POLITECHNIKA ŁÓDZKA

Wydział Elektrotechniki, Elektroniki,

Informatyki i Automatyki

Praca dyplomowa Inżynierska

System informatyczny do dzielenia informacji o cenach paliw na stacjach oparty o architekturę REST oraz PWA

IT system for sharing fuel price information based on REST architecture and PWA

Dawid Witaszek

Nr albumu 215920

Opiekun pracy:

Dr inż. Paweł Marciniak

Łódź, 2021

Streszczenie

Celem pracy dyplomowej było opracowanie aplikacji umożliwiającej dzielenie informacji o cenach paliw na stacjach oraz dodatkowe funkcjonalności. Całość systemu jest złożona z 3 modułów czyli aplikacji bazodanowej, serwerowej przeprowadzającej wszystkie operacje dziejące się w systemie oraz klienckiej dostarczającej interfejs użytkownika. Aplikacja serwerowa została zrealizowana dzięki frameworkowi Spring i rozszerzenie Hibernate umożliwiające mapowanie obiektowo-relacyjne. Aplikacja korzysta również z zewnętrznych serwisów w celu realizacji poszczególnych funkcjonalności.

Aplikacja kliencka jest aplikacją przeglądarkową opracowaną przy użyciu frameworka Angular. Interfejs użytkownika umożliwia sprawne poruszanie się po systemie oraz przeprowadzanie operacji. Dzięki zastosowaniu RWD aplikacja może być odtwarzana na urządzeniach mobilnych jak i desktopowych. Wykorzystanie funkcjonalności PWA pozwoliło poprawić doświadczenie użytkownika korzystającego z aplikacji w taki sposób że, odnosi on wrażenie korzystania z aplikacji natywnej.

Praca składa się z sześciu rozdziałów które opisują następujące tematy. Pierwszy rozdział mówi o motywacjach autora oraz celach i założeniach pracy. Drugi rozdział opisuję wykorzystane technologie przy pracy nad aplikacją. Kolejny rozdział został poświęcony praktyką zgodnie z którymi został opracowany system. Czwarty rozdział w szczegółowy sposób opisuję architekturę stworzonego systemu oraz argumentuje wybory podjęte przy projektowaniu systemu. Piąty rozdział poświęcony jest aplikacji klienckiej. Rozdział ten przedstawia poszczególne ekrany znajdujące się w aplikacji oraz opisuję dostępne funkcjonalności. W ostatnim rozdziale znajduje się podsumowanie całej pracy, które obejmuje zrealizowane cele.

Abstract

The objective of this work was to develop application that allows sharing fuel price inforation and additional assets. System is built with three modules which are data base application, server application that proces all opperations proceeded in system, and client application which provides user interface. Server application was implemented using Spring framework and Hibernate extension which provides object-relational mapping. Application also uses external services.

Client appliocation is a browser application made with Angular framework. User interface allows efficient moving around the system and invoking operations. Thanks to the use of RWD, application can run on mobile devices as well as on desktop devices. PWA functionalities increase user experiance in the way that user feels that he use native application.

The document consists of six parts that describe following subjects. First part tells about author’s motivation, objectives and requirements. Second one describes used technologies in project. Next part is about principels used in application and it’s design. Fourth part describes architechture of created system in details and shows reasons of choices durring designing system. Fifth part is about client application. That part contains screenshots of several views of application and describes functionalities. In the last part is summary that contains description of completed objectives.

**Spis treści**

[1. Wstęp 6](#_Toc64583183)

[1.1. Motywacja 7](#_Toc64583184)

[1.2. Cele i założenia 7](#_Toc64583186)

[2. Wykorzystane technologie oraz narzędzia 8](#_Toc64583187)

[2.1. Java 8](#_Toc64583188)

[2.2. Spring 8](#_Toc64583189)

[2.3. Spring MVC 9](#_Toc64583190)

[2.4. Spring Data 9](#_Toc64583191)

[2.5. Spring Context 9](#_Toc64583192)

[2.6. Spring Boot 10](#_Toc64583193)

[2.7. Bazy danych 10](#_Toc64583194)

[2.8. PostgreSQL 11](#_Toc64583195)

[2.9. H2 11](#_Toc64583196)

[2.10. Hibernate 11](#_Toc64583197)

[2.11. Zewnętrzne API 12](#_Toc64583198)

[2.12. Google APIs 12](#_Toc64583199)

[2.13. Infrastruktura paliw ciekłych w Polsce 12](#_Toc64583200)

