AKADEMIA TARNOWSKA

Wydział Politechniczny

Kierunek: ***Informatyka***

2024/2025

Justyna Kurcz

PRACA INŻYNIERSKA

Projekt i implementacja aplikacji webowej wspomagającej opiekę nad zwierzętami domowymi

Promotor pracy:

mgr inż. Tomasz Gądek

Tarnów, *2025*

Spis treści

1. Wstęp

W dzisiejszych czasach aplikacje internetowe stają się nieodłącznym elementem życia codziennego, zyskując na coraz większej popularności. Oferują one innowacyjne rozwiązania w wielu dziedzinach, otwierając nowe możliwości dla użytkowników. Szczególne zastosowanie znajdują w obszarze wspomagania opieki nad zwierzętami domowymi, które stają się coraz ważniejszą częścią rodzin.

1.1. Motywacja

Motywacją do zrealizowania aplikacji webowej wspomagającej opiekę nad zwierzętami domowymi była przede wszystkim osobista obserwacja problemów, z jakimi borykają się właściciele zwierząt w zakresie organizacji i przechowywania informacji o swoich pupilach. Zazwyczaj magazynują oni te informacje w różnych miejscach - część w papierowej książeczce zdrowia, część w notatkach w telefonie, a jeszcze inne w formie luźnych dokumentów czy zdjęć. Taka fragmentacja danych nie tylko utrudnia szybki dostęp do potrzebnych informacji, ale także zwiększa ryzyko ich utraty.

Istotnym czynnikiem motywującym było również przekonanie, że cyfrowa forma przechowywania informacji o pupilu może znacząco ułatwić użytkownikom aplikacji dostęp do historii leczenia podczas wizyt u różnych lekarzy weterynarii oraz innych specjalistów zajmujących się opieką nad zwierzętami. Posiadanie przez właściciela szybkiego dostępu do pełnej historii medycznej pupila może być kluczowe w sytuacjach wymagających nagłej interwencji weterynaryjnej lub podczas rutynowych wizyt kontrolnych, szczególnie gdy korzystamy z usług różnych placówek.

1.2. Cel pracy

Celem niniejszej pracy dyplomowej jest zaprojektowanie i zaimplementowanie aplikacji webowej, która usprawnia proces wspomagania opieki nad zwierzętami domowymi. Głównym założeniem jest umożliwienie przechowywania dokumentacji medycznej oraz historii wizyt weterynaryjnych w scentralizowanym miejscu, pozwalając na efektywne gromadzenie i zarządzanie informacjami o zwierzętach w jednym, łatwo dostępnym systemie.

1.3. Zakres pracy

Zakres pracy obejmuje kompletny cykl rozwoju oprogramowania, który został zrealizowany poprzez szereg kluczowych etapów, pozwalających na systematyczne i uporządkowane budowanie aplikacji. W ramach pracy wykonano następujące etapy:

1. **Analiza biznesowa wymagań użytkownika** - etap ten obejmował analizę potrzeb potencjalnych użytkowników systemu oraz zdefiniowanie głównych możliwości systemu. Określono wymagania dotyczące interfejsu użytkownika oraz sposobu prezentacji danych. Na podstawie zebranych informacji opracowano szczegółową specyfikację wymagań funkcjonalnych i niefunkcjonalnych.
2. **Dobór odpowiednich technologii** - etap ten polegał na wyborze nowoczesnych narzędzi programistycznych. W projekcie użyto technologii .NET do zbudowania aplikacji serwerowej. Do zaprojektowania interfejsu użytkownika posłużył framework Angular. Dane są przechowywane w bazie PostgreSQL oraz usłudze Blob Storage na platformie Azure. Wykorzystanie tych technologii umożliwiło zaimplementowanie wydajnej i łatwo skalowalnej aplikacji.
3. **Implementacja systemu** - w ramach części praktycznej pracy zaimplementowano aplikację serwerową w technologii .NET, zawierającą logikę biznesową oraz obsługę bazy danych. Równolegle powstawał interfejs użytkownika w technologii Angular. Końcowym etapem była integracja obu elementów systemu w jedną, spójną całość.
4. **Testowanie oprogramowania** - końcowy etap pracy obejmował przeprowadzenie testów jednostkowych dla poszczególnych komponentów systemu oraz testów integracyjnych weryfikujących poprawną współpracę wszystkich elementów aplikacji. Przeprowadzone testy umożliwiły wykrycie i eliminację potencjalnych błędów oraz zapewnienie wysokiej jakości końcowego oprogramowania.

Realizacja wszystkich powyższych etapów pozwoliła na zbudowanie kompleksowego rozwiązania, które w pełni odpowiada na potrzeby użytkowników końcowych w zakresie wspomagania opieki nad zwierzętami domowymi.

2. Analiza biznesowa

W niniejszym rozdziale zostaną omówione kluczowe założenia dotyczące możliwości projektowanego systemu. Ważnym aspektem tworzenia aplikacji jest podział na wymagania funkcjonalne i niefunkcjonalne. Aby aplikacja mogła działać w pełni efektywnie, powinna spełniać oba te rodzaje wymagań, wpływając na jakość jej działania.

2.1. Wymagania funkcjonalne

System powinien zapewniać użytkownikom szereg możliwości, które wspierają ich w opiece nad zwierzętami domowymi. Użytkownik powinien mieć możliwość:

- Rejestracji i logowania do systemu przy użyciu adresu e-mail oraz hasła, zapewniając ochronę danych osobowych.
- Dodawania zwierząt oraz edytowania ich danych, takich jak imię, rasa, data urodzenia i inne istotne informacje.
- Wprowadzania informacji o wizytach, w tym terminów, diagnoz oraz dodatkowych uwag.
- Dodawania i przeglądania informacji o szczepieniach, takich jak data szczepienia oraz nazwa podanej szczepionki.
- Otrzymywania powiadomień e-mail o zbliżających się terminach szczepień oraz wizytach weterynaryjnych.
- Przeglądania listy swoich zwierząt wraz z pełną historią zdrowotną każdego z nich, obejmującą wizyty oraz szczepienia.
- Usuwania wprowadzonych danych o zwierzętach wraz z ich historią zdrowotną, co ułatwia utrzymanie porządku w systemie.
- Trwałego usunięcia swojego konta z systemu wraz z wszystkimi powiązanymi danymi.

2.2. Wymagania niefunkcjonalne

W tej części pracy zostaną przedstawione aspekty systemu, które wpływają na jego jakość działania, stabilność oraz komfort użytkowania. Nie odnoszą się one bezpośrednio do konkretnych zadań aplikacji. Kluczowe wymagania niefunkcjonalne obejmują:

2.2.1. Bezpieczeństwo

Ochrona danych osobowych, w tym wrażliwych informacji oraz danych dotyczących zwierząt użytkowników, jest priorytetem w projektowaniu aplikacji. System powinien wykorzystywać mechanizm autoryzacji i uwierzytelniania oparty na technologii JWT (ang. *JSON Web Token*). Dzięki temu eliminujemy konieczność ciągłego wprowadzania danych logowania, przy jednoczesnym zachowaniu wysokiego poziomu bezpieczeństwa. Dodatkowo hasła użytkowników powinny być przechowywane w bezpieczny sposób, stosując zaawansowane algorytmy haszujące, co pozwala zapobiegać przechowywaniu haseł w formie jawnej.

2.2.2. Wydajność

Aplikacja powinna działać płynnie, bez względu na liczbę obsługiwanych użytkowników i ilość przechowywanych danych. Wszelkie zapytania od klienta do serwera należy sprawnie przetwarzać, aby czas oczekiwania na rezultat nie był znacząco wydłużony. Dzięki temu zapewni to sprawny przepływ informacji.

2.2.3. Intuicyjny interfejs użytkownika

Interfejs aplikacji powinien być zaprojektowany w sposób umożliwiający łatwe i intuicyjne poruszanie się po systemie, nawet dla nowych użytkowników. Formularze muszą być czytelne i proste w obsłudze, a każdy etap wprowadzania danych ma być zrozumiały dla użytkownika. Intuicyjne rozmieszczenie elementów nawigacyjnych, wyraźne etykiety oraz zrozumiałe wskazówki zagwarantują użytkownikom komfortowe korzystanie z aplikacji.

3. Stos technologiczny

W tym rozdziale przedstawiony zostanie stos technologii oparty na nowoczesnych narzędziach odpowiednio dobranych do potrzeb współczesnych aplikacji.

3.1. Aplikacja serwerowa

Warstwa serwerowa (ang. *backend*) została zaimplementowana jako interfejs webowy (ang. *Web Application Programming Interfaces*), który umożliwia dostęp poprzez podstawowe metody oraz nagłówki HTTP (ang. *Hypertext Transfer Protocol*) [1]. Aplikacja opiera się głównie na operacjach CRUD (ang. *create, read, update, delete*), czyli czterech podstawowych metodach do zarządzania zasobami przechowywanymi w bazie danych [5].

3.1.1. C#

Nowoczesny, zorientowany obiektowo język programowania, który został opracowany przez firmę Microsoft pod kierownictwem Andersa Hejlsberga. Jak sama nazwa wskazuje, pochodzi z rodziny C i C++, choć wykorzystuje także najlepsze cechy innych języków programowania takich jak Java [6]. Ważną zaletą tego języka jest LINQ (ang. *Language Integrated Query*), pozwala na wykonywanie zapytań na danych, przy użyciu składni przypominającej język SQL [4].

3.1.2. ASP.NET

Platforma typu open-source opracowana przez firmę Microsoft. Umożliwia tworzenie różnorodnych aplikacji - od stron internetowych, usług po aplikacje mobilne i systemy IoT (ang. *Internet of things*). Pozwala na uruchamianie aplikacji w środowiskach innych niż Windows np. MacOS oraz Linux. Umożliwia tworzenie aplikacji w dowolnym języku programowania [11]. Według przeprowadzonych badań odnośnie popularności frameworków webowych cieszy się dużą popularnością zajmując czołowe miejsce [17].

3.1.3. Entity Framework Core

Lekka, rozszerzalna biblioteka z otwartym dostępem do kodu opracowana przez Microsoft. Jest wieloplatformowa, więc działa na różnych systemach operacyjnych. Obsługuje wiele baz danych, takich jak PostgreSQL i MySQL. Służy jako narzędzie mapujące obiektowo-relacyjne (O/RM). Umożliwia automatyczne konwertowanie danych między obiektami aplikacji a bazą danych. Dostęp do danych opiera się za pomocą modelu, który reprezentuje strukturę aplikacji. Model ten składa się z klas jednostek, odzwierciedlających tabele w bazie danych oraz obiektu kontekstu, który umożliwia połączenie z bazą, wykonywanie zapytań czy zapisywanie danych. Funkcja migracji upraszcza tworzenie aplikacji poprzez automatyczne generowanie i aktualizację struktur baz danych w przypadku zmiany modeli [13].

