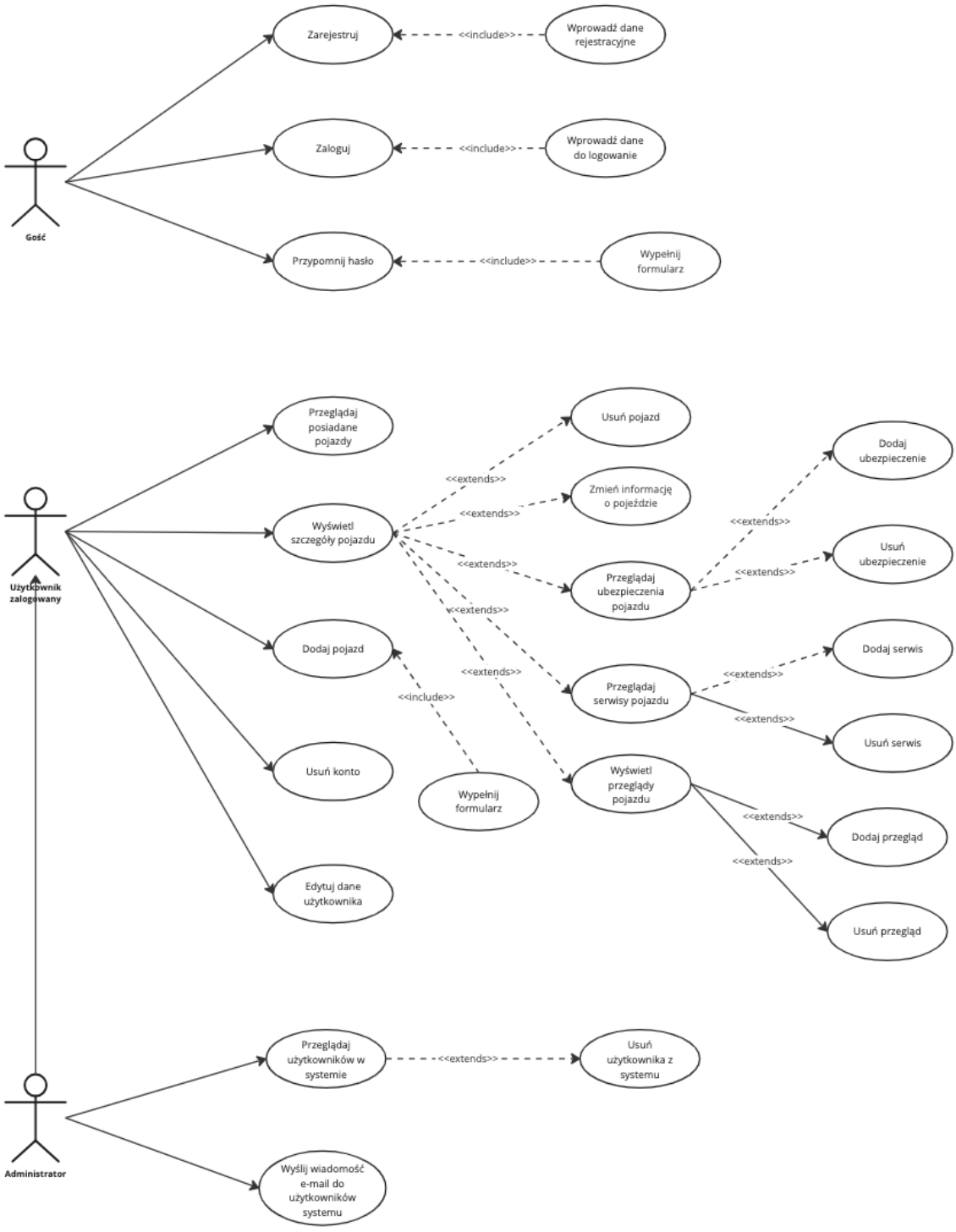
Rys. 4.2 - *CQRS - kwerenda [opracowanie własne]*

```
internal sealed class AddVehicleImageCommandHandler(
    IBlobStorageService blobStorageService,
    IVehicleRepository vehicleRepository,
    IImageRepository imageRepository
) : IRequestHandler<AddVehicleImageCommand, AddVehicleImageResponse>
{
    public async Task<AddVehicleImageResponse> Handle(
        AddVehicleImageCommand command,
        CancellationToken cancellationToken)
    {
        var vehicle = await vehicleRepository.GetAsync(
                            command.VehicleId,
                            cancellationToken
                          )
                      ?? throw new VehicleNotFoundException(command.VehicleId);

        if (vehicle.Image is not null)
        {
            await blobStorageService
                .DeleteImageAsync(vehicle.Image.BlobUrl, cancellationToken);

            await imageRepository
                .DeleteAsync(vehicle.Image, cancellationToken);
        }

        var blobUrl = await blobStorageService
                        .UploadImageAsync(command.Image, cancellationToken);

        var image = Image.Create(
            vehicle.Id,
            blobUrl,
            command.Image.FileName
        );

        await imageRepository.AddAsync(image, cancellationToken);
        await imageRepository.SaveChangesAsync(cancellationToken);

        return new AddVehicleImageResponse(image.BlobUrl);
    }
}
```

Rys. 4.3 - *CQRS - komenda [opracowanie własne]*

4.2. Diagram przypadków użycia

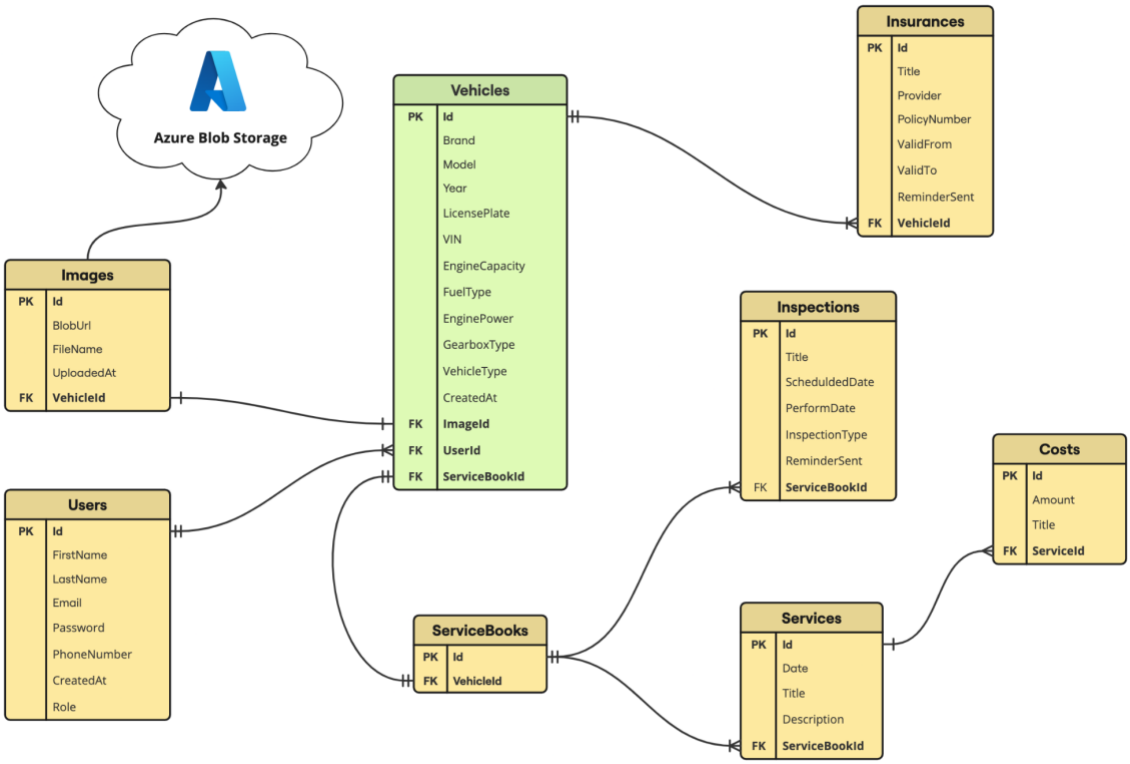
Diagram przypadków użycia (ang. *Use Case Diagram*) to część języka UML (ang. *Unified Modeling Language*). Umożliwia wizualizację dostępnych możliwości dla poszczególnych aktorów w systemie. W omawianej aplikacji można wyróżnić trzech aktorów: gość, użytkownik zalogowany, administrator. Każdy z nich posiada odpowiednie względem pełnionej roli funkcjonalności wchodzące w interakcję z systemem. Rysunek 4.4 przedstawia przypadki użycia, które zostały wdrożone w tworzonym oprogramowaniu.