[2.14. Java Faker 13](#_Toc64583201)

[2.15. MapStruct 13](#_Toc64583202)

[2.16. Lombok 14](#_Toc64583203)

[2.17. JJWT 14](#_Toc64583204)

[2.18. Swagger 15](#_Toc64583205)

[2.19. Postman 15](#_Toc64583206)

[2.20. Intelij IDEA 16](#_Toc64583207)

[2.21. Google Maps JavaScript API 16](#_Toc64583208)

[2.22. Angular 16](#_Toc64583209)

[2.23. Angular Material 17](#_Toc64583210)

[2.24. RXJS 17](#_Toc64583211)

[2.25. Sass 18](#_Toc64583212)

[2.26. GIT/GITHUB 18](#_Toc64583213)

[2.27. HEROKU 18](#_Toc64583214)

[3. Wykorzystane praktyki 19](#_Toc64583215)

[3.1. Clean Code 19](#_Toc64583216)

[3.2. Dobre praktyki 20](#_Toc64583217)

[3.3. Wzorce projektowe 20](#_Toc64583218)

[3.4. Paradygmaty programowania 21](#_Toc64583219)

[3.5. Programowanie funkcyjne 21](#_Toc64583220)

[3.6. Programowanie procedularne 22](#_Toc64583221)

[3.7. Programowanie Obiektowe 22](#_Toc64583222)

[3.8. Łączenie paradygmatów programowania 23](#_Toc64583223)

[4. Architektura aplikacji 24](#_Toc64583224)

[5. Aplikacja kliencka 39](#_Toc64583225)

[5.1.1. Ekran powitalny 40](#_Toc64583226)

[5.1.2. Ekran rejestracji 41](#_Toc64583227)

[5.1.3. Ekran powitalny 42](#_Toc64583228)

[5.1.4. Ekran powitalny 42](#_Toc64583229)

[5.1.5. Menu 45](#_Toc64583230)

[5.1.6. Ekran szczegółów stacji 46](#_Toc64583231)

[5.1.7. Ekran statystyk cen na stacji 47](#_Toc64583232)

[5.1.8. Ekran edycji szczegółów stacji 48](#_Toc64583233)

[5.1.9. Ekran rankingów 49](#_Toc64583234)

[5.1.10. Ekran dodania nowej stacji 50](#_Toc64583235)

[5.1.11. Profil użytkownika 51](#_Toc64583236)

[5.1.12. Ekran ulubionych stacji 51](#_Toc64583237)

[6. Podsumowanie 54](#_Toc64583238)

[7. Bibliografia 56](#_Toc64583239)

# Wstęp

Dawniej zapis takiej informacji jak cena na stacji paliw nie było możliwe ponieważ ceny te były ustalane lokalnie i nie były w żaden sposób dostępne dla informacji publicznej prócz znaku z cenami przed stacją paliw. W dzisiejszych czasach posiadamy narzędzia które umożliwiają prowadzenie takiego rejestru oraz udostępnianie go każdemu człowiekowi. Ponad to technologie te umożliwiają stworzenie systemu który jest automatyczny i nie wymaga pracy rzeszy osób taj jak miało to miejsce jeszcze kilkadziesiąt lat temu.

Wprowadzanie danych takich jak na przykład ceny paliw jest w dzisiejszych czasach możliwe ponieważ obecne telefony umożliwiają korzystanie z aplikacji do tego przeznaczonych. Jak podaje w swojej infografice serwis Domo[1] już w 2019 roku w ciągu jednej minuty użytkownicy wyszukują w serwisie Google około 4.5mln haseł. Ilość danych tworzonych przez użytkowników internetu jest przytłaczająca ale i posiada olbrzymią wartość rynkową. Serwis Forbes[2] w swoim artykule o wartości danych przez jednego użytkownika przywołuje badania MIT które określają WTA czyli „willingnes to accept”. WTA to nic innego jak utrata pieniężna po przez stratę użytkownika. WTA dla Facebooka w 2019 roku wynosiło około 50$. Jest to strata jaką ponosi Facebook po usunięciu konta użytkownika. Natomiast gdyby użytkownik wyszukiwarki Google przestał z niej korzystać firma straciłaby 8 000$. Kwoty te biorą się z tego że, firmy te nie będą dłużej agregować danych stworzonych przez użytkownika. Skłania to ku refleksji jak dużą wartość stanowi agregowanie danych.

