

POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Automatyka i Robotyka (AIR)

SPECJALNOŚĆ: Przemysł 4.0 (ARP)

PRACA DYPLOMOWA
INŻYNIERSKA

Serwis internetowy wspomagający planowanie
wyprowadzania psów na spacer

A website supporting the planning of taking
dogs for a walk

AUTOR:

Amadeusz Janiszyn

PROWADZĄCY PRACĘ:

Dr inż. Mariusz Uchroński,
Katedra Automatyki, Mechatroniki
i Systemów Sterowania

OCENA PRACY:

Spis treści

1. Wstęp	4
1.1. Cel oraz zakres pracy	4
1.2. Główne założenia projektowe	5
1.3. Układ pracy	5
2. Część teoretyczna	6
2.1. Wprowadzenie do projektu	6
2.2. Proces projektowania i implementacji	6
2.3. Wymagania funkcjonalne oraz niefunkcjonalne	7
2.3.1. Wymagania funkcjonalne	7
2.3.2. Wymagania niefunkcjonalne	7
2.4. Diagram oraz przypadki użycia	8
2.4.1. Diagramy użycia	8
2.4.2. Przypadki użycia	8
2.5. Widoki, mockupy	13
2.6. Backend	13
2.7. Frontend	14
3. Część implementacyjna	16
3.1. Zbiór narzędzi	16
3.2. Trello – deklaracja zadań	16
3.3. Adobe XD – projektownie prototypów	18
3.4. Java – backend aplikacji	20
3.4.1. Architektura projektu	20
3.4.2. Zbiór oraz opis endpointów	21
3.4.3. JWT – Autoryzacja i autentykacja	25
3.4.4. Implementacja wyszukiwania po lokalizacji	26
3.4.5. Encje	27
3.5. Angular – frontend	29
3.5.1. Konfiguracja projektu	29
3.5.2. Architektura projektu frontendowego	29
3.5.3. Działanie	31
3.5.4. JWT – Autoryzacja i autentykacja	33
3.5.5. Biblioteki i moduły	34
4. Podsumowanie	37
4.1. Rozwój aplikacji	37
Literatura	39

A. Opis załączonej płyty CD/DVD	40
---	----

Rozdział 1

Wstęp

Globalny dostęp do internetu w szybkim tempie zaczyna wypierać technologie desktopowe na rzecz aplikacji internetowych. W związku z tym coraz więcej przedsiębiorstw decyduje się na taką implementację swoich rozwiązań oraz produktów.

Zaletami aplikacji webowych są minimalne wymagania ze strony potencjalnego klienta oraz prostszy sposób implementacji. Od konsumenta wymagane jest jedynie urządzenie z dostępem do internetu. Taką aplikację można uruchomić na każdym urządzeniu codziennego użytku – od komputerów stacjonarnych aż po urządzenia mobilne. Od strony implementacyjnej należy zapewnić serwer z zasobami pozwalającymi na komfortowe korzystanie z danego serwisu. Zasoby są ustalane na etapie analizy systemowej, a w trakcie rozwoju aplikacji, takie zasoby mogą być rozszerzane.

W ciągu ostatnich kilku lat znacząco zmieniło się podejście do tworzenia aplikacji internetowych. Jeszcze kilka lat temu dużą popularnością cieszyło się podejście do tworzenia aplikacji wielostanicowych (*ang. MPA - Multi-Page application*). Oznacza to, że plik HTML był generowany po stronie backendu a następnie wysyłany do użytkownika. Takie podejście charakteryzowało się ciągłą potrzebą przeładowywania strony. Obecnie w wyniku wzrostu popularności języka *JavaScript*, pojawiło się niezliczona ilość frameworków oferujących tworzenie aplikacji jednostranicowych (*ang. SPA - Single-page application*). Cała logika stoi po stronie przeglądarki i na bieżąco aktualizuje swoją zawartość. Zaletą tego rozwiązania jest zrezygnowanie z widoku, generowanego po stronie backendu co znacznie poprawia wydajność oraz szybkość aplikacji. Takie cechy serwisów są obecnie bardzo pożądane przez fakt wzrostu wymagań dotyczących pozytywnych doświadczeń użytkownika płynących z korzystania z aplikacji.

Obecnie dużą popularnością cieszy się podejście do tworzenia tzw. Single-Page Application. Czyli aplikacji, których głównym założeniem jest wysyłanie do użytkownika jednego pliku HTML, który na bieżąco podmienia swoją zawartość z poziomu przeglądarki. W trakcie korzystania z programów, strony internetowe nie są przeładowywane, co powoduje szybsze działanie aplikacji oraz pozytywnie wpływa na doświadczenia użytkowników. W przypadku starego podejścia, tzn Multi-Page Application, pliki były generowane na bieżąco po stronie serwera i wysyłane do przeglądarki co znacząco wydłużało czas ładowania strony.

1.1. Cel oraz zakres pracy

Celem pracy jest zaprojektowanie i zaimplementowanie serwisu internetowego w oparciu o nowoczesne techniki programowania, którego zadaniem jest wspomaganie planowania wyprowadzania psów na spacer. Aplikacja skonstruowana jest w oparciu o dwie główne grupy docelowych użytkowników – właścicieli oraz opiekunów.

Zakres pracy obejmuje stworzenie aplikacji przy użyciu języka Java do stworzenia serwera oraz aplikacji dostępowej stworzonej przy pomocy framework'a Angular. Serwis pobiera i zapisuje dane przy użyciu MongoDB – nowoczesnej oraz nierelacyjnej bazie danych, która w ciągu ostatnich lat zdobywa coraz większą popularność. Wykorzystano również szereg narzędzi wspomagających i usprawniających tworzenie aplikacji.

1.2. Główne założenia projektowe

- Właściciele dostają szereg funkcjonalności, pozwalających na stworzenie profilu zwierzaka oraz zaplanowanie spacerów z deklaracją godziny oraz miejsca spaceru. Możliwe jest również wystawianie opinii po spacerze w celu weryfikacji kompetencji opiekuna;
- Opiekunowie otrzymują możliwość wyszukiwania spacerów, która bazuje na lokalizacji użytkownika. Aplikacja udostępnia tygodniowy oraz miesięczny wygląd planera ze spacerami, na które użytkownik się zpisał. Opiekunowie również mogą dodawać informacje o odbytym spacerze;
- Administracja dostaje możliwość podglądu bazy użytkowników, zwierząt oraz spacerów, jak i monitorowania samej aplikacji oraz błędów, które zgłosili inni użytkownicy;

1.3. Układ pracy

Praca została podzielona na niżej wymienione rozdziały:

- Część teoretyczna, opisuje dokładnie wykorzystane technologie, techniki oraz programy, które zostały wykorzystane przy implementacji pracy;
- Część implementacyjna, skupia się na szczegółach dotyczących implementacji aplikacji oraz zawiera opis funkcjonalny;
- Podsumowanie, streszcza proces tworzenia aplikacji, oraz prezentuje dalsze możliwości rozwoju aplikacji w przyszłości;

Rozdział 2

Część teoretyczna

2.1. Wprowadzenie do projektu

Inspiracją do stworzenia serwisu internetowego, omawianego w niniejszej pracy dyplomowej, było najbliższe środowisko autora. Po wielu rozmowach zauważono, że w obecnych czasach można spotkać coraz więcej właścicieli psów, którzy przez różne, często losowe sytuacje, nie mogą sobie pozwolić na regularne spacery ze swoimi zwierzętami. Ponadto można wyróżnić pewną grupę osób, często z kręgu uczniańskiego, które mają zamiłowanie do zajmowania się zwierzętami oraz znaczająco więcej wolnego czasu, który mogłyby przeznaczyć na spacery z psami. Po przeanalizowaniu rynku zauważono, że w Polsce nie istnieje żadna specjalistyczna aplikacja, oferująca możliwość udostępniania oraz zapisywania się na spacery dla osób potrzebujących.

Głównym czynnikiem, który wpłynął na wybór oraz zastosowanie technologii internetowej, była chęć autora do zapoznania się z popularnym wśród wielu firm stosem technologicznym do tworzenia serwisów internetowych oraz dotarcie do większej ilości potencjalnych klientów.

2.2. Proces projektowania i implementacji

Proces tworzenia aplikacji obejmował kilka etapów:

- Opracowanie wymagań funkcjonalnych oraz niefunkcjonalnych;
- Opracowanie widoków aplikacji;
- Opracowanie warstwy biznesowej po stronie backendu;
- Opracowanie warstwy dostępowej po stronie frontendu;

Organizacja pracy nad projektem odgrywa kluczowe znaczenie na każdym etapie implementacyjnym. Każde stadium projektu było zarządzane przy pomocy aplikacji Trello. Jest to darmowa aplikacja, pozwalająca na skuteczne kierowanie projektem przez jedną lub wiele osób, współpracujących ze sobą. Organizacja pracy polegała na tworzeniu tablic do poszczególnych etapów. Tablice były tworzone na podstawie tych używanych w metodologii Kanban. Następnie na początku każdego etapu były planowane zadania do zrobienia i wpisywane do aplikacji. Miało to na celu większą organizację oraz kontrolę nad projektem. Wczesne zaznajomienie się ze zwinnymi metodykami zarządzania produkcją jest obecnie cenione na rynku pracy, ponieważ wiele firm z sektora IT używa ich na codzień.

2.3. Wymagania funkcjonalne oraz niefunkcjonalne

W pierwszym etapie projektowania serwisu został stworzony spis wymagań funkcjonalnych oraz niefunkcjonalnych. Miało to na celu pogrupowanie zadań na odpowiednie podetapy. Pozwoliło skupić się na kluczowych, dla danej fazy, funkcjach.

2.3.1. Wymagania funkcjonalne

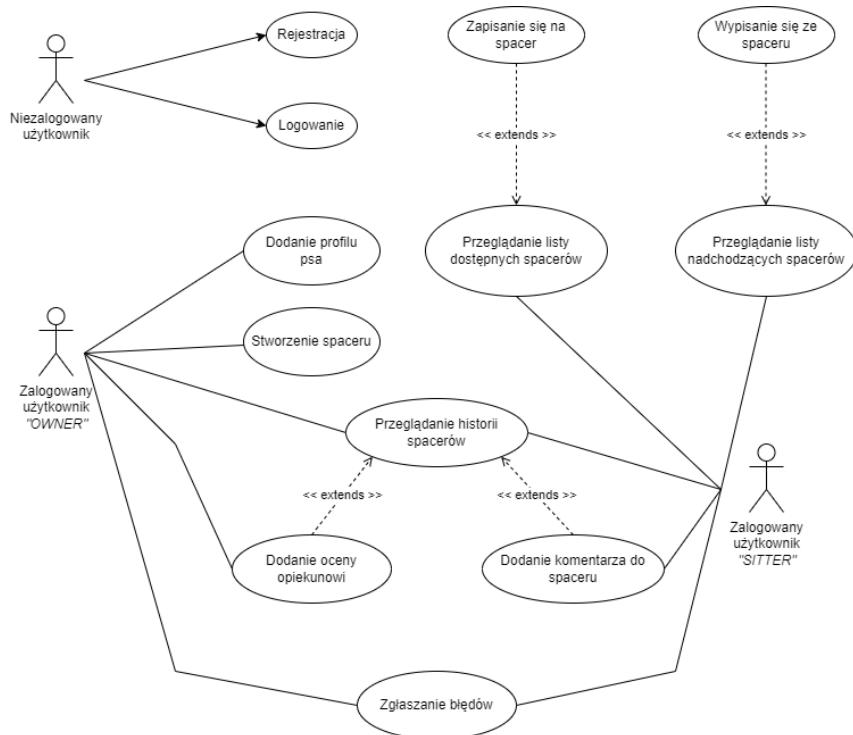
- Rejestracja nowych użytkowników z podziałem na role;
- Logowanie użytkowników do serwisu;
- Możliwość zmiany danych użytkownika – zdjęcie profilowe, dane osobowe, hasło;
- Możliwość dodania profilu swojego psa;
- Możliwość stworzenia spaceru;
- Ocena opiekunów po zakończonym spacerze;
- Przeglądanie listy dostępnych spacerów;
- Przeglądanie listy nadchodzących spacerów;
- Zapisywanie na spacery;
- Wypisywanie się ze spacerów;
- Dodawanie opinii o wyprowadzonym zwierzaków;
- Przeglądanie historii spacerów;
- Przeglądanie profili użytkowników;
- Przeglądanie profili spacerów;
- Zgłaszanie błędów w aplikacji przez użytkowników;
- Zarządzanie kontami użytkowników – blokowanie, banowanie;
- Wyświetlanie bazy danych użytkowników;
- Wyświetlanie bazy danych zwierzaków;
- Wyświetlanie bazy danych spacerów;
- System zarządzania zgłoszonymi błędami;

2.3.2. Wymagania niefunkcjonalne

- Dostęp do internetu;

2.4. Diagram oraz przypadki użycia

2.4.1. Diagramy użycia



Rys. 2.1: Diagram przypadków użycia

2.4.2. Przypadki użycia

Przypadek użycia: Rejestracja.

