COLLEGIUM DA VINCI

Wydział Nauk Stosowanych

Kierunek: INFORMATYKA
studia pierwszego stopnia

Dominik Szymański
Mateusz Wiemann

Aplikacja mobilna do analizy cen paliw
Mobile application for fuel price analysis

Praca inżynierska

Praca wykonana pod kierunkiem
dr. Tomasza Tyksińskiego

Poznań 2024

Spis treści

1. Wstęp

W dobie dynamicznych zmian cen paliw, które odgrywają kluczową rolę w codziennym życiu wielu osób, poniższa praca inżynierska koncentruje się na stworzeniu aplikacji mobilnej do analizy cen paliw. Celem projektu jest zapewnienie użytkownikom zaawansowanego narzędzia, które umożliwi im szybkie i dokładne porównywanie cen paliw na stacjach benzynowych w ich rejonie.

Projekt również dąży do zbadania związku między cenami hurtowymi a cenami na stacjach, aby uświadomić użytkownikom, jakie mechanizmy wpływają na zmiany cen paliw. Głównym celem jest nie tylko ułatwienie konsumentom wybierania najlepszego momentu i miejsca na zakup paliwa, ale również podniesienie ich świadomości odnośnie złożoności czynników wpływających na ceny paliw. Lepsze zrozumienie tych relacji może skłonić użytkowników do zmiany długoterminowych nawyków zakupowych, zachęcając do poszukiwania alternatywnych źródeł energii lub sposobów transportu, co może przyczynić się do jego zrównoważenia.

Realizacja tego projektu inżynierskiego jest reakcją na rosnące potrzeby społeczne dotyczące narzędzi, które umożliwiają skuteczniejsze zarządzanie budżetem przeznaczonym na paliwo. Wybór tematu nie jest przypadkowy – odzwierciedla on aktualne tendencje na rynku i zwiększającą się świadomość ekonomiczną w społeczeństwie. W świetle niestabilności cen ropy i różnic w polityce cenowej poszczególnych państw, konsumenci muszą podejść do zużycia paliwa bardziej świadomie. Dlatego opracowana aplikacja mobilna do analizy cen paliw może stać się niezbędnym narzędziem pomagającym użytkownikom podjąć najlepszą decyzję zakupową.

2. Aktualny stan wiedzy

2.1 Problematyka

Zagadnienia inżynierskie dotyczące analizy cen paliw i tworzenia aplikacji mobilnych do ich śledzenia są złożone, obejmują szerokie spektrum wiedzy ekonomicznej, technologicznej i społecznej. Kluczowym elementem jest dynamika cen paliw, charakteryzująca się dużą zmiennością i podlegająca wpływom zarówno globalnym, jak i lokalnym - od cen ropy na rynkach światowych, przez polityki podatkowe krajów, marże dystrybutorów, aż po lokalne podatki i opłaty. Ta zmienność ma bezpośredni wpływ na wybory konsumentów, którzy w obliczu wzrostu cen poszukują najbardziej korzystnych opcji zakupu paliwa.

Próbując rozwiązać problem aktualności danych o cenach paliw, nasze podejście opiera się na gromadzeniu informacji z różnych źródeł, zarówno poprzez integrację z innymi aplikacjami i portalami internetowymi, jak i bezpośrednie zbieranie danych od użytkowników. Ta strategia, choć obiecująca, rodzi nowe wyzwania i aspekty, które wymagają dalszej analizy i rozwoju.

Istotnym wyzwaniem jest zapewnienie wiarygodności i dokładności danych pochodzących z zewnętrznych źródeł. Różnice w częstotliwości aktualizacji cen paliw przez poszczególne aplikacje i portale mogą prowadzić do rozbieżności w prezentowanych informacjach. Wymaga to opracowania zaawansowanych algorytmów weryfikacji i konsolidacji danych, które będą w stanie ocenić ich aktualność i wiarygodność, zanim zostaną one zaprezentowane użytkownikom.

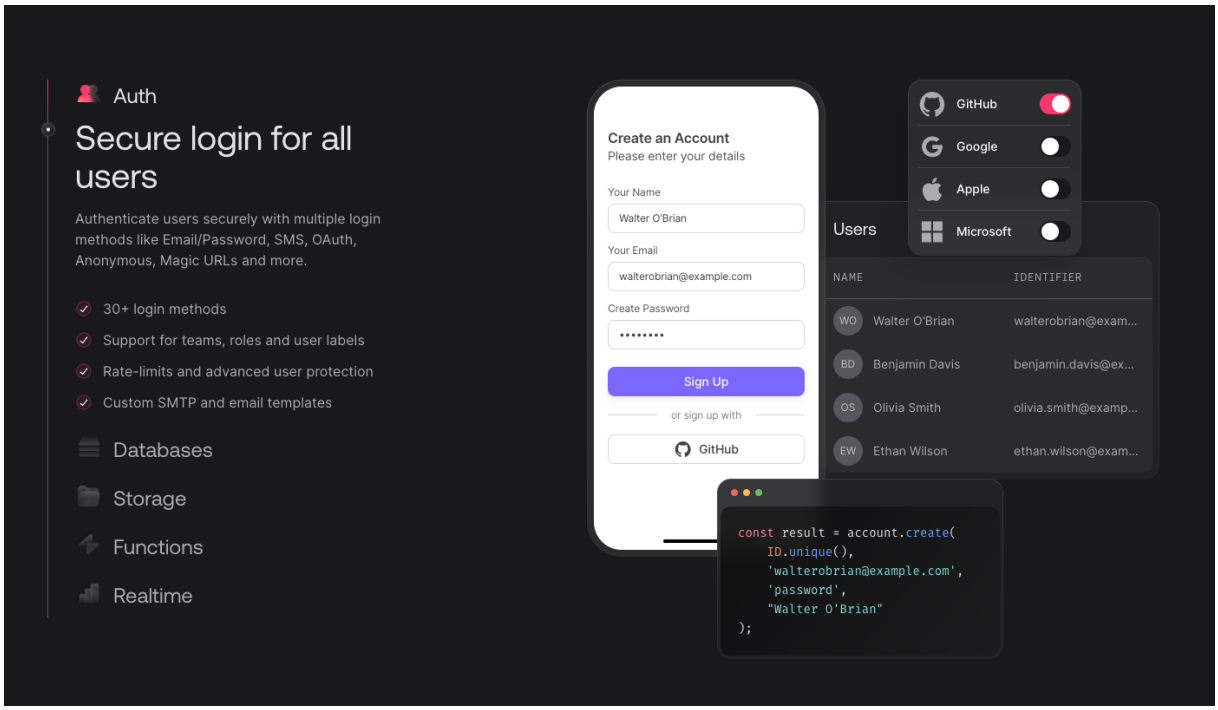
Następne wyzwanie wiąże się z zaangażowaniem i motywacją użytkowników do dzielenia się aktualnymi danymi cenowymi poprzez skanowanie pylonów paliwowych. Choć proces ten może być przyjazny dla użytkownika, jego skuteczność zależy od aktywności i chęci współpracy społeczności. Możliwe strategie zachęcania mogą obejmować systemy nagród lub gamifikację, które motywują użytkowników do regularnego dostarczania aktualnych danych.

2.2 Technologie informatyczne do rozwiązania problemu

Do rozwiązania wspominanego problemu można podejść na wiele sposobów. W projekcie zdecydowano się na wykorzystanie technologii **Appwrite** [16]: nowoczesnej platformy zapewniającej niezbędne komponenty backendowe oraz łatwą integracje do platform mobilnych.

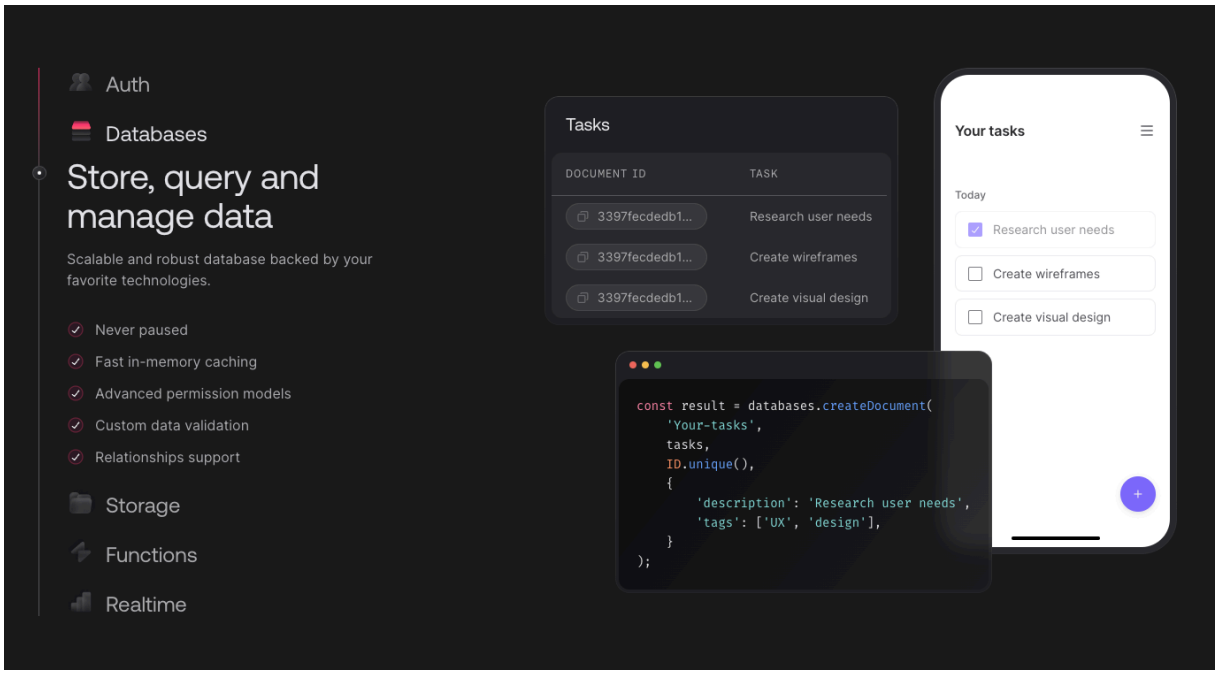
Rysunek 2.1 Logo Appwrite (źródło: *www.appwrite.io*)

Appwrite to produkt typu PaaS (ang. *platform as a service*), czyli aplikacja zapewniająca cały szereg gotowych rozwiązań powszechnie stosowanych w aplikacjach desktopowych, webowych i mobilnych. W projekcie wykorzystano komponenty **Auth**, **Databases**, **Storage** i **Functions**, które zostały opisane poniżej.



Rysunek 2.2 Strona główna projektu Appwrite opisująca komponent Auth (źródło: *https://appwrite.io*)

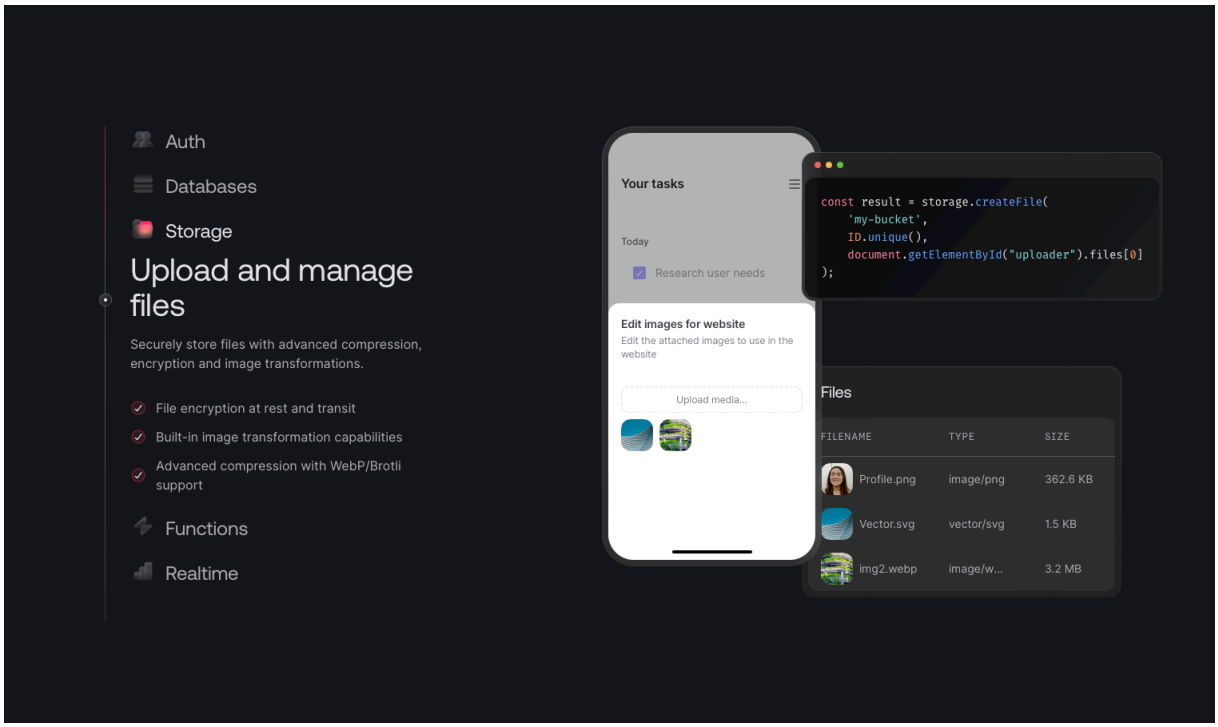
Komponent **Auth** to backend który pozwala na stworzenie zaplecza technologicznego do rejestracji, logowania i zarządzania użytkownikami. Auth posiada również implementacje logowania przez protokół OAuth2 [34], pozwalający na wykorzystanie innych serwisów (takich jak Github, Facebook lub Google) jako źródło poświadczeń użytkownika. W projekcie nie zdecydowano się na wykorzystanie go ze względu na brak czasu. Skorzystano za to z podstawowej wersji komponentu Auth, czyli zwykłego logowania i rejestracji parą adres email/hasło.



Rysunek 2.3 Strona główna projektu Appwrite opisująca komponent Databases (źródło: *https://appwrite.io. dostęp: 20.04.2024*)

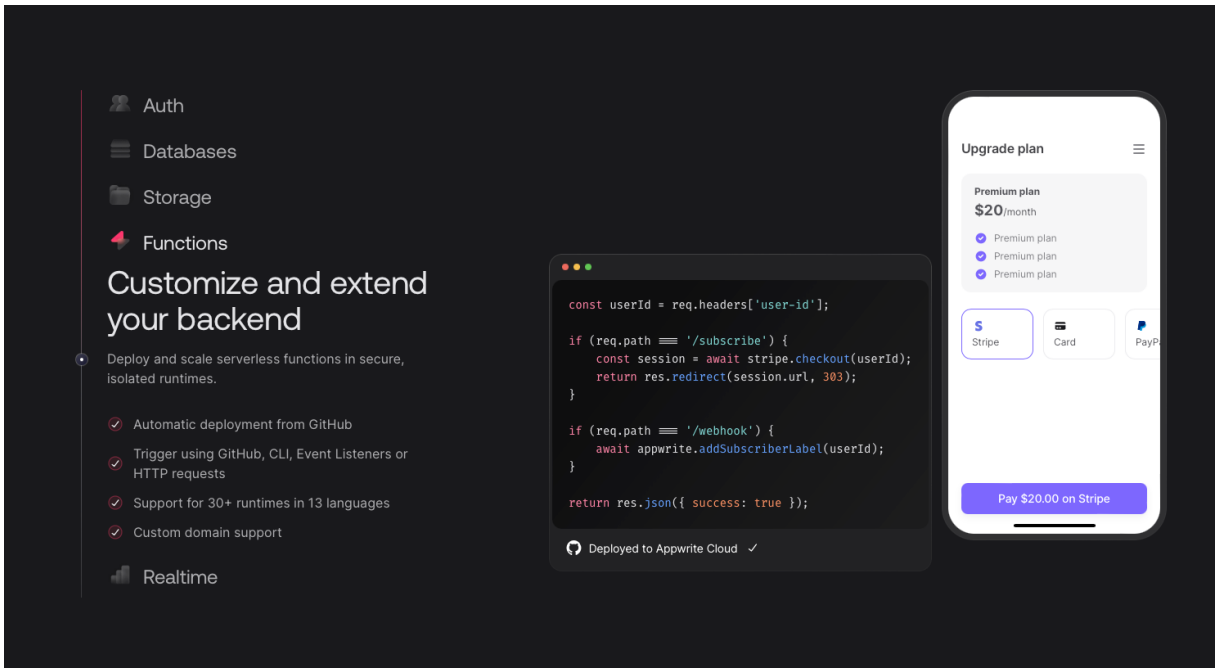
Databases to abstrakcyjna nakładka Appwrite na system bazodanowy działający „na zapleczu” platformy.
W rzeczywistości pod warstwą abstrakcji funkcjonuje MariaDB [30], czyli relacyjna baza danych SQL.

Sam komponent w swojej strukturze przypomina operowanie na bazach NoSQL. Nie mamy tutaj tabel i wierszy, jak w klasycznej bazie SQL, lecz kolekcje i dokumenty. W projekcie zdecydowano się na wybór tego komponentu jako głównej bazy danych ze względu na brak uzależnienia od dostawcy (ang. *vendor lock-in*). W każdej chwili jest możliwy dostęp do bazy danych MariaDB i migrowanie danych do innej struktury.



Rysunek 2.4 Strona główna projektu Appwrite opisująca komponent Storage (źródło: *https://appwrite.io. dostęp: 20.04.2024*)

Storage jest odpowiedzialny za obsługę przesyłania plików. Pozwala na zintegrowanie się z dostawcami chmurowymi (takimi jak AWS S3 [3], DigitalOcean Spaces [13], Backblaze [4]). Jest oparty na strukturze „wiadra” (ang. *bucket*) – powszechnej struktury wykorzystywanej w usługach wcześniej wymienionych dostawców chmurowych. Nasze wiadra są grupą plików, która może mieć narzucone określone właściwości (wymagane uprawnienia, dopuszczanie określonych rodzajów plików, szyfrowanie).



Rysunek 2.5 Strona główna projektu Appwrite opisująca komponent Functions (źródło: *https://appwrite.io. dostęp: 20.04.2024*)

Functions to najbardziej zaawansowany komponent Appwrite. Pozwala on na uruchamianie kodu w ponad 30 środowiskach uruchomieniowych (takich jak Python [37], PHP [35], NodeJS [33]) i skoordynowanie go ze środowiskiem (np. wyzwolić funkcję w Python, która zwraca dane do innej funkcji w PHP).

Jako główną technologię aplikacji mobilnej wybrano **Flutter** [20], który posiada dobrą integrację z Appwrite przez gotowe SDK (ang. *Software Development Kit*). Flutter jest technologią wieloplatformową, pozwala za pomocą jednego kodu zbudować aplikacje zarówno na iOS jak i Android.

Rysunek 2.6 Logo Flutter (źródło: *http://docs.flutter.dev*)

Flutter bazuje na języku **Dart** [11], który docelowo jest kompilowany do kodu maszynowego [27] co pozwala na uzyskanie wysokiej wydajności zarówno na urządzeniach iOS jak i Android. Jest opracowany przez Google [25].

Rysunek 2.7 Logo Dart (źródło: *https://dart.dev/brand*)

Dart swoją strukturą bardzo przypomina język Kotlin [28], w którym zespół projektowy miał już doświadczenie – był to również decyzyjny aspekt wyboru tego narzędzia.

W aplikacji mobilnej wykorzystano bibliotekę **Mapbox** [29] jako główny komponent mapy. W porównaniu do powszechnie znanego Google Maps [24], Mapbox oferuje dużo korzystniejszy cennik, co było decydującym czynnikiem przy wyborze tej technologii.

Rysunek 2.8 Logo Mapbox (źródło: *https://mapbox.com*)

Do prezentacji wykresów do analizy początkowo planowano wykorzystać technologię FlutterCharts [18]. Niestety ze względu na bardzo mało opcji dostosowania wykresów oraz problematyczną implementację wymagającą setek linijek kodu, zdecydowano się na skorzystanie z **Chart.js** [7] osadzonej na stronie internetowej HTML (ang. *HyperText Markup Language*). Strona została osadzona w komponencie WebView [42], czyli przeglądarce internetowej umieszczonej bezpośrednio w aplikacji mobilnej. Wybór był podyktowany zaawansowanymi możliwościami i bogatym zestawem funkcji, które znacznie przewyższają te dostępne w FlutterCharts.

Rysunek 2.9 Logo Chart.js (źródło: *https://chartjs.org*)

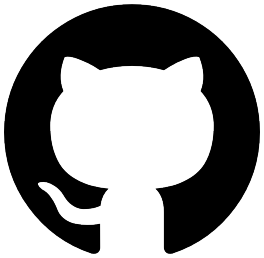
Do organizacji kodu zastosowano system VCS (ang. *Version Control System*), czyli system kontroli wersji. Najpopularniejszym systemem tego typu jest **Git** [22]. Pozwala on na łatwe śledzenie zmian i wycofywanie tych niepożądanych.

Rysunek 2.10 Logo Chart.js (źródło: *https://chartjs.org*)

Jest on swego rodzaju standardem w branży IT. Jego elastyczność i potężne funkcje przyspieszają współpracę w zespole, pozwalają na zastosowanie tzw. metodyki **git flow** [23] którą również wykorzystano w projekcie. Opiera się ona na zrównolegleniu pracy przez rozłączeniu głównej gałęzi (ang. *master branch*) kodu na tzw. gałęzie z funkcjonalnościami (ang.

feature branch), które docelowo po zakończeniu pracy nad funkcjonalnością wracają do głównej gałęzi z kodem źródłowym.

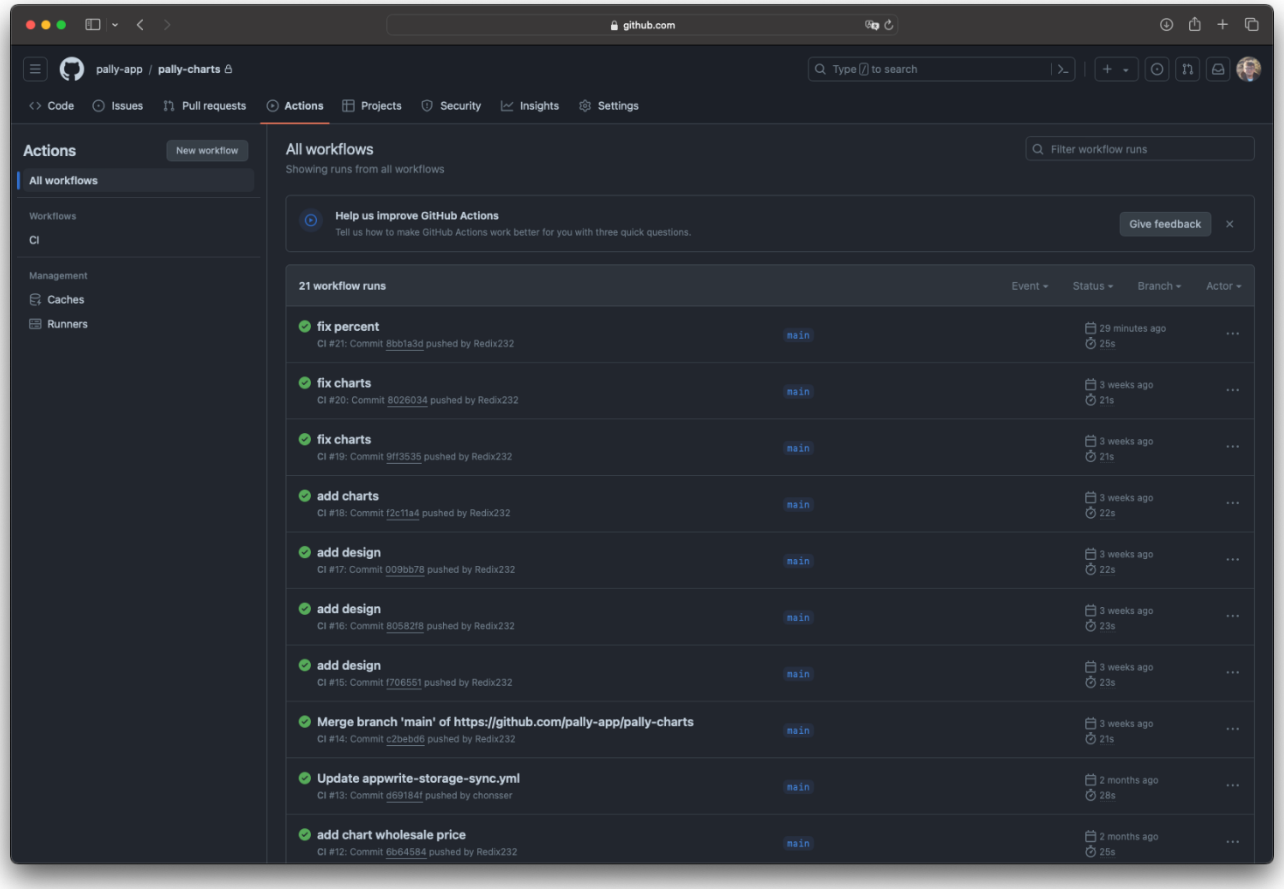
GitHub to serwis hostingowy dla kodu źródłowego, który operuje na systemie Git. Serwis poszerza te funkcje, oferując graficzny interfejs użytkownika (GUI) dla wielu opcji Git, co usprawnia zarządzanie projektem i współpracę. Dodatkowo oferuje takie funkcje jak żądania ściągnięcia (ang. *pull requests*) w trakcie którego można przejrzeć nanoszone zmiany w kodzie źródłowym (ang. *code review*) oraz **Github Actions** które pozwalają na automatyzacje cyklu życia oprogramowania.



Rysunek 2.11 Logo Github (źródło: *http://github.com*)

Rysunek 2.12 Logo Github Actions (źródło: *https://github.com/actions*)

Funkcjonalność pozwala na wdrożenie procesu CI/CD (ang. *continuous integration / continuous delivery*): ciągłą integrację i ciągłe dostarczanie (CI/CD), co było przydatnym aspektem w projekcie.



Rysunek 2.13 Przykład działania Github Actions (źródło: *opracowanie własne*)

Do organizacji i planowania projektu wykorzystano **Jira** [26]: oprogramowanie dostarczone przez firmę Atlassian. Pozwala ona na efektywne planowanie, śledzenie i zarządzanie zadaniami, co przyczynia się do zwiększania produktywności zespołu.

Rysunek 2.14 Logo Jira (źródło: *http://atlassian.com*)

Do Jira jest dostępnych wiele wtyczek (ang. *plug-in*). W projekcie zdecydowano się na wykorzystanie wtyczki BigGantt, która na podstawie istniejących zadań w systemie Jira potrafi tworzyć wykresy Gantta.

Rysunek 2.15 Logo BigGantt (źródło: *https://en.wikipedia.org/wiki/BigGantt*)

Do komunikacji w zespole wykorzystano aplikację **Discord** [15]. Jest to platforma komunikacyjna, która umożliwia zespołom prowadzenie rozmów głosowych, wideo oraz tekstowych. Discord oferuje integracje z Github poprzez webhooki. Automatycznie informują zespół o nowych commitach i zmianach, co ułatwia zarządzanie projektem i współpracę.

Rysunek 2.16 Logo Discord (źródło: *https://discord.com*)

Jako główne zintegrowane środowisko programistyczne (ang. *Integrated Development Environment*) wybrano **Visual Studio Code** [40].
Jest to zaawansowany edytor kodu źródłowego, stworzony przez firmę Microsoft. Swoją popularność zawdzięcza jego wszechstronności (szeroka gama wtyczek tworzonych przez społeczność) i bogaty zestaw funkcji.

Rysunek 2.17 Logo Visual Studio Code (źródło: *https://code.visualstudio.com/brand*)

Python [37] to wszechstronny język programowania. W projekcie został głównie użyty do przetwarzania i analizy danych w komponencie Appwrite Functions. Dzięki bibliotekom takim jak *Beautiful Soup, Python* umożliwia efektywny scrapping danych z internetu i ich przetwarzanie.

Rysunek 2.18 Logo języka Python (źródło: *www.python.org*)

Oprócz wcześniej wymienionego Pythona, użyto również **JavaScript**. W projekcie wykorzystano serwerową wersję JavaScript, czyli **NodeJS** [33], do stworzenia programów do scrappingu danych. Dzięki swojej wydajności i wsparciu dla asynchronicznych operacji, idealnie pasował do tego zadania, zapewniając szybkie przetwarzanie danych.

Rysunek 2.19 Logo języka JavaScript (źródło: *https://github.com/voodootikigod/logo.js*)

Docker to oprogramowanie służące do konteneryzacji. Pozwala on na uruchomienie aplikacji (np. Appwrite) na podstawie gotowego obrazu. Aplikacja zostaje umieszczona w odizolowanym kontenerze (przypominającym maszynę wirtualną, choć nią nie jest), co pozwala na niezależną od systemu hosta konfigurację środowiska wykonawczego, zwiększając tym samym mobilność oraz łatwość wdrażania aplikacji w różnych środowiskach. Kontenery Docker mogą komunikować się ze sobą przez zdefiniowane sieci i współdzielić zasoby, co ułatwia skalowanie i zarządzanie aplikacjami, podobnie jak maszyny wirtualne.

Rysunek 2.20 Logo Docker (źródło: *www.docker.com*)

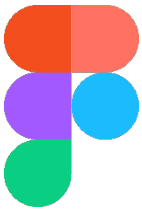
DigitalOcean [14] to popularny dostawca usług chmurowych, który oferuje prostotę, skalowalność i niskie koszty dla deweloperów i firm. Platforma ta umożliwia szybkie tworzenie i zarządzanie wirtualnymi maszynami (ang. *droplets*), które mogą być wykorzystywane jako serwery dla aplikacji webowych, baz danych, aplikacji backendowych i innych usług. DigitalOcean wyróżnia się intuicyjnym interfejsem użytkownika i prostym procesem wdrażania, co sprawia, że jest dostępny nawet dla początkujących użytkowników.

Rysunek 2.21 Logo DigitalOcean (źródło: *www.digitalocean.com*)

ChatGPT [8] to zaawansowany model językowy opracowany przez firmę OpenAI, który może być wykorzystywany do różnych celów, w tym do analizy i interpretacji treści zdjęć.
API CLIP (ang. *Contrastive Language–Image Pre-training*) to narzędzie łączące zaawansowane technologie przetwarzania języka naturalnego (ang. *natural language processing*) z rozpoznawaniem obrazów. Model ten został wytrenowany na ogromnej liczbie obrazów i opisów tekstowych, co pozwala mu na zrozumienie zawartości obrazu poprzez analizę i interpretację tekstu z nim powiązanego. Dzięki temu CLIP potrafi nie tylko identyfikować przedmioty i sceny na zdjęciach, ale również rozumieć bardziej złożone konteksty (np. odczytać cenę poszczególnych paliw z zdjęcia pylonu paliwowego).

Rysunek 2.22 Logo OpenAI (źródło: *www.openai.com*)

Figma [17] to oprogramowanie do tworzenia interfejsów użytkownika (ang. *user interface*) i planowania doświadczenia użytkownika (ang. *user experience*), które służy głównie do tworzenia makiet oraz designu w projektach. Dzięki niej zostały stworzone makiety oraz ścieżki użytkownika w aplikacji (czyli co może w niej zrobić).



Rysunek 2.23 Logo Figma (źródło: *www.figma.com*)

Android Studio [1] to zintegrowane środowisko programistyczne stworzone przez firmę Google do pisania aplikacji mobilnych. Program zawiera bardzo dobre wsparcie dla platformy Flutter, co było kluczowe sukcesu dla projektu.

Rysunek 2.24 Logo Android Studio (źródło: *https://developer.android.com/studio*)

SendGrid [38] to platforma do wysyłania e-maili, która umożliwia deweloperom łatwe zarządzanie i wysyłanie wiadomości w masowej skali. Oferuje wysokiej jakości interfejs API, który jest dobrze udokumentowany i zapewnia szeroki zakres funkcji, co sprawiło, że został wybrany do obsługi e-mailów w projekcie.