## ***Motywacja***

Motywacją do pisania niniejszej pracy jest chęć poznania nowych technologii umożliwiających sprawną implementację aplikacji internetowych. Wspomniane nowe technologie są to kolejno architektura aplikacji, protokoły sieciowe, wzorce projektowe oraz praktyki takie jak „Clean Code”[3]. Poznanie poruszonych tematów jest wymogiem aby sprawnie orientować się w wymaganiach rynku. Dzisiejszy rozwój sprawił że, programista musi posiadać bardzo szeroką gamę umiejętności. Znajomość języka programowania jest w dzisiejszym świecie jedynie zalążkiem. Przeciętny programista musi wykazywać się znajomością wielu języków, wzorców czy frameworków.

## Cele i założenia

Celem projektu jest opracowanie systemu informatycznego o architekturze REST oraz technologii służących bądź wspierających rozwój aplikacji internetowych. Kolejnym założeniem było wykorzystanie zewnętrznych API w celu zasilenia bazy danych informacjami oraz poszerzenie funkcjonalności aplikacji. Wykorzystane wzorce, koncepcje oraz praktyki mają umożliwić łatwe utrzymanie kodu aplikacji oraz jej rozszerzalność. Wybrana architektura aplikacji jest dziś swego rodzaju standardem. Niniejsza praca zakłada opisanie poszczególnych aspektów rozwoju aplikacji. Aplikacja służy do dzielenia się opiniami, cenami oraz udogodnieniami dostępnymi na stacjach paliw. Przeglądając dostępne aplikacje nie znajdziemy żadnej która jednocześnie udostępniałaby informacji o położeniu stacji oraz informacji o cenach znajdujących się na niej paliw. Całość aplikacji składa się z trzech modułów:

* aplikacja kliencka - PolishStation
* aplikacja serwerowa – PolishStationApi
* baza danych – PolishStationDB

# Wykorzystane technologie oraz narzędzia

1. Java

Java jest to kompilowany wysokopoziomowy język programowania [4][5]. Język jest oparty na klasach, współbieżny, obiektowy oraz ogólnego zastosowania. Java oryginalnie należąca do firmy Sun Microsystems została zaprojektowana przez Jamesa Gosslinga i wydana na świat w roku 1995. Cechami języka było oparcie go składniowo na starszym języku programowania C/C++ jednak zachowując większą zwięzłość kodu oraz prostatę. Prostota wynikała głównie z braku konieczności zarządzania pamięcią. Najważniejszą etykietą przypisywaną językowi w tamtym czasie był slogan wielokrotnie powtarzany przez jego twórców „write once, run anywhere” co miało oznaczać że kod napisany na jednej platformie będzie działał podobnie na wszystkich. Dzięki właśnie tej etykiecie język stał się bardzo modny i chętnie używany w projektach wszelakiego rodzaju. Przypisanie tej etykiety było umożliwione przez platformę na której działa Java czyli Java Virtual Machine. Dzięki świeżemu podejściu w którym kod jest wstępnie kompilowany a potem interpretowany przez wirtualną maszynę Java zyskała dużą popularność. W latach 2009-2010 firma Oracle przejęła firmę Sun Microsystem. .Przez długi okres Java była najpopularniejszym językiem programowania jednak od pewnego czasu zainteresowanie tym językiem spadło. Fenomen ten jest spowodowany wzrostem alternatyw do tego języka przez co zastosowanie Javy bardzo się zawęziło. Jednakże dzięki rozwojowi wielu bibliotek, konceptów oraz frameworku Java jest nadal bardzo chętnie wykorzystywana do implementacji aplikacji internetowych. Symultanicznie ze frameworkiem Spring świetnie sprawdza się jako podstawa pod duże projekty. Jedynym konkurentem Javy w tej niszy jest język C#.

1. Spring

Spring jest to framework służący do implementacji aplikacji internetowych. Spring jest najbardziej rozwiniętą biblioteką OpenSource do tego rodzaju aplikacji. Posiada wiele modułów oraz współpracuje z wieloma innymi bibliotekami takimi jak Hibernate czy Lombok. Dzięki swojej prostocie ale także złożoności dostępnych narzędzi Spring stał się w dzisiejszych czasach biblioteką bez której nie można wyobrazić sobie języka Java. Powyższy wniosek potwierdzają statystyki firmy JetBrains(twórcy najpopularniejszego IDE dla Javy)[6]. Statystyki z lat 2018-2020 pokazują że aż 61% programistów używa tej biblioteki na drugim miejscu czyli 42% jest biblioteka składowa Springa a dopiero na trzecim miejscu znajduje się JSF z 9%. Dlatego popularnym stało się utożsamianie programistów Java z programistami aplikacji serwerowych posługujących się frameworkiem Spring.