Aktor: Niezalogowany użytkownik.

Opis: Rejestracja w serwisie.

Warunki wstępne: Użytkownik niezalogowany, nieposiadający konta, wchodzący do serwisu po raz pierwszy.

Przebieg:

- Użytkownik kliką w odnośnik ...;
- Użytkownik zostaje przeniesiony do strony z formularzem rejestracyjnym;
- Użytkownik wypełnia dane;
 - Dane są niepoprawne – serwis informuje użytkownika o błędach, przycisk jest nieaktywny;
 - Dane są poprawne – przycisk jest aktywny i użytkownik może stworzyć konto;
- Wysłanie formularzu – użytkownik jest informowany o sukcesie bądź błędzie operacji;

Przypadek użycia: Logowanie.

Aktor: Niezalogowany użytkownik.

Opis: Logowanie do serwisu.

Warunki wstępne: Użytkownik niezalogowany, posiadający konto w serwisie.

Przebieg:

- Użytkownik wypełnia formularz logowania;
- Użytkownik kliką w przycisk *Zaloguj się*;

- W przypadku sukcesu użytkownik zostaje przekierowany do strony aplikacji;
- W przypadku błędu użytkownik jest informowany o błędzie;

Przypadek użycia: Zmiana zdjęcia profilowego.

Aktor: Zalogowany użytkownik.

Opis: Zmiana zdjęcia profilowego.

Warunki wstępne: Użytkownik zalogowany z aktywnym kontem.

Przebieg:

- Użytkownik wchodzi w zakładkę profil;
- Po najechaniu na zdjęcie profilowe klika w ikonę edycji;
- Użytkownik wgrywa nowe zdjęcie;
- Użytkownik klika w przycisk *Zmień zdjęcie*;
- Użytkownik otrzymuje informację zwrotną o statusie wykonanej operacji;

Przypadek użycia: Dodanie profilu psa.

Aktor: Zalogowany użytkownik.

Opis: Dodanie profilu zwierzaka do serwisu.

Warunki wstępne: Użytkownik zalogowany, z aktywnym kontem oraz rolą *OWNER*.

Przebieg:

- Użytkownik wchodzi w zakładkę *Dodaj spacer*;
- Użytkownik wypełnia formularz;
 - Dane są poprawne – aktywuje się submit button (przypis);
 - Dane są niepoprawne – pojawia się informacja zwrotna;
- Użytkownik klika w przycisk *Dodaj zwierzaka*;
 - W przypadku sukcesu użytkownik zostaje przekierowany do strony aplikacji;
 - W przypadku błędu użytkownik jest informowany o błędzie;

Przypadek użycia: Dodanie nowego spaceru.

Aktor: Zalogowany użytkownik.

Opis: Dodanie terminu spaceru.

Warunki wstępne: Użytkownik zalogowany, z aktywnym kontem oraz rolą *OWNER*.

Przebieg:

- Użytkownik wchodzi w zakładkę *Dodaj spacer*;
- Użytkownik wypełnia formularz;
 - Dane są poprawne – aktywuje się submit button (przypis);
 - Dane są niepoprawne – pojawia się informacja zwrotna;
- Użytkownik klika w przycisk *Dodaj zwierzaka*;
 - W przypadku sukcesu użytkownik zostaje przekierowany do strony aplikacji;
 - W przypadku błędu użytkownik jest informowany o błędzie;

Przypadek użycia: Przeglądanie historii spacerów.

Aktor: Zalogowany użytkownik.

Opis: Przeglądanie historii spacerów.

Warunki wstępne: Zalogowany użytkownik z aktywnym kontem oraz rolą *OWNER* lub *SITTER*.

Przebieg:

- Użytkownik wchodzi w zakładkę *Historia*;
- Wyświetlana jest lista spacerów w przeszłości;

Przypadek użycia: Skomentowanie spaceru.

Aktor: Zalogowany użytkownik.

Opis: Dodanie komentarza do spaceru.

Warunki wstępne: Zalogowany użytkownik z aktywnym kontem, rolą *SITTER* oraz spacerem w przeszłości.

Przebieg:

- Użytkownik wchodzi w zakładkę *Historia*;
- Użytkownik wybiera spacer, który chce skomentować;
- Użytkownik wypełnia formularz;
 - Dane są poprawne – aktywuje się submit button (przypis);
 - Dane są niepoprawne – pojawia się informacja zwrotna;
- Użytkownik kliką w przycisk *Dodaj komentarz*;
 - W przypadku sukcesu użytkownik zostaje przekierowany do strony aplikacji;
 - W przypadku błędu użytkownik jest informowany o błędzie;

Przypadek użycia: Ocena opiekuna.

Aktor: Zalogowany użytkownik.

Opis: Dodanie oceny opiekunowi.

Warunki wstępne: Zalogowany użytkownik z aktywnym kontem, rolą *OWNER* oraz zakończonym spacerem w przeszłości.

Przebieg:

- Użytkownik wchodzi w zakładkę *Historia*;
- Użytkownik wybiera spacer z opiekunem, którego chce ocenić;
- Użytkownik wypełnia formularz;
 - Dane są poprawne – aktywuje się submit button (przypis);
 - Dane są niepoprawne – pojawia się informacja zwrotna;
- Użytkownik kliką w przycisk *Dodaj ocenę*;
 - W przypadku sukcesu użytkownik zostaje przekierowany do strony aplikacji;
 - W przypadku błędu użytkownik jest informowany o błędzie;

Przypadek użycia: Przeglądanie listy dostępnych spacerów.

Aktor: Zalogowany użytkownik.

Opis: Przeglądanie listy dostępnych spacerów.

Warunki wstępne: Zalogowany użytkownik z aktywnym kontem oraz rolą *SITTER*.

Przebieg:

- Użytkownik wchodzi w zakładkę *Spacery* z aktywnym widokiem wszystkich spacerów;
- Pobierana jest lista psów, bazująca na lokalizacji;
- Lista spacerów wyświetlana jest użytkownikowi;

Przypadek użycia: Zapisanie się na spacer.

Aktor: Zalogowany użytkownik.

Opis: Zapisanie się na spacer.

Warunki wstępne: Zalogowany użytkownik z aktywnym kontem, rolą *SITTER* oraz spacer dostępny w przeszłości z wolnym miejscem do zapisu.

Przebieg: Scenariusz A: Użytkownik znajduje się w zakładce Spacery

- Użytkownik kliką w przycisk *Zapisz się*;
- Użytkownik jest informowany stosownym komunikatem;

Scenariusz B: Użytkownik znajduje się w profilu spaceru

- Użytkownik kliką w przycisk *Zapisz się*;
- Użytkownik jest informowany stosownym komunikatem;

Przypadek użycia: Wypisanie się ze spaceru.

Aktor: Zalogowany użytkownik.

Opis: Wypisanie się ze spaceru.

Warunki wstępne: Zalogowany użytkownik z aktywnym kontem, rolą *SITTER* oraz spacer w przyszłości na który użytkownik jest już zapisany.

Przebieg: Scenariusz A: Użytkownik znajduje się w zakładce Spacery

- Użytkownik klika w przycisk *Wypisz się*;
- Użytkownik jest informowany stosownym komunikatem;

Scenariusz B: Użytkownik znajduje się w profilu spaceru

- Użytkownik klika w przycisk *Wypisz się*;
- Użytkownik jest informowany stosownym komunikatem;

Przypadek użycia: Przeglądanie listy nadchodzących spacerów.

Aktor: Zalogowany użytkownik.

Opis: Przeglądanie listy nadchodzących spacerów na które użytkownik jest zapisany.

Warunki wstępne: Zalogowany użytkownik z aktywnym kontem, rolą *SITTER* oraz co najmniej jednym spacerem na który jest zapisany.

Przebieg:

- Użytkownik wchodzi w zakładkę *Spacery*;
- Użytkownik przełącza widok na *Nadchodzące*;
- Lista nadchodzących spacerów jest wyświetlana użytkownikowi;

Przypadek użycia: Przeglądanie terminarza ze spacerami.

Aktor: Zalogowany użytkownik.

Opis: Przeglądanie terminarza ze spacerami, na które użytkownik jest zapisany.

Warunki wstępne: Zalogowany użytkownik z aktywnym kontem oraz rolą *SITTER*.

Przebieg:

- Użytkownik wchodzi w zakładkę *Terminarz*;
- Użytkownik może wybrać tryb wyświetlania:
 - Miesiąc – widok miesiąca ze zdjęciami psów oraz odnośnikami do profilu spaceru;
 - Tydzień – widok tygodniowy z podstawowymi informacjami o danym spacerze oraz odnośnikiem do profilu spaceru;

Przypadek użycia: Przeglądanie profilu użytkownika.

Aktor: Zalogowany użytkownik.

Opis: Przeglądanie profilu użytkownika.

Warunki wstępne: Zalogowany użytkownik z aktywnym kontem oraz dowolną rolą.

Przebieg:

- Użytkownik klika w odnośnik przekierowujący na profil innego użytkownika;
- Wyświetlają się szczegółowe informacje o użytkowniku;

Przypadek użycia: Przeglądanie profilu psa.

Aktor: Zalogowany użytkownik.

Opis: Przeglądanie profilu psa.

Warunki wstępne: Zalogowany użytkownik z aktywnym kontem oraz dowolną rolą.

Przebieg:

- Użytkownik klika w odnośnik przekierowujący na profil psa;
- Wyświetlają się szczegółowe informacje;

Przypadek użycia: Przeglądanie profilu spaceru.

Aktor: Zalogowany użytkownik.

Opis: Przeglądanie profilu spaceru.

Warunki wstępne: Zalogowany użytkownik z aktywnym kontem oraz dowolną rolą.

Przebieg:

- Użytkownik kliką w odnośnik przekierowujący na profil spaceru;
- Wyświetlają się szczegółowe informacje o spacerze;

Przypadek użycia: Zgłaszcenie błędów.

Aktor: Zalogowany użytkownik.

Opis: Zalogowany użytkownik.

Warunki wstępne: Zalogowany użytkownik z aktywnym kontem oraz dowolną rolą.

Przebieg:

- Użytkownik kliką w ikonę błędu w prawym dolnym rogu;
- Użytkownik wypełnia formularz;
 - Dane są poprawne – aktywuje się submit button (przypis);
 - Dane są niepoprawne – pojawia się informacja zwrotna;
- Użytkownik kliką w przycisk *Zgłoś błąd*;
 - W przypadku sukcesu użytkownik zostaje przekierowany do strony aplikacji;
 - W przypadku błędu użytkownik jest informowany o błędzie;

Przypadek użycia: Przeglądanie listy wszystkich użytkowników.

Aktor: Zalogowany użytkownik.

Opis: Przeglądanie bazy użytkowników.

Warunki wstępne: Zalogowany użytkownik z rolą *ADMIN*.

Przebieg:

- Użytkownik wchodzi w zakładkę *Użytkownicy*;
- Pojawia się lista wszystkich użytkowników zarejestrowanych w serwisie;

Przypadek użycia: Przeglądanie listy wszystkich spacerów.

Aktor: Zalogowany użytkownik.

Opis: Przeglądanie bazy spacerów.

Warunki wstępne: Zalogowany użytkownik z rolą *ADMIN*.

Przebieg:

- Użytkownik wchodzi w zakładkę *Zwierzęta i spacy*;
- Użytkownik wybiera widok spacerów;
- Pojawia się lista wszystkich spacerów dodanych w serwisie;

Przypadek użycia: Przeglądanie listy wszystkich psów .

Aktor: Zalogowany użytkownik.

Opis: Przeglądanie bazy zwierząt.

Warunki wstępne: Zalogowany użytkownik z rolą *ADMIN*.

Przebieg:

- Użytkownik wchodzi w zakładkę *Zwierzęta i spacy*;
- Użytkownik wybiera widok zwierząt;
- Pojawia się lista wszystkich psów dodanych w serwisie;

Przypadek użycia: Przeglądanie aktywności użytkowników.

Aktor: Zalogowany użytkownik.

Opis: Przeglądanie aktywności użytkowników.

Warunki wstępne: Zalogowany użytkownik z rolą *ADMIN* .

Przebieg:

- Użytkownik wchodzi w zakładkę *Aktywność*;

- Pojawia się lista aktywności użytkowników na serwisie;

Przypadek użycia: Przeglądanie listy zgłoszonych błędów.