Rys. 4.4 - *Diagram przypadków użycia [opracowanie własne]*

4.3. Diagram ERD

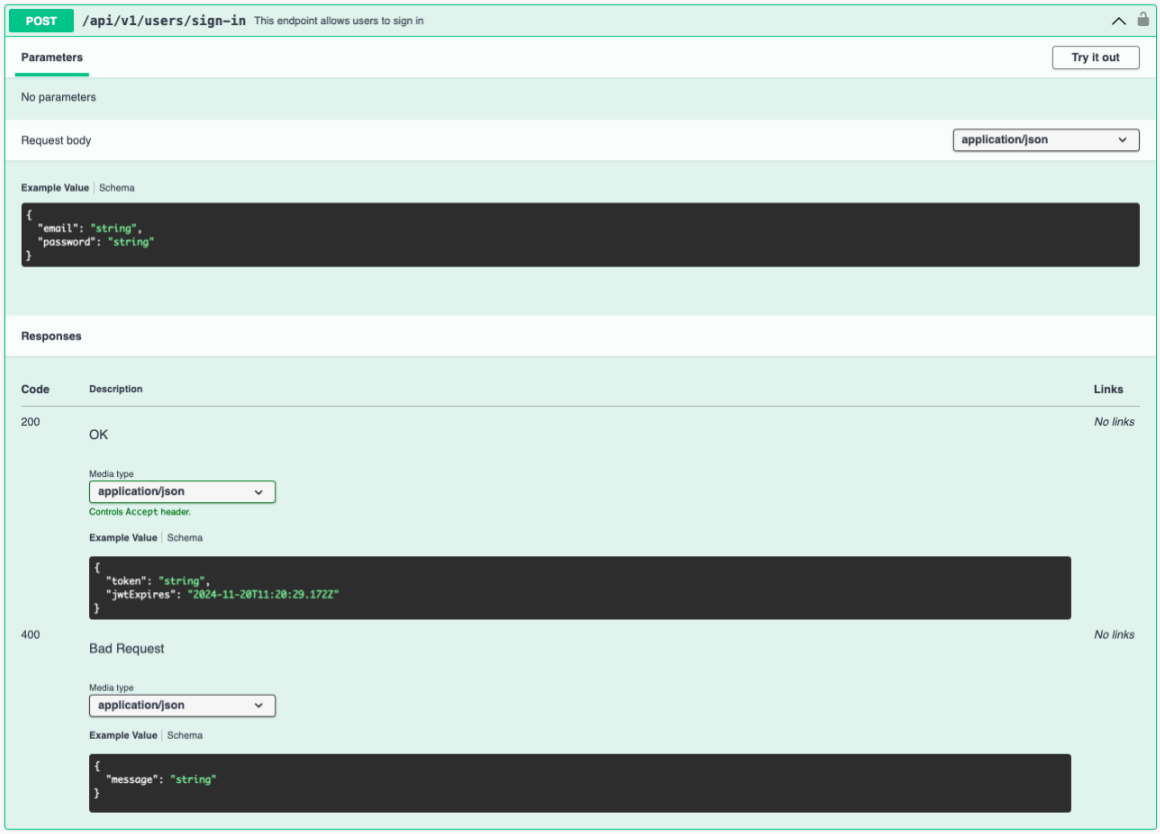
Diagram ERD (ang. *Entity Relationship Diagram*) to przedstawienie w sposób graficzny schematu bazy danych. Umożliwia to zwizualizowanie powiązań, relacji, przepływu danych od jednej encji do drugiej. Rysunek 4.5 przedstawia diagram ERD tworzonego oprogramowania. Kolorem zielonym oznaczono główną encję - *Vehicles*, która stanowi trzon struktury systemu i centralny punkt odniesienia dla pozostałych elementów. Encje oznaczone kolorem żółtym pozostają z nią w relacji, pełniąc role wspierające i uzupełniające w modelu danych. Dodatkowo encja *Images* jest połączona z *Azure Blob Storage*, który służy jako chmurowy magazyn do przechowywania wszystkich wgranych zdjęć w systemie.



Rys. 4.5 - *Diagram ERD [opracowanie własne]*

4.4. Dokumentacja API

Do opracowania dokumentacji API wykorzystano narzędzie *Swagger*, które automatycznie generuje przejrzystą i interaktywną dokumentację. Umożliwia ono graficzne przedstawienie wszystkich modułów systemowych oraz dostępnych endpointów, z których mogą korzystać programiści. Dzięki temu proces integracji oraz korzystania z interfejsu API jest znacznie ułatwiony i bardziej intuicyjny. Każdy endpoint został szczegółowo opisany aby jasno określić sposób działania oraz wymagania. Zawiera jakie dane wejściowe przyjmuje, jakie dane wyjściowe są zwracane oraz odpowiednie statusy protokołu HTTP. Przykład takiego endpointu został przedstawiony na rysunku 4.6.

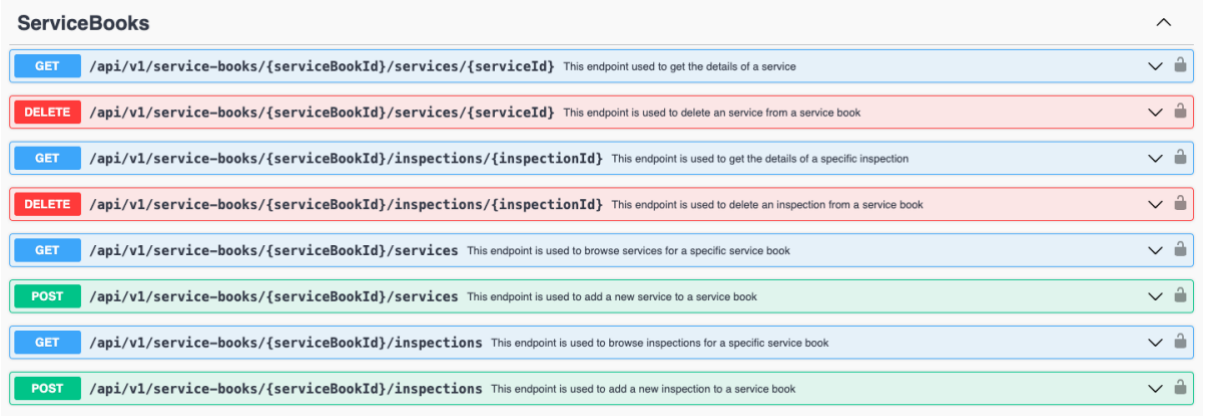


Rys. 4.6 - *Opis endpointu [opracowanie własne]*

Dodatkowo na rysunkach 4.7, 4.8, 4.9 i 4.10 zostały zaprezentowane poszczególne moduły wraz z endpointami w tworzonym systemie.

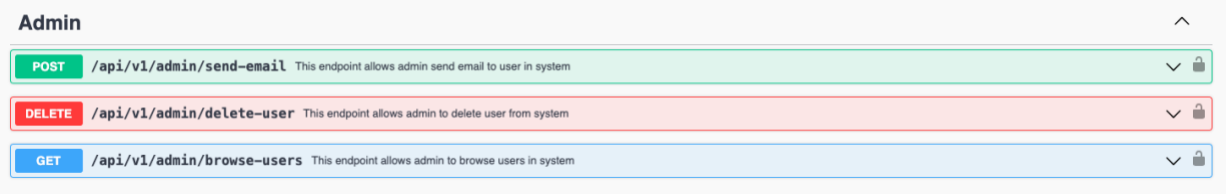
Rys. 4.7 - *Moduł Vehicles [opracowanie własne]*

Moduł *Vehicles* stanowi podstawę aplikacji. Punkt wyjściowy dla wszystkich operacji związanych z pojazdami. Umożliwia operacje z dodawaniem, pobieraniem, edycją czy usunięciem pojazdu. Dodatkowo użytkownik ma możliwość odwołania się w tej sekcji do ubezpieczeń przypisanych do poszczególnych pojazdów. Wszelkie działania są zabezpieczone mechanizmem autoryzacji, wymagającym podania prawidłowego tokenu wygenerowanego przez API.



Rys. 4.8 - *ServiceBooks [opracowanie własne]*

Moduł *ServiceBooks* jest ściśle powiązany z modułem *Vehicles* i stanowi jego uzupełnienie. Przy dodawaniu nowego pojazdu do systemu automatycznie tworzona jest dedykowana książka serwisowa w bazie danych. Umożliwia ona prowadzenie szczegółowej historii serwisowej, obejmującej wszystkie przeglądy oraz naprawy pojazdu, co w znaczny sposób usprawnia zarządzanie jego stanem technicznym.



Rys. 4.9 - *Admin [opracowanie własne]*

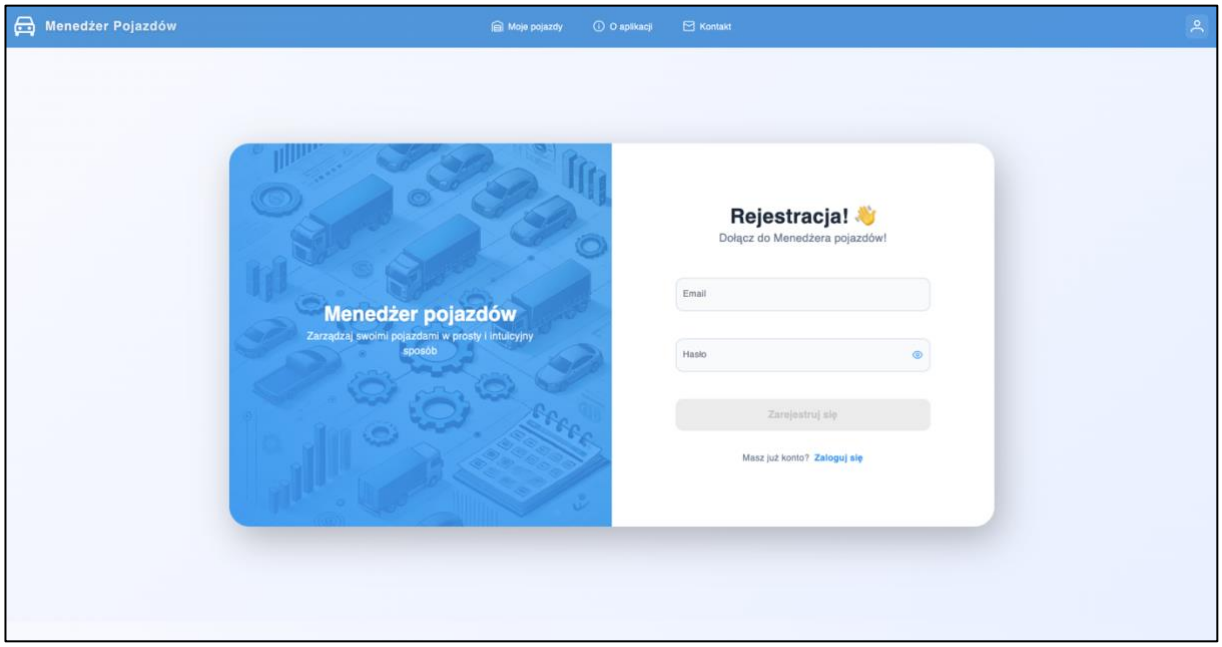
Moduł *Admin* jest najmniejszym komponentem systemu, dedykowanym wyłącznie działaniom administratora. Udostępnia możliwości monitorowania użytkowników oraz ewentualnie usuwanie ich z systemu. Dodatkowo moduł pozwala na wysyłanie wiadomości e-mail do użytkowników, aby ułatwić przekazywanie istotnych informacji.