1. Spring MVC

Spring MVC to narzędzie które dostarcza gotową implementację wzorca Model View Controler. Biblioteka jest oparta na adnotacjach które są refleksyjnie czytane przez bibliotekę po czym mapowane na punkty końcowe wystawiane przez aplikację. Podstawowo biblioteka była projektowana pod aplikacje typu Server-Side czyli aplikacje które są odpowiedzialne symultanicznie za prezentacje danych jak i ich serwowanie. Wraz z powstaniem architektury REST(Represenational State Transfer) biblioteka została rozszerzona o implementację właśnie tej architektury.

1. Spring Data

Spring Data to moduł biblioteki spring który jest odpowiedzialny za komunikację z bazą danych. Spring Data[7] we współpracy z biblioteką Hibernate jest jednym z najbardziej rozwiniętych modułów spring pod względem funkcjonalności. Umożliwia budowanie zapytań oraz mapowanie odpowiedzi do określonych przez nas projekcji narzucając jedynie obowiązek zaimplementowania interfejsów. Owe interfejsy są to zapytania zapisane w nazwie metody lub z odpowiednią adnotacją. Dzięki temu programista nie musi przejmować się implementacja zapytań w języku SQL. Bibliteka ma również wsparcie dla funkcjonalności udostępnionych przez Hibernate takich jak język HSQL czy Criteria API. Istnieje również możliwość napisania natywnego zapytania SQL które jest rzutowane na wskazaną przez nas projekcję.

1. Spring Context

Każdy większy projekt programistyczny potrzebuje oddzielnych modułów odpowiedzialnych tylko za budowanie aplikacji. Zgodnie z paradygmatami CleanCode wykutymi przez Roberta C. Martina odpowiedzialność budowania aplikacji powinna być oddzielona od implementacji właściwej logiki. Wspomniany proces budowania musi być dostosowany do złożoności projektu ponieważ zbyt skomplikowany proces budowania aplikacji zaciemni tylko faktyczny cel aplikacji. Kolejnym problemem jest ilość pracy która będzie powtarzana w każdym kolejnym projekcie co jest nazywane w śród programistów kodem „boiler plate”. Świetnym rozwiązaniem wspomnianych problemów jest właśnie Spring Context nazywana również kontenerem Springa. Spring context jest niczym innym jak świetnie przemyślaną i elastyczną implementacją DI(Dependency Injection). Dpendency Injection jest to wzorzec który zdejmuje z programisty odpowiedzialność tworzenia obiektów. Aplikacja sama wyszukuję zależności klasy po czym wstrzykuje dane zależności. Dzięki elastyczności świetnie nadaje się do tworzenia dużych jak i małych projektów. Sama implementacja Dependency Injection jest dopracowana na tyle że, aplikacja inteligentnie szuka implementacji typu zwracając uwagę na interfejsy czy typy generyczne. Dzięki takiemu dopracowaniu DI staję się potężnym narzędziem w rękach doświadczonego programisty.

1. Spring Boot

Uproszczenie konfiguracji biblioteki Spring wydanej w formacie Spring Boot było jednym z powodów tryumfu tej biblioteki nad JEE (Java Enterprise Edition). Jednym z bardzo rażących problemów JEE oraz pierwszych wersji Springa była żmudna konfiguracja i konieczność podłączenia kontenera serwletów. Jednak dzięki Spring Boot dla programistów Java zostało dostarczone narzędzie które automatycznie podłącza kontener serwletów, upraszcza tworzenie projektu i konfigurację. Uproszczenie konfiguracji polega na przeniesieniu konfiguracji tak zwanych Bean (czyli implementacji obiektów zarządzanych przez kontener Springa) z plików .xml do adnotacji. Tworzenie projektu zostało przeniesione do aplikacji Spring Initializer która w sposób graficzny pozwala określić strukturę oraz zależności obiektów. Kolejnym bardzo dużym plusem jest możliwość dostosowania konfiguracji do własnych potrzeb oraz tworzenia tak zwanych profili. Dzięki profilom możemy określić wariant konfiguracyjny dal aplikacji co znacząco ułatwia proces rozwoju aplikacji.