Rysunek 2.25 Logo SendGrid (źródło: *https://sendgrid.com/en-us/resource/brand*)

2.3 Geneza i uzasadnienie wyboru tematu

W obszarze zarządzania paliwem, rynek oferuje szereg rozwiązań, jednak brakuje narzędzia, które zapewniałoby kompleksowy dostęp do informacji w łatwo dostępnej formie. Rozwiązania obecnie dostępne na rynku można zasadniczo podzielić na dwie główne kategorie.

Pierwszą kategorię stanowią narzędzia analityczne, najczęściej w formie aplikacji webowych, skoncentrowane na dostarczaniu danych dotyczących cen hurtowych i detalicznych. Są one bardzo mocno rozbudowanymi platformami i codzienne korzystanie z nich nie jest tak przyjemne, jak z współczesnych aplikacji mobilnych. Ze względu na swój poziom skomplikowania, próg wejścia, żeby skorzystać z takiej aplikacji jest bardzo wysoki. Docelowo przekłada się to na małą bazę użytkowników, co również obniża wartość takiej platformy (brak aktualnych danych, nie ma dla kogo ich aktualizować).

Drugą kategorię tworzą aplikacje mobilne, które koncentrują się na zarządzaniu pojazdami, monitorowaniu zużycia paliwa i udostępnianiu informacji o bieżących cenach detalicznych paliw. Te aplikacje, skupiając się wyłącznie na cenach detalicznych, nie oferują pełnego obrazu sytuacji na rynku paliw.

W dalszej części zostanie przedstawiona lista popularnych aplikacji z obu kategorii, ukazując ich kluczowe funkcje, grupę docelową i ograniczenia.

> **Fuelo.net** [9] to serwis internetowy oferujący obszerną bazę danych obejmującą ponad 50 000 stacji benzynowych z 23 krajów Europy. Niestety, serwis nie posiada aplikacji mobilnej dla polskiego rynku. Aplikacja mobilna została przygotowana tylko dla pierwotnej grupy docelowej, czyli rynku bułgarskiego. Stanowi to poważne ograniczeniem, szczególnie w czasach, gdy dominują smartfony, a intuicyjność i szybkość dostępu do informacji są na wagę złota. Polscy użytkownicy, którzy są przyzwyczajeni do szybkiego dostępu do potrzebnych informacji przez kilka prostych kliknięć, mogą być zniechęceni do korzystania z serwisu, który wymaga od nich korzystania ze strony internetowej, co może być mniej wygodne na mniejszych ekranach swoich urządzeń mobilnych. Ponadto, kwestia przestarzałych informacji o cenach paliw w Polsce dodatkowo obniża użyteczność platformy dla polskich użytkowników.

Fuelio [21], jest aplikacją mobilną na system Android (brak wersji na iOS). Skupia się ona na lokalizowaniu stacji paliw i umożliwia aktualizację cen paliw (E5, E10, ON) przez użytkowników po rejestracji. Mimo, że na pierwszy rzut oka aplikacja zapewnia prostotę i bezpośredni dostęp do informacji o cenach, aplikacja posiada swoje ograniczenia.
Skupienie się wyłącznie na indywidualnych konsumentach i ignorowanie szerszych czynników wpływających na ceny paliw, takich jak ceny hurtowe czy zmiany cen ropy naftowej, może prowadzić do niepełnego zrozumienia rynku paliw przez użytkowników. Brak wglądu w te aspekty sprawia, że aplikacja oferuje ograniczoną perspektywę, co może nie satysfakcjonować użytkowników zainteresowanych głębszą analizą rynku.
Ponadto, opieranie się wyłączenie na aktualizacjach cen przez użytkowników może prowadzić do niespójności w dostępnych danych, co w konsekwencji może wprowadzać innych użytkowników w błąd. To z kolei podważa zaufanie do aplikacji jako wiarygodnego źródła informacji.
Brak funkcjonalności do analizy wpływu zewnętrznych czynników (rynkowych cen paliw i jego składowych, takich jak np. ropa) wyklucza grupę bardziej zaawansowanych użytkowników, którzy poszukują jednej, kompleksowej aplikacji.

Waze [41] jest aplikacją mobilną przede wszystkim skoncentrowaną na nawigacji i informowaniu o ruchu drogowym, oferującą również funkcjonalność wyświetlania cen paliw na różnych stacjach. Chociaż ta dodatkowa funkcja może wydawać się przydatna dla użytkowników szukających ekonomicznych opcji tankowania, to jednak nie jest ona głównym celem Waze, co może wpływać na aktualność i dokładność prezentowanych danych o cenach paliw. Autorzy aplikacji skupiają na zapewnieniu informacji o ruchu drogowym i optymalizacji tras.

Nie jest całkowicie jasne, skąd Waze czerpie informacje o cenach paliw. Możliwe, że są one aktualizowane przez społeczność użytkowników lub pozyskiwane z zewnętrznych baz danych, jednak brak przejrzystości co do źródła tych danych i metody ich weryfikacji może budzić wątpliwości co do ich wiarygodności. Użytkownicy, którzy polegają na tych informacjach przy podejmowaniu decyzji o tankowaniu, mogą być narażeni na ryzyko, że kierują się nieaktualnymi cenami, co w efekcie może prowadzić do nieoptymalnych decyzji.

Autocentrum.pl [2] wyróżnia się rozbudowaną bazą danych stacji paliw i aktywną społecznością, która regularnie aktualizuje ceny paliw, co stanowi cenny zasób dla kierowców. Niestety, brak dedykowanej aplikacji mobilnej znacząco ogranicza wygodę korzystania z serwisu. W dobie dominacji smartfonów, użytkownicy oczekują szybkiego i intuicyjnego dostępu do informacji, co nie zawsze jest możliwe przez przeglądarkę mobilną, szczególnie w podróży.

3. Cel i zakres projektu

3.1 Cel programu

Celem projektu jest stworzenie aplikacji mobilnej, która będzie łączyć w sobie zalety aplikacji konkurencyjnych (istniejące aplikacje na rynku, posiadające bogaty zasób danych i rozwiązywać ich mankamenty (toporny interfejs, brak dostępu aplikacji na platformy mobilne).

Główna idea stojąca za programem to dostarczenie użytkownikowi narzędzia do pomocy w wyborze stacji paliwa. Interfejs ma pozwolić wybrać najtańsze paliwo w okolicy. Oprócz tego, użytkownik ma mieć również możliwość podejrzenia wykresów powiązanych z cenami paliw na rynku (np. ceny ropy na giełdzie), co pozwoli mu na wybranie najlepszego (jego zdaniem) momentu na dokonania tankowania swojego pojazdu.

Jedną z zaawansowanych funkcjonalności aplikacji jest analiza zdjęć pylonów stacji benzynowych za pomocą API ChatGPT. Ta opcja pozwala na szybką i efektywną ekstrakcję cen z fotografii, co automatyzuje zbieranie danych i zwiększa ich wiarygodność. Dodatkowo, dynamiczne aktualizacje wykresów cen paliw, realizowane za pomocą biblioteki Chart.js, dostarczają użytkownikom narzędzie do dokonania analizy, które pozwala określić trendy cenowe. Wykresy te stanowią również atrakcyjny i interaktywny element prezentacji danych.

Projekt demonstruje wykorzystanie nowoczesnych technologii w praktyce. Skupia się na zastosowaniu dobrych praktyk programistycznych oraz na prezentacji możliwości oferowanych przez *Flutter, Appwrite*, *Mapbox, Chart.js* i inne wykorzystane narzędzia. Celem jest nie tylko dostarczenie wartościowej aplikacji użytkownikom, ale również pokazanie kompleksowego podejścia do wytwarzania i testowania oprogramowania w kontekście najnowszych trendów w branży IT.

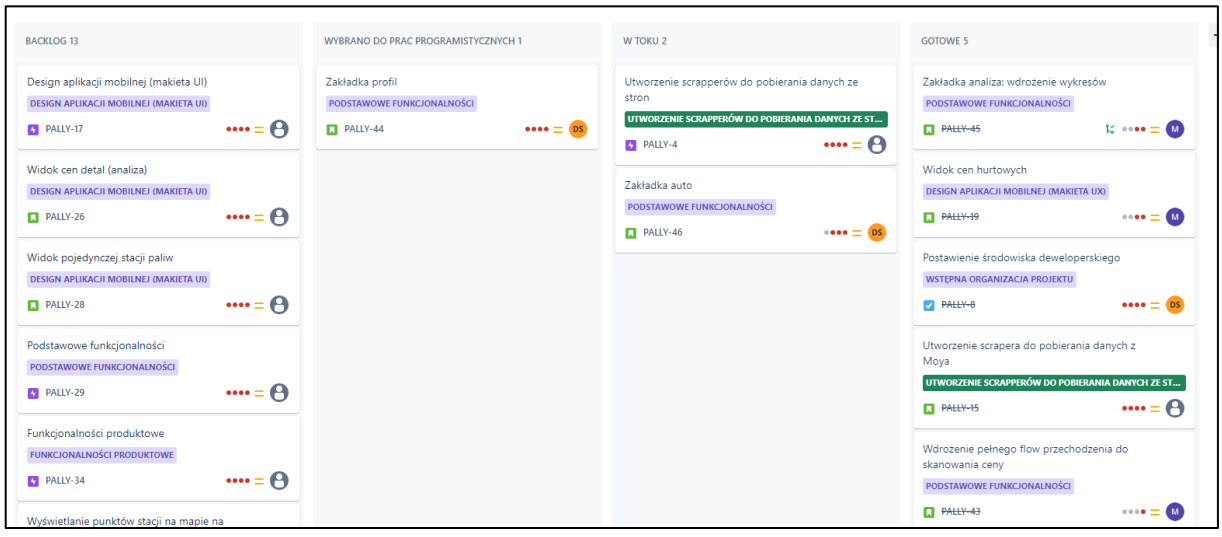
3.2 Podział zadań w projekcie

W celu zapewnienia efektywnego procesu tworzenia projektu, przyjęto zwinne metodyki pracy, które łączą w sobie najlepsze praktyki inżynierii oprogramowania z zastosowaniem nowoczesnych technologii informatycznych. Metodyka ewoluowała w projekcie: początkowo był to uproszczony Scrum, a w późniejszym etapie projektu przekształciło się to w tablicę Kanban.

Scrum [10] to zwinna metodyka zarządzania projektem, która dzieli proces tworzenia oprogramowania na iteracje, trwające od 1 do 4 tygodni, zwane sprintami. Każda iteracja ma wcześniej określony cel sprintu (ang. *sprint goal*). W początkowej fazie projektu uznano, że Scrum będzie najlepszym rozwiązaniem ze względu na utrzymanie ciągłej komunikacji w zespole. Codziennie ceremonie (ang. *stand-up*) zakładają dzielenie się informacjami o postępach prac przez każdego członka zespołu w kilku zdaniach. W dalszym etapie projektu zdecydowano porzucić Scrum na rzecz Kanban z powodu jego czasochłonności Codziennie ceremonie Scrum pochłaniały stosunkowo dużo czasu, który mógł być przeznaczony na realizacje zadań.

Kanban to zwinna metodyka, która polega na wizualizacji pracy za pomocą tablicy. W projekcie wykorzystano Jirę do cyfrowej implementacji tablicy Kanban, co pozwala na efektywne śledzenie postępu prac i zarządzanie przepływem zadań.

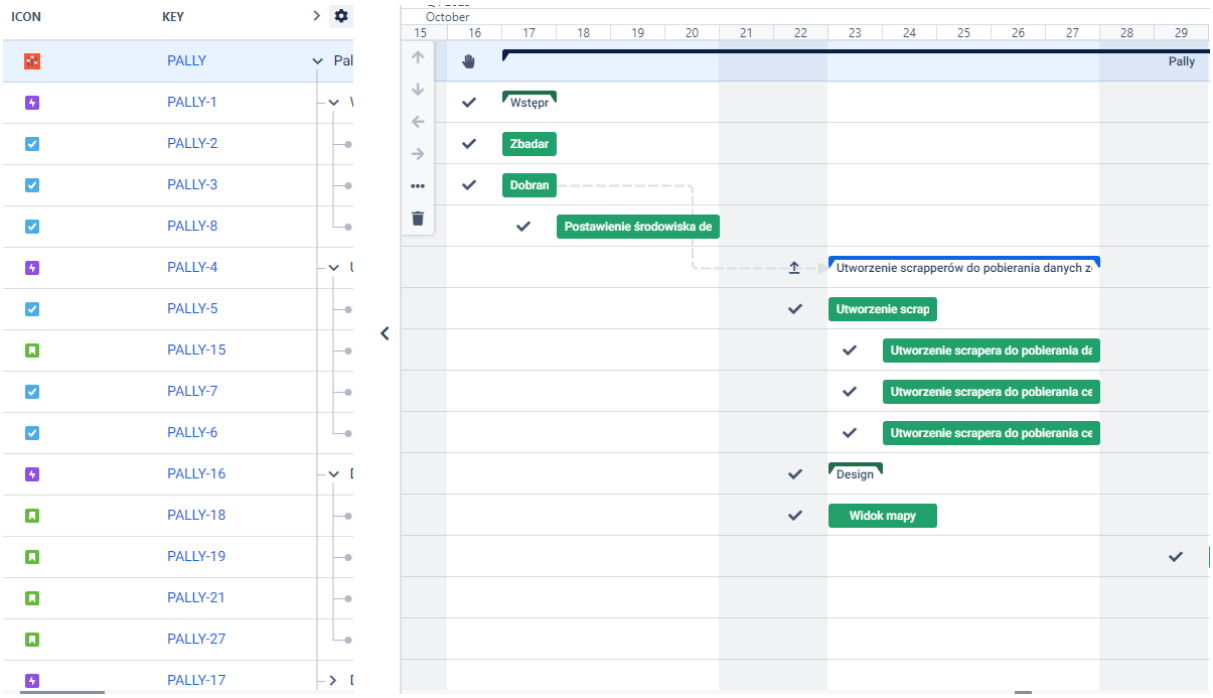
Tablica Kanban w Jirze umożliwia łatwe monitorowanie etapów prac od „Backlog” przez „Wybrano do prac programistycznych”, „W toku” do „Gotowe”. Dzięki temu każdy członek zespołu może na bieżąco śledzić postępy oraz identyfikować ewentualne zatory w przepływie pracy.



Rysunek 3.1: Kanban w aplikacji Jira (źródło: *opracowanie własne*)

W celu szczegółowego planowania sprintów i przydzielenia zadań w ramach projektu, jak również na monitorowanie zależności między zadaniami zastosowano wtyczkę BigGantt.

Głównym komponentem tej wtyczki jest widok harmonogramu, z której można w łatwy sposób ustalić kamienie milowe pracy oraz ustalać kluczowe terminy zakończenia poszczególnych etapów. Dodatkowo pozwala on na wyświetlenie zależności poszczególnych zadań (np. jedno zadanie wymaga ukończenia przed rozpoczęciem pracy nad kolejnym). Dzięki temu można w łatwy i szybki sposób dostosować harmonogram projektu i zaktualizować go w razie potrzeby.



Rysunek 3.2: Widok harmonogramu wtyczki BigGantt w aplikacji Jira (źródło: *opracowanie własne*)

Organizacja i podział zadań w projekcie:

1. Mateusz Wiemann – Backend developer, Frontend developer

Praca w projekcie – odpowiedzialny za:

- Implementacja systemu logowania i rejestracji
- Implementacja funkcji do zbierania cen: hurtowych oraz giełdy
- Implementacja zakładki wykresy oraz funkcji powiązanych
- Implementacja funkcji wyświetlania stacji
- Implementacja funkcji aparatu
- Implementacja testów

Edycja i redakcja pracy:

- Diagramy UML
- Tabele wymagań funkcjonalnych
- Frontend aplikacji
- Testy aplikacji

2. Dominik Szymański: Backend developer, Frontend developer, Application Architect

Praca w projekcie – odpowiedzialny za:

- Project manager – planowanie i rozpisywanie zadań
- Rozszerzenie funkcjonalności wyświetlania stacji
- Design i makiety w Figma
- Architektura aplikacji
- Wdrożenie funkcjonalności rabatów
- Implementacja funkcji do zbierania cen: detal
- Rozwój systemu logowania i rejestracji

Edycja i redakcja pracy:

- Praca edytorska
- Analiza konkurencji

4. Metodyka

4.1 Tabele wymagań funkcjonalnych

Tabele wymagań funkcjonalnych są narzędziem używanym do szczegółowego opisu funkcji i operacji systemu, które muszą być wykonane, aby spełnić potrzeby użytkowników końcowych. Pozwalają na uporządkowaną i klarowną prezentację funkcjonalności, jakie oprogramowanie lub system powinien oferować.

Tabela 4.1 Tabela wymagań funkcjonalnych: rejestracja użytkownika (źródło: *opracowanie własne*)

Nazwa funkcji	**Rejestracja użytkownika**
Opis funkcji	Funkcja umożliwia stworzenie nowego konta użytkownika.
Dane wejściowe	Email, hasło.
Źródło danych wejściowych	Klawiatura telefonu - Pierwsza strona.
Wynik funkcji	Dodanie nowe konta do bazy danych.
Warunek wstępny	Użytkownik poprawnie uzupełnił formularz podając adres email i hasło zawierające co najmniej 8 znaków oraz email nie znajduje się jeszcze w bazie danych.
Warunek końcowy	Użytkownik zostaje dodany do bazy danych oraz ma możliwość korzystania z aplikacji.
Efekt uboczne	Zalogowanie użytkownika do aplikacji.
Powód	Chęć korzystania z aplikacji.

Tabela 4.2 Tabela wymagań funkcjonalnych: reset hasła (źródło: *opracowanie własne*)

Nazwa funkcji	**Reset hasła**
Opis funkcji	Funkcja umożliwia ustawienie nowego hasła dla podanego adresu email.
Dane wejściowe	Email, nowe hasło.
Źródło danych wejściowych	Klawiatura telefonu - Formularz logowania - reset hasła.
Wynik funkcji	Zmiana hasła użytkownika.
Warunek wstępny	Użytkownik rozpoczął proces resetowania hasła poprzez przesłanie na email linku oraz podane przez użytkownika hasło jest zawiera co najmniej 8 znaków.
Warunek końcowy	Nowe hasło jest zgodne z warunkami aplikacji.
Efekt uboczne	Wylogowanie użytkownika z innych urządzeń/sesji
Powód	Chęć zresetowania hasła do aplikacji.

Tabela 4.3 Tabela wymagań funkcjonalnych: logowanie użytkownika (źródło: opracowanie własne)

Nazwa funkcji	**Logowanie użytkownika**
Opis funkcji	Funkcja umożliwia zalogowanie się do aplikacji na wcześniej utworzone konto.
Dane wejściowe	Email, hasło.
Źródło danych wejściowych	Klawiatura telefonu - Pierwsza strona, formularz logowania.
Wynik funkcji	Zalogowanie się na konto znajdujące się w bazie danych.
Warunek wstępny	Użytkownik poprawnie uzupełnił formularz podając swój adres email i hasło które znajdują się w bazie danych.
Warunek końcowy	Hasło pasuje do emaila znajdującego się w bazie danych.
Efekt uboczne	Zalogowanie użytkownika do aplikacji.

Powód	Chęć zalogowania się użytkownika na swoje konto.

Tabela 4.4 Tabela wymagań funkcjonalnych: wylogowanie użytkownika (źródło: opracowanie własne)

Nazwa funkcji	**Wylogowanie użytkownika**
Opis funkcji	Funkcja umożliwia wylogowanie się z aplikacji użytkownika.
Dane wejściowe	Aktywna sesja użytkownika.
Źródło danych wejściowych	Przycisk wyloguj znajdujący się w profilu użytkownika.
Wynik funkcji	Zamknięcie sesji użytkownika i powrót do ekranu logowania.
Warunek wstępny	Użytkownik jest zalogowany i nacisną przycisk wyloguj.
Warunek końcowy	Aktywna sesja użytkownika .
Efekt uboczne	Brak dostępu do części funkcji w aplikacji.
Powód	Chęć zalogowania się użytkownika na swoje konto.

Tabela 4.5 Tabela wymagań funkcjonalnych: monitorowanie cen paliw (źródło: *opracowanie własne*)

Nazwa funkcji	**Monitorowanie cen paliw**
Opis funkcji	Funkcja odpowiada za automatyczne zbieranie i archiwizowanie informacji na temat bieżących cen paliw z wybranych stron internetowych, zapisując dane w formacie *JSON*.
Dane wejściowe	Adres URL strony internetowej, klucze API do połączenia z systemem zarządzania bazą danych *Appwrite*.
Źródło danych wejściowych	Funkcja jest uruchamiana automatycznie w określonych interwałach czasowych, zgodnie z harmonogramem.
Wynik funkcji	Dane dotyczące cen paliw są przetwarzane i zapisywane w pliku *JSON*, który następnie jest przesyłany do określonego *bucket'a* w systemie przechowywania danych.
Warunek wstępny	Harmonogramowany wyzwalacz *cron* jest ustawiony na uruchamianie funkcji raz dziennie, aby zapewnić aktualność danych.
Warunek końcowy	Plik JSON zawierający najnowsze informacje o cenach paliw jest pomyślnie przesyłany i zapisywany na serwerze.
Efekt uboczne	W przypadku braku dostępu do strony internetowej pod podanym *URLem* (awaria strony, błąd sieciowy) program nie zapisuje danych na *buckecie*. Do aktualizacji cen paliw zostanie użyty ostatni prawidłowo pobrany plik.
Powód	Celem funkcji jest regularne aktualizowanie baz danych cen paliw, aby użytkownicy systemu mogli podejmować świadome decyzje zakupowe lub analityczne oparte na najświeższych danych.

Tabela 4.6 Tabela wymagań funkcjonalnych: przekazywanie cen paliw do bazy danych (źródło: opracowanie własne)

Nazwa funkcji	**Przekazywanie cen paliw do bazy danych**
Opis funkcji	Funkcja odpowiedzialna jest za przeszukiwanie plików *JSON*, które zawierają informacje o cenach paliw, zapisane w buckecie. Jej zadaniem jest ekstrakcja danych dotyczących cen paliw i zapisanie ich w bazie danych *Appwrite*.
Dane wejściowe	Plik *JSON* z danymi o cenach paliw.
Źródło danych wejściowych	*Bucket* przechowujący pliki *JSON* pobrane ze stron internetowych zawierających aktualne ceny paliw.
Wynik funkcji	Dane o cenach paliw zostają zapisane w bazie danych *Appwrite*.
Warunek wstępny	W buckecie musi znajdować się aktualny plik *JSON* z danymi o cenach paliw.
Warunek końcowy	Dane cen paliw są dodawane do bazy danych.
Efekt uboczne	Zmiana wykresów i cen w aplikacji.
Powód	Celem funkcji jest automatyzacja procesu aktualizacji cen paliw w bazie danych, co umożliwia użytkownikom dostęp do najnowszych informacji i analiz rynkowych.

Tabela 4.7 Tabela wymagań funkcjonalnych: analiza cen paliw (źródło: opracowanie własne)

Nazwa funkcji	**Analiza cen paliw**
Opis funkcji	Funkcja zapewnia użytkownikom wizualizację aktualnych trendów rynkowych cen paliw poprzez przystępne wykresy, umożliwiające łatwe śledzenie zmian cen oraz porównywanie różnych rodzajów paliw.
Dane wejściowe	Ceny paliw, rodzaj paliwa (np. benzyna, diesel, LPG), typ transakcji (hurt, detal, giełda), oraz zakres dat – wszystko pobrane z bazy danych.
Źródło danych wejściowych	Przejście użytkownika do zakładki analiza.
Wynik funkcji	Prezentacja interaktywnych wykresów w sekcji WebView aplikacji, które użytkownik może dostosowywać, zmieniając wybrane rodzaje paliw i zakres dat.
Warunek wstępny	Dane na temat cen paliw muszą być aktualne i dostępne w bazie danych; użytkownik powinien mieć możliwość wyboru zakresu dat oraz rodzaju paliwa do analizy.
Warunek końcowy	Użytkownik otrzymuje na ekranie wykresy, które jasno przedstawiają zmiany cen paliw w wybranym przez siebie okresie oraz dla wybranych rodzajów paliw.
Efekt uboczne	Brak.
Powód	Głównym celem funkcji jest umożliwienie użytkownikowi łatwego monitorowania i analizowania trendów cenowych paliw, co może pomóc w podejmowaniu lepszych decyzji zakupowych lub inwestycyjnych.

Tabela 4.8 Tabela wymagań funkcjonalnych: odczytanie cen paliw z pylonu (źródło: opracowanie własne)

Nazwa funkcji	**Odczytanie ceny paliw z pylonu**
Opis funkcji	Funkcja umożliwia rozpoznanie i odczytanie cen paliw z pylonu stacji benzynowej z wykonanego przez użytkownika zdjęcia, oferując możliwość weryfikacji i dodania tych cen do bazy danych po zatwierdzeniu przez użytkownika.
Dane wejściowe	Zdjęcie pylonu wykonane aparatem w aplikacji.
Źródło danych wejściowych	Przycisk „Zrób zdjęcie”, aktywowany w aplikacji mobilnej po znalezieniu się użytkownika w promieniu 500 metrów od stacji paliw.
Wynik funkcji	Odczytane ceny paliw są prezentowane użytkownikowi do weryfikacji.
Warunek wstępny	Użytkownik musi zrobić wyraźne zdjęcie pylonu, na którym widoczne są ceny paliw, w dowolnej orientacji.
Warunek końcowy	Użytkownik zatwierdza poprawność odczytanych cen, które następnie są dodawane do bazy danych.
Efekt uboczne	Dodanie do bazy cen przesłanych przez użytkownika, które jeśli zostaną zweryfikowane, zaktualizują ceny paliw w aplikacji, co przyczynia się do utrzymania najbardziej aktualnego przeglądu cen na stacjach benzynowych.
Powód	Celem jest zapewnienie użytkownikom dostępu do najnowszych i zweryfikowanych cen paliw, co ułatwia porównywanie ofert różnych stacji benzynowych i podejmowanie bardziej świadomych decyzji zakupowych.

Tabela 4.9 Tabela wymagań funkcjonalnych: wyświetlanie stacji benzynowych na mapie (źródło: opracowanie własne)

Nazwa funkcji	**Wyświetlanie stacji benzynowych na mapie**
Opis funkcji	Funkcja umożliwia wyświetlenie na mapie, dostępnej w aplikacji, stacji benzynowych zlokalizowanych w pobliżu użytkownika, wraz z ich cenami paliw oraz logiem stacji.
Dane wejściowe	Dostęp do lokalizacji użytkownika.
Źródło danych wejściowych	Telefon użytkownika - po wcześniejszym zezwoleniu na dostęp do lokalizacji.
Wynik funkcji	Na mapie, w aplikacji, są wyświetlane stacje benzynowe znajdujące się w pobliżu użytkownika, wraz z informacjami o cenach paliw oraz z logotypami stacji.
Warunek wstępny	Użytkownik wyraził zgodę na używanie lokalizacji przez aplikację i znajduje się na terenie Polski lub w jej pobliżu.
Warunek końcowy	Użytkownik nie zmienił swojej lokalizacji.
Efekt uboczne	Możliwość pojawienia się przycisku „Zrób zdjęcie” jeśli znajdujemy się w pobliżu stacji.
Powód	Głównym celem funkcji jest ułatwienie użytkownikom lokalizowania stacji benzynowych w ich bezpośrednim otoczeniu oraz porównywanie cen paliw, co pozwala na oszczędność czasu i środków przez wybór najbardziej optymalnej oferty.

Tabela 4.10 Tabela wymagań funkcjonalnych: wyświetlanie lokalizacji użytkownika na mapie (źródło: opracowanie własne)

Nazwa funkcji	**Wyświetlanie lokalizacji użytkownika na mapie**
Opis funkcji	Funkcja umożliwia wyświetlanie lokalizacji użytkownika na mapie po wcześniejszym wyrażeniem zgody dostępu do lokalizacji.
Dane wejściowe	Dostęp do lokalizacji użytkownika.
Źródło danych wejściowych	Telefon użytkownika - po wcześniejszym zezwoleniu na dostęp do lokalizacji.
Wynik funkcji	Na mapie, w aplikacji, wyświetlany jest punkt wskazujący na to gdzie znajduje się użytkownik wraz z możliwością przybliżenia do jego lokalizacji.
Warunek wstępny	Użytkownik wyraził zgodę na używanie lokalizacji przez aplikację.
Warunek końcowy	Użytkownika ma uruchomioną lokalizacje w telefonie.
Efekt uboczne	Możliwość pojawienia się przycisku „Zrób zdjęcie” jeśli znajdujemy się w pobliżu stacji oraz wyświetlenie się stacji benzynowych w pobliżu.
Powód	Głównym celem funkcji jest zlokalizowanie użytkownika w celu wyświetlenia mu stacji w pobliżu oraz możliwość zrobienia zdjęcia pylonowi jeśli jest w pobliżu.

Tabela 4.11 Tabela wymagań funkcjonalnych: analiza zdjęcia przez chat GPT (źródło: opracowanie własne)

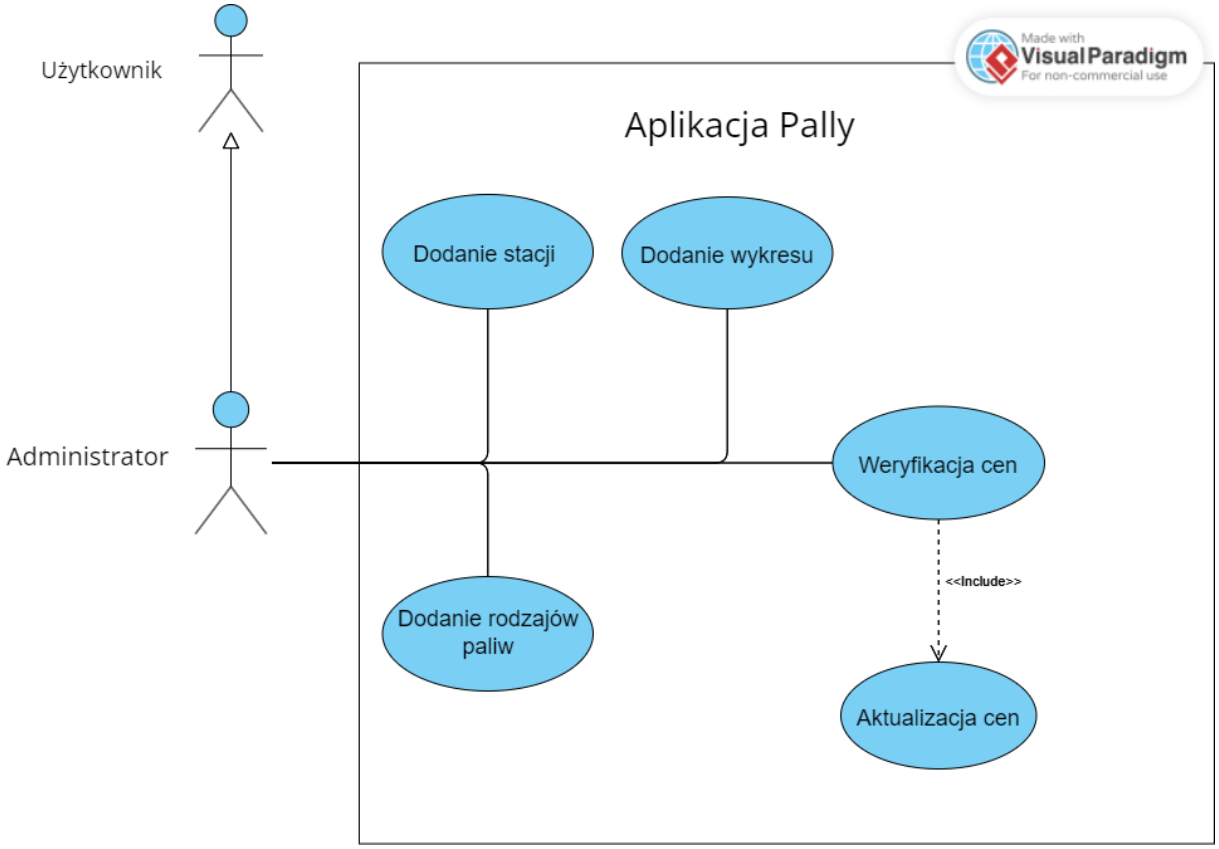
Nazwa funkcji	**Analiza zdjęcia przez ChatGPT**
Opis funkcji	Funkcja umożliwia analizowanie zdjęć pylonów wraz z przesłaniem cen znajdujących się na pylonach.
Dane wejściowe	Przesłanie zdjęcia do ChataGPT
Źródło danych wejściowych	Aparat telefonu, po naciśnięciu przycisku „Zrób zdjęcie”
Wynik funkcji	Zwrócenie poszczególnych ceny znajdujących się na pylonie.
Warunek wstępny	Użytkownik wyraził zgodę na dostęp do aparatu.
Warunek końcowy	Użytkownik zrobił wyraźne zdjęcie pylonu paliwowego.
Efekt uboczne	Wyświetlenie cen w aplikacji w celu zatwierdzenia przez użytkownika oraz dodanie zdjęcia do bazy danych.
Powód	Głównym celem funkcji jest szybkie i realne pobieranie cen z pylonów stacji paliw dzięki analizie zdjęć przez ChatGPT. Ceny przesłane przez użytkownika będą bardziej rzetelne, a proces szybszy niż ręczne wprowadzanie cen.