Aktor: Zalogowany użytkownik.

Opis: Przeglądanie listy zgłoszonych błędów przez użytkowników.

Warunki wstępne: Zalogowany użytkownik z rolą *ADMIN*.

Przebieg:

- Użytkownik wchodzi w zakładkę *Błędy*;
- Pojawiają się lista błędów podzielona na 3 grupy – nowe, w trakcie realizacji, naprawione;

Przypadek użycia: Przeglądanie dashboardu (przypis).

Aktor: Zalogowany użytkownik.

Opis: Przeglądanie dashboardu.

Warunki wstępne: Zalogowany użytkownik z rolą *OWNER* lub *SITTER*.

Przebieg:

- Użytkownik znajduje się w zakładce dashboard;
- Pojawia się lista nadchodzących spacerów oraz powiadomień;

2.5. Widoki, mockupy

Kluczowym elementem przy projektowaniu serwisów internetowych jest stworzenie prototypu aplikacji. Prototypowanie przyszłej aplikacji pozwala w krótkim czasie zaprojektować atrakcyjny wygląd, zgodny z technikami UI/UX (*ang. User Interface / User Experience*).

W trakcie etapu projektowania zdecydowano się na użycie *Adobe XD*. Jest to popularne narzędzie do prototypowania – głównie na platformy internetowe oraz mobilne. Program pozwala na projektowanie makiet aplikacji poprzez wykorzystanie podstawowych kształtów oraz ich właściwości. Zaletą specjalistycznych narzędzi do prototypowania jest zastosowanie zbioru właściwości, które służą do nadawania poszczególnym elementom pożądanego wyglądu, zazwyczaj z często używanego narzędzia do stylizacji plików HTML – kaskadowych arkuszy stylów. Takie rozwiązanie pozwala w błyskawiczny sposób przenieść prototypy do docelowego projektu.

2.6. Backend

Każda aplikacja internetowa posiada swój silnik, który jest określany jako *backend*. Jest to logika biznesowa, stojąca po stronie serwera. Odpowiada za pobieranie oraz przetwarzanie danych, komunikację z bazą danych oraz w większym stopniu za bezpieczeństwo całej aplikacji. Aplikacje backendowe udostępniają zbiór funkcjonalności, inaczej zwane REST API (*ang. ...*), które są wykorzystywane do transferu danych między zapleczem technicznym a warstwą dostępową aplikacji internetowej.

W omawianej aplikacji do implementacji backendu wykorzystano język *Java* - popularny język obiektowy oparty na klasach, charakteryzujący się kodem, który po komplikacji można uruchomić na każdej platformie. Niezbędny był również framework *Spring Boot*, dzięki któremu możliwe było stworzenie rozbudowanego API aplikacji. *Spring Boot* jest narzędziem opartym o framework *Spring*, zapewniającym dodatkową konfigurację oraz kontener aplikacji. Takie zastosowanie pozwala na szybsze tworzenie aplikacji bez konieczności dodatkowej konfiguracji. Użyty framework korzysta z mechanizmu wstrzykiwania zależności (*ang. Dependency Injection*). *Dependency injection* odpowiada za tworzenie i dostarczanie obiektów w odpowiednim momencie do danej klasy. Obecnie *Java* w połączeniu ze *Spring Boot*em jest powszechnie używanym narzędziem wśród wielu firm na całym świecie.

Do tworzenia warstw biznesowych przy wykorzystaniu wyżej wymienionch narzędzi, niezbędnym elementem jest zbiór zależności. Pozwalają rozszerzyć bazowe funkcjonalności o nowe, skonkretyzowane metody i rozwiązania. W przypadku aplikacji, która jest tematem niniejszej pracy wykorzystano następujące zależności:

- **Spring Web** – wprowadza do projektu podstawowe funkcje integracyjne, inicjalizuje kontener IoC, zorientowuje kontekst aplikacji na sieć oraz dodaje klienta HTTP.
- **Spring Security** – konfigurowalna platforma zapewniająca kontrolę nad dostępem do aplikacji. Stanowi standard zabezpieczeń oraz pozwala na uwierystelnianie użytkowników.
- **JSON Web Token** – popularne rozwiązanie do uwierystelniania użytkowników. Wykorzystywany jako token autoryzacyjny, generowany po stronie backendu i przechowywany po stronie frontendu, umożliwia na dostęp do zasobów API aplikacji.
- **MongoDB Driver** – zależność umożliwiająca skonfigurowanie aplikacji z bazą danych. Udostępnia szereg metod umożliwiający wykonywanie operacji z przetwarzaniem danych w bazie.
- **SwaggerUI** – dostarcza wizualną reprezentację oraz dostęp do punktów końcowych (*ang. endpoint*) aplikacji. Umożliwia na testowanie zaimplementowanych funkcjonalności bezpośrednio po stronie backendu.
- **Lombok** – jest procesorem adnotacji, umożliwiającym na generowanie kodu przed komilacją po zastosowaniu odpowiedniej adnotacji nad klasą. Wykorzystywany jest przy obiektach do generowania powtarzalnych metod takich jak konstruktory, gettery oraz settery.

W trakcie implementacji technicznej strony aplikacji niezbednym etapem jest wybór bazy danych dla serwisu. Obecnie od wielu lat wykorzystywane są relacyjne bazy danych. Jednak w ciągu ostatnich lat można zauważać rosnącą popularność baz nierelacyjnych. Wśród nich w czołówce plasuje się MongoDB. Charakterystycznymi cechami omawianej bazy danych jest zrezygnowanie z relacji między kolekcjami, operacja na dokumentach oraz przechowywanie danych w formacie BSON - jest to format JSON w postaci binarnej. Mongo, tak jak wiele innych baz NoSQL nie posiada żadnych reguł implementacyjnych co pozwala na dowolność w kwestii przechowywania danych. Jest to również nowoczesne podejście do modelowania baz danych. Dlatego też zdecydowano się na zastosowanie MongoDB jako bazy danych dla omawianej aplikacji.

Warstwa biznesowa aplikacji została zaimplementowana przy użyciu języka Java oraz frameworka Spring. Połączenie tych dwóch narzędzi pozwala stworzyć rozbudowany silnik aplikacji zwany API (*ang. Application Programming Interface*) Java jest wysokopoziomowym, obiektowym językiem, często stosowanym przez firmy IT do budowania warstw backendowych lub aplikacji mobilnych.

2.7. Frontend

Do komunikacji między użytkownikami a backendem używa się warstw dostępowych, znanych również w środowisku programistycznym jako frontend. Frontend służy do odbierania danych od użytkowników, przetwarzania ich do dopowiedniego formatu, wymaganego przez API oraz wysyłanie danych do backendu. W obecnych czasach dużą popularnością cieszą się frameworki oparte na języku JavaScript - języku skryptowym który umożliwia wprowadzenie funkcjonalności do plików HTML. Takie frameworki pozwalają na tworzenie aplikacji w technice SPA.

Frontend został zaimplementowany przy wykorzystaniu frameworku *Angular*, który obecnie znajduje się w czołówce popularności. Pozwala na tworzenie wydajnych aplikacji SPA. Charakterystyczną cechą omawianego frameworka jest wykorzystanie komponentów – mogą być

składowymi lub definiować całe widoki z dołączoną logiką do obsługi danych. Angular wykorzystuje również mechanizm dependency injection, który jest niezbędny do tworzenia modułowych komponentów, które można wielokrotnie używać. Znaczącą rolę w działaniu aplikacji pełnią moduły. Są to zarówno zewnętrzne biblioteki, posiadające szereg rozwiązań implementacyjnych oraz komponenty tworzone przez programistę. W trakcie implementacji warstwy dostępowej zastosowano zbiór zewnętrznych modułów oraz bibliotek, zapewniających prawidłowe działanie aplikacji.

- **BrowserModule** – odpowiada za uruchamianie aplikacji w przeglądarce.
- **AppRoutingModule** – dodaje możliwość nawigacji z uwzględnieniem odpowiednich adresów URL.
- **ReactiveFormsModule** – odpowiada za tworzenie reaktywnych formularzy.
- **HttpClientModule** – pozwala na komunikację poprzez protokoły HTTP z API.
- **RxJS** – biblioteka wprowadzająca możliwość programowania reaktywnego w aplikacji.

Doświadczenia wizualne płynące z korzystania z serwisów internetowych są w obecnych czasach niezwykle istotne – można by założyć, że atrakcyjność wyglądu góruje nad funkcjonalnościami aplikacji. Dlatego też bardzo ważnym elementem przy tworzeniu frontendowych aplikacji jest zadbanie o odpowiedni wygląd aplikacji poprzedzone dołożenie do projektu arkuszy stylów. Najpopularniejszym i najczęściej używanym obecnie jest *CSS*. Pozwala w prosty sposób tworzyć selektory klas i stylizować poszczególne elementy składające się na cały widok. W trakcie implementacji makiet widoków zdecydowano się na użycie bardziej rozbudowanego narzędzia do tworzenia arkuszy stylów – *SCSS*. Posiada ten sam zbiór właściwości które można stylizować, natomiast rozszerza bazową wersję o możliwość definiowania zmiennych oraz tworzenie zagnieżdżonych selektorów. Takie dodatkowe funkcjonalności pozwalają tworzyć mniej rozbudowane, prostsze arkusze stylów, które w znaczący sposób są podatne na późniejsze modyfikacje.

Rozdział 3

Część implementacyjna

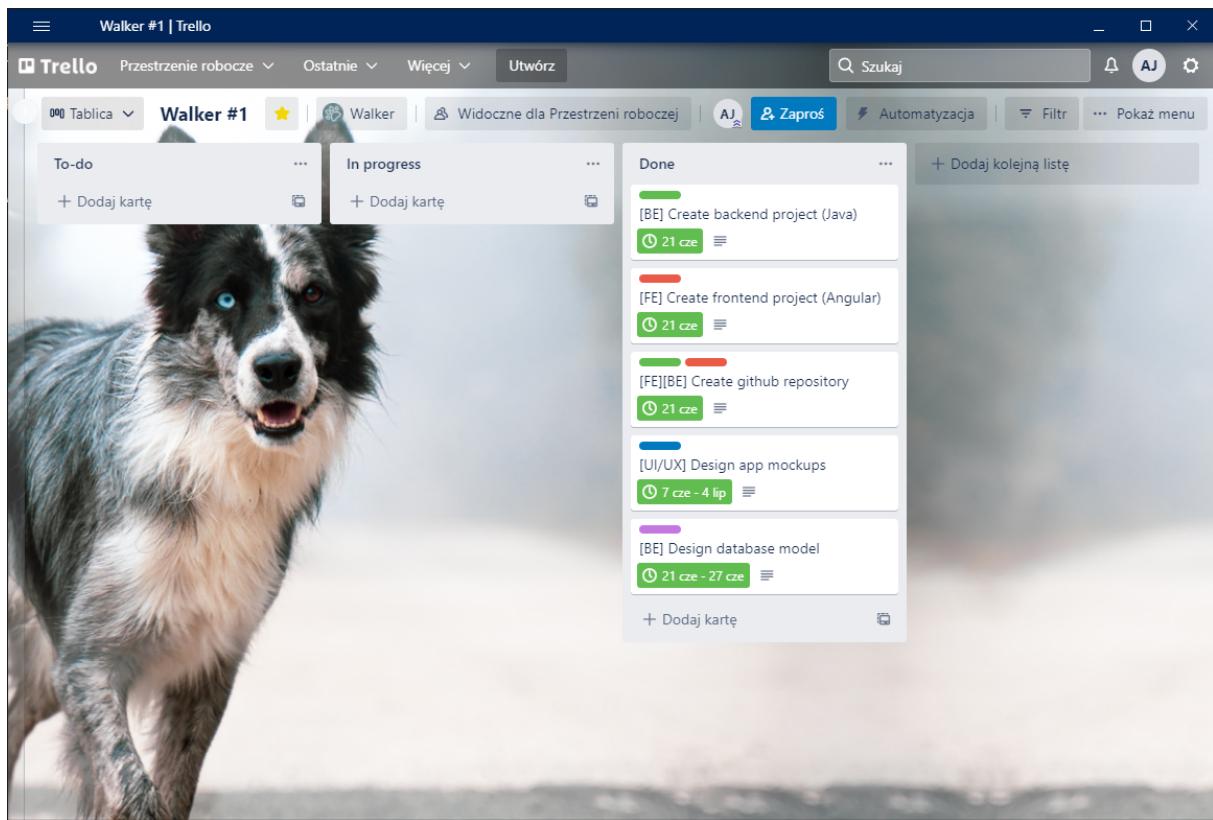
3.1. Zbiór narzędzi

Implementacja robudowanego projektu nie była by możliwa gdyby nie zbiór narzędzi które w znaczy sposób ułatwiły i przyspieszyły cały proces.