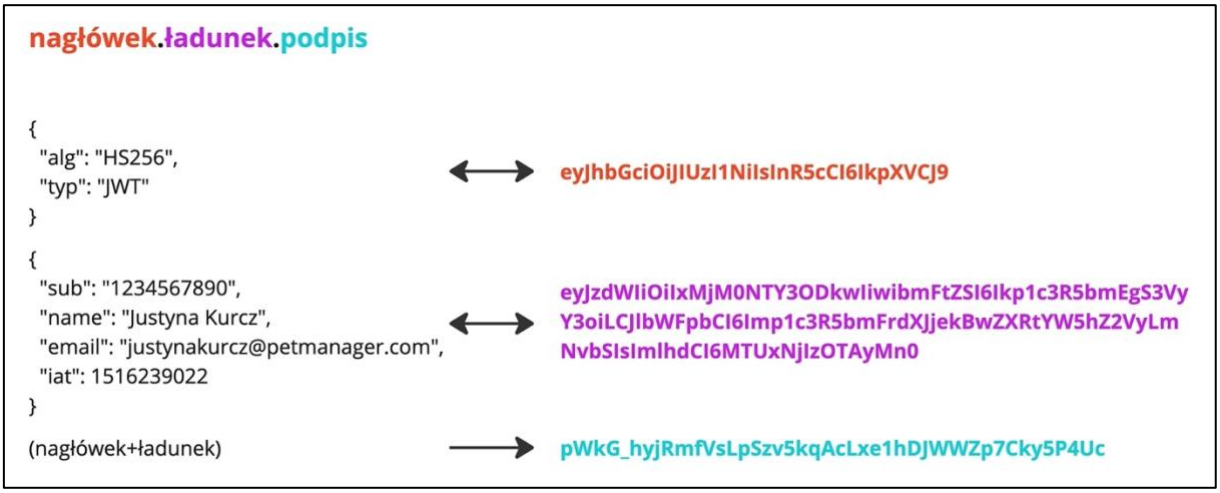
3.1.4. JSON Web Token

JSON Web Token to standard, który określa sposób przekazywania informacji między stronami jako obiekt JSON (ang. *JavaScript Object Notation*). Przesyłane dane mogą zostać zweryfikowane, ponieważ są podpisane cyfrowo. JWT jest najczęściej używany do uwierzytelniania użytkowników, ale służy również do bezpiecznej wymiany informacji między stronami.

Struktura tokenu składa się z trzech części oddzielonych kropkami:

- **Nagłówek** - typ tokenu oraz zastosowany algorytm podpisu
- **Ładunek** - roszczenia, czyli oświadczenia o jednostce
- **Podpis** - umożliwia weryfikację czy token nie został zmodyfikowany [14]

Przykład tokenu wraz z dekodowaniem został przedstawiony rysunku 3.1.



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

3.1.5. xUnit

Bezpłatne narzędzie do testowania kodu na platformie .Net. Framework umożliwia oznaczenie metod testowych za pomocą atrybutów *[Fact]* lub *[Theory]*, aby wskazać, że dany kod jest testem. Pierwszy sygnalizuje, że funkcja nie może przyjąć żadnych parametrów wejściowych. Z kolei drugi atrybut umożliwia testowanie metody na różnych zestawach danych, dzięki czemu można zobaczyć różne scenariusze działania kodu. Przypadki testowe można przekazywać przy użyciu atrybutów *[InlineData]* lub *[ClassData]* [19].

3.1.6. Swagger

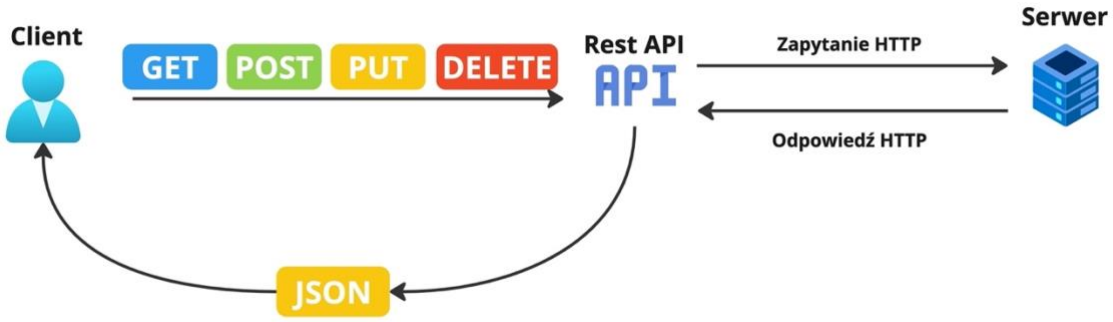
Swagger czyli specyfikacja OpenAPI to zestaw narzędzi typu open source wspierający pracę z interfejsami REST API. Pozwala łatwo projektować, budować, dokumentować i testować interfejsy API [12]. Dokumentacja generuje się automatycznie na podstawie kontrolerów, punktów końcowych oraz adnotacji w kodzie. Swagger UI pozwala przeglądać i testować metody API bezpośrednio w przeglądarce. Narzędzie to zapewnia, że dokumentacja API jest zawsze aktualna, co minimalizuje konieczność ręcznego jej tworzenia lub aktualizowania [16].

3.1.7. Azure

Microsoft Azure to zaawansowana platforma chmurowa firmy Microsoft oferująca usługi w modelu PaaS (ang. *Platform as a Service*). Działa w oparciu o technologię wirtualizacji i może być obsługiwana na całym świecie przez serwery Microsoft zlokalizowane w globalnych centrach danych, eliminując potrzebę posiadania własnej infrastruktury. Platforma Azure zapewnia szeroką gamę aplikacji, od przechowywania danych i zarządzania infrastrukturą IT po tworzenie aplikacji i usług obliczeniowych. Udostępnia wiele możliwości, takich jak przetwarzanie danych, przechowywanie i wgrywanie oprogramowania na wersję produkcyjną. Ponadto firma Microsoft posiada zaawansowane zabezpieczenia w celu ochrony danych, tożsamości i zasobów [2].

3.1.8. REST API

REST (ang. *Representational State Transfer*) to styl architektury oprogramowania oparty na określonym zbiorze reguł opisujących sposób definiowania zasobów i zapewniający dostęp do nich. Został wprowadzony przez Roya Fieldinga. Styl REST wykorzystuje protokół HTTP do realizacji większości architektonicznych ograniczeń, jakie definiuje. Mówiąc o REST-owym API zakładamy, że spełnia następujące kryteria: oddzielenie interfejsu użytkownika od działań wykonywanych po stronie serwera, bezstanowość, wykorzystanie mechanizmu cache (buforowania), punkty końcowe powinny jawnie wskazywać do jakiego zasobu się odwołujemy i separacji warstw. Przykład komunikacji z użyciem stylu REST w API został przedstawiony na rysunku 3.2 [3].



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

3.2. Aplikacja kliencka

Interfejs użytkownika (ang. *frontend*) zrealizowano jako aplikację kliencką komunikującą się z warstwą serwerową poprzez API.

3.2.1. Angular

Angular to framework do tworzenia aplikacji internetowych. Został opracowany przez Google i nadal jest rozwijany. Oferuje szeroką gamę narzędzi, interfejsów API oraz bibliotek, które upraszczają, optymalizują przepływ prac programistycznych. Zapewnia solidną platformę do tworzenia szybkich i niezawodnych aplikacji, które można skalować zarówno w zależności od wielkości zespołu, jak również rozmiaru kodu [10].

3.2.2. CSS

CSS (ang. *Cascading Style Sheets*) to kaskadowe arkusze stylów, które umożliwiają elastyczne rozmieszczenie elementów na stronach internetowych. Pozwalają one rozdzielić strukturę dokumentu (HTML) od struktury prezentacji. Dzięki temu kod jest lepiej zorganizowany i łatwiejszy w utrzymaniu. Pozwalają łatwo modyfikować elementy wizualne - kolory, czcionki, układ czy marginesy [9].

3.3. Baza danych

Baza danych to system przechowywania danych, zarządzania nimi w uporządkowany i łatwo dostępny sposób. Główną cechą bazy danych jest jej wydajność, która zapewnia szybką realizację operacji oraz sprawne zarządzanie dużą ilością informacji.

3.3.1. PostgreSQL

Zaawansowany system relacyjnych baz danych typu open source. Wykorzystuje SQL zapewniając wiele dodatkowych możliwości wspierających bezpieczne przechowywanie i skalowanie dużych ilości danych. Powstał w 1986 roku w ramach projektu POSTGRES Uniwersytetu Kalifornijskiego w Berkeley i jest stale rozwijany. Działa na wszystkich popularnych systemach operacyjnych, przestrzega zasad ACID, zapewniając wysoką spójność danych. PostgreSQL obsługuje różne typy danych, w tym: JSON, XML oraz typy geometryczne, takie jak punkty i wielokąty. Dzięki temu nadaje się do szerokiego zakresu zastosowań. Ponieważ użytkownicy mogą definiować własne typy danych lub zapisywać funkcje w różnych językach programowania, PostgreSQL jest niezwykle elastyczny oraz można go łatwo dostosować do konkretnych potrzeb. Baza danych posiada również rozbudowane mechanizmy indeksowania i optymalizacji zapytań, co zapewnia szybkie przetwarzanie danych nawet przy dużych obciążeniach [15].

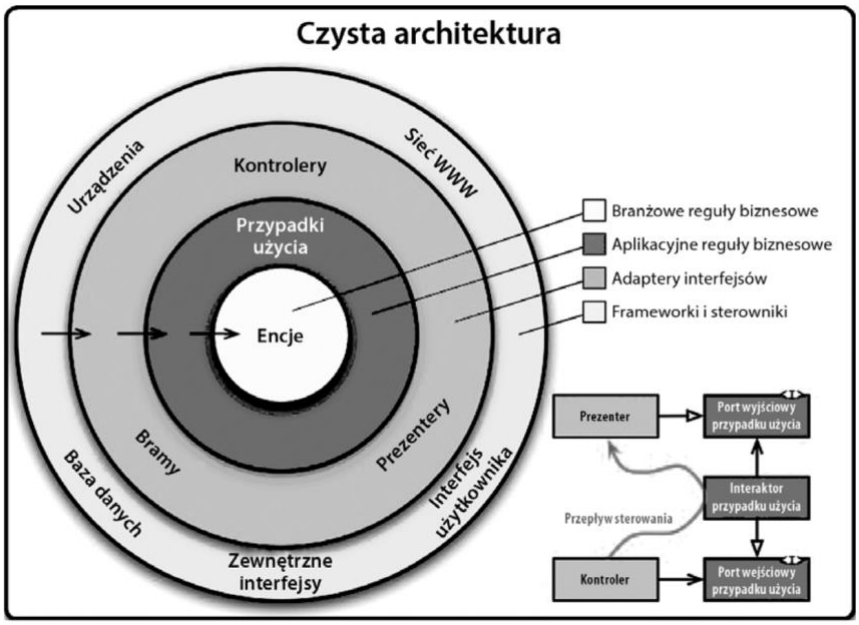
4. Implementacja systemu

W tej części zostaną przedstawione szczegóły techniczne aplikacji. Obejmują one opis architektury oprogramowania, strukturę bazy danych zilustrowaną za pomocą diagramu ERD, diagram przypadków użycia przedstawiający interakcje użytkowników z systemem oraz specyfikację API.

4.1. Architektura systemu

Wybór właściwej architektury systemu ma kluczowe znaczenie, dlatego warto dobrze się nad nią zastanowić. System został zaimplementowany wykorzystując czystą architekturę (ang. *clean architecture*). Jej celem jest podzielenie oprogramowania na warstwy, z których każda ma określone zadanie i może wykonywać swoje działania niezależnie od pozostałych. Wyróżniamy warstwy:

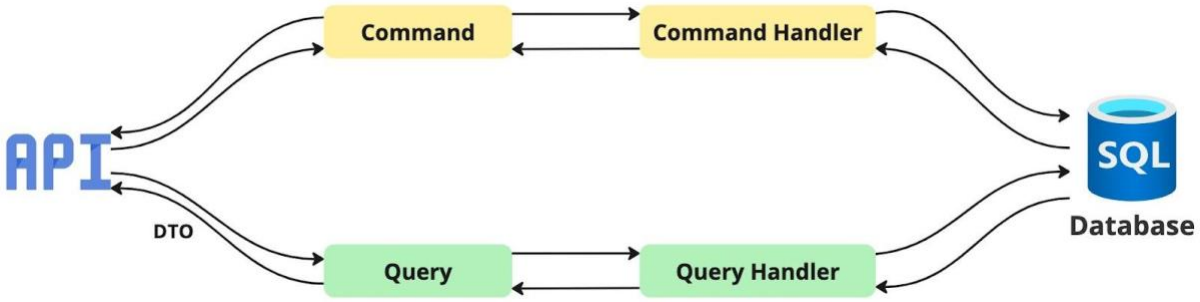
- **interfejsu użytkownika** (ang. *Api*) odpowiada za prezentację danych.
- **aplikacji** (ang. *Application*) zawiera logikę biznesową.
- **domeny** (ang. *Core*) jest sercem systemu i zawiera najważniejsze reguły biznesowe oraz logikę aplikacyjną.
- **infrastruktury** (ang. *Infrastructure*) odpowiada za realizację aspektów technicznych, takich jak dostęp do baz danych, komunikację z usługami zewnętrznymi oraz zarządzanie danymi.