4.2 Diagramy przypadków użycia

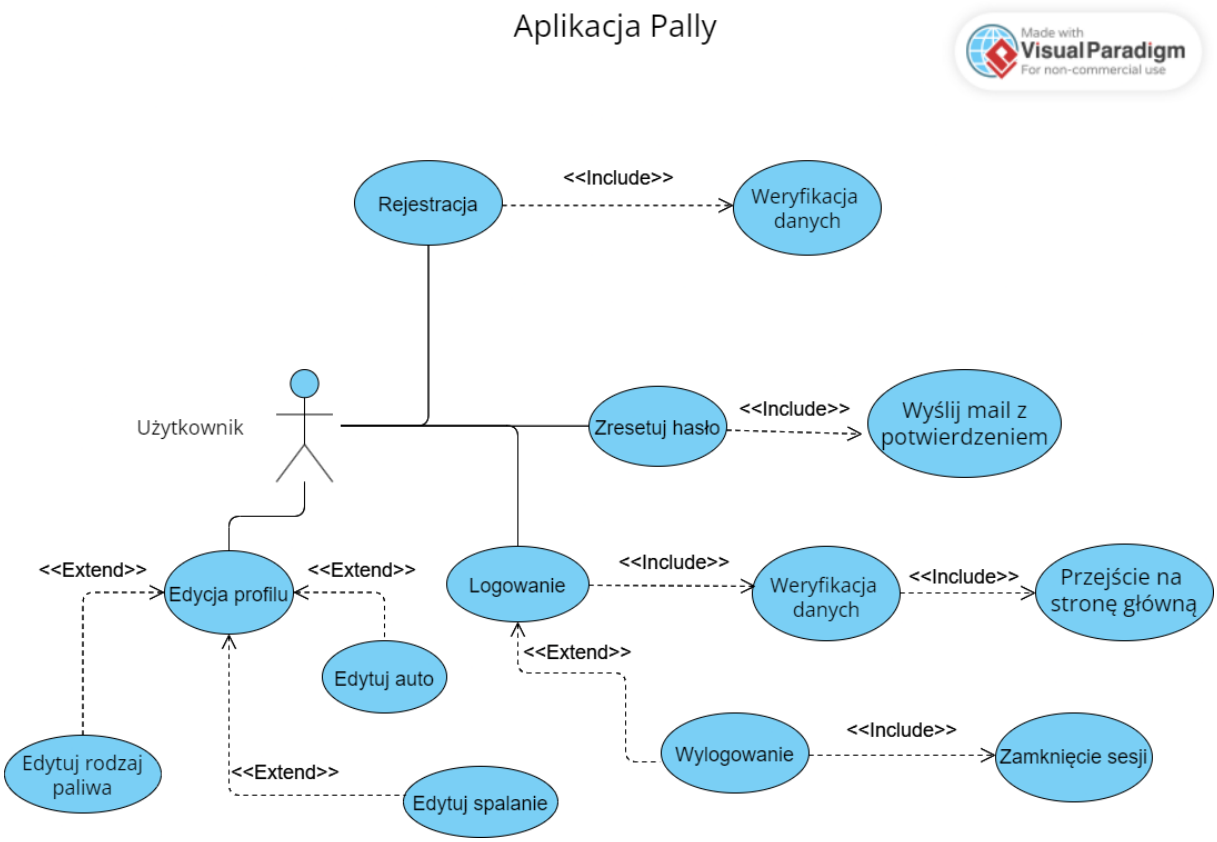
Diagram UML (ang. *Unified Modelling Language*) jest standardowym sposobem wizualizacji projektowania systemu, umożliwiającym inżynierom oprogramowania i projektantom systemów przedstawienie, określenie specyfikacji, konstrukcji oraz dokumentację artefaktów oprogramowania. UML jest językiem modelowania, który zapewnia standardowe metody do przedstawiania struktury i zachowania systemu oprogramowania.

Diagram przypadków użycia w UML to diagram behawioralny, który ilustruje możliwe interakcje między aktorami (użytkownikami lub innymi systemami), a analizowanym systemem. Jest on skonstruowany z przypadków użycia, definiujących konkretne akcje możliwe do wykonania przez system oraz z aktorów, którzy te akcje inicjują.

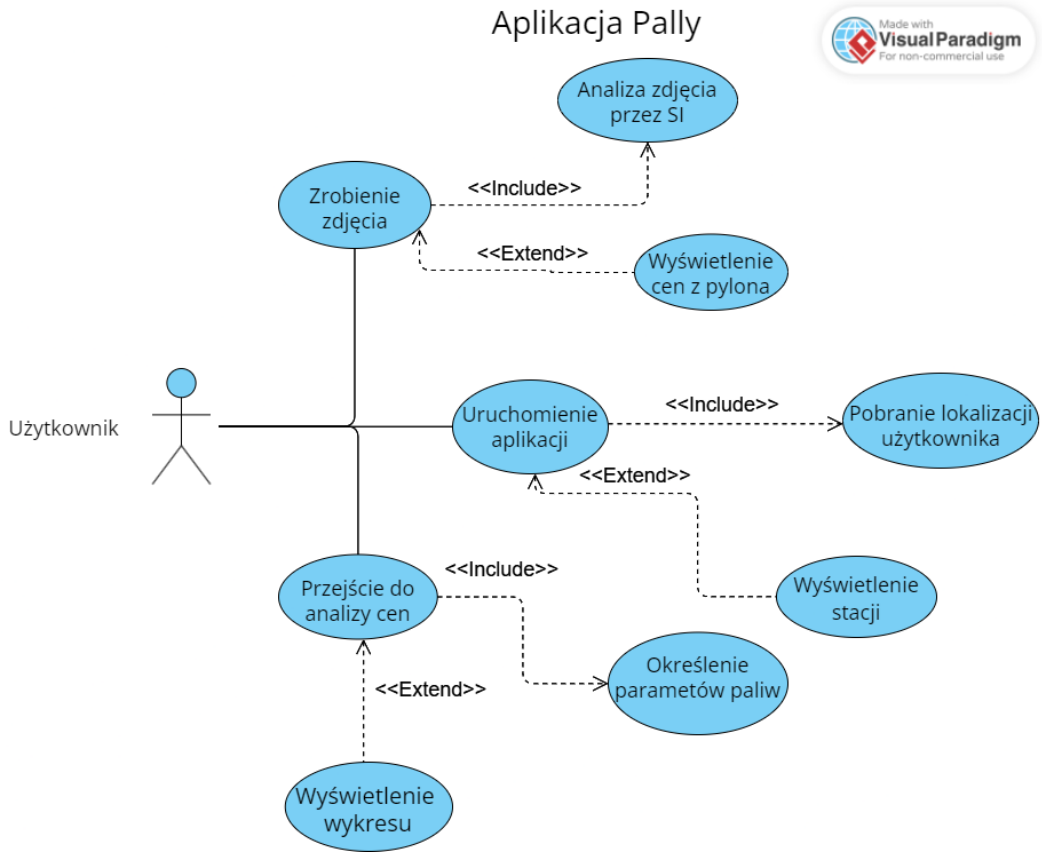
Diagram przypadków użycia administratora aplikacji



Rysunek 4.1 Diagram przypadków użycia administratora aplikacji (źródło: *opracowanie własne*)



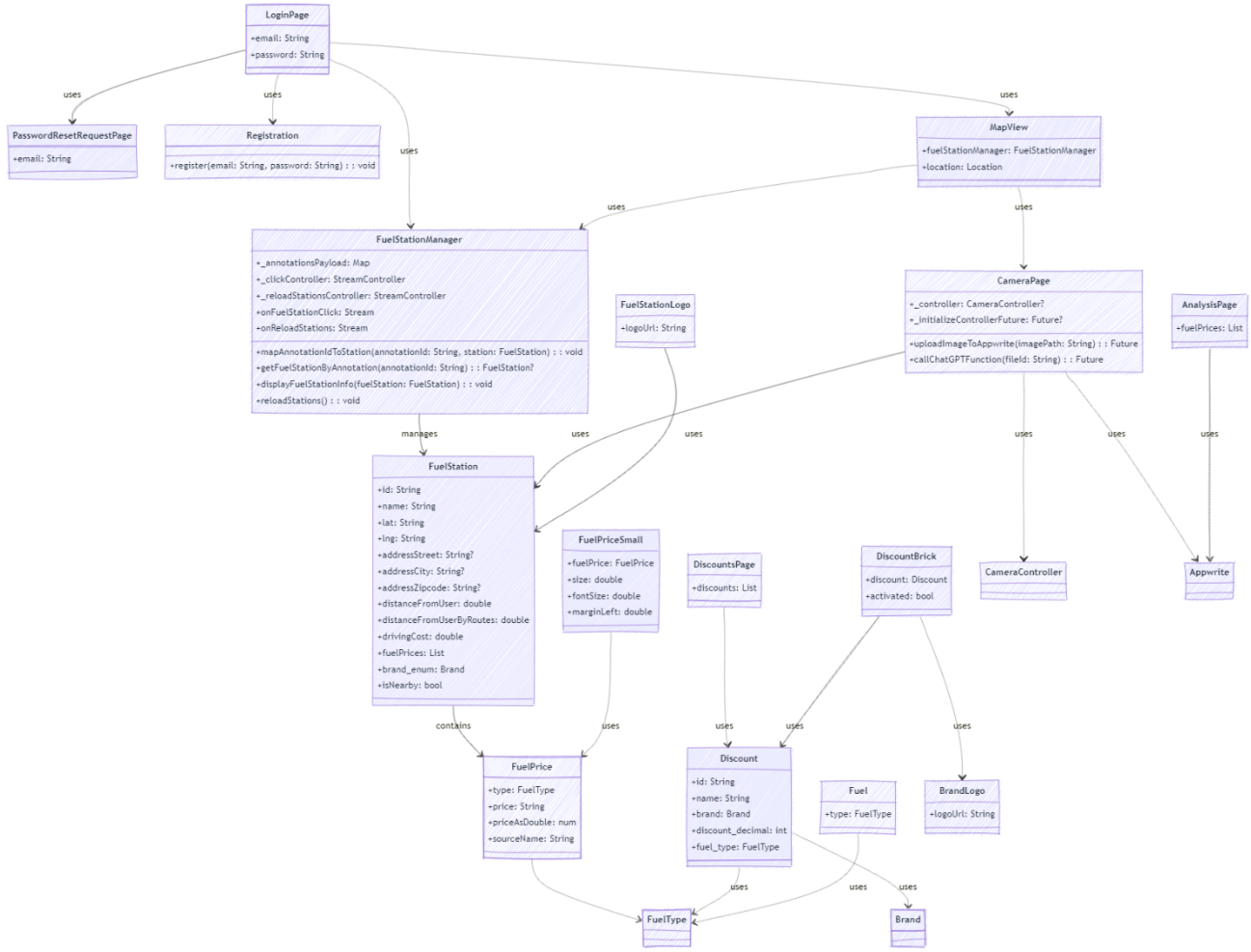
Rysunek 4.2 Diagram przypadków użycia użytkownik aplikacji (źródło: *opracowanie własne*)



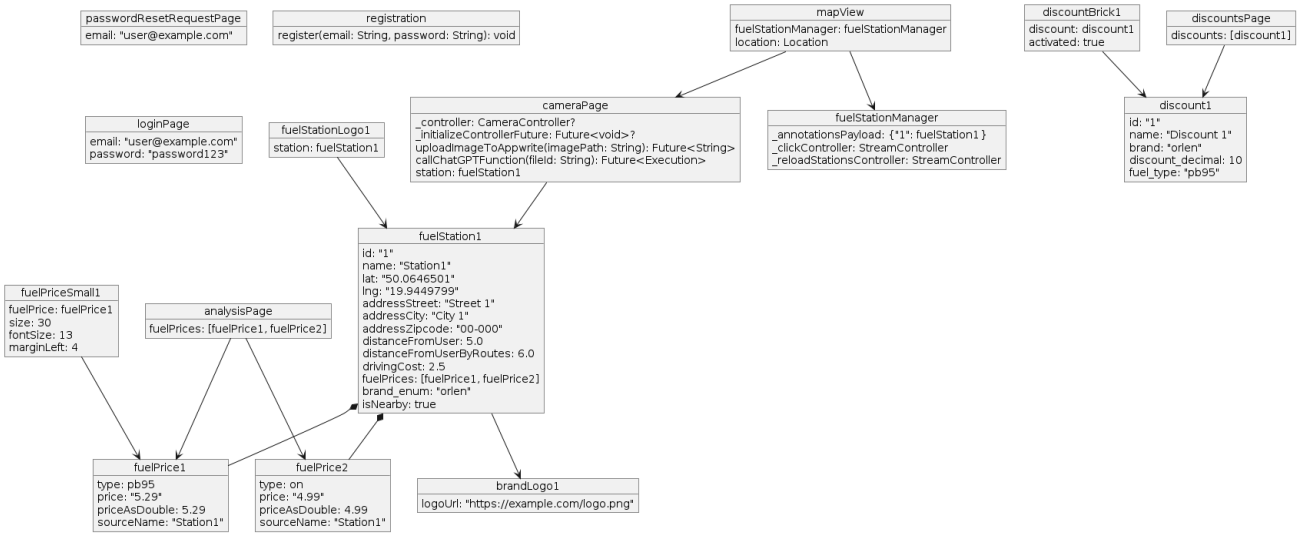
Rysunek 4.3 Diagram przypadków użycia użytkownika aplikacji (źródło*:* opracowanie własne)

4.3 Diagramy klas i diagramy obiektów

Diagram klas jest fundamentalnym elementem w obiektowym projektowaniu systemów, służącym do przedstawiania struktury systemu poprzez określenie klas, ich atrybutów oraz metod. Każda z klas symbolizuje pewien obiekt lub pojęcie, przy czym jej atrybuty oraz metody opisują charakterystykę i sposoby działania tego obiektu. Diagram ten ukazuje również relacje między klasami, takie jak dziedziczenie, asocjacje czy zależności. Stanowi on zatem kluczowe narzędzie nie tylko do projektowania i analizy, ale i do dokumentowania obiektowo zorientowanych systemów.



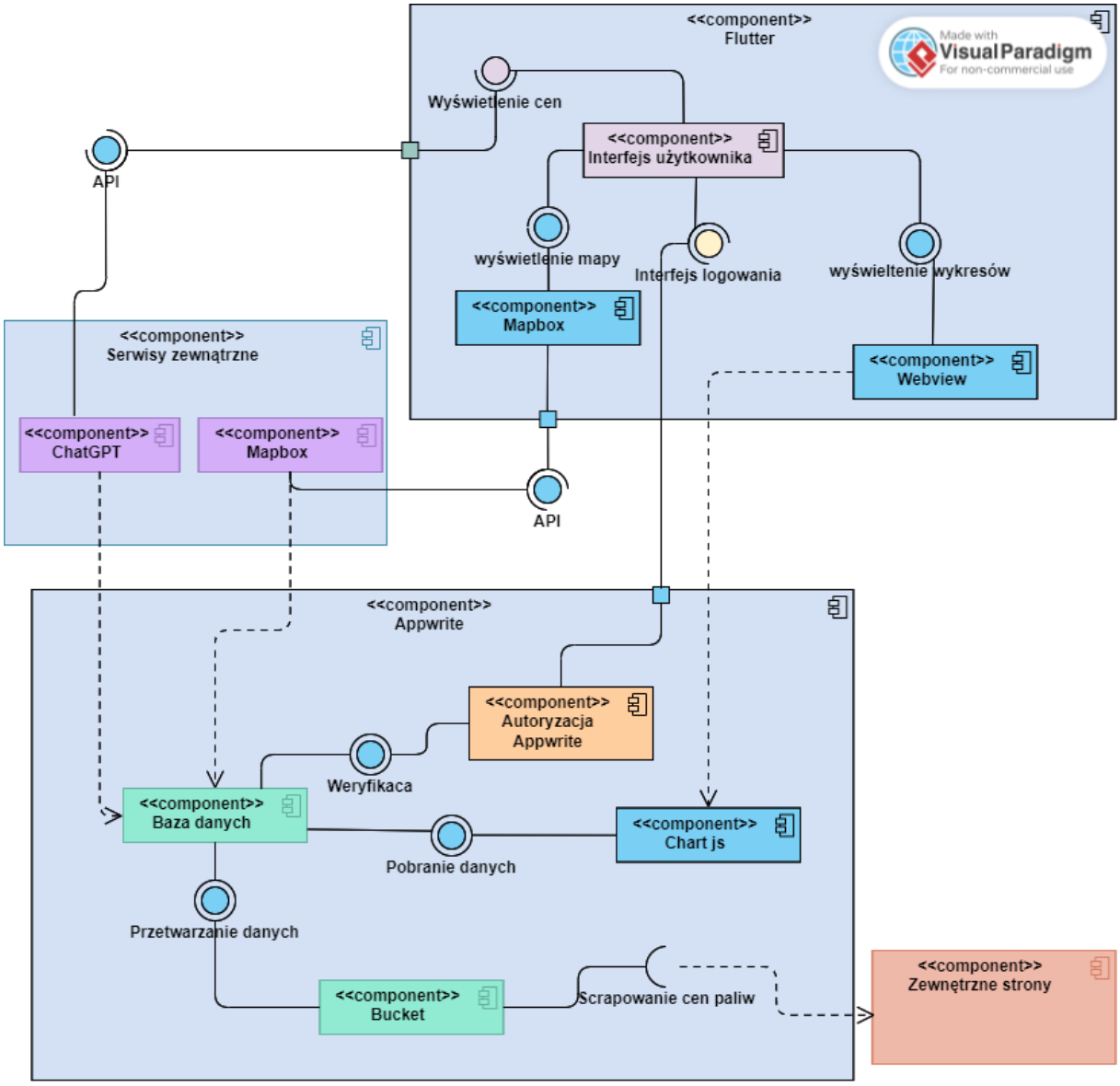
Rysunek 4.4 Diagram klas (źródło: *opracowanie własne*)



Rysunek 4.5 Diagram obiektów (źródło: *opracowanie własne*)

4.4 Diagram komponentów

Diagram komponentów to rodzaj diagramu strukturalnego służącego do ilustrowania struktury organizacyjnej i wzajemnych relacji pomiędzy komponentami systemu. Te komponenty stanowią moduły oprogramowania, charakteryzujące się niezależnością i możliwością wymiany, obejmujące elementy takie jak biblioteki, pakiety czy pliki. Umożliwia on przedstawienie architektury kodu, wskazując, jak poszczególne komponenty współdziałają oraz jak są zorganizowane i połączone. W notacji *UML*, komponenty reprezentowane są przez prostokąty, które mogą zawierać nazwy komponentów, a interfejsy oferowane lub wymagane przez komponent ukazywane są za pomocą półokrągłych lub kwadratowych elementów na obrzeżach prostokąta.

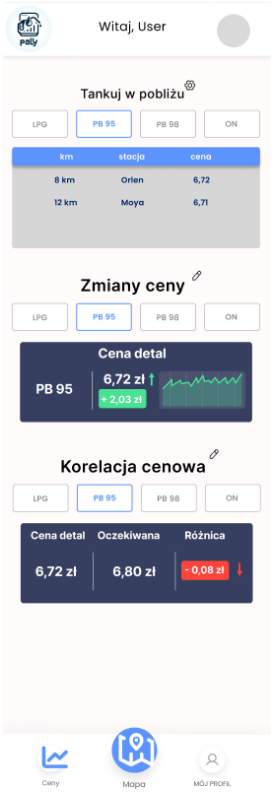


Rysunek 4.6 Diagram komponentów (źródło: *opracowanie własne*)

5. Opis realizacji projektu oraz procedury testowania

5.1 Prototypowanie

Wczesny proces prototypowania rozpoczął się od MVP (ang. *Minimum Viable Product*), czyli minimalnej wersji aplikacji. Pierwsze interfejsy oraz funkcjonalności zostały przygotowane za pomoą Figmy, która była najlepszym wyborem do projektowania UX/UI. Figma zawiera szereg przydatnych funkcji, takich jak możliwość prototypowania, co pozwoliło na testowanie interfejsu użytkownika na wczesnym etapie. Jedną z zasad przy projektowaniu było wykorzystanie prawa Millera [36], które skupia się na minimalizowaniu potrzeby zapamiętywania informacji przez użytkownika.



Rysunek 5.1 Makieta pierwszego ekranu w Figmie (źródło: *opracowanie własne*)

Rysunek 5.2 Makieta przedstawiająca mapę (źródło: *opracowanie własne*)

Rysunek 5.3 Komponent nawigacji z Figmy (źródło: *opracowanie własne*)

Pierwsze makiety skupiały się na najważniejszych funkcjonalnościach, takich jak: mapa, analiza cen paliw oraz odległości od stacji. Proces ten pozwolił wyłonić kluczowe komponenty w aplikacji, takie jak nawigacja, która musi być czytelna oraz użyteczna, aby nie zrazić użytkownika.

Rysunek 5.4 Widok mapy uzupełniony o komponenty (źródło: *opracowanie własne*)

Kolejnym ważnym komponentem, który wymagał dużo pracy, był ekran główny aplikacji zawierający mapę oraz kluczową funkcjonalność sprawdzania i dodawania cen na stacjach benzynowych. Ekran został uzupełniony o dodatkowe informacje, dzięki czemu użytkownik ma wszystkie najważniejsze funkcjonalności pod ręką. Wprowadzono możliwość wyświetlania kafelków do skanowania ceny wraz z informacją dotyczącą stacji, której to dotyczy, w przeciwieństwie do pierwotnej wersji, w której dostępny był jedynie przycisk umożliwiający skanowanie ceny.

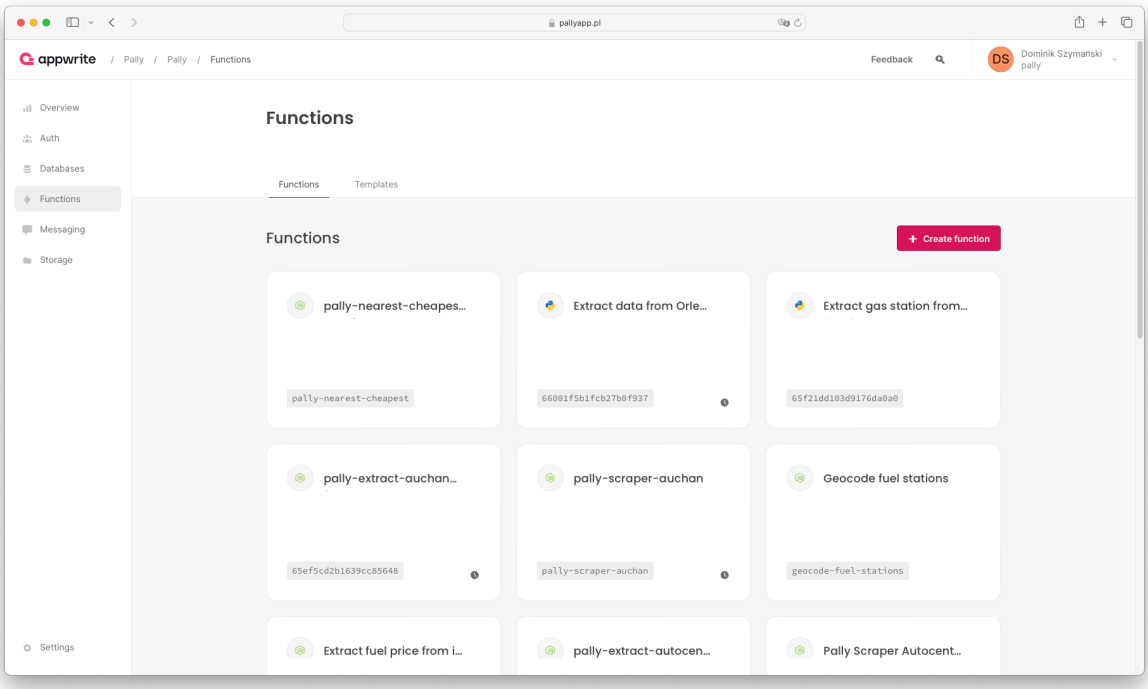
Rysunek 5.5 Widok aparatu w aplikacji (źródło: *opracowanie własne*)

W przypadku użycia aparatu aplikacji miała dawać stałą informację jakiej stacji dotyczy proces w celu uniknięcia niepewności użytkownika w przypadku znajdywanie się w pobliżu dwóch różnych stacji.

5.2 Backend

Backend aplikacji w całości opiera się na platformie Appwrite. Główne założenie platformy backendowej było takie, żeby wspierało wiele środowisk uruchomieniowych (z naciskiem na Python i NodeJS z uwagi na wcześniejsze doświadczenie członków zespołu z tymi technologiami).

Przy wykorzystaniu Appwrite Functions stworzono szereg funkcji, które są wykorzystane w aplikacji mobilnej do prawidłowego działania.

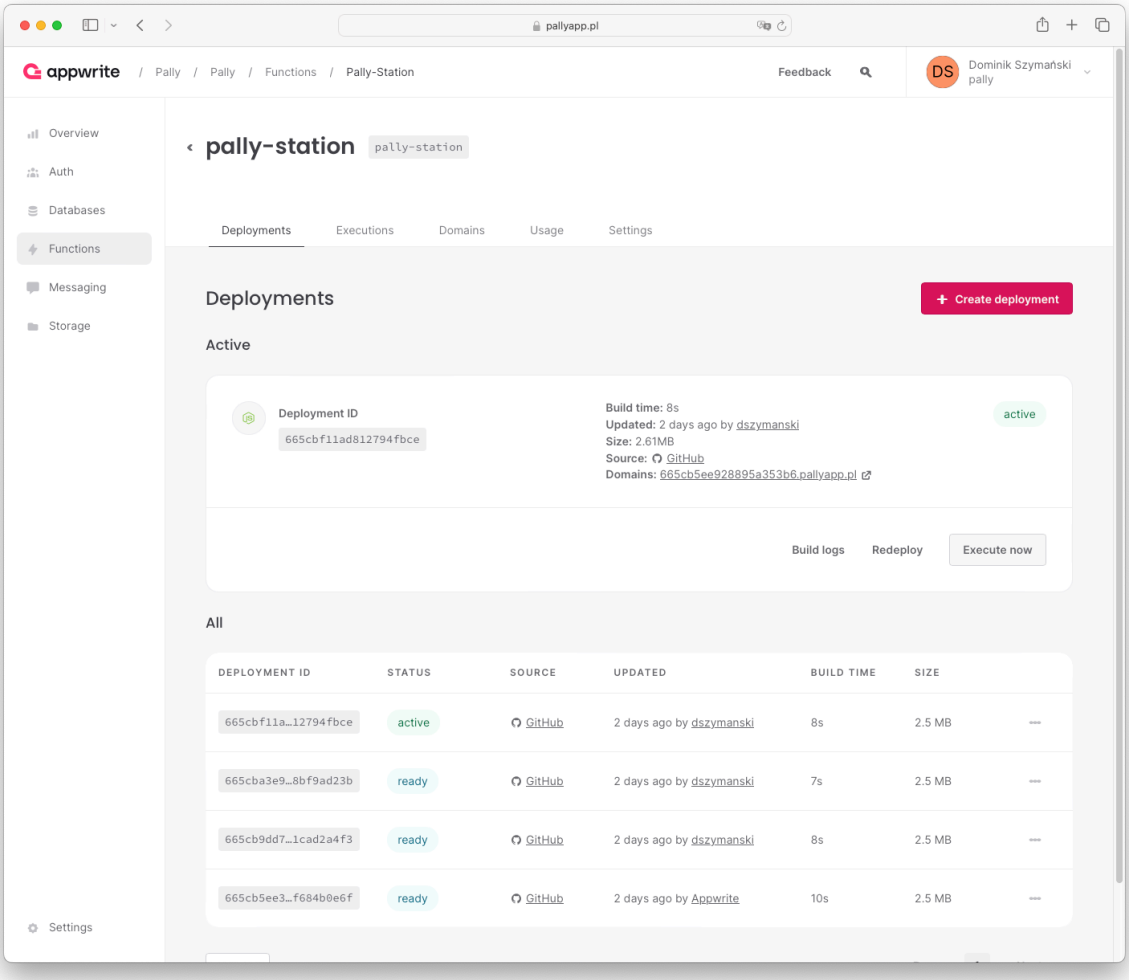


Rysunek 5.6 Podstrona Appwrite Functions z listą funkcji (źródło: *opracowanie własne*)

Każda z funkcji jest przechowywana w osobnym repozytorium Git. Dzięki temu każdy z członków zespołu mógł pracować w wyizolowanym środowisku nad swoją częścią kodu. Dodatkowo, taka organizacji kodu pozwala na używaniu różnych wersji kodu jednocześnie: wdrażanie zmian bez konieczności modyfikacji i wdrażania innych funkcji.

Każda metoda Appwrite składa się z ustandaryzowanej funkcji: zazwyczaj o nazwie „main”, która przyjmuje 4 argumenty: „request”, „response”, „log” i „error”. Niezależnie od wybranego środowiska i języka, te argumenty tych funkcji są takie same i posiadają bliźniaczą strukturę. Różnią się tylko użytym typem danych w implementacji (np. obiekty w JavaScript, klasy w PHP).

W backendzie stworzono 17 funkcji: 7 funkcji w środowisku Python i pozostałe 10 w środowisku NodeJS.