- Trello – opracowanie zadań do poszczególnych etapów;
- Adobe XD – prototypowanie aplikacji;
- Visual Studio Code – środowisko programistyczne;
- Java 11 + Spring Boot – backend;
- MongoDB – baza danych;
- Angular 12 – frontend;
- GitHub – repozytorium kodu.

3.2. Trello – deklaracja zadań

Przed rozpoczęciem pracy nad danym etapem, deklarowane były zadania przy użyciu programu *Trello*. W każdym z etapów, w szczególności implemetacyjnych, starannie były grupowane opracowywane funkcje. Następnie były one rozkładane na podzadania, które ostatecznie deklarowano w programie. Takie rozwiązanie pozwoliło na implementację funkcjonalności powiązanych ze sobą w tym samym etapie. Każdy z etapów trwał mniej więcej dwa tygodnie.



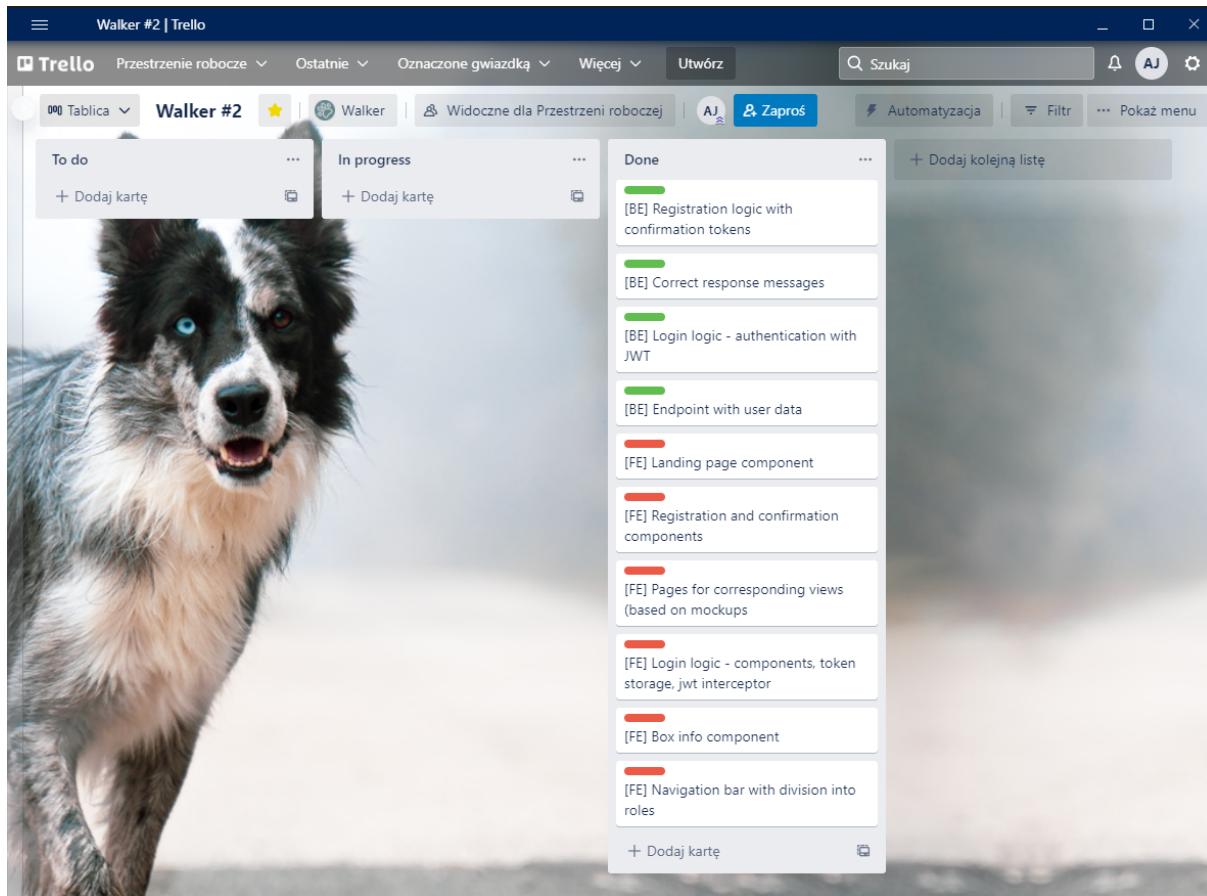
Rys. 3.1: Trello - tablica do pierwszego etapu

Do deklarowanych zadań dołączano odpowiednią etykietę, opis oraz listę podzadań, które były niezbędne przy danej funkcjonalności. Odpowiednie etykietowanie pozwoliło rozróżnić poszczególne zadania między sobą, a opis zawierał zaplanowane przez autora przypadki użycia danej funkcji. Opisy etykiet:

- Zielony – zadania dotyczące backendu;
- Czerwony – zadania dotyczące frontendu;
- Niebieski – zadania dotyczące projektu UI/UX aplikacji;
- Fioletowy – zadania dotyczące bazy danych
- Żółty – zadania oznaczone jako bugi, problemy w programie

Łącznie powstały cztery tablice, różniące się ilością oraz stopniem skomplikowania zadań.

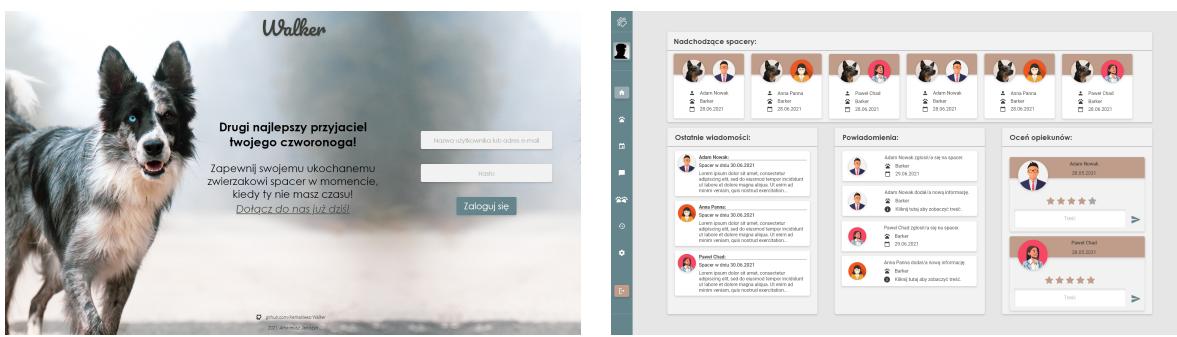
1. **Walker #1** – etap pierwszy, skupiający się na inicjalizacji projektu, stworzenie prototypów aplikacji oraz zamodelowanie bazy danych;
2. **Walker #2** – etap drugi, skupiający się na implementacji rejestracji oraz logowania użytkowników, oraz podstawowych komponentów aplikacji frontendowej między innymi pasiek nawigacyjny, strona tytułowa, strona główna;
3. **Walker #3** – etap trzeci, skupiający się na podstawowych funkcjonalnościach serwisu – tworzenie profilu psa, spaceru, pobieranie listy dostępnych spacerów, zapisywanie oraz wypisywanie się ze spaceru. Ponad to dodano możliwość edycji danych użytkownika;
4. **Walker #4** – etap czwarty, skupiający się na implementacji oceny opiekunów oraz psów. Dodano również panel administracyjny do zarządzania aplikacją.



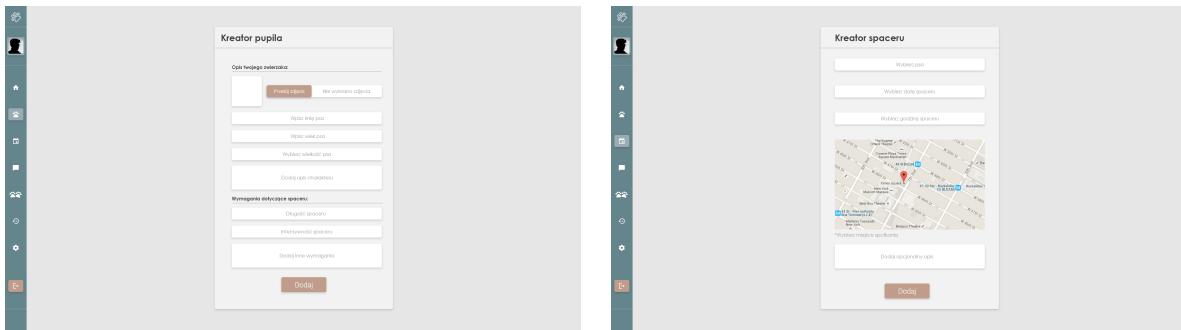
Rys. 3.2: Trello - tablica do drugiego etapu

3.3. Adobe XD – projektownie prototypów

Etap projektowania rozpoczęto od wyboru odpowiedniego zdjęcia przewodniego aplikacji, dobraniu palety kolorystycznej oraz stworzeniu logo serwisu. Zdjęcie przewodnie zostało znalezione na stronie z darmowymi zdjęciami. Na podstawie tego zdjęcia wygenerowano paletę kolorów, która została wykorzystana najpierw do prototypowania, a następnie już w samej aplikacji.

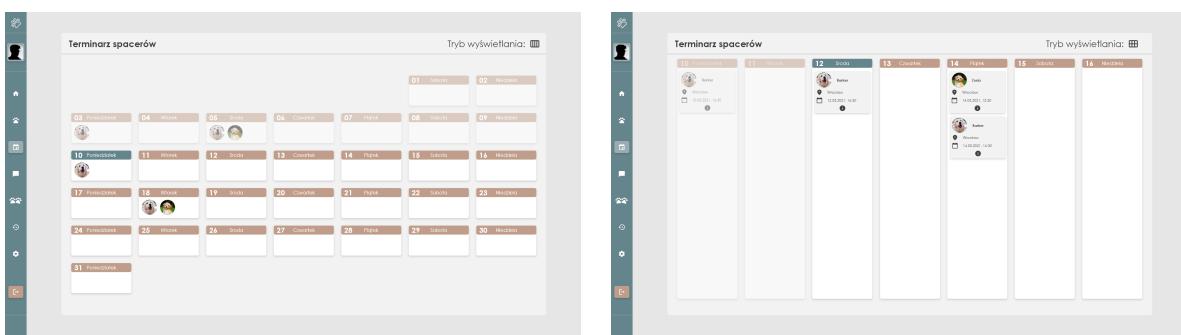


Rys. 3.3: Przykładowe prototypy aplikacji – widok startowy, widok dashboardu

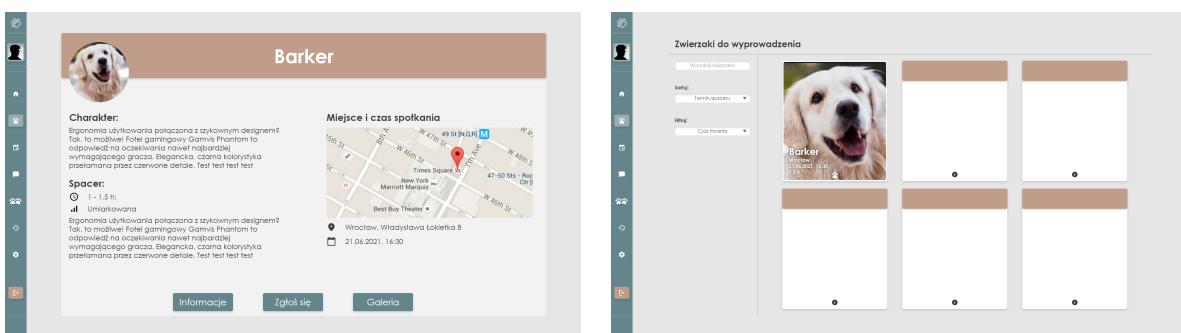


Rys. 3.4: Przykładowe prototypy aplikacji – widok startowy, widok dashboardu

Przy projektowaniu widoków posłużono się podstawowymi kształtami, głównie różnego rodzaju prostokątami oraz polami tekstowymi. Zastosowanie podstawowych właściwości tych elementów umożliwiło tworzenie atrakcyjnych komponentów, które były następnie rozmieszane na poszczególnych widokach. Cały proces miał na celu stworzenie makiet aplikacji, które cechowały by się atrakcyjnym wyglądem oraz umożliwiły by wizualizację założonych funkcjonalności. Na podstawie prototypów możliwe było również określenie maskowej długości znaków, które byłyby wyświetlane już w domyślnej aplikacji. Pozwoliło to na dostosowanie pól tekstowych tak, aby w dużym stopniu integrowały się z całym wyglądem aplikacji.