Rys. 4.1 - *Czysta architektura [7]*

Główną regułą tej architektury jest zasada zależności mówiąca, że relacje powinny być skierowane do wnętrza architektury, w stronę reguł o wyższym poziomie (najwyższy poziom znajduje się w centralnym okręgu na rysunku 4.1). Granice między okręgami zostały tak wyznaczone, aby elementy znajdujące się wewnątrz danego okręgu nie miały dostępu do elementów w warstwach wyższych. Najważniejsze zalety wynikające z zastosowania czystej architektury to łatwość testowania i wprowadzania zmian dzięki zastosowaniu oddzielnych modułów [7].

Przy implementacji systemu wykorzystano także wzorzec projektowy CQRS (ang. *Command Query Responsibility Segregation*). Pozwala to na rozdzielenie operacji na dwie grupy: komendy (ang. *command*) oraz kwerendy (ang. *query*). Pierwsze z nich modyfikują stan bazy danych, czyli wszystkie operacje związane z przechowywaniem, takie jak dodawanie, usuwanie lub edytowanie danych. Z kolei drugie nie zmieniają stanu bazy danych, a jedynie pobierają odpowiednie dane. CQRS zwiększa bezpieczeństwo, upraszcza przetwarzanie danych oraz zwiększa elastyczność i wydajność systemu [18]. W podejściu CQRS zaleca się użycie behawioralnego wzorca projektowego Mediator. Jego zadaniem jest zarządzanie przepływem komend i kwerend oraz kierowanie ich do odpowiednich handlerów. Na rysunku 4.2 została przedstawiona przykładowa komunikacja API z wykorzystaniem wzorca CQRS z podziałem na odpowiednie akcje serwera.



Rys. 4.2 - *CQRS [opracowanie własne]*

Aby zilustrować użycie wzorca CQRS, przykładową implementację komendy pokazano na rysunku 4.3, a przykład kwerendy na rysunku 4.4.

```
internal record AddImageToPetCommand(Guid PetId, IFormFile File) : IRequest<AddImageToPetResponse>;

internal sealed class AddImageToPetCommandHandler(
    IBlobStorageService blobStorageService,
    IPetRepository petRepository,
    IImageRepository imageRepository
) : IRequestHandler<AddImageToPetCommand, AddImageToPetResponse>
{
    public async Task<AddImageToPetResponse> Handle(AddImageToPetCommand command,
        CancellationToken cancellationToken)
    {
        var pet = await petRepository.GetAsync(x => x.Id == command.PetId, cancellationToken)
                  ?? throw new PetNotFoundException(command.PetId);

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

            await imageRepository.DeleteAsync(pet.Image, cancellationToken);
            await imageRepository.SaveChangesAsync(cancellationToken);
        }

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

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

        pet.SetImage(image);

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

        return new AddImageToPetResponse(image.Id);
    }
}

internal record AddImageToPetResponse(Guid ImageId);
```

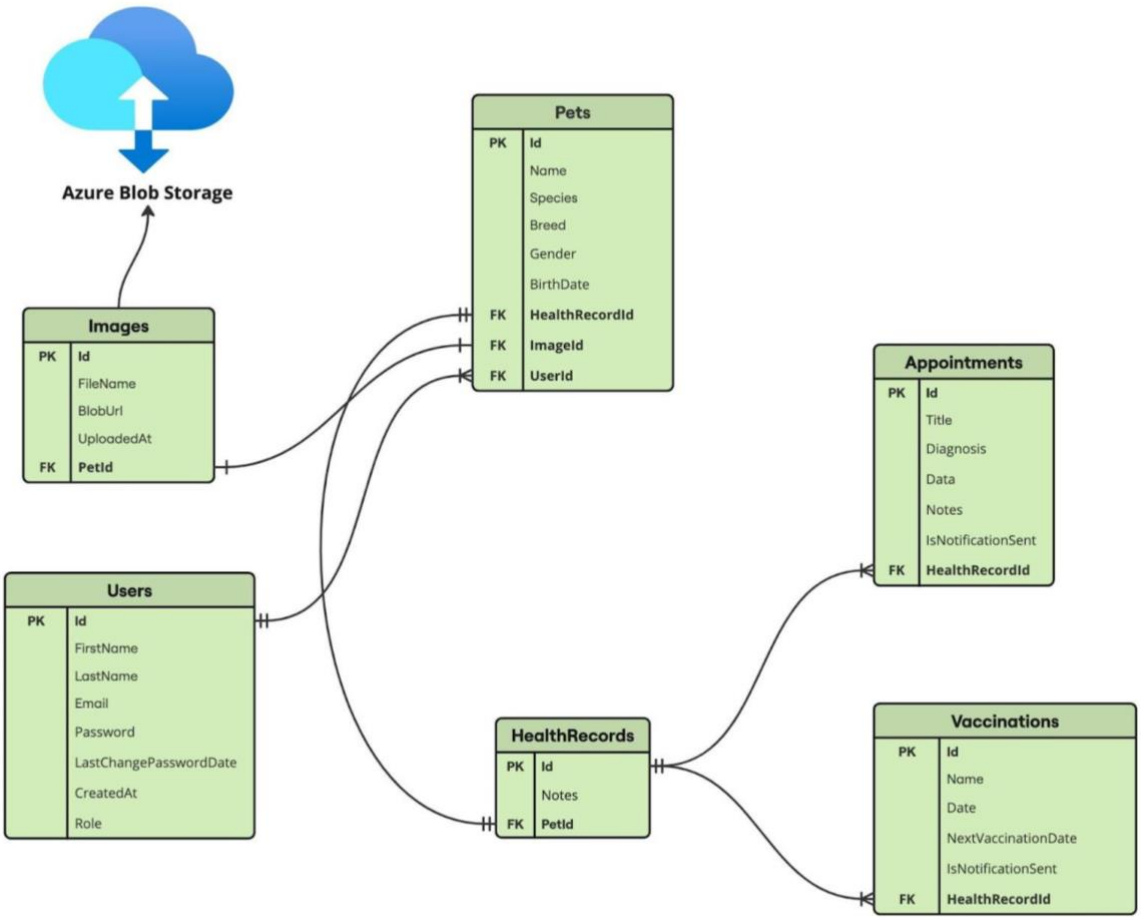
Rys. 4.3 - *Komenda [opracowanie własne]*

```
internal record GetCurrentUserDetailsQuery : IRequest<CurrentUserDetailsDto>;

internal sealed class GetCurrentUserDetailsQueryHandler(
    IUserRepository userRepository,
    IContext context
) : IRequestHandler<GetCurrentUserDetailsQuery, CurrentUserDetailsDto>
{
    public async Task<CurrentUserDetailsDto> Handle(GetCurrentUserDetailsQuery query,
        CancellationToken cancellationToken)
    {
        var user = await userRepository.GetAsync(x => x.Id == context.UserId, cancellationToken)
                   ?? throw new UserNotFoundException(context.UserId);

        if (user is null)
            throw new UserNotFoundException(context.UserId);

        return user.AsDetailsDto();
    }
}

internal record CurrentUserDetailsDto(
    Guid UserId,
    string FirstName,
    string LastName,
    string Email,
    DateTimeOffset? LastChangePasswordDate,
    DateTimeOffset CreatedAt,
    string Role,
    int PetsCount
);
```

Rys. 4.4 - *Kwerenda [opracowanie własne]*

4.2. Diagram ERD

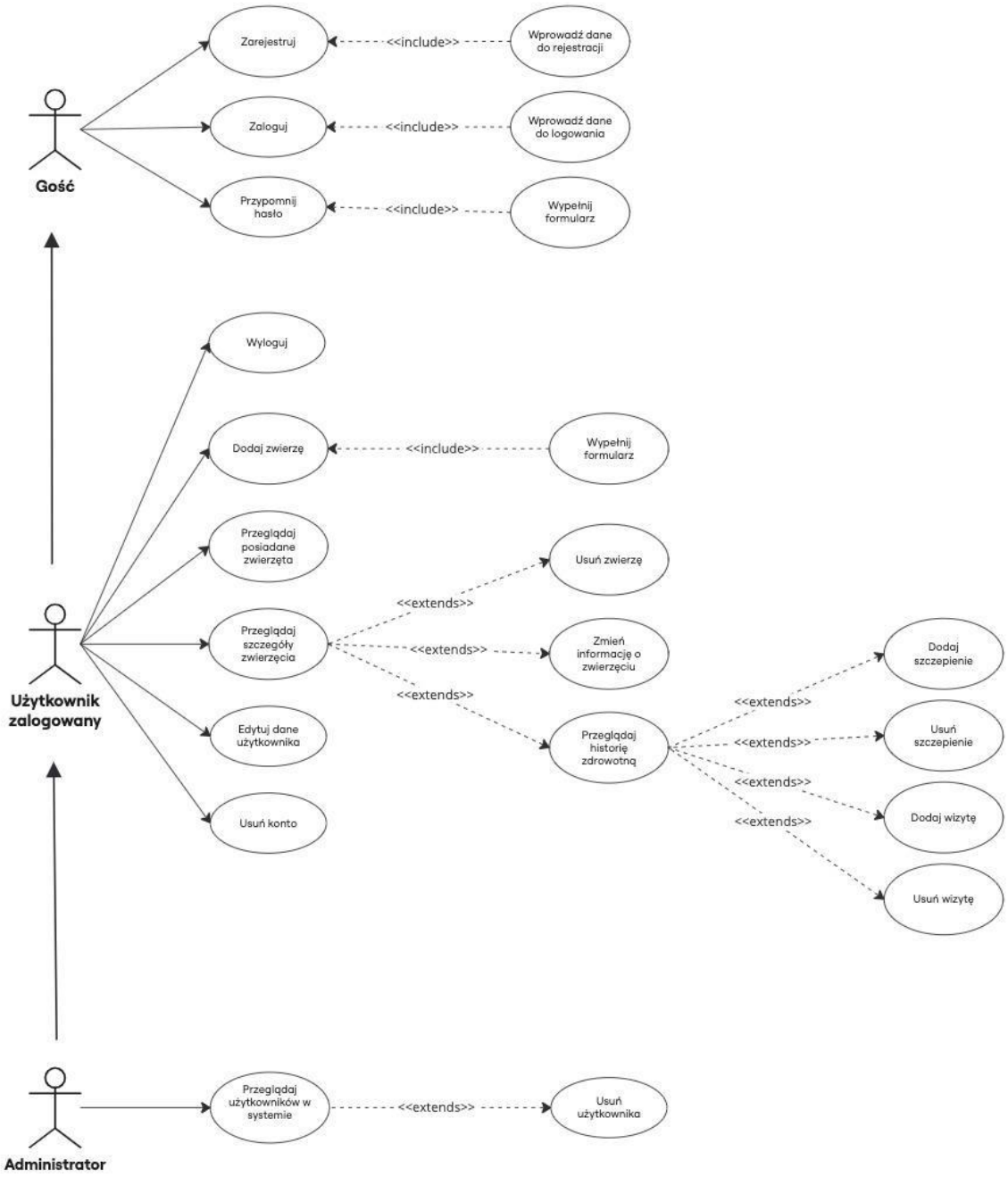
Diagram ERD (ang. *Entity Relationship Diagram*) przedstawia w sposób graficzny schemat bazy danych, wizualizuje encje, ich atrybuty oraz relacje między nimi. Rysunek 4.5 przedstawia diagram ERD opracowany dla implementowanego systemu. Zdjęcia powiązane z systemem są przechowywane w Azure Blob Storage, co umożliwia bezpieczne i skalowalne zarządzanie danymi multimedialnymi.