Rysunek 5.7 Strona funkcji „pally-station” napisanej w NodeJS (źródło: *opracowanie własne*)

Funkcja „pally-station” zaprezentowana na rysunku 5.7 to funkcja odpowiadająca za aktualizacje wpisów w bazie danych o nowe ceny (np. zaktualizowane podczas skanowania pylonu paliwowego).

```
import {Client, Databases, ID} from 'node-appwrite';
import {createHash} from 'crypto';

let md5 = (string) => {
    return createHash('md5').update(string).digest('hex');
}

export default async ({req, res, log, error}) => {
    const client = new Client()
        .setEndpoint('https://pallyapp.pl/v1')
        .setProject(process.env.APPWRITE_FUNCTION_PROJECT_ID)
        .setKey(process.env.APPWRITE_API_KEY);

    const databases = new Databases(client);

    if (req.method === 'POST') {
        let success = true;

        let addFuel = async (fuelStationId, fuelType, fuelPrice,) => {
            let doc = {
                price: fuelPrice,
                price_datetime: new Date(),
                source_name: 'User',
                fuel_station_id: fuelStationId,
                fuel_station_id_raw: fuelStationId,
                fuel_type: fuelType,
            }

            doc.snowflake = md5(
                ['user', fuelType, doc.price, doc.price_datetime.toString(), doc.fuel_station_id].join('
')
            );

            try {
                await databases.createDocument(
                    'fuel-price-detal',
                    'fuel-price',
                    ID.unique(),
                    doc,
                );
            } catch (e) {
                success = false;
                error(e)
            }
        }

        let body = JSON.parse(req.body)
        let fuels = body.fuel_prices;
        fuels.forEach((fuel) => {
            if (fuel.price > 0) {
                addFuel(body.fuel_station_id, fuel.type, fuel.price)
            }
        })

        return res.json({
            success: success,
        })
    }

    return res.json({
        success: false
    })
}
```

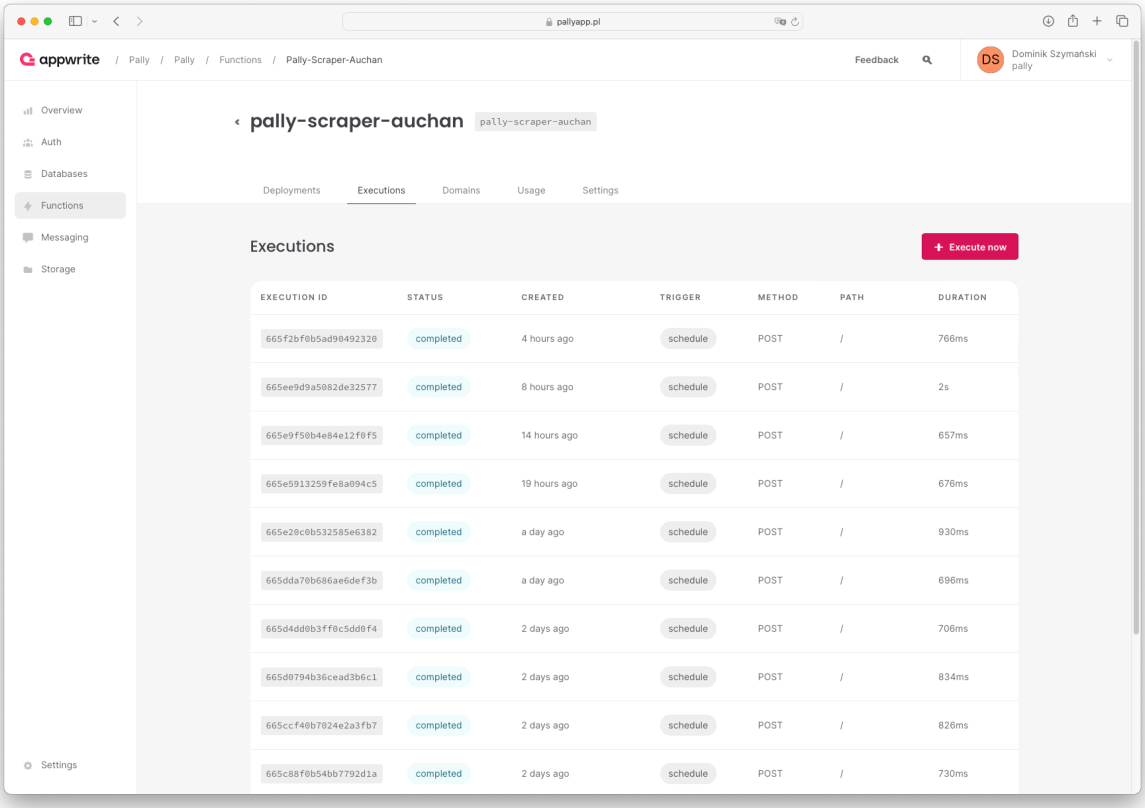
Rysunek 5.8 Kod źródłowy funkcji „pally-station” (źródło: *opracowanie własne*)

Na rysunku 5.8 zaprezentowano źródło funkcji i wykonywaną przez nią logikę biznesową. Klasa „Databases” dostarczana przez SDK Flutter komunikuje się z backendem i za pomocą funkcji „createDocument” tworzy nowy wpis, z przesłaną w zapytaniu HTTP, ceną dla danej stacji paliw.

Dzięki takiemu podejściu, w którym logika biznesowa jest rozbita na „atomowe” części, można łatwo rozwijać i wprowadzać zmiany bez obaw o wpływ na pozostały kod.

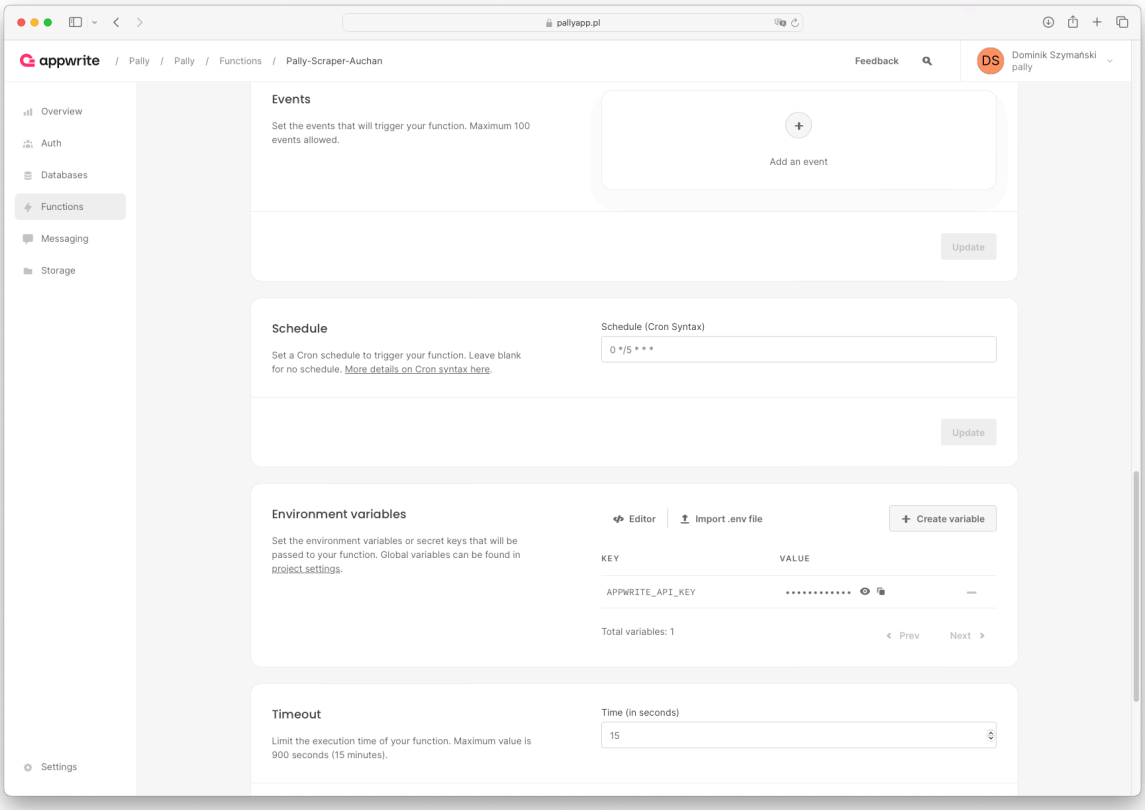
Można wyróżnić trzy kategorie funkcji ze względu na odpowiedzialność w domenie projektu

- Funkcje scrapujące (ang. *scrapping*): odpowiadają za pobieranie danych z pozostałych źródeł, ale nie zajmują się ich procesowaniem (czyli nie dokonują żadnej aktualizacji w bazie danych, jedynie co robią to zapisują pobrane pliki do aplikacji). Ekstrakcją danych z plików zajmuje się kolejna kategoria funkcji.
- Funkcje procesujące: zajmują się odczytaniem pobranych wcześniej plików i wyeksportowaniem z nich interesujących nas danych (np. danych o liście stacji paliw, ceny paliw w danym źródle).
- Funkcje pozostałe: funkcje pozostałe, które zajmują się np. aktualizowaniem danych w bazie (np. funkcja „pally-station”).



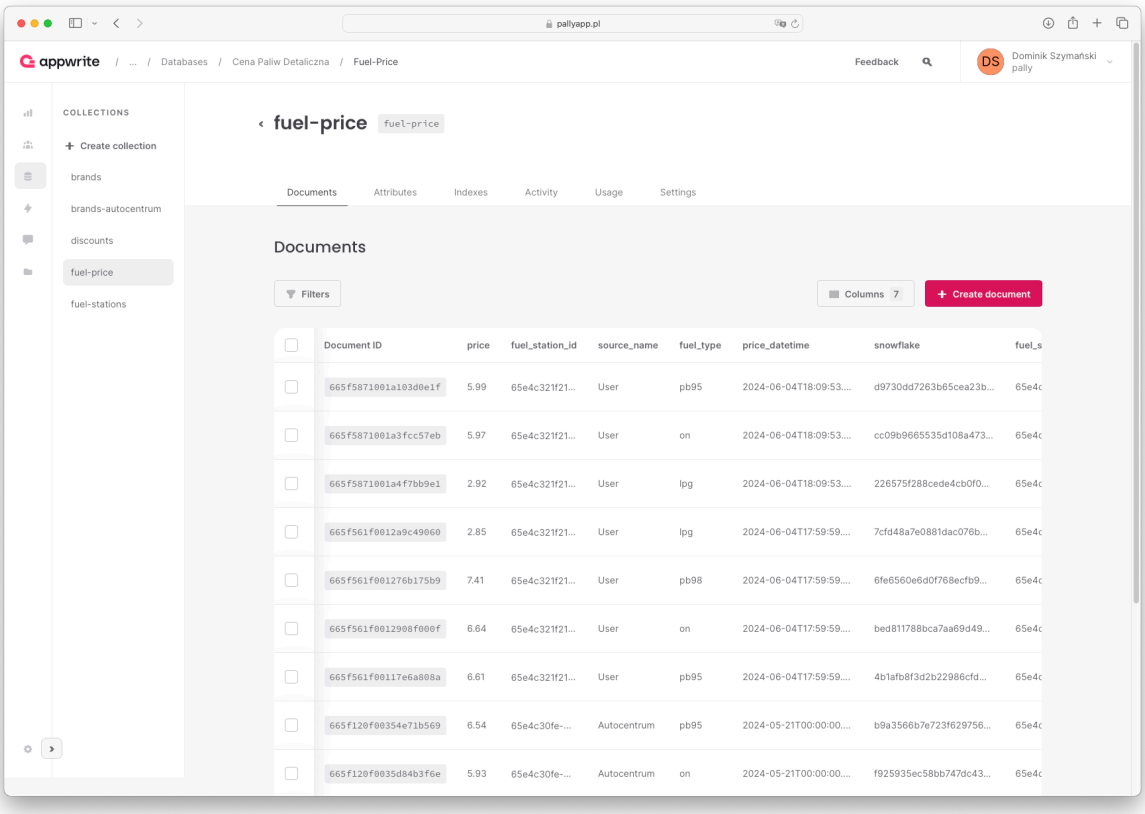
Rysunek 5.9 Lista wykonań funkcji „pally-scraper-auchan” (źródło: *opracowanie własne*)

Funkcje scrapujące oraz funkcje procesujące są uruchamiane cyklicznie na podstawie wyrażenia „cron”. Dzięki temu można zaplanować w jakiej kolejności i w jakich godzinach ma się wykonać proces scrappingu, a po nim wykonanie procesowania plików.



Rysunek 5.10 Zdefiniowane wyrażenie „cron” funkcji „pally-scraper-auchan” (źródło: *opracowanie własne*)

Dane, które są już przeprocesowane i wdrożone do bazy danych są umieszczane w kolekcji „fuel-prices”. Każdy rodzaj paliwa jest trzymany jako osobny wpis (nie każda stacja paliw ma dostępne wszystkie rodzaje paliwa na rynku).



Rysunek 5.11 Wpisy w kolekcji „fuel-price” (źródło: *opracowanie własne*)

Docelowo te dane trafiają do aplikacji mobilnej dzięki funkcji „pally-nearest-chepeast”, która na podstawie przesłanych współrzędnych geograficznych wykonuje następujące kroki:

- Filtruje stacje, które są w promieniu 10km od punktu podanych współrzędnych oraz mają dostępne ceny paliw co najmniej sprzed tygodnia
- Upewnia się, że stacje nie są wyłączone w obiegu (atrybut „listable” jest ustawiony na „true”).
- Odlicza rabat na dane paliwa na podstawie przesłanych danych do funkcji
- Na podstawie współrzędnych przesłanych w zapytaniu do funkcji (punkt A) oraz współrzędnych stacji paliw (punkt B) wyznacza trasę dzięki integracji z Mapbox. Dodaje długość wyznaczonej trasy w odpowiedzi do tej funkcji. Ta długość jest kluczowa do prognozowania kosztu dojazdu do danej stacji benzynowej – czyli przeanalizowania, gdzie jest najkorzystniej zatankować (biorąc pod uwagę aktualna pozycję użytkownika, spalanie jego auta oraz dostępne dla niego rabaty w danych sieciach stacji paliw).

```
fuelStations = await Promise.all(fuelStations.map(async (fS) => {
    let waypoints = [
        {
            coordinates: [userLongitude, userLatitude]
        },
        {
            coordinates: [parseFloat(fS['long']), parseFloat(fS['lat'])]
        }
    ]

    let distanceFromUserByRoutes = 0;

    try {
        let response = await directionsService.getDirections({waypoints: waypoints}).send()
        if (response.body?.routes?.length > 0) {
            distanceFromUserByRoutes = response.body.routes[0].distance;
        }
    } catch (err) {
        log(err);
        error(err)
    }

    return Object.assign({}, fS, {
        distance_from_user_by_routes: distanceFromUserByRoutes
    })
}))
```

Rysunek 5.12 Fragment funkcji „pally-nearest-cheapest” odpowiadający za obliczanie długości trasy (źródło: *opracowanie własne*)

Ekstrakcja cen z serwisu autocentrum.pl, ze względu na swój poziom rozbudowania (dane liczące w tysiącach plików) wymagała stworzenia dodatkowej funkcji „pally-extract-autocentrum-fuel-price” przeznaczonej tylko do ekstrakcji cen paliw. Dzięki takiemu rozwiązaniu udało się skonstruować funkcję tak, że jedna metoda wywoływała tą samą metodę, tylko z innymi parametrami, rekurencyjnie.

```
    if (req.path == '/fetch-from-file') {
        const databases = new Databases(client);

        let fileId = req.body.file_id
        log(`Fetching ${fileId} from Storage`);
        var enc = new TextDecoder("utf-8");

        let fileContentBuffer = await storage.getFileView(
            'pally-scraper-autocentrum-detal',
            fileId,
        )
        let fileContent = enc.decode(fileContentBuffer);
        let parsed = parse(fileContent);
        let prices = {
            pb95: [],
            pb98: [],
            on: [],
            gas: []
        }
        let findFuelStationByInternalId = async (internalStationId) => {
            log(`Fetching info from DB about fuel station autocentrum source_id: ${internalStationId}`)
            let documents = await databases.listDocuments('fuel-price-detal', 'fuel-stations', [
                Query.equal('source_id', internalStationId.toString()),
                Query.equal('source_name', 'autocentrum'),
                Query.limit(1)
            ]);
            if (documents.length === 0) {
                error(`Cannot find fuel station ${internalStationId}`)
                return null
            } else {
                return documents.documents[0];
            }
        }
        let addFuelPriceToDatabase = async (fuelType, fuelPrice, fuelDate, city, street,
internalStationId) => {
            log(fuelType, fuelPrice, fuelDate, internalStationId)
            let snowflake = md5(
                ['autocentrum', fuelType, fuelPrice, fuelDate, internalStationId].join(' ')
            )
            let databaseFuelStation = await findFuelStationByInternalId(internalStationId);
            if (!databaseFuelStation) {
                log(`Skipping: cannot find ${internalStationId} in the db`)
                return;
            }
            let updateFields = {};
            if (!databaseFuelStation.address_street && street.length > 0) {
                updateFields.address_street = street;
            }
            if (!databaseFuelStation.address_city && city.length > 0) {
                updateFields.address_city = city;
            }
            updateStation(updateFields, databaseFuelStation.$id);
            let doc = {
                price: fuelPrice,
                price_datetime: fuelDate,
                source_name: 'Autocentrum',
                fuel_station_id: databaseFuelStation.$id,
                fuel_station_id_raw: databaseFuelStation.$id,
                fuel_type: fuelType,
                snowflake
            }

            try {
                await databases.createDocument(
                    'fuel-price-detal',
                    'fuel-price',
                    ID.unique(),
                    doc
                )
            } catch (err) {
                if (err.response.type === 'document_already_exists') {
                    log('Skipping, fuel price already exists');
                } else {
                    error(err)
                }
            }
        }
```

Rysunek 5.13 Fragment funkcji „pally-extract-autocentrum-fuel-price” odpowiadający za aktualizację cen dla serwisu autocentrum.pl (źródło: *opracowanie własne*)

Backend również jest odpowiedzialny za odczyt ceny paliwa z pylonu paliwowego dzięki funkcji „pally-openai-fuel-ocr”. Funkcja wykonywała zapytanie do API OpenAI, do modelu GPT 4.0o z zapytaniem:

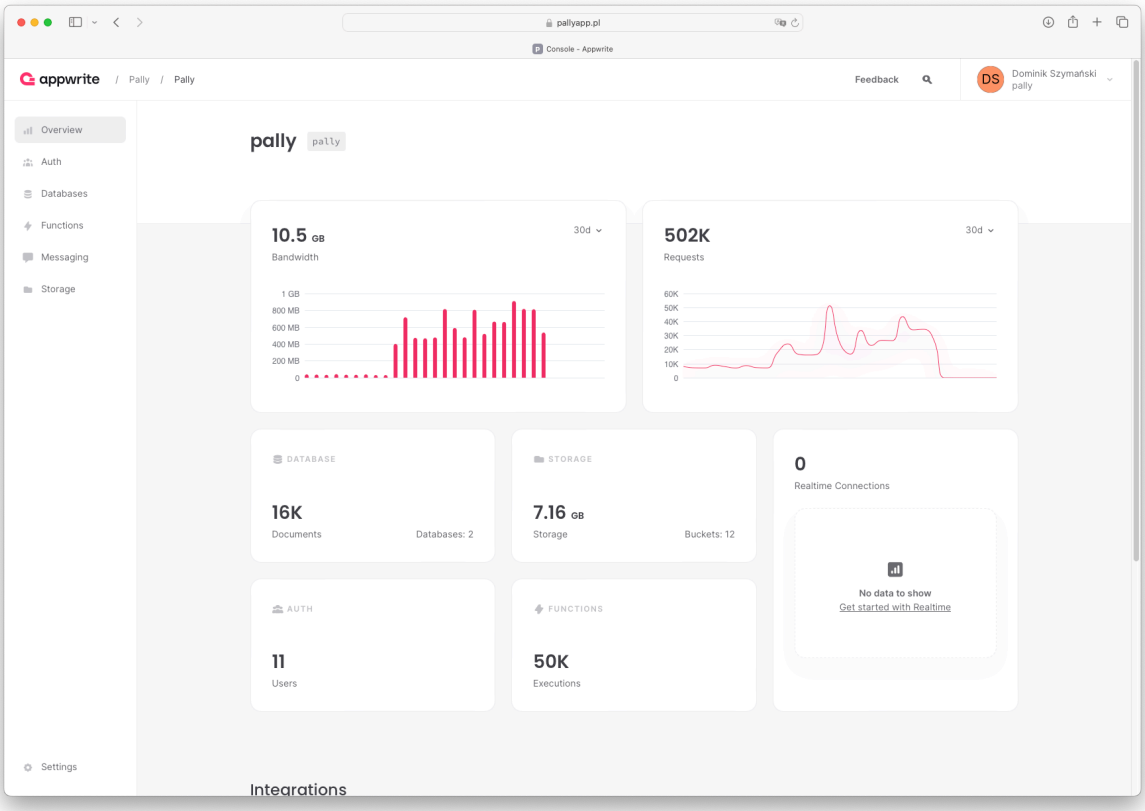
„Podaj ceny paliwa PB95, PB98, PB100, LPG i ON na zdjęciu?
Podaj informacje w formacie: RODZAJ_PALIWA=CENA, (rodzaje paliwa to PB95, PB98, LPG i ON) podajac w nowej linii inny rodzaj paliwa.
Jeżeli podanego paliwa nie ma na zdjęciu, po prostu go nie podawaj w odpowiedzi.
Pamiętaj, żeby nie podawać żadnego dodatkowego komentarza do odpowiedzi. Nie używaj cen paliw premium - jeżeli są dwa paliwa tego samego rodzaju, użyj tańszego.
Jeżeli nie ma wymienionych rodzajów paliwa na pylonie, przyjmij że ich cena to 0.00.”

Oprócz tego pytania do ChatGPT załączano przesłane przez użytkownika zdjęcie (dokładniej link do obrazu). ChatGPT zgodnie z podanym zapytaniem odpowiadał w wymaganym przez aplikację formacie. Niekiedy model pomijał paliwa, których nie było widocznych na zdjęciu, ale kod dodawał wartość domyślną (0,00) dla wszystkich rodzajów paliwa.

```
import OpenAI from "openai";

export default async ({ req, res, log, error }) => {
  const openai = new OpenAI();

  async function getFuelPricesAsString(url) {
    let prompt = `
Podaj ceny paliwa PB95, PB98, PB100, LPG i ON na zdjęciu?
Podaj informacje w formacie: RODZAJ_PALIWA=CENA, (rodzaje paliwa to PB95, PB98, LPG i ON) podajac w nowej
linii inny rodzaj paliwa.
Jeżeli podanego paliwa nie ma na zdjęciu, po prostu go nie podawaj w odpowiedzi.
Pamiętaj, żeby nie podawać żadnego dodatkowego komentarza do odpowiedzi. Nie używaj cen paliw premium -
jeżeli są dwa paliwa tego samego rodzaju, użyj tańszego.
Jeżeli nie ma wymienionych rodzajów paliwa na pylonie, przyjmij że ich cena to 0.00.
`
    const response = await openai.chat.completions.create({
      model: "gpt-4o",
      messages: [
        {
          role: "user",
          content: [
            { type: "text", text: prompt },
            {
              type: "image_url",
              image_url: {
                url,
                detail: 'low'
              },
            },
          ],
        },
      ],
      max_tokens: 1000
    });
    return response.choices[0].message.content
  }

  if (req.method !== 'POST') {
    return res.json({
      error: 'Only POST allowed'
    })
  }

  let body
  try {
    body = JSON.parse(req.body)
  }catch (e) {
    return res.json({
      error: 'Invalid json'
    })
  }
  let fileId = body.file_id;
  if(!fileId) {
    return res.json({
      error: 'Missing key ("file_id")'
    })
  }
  let url = `https://pallyapp.pl/v1/storage/buckets/65e72fa16e3718a50efd/files/${fileId}/view?
project=pally`
  log(`Trying to fetch ${url}`)

  let fuelsFromOpenAi = await getFuelPricesAsString(url);
  log(`Fuels from OpenAI: ${fuelsFromOpenAi}`)
  let fuels = (fuelsFromOpenAi).split('\n')
  fuels = Object.fromEntries(new Map(fuels.map(f => f.split('='))))

  fuels = Object.assign({
    PB95: '0.00',
    PB98: '0.00',
    ON: '0.00',
    LPG: '0.00'
  }, fuels)

  return res.json(fuels)
};
```

Rysunek 5.14 Fragment funkcji „pally-openai-fuel-ocr” do odczytywania cen paliw (źródło: *opracowanie własne*)

Konsola przeglądarkowa Appwrite zaprezentowana na wcześniejszych rysunkach jest bardzo rozbudowana i pozwala na bardzo łatwy sposób wdrażania zmian w bazie danych, zarządzanie użytkownikami w systemie i szybki podgląd statystyk naszej aplikacji.



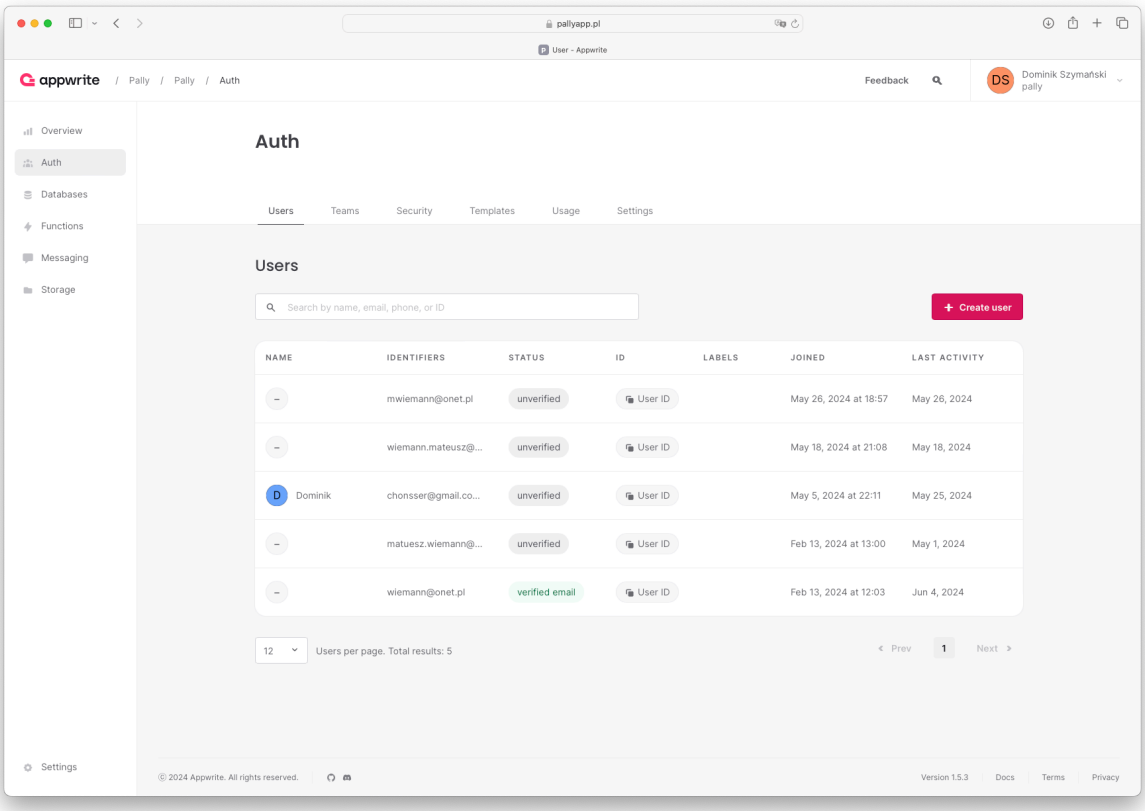
Rysunek 5.15 Konsola Appwrite projektu (źródło: *opracowanie własne*)

Na rysunku 5.15 widać zużycie sieci w związku z zapytaniami do API projektu (ang. *bandwidth*) oraz ile było zapytań do API. Dodatkowo mamy kilka statystyk, m.in.

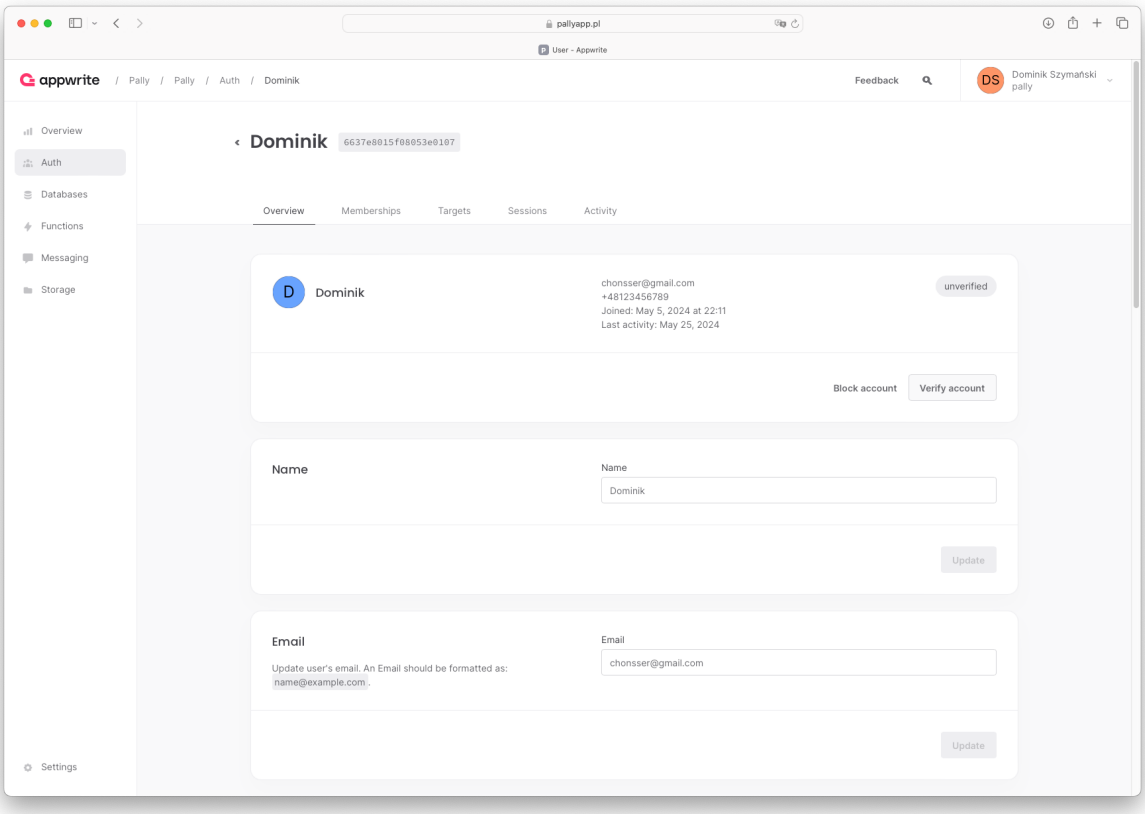
- ilu użytkowników znajduje się na dany moment w bazie danych
- ile było wykonań Appwrite Functions
- ile jest zajętej przestrzeni dyskowej w Appwrite Storage
- ile jest rekordów w Appwrite Databases

oraz ile jest aktualnie aktywnych połączeń na żywo (te akurat nie były użyte w projekcie).

W zakładce „Auth” mamy dostępne podstawowe funkcjonalności potrzebne do zarządzania użytkownikami: ich listę, status, możliwość tworzenia nowych użytkowników oraz modyfikację istniejących.