1. Bazy danych

Baza danych jest to najniższa warstwa aplikacji. Dobranie odpowiedniej bazy do projektu jest rzeczą kluczową. Poszczególne bazy danych różnią się logiką zapisu danych a także miejscem zapisu danych.

1. PostgreSQL

Postgre sql czyli najpopularniejszy otwarto źródłowy silnik bazy relacyjno-obiektowej. Wspomniany silnik bazy danych oferuje bardzo dużo usprawnień których próżno szukać w innych silnikach. Usprawnieniami są kolejno:

MVCC – Multiversion Concurrency Control jest to mechanizm który tworzy zdjęcie obecnego stanu bazy danych dla każdej transakcji dzięki czemu transakcje nie mogą na siebie wpłynąć. Większości eliminuję to potrzebę użycia blokad do czytania.

ACID – jest to zbiór własności które musi posiadać baza danych aby poprawnie przetworzyć transakcję. Pierwsza litera odpowiadam cesze niepodzielności co oznacza że każda transakcja wykona się w całości lub nie wykona się wcale. Następną zasadą jest spójność. Spójność co definiuje że dane po zakończeniu transakcji będą w zgodne z zasadami integralności w bazie danych. Kolejną litera oznacza izolację transakcji co oznacza tyle że dwie transakcje nie mogą na siebie wpływać. Ostatnią literą jest trwałość. Cecha ta gwarantuje że dane zapisane po transakcji będą dostępne w całym systemie oraz będą dostępne nawet po restarcie systemu.

1. H2

Baza danych sql która jest alternatywą dal PostgreSQL. Nie jest zapisywana w pamięci zewnętrznej tylko w pamięci operacyjnej. Przez tę cechę nie nadaję się na produkcję. Jednak dzięki profilom Spring możemy skonfigurować aby aplikacja używała tej bazy danych przy wariancie aplikacji do rozwoju. Dzięki temu programista nie jest spowolniony przez konieczność łączenia się z zewnętrzną bazą danych lub obciążaniem swojego sprzętu instancją bazy. W efekcie H2 jest świetną lekką i wybaczającą błędy alternatywą do produkcyjnej bazy danych.

1. Hibernate

Jeden z najbardziej rozwiniętych implementacji ORM(Object–relational mapping). Odpowiada za tworzenie tabel według zwykłych klas POJO(Plain Old Java Object) ubranych w odpowiednie adnotacje takie jak np. @Entity. Odpowiada także za mapowanie odpowiedzi bazy danych na właściwe obiekty. Dodatkowo udostępnia funkcjonalności takie jak język HSQL czyli mieszanka języka SQL z Javą przez co pisanie zapytań w tym języku jest bardziej intuicyjne. Kolejną bardzo przydatną funkcjonalnością jest Criteria API które umożliwia dynamicznie tworzenie zapytań za pomocą predykcji. Dzięki temu narzędziu połączenie z bazą danych oraz tworzenie zapytań jest w pełni automatyczne.

1. Zewnętrzne API

Dzięki rozpowszechnieniu standardów takich jak architektura REST API, model trójwarstwowy czy protokół http wiele aplikacji jest dzielona na poszczególne elementy składowe jak np. API. Dzięki temu część logiki większego systemu może być wykorzystany przez inny. Rynek jest pełen rozwiązań chmurowych które posiadają zbiory właśnie takich API i wydane są w modelu PaaS(Platform as a service) lub SaaS(Software as a service).

1. Google APIs

Google APIs czyli zbiór REST API udostępnione przez Google w ramach modelu SaaS(Software as a service)[8]. API są podzielone w zależności od funkcjonalności które spełniają. Dzięki podziałowi na funkcjonalności abonament płaci tylko za udostępnienie API którego używa. Kwota jaka płacą użytkownicy jest zależna od ilości zapytań wysłanych przez zarejestrowaną aplikację. Użyte zostały następujące API:

* Google Places API odpowiadające za udostępnianie informacji o obiektach na mapie.
* Google Geocoding API odpowiadające za konwersję lokalizacji podanej w formie długości i szerokości geograficznej na fizyczny adres.
* Google Geolocation API na podstawię wieży komórkowych oraz WiFi może określić lokalizację użytkownika w formie długości i szerokości geograficznej.

Dużą zaletą zaraz obok rzetelnych danych jest wsparcie dla programistów ze strony Google. Większość najpopularniejszych języków została obdarowana SDK(Standard development kit) który zdejmuje z programisty obowiązek ustalenia komunikacji z API oraz mapowania obiektów.