Rys. 4.10 - *Users [opracowanie własne]*

Moduł *Users* odpowiedzialny jest za działania związane z obsługą użytkowników systemu. Umożliwia utworzenie konta, logowanie, uzupełnienie opcjonalnych danych, pobranie informacji, przypomnienie hasła czy usunięcie swojego profilu. Główną odpowiedzialnością modułu jest zapewnienie autoryzacji i uwierzytelniania użytkowników. Tym samym zwiększa bezpieczeństwo systemu i ochronę danych. Podczas logowania użytkownik dostaję unikalny token który umożliwia mu wykonywanie dalszych akcji w systemie, zapewniając jednocześnie kontrolowany i bezpieczny dostęp do zasobów.

5. Interfejsy użytkownika

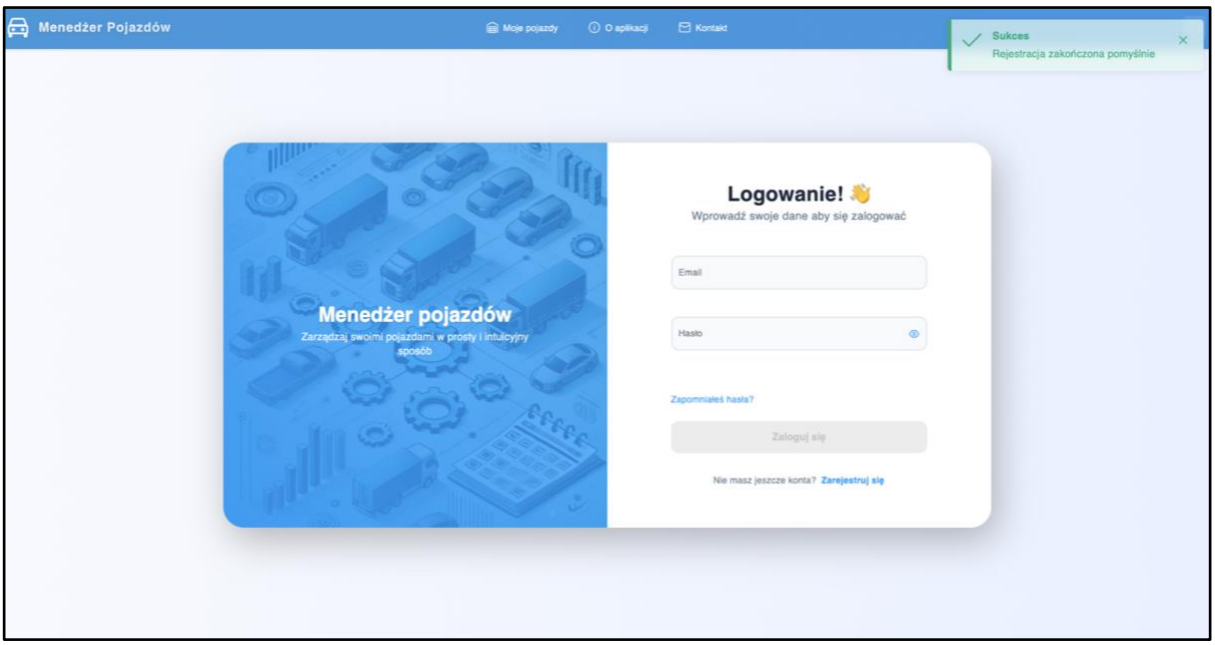
Aplikacja kliencka została zaprojektowana w formie aplikacji webowej, zapewniając użytkownikowi łatwy dostęp za pomocą przeglądarki internetowej. Projekt charakteryzuje się dbałością o estetykę detali oraz intuicyjnością, co przekłada się na to, że korzystanie z niej jest wygodne i przyjemne dla klientów. Opracowane interfejsy graficzne użytkownika zostały przedstawione i omówione w dalszej części rozdziału.



Rys. 5.1 - *Strona rejestracji [opracowanie własne]*

W celu rozpoczęcia korzystania z aplikacji użytkownik musi zarejestrować się w systemie, wypełniając formularz rejestracyjny, przedstawiony na rysunku 5.1. Proces rejestracji został uproszczony do wprowadzenia podstawowych danych, takich jak adres e-mail oraz hasło, pozwalając na szybkie i sprawne założenie konta. Po wprowadzeniu poprawnych danych przycisk *„Zarejestruj się”* staje się aktywny, umożliwiając wysłanie danych z formularza do serwera. Po pomyślnym przetworzeniu zapytania użytkownik zostaje przekierowany na stronę logowania, a także otrzymuje powitalną wiadomość e-mail.

W przypadku gdy użytkownik trafi na stronę rejestracji, mimo że posiada już konto w systemie, ma możliwość skorzystania z opcji *„Masz już konto? Zaloguj się”*, która umożliwia przejście do panelu logowania.



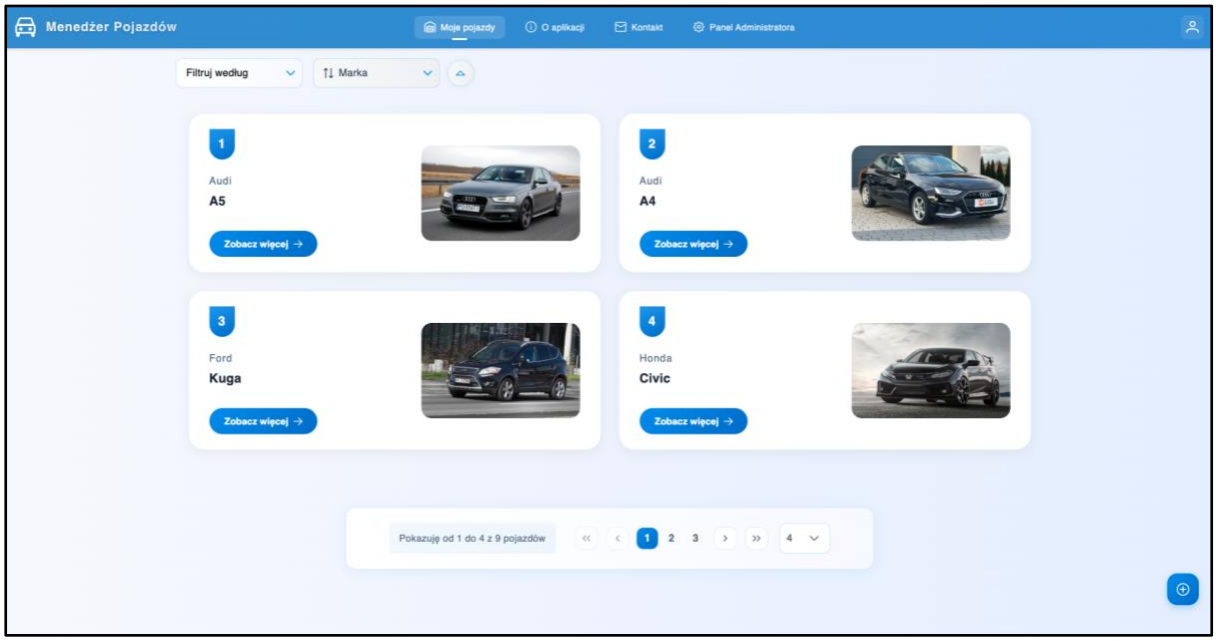
Rys. 5.2 - *Strona logowania [opracowanie własne]*

Po pomyślnym przejściu przez proces rejestracji, użytkownik zostaje przekierowany na stronę logowania, a w prawym górnym rogu aplikacji wyświetlana jest odpowiednia wiadomość, jak przedstawiono na rysunku 5.2. Aby zalogować się do systemu, klient musi podać te same dane, które zostały wprowadzone podczas rejestracji, czyli adres e-mail oraz hasło. Po poprawnym wprowadzeniu danych i pomyślnej walidacji, przycisk *„Zaloguj się”* staje się aktywny, co umożliwia autoryzację użytkownika i w przypadku sukcesu, przekierowanie na podstronę *„Moje pojazdy”*.

Opcja *„Zapomniałeś hasła?”* umożliwia użytkownikowi odzyskanie dostępu do swojego konta w przypadku zapomnienia hasła. Po kliknięciu w ten link, użytkownik zostaje poproszony o podanie adresu e-mail, który jest powiązany z jego kontem w systemie. Po wprowadzeniu adresu e-mail i zatwierdzeniu formularza, system automatycznie wysyła wiadomość e-mail z niezbędnymi informacjami.

Natomiast opcja *„Nie masz jeszcze konta? Zarejestruj się”* umożliwia użytkownikowi założenie nowego konta w systemie. Po kliknięciu w ten link, użytkownik zostaje przeniesiony do panelu rejestracji, gdzie może wprowadzić swoje dane, takie jak adres e-mail oraz hasło, które będą używane do logowania się do aplikacji.