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

4.3. Diagram przypadków użycia

Diagram przypadków użycia (ang. *Use Case Diagram*) jest częścią języka UML (ang. *Unified Modeling Language*). Daje możliwość graficznego przedstawienia dostępnych dla aktorów działań. W omawianym systemie możemy wyróżnić trzech aktorów: gościa, użytkownika zalogowanego oraz administratora. Rysunek 4.6 przedstawia przypadki użycia dla opisywanego systemu.



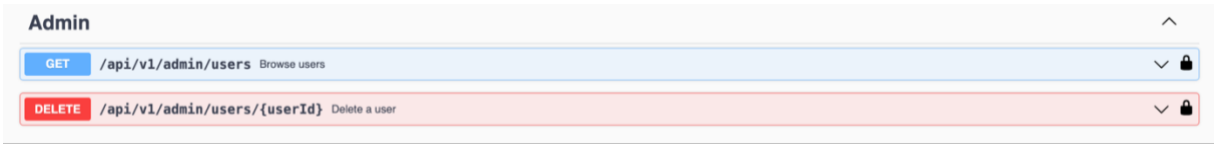
Rys. 4.6 - *Diagram przypadków użycia [opracowanie własne]*

4.4. Dokumentacja API

Do przygotowania dokumentacji API wykorzystano narzędzie *Swagger*, które automatycznie generuje przejrzystą dokumentację. Pozwala na graficzne zobrazowanie modułów systemowych oraz dostępnych endpointów. Każdy endpoint został szczegółowo opisany, obejmując jego zastosowanie, wymagane dane wejściowe, zwracane dane wyjściowe oraz statusy protokołu HTTP. Na rysunkach 4.7, 4.8, 4.9 i 4.10 zostały przedstawione moduły aplikacji wraz z endpointami.

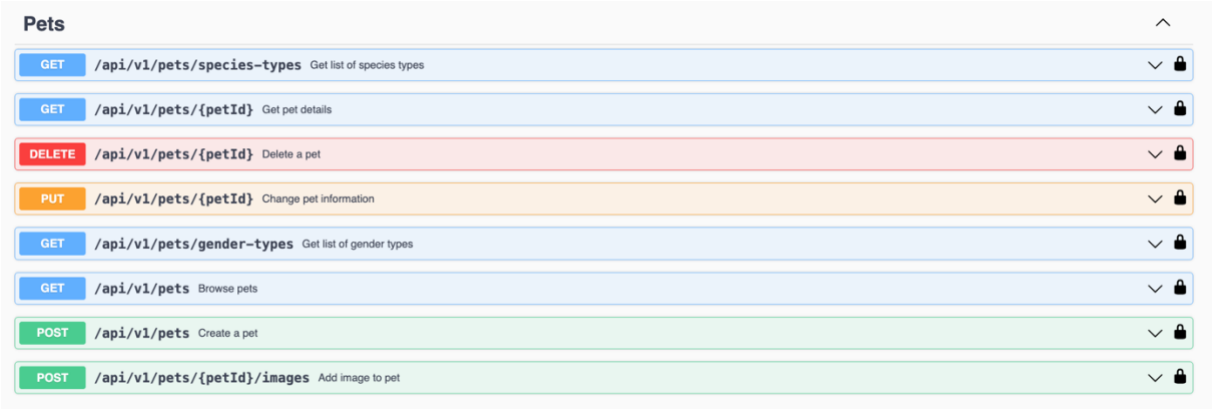
Rys. 4.7 - *Users [opracowanie własne]*

Moduł *Users* jest odpowiedzialny za obsługę użytkowników w systemie. Jego głównym zadaniem jest autoryzacja i uwierzytelnianie użytkowników. Pozwala na utworzenie konta, logowanie, edycję danych, pobranie danych, a także usunięcie konta z systemu. Po zalogowaniu użytkownik otrzymuje unikalny token, który umożliwia dostęp do innych zasobów systemu i zapewnia bezpieczne uwierzytelnianie przy kolejnych żądaniach.



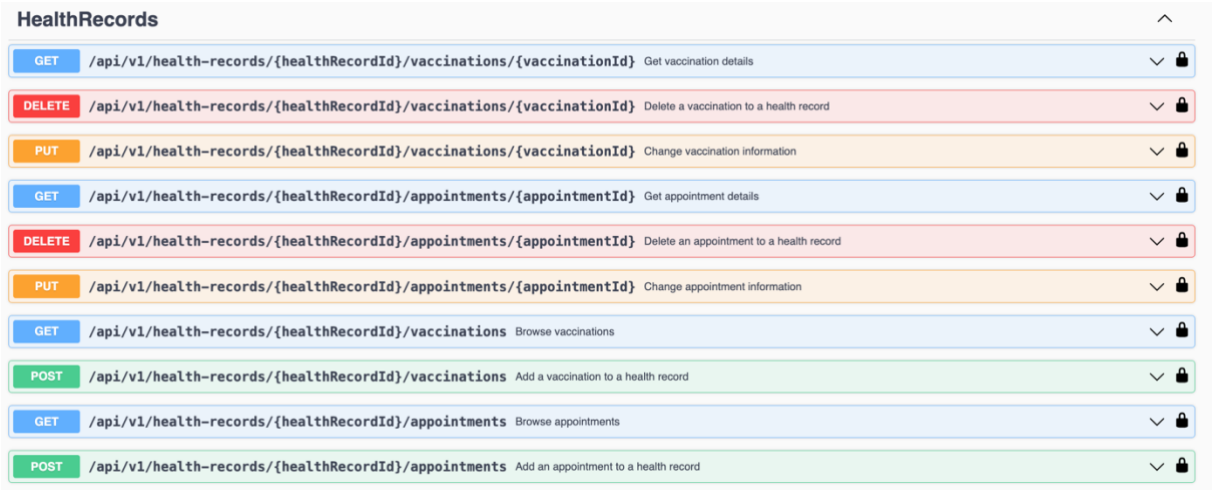
Rys. 4.8 - *Admin [opracowanie własne]*

Moduł *Admin* jest zarezerwowany dla administratora systemu umożliwiając zarządzanie użytkownikami. Pozwala monitorować użytkowników poprzez przeglądanie kont i w razie potrzeby daje możliwość usunięcia konta. Moduł ten pozwala na utrzymanie porządku w systemie.



Rys. 4.9 - *Pets [opracowanie własne]*

Moduł *Pets* jest odpowiedzialny za zarządzaniem wszystkimi aspektami dotyczącymi zwierząt. Umożliwia przeglądanie istniejących pupili, pozyskiwanie szczegółowych informacji o każdym z nich oraz dodawanie nowych do bazy danych. Dodatkowo pozwala na edycję oraz usuwanie danych dotyczących zwierząt, co zapewnia pełną kontrolę nad zarządzaniem.

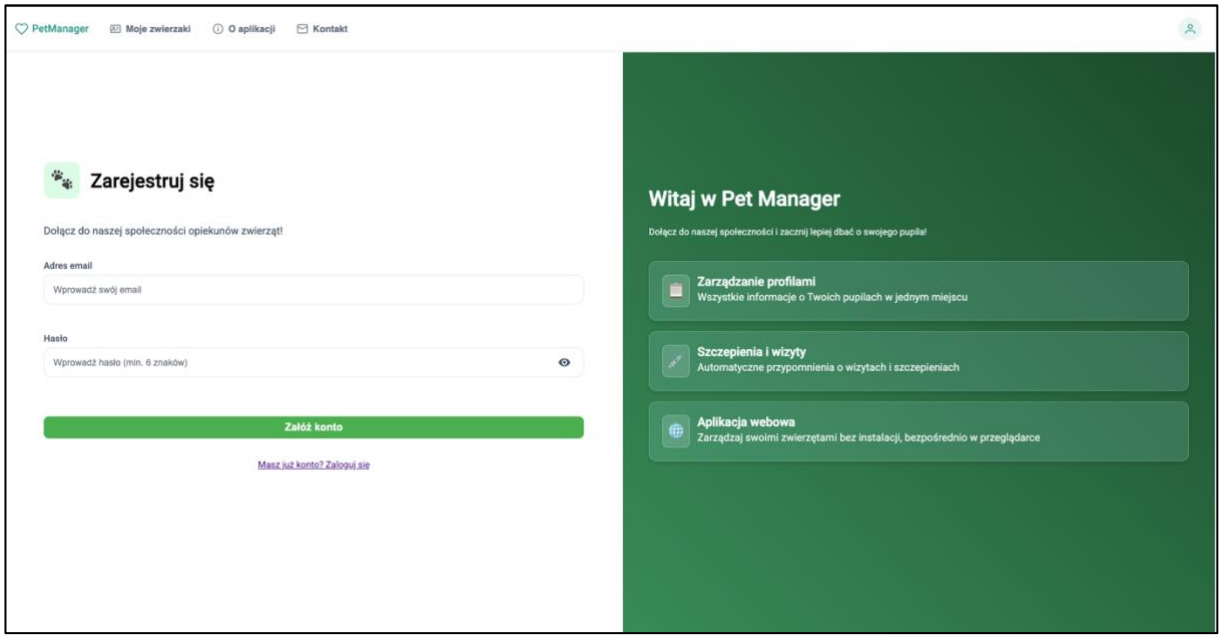


Rys. 4.10 - *HealthRecords [opracowanie własne]*

Moduł *HealthRecords* jest powiązany z modułem *Pets* i stanowi jego rozwinięcie. Podczas dodawania nowego zwierzęcia automatycznie tworzona jest jego karta zdrowia, która umożliwia dokumentowanie jego historii zdrowotnej. Pozwala na dodawanie oraz usuwanie zapisów dotyczących szczepień i wizyt weterynaryjnych. Dzięki temu w przejrzysty sposób można śledzić informacje o stanie zdrowia zwierząt.

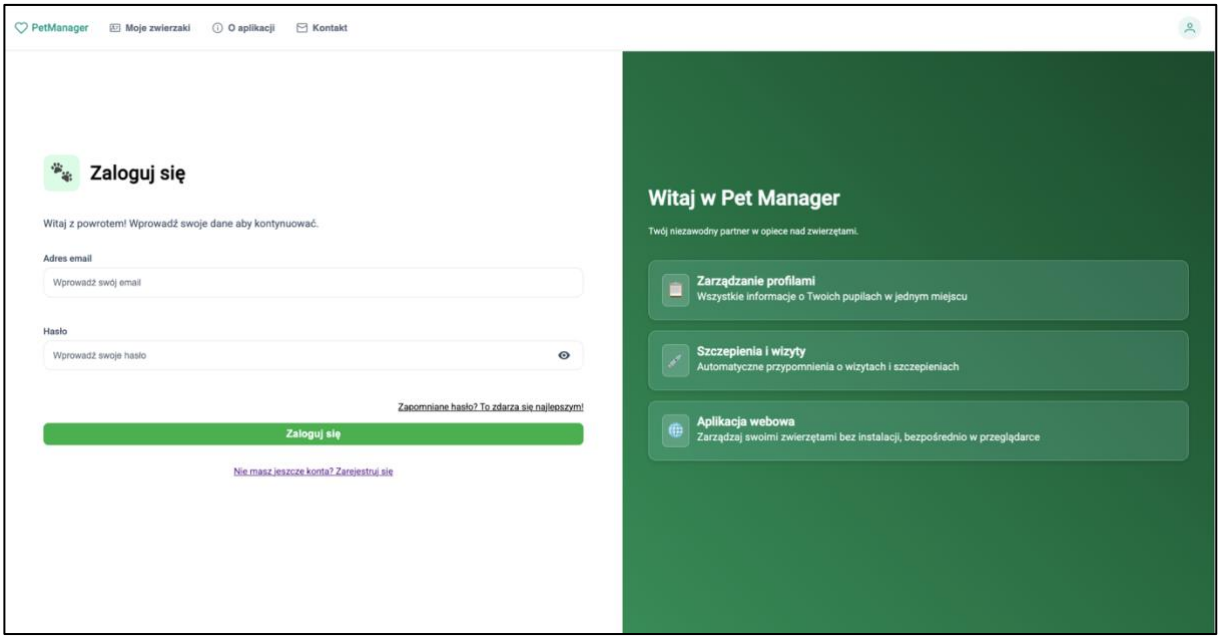
5. Interfejsy użytkownika

Aplikacja kliencka została zaimplementowana w formie aplikacji webowej, co sprawia, że jest łatwo dostępna bezpośrednio z poziomu przeglądarki internetowej, bez konieczności instalowania dodatkowego oprogramowania. Projekt interfejsu został starannie zaprojektowany, aby zapewnić intuicyjną obsługę, co znacząco zwiększa wygodę użytkowania. W dalszej części rozdziału przedstawiono szczegółowy opis poszczególnych interfejsów graficznych.