Rys. 3.5: Przykładowe prototypy aplikacji – widok startowy, widok dashboardu



Rys. 3.6: Przykładowe prototypy aplikacji – widok startowy, widok dashboardu

3.4. Java – backend aplikacji

Proces tworzenia warstwy logicznej rozpoczęło skonfigurowanie pliku *application.properties*. Przechowuje on dane w postaci klucz - wartość, które służą między innymi do konfiguracji różnych składowych aplikacji np. połączenia z bazą danych lub konfiguracji przesyłanych plików. Można również deklarować dane, które można wykorzystać następnie w aplikacji. W taki sposób można zdefiniować globalne, stałe wartości, które można wykorzystać w trakcie pisania programu.

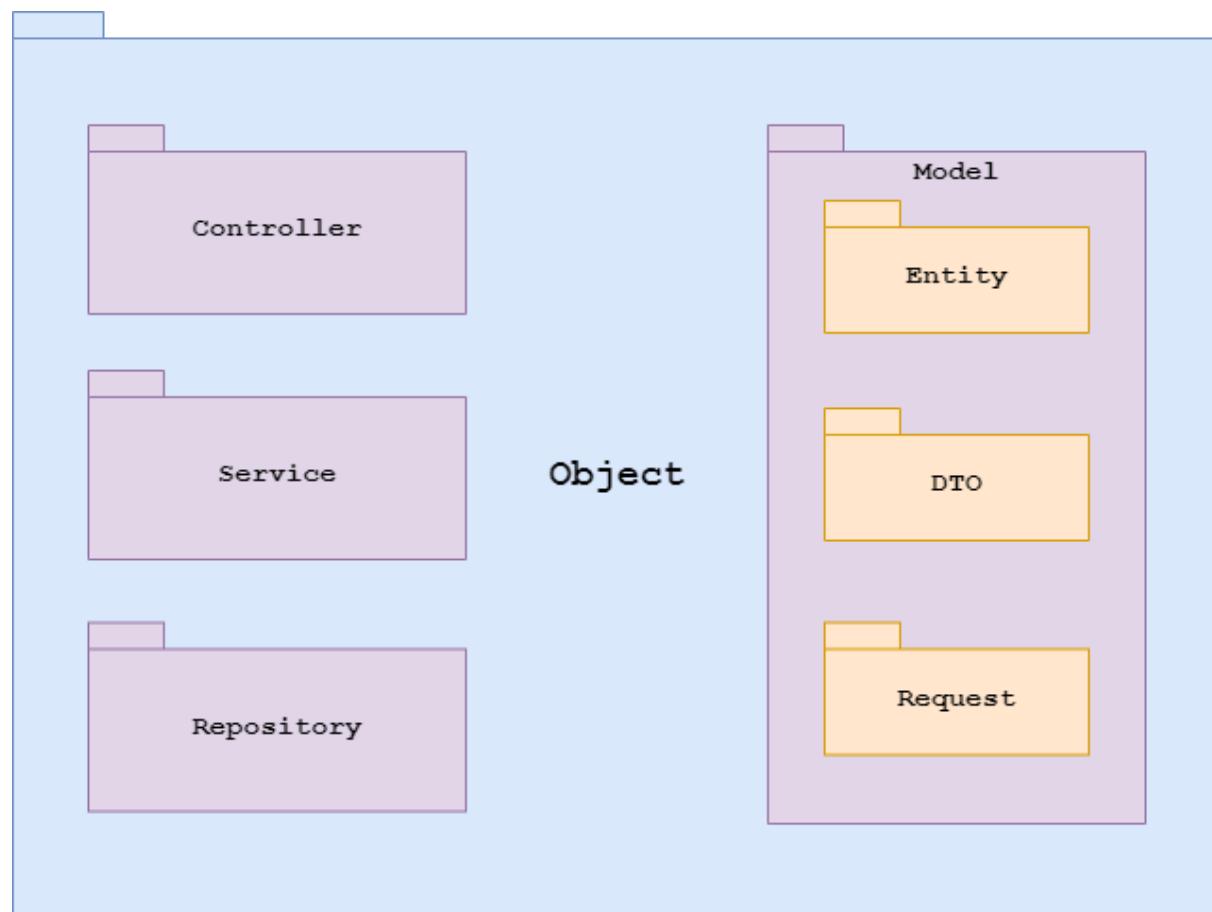
```

1 spring.data.mongodb.database=walker
2 spring.data.mongodb.host=localhost
3 spring.data.mongodb.port=27017
4
5 jwtSecret=a43b97a5-4d2a-47bc-b74f-e677ebed35db
6 jwtExpiration=3600000
7
8 spring.servlet.multipart.max-file-size=10MB
9 spring.servlet.multipart.max-request-size=10MB
10 spring.servlet.multipart.enabled=true

```

3.4.1. Architektura projektu

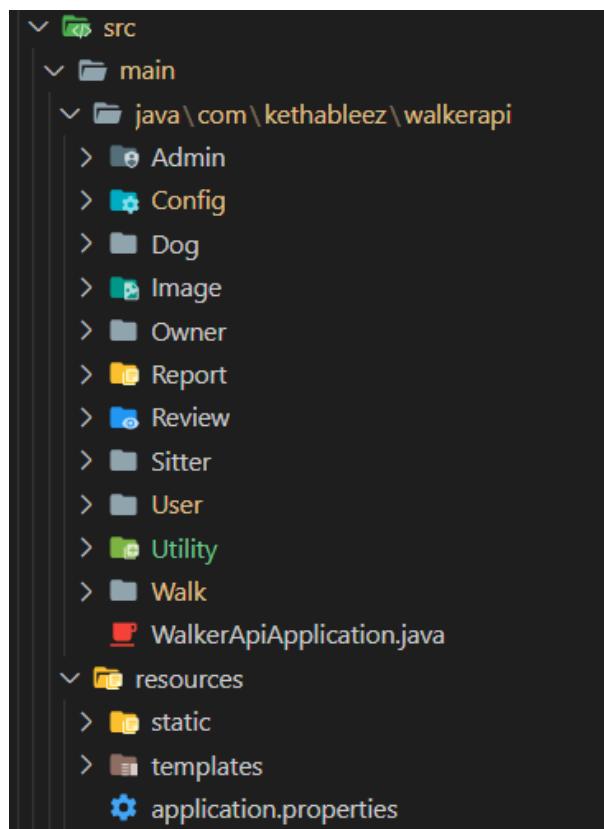
Kolejnym, ważnym krokiem jest wykorzystanie odpowiedniej architektury projektu, pozwalającej na zachowanie porządku oraz grupowanie plików na postawie implementowanej logiki.



Rys. 3.7: Szczegółowa architektura - reprezentacja podfolderów

Postanowiono na podzielenie głównych folderów ze względu na konkretne obiekty. W folderach głównych znajdują się podfoldery odpowiedzialne za przechowywanie:

- *Controller* – przechowuje plik kontrolera, który udostępnia endpointy;
- *Service* – przechowuje plik serwisu, który posiada główną logikę związaną z danym obiektem;
- *Repository* – przechowuje plik repozytorium obiektu potrzebnego do komunikacji z bazą danych;
- *Model*:
 - *Entity* – obiekt rzutowany bezpośrednio na bazę danych;
 - *DTO* – obiekt wysyłany do aplikacji dostępowej, bardziej rozbudowana encja o dodatkowe pola;
 - *Request* – obiekt przesyłany bezpośrednio z aplikacji dostępowej po wypełnieniu formularzu;



Rys. 3.8: Architektura projektu backendowego

Realna architектura projektu zawiera dodatkowo folder konfiguracyjny. Znajdują się w nim pliki odpowiedzialne za konfigurację wykorzystanych zależności. Ponadto zdecydowano się na stworzenie folderu o nazwie *Utility*, zawierającego narzędzia oraz funkcje wspomagające działanie aplikacji, na przykład funkcjonalność związaną z wyszukiwaniem spacerów po lokalizacji użytkownika.

3.4.2. Zbiór oraz opis endpointów

Punkty końcowe, znane w środowisku programistycznym też jako endpointy, wystawiane po stronie backendu służą do komunikacji z aplikacją dostępową poprzez protokół HTTP oraz jego metody. Do podstawowych metod można zaliczyć:

- *GET* – pobieranie zasobu;
- *POST* – przesyłanie danych, często w ramach tworzenia nowego obiektu;
- *PUT* – przesyłanie danych, często w ramach edycji istniejącego już obiektu;
- *DELETE* – usuwanie zasobu.

address-controller Address Controller		
GET	/address/getAddress/{postCode} getAddress	🔒
POST	/address/uploadConfig uploadConfigFile	🔒
admin-controller Admin Controller		
GET	/admin/activities getActivities	🔒
POST	/admin/ban/{userId} banUser	🔒
POST	/admin/block/{userId} blockUser	🔒
GET	/admin/dogs getDogs	🔒
POST	/admin/report/{reportId}/{status} changeReportStatus	🔒
GET	/admin/reports/{status} getReportsByStatus	🔒
POST	/admin/unban/{userId} unbanUser	🔒
POST	/admin/unblock/{userId} unblockUser	🔒
GET	/admin/users getUsers	🔒
GET	/admin/walks getWalks	🔒

Rys. 3.9: Dokumentacja - AddressController, AdminController

Address Controller

Kontroler ten odpowiada za operacje związane z wyszukiwaniem adresu na podstawie kodu pocztowego. Ta funkcjonalność jest używana przy wyświetlaniu listy dostępnych spacerów na podstawie lokalizacji użytkownika, a dokładniej kodu pocztowego, który jest deklarowany przy rejestracji.

- *uploadConfig* odpowiada za załadowanie pliku konfiguracyjnego, który jest odpowiednio przygotowanym plikiem, zawierającym zbiór województw, powiatów oraz ich unikalnych kodów. Taki plik jest przerabiany na encje, które następnie są ładowane do bazy danych. Pozwala to na wyszukiwanie odpowiedniej lokalizacji z poziomu backendu.
- *getAddress* odpowiada za otrzymanie kodu związanego z powiatem na podstawie kodu pocztowego. Ta funkcjonalność opiera się o darmowe API Poczty Polskiej, pozwalające na otrzymanie odpowiedzi w formacie JSON z danymi o miejscowości po podaniu kodu pocztowego.

Admin Controller

Udostępnia funkcje potrzebne na zarządzanie aplikacją przez administrację. Pozwala na blokowanie, banowanie oraz cofanie restrykcji na użytkownikach (endpoints: *block*, *ban*, *unblock*, *unban*). Umożliwia pobieranie całej listy użytkowników, aktywności użytkowników, psów oraz spacerów z bazy danych. Dostarcza również metody związane z zarządzeniem zgłoszonymi błędami.

auth-controller	Auth Controller
PUT	/auth/confirm/{token} confirmUser
POST	/auth/login login
POST	/auth/register registerUser
POST	/auth/register/05da579b-cafe-4395-8eeb-88826dfd6cc9 registerAdmin
dog-controller	Dog Controller
GET	/dog/{id} getDogInfo
POST	/dog/create createDog
DELETE	/dog/delete_dog/{id} deleteDog

Rys. 3.10: Dokumentacja - AuthController, DogController

Auth Controller

Kontroler odpowiada za funkcjonalności związane z autoryzacją użytkowników. Pozwala na rejestrację oraz logowanie użytkowników. Szczegóły zostały opisane w podrozdziale 3.4.3. *JWT – Autoryzacja i autentykacja*.

Dog Controller

Odpowiada za operacje związane z obiektem *Dog*. Pozwala tworzyć nowy profil psa oraz pobierać szczegółowe informacje o danym zwierzaku po podaniu jego numeru ID.

image-controller	Image Controller
GET	/image/dog/{dogId}/{filename} getDogImage
POST	/image/dog/upload/{dogId} uploadDogPhoto
GET	/image/review/d/{dogId}/{filename} getDogReviewImage
GET	/image/review/r/{reviewId} getReviewImage
GET	/image/review/u/{sitterId}/{filename} getUserReviewImage
POST	/image/review/upload/{reviewId} uploadDogReviewPhoto
GET	/image/user/{userId}/{filename} getUserImage
POST	/image/user/upload/{userId} uploadUserPhoto
operation-handler	Operation Handler
owner-controller	Owner Controller
GET	/owner/dogs getDogs
GET	/owner/history getOwnerHistory
GET	/owner/images getImages
GET	/owner/ownerData getOwnerData
GET	/owner/ownerData/{username} getOwnerData
GET	/owner/walks getWalks

Rys. 3.11: Dokumentacja - ImageController, OwnerController

Image Controller

Udostępnia szereg endpointów pozwalających na zarządzanie zdjęciami. Umożliwia zapisywanie poszczególnych rodzajów zdjęć - użytkowników, psów oraz opinii, oraz otrzymywanie takich zdjęć z bazy danych.