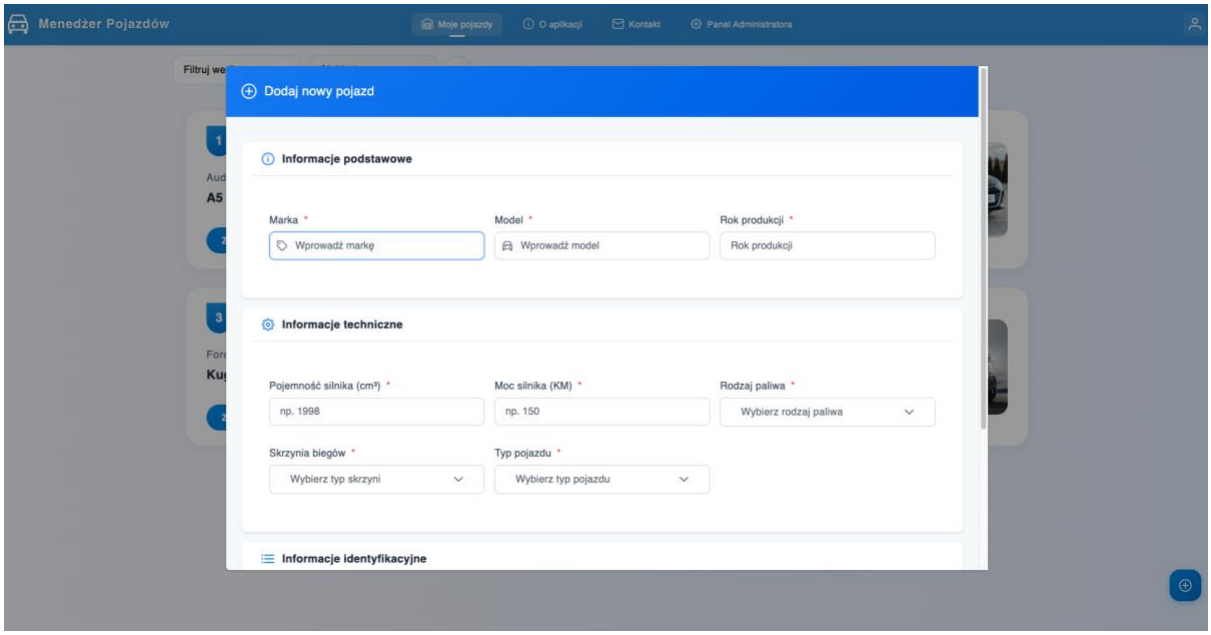
Po pomyślnym zalogowaniu użytkownik zostaje przeniesiony na widok *„Moje pojazdy”*, gdzie wyświetlane są pojazdy przypisane do jego konta, pobrane z bazy danych. W ramach wprowadzonych udoskonaleń, użytkownik ma możliwość filtrowania pojazdów według marki oraz sortowania ich w kolejności rosnącej lub malejącej, na podstawie takich parametrów jak marka czy model. Cały proces odbywa się za pomocą eleganckiego i intuicyjnego interfejsu graficznego, który został przedstawiony na rysunku 5.3.



Rys. 5.3 - *Strona - moje pojazdy [opracowanie własne]*

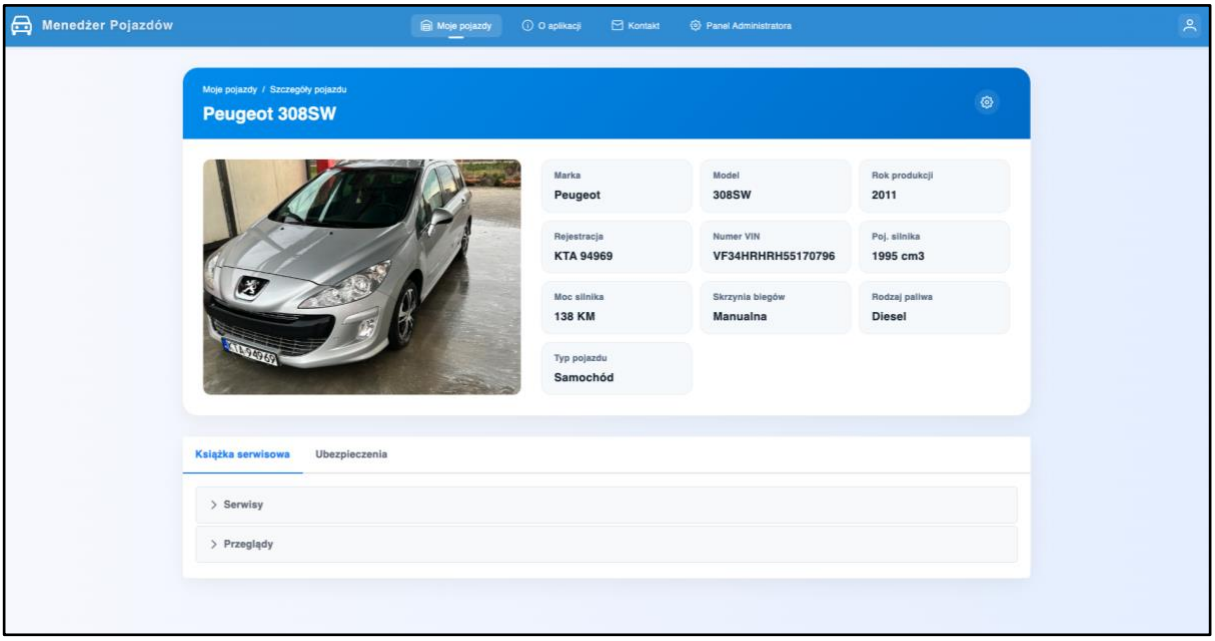
Aby zminimalizować obciążenie serwera i ograniczyć ilość pobieranych danych w jednym momencie, zastosowano mechanizm paginacji stron, który pozwala na pobieranie tylko tych danych, które są aktualnie potrzebne.

Dodatkowo, użytkownik ma możliwość dodania nowego pojazdu do systemu za pomocą przycisku „+”, który pojawia się w prawym dolnym rogu ekranu po załadowaniu komponentu. Po kliknięciu w ten przycisk, wyświetla się okno dialogowe zawierające formularz, w którym użytkownik może wprowadzić szczegółowe informacje na temat pojazdu. Formularz ten pozwala na podanie takich danych jak marka, model, rocznik, numer rejestracyjny czy inne istotne informacje, które pozwolą na dokładne przypisanie pojazdu do konta użytkownika. Przykładowy formularz z danymi dotyczącymi pojazdu został przedstawiony na rysunku 5.4. Proces dodawania pojazdu jest prosty i intuicyjny, co zapewnia użytkownikom łatwość w zarządzaniu swoimi pojazdami w systemie.



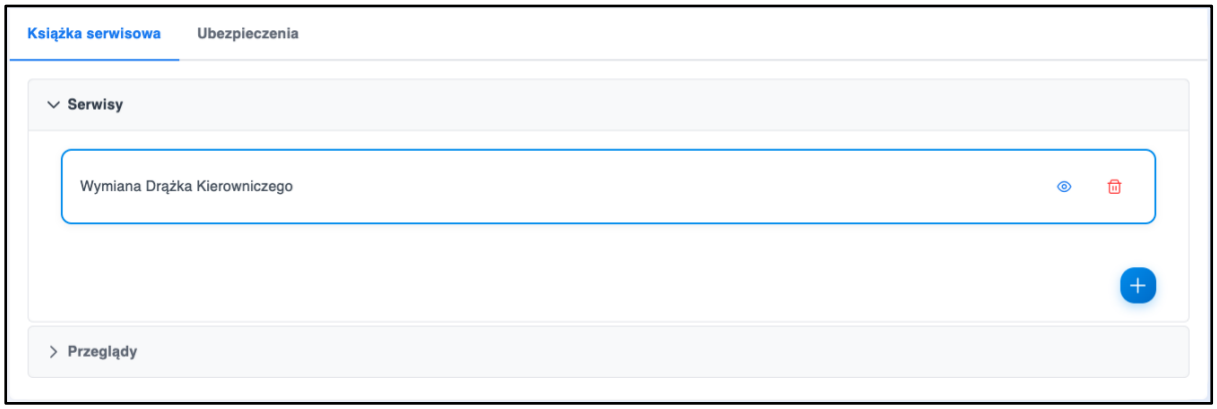
Rys. 5.4 - *Formularz - dodaj nowy pojazd [opracowanie własne]*

Przechodząc do ostatniej możliwości dostępnej na tym ekranie, użytkownik może zobaczyć szczegóły danego pojazdu, klikając przycisk *„Zobacz więcej”*, który jest wyświetlany obok każdego pojazdu. Po kliknięciu, użytkownik zostaje przeniesiony do komponentu, którego zadaniem jest wyświetlenie szczegółowych informacji o wybranym pojeździe. Przykład takiego widoku został przedstawiony na rysunku 5.5.