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

Aby rozpocząć korzystanie z aplikacji, użytkownik musi zarejestrować swoje konto w systemie. Jak pokazano na rysunku 5.1, strona rejestracji została zaprojektowana w sposób prosty i intuicyjny. Formularz wymaga podania jedynie podstawowych danych, takich jak adres e-mail oraz hasło. Takie uproszczenie ma na celu maksymalne przyspieszenie procesu rejestracji, aby użytkownik mógł jak najszybciej rozpocząć korzystanie z aplikacji. Po wprowadzeniu wymaganych informacji należy nacisnąć przycisk *„Załóż konto”*. W przypadku prawidłowego wypełnienia formularza i pomyślnego zakończenia procesu rejestracji, użytkownik zostaje automatycznie przeniesiony do strony logowania.



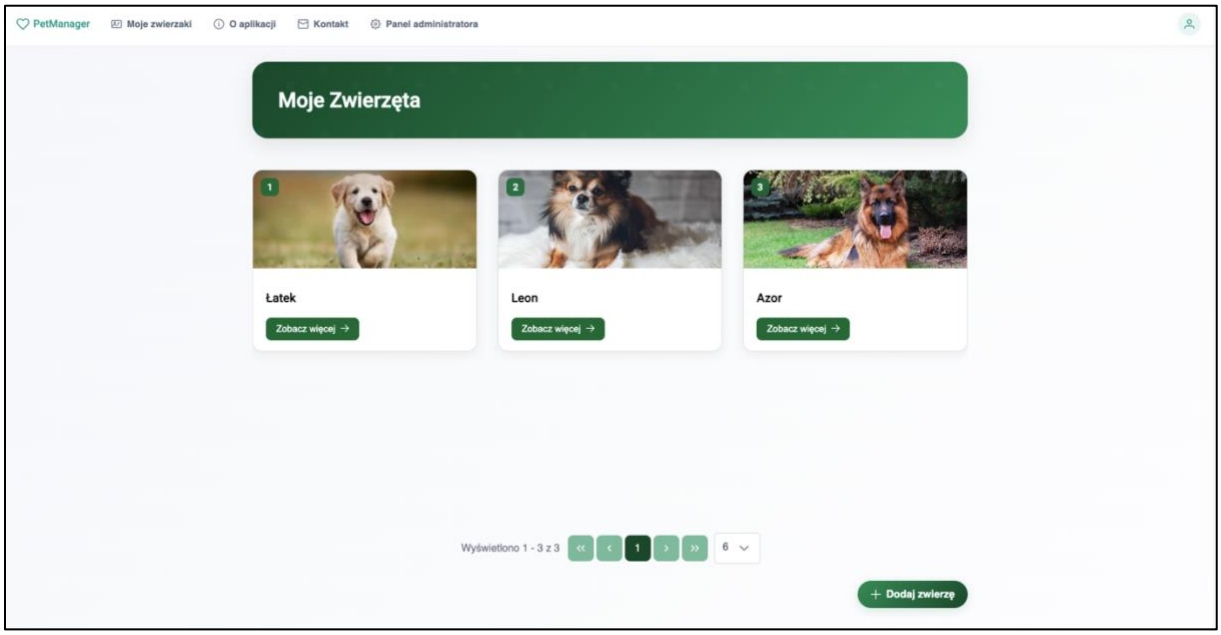
Rys. 5.2 - *Strona logowania [opracowanie własne]*

Strona logowania, przedstawiona na rysunku 5.2, jest pierwszą stroną, którą zostaje wyświetlona użytkownikowi po wejściu do aplikacji. Formularz składa się z dwóch podstawowych pól: adresu e-mail oraz hasła, które użytkownik musi wprowadzić, aby uzyskać dostęp do swojego konta. Po wprowadzeniu danych wystarczy kliknąć przycisk *„Zaloguj się”*, aby móc korzystać w pełni z możliwości systemu.

Jeśli użytkownik nie posiada jeszcze konta w systemie, poniżej formularza logowania znajduje się przycisk: *„Nie masz jeszcze konta? Zarejestruj się”*, który przenosi użytkownika bezpośrednio do strony rejestracji, gdzie może założyć nowe konto i rozpocząć korzystanie z aplikacji.

Dodatkowo strona logowania oferuje opcję odzyskania dostępu do swojego konta. W przypadku, gdy użytkownik zapomniał swojego hasła, wystarczy, że kliknie przycisk *„Zapomniane hasło? To zdarza się najlepszym!”*, który przekieruje go do procesu odzyskiwania dostępu do konta. Dzięki tej opcji, użytkownicy mogą łatwo odzyskać swoje dane logowania, jeśli napotkają jakiekolwiek trudności.

Po zalogowaniu użytkownik trafia na stronę *„Moje zwierzęta”*, która została przedstawiona na rysunku 5.3 gdzie widzi imiona i zdjęcia swoich pupili. Każdy z nich jest przedstawiony jako miniaturka z przyciskiem *„Zobacz więcej”*, prowadzącym do szczegółów na jego temat.



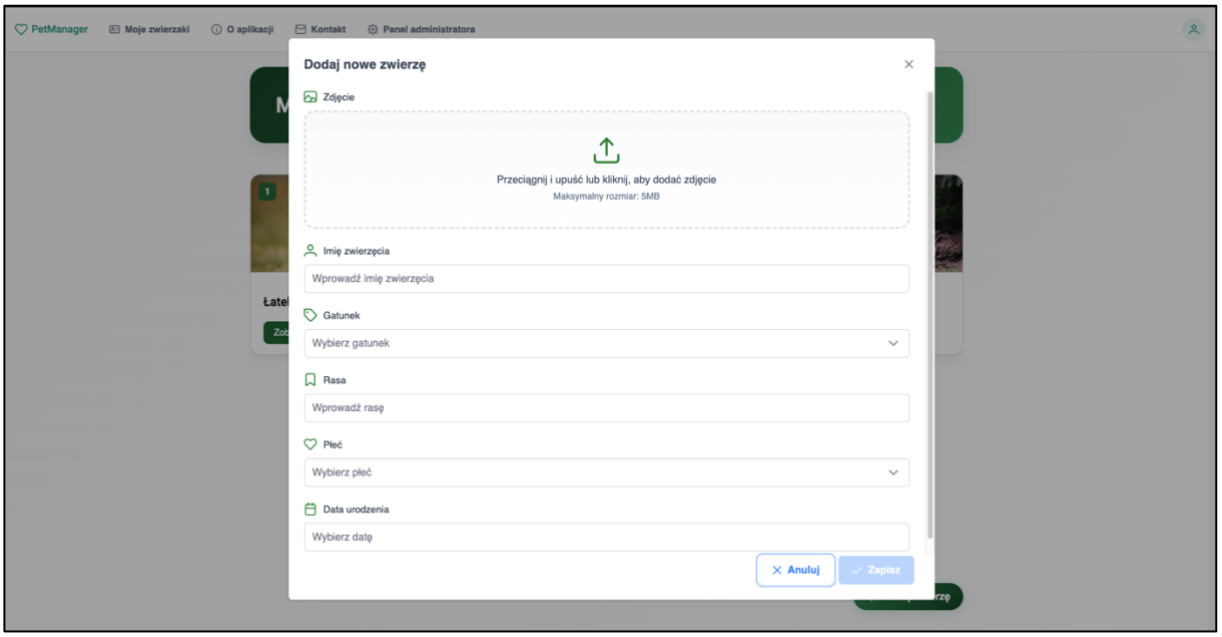
Rys. 5.3 - *Moje zwierzęta [opracowanie własne]*

Strona posiada także mechanizm paginacji, który pozwala przeglądać dłuższe listy zwierząt w mniejszych częściach. Dzięki temu użytkownik nie musi oglądać wszystkich naraz i może łatwiej poruszać się po stronie. Liczbę zwierząt wyświetlanych na jednej stronie można dostosować, a nawigacja między stronami jest prosta i przejrzysta. Funkcja ta sprawia, że aplikacja działa sprawnie, nawet przy dużej liczbie danych. Przejrzysty interfejs i dobrze przemyślany układ sprawiają, że strona główna jest wygodnym miejscem do zarządzania zwierzętami, łącząc prostotę i praktyczność.

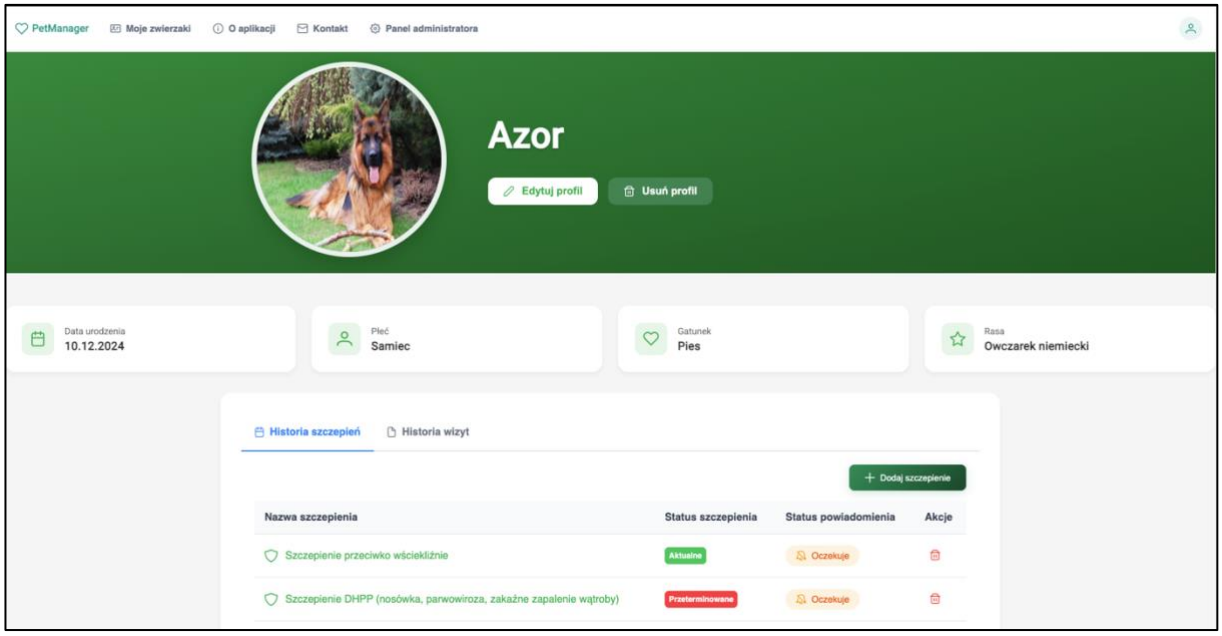
Dodatkowo w prawym dolnym rogu znajduje się przycisk *„Dodaj zwierzę”*, po jego kliknięciu użytkownikowi wyświetla się okno dialogowe zawierające formularz, który został przedstawiony na rysunku 5.4.

Formularz został podzielony na logiczne sekcje, rozpoczynając się od możliwości dodania fotografii pupila. Użytkownik może przeciągnąć zdjęcie lub kliknąć w wyznaczony obszar, aby dodać zdjęcie, przy czym system informuje o maksymalnym dopuszczalnym rozmiarze pliku wynoszącym 5 MB. Poniżej znajdują się starannie opisane pola służące do wprowadzenia podstawowych informacji o zwierzęciu, takich jak imię, gatunek, rasa, płeć, data urodzenia. Każde pole formularza zostało opatrzone odpowiednią etykietą oraz ikoną, co znacząco ułatwia użytkownikowi zrozumienie, jakie dane powinny zostać wprowadzone.