Rysunek 5.16 Panel administracyjny do zarządzania użytkownikami (źródło: *opracowanie własne*)



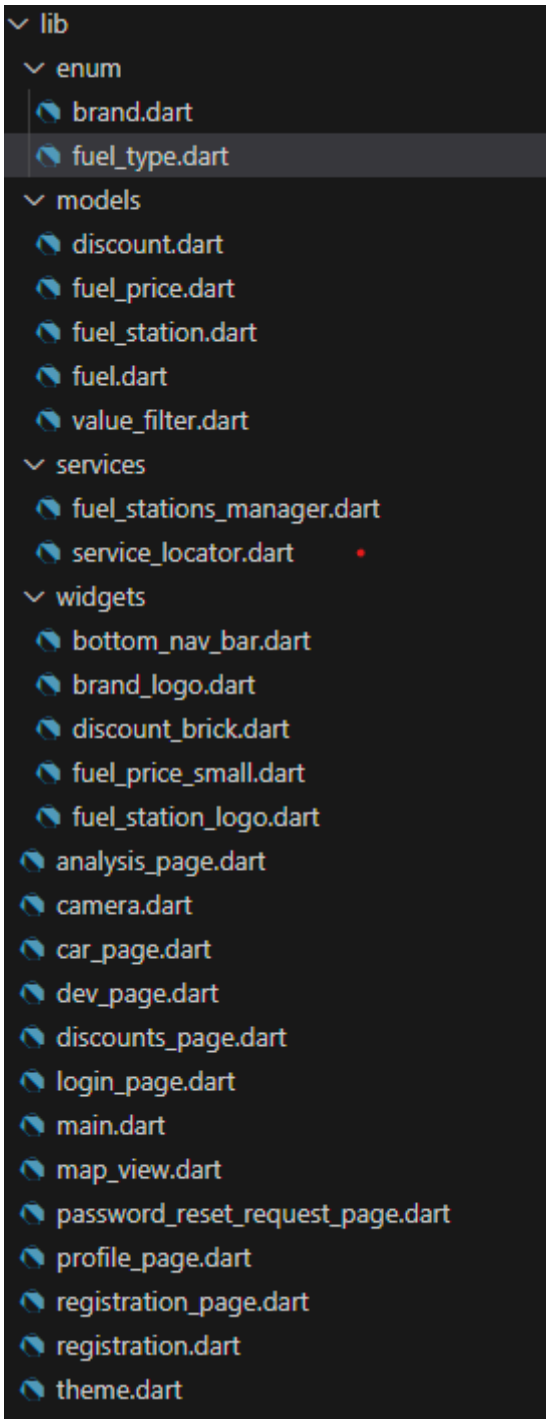
Rysunek 5.17 Szczegółowy widok pojedynczego użytkownika (źródło: *opracowanie własne*)

W backendzie oprócz wymienionych wcześniej funkcji mamy również dostępne:

- pally-discounts: odpowiedzialna za pobranie dostępnych rabatów na rynku (użyte na ekranie „Rabaty” w aplikacji mobilnej)
- extract-data-from-orlen-lpg: odpowiedzialna za odczytanie hurtowych cen gazu z pobranych plików
- extract-gas-station-from-database: odpowiedzialna za pobieranie listy stacji z bazy danych na podstawie przesłanych do backendu współrzędnych geograficznych
- pally-extract-auchan: odpowiedzialna za aktualizacje cen paliw na stacjach Auchan
- pally-extract-autocentrum-detal: odpowiedzialna za pobieranie listy stacji paliw z serwisu Autocentrum.pl
- pally-scraper-autocentrum-detal: odpowiedzialna za pobieranie stron z danymi z serwisu Autocenturm.pl
- extract-data-from-epetrol: odpowiedzialna za pobieranie danych z serwisu epetrol.pl
- pally-scraper-gielda: odpowiedzialna za pobieranie danych o hurtowych cenach na giełdzie w serwisie bankier.pl
- pally-scraper-orlen: odpowiedzialna za pobieranie hurtowych cen paliw ze strony koncernu Orlen

5.3 Frontend

Frontend aplikacji został stworzony w środowisku Flutter w celu osiągnięcia optymalnych rozwiązań zarówno pod kątem przejrzystości interfejsu użytkownika, jak i szybkości działania. W aplikacji zostało zastosowane wiele rozwiązań w celu optymalizacji użycia backendu, dzięki czemu w projekcie wiele operacji jest wykonywanych po stronie frontendu takich jak dynamiczne wczytywanie stacji benzynowych.



Rysunek 5.18 Struktura projektu Flutter. (źródło: *opracowanie własne*)

Struktura aplikacji w Flutter

Struktura projektu jest podzielona na kilka kluczowych katalogów i plików, które grupują poszczególne aspekty aplikacji:

- **lib**: Główny katalog zawierający kod źródłowy aplikacji.
    - **enum**: Katalog zawierający definicje typów wyliczeniowych.
    - **models**: Katalog definicji modeli danych, które przechowują struktury danych używane w aplikacji.
    - **services**: Katalog zawierający logikę biznesową i warstwę komunikacji z zewnętrznymi usługami.
    - **widgets**: Katalog zawierający widżety, które budują interfejs użytkownika.
    - **main.dart:** Główny plik uruchamiający aplikację, który inicjuje aplikację Flutter.

Podział kodu między poszczególne pliki:

„fuel_type.dart" – Zawiera rodzaje paliw w formie listy wyliczeniowej (enum), co pozwala na jednolite i bezpieczne odwoływanie się do typów paliw w całej aplikacji.

„fuel.dart" – Zawiera ścieżki do zdjęć poszczególnych rodzajów paliw, co ułatwia ich dynamiczne wyświetlanie w interfejsie użytkownika w zależności od kontekstu (np. na stronie szczegółów stacji paliw).

„fuel_price.dart" – Zawiera definicję klasy „FuelPrice", która służy do przechowywania informacji o cenach paliw, w tym typu paliwa oraz aktualnej ceny.

„fuel_station.dart" – Zawiera definicję klasy „FuelStation", która przechowuje dane dotyczące stacji paliw, takie jak lokalizacja, nazwa, dostępne paliwa, co umożliwia ich wyświetlanie na mapie oraz w innych częściach aplikacji.

„fuel_stations_manager.dart" – Zawiera klasę „FuelStationManager", która zarządza powiązaniami między identyfikatorami adnotacji na mapie a konkretnymi stacjami paliw, ułatwiając zarządzanie interakcjami użytkownika ze stacjami na mapie.

„service_locator.dart" – Zawiera konfigurację dla mechanizmu wstrzykiwania zależności „GetIt", który przechowuje globalne instancje takie jak „Client" z Appwrite i „FuelStationManager", zapewniając łatwy dostęp do nich w różnych częściach aplikacji.

„bottom_nav_bar.dart" – Zawiera definicję komponentu „BottomNavBar", który implementuje pasek nawigacyjny wyświetlany na dole ekranu w głównych widokach aplikacji, umożliwiając łatwą nawigację między kluczowymi sekcjami aplikacji.

„fuel_price_small.dart" – Zawiera komponent „FuelPriceSmall", który jest „StatelessWidget" w aplikacji Flutter. Ten komponent wyświetla mały widok z ceną i logiem danego typu paliwa, służąc jako element UI prezentujący szczegółowe informacje.

„fuel_station_logo.dart” – Zawiera komponent FuelStationLogo, który wyświetla logo stacji paliw.

„analysis_page.dart” – Zawiera komponent „AnalysisPage”, który wykorzystuje bibliotekę „webview_flutter” do wyświetlania strony internetowej zawierającej wykresy danych.

„camera.dart” – Zawiera klasę „CameraPage”, która służy do inicjalizacji kamery wewnętrza aplikacji oraz do możliwości robienia i przysyłania zdjęć.

„car_page.dart” – Zawiera klasę „CarPage”, która prezentuje szczegóły samochodu użytkownika, umożliwiając edycję typu paliwa pojazdu.

„dev_page.dart” – Zawiera klasę „DevPage”, komponent ten służy jako strona deweloperska, oferująca szybki dostęp do funkcji aplikacji.

„discounts_page.dart” – Zawiera komponent z listą dostępnych rabatów, które wpływają na wyświetlana cenę paliwa

„login_page.dart” – Zawiera komponent logowania do aplikacji.

„map_view.dart” – Zawiera klasę „MapView” służąc do obsługi biblioteki „mapbox_maps_flutter” , która odpowiada za wyświetlanie mapy w aplikacji.

„password_reset_request_page.dart” – Zawiera klasę „PasswordResetRequestPage” służącą do przesyłania linku z resetem hasła na wskazany adres mail który znajduje się w bazie danych.

„profile_page.dart” – Zawiera klasę „ProfilePage” służącą jako panel użytkownika, umożliwiający wylogowanie się z aplikacji.

„registration_page.dart” – Zawiera komponent rejestracji do aplikacji.

„registration.dart” – Zawiera klase „Registration”, która służy do komunikacji z Appwrite w celu utworzenia konta.

„theme.dart” – Zawiera style globalne aplikacji.

Kluczowe komponenty

Aplikacja wykorzystuje modularną architekturę, dzięki której łatwiej jest ją utrzymywać i rozwijać poprzez wyraźne oddzielenie komponentów. Modele są używane do reprezentowania danych i logiki niezwiązanej bezpośrednio z interfejsem użytkownika, na przykład definiują strukturę danych stacji paliw i cen paliw. Widżety w Flutter odpowiadają za generowanie interfejsu użytkownika, zapewniając dynamiczne i responsywne środowisko dla użytkowników. Serwisy, zajmujące się operacjami w tle takimi jak łączność z backendem czy zarządzanie danymi, umożliwiają efektywne oddzielenie logiki aplikacji od interfejsu użytkownika co zwiększa jej skalowalność i elastyczność.

```
5   final getIt = GetIt.instance;
6
7   void setupLocator() {
8     final client = Client()
9         .setEndpoint('https://pallyapp.pl/v1')
0         .setProject('pally');
1
2     getIt.registerSingleton<Client>(client);
```

Rysunek 5.19 Fragment kodu serwisu pliku service-locator.dart. (źródło: *opracowanie własne*)

Przykład serwisu w aplikacji

Ten kod tworzy instancję „GetIt", która będzie używana do rejestrowania i odzyskiwania zależności w całej aplikacji. Pochodzi on z popularnej biblioteki „get_it" [6] stosowanej w aplikacjach Flutter do implementacji wzorca wstrzykiwania zależności (ang. *dependency injection*) [12]. „GetIt.instance" zwraca singleton instancji „GetIt", umożliwiając globalny dostęp do kontenera zależności. Singleton jest wzorcem projektowym, który zapewnia że klasa jest używana tylko jedne raz i zapewnia globalny punkt dostępu do niej.

Funkcja setupLocator() służy do konfiguracji i rejestrowania zależności w kontenerze GetIt. Następnie odbywa się konfiguracja endpointa API oraz identyfikatora projektu. Jest to konieczne, aby klient mógł komunikować się z serwerem Appwrite, który obsługuje operacje backendowe aplikacji. getIt.registerSingleton<Client>(client); rejestruje instancję klienta Appwrite w kontenerze GetIt jako singleton, co oznacza, że wszędzie w aplikacji, gdzie będzie potrzebny klient Appwrite, będzie używana ta sama instancja.

Komponenty

W aplikacji wykorzystano komponent z biblioteki „material.dart", która jest integralną częścią Fluttera. Te komponenty opierają się na klasie „Material", która jest odpowiedzialna za implementację „Material Design"[31] , systemu projektowania opracowanego przez Google. Komponent promuje spójność interfejsu użytkownika poprzez zdefiniowane wytyczne, które są łatwo adaptowalne i interpretowane przez system Android. Dzięki temu, używając komponentów z „material.dart", dzięki czemu można tworzyć spójny i responsywny wygląd, który zachowuje takie samo działanie na różnych urządzeniach i platformach.

W aplikacji, zarządzanie routingiem i nawigacją jest realizowane za pomocą klasy „Navigator" [32], która jest dostarczana przez Flutter. Klasa ta umożliwia łatwe przełączanie między ekranami oraz zarządzanie historią nawigacji, co pozwala użytkownikom na płynne przemieszczanie się po różnych częściach aplikacji.

Integracja z backendem

Aplikacja korzysta z SDK Appwrite, co umożliwia bezpośrednie połączenie z serwerem i realizację funkcji związanych z zarządzaniem danymi użytkowników. Wykorzystuje się tu obiekt Client z Appwrite do konfiguracji i utrzymania sesji użytkownika, a także do operacji takich jak tworzenie konta i logowanie.

```
void registerAccount(String userId, String email, String password) {
  Future result = account.create(
    userId: userId,
    email: email,
    password: password,
  );
```

Rysunek 5.20 Fragment kodu służący do utworzenia konta (źródło: *opracowanie własne*)

Rejestracja nowych użytkowników jest realizowana przez klasę „Registration", która przyjmuje obiekt Client i używa go do utworzenia nowego konta za pomocą metody „create" w klasie „Account". Proces ten obejmuje przekazanie danych użytkownika, takich jak identyfikator, e-mail i hasło.

```dart
class LoginPage extends StatefulWidget {
  final Client client = getIt<Client>();

  @override
  _LoginPageState createState() => _LoginPageState();
}
```

Rysunek 5.21 Fragment kodu odpowiadający za połączenie z Appwrite z login_page.dart (źródło: *opracowanie własne*)

```dart
Future result = account.createEmailSession(
  email: email,
  password: password,
);
```

Rysunek 5.22 Fragment kodu odpowiadający za logowanie (źródło: *opracowanie własne*)

Logowanie użytkowników odbywa się za pośrednictwem klasy LoginPage, która również korzysta z obiektu Client przekazanego przez GetIt do zarządzania sesją. Logowanie realizowane jest poprzez metodę createEmailSession, co pozwala użytkownikowi na dostęp do swojego konta za pomocą adresu e-mail i hasła.

Obsługa stanu

Stan w aplikacji jest zarządzany lokalnie w kontekście poszczególnych widoków za pomocą „StatefulWidget”. [39] Klasa ta to podstawowy komponent w frameworku Flutter, który pozwala tworzyć dynamiczne interfejsy użytkownika, reagujące na zmiany danych w czasie rzeczywistym. „StatefulWidget” jest zaprojektowany do zarządzania stanem, który może ulegać zmianom, umożliwiając tworzenie aplikacji interaktywnych i elastycznych.

```dart
setState(() {
  _isLoading = true;
});

Future result = account.createEmailSession(
  email: email,
  password: password,
);

result.then((response) {
  print(response);
  // Navigate to main.dart after successful
  Navigator.pushReplacement(
    context,
    MaterialPageRoute(
        builder: (context) => MapView()), /
  );
}).catchError((error) {
  print(error.response);
  setState(() {
    _isLoading = false;
  });
});
```

Rysunek 5.23 Fragment kodu zarządzania stanem (źródło: *opracowanie własne*)

Zarządzanie stanem UI

W aplikacji używane są zmienne stanu do kontrolowania reakcji interfejsu użytkownika na działania użytkownika oraz na procesy sieciowe. Przykładem jest zmienna _isLoading używana na stronie logowania, która informuje interfejs użytkownika, kiedy aktywować i dezaktywować wskaźnik ładowania, co jest szczególnie przydatne podczas oczekiwania na odpowiedź serwera po próbie logowania. Dzięki temu użytkownik wie, że jego żądanie jest przetwarzane i że musi poczekać na zakończenie procesu.

```
Padding(
  padding: const EdgeInsets.fromLTRB(0, 20, 0, 20),
  child: TextFormField(
    controller: _passwordController,
    decoration: InputDecoration(labelText: 'Hasło'),
    obscureText: true,
    validator: (value) {
      if (value == null || value.isEmpty) {
        return 'Proszę podać hasło';
      } else if (value.length < 8) {
        return 'Podano nieprawidłowe hasło';
      }
      return null;
    },
  ), // TextFormField
```

Rysunek 5.24 *Fragment kodu sprawdzający hasło (źródło: opracowanie własne*)

Walidacja danych

Proces walidacji danych jest bardzo ważny w kontekście logowania i rejestracji dla zachowania integralności danych oraz upewnienia się, że użytkownik podał wszystkie niezbędne informacje. W procesie tworzenia i użytkowania konta, używany jest komponent Form. Pozwala to na centralne zarządzanie stanem formularza i na jego walidację.

Walidatory przyłączone do pól formularza w Flutterze służą do sprawdzania poprawności danych wprowadzonych przez użytkownika przed ich wysłaniem do backendu. Dzięki temu mechanizmowi, aplikacja ogranicza niepotrzebną komunikację z serwerem, eliminując próby przesłania nieprawidłowych danych. Walidacja ta zapewnia, że tylko dane spełniające określone kryteria (takie jak poprawny format adresu e-mail czy minimalna długość hasła) zostaną przetworzone, co zwiększa efektywność aplikacji i poprawia bezpieczeństwo.

Opis implementacji i konfiguracji mapy w aplikacji

Integracja z Mapbox w aplikacji Flutter wykorzystuje bibliotekę „mapbox_maps_flutter” do tworzenia interaktywnej i funkcjonalnej mapy. Głównym elementem tej integracji jest komponent MapboxMap, który umożliwia personalizację mapy. Dodatkowo biblioteka „geolocator” dostarcza nam możliwość korzystania z lokalizacji użytkownika.

```
this
    .mapboxMap
    ?.annotations
    .createPointAnnotationManager()
    .then((PointAnnotationManager val) async {
  pointAnnotationManager = val;
  pointAnnotationManager
      ?.addOnPointAnnotationClickListener(AnnotationClickListener());
});
```

Rysunek 5.25 Fragment kodu tworzenia instancji „PointAnnotationManager” (źródło: *opracowanie własne*)

Personalizacja mapy

Mapa w aplikacji jest dostosowywana za pomocą ustawień stylów oraz przez dodawanie markerów, które reprezentują lokalizacje stacji paliw. „PointAnnotation” jest używany do oznaczania konkretnych punktów na mapie z dodatkowymi informacjami, takimi jak lokalizacja czy wyświetlane obrazy. Funkcja umożliwia łatwe dodawanie interaktywnych elementów do mapy, które reagują na kliknięcia użytkownika, wywołując szczegółowe informacje o poszczególnych lokalizacjach.

```
  try {
    print('Fetch current location');
    geo.Position pos = await geo.Geolocator.getCurrentPosition(
        desiredAccuracy: geo.LocationAccuracy.lowest,
        timeLimit: Duration(milliseconds: 150));
    return Position(pos.longitude, pos.latitude);
  } catch (err) {
    geo.Position? lastKnownPosition =
        await geo.Geolocator.getLastKnownPosition();
    if (lastKnownPosition != null) {
      print('Return last known position');
      return Position(
          lastKnownPosition.longitude, lastKnownPosition.latitude);
    } else {
      print('Return default position');
      return defaultPosition;
    }
  }
}
```

Rysunek 5.26 Fragment kodu do pobierania lokalizacji za pomocą blilioteki geolocator (źródło: *opracowanie własne*)

```dart
void _zoomToCurrentLocation() async {
  Position position = await _determinePosition();
  print('Zoom to current position: ${position.lat} ${position.lng}');

  mapboxMap?.flyTo(
      CameraOptions(
        center: Point(coordinates: position).toJson(),
        bearing: 0,
        zoom: 13.0,
      ), // CameraOptions
      MapAnimationOptions(duration: 500, startDelay: 0));

  _determinePositionAndLoadFuelStations();
}
```

Rysunek 5.27 *Fragment kodu przybliżenia do aktualnej lokalizacji (źródło: opracowanie własne)*

Wykorzystanie funkcji geolokalizacyjnych i interaktywności Mapbox

Aplikacja wykorzystuje usługi geolokalizacji poprzez integrację z biblioteką „geolocator", co umożliwia śledzenie pozycji użytkownika w czasie rzeczywistym, centrowanie mapy na bieżącej lokalizacji użytkownika oraz aktualizowanie informacji o pobliskich stacjach paliw, wymagając przy tym dostępu do lokalizacji użytkownika dla poprawnego działania. Mapa jest również interaktywna, reaguje na działania takie jak przesuwanie, zoomowanie i klikanie w punkty, a metody takie jak „flyTo" pozwalają na animowane przemieszczanie kamery do nowej lokalizacji.

```dart
class _AnalysisPageState extends State<AnalysisPage> {
  var url = 'https://pallyapp.pl/v1/storage/buckets/65c92fff29bfb0f4a1dd/files/chart-cena-gielda/view?project=pally&t=' +
  DateTime.now().millisecondsSinceEpoch.toString();
  bool _isLoading = true;

  @override
  Widget build(BuildContext context) {
    WebViewWidget webView = WebViewWidget(controller: WebViewController()
      ..setJavaScriptMode(JavaScriptMode.unrestricted)
      ..setBackgroundColor(const Color(0x00000000))
      ..setNavigationDelegate(
        NavigationDelegate(
          onProgress: (int progress) {
            // Update loading bar.
          },
          onPageStarted: (String url) {},
          onPageFinished: (String url) {},
          onWebResourceError: (WebResourceError error) {},
          onNavigationRequest: (NavigationRequest request) {
            return NavigationDecision.navigate;
          },
        ), // NavigationDelegate
      )
      ..loadRequest(Uri.parse(url))); // WebViewWidget
    return Scaffold(
      appBar: AppBar(title: const Text('Analiza')),
      body: webView
    ); // Scaffold
  }
```

Rysunek 5.28 Fragment kodu implementacji strony jako webView (źródło: *opracowanie własne*)

Integracja Chart.js z Flutter przez WebView

Integracja odbywa się poprzez umieszczenie wykresów w WebView, co pozwala na prezentację zaawansowanych wizualizacji danych bezpośrednio w aplikacji mobilnej. Dzięki temu można łatwo wyświetlać skompilowane wykresy bez konieczności implementowania natywnych komponentów graficznych w Flutterze.

```
const client = new Appwrite.Client();

client
    .setEndpoint('https://pallyapp.pl/v1')
    .setProject('pally');

const databases = new Appwrite.Databases(client);
const ctx = document.getElementById('myChart').getContext('2d');
let chart;
let currentOilType = 'Ropa_wti_';
// Pobierz referencje do elementów HTML
var ropaElement = document.getElementById('ropa');
var percentElement = document.getElementById('percent');
var priceElement = document.getElementById('price');
var arrowElement = document.getElementById('arrow');

function updateChart(days, oilType) {
    let query;
    if (oilType === 'Ropa_wti_') {
        query = Appwrite.Query.startsWith('sorce', oilType);
    } else {
        query = Appwrite.Query.startsWith('sorce', 'Ropa_1');
    }
    databases.listDocuments(
        '656c5f9a89e1fa18f815',
        '656c5fafb62f780baad3',
        [
            Appwrite.Query.orderDesc('sorce'),
            Appwrite.Query.limit(days),
            query,
        ]
    )
        .then(response => {
            if (chart) {
                chart.destroy();
            }
```

Rysunek 5.29 Fragment kodu przedstawiający pobranie danych z Appwrite (źródło: *opracowanie własne*)

```
            const data = response.documents;
            const prices = data.map(item => item[currentFuelType]);
            const minPrice = Math.floor(Math.min(...prices) / 10) * 10 + -10;
            const maxPrice = Math.ceil(Math.max(...prices) / 10) * 10 + 10;

            const configuration = {···

            // Oblicz wartości min, max i procentową zmianę dla wybranego zakresu czasu
            var lastPrice = prices[0];
            var firstPrice = prices[prices.length - 1];
            var percentChange = ((lastPrice - firstPrice) / firstPrice) * 100;
            configuration.data.labels = data.map(item => new Date(item.$createdAt));
            configuration.data.datasets[0].data = data.map(item => item[currentFuelType]);
            chart2 = new Chart(ctx2, configuration);
```

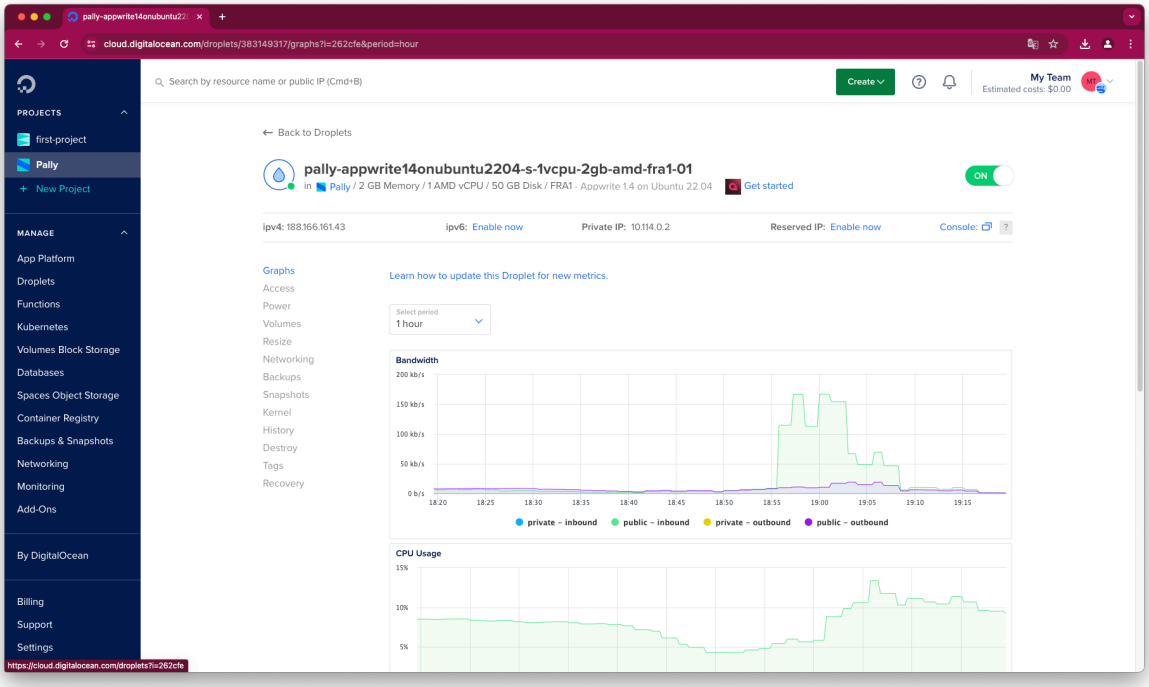
Rysunek 5.30 Fragment kodu przedstawiający konfiguracje wykresu (źródło: *opracowanie własne*)

Sposób przetwarzania i przekazywania danych do wykresów:

Dane pobierane z Appwrite za pomocą odpowiednich zapytań do bazy są przetwarzane oraz agregowane w JavaScript, co umożliwia ich filtrowanie według różnych parametrów. Dzięki temu użytkownicy mogą dynamicznie wyświetlać dane i wybierać parametry, które odpowiadają ich potrzebom. Wykresy generowane za pomocą Chart.js pozwalają na tworzenie interaktywnych wykresów liniowych, które są przejrzyste. Użytkownicy mają możliwość aktualizowania wykresów poprzez dostosowanie takich parametrów jak okres czasowy, rodzaj paliwa lub lokalizacja. Dzięki wykorzystaniu JavaScript, wykresy są wyświetlane dynamicznie, co pozwala na ich przeglądanie bez konieczności odświeżania strony lub zmiany ekranów, zapewniając płynną i efektywną interakcję.

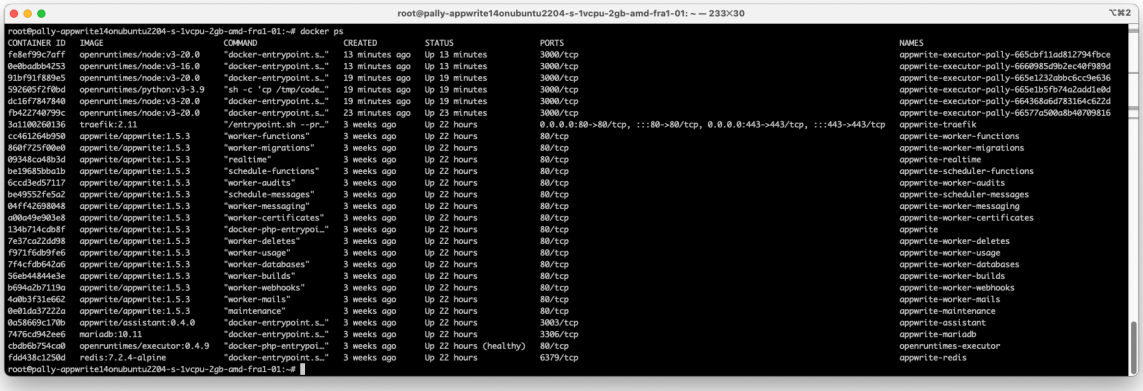
5.4 Architektura / platforma

W projekcie zdecydowano się na wykorzystanie systemu operacyjnego Ubuntu 22.04 uruchomionego na serwerze w serwisie Digitalocean. Było to argumentowane doświadczeniem zespołu w tym systemie operacyjnym. Wybór hostingu był podyktowany korzystaną ofertą w postaci kredytu w wysokości 200 amerykańskich dolarów do wykorzystania w projekcie.



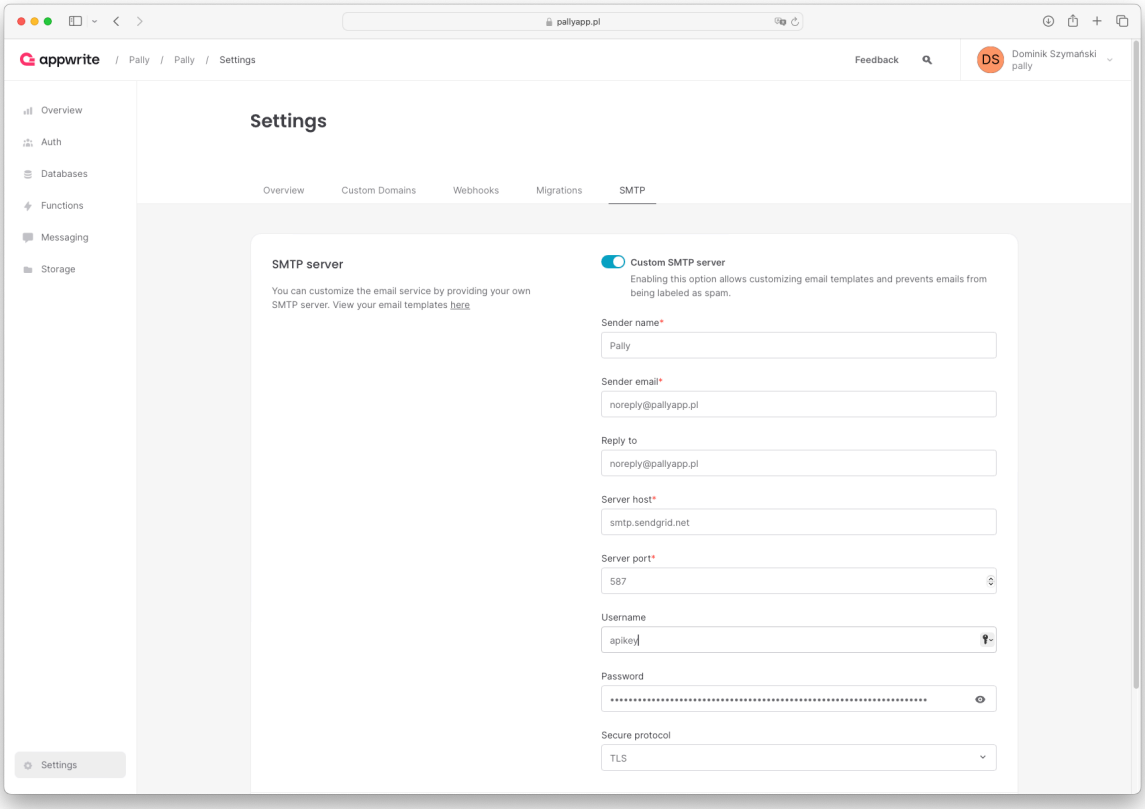
Rysunek 5.31 Panel DigitalOcean prezentujący serwer projektu (źródło: *opracowanie własne*)

Appwrite, ze względu na swoją specyfikę, wymaga uruchomienia na Dockerze. Każde środowisko uruchomieniowe (NodeJS, Python) posiada swój własny kontener, w którym jest uruchamiany kod projektu.