Rys. 5.5 - *Szczegóły pojazdu [opracowanie własne]*

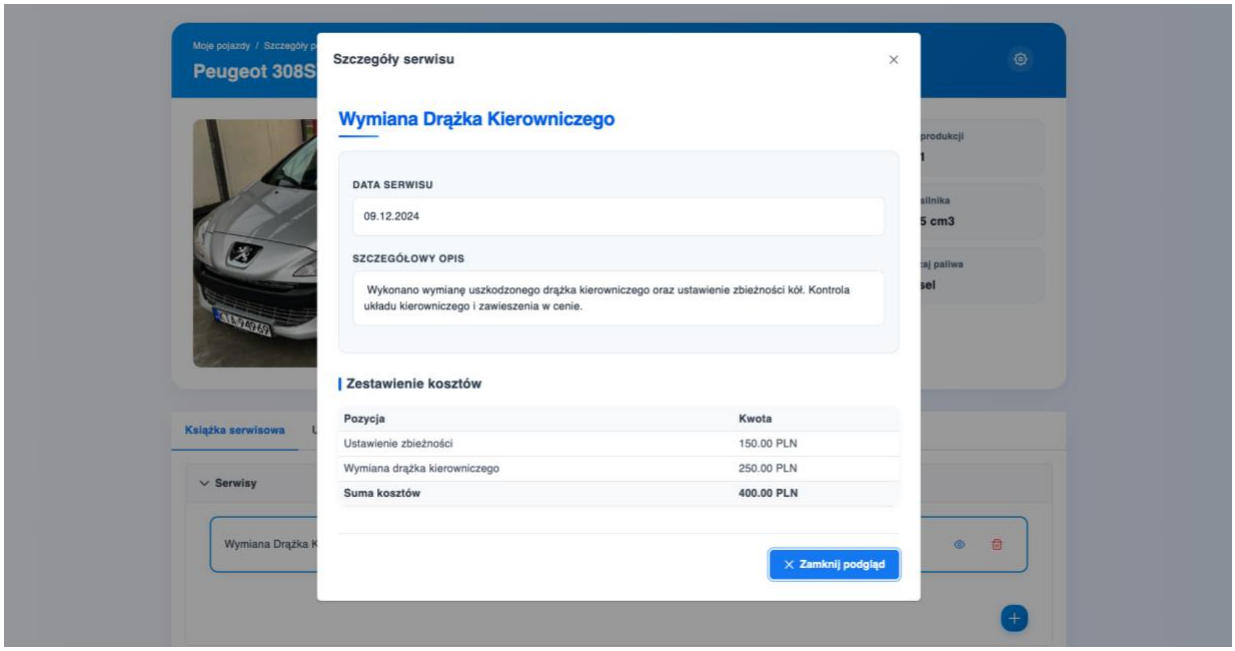
W tym widoku szczegółowym użytkownik ma dostęp do różnych funkcji poprzez interfejs. Klikając w ikonę koła zębatego w prawym górnym rogu, może wybrać jedną z dwóch opcji: edycję danych pojazdu lub jego usunięcie z systemu. Interfejs podzielony jest na zakładki, które umożliwiają dostęp do różnych informacji. W zakładce książki serwisowej, jak pokazano na rysunku 5.6, użytkownik może przeglądać historię serwisów i przeglądów pojazdu. W razie potrzeby ma możliwość usunięcia lub dodania nowego wpisu.



Rys. 5.6 - *Zakładka - Książka serwisowa [opracowanie własne]*

Po kliknięciu w ikonę oka przy wybranym wpisie, system wyświetla okno dialogowe prezentujące szczegółowe informacje o przeprowadzonym serwisie lub przeglądzie. Na rysunku 5.7 przedstawiono przykładowy wpis dotyczący *Wymiany drążka kierowniczego*. Okno dialogowe zawiera kompleksowe zestawienie informacji, takich jak data wykonania usługi, opis przeprowadzonej usługi, lista wymienionych części oraz koszty - zarówno robocizny, jak i całkowity wydatek poniesiony podczas wizyty w warsztacie.

Taki sposób dokumentacji pozwala użytkownikowi na precyzyjne monitorowanie historii napraw pojazdu i związanych z nimi wydatków. Jest to szczególnie pomocne przy planowaniu przyszłego budżetu na utrzymanie samochodu. Dodatkowo, szczegółowa historia serwisowa może służyć jako wiarygodne potwierdzenie regularnej konserwacji pojazdu podczas jego późniejszej sprzedaży.



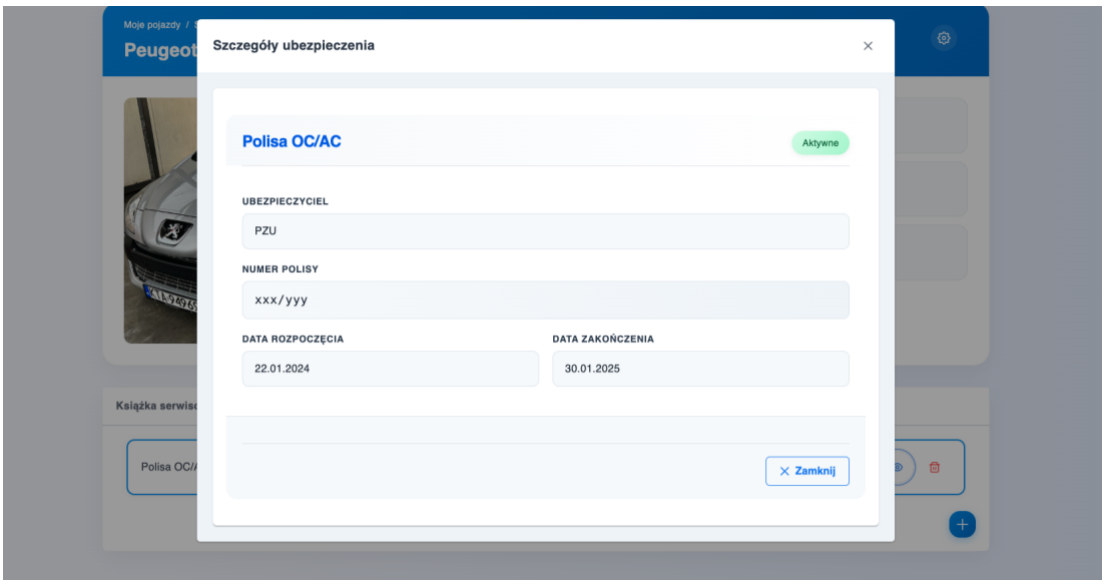
Rys. 5.7 - *Okno dialogowe - szczegóły serwisu [opracowanie własne]*

W zakładce *Ubezpieczenia* użytkownik ma dostęp do kompleksowego zestawienia wszystkich polis ubezpieczeniowych zarejestrowanych w systemie dla danego pojazdu. Interfejs umożliwia zarządzanie ubezpieczeniami poprzez dodawanie nowych wpisów, usuwanie nieaktualnych oraz przeglądanie szczegółowych informacji o każdej polisie.

Po kliknięciu w ikonę oka przy wybranym ubezpieczeniu, system wyświetla okno dialogowe, które zostało przedstawione na rysunku 5.8 zawierające szczegółowe dane, takie jak:

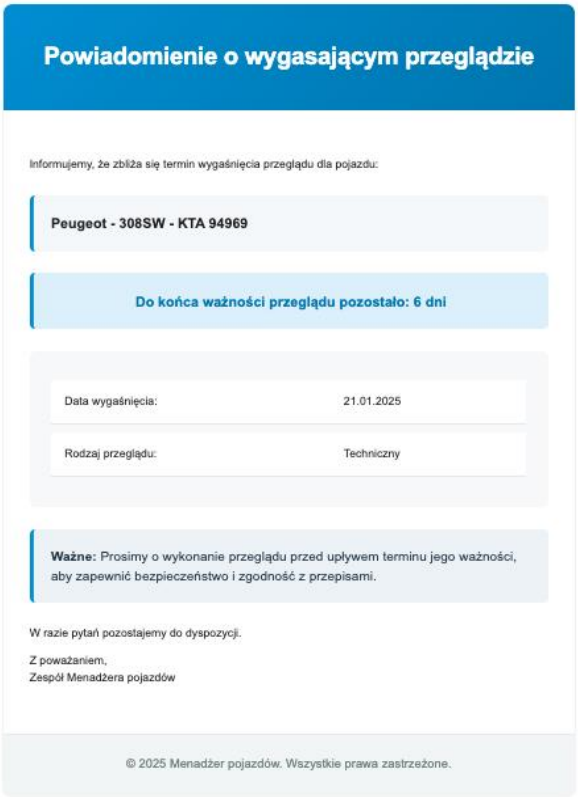
- Tytuł ubezpieczenia
- Szczegółowy opis zakresu ochrony
- Nazwę ubezpieczyciela
- Numer polisy
- Okres ważności (data rozpoczęcia i zakończenia ochrony ubezpieczeniowej)

Utrzymywanie danych w ten sposób pozwala na efektywne zarządzanie dokumentacją ubezpieczeniową pojazdu oraz pomaga w kontrolowaniu terminów ważności poszczególnych polis. Użytkownik może w łatwy sposób monitorować status ubezpieczeń i planować ich odnowienie z odpowiednim wyprzedzeniem.