Na dole formularza umieszczono intuicyjne przyciski kontrolne. Biały przycisk *„Anuluj”* pozwala na bezpieczne zamknięcie okna bez zapisywania wprowadzonych danych, podczas gdy niebieski przycisk *„Zapisz”* zatwierdza wszystkie wprowadzone informacje. System został wyposażony w mechanizm walidacji - przycisk *„Zapisz”* pozostaje nieaktywny do momentu, gdy wszystkie wymagane pola zostaną poprawnie wypełnione. Użyty mechanizm zapobiega zapisywaniu niekompletnych lub niepoprawnych danych, jednocześnie informując użytkownika o konieczności uzupełnienia wszystkich wymaganych informacji.



Rys. 5.4 - *Formularz - dodaj nowe zwierzę [opracowanie własne]*



Rys. 5.5 - *Strona - szczegóły zwierzęcia [opracowanie własne]*

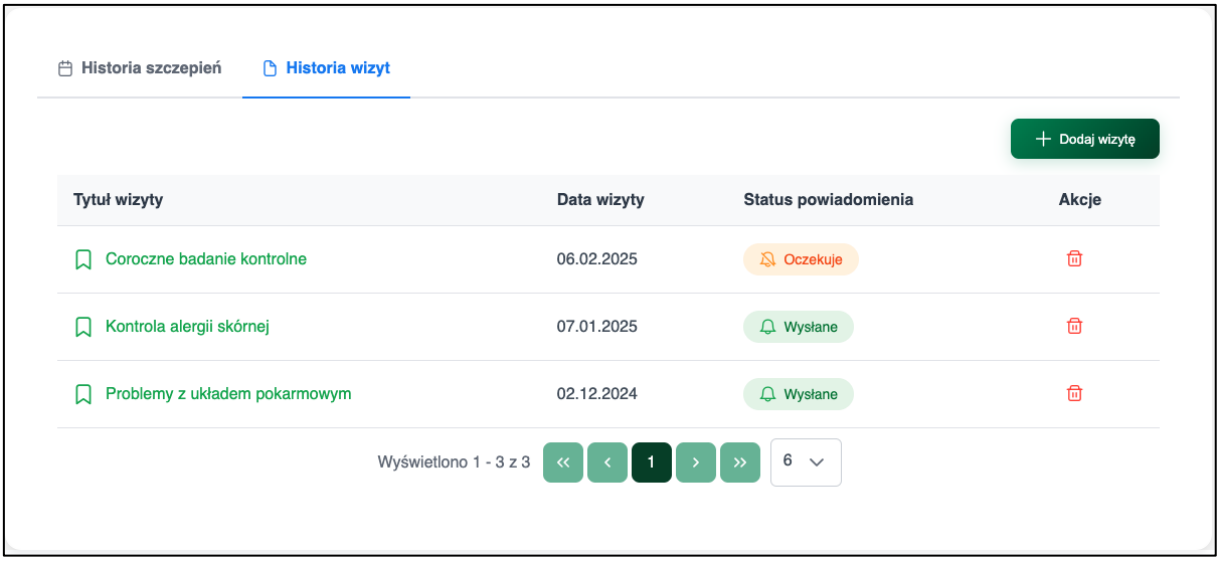
Po kliknięciu przycisku *„Zobacz więcej”* przy wybranym zwierzęciu, użytkownik zostaje przeniesiony na stronę szczegółów, przedstawioną na rysunku 5.5. Na górze strony znajduje się zdjęcie zwierzęcia wraz z jego imieniem.

W centralnej części ekranu wyświetlane są kluczowe informacje o pupilu, przedstawione w przejrzystej formie kafelków z odpowiednimi ikonami:

- Data urodzenia
- Płeć
- Gatunek
- Rasa

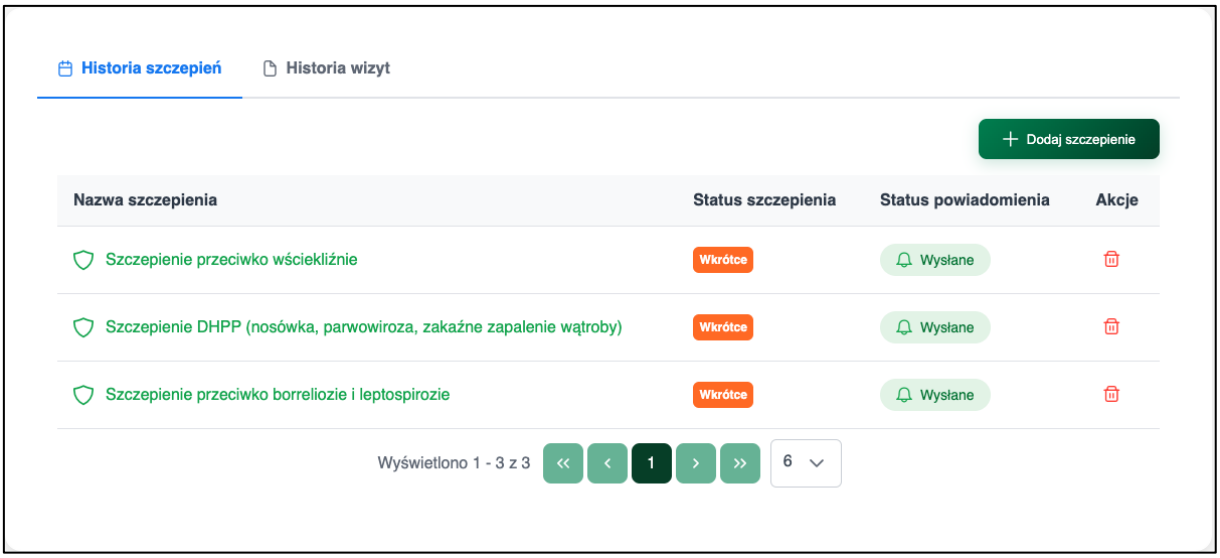
Poniżej znajdują się dwie zakładki nawigacyjne: *„Historia szczepień”* oraz *„Historia wizyt”*, pozwalające na przełączanie się między różnymi rodzajami informacji medycznych.

W zakładce historii wizyt przedstawionej na rysunku 5.6 prezentowana jest tabela zawierająca spis wszystkich wizyt weterynaryjnych, wraz z ich tytułami, datami oraz statusem czy przypomnienie zostało już wysłane. System umożliwia przeglądanie wpisów przy pomocy paginacji, a każdy wpis zawiera istotne informacje medyczne oraz zalecenia dotyczące dalszego postępowania.



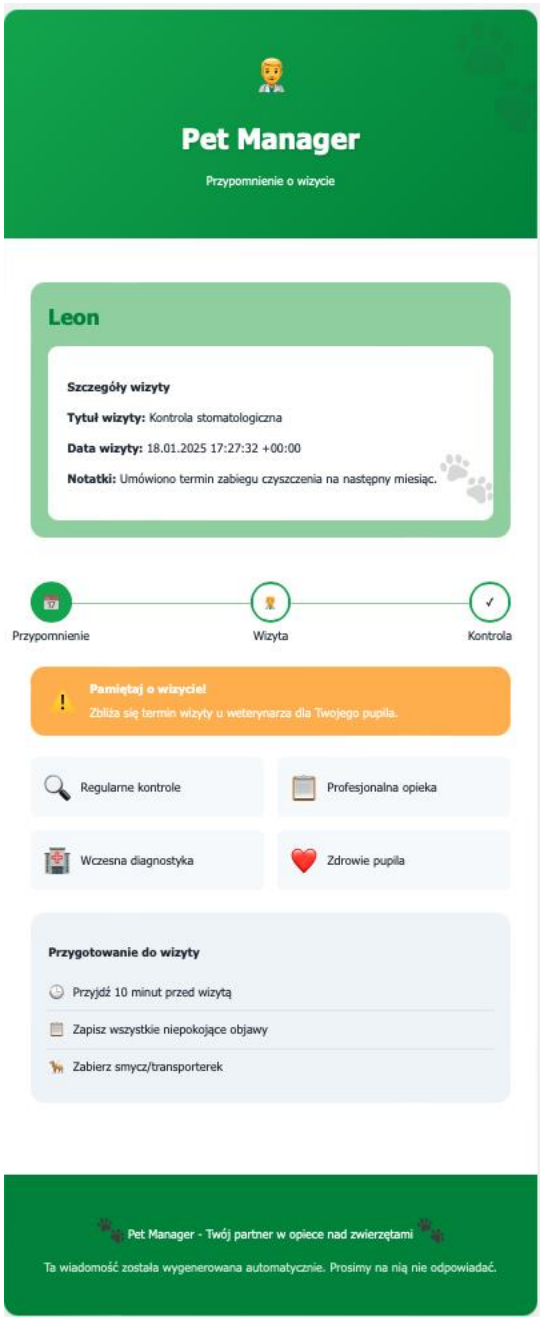
Rys. 5.6 - *Zakładka Historia wizyt [opracowanie własne]*

Zakładka *„Historia szczepień”*, przedstawiona na rysunku 5.7, prezentuje kompleksowy rejestr wszystkich przeprowadzonych szczepień. Tabela zawiera następujące informacje: nazwa szczepienia oraz status szczepienia (oznaczony kolorami: pomarańczowy dla zbliżających się terminów, zielony dla aktualnych). Każde szczepienie ma jasny status informujący o jego aktualności. System automatycznie oznacza nadchodzące terminy kolejnych dawek, co ułatwia utrzymanie regularnego harmonogramu. System paginacji obecny w obu zakładkach pozwala na efektywne zarządzanie większą ilością wpisów, a jednolity styl prezentacji danych zapewnia spójność wizualną całej aplikacji.



Rys. 5.7 - *Zakładka Historia szczepień [opracowanie własne]*

System wysyła automatyczne powiadomienia e-mail przypominające o wizytach weterynaryjnych i szczepieniach, którego przykład został pokazany na rysunku 5.8. Wiadomość e-mail ma prostą, przejrzystą strukturę zawierającą zielone nagłówki z imieniem zwierzęcia, szczegóły wizyty oraz dodatkowe notatki w białym polu. W dolnej części znajdują się praktyczne ikony, wskazówki dotyczące przygotowania do wizyty, a całość zamyka zielona stopka.

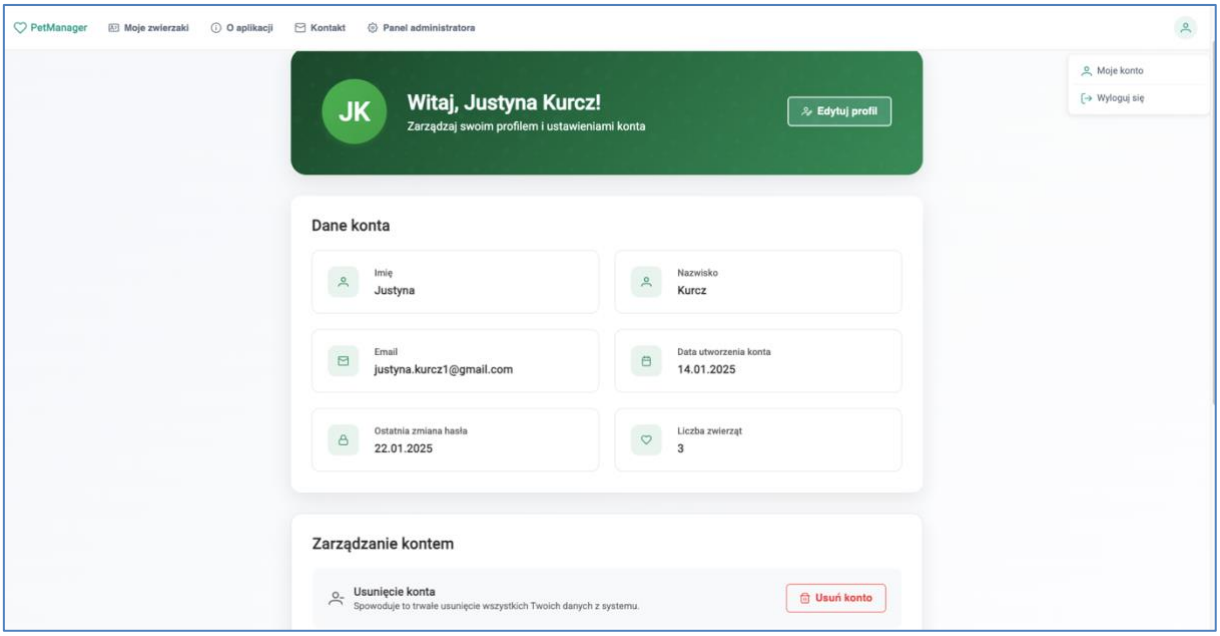


Rys. 5.8 - *E-mail [opracowanie własne]*

Dostęp do szczegółów konta możliwy jest poprzez kliknięcie w ikonę użytkownika w prawym górnym rogu aplikacji, a następnie wybranie opcji *„Moje konto”* z rozwijanego menu. Strona szczegółów konta, przedstawiona na rysunku 5.9, prezentuje podstawowe informacje o koncie użytkownika w przejrzystej formie.