Owner Controller

Kontroler odpowiada za wyświetlanie danych związanych z użytkownikami z rolą OWNER. Pozwala na wyświetlanie danych o właściwemu, jego historii, posiadanych psach oraz wszystkich zdjęć, które zostały dodane do jego psów.

The screenshot shows two sections of a Swagger UI interface:

- review-controller** (Review Controller):
 - GET /review/dog/{dogId}** getDogReviews (blue button)
 - POST /review/dog/add** addDogReview (green button)
 - GET /review/sitter/{sitterId}** getUserReviews (blue button)
 - POST /review/sitter/add** addSitterReview (green button)
- sitter-controller** (Sitter Controller):
 - GET /sitter/history** getWalkHistory (blue button)
 - GET /sitter/images** getImages (blue button)
 - GET /sitter/reviews** getReviews (blue button)
 - GET /sitter/sitterData** getSitterData (blue button)
 - GET /sitter/sitterData/{username}** getSitterData (blue button)
 - GET /sitter/walks** getSitterWalks (blue button)

Rys. 3.12: Dokumentacja - ReviewController, SitterController

Review Controller

Udostępnia możliwość dodawania opinii do psów oraz opiekunów oraz pobieranie listy komentarzy.

Sitter Controller

Kontroler odpowiada za wyświetlanie danych związanych z użytkownikami z rolą SITTER. Pozwala na wyświetlanie danych o opiekunie, jego historii, opiniach dodanych przez innych użytkowników oraz wszystkich dostępnych zdjęciach.

user-controller User Controller		
GET	/user/{id} getUserInfo	🔒
GET	/user/all getAll	🔒
PUT	/user/change_avatar changeAvatar	🔒
PUT	/user/change_data changeData	🔒
PUT	/user/change_description changeDescription	🔒
PUT	/user/change_password changePassword	🔒
GET	/user/get_data getData	🔒
GET	/user/get_data/{username} getData	🔒
GET	/user/notifications getUserNotifications	🔒
POST	/user/notifications/{notificationId}/markAsRead markNotificationAsRead	🔓
POST	/user/report createReport	🔒
GET	/user/role getUserRole	🔒
GET	/user/role/{username} getUserRole	🔒

Rys. 3.13: Dokumentacja - UserController

User Controller

Pozwala na edycję danych użytkownika oraz pobieranie niezbędnych do działania aplikacji, informacjach.

walk-controller Walk Controller		
GET	/walk/{id} getWalkCard	🔒
DELETE	/walk/{id} deleteWalk	🔒
GET	/walk/all getWalks	🔒
GET	/walk/allWithFilters getWalksWithFilters	🔒
POST	/walk/create createWalk	🔓
POST	/walk/disenroll/{id} disenroll	🔒
POST	/walk/enroll/{id} enroll	🔓

Rys. 3.14: Dokumentacja - WalkController

Walk Controller

Kontroler odpowiedzialny za tworzenie nowych spacerów, pobieranie listy dostępnych spacerów, zapisywanie oraz wyspisywanie się ze spacerów.

3.4.3. JWT – Autoryzacja i autentykacja

Autoryzacja użytkowników jest obecnie często używanym mechanizmem. Pozwala na nadawanie użytkownikom odpowiednich ролей, zabezpieczaniem endpointów oraz kontrolerów aplikacji backendowej przed dostępem dla osób niepowołanych. Umożliwia również zabezpieczanie kont użytkowników poprzez zastosowanie enkrypcji haseł odpowiednim algorytmem, który jest dostępny od razu po dołączeniu do projektu zależności *Spring Security*.

```

61 	@PostMapping("/login")
62 	public ResponseEntity<?> login(@Valid @RequestBody LoginRequest request) {
63 		Optional<User> user = userRepository.findByUsername(request.getUsername());
64 		if (user.isEmpty()) {
65 			return ResponseEntity.badRequest().body("Taki użytkownik nie istnieje!");
66 		} else {
67 			if (!user.get().getIsActive()) {
68 				return ResponseEntity.badRequest().body("Konto nie jest aktywowane!! ");
69 			} else if (user.get().getIsBlocked()) {
70 				return ResponseEntity.badRequest().body("Konto jest zablokowane!");
71 			} else if (user.get().getIsBanned()) {
72 				return ResponseEntity.badRequest().body("Konto zostało zbanowane!");
73 			} else {
74 				if (encoder.bCryptPasswordEncoder().matches(request.getPassword(), user.get().getPassword())) {
75 					Authentication authentication = authenticationManager.authenticate(
76 						new UsernamePasswordAuthenticationToken(request.getUsername(), request.getPassword()));
77 					SecurityContextHolder.getContext().setAuthentication(authentication);
78 					String jwt = jwtUtils.generateJwtToken(authentication);
79
80 					UserDetailsImpl userDetails = (UserDetailsImpl) authentication.getPrincipal();
81 					List<String> roles = userDetails.getAuthorities().stream().map(item → item.getAuthority())
82 						.collect(Collectors.toList());
83
84 					return ResponseEntity.ok(new JwtResponse(jwt, userDetails.getId(), userDetails.getUsername(),
85 						userDetails.getEmail(), roles));
86 				} else {
87 					return ResponseEntity.badRequest().body("Nazwa użytkownika lub hasło jest niepoprawne");
88 				}
89 			}
90 		}
91 	}

```

Rys. 3.15: Implementacja logowania przy pomocy JWT

Proces implementacji rozpoczęto od zamodelowania klasy użytkownika z odpowiednimi polami oraz stworzeniem ról użytkowników. Następnie dodano w klasie serwisowej logikę odpodzielczą za dodawanie do użytkowników ról oraz zapisywanie takich obiektów w bazie danych.

Kolejnym etapem była implementacja logiki związanej z JWT oraz logowaniem użytkowników. Proces logowania prezentuje się w następujący sposób:

- Przesłanie formularzu logowania z danymi – nazwą użytkownika oraz hasłem;
- Sprawdzenie czy taki użytkownik znajduje się w bazie danych;
- Sprawdzenie, czy hasło odpowiada temu, które jest zapisane i zdekryptowane odpowiednim algorytmem;
- W przypadku sukcesu, generowany jest token, zawierający dane: id, nazwę, email oraz role użytkownika;
- W przypadku niepowodzenia, do klienta wysyłany jest odpowiedni komunikat o błędzie.

3.4.4. Implementacja wyszukiwania po lokalizacji

Podczas tworzenia spaceru bądź rejestracji użytkownika, na podstawie kodu pocztowego podanego w formularzu, wysyłane jest zapytanie do API Poczty Polskiej o informacje związane z miejscowością, województwem oraz powiatem. Na podstawie tej odpowiedzi, w bazie danych wyszukiwany jest odpowiedni region i zwracane są kody – powiatu oraz województwa. Kody następnie są dodawane do tworzonego obiektu.

```

66  public void saveDistricts(MultipartFile config) throws IOException {
67      Scanner scanner = new Scanner(config.getInputStream());
68
69      List<District> districts = scanner.useDelimiter("\n").tokens()
70          .map(line → line.split("\t"))
71          .map(array → this.districtMapper(array))
72          .collect(Collectors.toList());
73
74      scanner.close();
75      districtRepository.saveAll(districts);
76  }

```

Rys. 3.16: Implementacja zapisywania pliku konfiguracyjnego

Podczas wysyłania zapytania o listę dostępnych spacerów, po stronie backendu odbywa się filtrowanie listy wszystkich dostępnych spacerów. Pierwszym etapem jest pobranie informacji o regionie użytkownika, który wysłał zapytanie, następnie następuje filtrowanie spacerów po kodzie powiatu. W przypadku uzyskania pustej listy, następuje dodatkowe filtrowanie po kodzie województwa. Taka lista ostatecznie przesyłana jest do aplikacji dostępowej w celu jej prezentacji użytkownikowi.

```

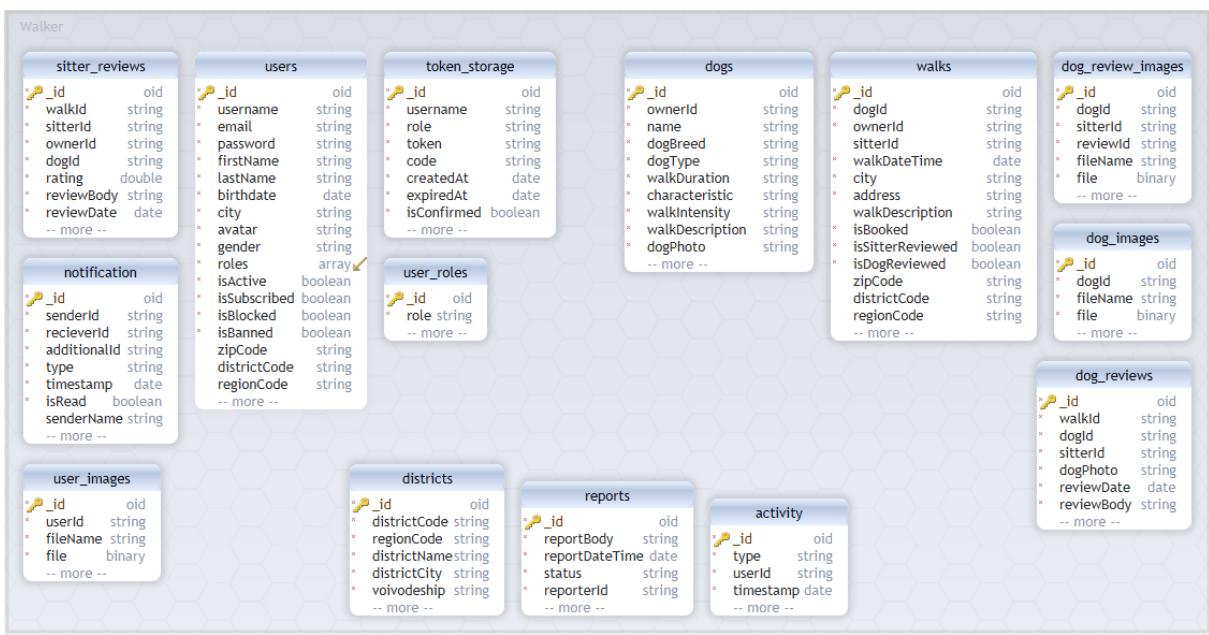
35  public District findCity(String postCode) {
36      Response response = this.getDistrictCode(postCode);
37      if(!districtRepository.findByDistrictName(response.getDistrict()).isEmpty()) {
38          return districtRepository.findByDistrictName(response.getDistrict()).get(0);
39      }
40      else return districtRepository.findByDistrictCity(response.getCity()).get(0);
41  }
42
43  public Response getDistrictCode(String postCode) {
44      RestTemplate template = new RestTemplate();
45      HttpHeaders headers = new HttpHeaders();
46      headers.set(HttpHeaders.ACCEPT, MediaType.APPLICATION_JSON_VALUE);
47      HttpEntity<?> entity = new HttpEntity<>(headers);
48
49
50      UriComponentsBuilder completeUrl =
51          .fromHttpUrl(String.format(apiUrl, postCode))
52          .build()
53          .encode()
54          .toUri();
55
56      ResponseEntity<Response[]> response = template.exchange(
57          completeUrl,
58          HttpMethod.GET,
59          entity,
60          Response[].class);
61
62      return response.getBody()[0];
63  }
64

```

Rys. 3.17: Lista metod odpowiedzialnych za sprawdzanie lokalizacji

3.4.5. Encje

Nierelacyjne bazy danych charakteryzują się brakiem relacji między obiektami. Dlatego też, żeby połączyć odpowiadające obiekty w bazie, dodano do odpowiednich obiektów pola reprezentujące id powiązanych obiektów. Pozwoliło to na zachowanie porządku i spójności danych.



Rys. 3.18: Widok bazy danych – zbiór obiektów

Obiekty, które są wysyłane do aplikacji dostępowej są złożonymi z kilku encji obiektami. Do tego użytko funkcji mapujących po stronie backendu. Takie funkcje mają za zadanie łączyć wiele obiektów ze sobą na podstawie wyżej wymienionego, dodatkowego pola id powiązanego obiektu.

3.5. Angular – frontend

3.5.1. Konfiguracja projektu

Proces konfiguracji rozpoczęło przystosowanie pliku *environment.ts*.

```
1  export const environment = {
2    apiUrl: 'http://localhost:8080',
3    urls,
4  };
```

- *apiBaseUrl* – deklaracja adresu URL, pod którym znajduje się backend aplikacji;
- *urls* – zbiór dostępnych endpointów, pod które można wysyłać zapytanie.