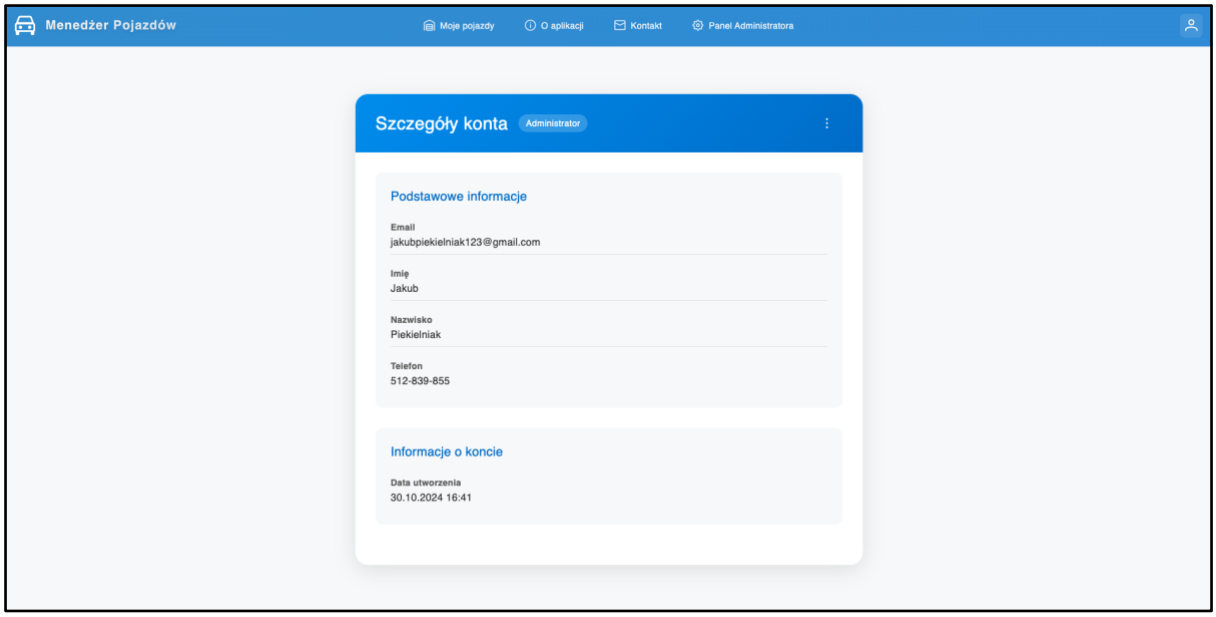
Rys. 5.8 - *Okno dialogowe - szczegóły ubezpieczenia [opracowanie własne]*

Wraz z końcem ważności przeglądu użytkownik otrzymuje stosowne powiadomienie w postaci wiadomości e-mail na adres wskazany podczas procesu *Rejestracji*. Wiadomość zawiera informację o zbliżającym się terminie wygaśnięcia przeglądu dla danego pojazdu. Przykładowy wygląd z informacjami został przedstawiony na rysunku 5.9.



Rys. 5.9 - *Powiadomienie E-mail [opracowanie własne]*

W sekcji *"Moje konto"* użytkownik ma dostęp do kompleksowego przeglądu i zarządzania swoim profilem. Interfejs prezentuje podstawowe informacje o koncie, takie jak adres e-mail, przypisaną rolę w systemie oraz datę dołączenia do platformy. System umożliwia również uzupełnienie opcjonalnych danych osobowych: imię, nazwisko i numer telefonu komórkowego. Przykładowy interfejs został przedstawiony na rysunku 5.10.



Rys. 5.10 - *Ekran - szczegóły konta [opracowanie własne]*

W ramach zarządzania kontem, użytkownik ma możliwość modyfikacji wprowadzonych danych oraz, w razie potrzeby, trwałego usunięcia swojego konta z systemu. Po wybraniu opcji usunięcia konta i potwierdzeniu tej decyzji, wszystkie dane użytkownika zostają trwale usunięte z systemu.

6. Testy

Testowanie oprogramowania stanowi fundamentalny element procesu wytwarzania systemów informatycznych, będąc kluczowym czynnikiem, który wpływa na jakość ostatecznego produktu. Pisanie testów nie zamyka się wyłącznie na weryfikacji poprawności działania poszczególnych możliwości, ale stanowi podejście do zapewnienia jakości tworzonego oprogramowania. Aplikacje często obsługują krytyczne procesy biznesowe gdzie nawet najdrobniejsze błędy, które wystąpiły, a nie zostały obsłużone podczas implementacji testów mogą prowadzić do dużych strat finansowych i reputacji danej organizacji.

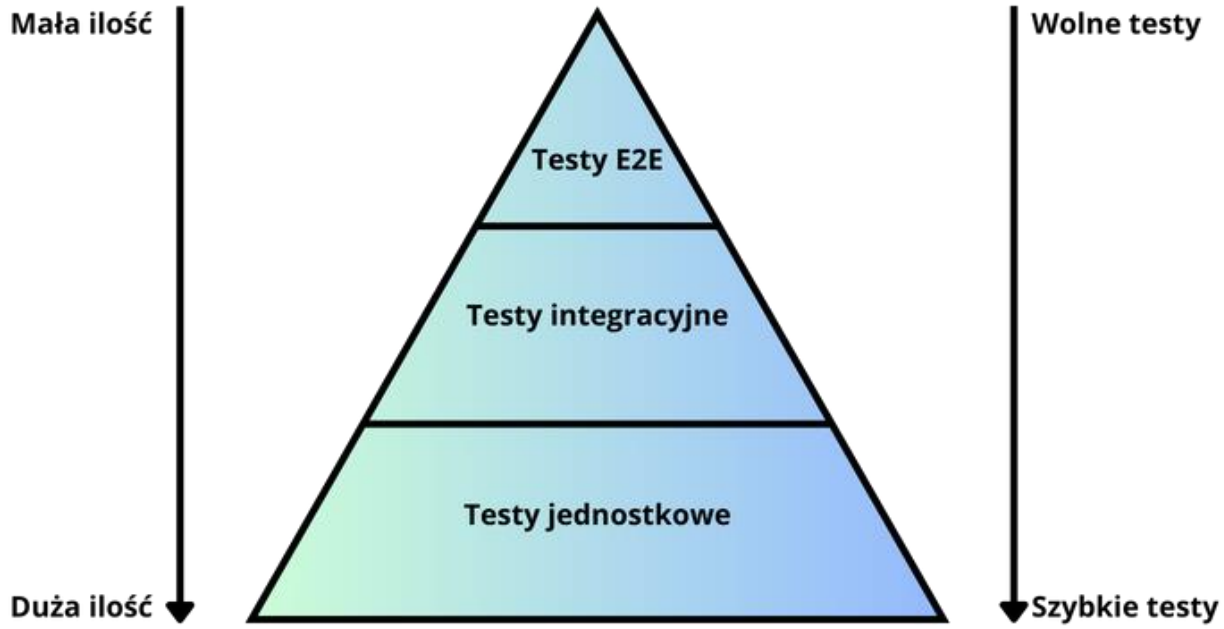
Pisząc testy systematycznie względem tworzonych możliwości zespół deweloperski może:

- Szybko wykrywać i eliminować błędy, redukując koszty naprawy w późniejszych etapach projektu.
- Utrzymywać wysokiej jakości kod.
- Zapewnić zgodność implementacji z założonymi wcześniej wymaganiami funkcjonalnymi i niefunkcjonalnymi.
- Zwiększyć stabilność działania systemu w środowisku produkcyjnym.

W ramach niniejszej pracy inżynierskiej, proces testowania skupia się na weryfikacji poprawności działania aplikacji webowej, która umożliwia zarządzania pojazdami. Główne cele testowania obejmują:

- Weryfikację poprawności implementacji logiki biznesowej.
- Sprawdzenie integralności danych w komunikacji między poszczególnymi komponentami.
- Zapewnienie odpowiedniej wydajności systemu pod względem przewidywanego obciążenia.

Strategia testowania w projekcie została oparta na podejściu wielopoziomowym, co pozwala na weryfikację systemu na różnych poziomach abstrakcji. Implementacja tej strategii koncentruje się na dwóch fundamentalnych poziomach testów: jednostkowych oraz integracyjnych, które stanowią podstawowe warstwy oraz większość piramidy testów zaprezentowanej na rysunku 6.1. Szczegółowa charakterystyka każdego z tych poziomów zostanie przedstawiona w kolejnych podrozdziałach.



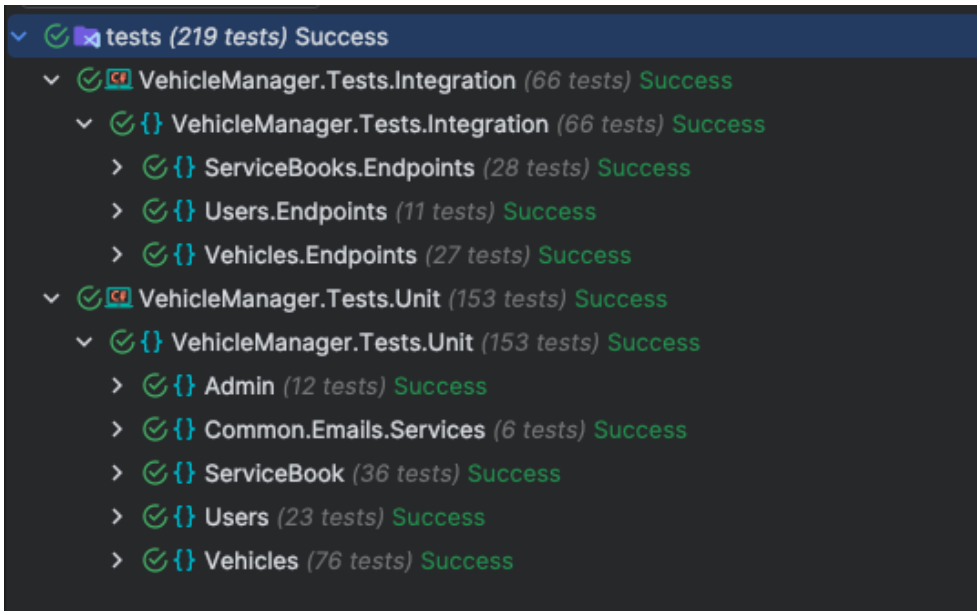
Rys. 6.1 - *Piramida testów [opracowanie własne]*

Jednym z kluczowych aspektów tworzenia wysokiej jakości testów jednostkowych jest zastosowanie podejścia AAA (ang. *Arrange-Act-Assert*). Ten powszechnie stosowany wzorzec zapewnia prostą i jednolitą strukturę oraz dzieli każdy przypadek testowy na trzy główne fazy:

1. **Arrange** (tłum. *Przygotuj*) - faza przygotowawcza, w której konfigurujemy warunki początkowe testu. Obejmuje tworzenie obiektów, inicjalizację danych testowych oraz konfigurację niezbędnych zależności.
2. **Act** (tłum. *Zrób*) - faza wykonawcza, podczas której następuje wywołanie testowanej funkcjonalności, najczęściej pojedynczej metody z wcześniej przygotowanymi danymi.
3. **Assert** (tłum. *Sprawdź*) - faza sprawdzająca, w która weryfikuje czy testowany kod zachował się zgodnie z oczekiwaniami. Może to obejmować sprawdzenie wartości zwracanej, nowego stanu obiektów lub weryfikację interakcji między komponentami.