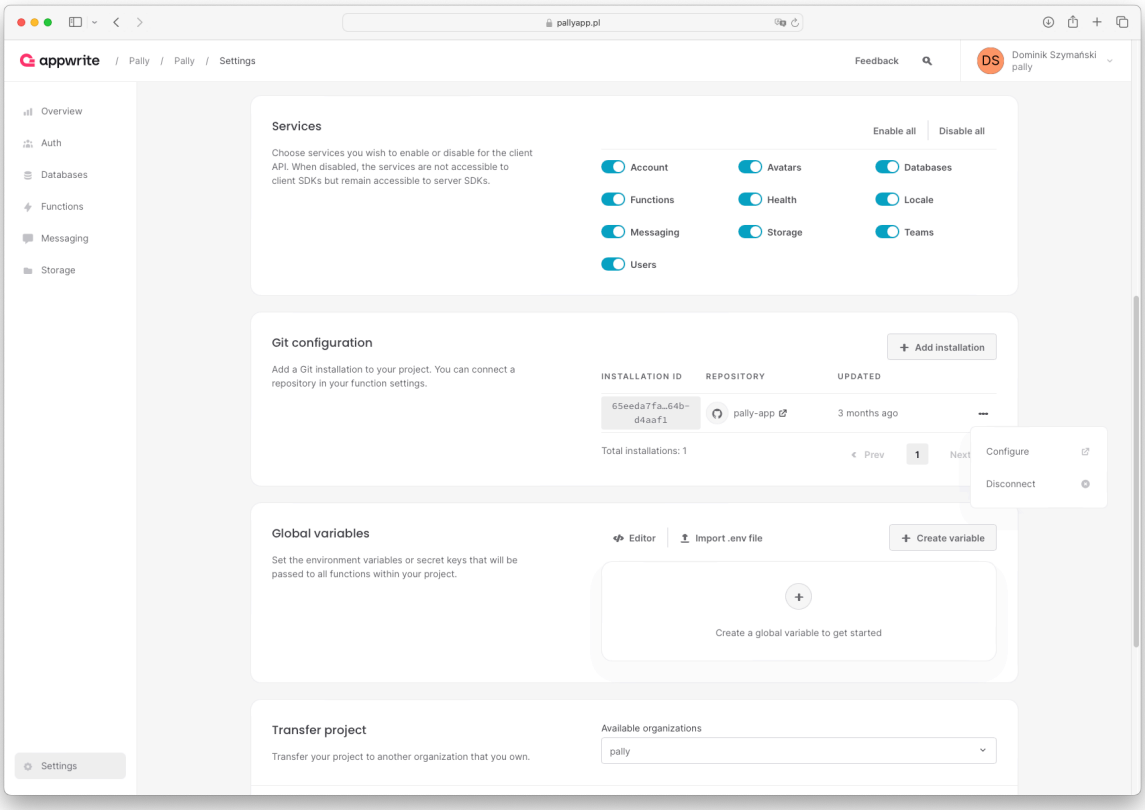
Rysunek 5.32 Uruchomione kontenery Docker na serwerze projektu (źródło: *opracowanie własne*)

Do wysyłki maili w projekcie wykorzystano SendGrid, który został skonfigurowany w platformie Appwrite za pomocą protokołu SMTP (za pomocą którego też dokonuje wysyłki wiadomości do użytkowników, np. do zresetowania hasła).



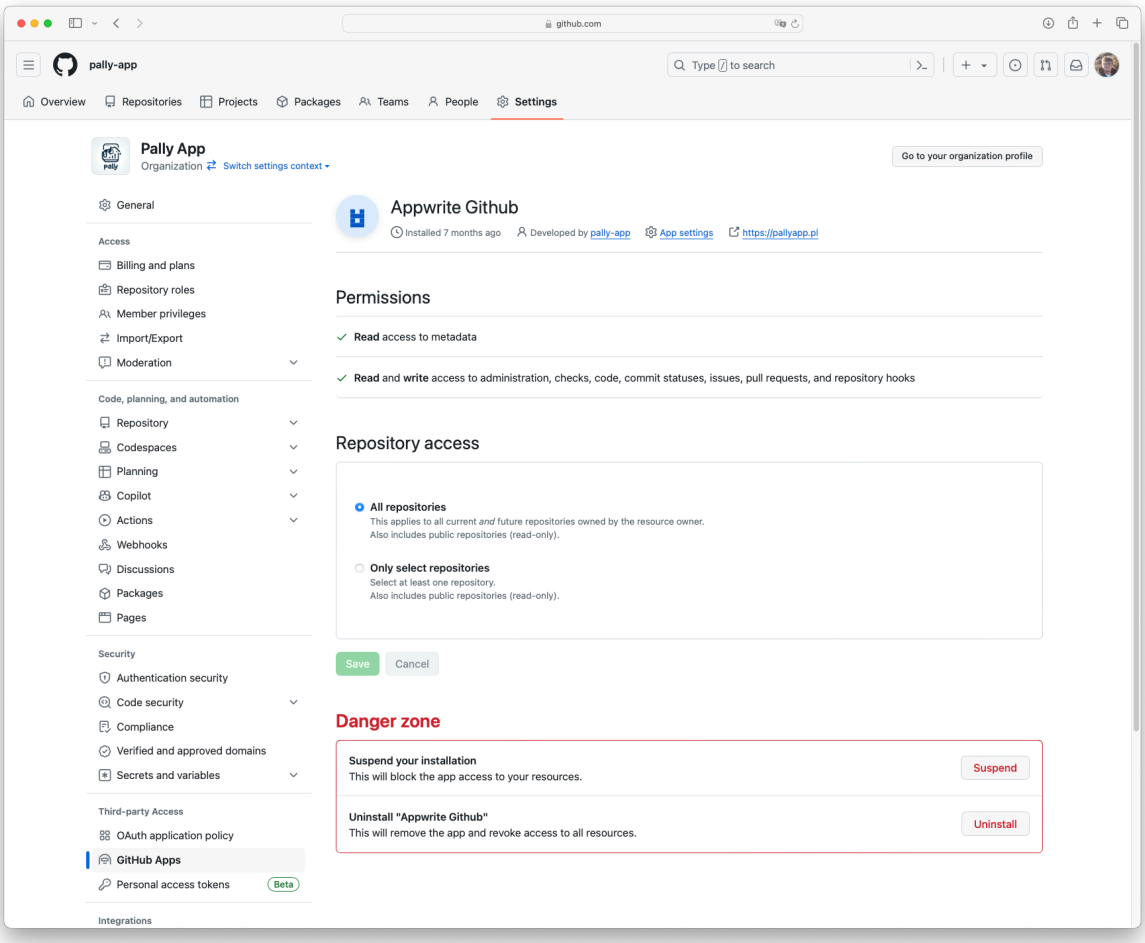
Rysunek 5.33 Konfiguracja wysyłki maili z aplikacji (źródło: *opracowanie własne*)

Do sprawnego wdrażania kodu na serwer w projekcie wykorzystano procedurę ciągłego wdrażania (*ang. Continuous Deployment*, w skrócie *CD*), czyli automatyzacje procesu wdrażania oprogramowania na serwer. Całość bazuje na integracji Github z platformą Appwrite.

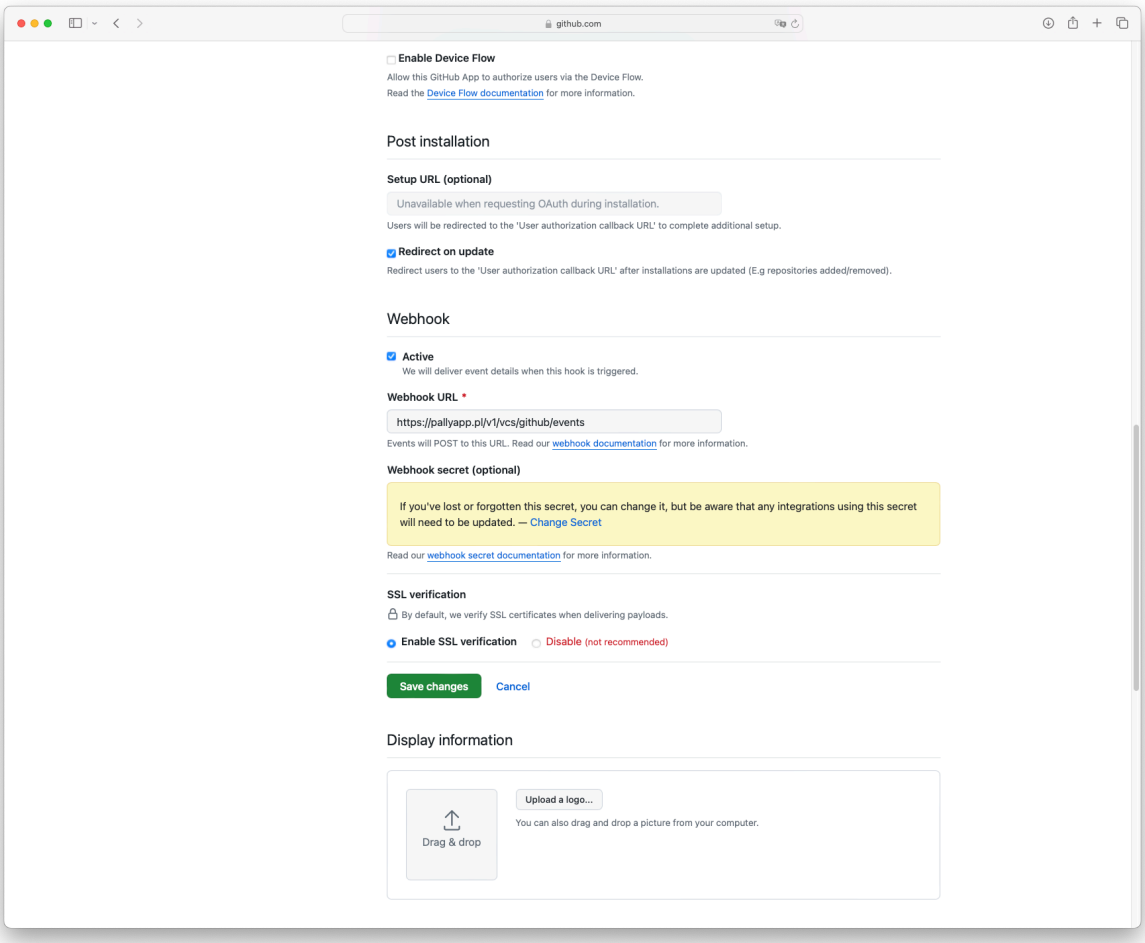


Rysunek 5.34 Skonfigurowana integracja Github z Appwrite (źródło: *opracowanie własne*)

Aby integracja działała poprawnie, wymagane było stworzenie wpisu aplikacji na platformie Github, a następnie skonfigurowanie jej o adresy wskazujące na domenę projektu.

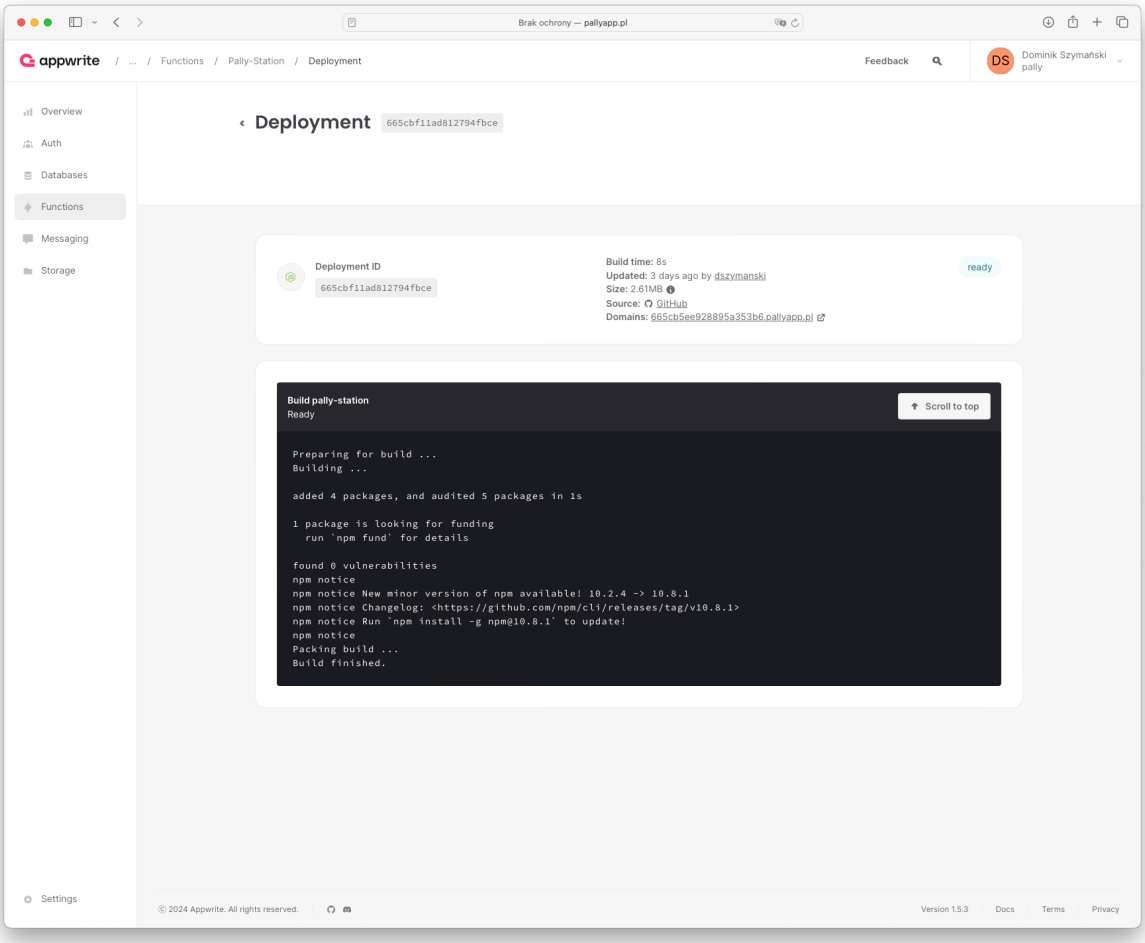


Rysunek 5.35 Aplikacja „Appwrite Github” stworzona na potrzeby procesu CD (źródło: *opracowanie własne*)



Rysunek 5.36 Konfiguracja adresów do webhooków integracji (źródło: *opracowanie własne*)

Po poprawnej konfiguracji, Appwrite przy jakiejkolwiek nowej zmianie przesłanej do repozytorium wykona proces wdrożenia zmiany na serwer.



Rysunek 5.37 Logi procesu wdrażania zmian do funkcji „pally-station” na serwer (źródło: *opracowanie własne*)

5.5 Testowanie – testy automatyczne

Testy automatyczne są podstawą do zachowania wysokiej sprawności i niezawodności kodu. Dzięki testom automatycznym udało się zniwelować wiele problemów które nie zostały wykryte w testach manualnych. W pracy zostały wykorzystane dwa rodzaje testów automatycznych, testy jednostkowe (ang. *unit tests*) oraz testy widżetów (ang. *widget tests*). Poniżej znajduje się szczegółowy opis wraz z przykładem takiego testu przeprowadzonego w aplikacji Flutter.

```dart
import 'package:flutter_test/flutter_test.dart';
import 'package:pally/enum/fuel_type.dart';

Run | Debug
void main() {
  Run | Debug
  group('FuelTypeExtension', () {
    Run | Debug
    test('fuel_name returns correct name for each FuelType', () {
      expect(FuelType.pb95.fuel_name, 'Benzyna (E5/PB95)');
      expect(FuelType.pb98.fuel_name, 'Benzyna (E10/PB98)');
      expect(FuelType.on.fuel_name, 'Diesel');
      expect(FuelType.gas.fuel_name, 'Gaz');
    });

    Run | Debug
    test('fuel_name_short returns correct short name for each FuelType', () {
      expect(FuelType.pb95.fuel_name_short, 'E5/PB95');
      expect(FuelType.pb98.fuel_name_short, 'E10/PB98');
      expect(FuelType.on.fuel_name_short, 'ON');
      expect(FuelType.gas.fuel_name_short, 'Gaz');
    });
  });
```

Rysunek 5.38 Fragment kod fuel_type_test.dart (źródło: *opracowanie własne*)

Testy jednostkowe

Skupiają się one na weryfikacji poprawnego działania pojedynczej funkcji lub metody. Są szybkie do wykonania i koncertują się na małych częściach kodu. Przykładem jest test sprawdzający poprawność zwracania nazw dla poszczególnych rodzajów paliw oraz ich skróconych wersji.

Przykłady testów jednostkowych:

- **fromJson creates a Discount from JSON:** Test sprawdza, czy metoda „fromJson" prawidłowo tworzy obiekt „Discount" na podstawie dostarczonego JSON-a. Test obejmuje różne przypadki użycia, aby upewnić się, że wszystkie właściwości są poprawnie mapowane.
- **logo returns correct AssetImage for each Brand**: Test sprawdza, czy metoda „logo" dla każdej wartości enum „Brand" zwraca poprawny obraz.
- **fromJson creates a FuelStation from JSON**: Test sprawdza, czy metoda „fromJson" poprawnie tworzy obiekt „FuelStation" na podstawie danych JSON.

- **getAddressAsString returns correct address string**: Test sprawdza, czy metoda „getAddressAsString” zwraca poprawny adres w formie łańcucha znaków.
- **getPosition returns correct Position**: Test sprawdza, czy metoda „getPosition” zwraca poprawną pozycję na podstawie współrzędnych stacji paliw.

```
Run | Debug
testWidgets('RegistrationPage renders correctly',
    (WidgetTester tester) async {
  await tester.pumpWidget(MaterialApp(home: RegistrationPage()));

  expect(find.text('Rejestracja'), findsOneWidget);
  expect(find.text('Zarejestruj się'), findsOneWidget);
  expect(find.byType(TextFormField), findsNWidgets(2));
});

Run | Debug
testWidgets('Register button is disabled when form is invalid',
    (WidgetTester tester) async {
  await tester.pumpWidget(MaterialApp(home: RegistrationPage()));

  final button = find.widgetWithText(ElevatedButton, 'Zarejestruj się');
  expect(tester.widget<ElevatedButton>(button).onPressed, isNull);
});

Run | Debug
testWidgets('Register button is enabled when form is valid',
    (WidgetTester tester) async {
  await tester.pumpWidget(MaterialApp(home: RegistrationPage()));

  await tester.enterText(
      find.widgetWithText(TextFormField, 'Adres e-mail'), 'test@example.com');
  await tester.enterText(
      find.widgetWithText(TextFormField, 'Hasło'), 'password123');

  await tester.pump();

  final button = find.widgetWithText(ElevatedButton, 'Zarejestruj się');
  expect(tester.widget<ElevatedButton>(button).onPressed, isNotNull);
});
```

Rysunek 5.39 Fragment kod registraion_page_test.dart (źródło: *opracowanie własne*)

Testy widżetów

Testy widżetów służą do testowania widżetów znajdujących się w aplikacji, renderując je i wykonując na nich operacje. Pozwalają one na testowanie interfejsów użytkownika w izolacji dzięki budowaniu sztucznych klas aplikacji. Do tego celu wykorzystywana jest biblioteka „mockito”[5].
Przykładem testów widżetów są testy rejestracji, które uzupełniają formularz i reagują na jego zmiany. Dzięki zastosowaniu testów widżetów w aplikacji zostało poprawione sprawdzanie pól oraz blokowanie przycisków w momencie, kiedy nie są wypełnione wszystkie wymagane pola.

Przykłady testów widżetów:

- **RegistrationPage renders correctly:** Test sprawdza, czy strona rejestracji renderuje się poprawnie, w tym czy wyświetlane są odpowiednie pola i przyciski.
- **Register button is disabled when form is invalid:** Test sprawdza, czy przycisk rejestracji jest wyłączony, gdy formularz jest nieprawidłowy (pola formularza są puste).
- **Register button is enabled when form is valid:** Test sprawdza, czy przycisk rejestracji jest włączony, gdy formularz jest poprawnie wypełniony.
- **LoginPage renders correctly**: Test sprawdza, czy strona logowania renderuje się poprawnie, w tym czy wyświetlane są odpowiednie pola i przyciski.
- **Login button is disabled when form is invalid**: Test sprawdza, czy przycisk logowania jest wyłączony, gdy formularz jest nieprawidłowy (pola formularza są puste).
- **Login button is enabled when form is valid**: Test sprawdza, czy przycisk logowania jest włączony, gdy formularz jest poprawnie wypełniony.
- **LoginPage navigates to registration page when register button is tapped**: Test sprawdza, czy po kliknięciu przycisku „Zarejestruj się” użytkownik jest przekierowywany na stronę rejestracji.
- **LoginPage navigates to reset password page when reset password button is tapped**: Test sprawdza, czy po kliknięciu przycisku „Zresetuj hasło” użytkownik jest przekierowywany na stronę resetowania hasła.
- **BottomNavBar renders correctly**: Test sprawdza, czy menu nawigacji renderuje się poprawnie.
- **BottomNavBar navigates to CarPage**: Test sprawdza, czy kliknięcie przycisku nawigacyjnego „Auto” przenosi użytkownika na stronę „CarPage”.

5.6 Testowanie – testy manualne

Tabela 5.1 Testowanie rejestracji nowego użytkownika (źródło: opracowanie własne)

	Cel	Warunki wstępne
	Sprawdzenie czy nowy użytkownik może się zarejestrować	• Aplikacja jest uruchomiona • Użytkownik ma dostęp do internetu
Kroki	• Otwórz aplikację. • Wybierz Profil z nawigacji. • Kliknij w przycisk „Nie masz konta? Zarejestruj się” • Wprowadź poprawny adres e-mail, np. test@example.com. • Wprowadź poprawne hasło, np. password123. • Kliknij na przycisk "Zarejestruj się".	
Oczekiwany wynik	Użytkownik zostaje zarejestrowany i przekierowany na stronę logowania.	
Otrzymany rezultat	Aplikacja przekierowuje do strony logowania	

Tabela 5.2 Testowanie logowania użytkownika (źródło: opracowanie własne)

	Cel	Warunki wstępne
	Sprawdzenie czy użytkownik może się zalogować	• Aplikacja jest uruchomiona • Użytkownik ma dostęp do internetu
Kroki	• Otwórz aplikację. • Wybierz Profil z nawigacji. • Wprowadź poprawny adres e-mail, np. test@example.com. • Wprowadź poprawne hasło, np. password123. • Kliknij na przycisk "Zaloguj się".	
Oczekiwany wynik	Użytkownik został zalogowany do aplikacji.	
Otrzymany rezultat	Użytkownik został przekierowany do strony głównej aplikacji.	

Tabela 5.3 Testowanie resetowania hasła (źródło: opracowanie własne)

	Cel	Warunki wstępne
	Resetowanie hasła przez użytkownika	• Aplikacja jest uruchomiona • Użytkownik ma dostęp do internetu • Użytkownik posiada konto w aplikacji
Kroki	• Otwórz aplikację. • Wybierz Profil z nawigacji. • Kliknij w przycisk „Nie pamiętasz hasła? Zresetuj hasło” • Wprowadź poprawny adres e-mail • Kliknij na przycisk "Zresetuj hasło".	
Oczekiwany wynik	Użytkownik otrzymuje maila do resetu hasła.	
Otrzymany rezultat	Aplikacja informuje że Link do resetu hasła został wysłany.	

Tabela 5.4 Testowanie zakładki analiza (źródło: opracowanie własne)

	Cel	Warunki wstępne
	Sprawdzenie ceny hurtowej dla paliwa PB98 za okres 3 Miesięcy.	• Aplikacja jest uruchomiona • Użytkownik ma dostęp do internetu
Kroki	• Otwórz aplikację. • Wybierz Analiza z nawigacji. • Przejdź do kafelka „Cena hurt”. • Wybierz przycisk PB98 oraz 3 Miesiące.	
Oczekiwany wynik	Użytkownik otrzymuje wykres prezentujący dane ceny paliwa PB95 z ostatnich 3 miesięcy.	
Otrzymany rezultat	Aplikacja przeładowuje wykres z odpowiednimi parametrami.	

Tabela 5.5 Testowanie zakładki analiza (źródło: opracowanie własne)

	Cel	Warunki wstępne
	Sprawdzenie ceny hurtowej dla paliwa PB98 za okres 3 Miesięcy.	• Aplikacja jest uruchomiona • Użytkownik ma dostęp do internetu
Kroki	• Otwórz aplikację. • Wybierz „Analiza” z nawigacji. • Przejdź do kafelka „Cena hurt”. • Wybierz przycisk PB98 oraz 3 Miesiące.	
Oczekiwany wynik	Użytkownik otrzymuje wykres prezentujący dane ceny paliwa PB95 z ostatnich 3 miesięcy.	
Otrzymany rezultat	Aplikacja przeładowuje wykres z odpowiednimi parametrami.	

Tabela 5.6 Testowanie zakładki auto (źródło: opracowanie własne)

	Cel	Warunki wstępne
	Ustawienie preferowanego paliwa na benzynę PB98 oraz wybranie średniego spalania na 10L.	• Aplikacja jest uruchomiona • Użytkownik ma dostęp do internetu • Użytkownik jest zalogowany
Kroki	• Otwórz aplikację. • Wybierz „Auto” z nawigacji. • Naciśnij przycisk „Edytuj”. • Wybierz paliwo PB98 oraz ustaw średnie spalanie na 10L.	
Oczekiwany wynik	Paliwo wyświetlane na mapie jest zgodne z wybranym.	
Otrzymany rezultat	Aplikacja zapamiętuje wybrane parametry.	

Tabela 5.7 Testowanie zakładki rabaty (źródło: opracowanie własne)

	Cel	Warunki wstępne
	Włączenie rabatu do Orlenu za kartę dużej rodziny.	• Aplikacja jest uruchomiona • Użytkownik ma dostęp do internetu • Użytkownik jest zalogowany
Kroki	• Otwórz aplikację. • Wybierz „Rabaty” z nawigacji. • Naciśnij na kafelek „Karta dużej rodziny - Orlen”.	
Oczekiwany wynik	Cena paliwa na stacji bierze pod uwagę mój rabat.	
Otrzymany rezultat	Aplikacja koloruje na zielono kafelek oraz zaznaczając że jest włączony.	

Tabela 5.8 Testowanie skanowania cen (źródło: opracowanie własne)

	Cel	Warunki wstępne
	Zeskanowanie ceny z pylona i dodanie jej do bazy danych.	• Aplikacja jest uruchomiona • Użytkownik ma dostęp do internetu • Użytkownik jest zalogowany • Użytkownik wyraził zgodę na lokalizację • Użytkownik wyraził zgodę na używanie aparatu • Użytkownik znajduje się 100m od stacji
Kroki	• Otwórz aplikację. • Wybierz „Mapa” z nawigacji. • Wybierz stacji z informacją „Możliwość zeskanowania ceny”. • Naciśnij na przycisk „Zeskanuj ceny”. • Zrób zdjęcie pylonu. • Przytnij zdjęcie tak żeby było widać tylko ceny. • Zatwierdź ceny.	
Oczekiwany wynik	Ceny zostają przesłane do bazy danych i wpływają na cenę na mapie.	
Otrzymany rezultat	Aplikacja przekierowuje na mapę.	

Tabela 5.9 Testowanie skanowania cen – złe dane (źródło: opracowanie własne)

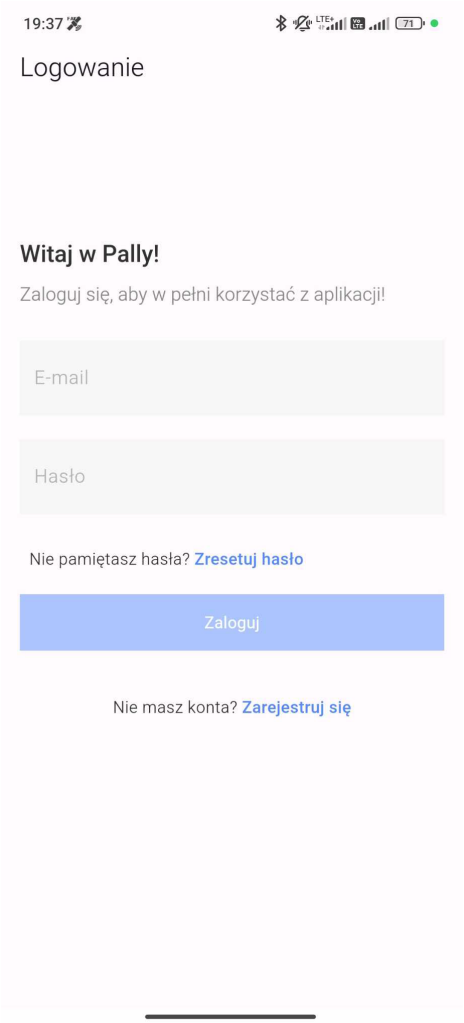
	Cel	Warunki wstępne
	Zrobienie zdjęcia nie zawierającego pylonu z cenami.	• Aplikacja jest uruchomiona • Użytkownik ma dostęp do internetu • Użytkownik jest zalogowany • Użytkownik wyraził zgodę na lokalizację • Użytkownik wyraził zgodę na używanie aparatu • Użytkownik znajduje się 100m od stacji
Kroki	• Otwórz aplikację. • Wybierz „Mapa” z nawigacji. • Wybierz stacji z informacją „Możliwość zeskanowania ceny”. • Naciśnij na przycisk „Zeskanuj ceny”. • Zrób zdjęcie niezwierające cen. • Przytnij zdjęcie.	
Oczekiwany wynik	Komunikat z informacją że nie udało się odczytać cen.	
Otrzymany rezultat	Aplikacja wyświetla informację że nie udało się zeskanować cen.	

6. Sposób wdrożenia i eksploatacji

6.1 Interfejs użytkownika aplikacji

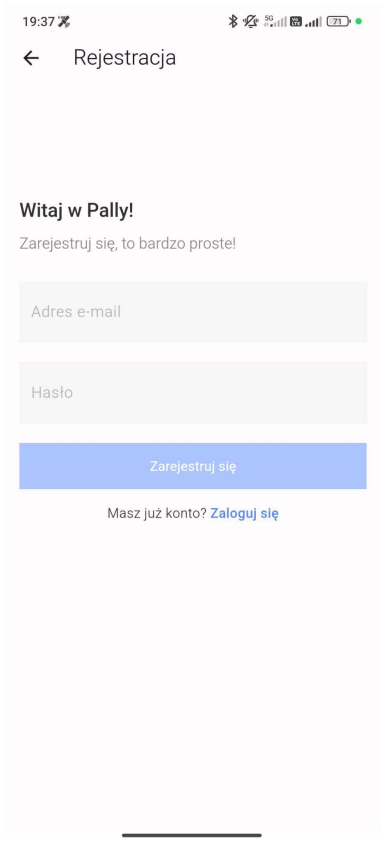
Rozdział prezentuje ekrany aplikacji Pally wraz z opisem ich zastosowania. Poniżej znajdują się zrzuty ekranów z aplikacji, które stanowią ważną część dokumentacji technicznej, pozwalając na przedstawienie interfejsu użytkownika. Będą prezentowane kluczowe funkcje i elementy aplikacji, umożliwiające pełne zrozumienie funkcjonalności.

Rysunek 6.1 przedstawia ekran logowania aplikacji Pally, zawierający pola do wprowadzenia adresu e-mail i hasła, oraz opcje resetowania hasła i rejestracji użytkownika.



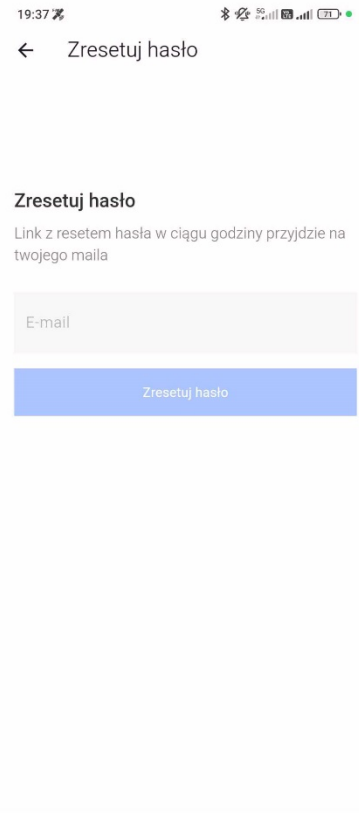
Rysunek 6.1 Ekran logowania do aplikacji (źródło: *opracowanie własne*)

Rysunek 6.2 przedstawia ekran rejestracji aplikacji Pally, zawierający pola do wprowadzenia adresu e-mail i hasła, oraz opcje rejestracji nowego użytkownika i przejścia do ekranu logowania dla tych, którzy już posiadają konto.