Obiekt *urls* jest zdefiniowany w następujący sposób:

```
1  ControllerName: {
2    prefix: prefixName,
3    calls: {
4      endpoint: endpointName,
5    },
6  }
```

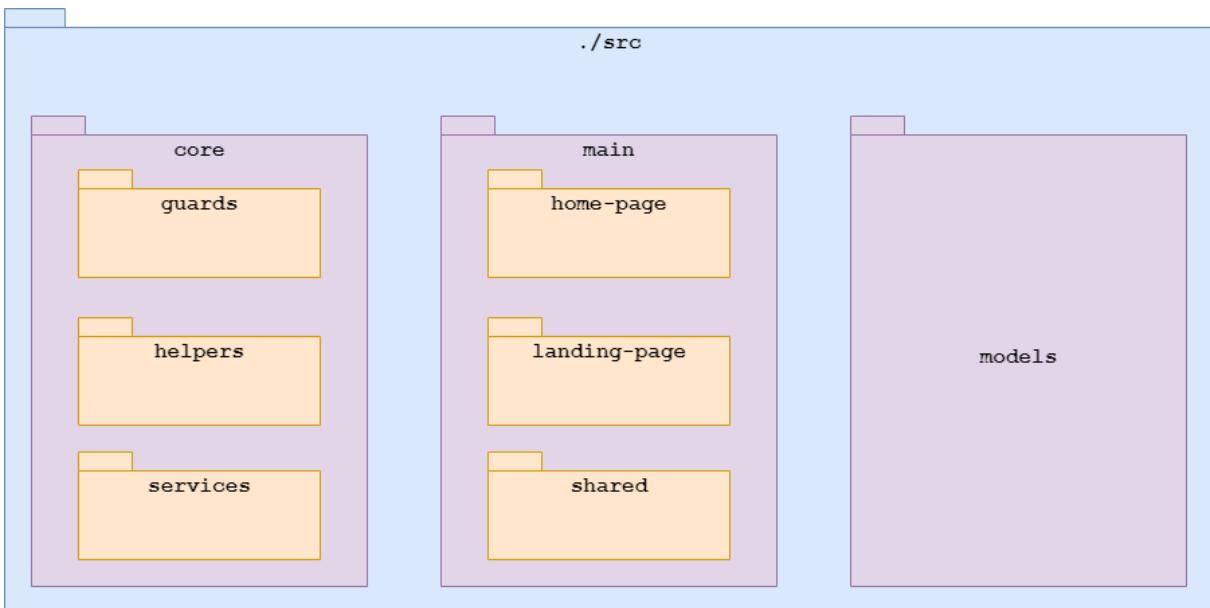
Takie rozwiązanie pozwoliło na deklarację wszystkich niezbędnych endpointów w jednym pliku konfiguracyjnym. Na potrzeby tego rozwiązania został stworzony plik serwisowy, który udostępnia szereg metod odpowiedzialnych za pobieranie, parsowanie oraz zwarcie odpowiednio sparenowanego adresu URL.

```
1  public getAuthUrl(action: string, params?: any) {
2    return this.getUrl('auth', action, params);
3  }
```

Wywołanie przykładowej metody rozpoczyna przekazanie do niej parametru *action*, który określa pod jaki konkretnie endpoint będzie szło zapytanie oraz opcjonalny parametr *params*, czyli zbiór parametrów wstrzykiwanych do adresu. Tak sparenowy adres jest przekazywany do metody odpowiedzialnej za wywoływanie zapytań oraz odbieranie odpowiedzi ze strony aplikacji backendowej.

3.5.2. Architektura projektu frontendowego

Charakterystyka plików tworzonych w aplikacji frontendowej nie pozwala na zastosowanie omawianej wcześniej architektury użytej przy projekcie backendowym.



Rys. 3.19: Architektura – reprezentacja graficzna

Podział został uwarunkowany przeznaczeniem plików. Architektura prezentuje się zatem następująco:

- **core** – przechowuje podstawowe funkcjonalności:
 - **guards** – zawiera implementację klasy, która na podstawie roli użytkownika, pozwala lub wzbrania na dostęp do poszczególnych widoków aplikacji;
 - **helpers** – zawiera implementację klasy, odpowiedzialnej za wstrzykiwanie do każdego wysyłanego zapytania token autoryzacyjny zalogowanego użytkownika;
 - **services** – zawiera wszystkie serwisy, które są używane w aplikacji.
- **main** – przechowuje główne widoki oraz komponenty z wyróżnieniem na dwa główne widoki: stronę domową oraz stronę startową aplikacji:
 - **home-page** – zawiera implementację widoków, komponentów z rozróżnieniem na role użytkowników;
 - **landing-page** – przechowuje komponenty wykorzystywane przy stronie startowej takie, jak formularze logowania, rejestracji;
 - **shared** – w tym folderze zawarte są implementacje komponentów, które są wykorzystywane w każdym miejscu w aplikacji na przykład komponent odpowiedzialny za wyświetlanie komunikatów;
- **models** – przechowuje modele obiektów używanych w aplikacji.

3.5.3. Działanie



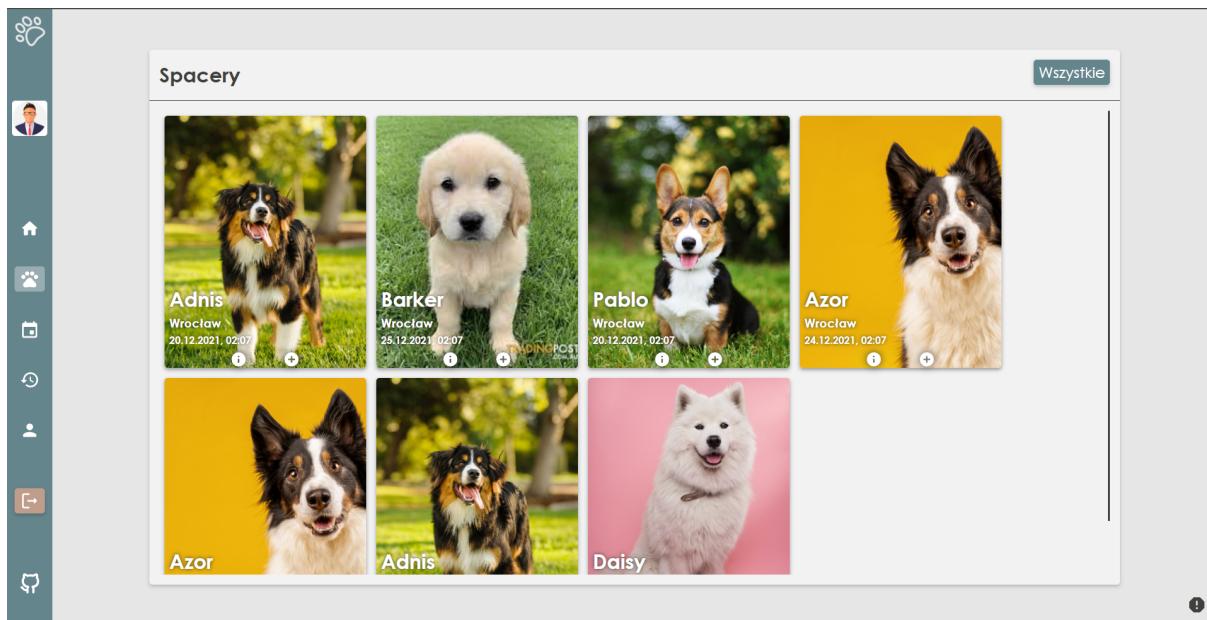
Rys. 3.20: Strona startowa

	Dog Name	Location	Status	Date
	Adnis	Wrocław	Wolny	20.12.2021r
	Barker	Wrocław	Wolny	25.12.2021r
	Pablo	Wrocław	Wolny	20.12.2021r
	Azor	Wrocław	Wolny	24.12.2021r
	Azor	Wrocław	Wolny	28.12.2021r
	Adnis	Wrocław	Wolny	26.12.2021r
	Daisy	Wrocław	Wolny	26.12.2021r

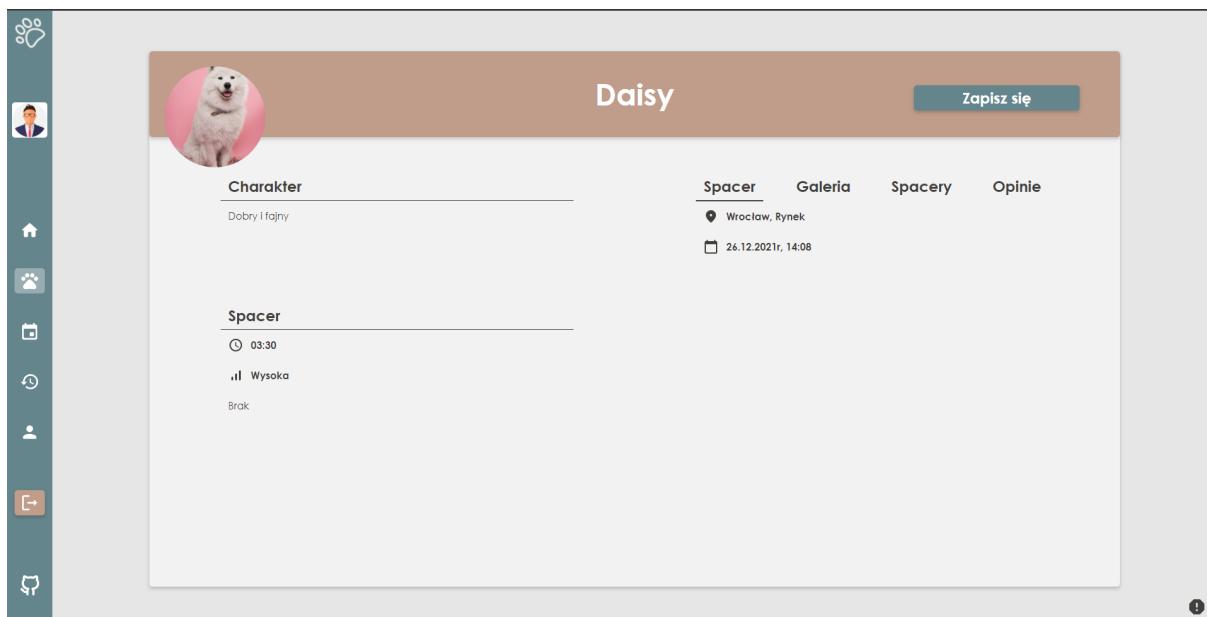
Powiadomienia

- 26/10/2021 17:15
Użytkownik sitter zapisał(a) się na spacer
- 26/10/2021 17:15
Użytkownik sitter wypisał(a) się ze spaceru
- 26/10/2021 20:52
Użytkownik sitter zapisał(a) się na spacer
- 26/10/2021 20:52
Użytkownik sitter zapisał(a) się na spacer
- 26/10/2021 20:52
Użytkownik sitter zapisał(a) się na spacer
- 26/10/2021 20:48
Użytkownik sitter zapisał(a) się na spacer
- 26/10/2021 20:48
Użytkownik sitter zapisał(a) się na spacer