Wzorzec AAA zwiększa czytelność testów i zmniejsza koszty utrzymania całego zestawu poprzez jasne rozgraniczenie odpowiedzialności poszczególnych sekcji. Zaleca się również stosowanie pustych linii między fazami testu, co dodatkowo poprawia jego wizualną przejrzystość [2].

W trakcie projektowania warstwy serwerowej zaimplementowano łącznie 219 testów, w tym 153 testy jednostkowe oraz 66 testów integracyjnych. Taki podział testów spełnia wymagania piramidy testów przedstawionej na rysunku 6.1, zgodnie z którą liczba testów jednostkowych powinna być znacznie większa niż testów integracyjnych. Rysunek 6.2 prezentuje szczegółowe informacje na temat zaimplementowanych testów w aplikacji, potwierdzając ich prawidłowy podział oraz fakt, że wszystkie testy kończą się pozytywnym wynikiem.



Rys. 6.2 - *Testy aplikacji [opracowanie własne]*

Rysunek 6.3 podsumowuje wynik pokrycia testami na poziomie 89%, co świadczy o solidnym przetestowaniu aplikacji. Poszczególne moduły osiągają pokrycie od 80% do 86%, potwierdzając dokładne sprawdzenie poprawności działania kluczowych elementów systemu.

Symbol	Coverage (%)	Uncovered/Total Stmts.
Total	89%	645/5752
src	83%	639/3779
VehicleManager.Api	86%	63/435
VehicleManager.Application	80%	88/451
VehicleManager.Core	83%	84/497
VehicleManager.Infrastructure	83%	404/2396
tests	99%	6/1973
VehicleManager.Tests.Integration	100%	0/652
VehicleManager.Tests.Unit	99%	6/1321

Rys. 6.3 – *Pokrycie testów [opracowanie własne]*

6.1. Testy jednostkowe

Testy jednostkowe (ang. *Unit Tests*) to wyspecjalizowane fragmenty kodu służące do weryfikacji poprawności działania pojedynczych komponentów programu w środowisku izolowanym. Proces implementacji rozpoczyna się od podziału aplikacji na mniejsze, niezależne elementy, które można łatwo testować. Kluczowym aspektem jest zapewnienie pełnej izolacji testów - powinny one działać niezależnie od siebie, a ich kolejność wykonywania nie może wpływać na wyniki [1].

Rysunek 6.4 przedstawia przykład testu jednostkowego wykorzystującego AAA do weryfikacji poprawności działania wzorca *Builder* obiektu *Vehicle*. Test został zaimplementowany z wykorzystaniem biblioteki *xUnit* oraz jej mechanizmu teorii (ang. *Theory*), który umożliwia przeprowadzenie tego samego testu dla wielu przypadków danych.

```
[Theory]
[ClassData(typeof(VehicleBuilderValidTestData))]
public void given_valid_data_to_builder_should_build_vehicle_success(
    VehicleBuilderParams vehicleBuilderParams
)
{
    //arrange & act
    var vehicle = Act(vehicleBuilderParams);

    //assert
    vehicle.ShouldNotBeNull();
    vehicle.Id.ShouldNotBeSameAs(Guid.Empty);
    vehicle.ServiceBook.ShouldNotBeNull();
}
```

Rys. 6.4 - *Vehicle Builder - test jednostkowy [opracowanie własne]*

Atrybut *[ClassData]* wskazuje na klasę *VehicleBuilderValidTestData*, dziedziczącą po generycznej klasie *TheoryData<T>*, która dostarcza zestawy danych testowych przy użyciu biblioteki do generowania danych *Bogus* zgodnie z implementacją przedstawioną na rysunku 6.5. W implementacji zastosowano również konwencję nazewniczą testów *given_when_then*, która jasno określa kontekst testu, wykonywane działanie oraz oczekiwany rezultat.

```
internal class VehicleBuilderValidTestData : TheoryData<VehicleBuilderParams>
{
    private readonly Faker _faker = new();

    public VehicleBuilderValidTestData()
    {
        Add(new VehicleBuilderParams
        {
            Brand = _faker.Vehicle.Manufacturer(),
            Model = _faker.Vehicle.Model(),
            Year = _faker.Date.Past().Year,
            EngineCapacity = _faker.Random.Int(1000, 5000),
            EnginePower = _faker.Random.Int(50, 500),
            LicensePlate = GenerateLicensePlate(),
            VIN = _faker.Vehicle.Vin(),
            FuelType = FuelType.Diesel,
            GearboxType = GearboxType.Automatic,
            VehicleType = VehicleType.Car
        });
    }
}
```

Rys. 6.5 - *Implementacja klasy VehicleBuilderValidTestData [opracowanie własne]*

W przypadku testów jednostkowych szczególnie istotną kwestią jest zapewnienie izolacji testowanego komponentu od jego zewnętrznych zależności. W projekcie wykorzystano techniki mockowania przy użyciu biblioteki *NSubstitute* do symulowania zachowania zależności. Na rysunku 6.6 przedstawiono konfigurację zależności dla handlera korzystającego ze wzorca *CQRS*, gdzie zasymulowano wymagane repozytoria oraz kontekst.

```
private readonly IRequestHandler<CreateVehicleCommand, CreateVehicleResponse> _handler;
private readonly IVehicleRepository _vehicleRepository;
private readonly IUserRepository _userRepository;
private readonly IContext _context;
private readonly VehicleTestFactory _factory = new();

public CreateVehicleCommandHandlerTests()
{
    _vehicleRepository = Substitute.For<IVehicleRepository>();
    _userRepository = Substitute.For<IUserRepository>();
    _context = Substitute.For<IContext>();

    _handler = new CreateVehicleCommandHandler(
        _vehicleRepository,
        _userRepository,
        _context
    );
}
```

Rys 6.6 - *Mockowanie zależności [opracowanie własne]*

```
[Fact]
public async Task given_valid_data_should_create_vehicle_success()
{
    // Arrange
    var command = _factory.CreateVehicleCommand();
    var userId = Guid.NewGuid();
    _vehicleRepository
        .ExistsAsync(Arg.Any<Expression<Func<Vehicle, bool>>>(), Arg.Any<CancellationToken>())
        .Returns(false);
    _userRepository.AnyAsync(Arg.Any<Expression<Func<User, bool>>>(), Arg.Any<CancellationToken>())
        .Returns(true);
    _context.Id.Returns(userId);

    // Act
    var response = await Act(command);

    // Assert
    response.ShouldNotBeNull();
    response.VehicleId.ShouldNotBeSameAs(Guid.Empty);
    response.ShouldBeOfType<CreateVehicleResponse>();
    await _vehicleRepository.Received(1).AddAsync(Arg.Any<Vehicle>(), Arg.Any<CancellationToken>());
    await _vehicleRepository.Received(1).SaveChangesAsync(Arg.Any<CancellationToken>());
}
```

Rys. 6.7 - *Test jednostkowy handlera [opracowanie własne]*

Rysunek 6.7 prezentuje implementację testu jednostkowego dla handlera komendy CreateVehicle. Test sprawdza poprawność tworzenia pojazdu przy prawidłowych danych wejściowych oraz weryfikuje liczbę wywołań poszczególnych zależności, korzystając z wcześniej skonfigurowanych atrap.

6.2. Testy integracyjne

Testy integracyjne (ang. *Integration Tests*) skupiają się na weryfikacji współdziałania wszystkich komponentów systemu w przeciwieństwie do testów jednostkowych, które sprawdzają komponenty w izolacji. W kontekście aplikacji webowych testy te obejmują pełny przepływ żądania, uwzględniając wszystkie elementy stosu aplikacji, takie jak kontrolery, filtry czy procedury obsługi komunikatów. Pozwalają one na weryfikację poprawności działania całego systemu w warunkach zbliżonych do produkcyjnych, sprawdzając komunikację między poszczególnymi modułami oraz ich integrację [1].

Podczas implementacji testów integracyjnych kluczowe jest odizolowanie środowiska testowego od produkcyjnego. Wymaga to zastąpienia komponentów systemu ich testowymi odpowiednikami. Przykładem jest konfiguracja bazy danych przedstawiona na rysunku 6.8, gdzie klasa *VehicleManagerTestFactory* nadpisuje domyślną konfigurację kontekstu bazy danych. Proces rozpoczyna się od pobrania konfiguracji z pliku *appsettings.Test.json*.