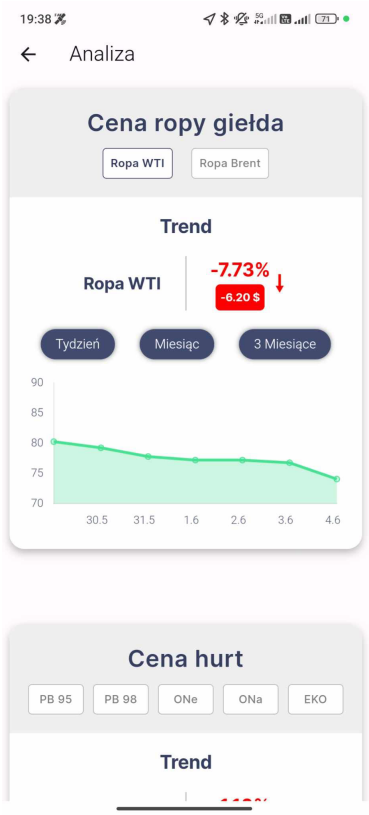
Rysunek 6.2 Ekran rejestracji do aplikacji (źródło: *opracowanie własne*)

Rysunek 6.3 przedstawia ekran resetowania hasła aplikacji Pally, zawierający pole do wprowadzenia adresu e-mail, na który zostanie wysłany link do zresetowania hasła.



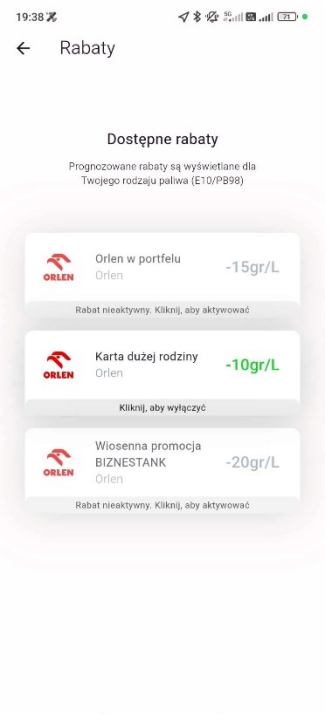
Rysunek 6.3 Ekran resetu hasła do aplikacji (źródło: *opracowanie własne*)

Rysunek 6.4 przedstawia analizę cen paliw hurtową oraz giełdę wraz z możliwościami dostosowania przedziału czasu oraz rodzaju paliwa.



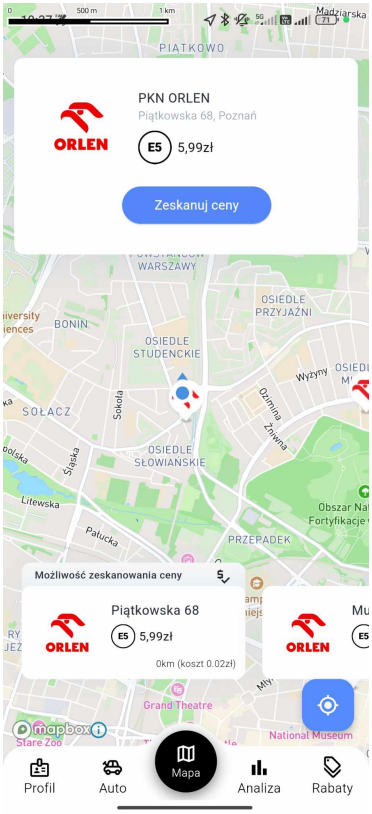
Rysunek 6.4 Ekran analizy cen paliw hurt i giełda (źródło: *opracowanie własne*)

Rysunek 6.5 przedstawia dostępne rabaty w aplikacji które użytkownik może aktywować.



Rysunek 6.5 Ekran dostępnych rabatów w aplikacji (źródło: *opracowanie własne*)

Rysunek 6.6 przedstawia mapę wraz z najbliższymi stacjami benzynowymi oraz cenami.

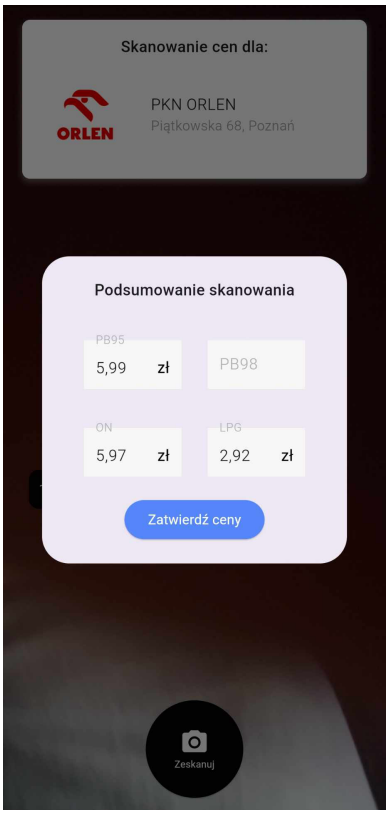


Rysunek 6.6 Ekran mapy ze stacjami paliw (źródło: *opracowanie własne*)

Rysunek 6.7 przedstawia możliwość przycinania cen przed zatwierdzeniem do analizy.

Rysunek 6.7 Przycinanie zdjęcia pylonu (źródło: *opracowanie własne*)

Rysunek 6.8 ekran prezentujący ceny odczytane ze zdjęcia.



Rysunek 6.8 *Zatwierdzanie cen paliw (źródło: opracowanie własne)*

Rysunek 6.9 ekran prezentujący ustawienia użytkownika dotyczące preferowanego paliwa oraz średniego spalania.

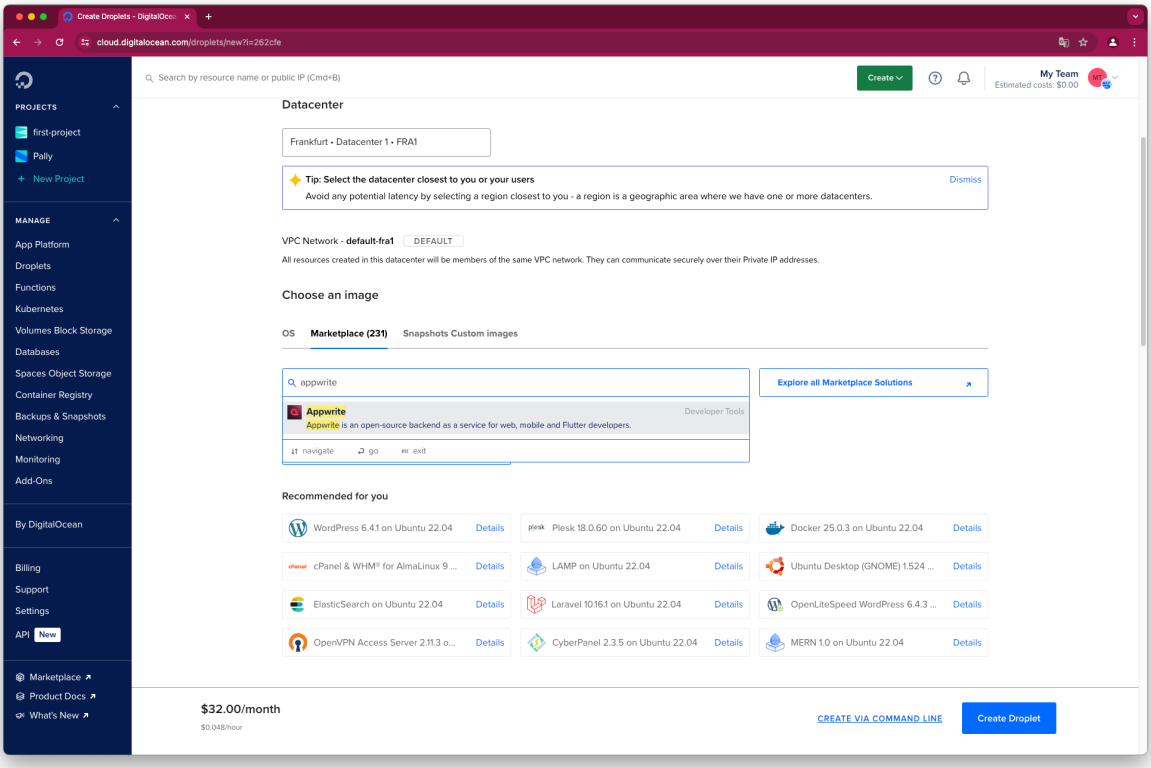
Rysunek 6.9 Ekran edycji spalania oraz typu paliwa (źródło: *opracowanie własne*)

6.2 Sposób wdrożenia backendu

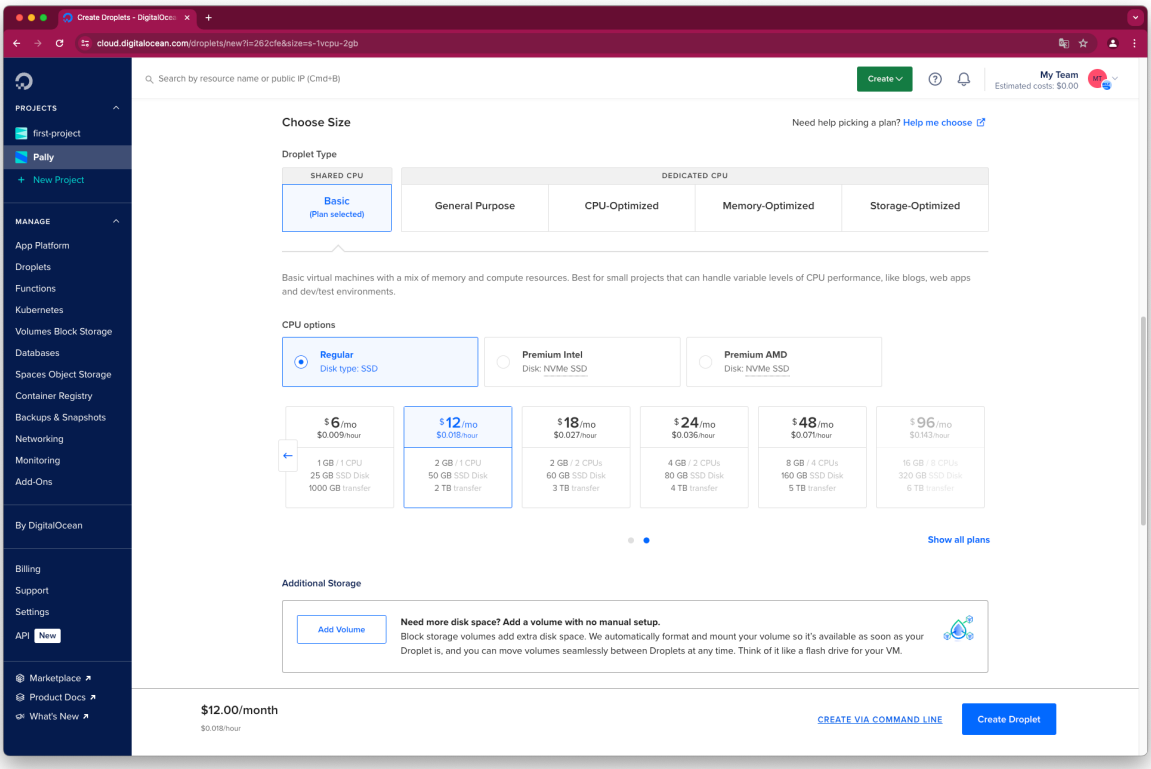
Zawartość serwera backendowego, a dokładniej wolumeny Docker, zostały skopiowane i zarchiwizowane za pomocą polecenia „tar” w systemie Linux. Dzięki temu można w bardzo łatwy sposób uruchomić cały backend, razem z danymi które udało się zebrać w ramach projektu bez dodatkowej konfiguracji.

Aby uzyskać identyczne środowisko uruchomieniowe jak w projekcie należy skorzystać z serwisu DigitalOcean. Istnieje możliwość wykorzystania dostarczonych w dowolnym środowisku Linux, jednak instrukcje skupią się na uzyskaniu bliźniaczego środowiska.

Po zalogowaniu do serwisu DigitalOcean należy przejść do zakładki „Create”. Należy wybrać dowolny region. Najważniejsze jest wybranie obrazu Appwrite z zakładki „Marketplace”. Należy wyszukać obraz „Appwrite 1.5 on Ubuntu 22.04”, a następnie dobrać rodzaj serwera. W przypadku tego projektu, minimalna instancja, na której da się uruchomić projekt to serwer z 2GB pamięci RAM, 1vCPU, 50GB przestrzeni dyskowej oraz 2TB transferu sieciowego (standardowy rodzaj CPU, dysk SSD).

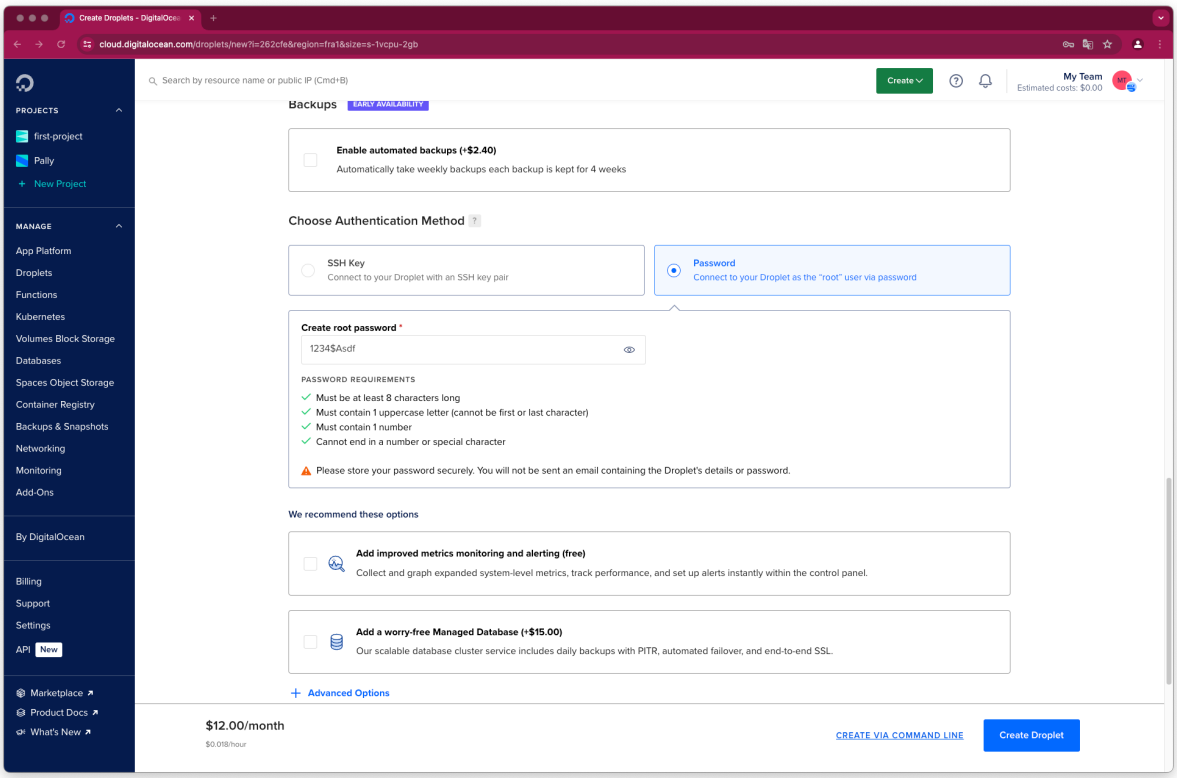


Rysunek 6.10 Wybór obrazu Appwrite do serwera (źródło: *opracowanie własne*)

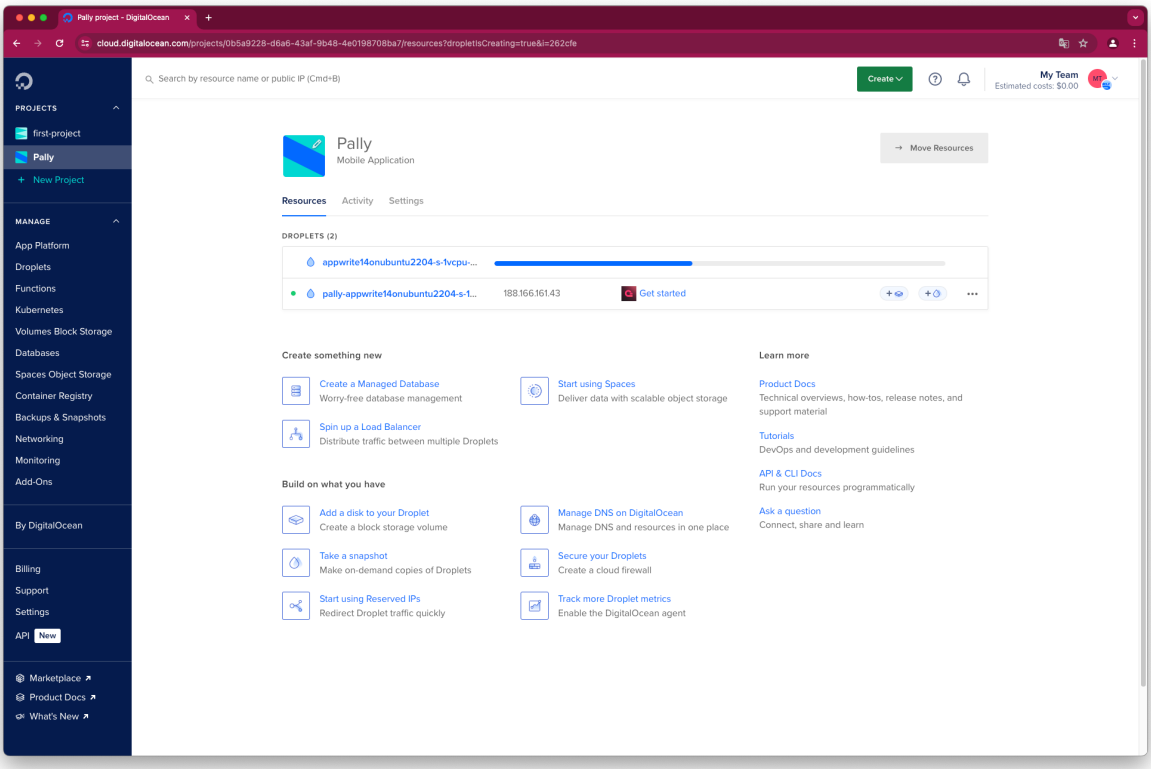


Rysunek 6.11 Wybór odpowiedniego rozmiaru instancji (źródło: *opracowanie własne*)

W sekcji „Choose Authentication Method” należy wybrać autoryzacje hasłem (ang. *Password*) oraz wypełnić je silnym hasłem. Hasło będzie potrzebne do późniejszego zalogowania się do serwera i przeniesienia plików. W pozostałych komórkach należy pozostawić domyślne wartości (lub dostosować je do swoich preferencji). Następnie należy kliknąć „Create Droplet” w celu utworzenia serwera.

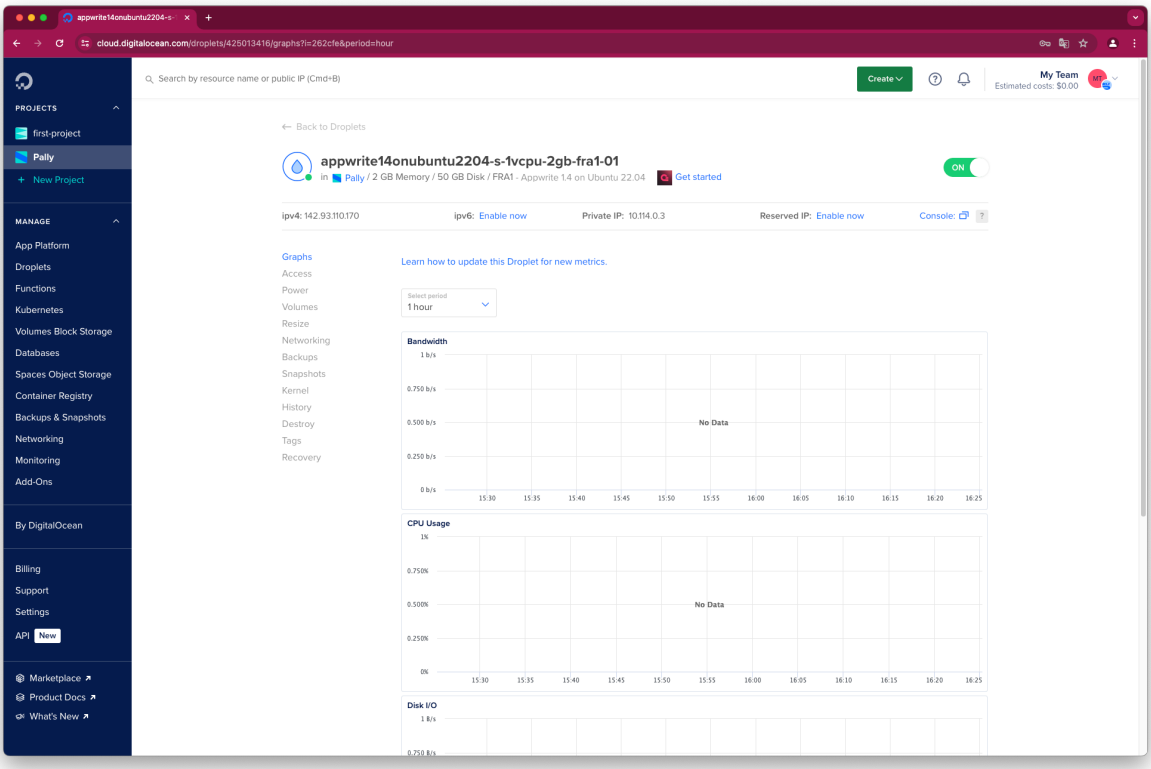


Rysunek 6.12 Wybór metody logowania do serwera (źródło: *opracowanie własne*)



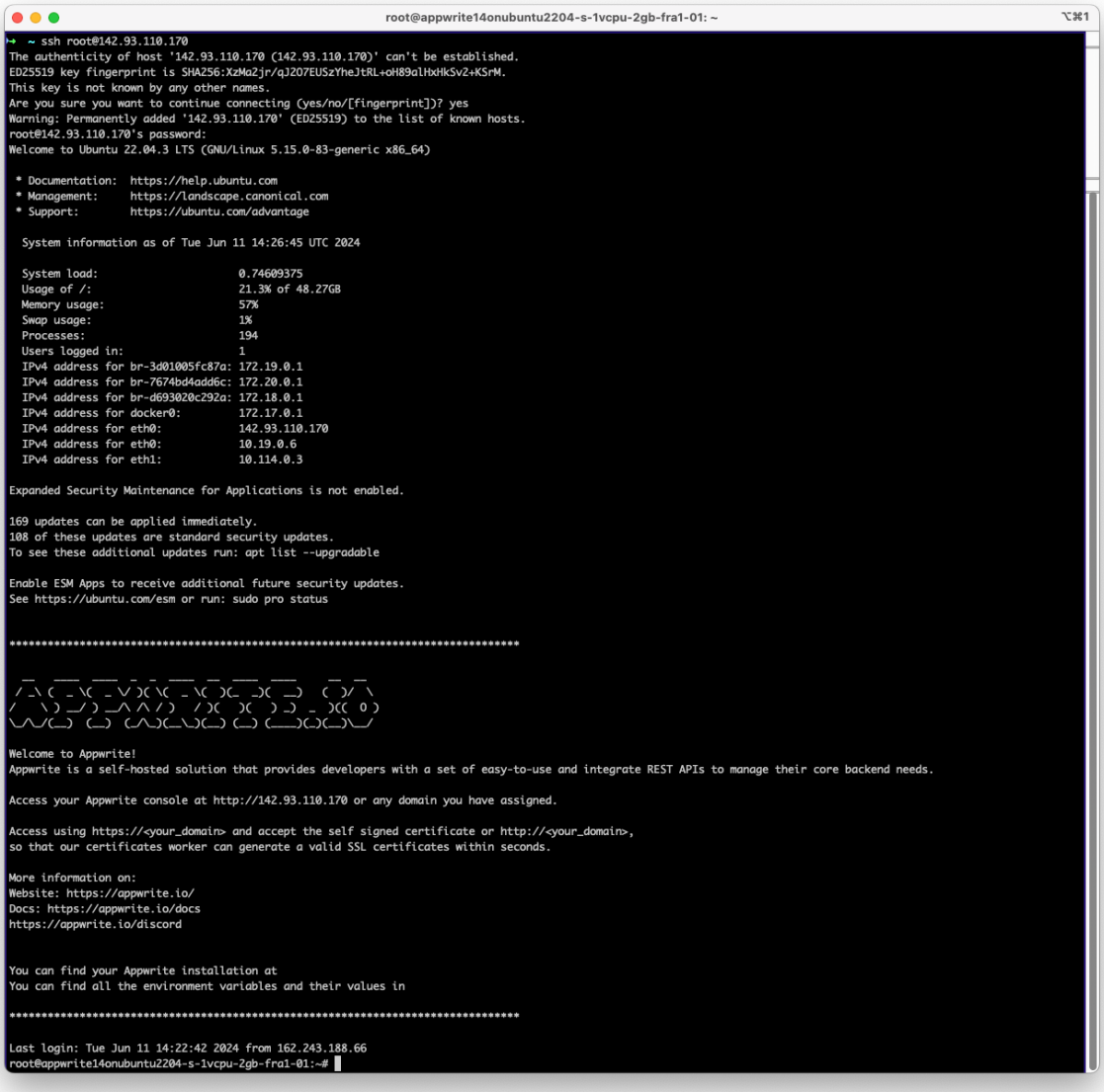
Rysunek 6.13 Proces tworzenia serwera w DigitalOcean (źródło: *opracowanie własne*)

Po zakończeniu tworzenia serwera kliknąć w wiersz z nazwą naszego serwera (w tym przypadku w wiersz o nazwie „appwrite14onubuntu2204-s-1vcpu-2gb-fra1-01”), który przekieruje na stronę z szczegółami dot. serwera.



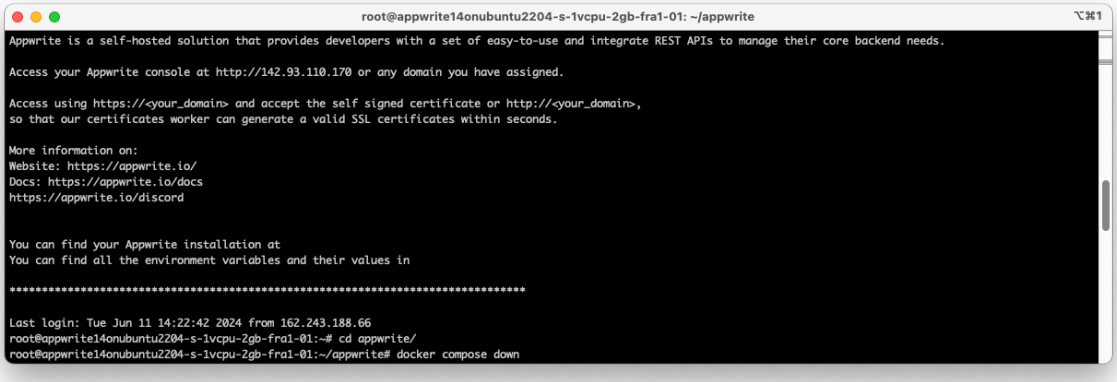
Rysunek 6.14 Szczegółowy podgląd nowoutworzonego serwera (źródło: *opracowanie własne*)

Należy skopiować adres IPv4 oraz połączyć się z serwerem za pomocą polecenia SSH.



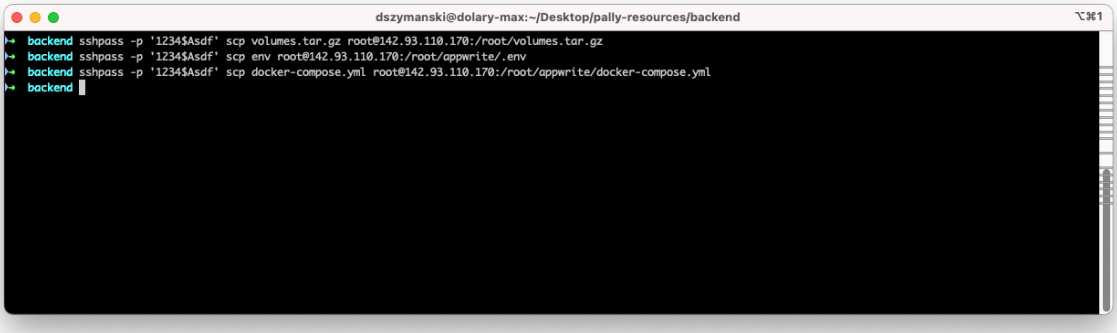
Rysunek 6.15 Połączenie SSH z serwerem (źródło: *opracowanie własne*)

Następnie należy przejść do katalogu „appwrite”, uruchomić polecenie „docker compose down” w celu zatrzymania aplikacji Appwrite.



Rysunek 6.16 Wyłączanie Appwrite na serwerze (źródło: *opracowanie własne*)

Po pomyślnym wyłączeniu, należy przejść do kopiowania plików projektu na serwer. Aby tego dokonać, należy utworzyć nowe okno terminala, przejść do katalogu z dostarczonymi plikami i wykonać polecenie „sshpass” oraz „scp” jak na rysunku 6.17.



Rysunek 6.17 Kopiowanie plików projektu na serwer (źródło: *opracowanie własne*)

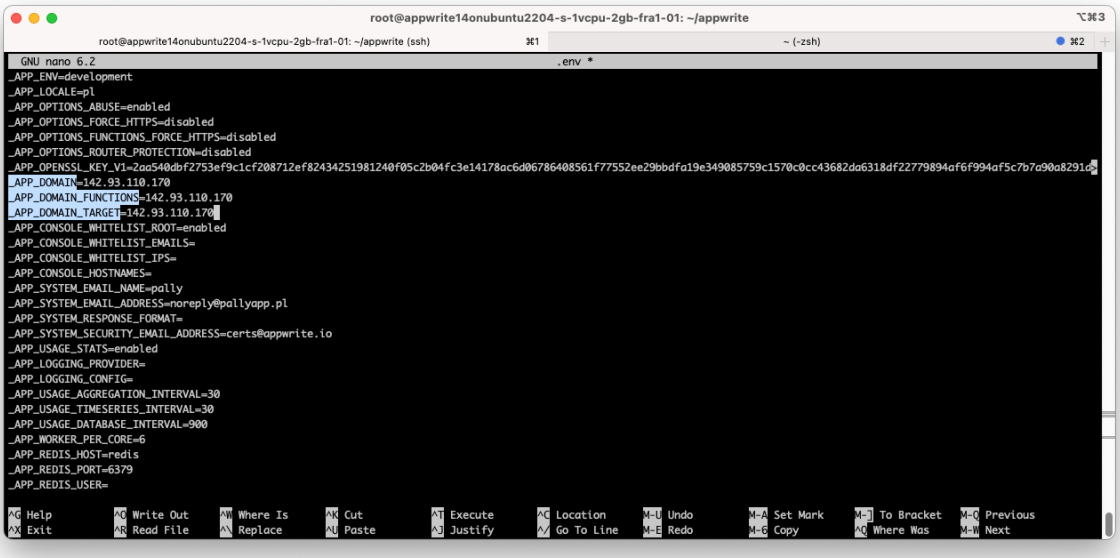
Następnie należy ponownie zalogować się do serwera za pomocą polecenia SSH (lub wykorzystać poprzednie połączenie) i wykonać polecenie „tar -xvzf volumes.tar.gz -C /” aby rozpakować wolumeny Dockera.