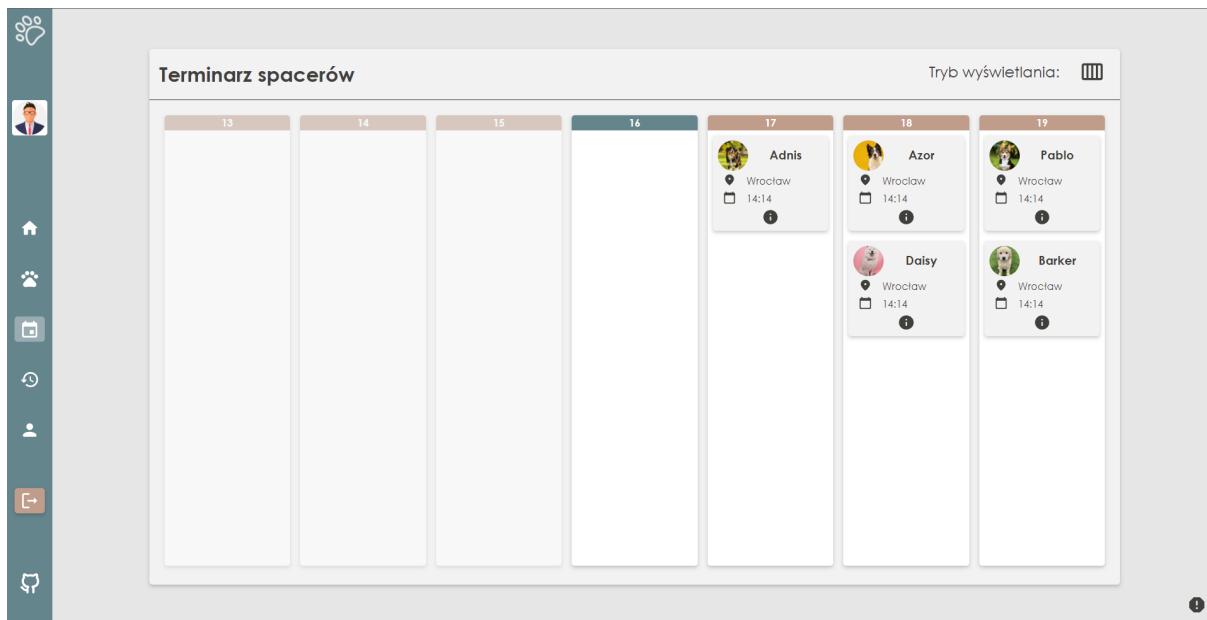
Rys. 3.21: Dashboard



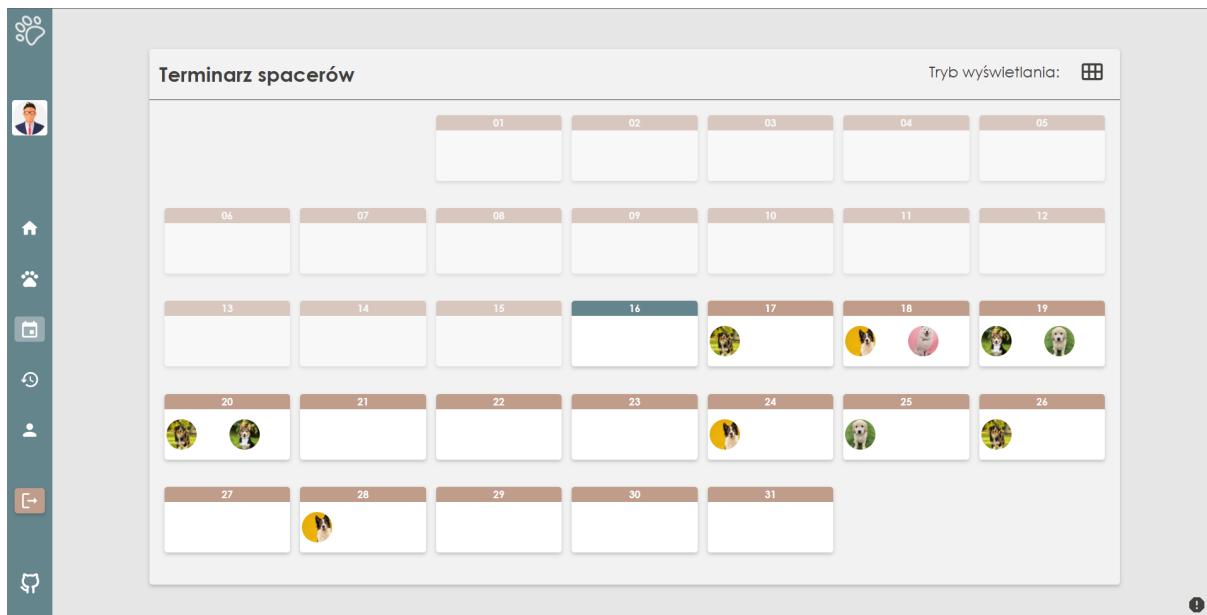
Rys. 3.22: Lista spacerów



Rys. 3.23: Profil psa



Rys. 3.24: Kalendarz – widok tygodnia



Rys. 3.25: Kalendarz – widok miesiąca

3.5.4. JWT – Autoryzacja i autentykacja

Proces implementacji autoryzacji użytkowników przez mechanizm JWT rozpoczęto od stworzenia metody odpowiedzialnej za wysyłanie formularza logowania do backendu:

```

1  loginUser(credentials: any): Observable<any> {
2      const url = this.setting.getAuthUrl('loginUser');
3
4      return this.http.post(url, {
5          username: credentials.username,
6          password: credentials.password
7      }, httpOptions);
8  }

```

Metoda zostaje wywołana w momencie naciśnięcia przycisku *Zaloguj się*.

```

1  loginUser(): void {
2      this.authService.loginUser(this.loginForm.value).subscribe(
3          data => {
4              this.saveAuthTokenAfterLogin(data.token);
5          }
6      );
7  }

```

Po pomyślnym zalogowaniu, do aplikacji dostępowej zostaje wysłany token autoryzacyjny, który następnie przy pomocy klasy *TokenStorageService* zapisuje go w sesji przeglądarki.

```

1  const TOKEN_KEY = 'auth-token';
2
3  public saveToken(token: string): void {
4      window.sessionStorage.removeItem(TOKEN_KEY);
5      window.sessionStorage.setItem(TOKEN_KEY, token);
6  }

```

Zapisany token zostaje następnie wstrzykiwany do każdego zapytania wysyłanego do backendu przez zalogowanego użytkownika. Odpowiada za to klasa *AuthInterceptor* oraz metoda *intercept*:

```

1  intercept(req: HttpRequest<any>, next: HttpHandler):
2      Observable<HttpEvent<any>> {
3      let authReq = req;
4      const token = this.token.getToken();
5      if (token != null) {
6          authReq = req.clone({
7              headers: req.headers.set(TOKEN_HEADER_KEY, 'Bearer ' + token)
8          });
9      }
10     return next.handle(authReq);
11 }

```

3.5.5. Biblioteki i moduły

Wykorzystane biblioteki oraz moduły wspomogły proces implementacyjny oraz udostępnili szereg funkcjonalności, bez których nie byłoby możliwe stworzenie rozbudowanej aplikacji dostępowej.

```

1  imports: [
2      BrowserModule,
3      AppRoutingModule,
4      ReactiveFormsModule,
5      HttpClientModule,
6      NgbModule
7  ]

```

AppRoutingModule

```

26 const routes: Routes = [
27   {
28     path: 'start',
29     component: LandingPageComponent,
30     children: [
31       { path: 'register', component: RegisterComponent },
32       { path: 'register/:token', component: RegisterComponent },
33       { path: 'confirm/:token', component: ConfirmationComponent },
34       { path: '**', component: PageNotFoundComponent}
35     ],
36   },
37   { path: 'home',
38     component: HomePageComponent,
39     canActivate: [AuthGuard],
40     data: {
41       role: 'ROLE_USER'
42     },
43     children: [
44       { path: 'dashboard', component: DashboardComponent, canActivate: [AuthGuard], data: {role: 'ROLE_USER'}},
45       { path: 'creator', component: DogCreatorComponent, canActivate: [AuthGuard], data: {role: 'ROLE_OWNER'}},
46       { path: 'walks', component: DogWalksComponent, canActivate: [AuthGuard], data: {role: 'ROLE_SITTER'}},
47       { path: 'reports', component: BugReportComponent, canActivate: [AuthGuard], data: {role: 'ROLE_ADMIN'}},
48       { path: 'walk_planner', component: WalkPlannerComponent, canActivate: [AuthGuard], data: {role: 'ROLE_OWNER'}},
49       { path: 'planner', component: PlannerComponent, canActivate: [AuthGuard], data: {role: 'ROLE_SITTER'}},
50       { path: 'users', component: UsersComponent, canActivate: [AuthGuard], data: {role: 'ROLE_ADMIN'}},
51       { path: 'user_dogs', component: DogsComponent, canActivate: [AuthGuard], data: {role: 'ROLE_ADMIN'}},
52       { path: 'dogs', component: OwnerDogsComponent, canActivate: [AuthGuard], data: {role: 'ROLE_OWNER'}},
53       { path: 'history', component: HistoryComponent, canActivate: [AuthGuard], data: {role: 'ROLE_USER'}},
54       { path: 'profile', component: ProfileComponent, canActivate: [AuthGuard], data: {role: 'ROLE_USER'}},
55       { path: 'profile/:username', component: ProfileComponent, canActivate: [AuthGuard], data: {role: 'ROLE_USER'}},
56       { path: 'walk/:id', component: WalkComponent, canActivate: [AuthGuard], data: {role: 'ROLE_USER'}},
57       { path: 'dog/:id/walkid', component: DogComponent, canActivate: [AuthGuard], data: {role: 'ROLE_USER'}},
58       { path: 'dog/:id', component: DogComponent, canActivate: [AuthGuard], data: {role: 'ROLE_USER'}},
59       { path: 'error404', component: PageNotFoundComponent, canActivate: [AuthGuard], data: {role: 'ROLE_USER'}},
60       { path: '**', redirectTo: 'start' }
61     ]
62   },
63   {
64     },
65   { path: '**', redirectTo: 'start' }
66 ];

```

Rys. 3.26: Implementacja nawigacji

Dzięki temu modułowi, możliwe było zadeklarowanie ścieżek aplikacji, z uwzględnieniem poszczególnych komponentów oraz zabezpieczenie przed użytkownikami, którzy nie posiadają odpowiedniej roli.

ReactiveFormsModule

```

1  dogForm = this.builder.group({
2    name: ['', Validators.required],
3    dogBreed: ['', Validators.required],
4    dogPhoto: ['', Validators.required],
5    dogType: ['', Validators.required],
6    characteristic: ['', [
7      Validators.required,
8      Validators.maxLength(255)]],
9    walkDuration: ['', [
10      Validators.required,
11      Validators.pattern('[0-9]{2}:[0-9]{2}')]],
12    walkIntensity: ['', Validators.required],
13    walkDescription: ['', [
14      Validators.required,
15      Validators.maxLength(255)]],
16  });

```

HttpClientModule

```

1  getOwnerData(username?: string) {
2    const url = this.setting.getOwnerUrl(
3      'getData',

```

```
4     {username: username}
5   );
6
7   return this.http.get<OwnerData>(url);
8 }
```

NgbModule

```
1 <ngb-rating
2   [max] = "5"
3   formControlName = "rating">
4 </ngb-rating>
```

RxJS

```
1 this.users = this.users.pipe(
2   map((users: UserWithInfo[]) => {
3     return removeById(users, user.id);
4   })
5 );
```

Rozdział 4

Podsumowanie

Aplikację udało się opracować oraz zaimplementować założone funkcjonalności. W przyszłości planowany jest dalszy rozwój aplikacji.

4.1. Rozwój aplikacji

- Analiza rynku pod kątem wdrożenia – sprawdzenie, czy aplikacja mogłaby potencjalnie wejść na rynek, czy spełnia oczekiwania konsumentów;
- Rozszerzenie aplikacji o dodatkowe elementy:
 - Implementacja menedżera stanu aplikacji – moduł NgRx
 - Wymiana wiadomości między użytkownikami
 - Monitorowanie stanu aplikacji
 - Responsywność
 - Udostępnienie aplikacji na serwerze
- Przeprowadzenie testów obciążeniowych na rzeczywistym, podstawowym serwerze

Spis rysunków

2.1. Diagram przypadków użycia	8
3.1. Trello - tablica do pierwszego etapu	17
3.2. Trello - tablica do drugiego etapu	18
3.3. Przykładowe prototypy aplikacji – widok startowy, widok dashboardu	18
3.4. Przykładowe prototypy aplikacji – widok startowy, widok dashboardu	19
3.5. Przykładowe prototypy aplikacji – widok startowy, widok dashboardu	19
3.6. Przykładowe prototypy aplikacji – widok startowy, widok dashboardu	19
3.7. Szczegółowa architektura - reprezentacja podfolderów	20
3.8. Architektura projektu backendowego	21
3.9. Dokumentacja - AddressController, AdminController	22
3.10. Dokumentacja - AuthController, DogController	23
3.11. Dokumentacja - ImageController, OwnerController	23
3.12. Dokumentacja - ReviewController, SitterController	24
3.13. Dokumentacja - UserController	25
3.14. Dokumentacja - WalkController	25
3.15. Implementacja logowania przy pomocy JWT	26
3.16. Implementacja zapisywania pliku konfiguracyjnego	27
3.17. Lista metod odpowiedzialnych za sprawdzanie lokalizacji	27
3.18. Widok bazy danych – zbiór obiektów	28
3.19. Architektura – reprezentacja graficzna	30
3.20. Strona startowa	31
3.21. Dashboard	31
3.22. Lista spacerów	32
3.23. Profil psa	32
3.24. Kalendarz – widok tygodnia	33
3.25. Kalendarz – widok miesiąca	33
3.26. Implementacja nawigacji	35

Literatura

Dodatek A

Opis załączonej płyty CD/DVD

Tutaj jest miejsce na zamieszczenie opisu zawartości załączonej płyty. Należy wymienić, co zawiera.