Następnie, w metodzie *ConfigureWebHost*, oryginalny deskryptor kontekstu *VehicleManagerDbContext* jest zastępowany nowym, wykorzystującym unikalną nazwę bazy danych generowaną przez Guid. Takie podejście zapewnia, że każde wykonanie testów korzysta z osobnej, izolowanej bazy danych, eliminując ryzyko konfliktów i zapewnia czystość danych testowych bez ingerencji w środowisko produkcyjne.

```
public class VehicleManagerTestFactory : WebApplicationFactory<Api.Program>
{
    private const string ConnectionStringSection = "VehicleManagerTest";

    private readonly IConfiguration _configuration = new ConfigurationBuilder()
        .AddJsonFile("appsettings.Test.json")
        .Build();

    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.ConfigureTestServices(services =>
        {
            var descriptor = services
            .SingleOrDefault(x => x.ServiceType == typeof(DbContextOptions<VehicleManagerDbContext>));

            if (descriptor is not null)
            {
                services.Remove(descriptor);
            }

            var connectionString = _configuration.GetConnectionString(ConnectionStringSection);
            var dbName = $"vehicle_manager_test_{FastGuid.NewGuid()}";
            services.AddDbContext<VehicleManagerDbContext>(options =>
            {
                options.UseNpgsql(connectionString?.Replace("vehicle_manager_test", dbName))
                    .EnableSensitiveDataLogging(false)
                    .UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);
            });
        });
    }
}
```

Rys. 6.8 - *Konfiguracja testowej bazy danych [opracowanie własne]*

W systemie testy integracyjne zostały podzielone na dwie główne kategorie ze względu na wymagania dotyczące uwierzytelniania użytkownika. Pierwsza kategoria obejmuje testy endpointów, które nie wymagają autoryzacji - przykładowo rejestracja czy logowanie użytkownika. Druga kategoria dotyczy zasobów chronionych, gdzie konieczne jest wcześniejsze uwierzytelnienie użytkownika poprzez token JWT.

Na rysunku 6.9 przedstawiono przykład testu integracyjnego dla endpointu logowania użytkownika, który należy do kategorii niewymagającej uwierzytelnienia. Test weryfikuje scenariusz poprawnego logowania do systemu. W fazie początkowej przygotowywane są dane testowe oraz dodawany jest użytkownik do bazy danych. W fazie działania wykonywane jest żądanie HTTP POST do endpointu */users/sign-in* z danymi logowania. W ostatniej fazie

weryfikowane jest, czy serwer zwrócił kod odpowiedzi *200 OK* oraz czy odpowiedź zawiera oczekiwany format danych w postaci obiektu *SignInResponse*. Test potwierdza poprawność procesu logowania dla właściwych danych uwierzytelniających.

```
[Fact]
public async Task post_sign_in_with_valid_data_should_return_200_status_code()
{
    //arrange
    var command = _factory.CreateSignInCommand();
    var hashedPassword = PasswordHasher.HashPassword(command.Password);
    var user = _factory.CreateUser(command.Email, hashedPassword);
    await SeedDataAsync(user);

    //act
    var response = await Client.PostAsJsonAsync(UserEndpoints.SignIn, command);

    //assert
    response.StatusCode.ShouldBe(HttpStatusCode.OK);
    await response.Content.ReadFromJsonAsync<SignInResponse>().ShouldNotBeNull();
}
```

Rys. 6.9 - *Test integracyjny logowania [opracowanie własne]*

Aby przetestować endpointy zaliczane do drugiej kategorii, wymagającej uwierzytelnienia należy skorzystać z metody *Authorize* przedstawionej na rysunku 6.10. Metoda przyjmuje dwa argumenty: identyfikator użytkownika (*userId*) oraz rolę (*role*), na podstawie których generuje token JWT. Wygenerowany token jest następnie dodawany do nagłówka *Authorization* klienta HTTP, co pozwala na wykonywanie autoryzowanych żądań podczas testów.

```
protected void Authorize(Guid userId, string role)
{
  var jwt = _authManager.GenerateToken(userId, role);
  Client.DefaultRequestHeaders.Authorization =
    new AuthenticationHeaderValue("Bearer", jwt.AccessToken);
}
```

Rys. 6.10 - *Metoda Authorize [opracowanie własne]*

Rysunek 6.11 przedstawia przykład testu integracyjnego dla endpointu tworzenia pojazdu, należącego do kategorii wymagającej uwierzytelnienia. W fazie początkowej przygotowywane są dane testowe oraz użytkownik, który jest dodawany do bazy danych. Następnie realizowane jest uwierzytelnienie poprzez metodę *Authorize*. W fazie działania system wysyła żądanie HTTP POST do endpointu */vehicles* z danymi pojazdu oraz odpowiednimi nagłówkami. W fazie końcowej weryfikowane jest, czy serwer zwrócił kod odpowiedzi *201 Created* oraz czy odpowiedź zawiera nagłówek *Location* wskazujący na utworzony zasób.

```
[Fact]
public async Task post_create_vehicle_with_valid_data_should_return_201_status_code()
{
    //arrange
    var command = _factory.CreateVehicleCommand();
    var user = _factory.CreateUser();
    await SeedDataAsync(user);
    Authorize(user.Id, user.Role.ToString());

    //act
    var response = await Client.PostAsJsonAsync(VehicleEndpoints.BasePath, command);

    //assert
    response.StatusCode.ShouldBe(HttpStatusCode.Created);
    response.Headers.Location.ShouldNotBeNull();
}
```

Rys. 6.11 - *Test integracyjny dodania pojazdu [opracowanie własne]*

7. Podsumowanie i wnioski

Zgodnie z założeniami pracy dyplomowej zrealizowano aplikację webową służącą do kompleksowego zarządzania pojazdami przez użytkowników końcowych. W toku implementacji projektu pomyślnie wdrożono wszystkie wymagania funkcjonalne i niefunkcjonalne, szczegółowo opisane w rozdziale drugim.

Realizacja została oparta o nowoczesny stos technologiczny przedstawiony w rozdziale trzecim, wykorzystujący framework .NET 8.0 wraz z językiem programowania C# po stronie serwerowej oraz Angular po stronie klienckiej. Zastosowanie tych technologii, wraz z bazą danych PostgreSQL oraz usługami Microsoft Azure, zapewniło wysoką wydajność systemu oraz intuicyjny interfejs użytkownika.

Wybrane technologie nie tylko reprezentują aktualne trendy w rozwoju aplikacji webowych, ale także gwarantują stabilność i skalowalność rozwiązania. Szczególne znaczenie ma wykorzystanie platformy Azure, która zapewnia niezawodną infrastrukturę chmurową dla aplikacji.

Potencjalne kierunki rozwoju systemu mogą koncentrować się na:

- implementacji powiadomień SMS o zbliżającym się terminie wygaśnięcia ubezpieczenia;
- wprowadzeniu możliwości generowania raportów z książki serwisowej i historii ubezpieczeń;
- integracji z Google Maps API umożliwiając lokalizację najbliższych warsztatów samochodowych;
- rozszerzeniu możliwości uwierzytelniania o popularne serwisy społecznościowe.

Reasumując, zaimplementowane rozwiązanie spełnia wszystkie założone wymagania i jest gotowe do użytku produkcyjnego. Aplikacja oferuje kompleksowe możliwości zarządzania pojazdami, prezentowane w przejrzystym interfejsie graficznym. Projekt pozwolił poszerzyć wiedzę w zakresie tworzenia nowoczesnych aplikacji webowych, ze szczególnym uwzględnieniem testów oraz narzędzi udostępnionych przez firmę Microsoft. Praktyczne doświadczenie w implementacji wzorców projektowych i zasad czystego kodu stanowi cenną wartość dodaną zrealizowanej pracy inżynierskiej.

Bibliografia

[1] Glenn Block, Pablo Cibraro, Pedro Felix, Howard Dierking, Darrel Miller. *Nowoczesne API. Ewoluujące aplikacje sieciowe w technologi ASP.NET*

[2] Vladimir Khorikov *Testy jednostkowe. Zasady, praktyki i wzorce*

[3] Robert C. Martin *Czysta architektura. Struktura i design oprogramowania. Przewodnik dla profesjonalistów*

[4] Eric A. Meyer, Estelle Weyl. *CSS: The Definitive Guide 5th Edition Web Layout and Presentation*

[5] Mark Michaelis *Essential C# 8.0*

[6] Angular [online] - dostęp 6.11.2024
https://angular.dev/overview

[7] ASP. NET Core [online] - dostęp 5.11.2024
https://learn.microsoft.com/pl-pl/aspnet/core/introduction-to-aspnet-core

[8] CRUD [online] - dostęp 4.11.2024
https://codenga.pl/strony/poradniki/crud_podstawowe_operacje_na_bazie_danych

[9] DB-Engines Ranking - Trend Popularity [online] - dostęp 5.11.2024
https://db-engines.com/en/ranking_trend

[10] Entity Framework Core [online] - dostęp 5.11.2024
https://learn.microsoft.com/en-us/ef/core/

[11] JSON Web Token [online] - dostęp 6.11.2024
https://jwt.io/introduction

[12] Microsoft Azure [online] - dostęp 11.11.2024
https://learn.microsoft.com/pl-pl/azure/cloud-adoption-framework/get-started/what-is-azure

[13] PostgreSQL [online] - dostęp 5.11.2024
https://www.postgresql.org/about/

[14] Swagger [online] - dostęp 6.11.2024
https://learn.microsoft.com/pl-pl/aspnet/core/tutorials/web-api-help-pages-using-swagger

[15] Wzorzec CQRS [online] - dostęp 12.11.2024
https://learn.microsoft.com/pl-pl/azure/architecture/patterns/cqrs

[16] xUnit [online] - dostęp 6.11.2024
https://xunit.net/#documentation

[17] .NET [online] - dostęp 5.11.2024
https://learn.microsoft.com/pl-pl/dotnet/core/introduction

Spis rysunków