Rysunek 6.18 Proces rozpakowywania wolumenu Docker (źródło: *opracowanie własne*)

Po rozpakowaniu archiwum tar, należy przejść ponownie do katalogu „~/appwrite”. W katalogu, za pomocą polecenia „nano .env” należy otworzyć plik „.env” i zmodyfikować zmienne środowiskowe

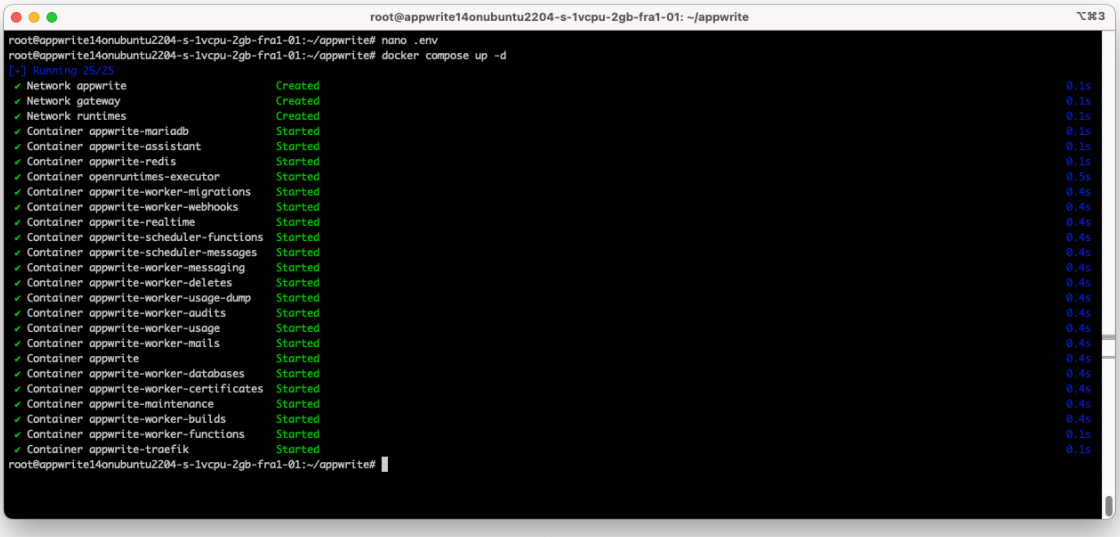
- _APP_DOMAIN
- _APP_DOMAIN_FUNCTIONS
- _APP_DOMAIN_TARGET

na adres IPv4 serwera DigitalOcean. Zapisać plik za pomocą kombinacji klawiszy „CTRL + O” oraz opuścić edytor „nano” za pomocą „CTRL + X”.



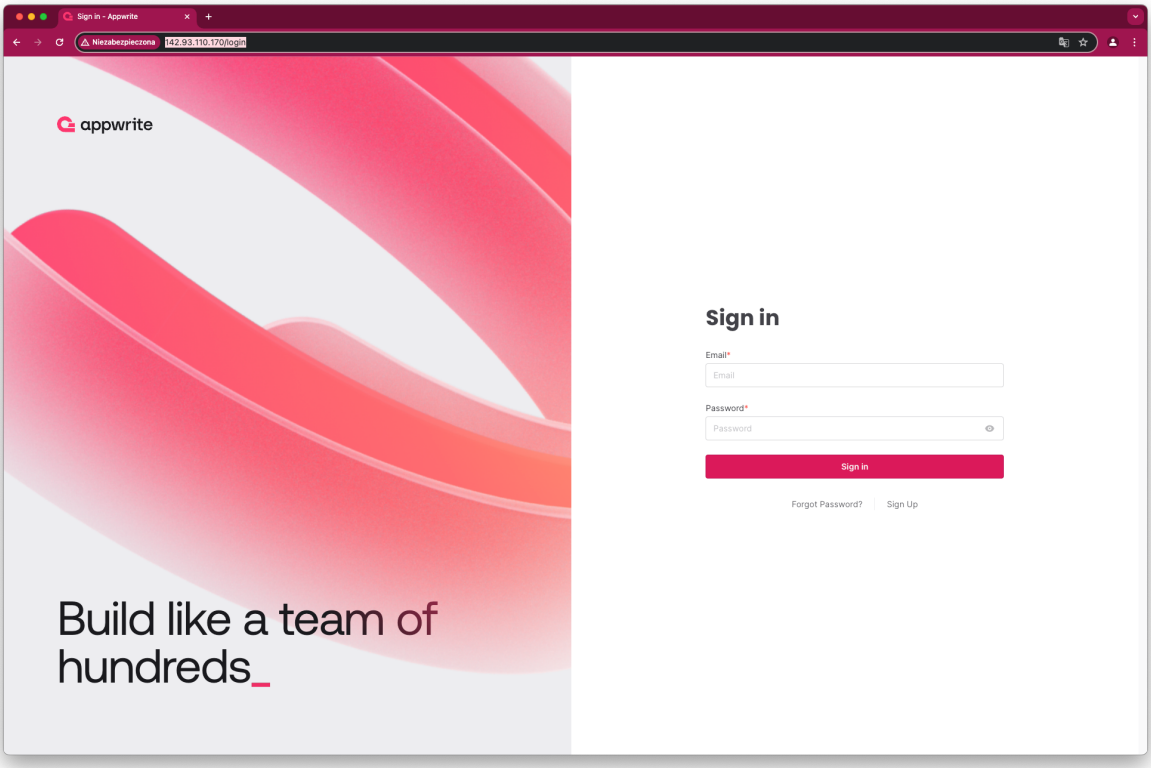
Rysunek 6.19 Zmodyfikowany plik „.env”(źródło: *opracowanie własne*)

Następnie należy ponownie uruchomić projekt za pomocą polecenia „docker compose up -d”.



Rysunek 6.20 Proces ponownego uruchamiania aplikacji (źródło: *opracowanie własne*)

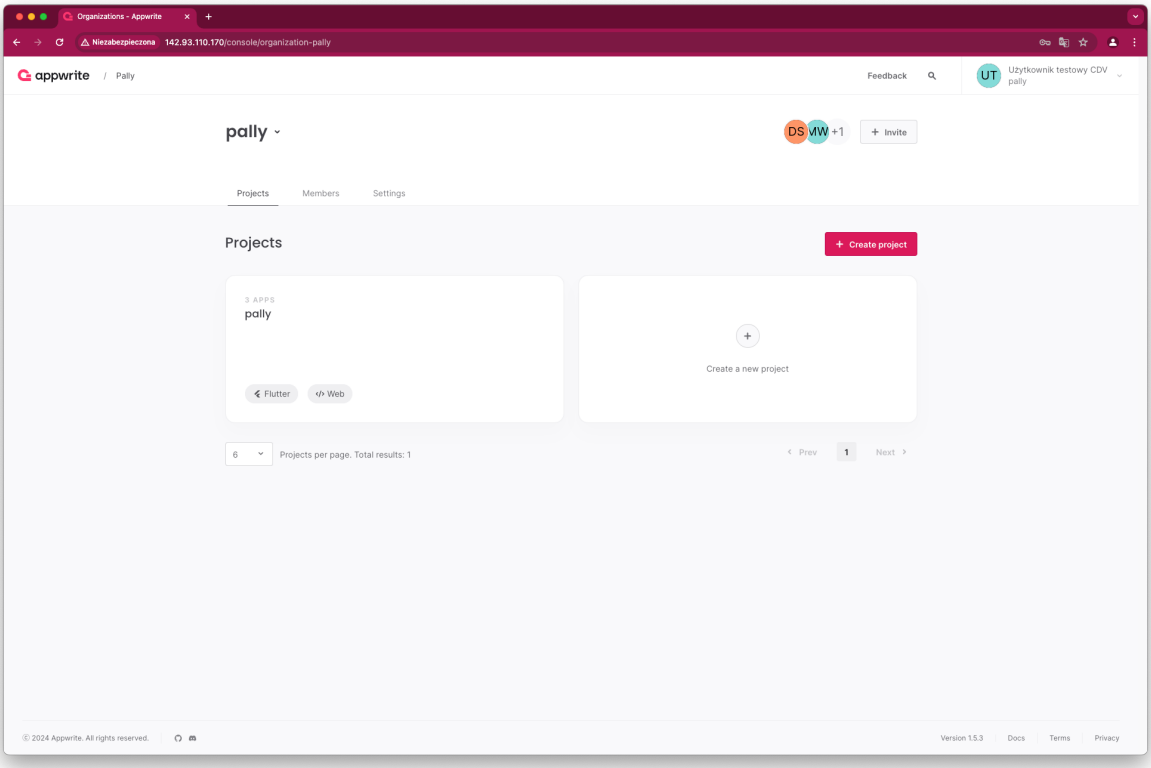
Backend aplikacji po kilku minutach będzie dostępny pod adresem IP naszej maszyny.



Rysunek 6.21 Działający backend na nowej instancji serwera (źródło: *opracowanie własne*)

Aby zalogować się do platformy, należy użyć następujących danych:

- Adres email: pally@edu.cdv.pl
- Hasło: mymso5-pyzMoz-vekfom

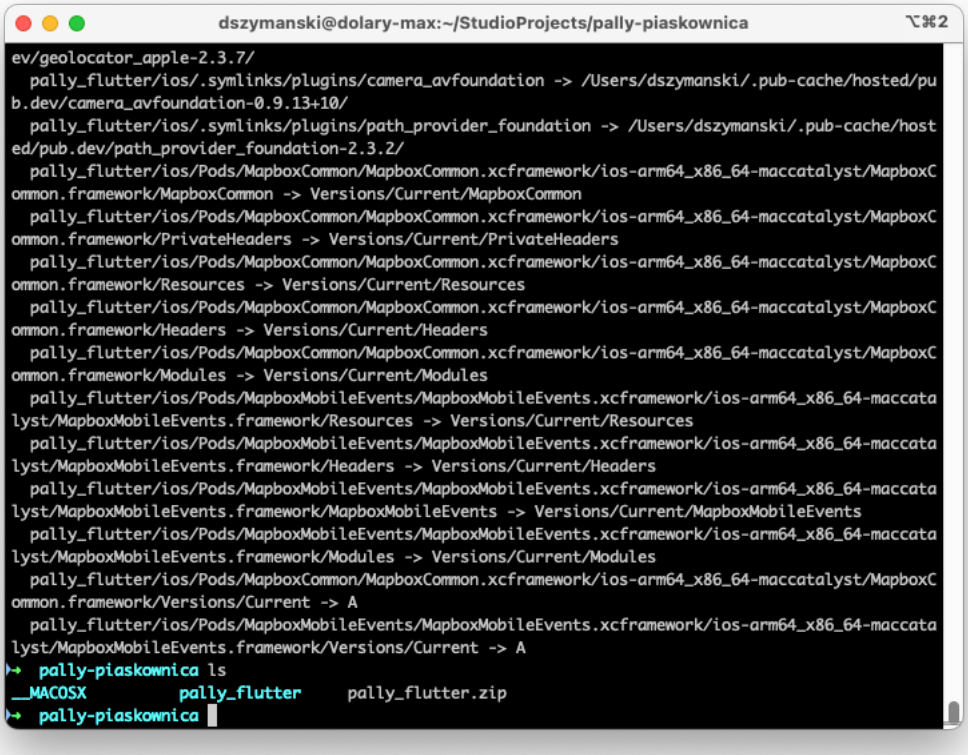


Rysunek 6.22 Widok testowego użytkownika po zalogowaniu (źródło: *opracowanie własne*)

6.3 Sposób wdrożenia aplikacji mobilnej

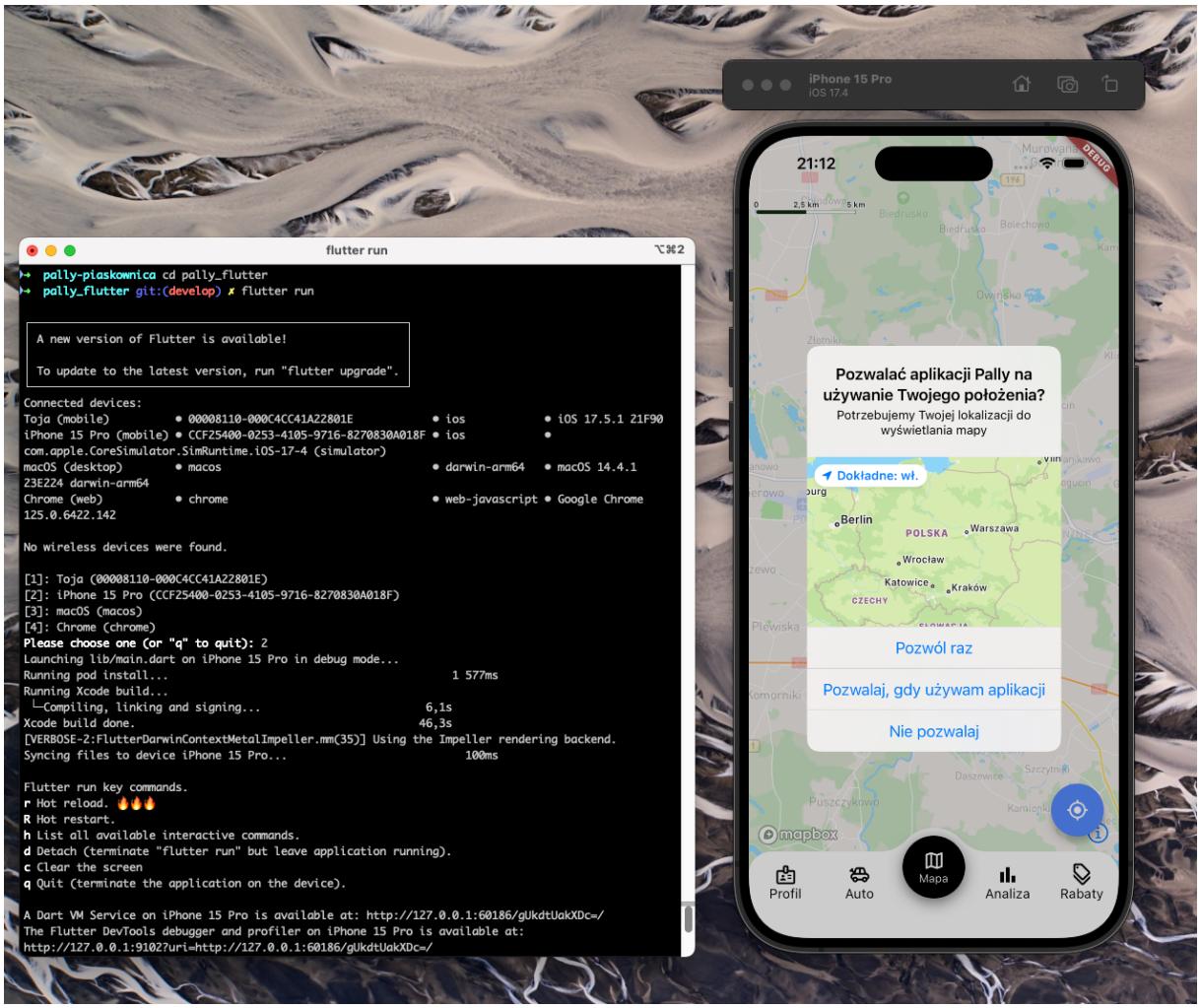
Aby uruchomić aplikację mobilna na swoim lokalnym komputerze należy zainstalować wcześniej środowisko Flutter zgodnie z instrukcjami w dokumentacji Flutter [19].

Po instalacji Flutter na komputerze, należy rozpakować dostarczony plik pally_flutter.zip za pomocą polecenia „unzip pally_flutter.zip”



Rysunek 6.23 Rozpakowywanie pliku pally_flutter.zip (źródło: *opracowanie własne*)

Po przejściu do katalogu „pally_fluttter”, należy uruchomić polecenie „flutter run”. Należy mieć uruchomiony emulator (Android lub iOS) lub podłączony smartfon w trybie debugowania.



Rysunek 6.24 Uruchamianie aplikacji mobilnej (źródło: *opracowanie własne*)

Polecenie pobierze wszystkie wymagane biblioteki potrzebne do uruchomienia projektu oraz uruchomi ją na docelowym urządzeniu.

7. Podsumowanie i wnioski

7.1 Podsumowanie

Celem głównym aplikacji była analiza cen paliw, ich dostępności na stacjach oraz monitorowanie trendów. Aplikacja została zrealizowana zgodnie z założeniami początkowymi. Dzięki Flutterowi osiągnęliśmy wydajną aplikację działającą na iOS i Androidzie, z możliwością swobodnej rozbudowy dzięki dynamicznie rozwijanej technologii.

Użycie Appwrite na backendzie pozwoliło stworzyć wielozadaniowy system, który obsługuje nie tylko komponenty aplikacji, takie jak logowanie oraz zbieranie i przetwarzanie danych z aplikacji, ale również scrapery służące do zbierania danych z innych źródeł oraz komunikację z API, takimi jak ChatGPT. Jest to nowoczesne podejście do budowy backendu, oferujące duże możliwości rozwoju aplikacji.

Aplikacja została przetestowana za pomocą testów jednostkowych oraz testów widżetów, które potwierdziły jej gotowość do wdrożenia. Dzięki serwerowi hostowanemu na Digital Ocean, aplikacja została przetestowana pod kątem działania nie tylko lokalnie, ale również w środowisku produkcyjnym.

7.2 Dalsze możliwości rozwoju aplikacji

Aplikacja Pally jest gotowa do wczesnego wdrożenia, co umożliwiłoby przetestowanie jej na większej liczbie odbiorców i zebraniu większej ilości informacji zwrotnych od użytkowników. Aby wyróżnić się na tle konkurencji, aplikacja powinna zostać rozwinięta w kilku kluczowych aspektach:

Rozbudowa na wszystkie możliwe stacje benzynowe

W aktualnej wersji aplikacja dostarcza informacje na temat cen paliw oraz lokalizacji tylko kilku marek, takich jak Orlen i Auchan. Rozbudowa na wszystkie stacje benzynowe oraz ich programy partnerskie pozwoliłaby użytkownikowi na pełną analizę cen, co przełożyłoby się na lepsze doświadczenie z aplikacją.

Implementacja modułu grywalizacji

Grywalizacja w aplikacji zachęciłaby użytkowników do częstszego skanowania cen na stacjach, co zmniejszyłoby potrzebę korzystania z zewnętrznych źródeł i zwiększyłoby niezależność oraz rzetelność dostarczanych danych o cenach paliw. Grywalizacja mogłaby polegać na zbieraniu punktów za aktualizacje cen, co mogłoby przekładać się na rabaty w konkretnych sieciach paliwowych.

Rozbudowa profilu użytkownika oraz analiza danych aplikacji

Kolejną możliwością rozwoju jest większa personalizacja oraz analiza użytkowania aplikacji poprzez rozbudowę profilu użytkownika o informacje takie jak rodzaj oraz pojemność samochodu czy ilość zatankowanego paliwa. Aplikacja byłaby w stanie, na podstawie danych historycznych, proponować, kiedy i gdzie warto zatankować. Dzięki analizie danych giełdowych oraz hurtowych mogłaby oszacować, czy ceny paliwa będą rosnąć czy spadać, a dzięki informacjom na temat pojemności oraz ilości zatankowanego paliwa mogłaby proponować optymalny moment i miejsce tankowania.

7.3 Wnioski

Praca inżynierska skupiająca się na analizie cen paliw oraz różnych aspektach jej użycia przyniosła szereg istotnych wniosków i korzyści. W trakcie realizacji projektu zespół rozwijał umiejętności w nowych, wcześniej nieużywanych technologiach, a także doskonalił kompetencje w zakresie planowania, programowania, testowania i wdrożenia aplikacji.

Wykorzystanie Appwrite umożliwiło połączenie wiedzy programistycznej z różnych technologii oraz rozwinięcie umiejętności w budowaniu powiązań między niezależnymi funkcjami. Aplikacja może stanowić znaczącą pomoc dla kierowców, którzy dzięki jej używaniu mogą oszczędzać pieniądze oraz optymalizować proces zakupu paliwa.

Projekt oferuje również możliwości marketingowe dla stacji benzynowych. Dodatkowe rabaty uwzględniające ceny na stacjach oraz rozbudowa o funkcje grywalizacji, pozwalająca przydzielać różne ilości punktów w zależności od stacji, na której znajduje się użytkownik, mogą napędzać ruch na tych stacjach.

Podsumowując, praca nad projektem nie tylko przyczyniła się do rozwoju technicznego zespołu, ale także stworzyła potencjał do realnych korzyści dla użytkowników i stacji benzynowych.

8. Bibliografia

[1] *Android Studio*, https://developer.android.com/studio?hl=pl (dostęp: 23.10.2023)
[2] *Autocentrum.pl*, https://www.autocentrum.pl (dostęp: 18.10.2023)
[3] *AWS S3*, https://aws.amazon.com/s3/ (dostęp: 14.12.2023)
[4] *Backblaze*, https://www.backblaze.com (dostęp: 09.05.2024)
[5] *Biblioteka „mockito”*, https://pub.dev/packages/mockito (dostęp: 05.02.2024)
[6] *Biblioteka „get_it”*, https://pub.dev/packages/get_it (dostęp: 03.02.2024)
[7] *Chart js - Step-by-step guide,* https://www.chartjs.org/docs/latest/getting-started/usage.html (dostęp: 18.02.2024)
[8] *ChatGPT*, https://openai.com/chatgpt/ (dostęp: 24.10.2023)
[9] *Co to jest Fuelo.net?*, https://fuelo.net/about?lang=pl (dostęp: 12.10.2023)
[10] *Czym jest Scrum?*, https://www.atlassian.com/pl/agile/scrum (dostęp: 15.10.2023)
[11] *Dart*, https://dart.dev (dostęp: 02.12.2023)
[12] *Dependency injection*, https://en.wikipedia.org/wiki/Dependency_injection (dostęp: 01.02.2024)
[13] *DigitalOcean Spaces,* https://www.digitalocean.com/products/spaces (dostęp: 08.05.2024)
[14] *DigitalOcean*, https://try.digitalocean.com/cloud/ (dostęp: 03.11.2023)
[15] *Discord*, https://discord.com/ (dostęp: 23.10.2023)
[16] *Dokumentacja Appwrite*, https://appwrite.io/docs (dostęp: 12.12.2023)
[17] *Figma*, https://www.figma.com (dostęp: 20.10.2023)
[18] *Flutter Charts*, https://pub.dev/packages/fl_chart (dostęp: 01.06.2024)
[19] *Flutter installation*, https://docs.flutter.dev/get-started/install (dostęp: 04.12.2023)
[20] *Flutter*, https://docs.flutter.dev/ (dostęp: 12.12.2023).
[21] *Fuelio: dziennik I ceny paliwa*, https://play.google.com/store/apps/details?id=com.kajda.fuelio&hl=pl (dostęp: 03.03.2024)
[22] *Git*, https://www.git-scm.com (dostęp: 08.02.2024)
[23] *Gitflow workflow*, https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow (dostęp: 20.10.2023)
[24] *Google Maps*, https://www.google.com/maps/about/#!/ (dostęp: 13.01.2024)
[25] *Google*, https://pl.wikipedia.org/wiki/Google (dostęp: 02.04.2024)
[26] *Jira*, https://www.atlassian.com/pl/software/jira (dostęp: 23.10.2023)
[27] *Kod maszynowy*, https://pl.wikipedia.org/wiki/Język_maszynowy (dostęp: 04.04. 2024)
[28] *Kotlin*, https://kotlinlang.org (dostęp: 23.03.2024)
[29] *Mapbox*, https://www.mapbox.com/ (dostęp: 11.11.2023)
[30] *MariaDB*, https://mariadb.org (dostęp: 03.01.2024)
[31] *Material Desing for Flutter*, https://docs.flutter.dev/ui/design/material (dostęp: 03.02.2024)

[32] *Navigator class*, https://api.flutter.dev/flutter/widgets/Navigator-class.html (dostęp: 03.02.2024)
[33] *NodeJS*, https://nodejs.org/en (dostęp: 05.02.2024)
[34] *OAuth2*, https://oauth.net/2/ (dostęp: 04.05.2024)
[35] *PHP*, https://www.php.net (dostęp: 04.02.2024)
[36] *Prawo Millera*, https://stripes-design.pl/ux/prawo-millera/ (dostęp: 23.10.2023)
[37] *Python*, https://www.python.org (dostęp: 03.022024)
[38] *SendGrid*, https://sendgrid.com/en-us (dostęp: 21.10.2023)
[39] *StatefulWidget class*, https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html (dostęp: 03.02.2024)
[40] *Visual Studio Code*, https://code.visualstudio.com (dostęp: 22.10.2023)
[41] *Waze*, https://www.waze.com/company (dostęp: 04.04.2024)
[42] *webview_flutter*, https://pub.dev/packages/webview_flutter (dostęp: 02.02.2024)

9. Spis rysunków

10. Spis tabel

Streszczenie

Projekt inżynierski koncentruje się na stworzeniu aplikacji mobilnej do analizy cen paliw o nazwie „Pally”, umożliwiającej użytkownikom szybkie i dokładne porównywanie cen paliw na stacjach benzynowych w ich rejonie. Aplikacja, stworzona w technologii Flutter, działa zarówno na systemach iOS, jak i Android. Backend wykorzystuje Appwrite, wspierający wielozadaniowe zarządzanie danymi oraz integrację z różnymi API, w tym ChatGPT, do przetwarzania zdjęć pylonów z cenami paliw.

Pally oferuje użytkownikom funkcje takie jak monitorowanie cen paliw, przesyłanie danych do bazy, wyświetlanie stacji na mapie oraz analizę cen na podstawie zdjęć pylonów. Użytkownicy mogą rejestrować konta, logować się, resetować hasła oraz przeglądać wykresy cen, co pomaga im w wyborze najlepszych momentów do tankowania.

Projekt obejmuje zaawansowaną analizę relacji między cenami hurtowymi a detalicznymi na stacjach benzynowych. To podejście ma na celu zwiększenie świadomości użytkowników na temat mechanizmów wpływających na zmiany cen paliw, co może prowadzić do bardziej świadomych decyzji zakupowych i zmniejszenia zużycia paliwa poprzez wybór alternatywnych środków transportu.

Aplikacja została zaprojektowana z myślą o elastyczności i przyszłym rozwoju. Plany na przyszłość obejmują rozszerzenie funkcjonalności o grywalizację, mającą na celu zachęcenie użytkowników do częstszego aktualizowania cen paliw oraz integrację z większą liczbą stacji benzynowych i programów partnerskich. Grywalizacja może obejmować zbieranie punktów za skanowanie cen, które mogą być wymieniane na rabaty na stacjach paliw.

Pally ma na celu nie tylko ułatwienie konsumentom zarządzania budżetem paliwowym, ale również przyczynienie się do bardziej ekologicznego i ekonomicznego korzystania z zasobów.

Słowa kluczowe: analiza cen paliw, aplikacja mobilna, przetwarzanie zdjęć, scrapping, ChatGPT, Flutter, Appwrite

Abstract

The engineering project focuses on creating the "Pally" mobile application for fuel price analysis, allowing users to quickly and accurately compare fuel prices at gas stations in their area. The application, developed using Flutter technology, operates on both iOS and Android systems. The backend utilizes Appwrite, supporting multitasking data management and integration with various APIs, including ChatGPT, to process images of fuel price pylons.

Pally offers users features such as fuel price monitoring, data submission to the database, displaying stations on a map, and price analysis based on pylon images. Users can register accounts, log in, reset passwords, and view price charts, helping them choose the best times to refuel.

The project includes advanced analysis of the relationship between wholesale and retail fuel prices at gas stations. This approach aims to increase users' awareness of the mechanisms influencing fuel price changes, leading to more informed purchasing decisions and reduced fuel consumption through the selection of alternative transportation methods.

The application is designed with flexibility and future development in mind. Future plans include expanding functionality with gamification to encourage users to update fuel prices more frequently and integrating with more gas stations and partner programs. Gamification may involve collecting points for scanning prices, which can be exchanged for discounts at gas stations.

Pally aims not only to help consumers manage their fuel budget but also to contribute to more ecological and economical resource usage.

Keywords: fuel price analysis, mobile application, image processing, scrapping, ChatGPT, Flutter, Appwrite

COLLEGIUM DA VINCI

UCZELNIA LUDZI CIEKAWYCH

ARKUSZ PODZIAŁU PRAC
WYKONYWANYCH PRZEZ STUDENTÓW W RAMACH REALIZACJI
WIELOAUTORSKICH (ZESPOŁOWYCH) PRAC INŻYNIERSKICH
NA WYDZIALE NAUK STOSOWANYCH
COLLEGIUM DA VINCI W POZNANIU

Temat pracy: Aplikacja mobilna do analizy cen paliw

WSPÓŁAUTOR 1:	Dominik	Szymański	
	imię	nazwisko	
WSPÓŁAUTOR 2:	Mateusz	Wiemann	
	imię	nazwisko	
WSPÓŁAUTOR 3:			
	imię	nazwisko	
WSPÓŁAUTOR 4:			
	imię	nazwisko	
WSPÓŁAUTOR 5:			
	imię	nazwisko	
PROMOTOR:	dr	Tomasz	Tyksiński
	tytuł/stopień	imię	nazwisko

Collegium Da Vinci
ul. Gen. Tadeusza Kutrzeby 10
61-719 Poznań
61 271 10 10 / info@cdv.pl

1. Całościowy udział w pracy określony procentowo przez studenta weryfikowany przez promotora w procesie dyplomowania:

	PROCENTOWY UDZIAŁ WSPÓŁAUTORA W PRACY	SUMA W %
WSPÓŁAUTOR 1	50%	100%
WSPÓŁAUTOR 2	50%	
WSPÓŁAUTOR 3		
WSPÓŁAUTOR 4		
WSPÓŁAUTOR 5		

2. Procentowy udział współautorów pracy w realizacji zadań wynikających z wymogów stawianych inżynierskim pracom dyplomowym*:

LP	OPIS ZADANIA	PROCENTOWY UDZIAŁ WSPÓŁAUTORA W WYKONANIU ZADANIA					SUMA W %
		WSPÓŁAUTOR 1	WSPÓŁAUTOR 2	WSPÓŁAUTOR 3	WSPÓŁAUTOR 4	WSPÓŁAUTOR 5	
1	Opracowanie wstępu pracy	60%	40%				100%
2	Określenie aktualnego stanu wiedzy	80%	20%				100%
3	Określenie celu, zakresu projektu i podział zadań w projekcie	90%	10%				100%
4	Opracowanie metodyki pracy	50%	50%				100%
5	Wytworzenie (kodowanie) front-end	30%	70%				100%
6	Wytworzenie (kodowanie) back-end	50%	50%				100%
7	Przygotowanie dokumentacji zgodnej z wymaganiami inżynierii oprogramowania	10%	90%				100%
8	Przygotowanie opisu projektu	40%	60%				100%
9	Procedura testowania oprogramowania	0%	100%				100%
10	Przygotowanie opisu sposobu wdrożenia/instalacji/eksploatacji	100%	0%				100%
11	Opracowanie podsumowania, wniosków	0%	100%				100%
12	Edycja i redakcja pracy (zgodnie z obowiązującym dokumentem – Wymogi Formalne - Informatyka	80%	20%				100%
13	DevOps	100%	0%				100%

* Proszę wypełniać tylko zadania realizowane w ramach pracy.

COLLEGIUM DA VINCI

UCZELNIA LUDZI CIEKAWYCH

Załącznik stanowi podstawę oceny pracy indywidualnego wkładu każdego studenta w realizacji pracy dyplomowej i stanowi integralną część pracy dyplomowej jako ostatnia nienumerowana strona pracy.

Załącznik podpisują autorzy pracy i jest on zatwierdzony przez promotora.

Realizacja poszczególnych zadań przez autorów zależna jest od charakteru pracy.

WSPÓŁAUTOR 1: Dominik Sygocki
podpis

WSPÓŁAUTOR 2: Mateusz Wiesmann
podpis

WSPÓŁAUTOR 3:
.....
podpis

WSPÓŁAUTOR 4:
.....
podpis

WSPÓŁAUTOR 5:
.....
Podpis

PROMOTOR:
.....
podpis

Poznań, 7.06.2024
miejscowość, data