Widok zawiera kluczowe informacje o użytkowniku, w tym jego imię, nazwisko, adres e-mail oraz data utworzenia konta. W prawym górnym rogu znajduje się biały przycisk *„Edytuj profil”*, który po kliknięciu wyświetla formularz edycji danych użytkownika.

System daje również użytkownikowi możliwość trwałego usunięcia swojego konta poprzez opcję *„Usuń konto”*. Po wybraniu tej opcji i potwierdzeniu decyzji, wszystkie dane użytkownika oraz powiązane z nim informacje są nieodwracalnie usuwane z systemu.



Rys. 5.9 - *Ekran - szczegóły konta [opracowanie własne]*

6. Testy

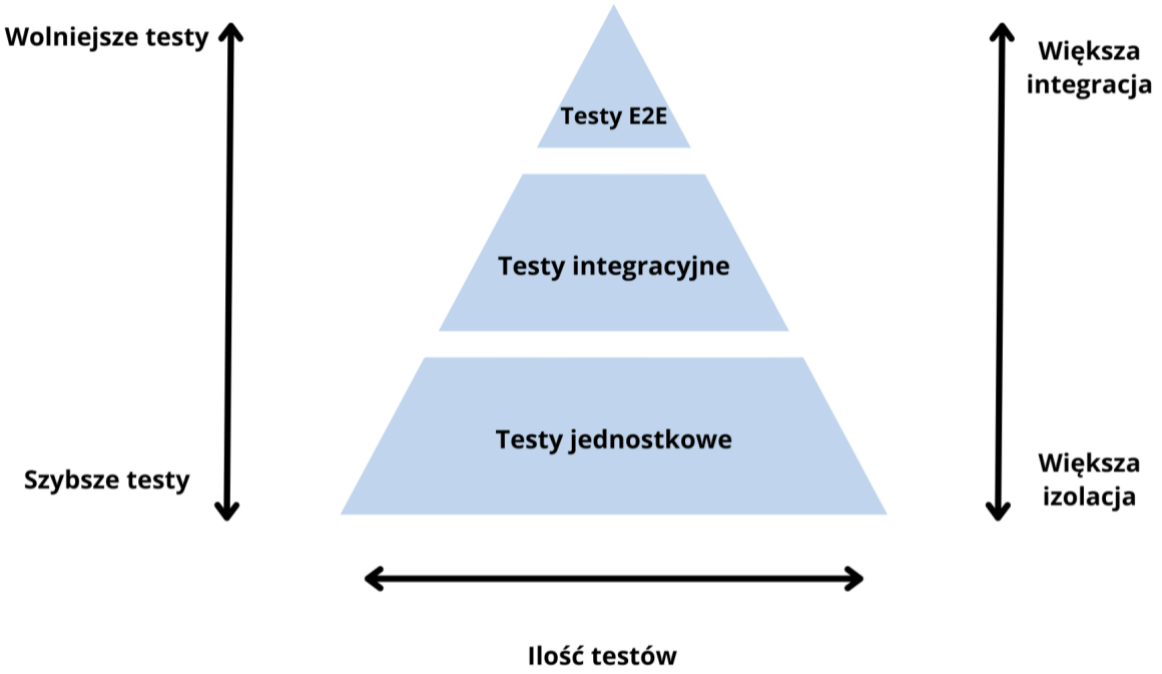
Testowanie oprogramowania to podstawowy filar procesu budowania systemów informatycznych, który nie tylko weryfikuje zgodność z wymaganiami, ale również gwarantuje wysoką jakość końcowego produktu. W obecnych czasach systemy informatyczne często obsługują krytyczne procesy biznesowe, gdzie nawet pozornie drobne błędy mogą prowadzić do poważnych konsekwencji finansowych, wizerunkowych, a w skrajnych przypadkach nawet zagrażać bezpieczeństwu użytkowników.

Systematyczne podejście do testowania w trakcie rozwoju aplikacji zapewnia następujące korzyści:

- Wczesne wykrywanie i poprawianie błędów, co znacząco zmniejsza koszty ich naprawy.
- Utrzymanie wysokiej jakości kodu poprzez wymuszanie dobrych praktyk programistycznych.
- Szybsze wykrywanie problemów w integracji między komponentami i automatyczne sprawdzanie poprawności działania systemu.
- Minimalizację ryzyka wystąpienia błędów na środowisku produkcyjnym.

Testowanie implementowanej aplikacji webowej opiera się na podejściu wielopoziomowym, gdzie system jest weryfikowany na różnych poziomach abstrakcji, co zapewnia kompleksowe sprawdzenie jego działania. Przedstawiono to na rysunku 6.1 w formie piramidy testów, której struktura przedstawia relacje między poszczególnymi rodzajami testów. Szerokość każdego poziomu piramidy odzwierciedla ilość testów danego typu, natomiast pionowe osie pokazują dodatkowe charakterystyki. Lewa oś wskazuje na szybkość wykonywania testów, które są tym szybsze, im niżej znajdują się w piramidzie, podczas gdy prawa oś określa poziom integracji komponentów, który rośnie wraz z wysokością w piramidzie.

W implementowanym systemie wykorzystywane są dwa podstawowe poziomy testów przedstawione w piramidzie. Bazę stanowią testy jednostkowe, będące najliczniejszą i najszybszą w wykonaniu grupą testów. Nad nimi znajdują się testy integracyjne, które weryfikują współdziałanie między komponentami systemu. Szczegółowy opis obu poziomów testów zostanie przedstawiony w dalszej części rozdziału.



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

Dodatkowo w procesie implementacji testów zastosowano wzorzec AAA (ang. *Arrange-Act-Assert*), który porządkuje strukturę testu poprzez podział na trzy główne części:

- **Arrange** (tłum. *Przygotowanie*) w tej fazie inicjalizujemy potrzebne zasoby, konfigurujemy zależności i ustalamy warunki początkowe potrzebne do wykonania testu.
- **Act** (tłum. *Działanie*) w tym etapie dokonujemy konkretnych operacji, które chcemy przetestować.
- **Assert** (tłum. *Asercja*) w końcowym etapie sprawdzamy, czy otrzymany rezultat jest zgodny z założonymi oczekiwanymi.

Stosowanie wzorca AAA zapewnia wysoką czytelność i przejrzystość testów, ułatwiając ich utrzymanie oraz pozwala szybko zidentyfikować cel i sposób działania każdego testu [8].

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

Na rysunku 6.2 przedstawiono wyniki testów zaimplementowanych w projekcie, których łączna liczba wynosi 213. Wśród nich znajduje się 88 testów integracyjnych oraz 125 testów jednostkowych.

Taki podział testów odpowiada założeniom piramidy testów, która została zaprezentowana na rysunku 6.1. Zgodnie z jej wymaganiami największą liczbę testów w projekcie powinny stanowić testy jednostkowe, które są szybkie, łatwe w implementacji i pozwalają na szczegółowe sprawdzenie działania poszczególnych elementów systemu. Mniejszą część piramidy, co widoczne jest także w implementacji projektu, stanowią testy integracyjne, które weryfikują współdziałanie różnych modułów systemu.

Wyniki zaprezentowane na rysunku 6.2 wskazują, że wszystkie testy zakończyły się sukcesem. Świadczy to o poprawnym zaimplementowaniu wszelkich możliwości systemu i ich zgodności z przyjętymi założeniami projektowymi.

6.1. Testy jednostkowe

Testy jednostkowe (ang. *Unit Tests*) stanowią fundament procesu weryfikacji oprogramowania. Ich głównym celem jest weryfikacja poprawności działania jednostek, czyli pojedynczych komponentów systemu, w oderwaniu od reszty aplikacji. Jest to mechanizm weryfikujący czy testowany komponent spełnia założone wymagania poprzez wywołanie określonej operacji i sprawdzenie jej wyniku. Zakres testu jednostkowego może obejmować zarówno pojedynczą metodę lub współpracujące ze sobą klasy. Charakteryzują się automatyzacją, łatwością implementacji oraz szybkością wykonania [8].

Przykład testu jednostkowego weryfikującego poprawność tworzenia nowego obiektu *User* przedstawiono na rysunku 6.3. W implementacji wykorzystano mechanizm teorii (ang. *Theory*) z biblioteki *xUnit*, która pozwala na wykonanie testu z różnymi zestawami danych testowych. Test zaprojektowano zgodnie z wzorcem AAA, a jego nazwa odzwierciedla konwencję *given_when_then*, opisującą warunki początkowe, wykonywaną akcję oraz oczekiwany rezultat.

```
[Theory]
[ClassData(typeof(UserValidTestData))]
public void given_valid_data_when_create_user_then_should_create_new_user(
  UserParams userParams)
{
    // Arrange & Act
    var user = Act(userParams);

    // Assert
    user.ShouldNotBeNull();
    user.ShouldBeOfType<User>();
    user.Id.ShouldNotBe(Guid.Empty);
    user.Email.ShouldBe(userParams.Email);
    user.Password.ShouldBe(userParams.Password);
    user.Role.ShouldBe(userParams.Role);
    user.CreatedAt.ShouldNotBe(default);
}
```

Rys. 6.3 - *User - test jednostkowy [opracowanie własne]*

Atrybut *[ClassData]* wskazuje na klasę *UserValidTestData*, która dziedziczy po klasie generycznej *TheoryData<T>*, przedstawionej na rysunku 6.4. Klasa ta jest odpowiedzialna za dostarczenie danych testowych wygenerowanych przy użyciu biblioteki *Bogus*.

```
internal sealed class UserValidTestData : TheoryData<UserParams>
{
    private readonly Faker _faker = new();

    public UserValidTestData()
    {
        Add(new UserParams(
            _faker.Person.Email,
            _faker.Internet.Password(),
            _faker.PickRandom<UserRole>()
        ));
    }
}
```

Rys. 6.4 - *Implementacja klasy generującej dane testowe [opracowanie własne]*

Rysunek 6.5 przedstawia proces mockowania zależności w handlerze wykorzystującym wzorzec CQRS. Konfiguracja testowa obejmuje symulację repozytoriów oraz kontekstu za pomocą mocków dostarczonych przez bibliotekę *NSubstitute*. Zastosowane rozwiązanie pozwala na skuteczne testowanie logiki serwisów oraz repozytoriów bez konieczności wykorzystania ich faktycznych implementacji.

```
    private readonly IRequestHandler<CreatePetCommand, CreatePetResponse> _handler;

    private readonly IUserRepository _userRepository;
    private readonly IPetRepository _petRepository;
    private readonly IContext _context;
    private readonly PetTestFactory _petFactory = new();

    public CreatePetCommandHandlerTests()
    {
        _userRepository = Substitute.For<IUserRepository>();
        _petRepository = Substitute.For<IPetRepository>();
        _context = Substitute.For<IContext>();

        _handler = new CreatePetCommandHandler(
            _context,
            _userRepository,
            _petRepository
        );
    }
```

Rys. 6.5 - *Mockowanie zależności [opracowanie własne]*

Rysunek 6.6 przedstawia implementację testu jednostkowego dla handlera komendy *CreatePet*, który sprawdza poprawność logiki odpowiedzialnej za tworzenie nowego zwierzęcia przy poprawnych danych wejściowych. Dodatkowo sprawdzana jest liczba wywołań kluczowych zależności, takich jak repozytoria użytkowników i zwierząt, aby upewnić się, że interakcje są zgodne z oczekiwanym scenariuszem. Test zapewnia dokładną weryfikację zarówno logiki biznesowej, jak i interakcji z zależnościami.

```
[Fact]
public async Task Given_Valid_Data_When_Create_Pet_Then_Should_Create_Pet()
{
    // Arrange
    var command = _petFactory.CreatePetCommand();
    var userId = Guid.NewGuid();
    _context.UserId.Returns(userId);

    _userRepository
        .ExistsAsync(Arg.Any<Expression<Func<User, bool>>>(), Arg.Any<CancellationToken>())
        .Returns(true);

    _petRepository
        .AddAsync(Arg.Any<Pet>(), Arg.Any<CancellationToken>())
        .Returns(Task.CompletedTask);

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

    // Assert
    response.ShouldNotBeNull();
    response.ShouldBeOfType<CreatePetResponse>();
    response.PetId.ShouldNotBe(Guid.Empty);

    await _userRepository
        .Received(1)
        .ExistsAsync(Arg.Any<Expression<Func<User, bool>>>(), Arg.Any<CancellationToken>());

    await _petRepository
        .Received(1)
        .AddAsync(Arg.Any<Pet>(), Arg.Any<CancellationToken>());
}
```

Rys. 6.6 - *Test jednostkowy handlera CreatePetCommand [opracowanie własne]*

6.2. Testy integracyjne

Testy integracyjne (ang. *Integration Tests*) to rodzaj testów, które sprawdzają, czy różne komponenty systemu współdziałają poprawnie jako zintegrowana całość. W przeciwieństwie do testów jednostkowych, testy integracyjne wykorzystują rzeczywiste zależności, takie jak bazy danych, systemy plików i połączenia sieciowe. Charakteryzują się one dłuższym czasem wykonania oraz mniejszą kontrolą nad testowanym środowiskiem. Celem testów integracyjnych jest sprawdzenie czy moduły systemu potrafią ze sobą współpracować i prawidłowo się komunikować, tworząc spójnie działającą aplikację [8].

W kontekście testów integracyjnych kluczowe znaczenie ma odpowiednia izolacja środowiska testowego, szczególnie w zakresie bazy danych. Baza danych wykorzystywana w testach musi być całkowicie odseparowana od środowiska produkcyjnego, zapewniając tym samym bezpieczeństwo danych produkcyjnych oraz gwarantując powtarzalność testów.

```
public class PetManagerTestFactory : WebApplicationFactory<Api.Program>
{
    private const string AppSettings = "appsettings.test.json";
    private const string ConnectionString = "PetManagerDbTest";
    private string _dbName;

    public PetManagerTestFactory()
    {
        _dbName = $"PetManagerDbTest_{Guid.NewGuid()}";
    }

    private readonly IConfiguration _configuration = new ConfigurationBuilder()
        .AddJsonFile(AppSettings)
        .Build();

    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.ConfigureServices(services =>
        {
            var descriptor = services
                .SingleOrDefault(d => d.ServiceType == typeof(DbContextOptions<PetManagerDbContext>));

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

            var connectionString = _configuration.GetConnectionString(ConnectionString);

            services.AddDbContext<PetManagerDbContext>(options =>
            {
                options.UseNpgsql(connectionString?.Replace("PetManager_test", _dbName))
                    .EnableSensitiveDataLogging(false)
                    .UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);
            });
        });
    }
}
```

Rys. 6.7 - *Konfiguracja bazy danych dla testów integracyjnych [opracowanie własne]*

W przedstawionym projekcie wykorzystano klasę *PetManagerTestFactory* dziedziczącą po *WebApplicationFactory<Program>*, która odpowiada za konfigurację izolowanego środowiska testowego. Jak pokazano na rysunku 6.7, klasa ta implementuje mechanizm tworzenia unikalnej bazy danych dla każdego uruchomienia testów poprzez generowanie dynamicznej nazwy z wykorzystaniem identyfikatora GUID.

Zaimplementowane testy integracyjne można podzielić na dwie kategorie: testy wymagające autoryzacji oraz testy, które jej nie wymagają. Testy wymagające autoryzacji wykorzystują tokeny JWT, które są generowane podczas procesu logowania i następnie używane do uwierzytelniania kolejnych żądań.

Przykładem testu niewymagającego autoryzacji jest proces logowania użytkownika, przedstawiony na rysunku 6.8. Test ten weryfikuje poprawność procesu logowania do systemu. W pierwszym etapie testu następuje przygotowanie danych testowych - tworzona jest komenda rejestracji użytkownika, a następnie hasło użytkownika jest hashowane. Na podstawie tych danych tworzony jest nowy użytkownik, który zostaje dodany do bazy danych. W kolejnym etapie wykonywane jest żądanie POST pod adres */api/v1/users/sign-in* z danymi logowania. Ostatni etap weryfikuje, czy serwer zwrócił kod odpowiedzi *200 OK*.

```
[Fact]
public async Task Post_Sign_In_With_Valid_Credentials_Should_Return_200_Status_Code()
{
    // Arrange
    var command = _userFactory.CreateSignInCommand();
    var user = _userFactory.CreateUser(command.Email, command.Password);
    await AddAsync(user);

    // Act
    var response = await _client.PostAsJsonAsync(UserEndpoints.SignIn, command);

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

Rys. 6.8 - *Test integracyjny logowania [opracowanie własne]*

Na rysunku 6.9 przedstawiono test integracyjny dla endpointu tworzenia zwierzęcia, który wymaga autoryzacji. W fazie przygotowawczej generowane są niezbędne dane testowe oraz tworzony jest użytkownik, który następnie zostaje dodany do bazy danych. W dalszej kolejności następuje proces uwierzytelnienia, przedstawiony szczegółowo na rysunku 6.10.

Metoda *Authenticate* przyjmuje dwa kluczowe parametry: identyfikator użytkownika (*userId*) oraz jego rolę (*role*). Na podstawie tych danych generowany jest token JWT, który zostaje dodany do nagłówka autoryzacji HTTP. Mechanizm ten zapewnia, że każde kolejne żądanie HTTP będzie zawierało odpowiedni token uwierzytelniający, co jest niezbędne do uzyskania dostępu do chronionych zasobów systemu.

W fazie działania wykonywane jest żądanie HTTP typu POST do endpointu */api/v1/pets*. Ostatni etap testu skupia się na weryfikacji odpowiedzi serwera, sprawdzając czy zwrócony został kod statusu HTTP *201 Created*, co potwierdza poprawne utworzenie nowego zasobu w systemie.

```
[Fact]
public async Task Create_Pet_Given_Valid_Command_Should_Return_201_Status_Code()
{
    // Arrange
    var command = _petFactory.CreatePetCommand();
    var user = _userFactory.CreateUser();
    await AddAsync(user);
    await Authenticate(user.Id, user.Role.ToString());

    // Act
    var response = await _client.PostAsJsonAsync(PetEndpoints.CreatePet, command);

    // Assert
    response.StatusCode.ShouldBe(HttpStatusCode.Created);
    await response.Content.ReadFromJsonAsync<CreatePetResponse>().ShouldNotBeNull();
}
```

Rys. 6.9 - *Test integracyjny dodawania zwierzęcia [opracowanie własne]*

```
protected async Task Authenticate(Guid userId, string role)
{
    var token = await _authenticationManager.GenerateToken(userId, role);
    _client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);
}
```

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

7. Podsumowanie i wnioski

Zgodnie z założeniami pracy dyplomowej, zrealizowano aplikację webową wspomagającą opiekę nad zwierzętami domowymi. Warto podkreślić, że wszystkie wymagania funkcjonalne i niefunkcjonalne przedstawione w rozdziale drugim zostały w pełni wdrożone.

Do implementacji wykorzystano nowoczesny stos technologiczny opisany w rozdziale trzecim. Warstwę serwerową zrealizowano przy użyciu platformy ASP.NET w wersji 8.0 i języka programowania C#, natomiast aplikację kliencką wykonano w technologii Angular. Dane przechowywane są w bazie PostgreSQL oraz w platformie chmurowej Microsoft Azure Blob Storage.

Połączenie tych technologii zapewnia wysoką wydajność systemu, jego optymalizację oraz intuicyjny interfejs użytkownika. Wykorzystanie platformy chmurowej Azure gwarantuje stabilność aplikacji oraz możliwość jej łatwego skalowania.

Planowane kierunki rozwoju aplikacji obejmują:

- integrację z urządzeniami GPS takimi jak obroże dla efektywnego lokalizowania zwierząt, zwiększając bezpieczeństwo pupili;
- implementację aplikacji mobilnej na platformy iOS i Android udostępniając wszelkie możliwości systemu;
- integrację z Google Maps API w celu ułatwienia lokalizacji placówek weterynaryjnych wraz z podstawowymi danymi;
- uwierzytelnianie przez portale społecznościowe, ułatwiając tym samym proces rejestracji i logowania dla nowych użytkowników.

Podsumowując, zrealizowana aplikacja spełnia wszystkie założone wymagania i jest gotowa do wdrożenia produkcyjnego. Realizacja tego projektu znacząco poszerzyła wiedzę w zakresie tworzenia aplikacji webowych oraz testowania oprogramowania. Praktyczne wykorzystanie wzorców projektowych oraz implementacja zasad czystego kodu znacząco wzbogaciły umiejętności programistyczne, dając solidne podstawy do projektowania wysokiej jakości oprogramowania w przyszłości.

Bibliografia

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

[2] Zbigniew Fryźlewicz, Łukasz Leśniczek. *Usługi Microsoft Azure Programowanie aplikacji*

[3] James Gough, Daniel Bryant, Matthew Auburn. *Architektura API. Projektowanie, używanie i rozwijanie systemów opartych na API*

[4] Jennifer Greene, Andrew Stellman. *C#. Rusz głową! Wydanie III*

[5] Ian Griffiths, Matthew Adams, Jesse Liberty. *C#. Programowanie. Wydanie VI*

[6] Marcin Lis. C# Praktyczny kurs. Wydanie III

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

[8] Roy Osherove. Testy jednostkowe. Świat niezawodnych aplikacji. Wydanie II

[9] Łukasz Pasternak. *CSS3. Tworzenie nowoczesnych stron WWW*

[10] Angular [online] - dostęp 06.11.2024
https://angular.dev/overview

[11] ASP.NET Core [online] - dostęp 12.11.2024
https://learn.microsoft.com/pl-pl/aspnet/core/introduction-to-aspnet-core

[12] Co to jest OpenAPI? [online] - dostęp 06.11.2024
https://swagger.io/docs/specification/v3_0/about/

[13] Entity Framework Core [online] - dostęp 06.11.2024
https://learn.microsoft.com/pl-pl/ef/core/

[14] JSON Web Token [online] - dostęp 06.11.2024
https://jwt.io/introduction

[15] PostgreSQL [online] - dostęp 06.11.2024
https://www.postgresql.org/about/

[16] Swagger [online] - dostęp 06.11.2024
https://learn.microsoft.com/pl-pl/aspnet/core/tutorials/web-api-help-pages-using-swagger

[17] Web frameworks and technologies [online] - dostęp 06.11.2024
https://survey.stackoverflow.co/2024/technology#most-popular-technologies-language-other

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

[19] xUnit [online] - dostęp 06.11.2024
https://xunit.net/#documentation

Spis rysunków