1. Infrastruktura paliw ciekłych w Polsce

Jest to otwarte API udostępnione przez Urząd Regulacji Energetyki[9]. Teoretycznie zasób ten powinien posiadać informacje o rodzajach paliw oraz lokalizacji przedsiębiorstw posiadających koncesję na sprzedaż paliw. Jednak w praktyce API posiada bardzo dużo źle zaprojektowanych elementów. Pierwszym co rzuca się w oczy jest użycie polskich nazw. Standardem jest używanie języka angielskiego ponieważ jest to język który jest używany przez wszystkie języki programowania. Kolejnym problemem są puste pola dla współrzędnych. Owe puste pola zwiększają objętość obiektu JSON nie dostarczając żadnych informacji. Dużym problemem jest także używanie wielu pól aby z wartościami 0 lub 1 aby przekazać informację czy dany rodzaj paliwa jest dostępny na stacji. Takie przekazywanie tych informacji powoduje duży nakład pracy programisty który chce skorzystać z tego zasobu. Jeżeli chcemy skorzystać z zewnętrznego API musimy zaimplementować komunikację pomiędzy naszą aplikacją a zasobem do którego się odwołujemy. Niestety przez ukazanie dostępności typów paliw w takiej formie wymuszamy na użytkowniku konieczność mapowania obiektu na inny typ danych. Największym problemem API co czyni go bezużytecznym jest niezgodność danych z faktycznym położeniem stacji. Adres podany w API zwykle nie jest faktycznym adresem stacji paliw zwykle jest to adres w pobliżu który wskazuje na blok mieszkalny lub łąkę. Stacja paliw o identycznej nazwie zwykle znajduje się w pobliżu tego adresu.

1. Java Faker

Biblioteka odpowiadająca za generowanie testowych danych na podstawie literatury, wydarzeń czy prawdziwych lokalizacji. Dzięki bardzo dużemu zasobowi danych świetnie sprawdza się jako narzędzie zapełniające bazę danych. Implementacja generatora danych opartego na tej bibliotece umożliwia sprawne testowanie oraz rozwój aplikacji.

1. MapStruct

Wzorzec DTO(Data transfer object) odpowiada za zstąpienie obiektu działającego w obrębie aplikacji na jego projekcje. Dzięki temu oddzielamy części aplikacji od siebie. Jest to szczególnie ważne wraz z użyciem ORM który mapuje faktyczne klasy na tabele w bazie danych. Niestety podejście takie wiąże się z dużą ilością powtarzalnego kodu typu „boiler plate”. Powtarzalny kod odpowiada za przelewanie danych z jednego typu na drugi. Mapstruct minimalizuje ten proces do trywialnej konfiguracji opartej o interfejsy oraz adnotacje[10]. Biblioteka MapStruct jest oparta o pluginy mavena i refleksje. Kompilator w trakcie działania może odczytać nazwy metod typy parametrów oraz adnotacje. Dzięki dostępności tych metadanych podczas kompilacji zostanie dodany kod który mapuję obiekty zgodnie z konfiguracją nałożoną na metodę w postaci parametrów oraz adnotacji. Dodatkowo biblioteka oferuje dużą ilość luk w których możemy umieścić fragment swojej implementacji mapowania. Kolejnym plusem jest wsparcie dla najpopularniejszych bibliotek Java takich jak na przykład Lombok lub Spring.

1. Lombok

Wielu przeciwników języka Java wytyka jego największą wadę czyli kod „boiler plate”. Kod tego typu jest niemal zawsze identyczny lecz potrzebny do poprawnego rozumienia programu przez interpreter oraz biblioteki. W celu spełnienia praktyk takich jak SOLID, DRY i KISS wymagane jest zaimplementowanie wielu wzorców projektowych. Implementacja wzorców takich jak np.. Builder także zalicza się do powtarzalnego kodu. Dodatkowym problem stanowi specyfikacja POJO(Plain Old Java Objkect) jest to obiekt który może stać się beanem (czyli obiektem zarządzanym przez kontekst Springa), która mówi że każdy obiekt powinien być enkapsulowany czyli w praktyce powinien posiadać wszystkie pola prywatne oraz publiczny getter i seter do każdego pola. w przypadku prostego obiektu typu DTO dostajemy klasę w której około 80% kodu jest dla nas zbędne. Lekarstwem na te problemy jest biblioteka Lombok oparta podobnie jak MapStruct na generowaniu kodu podczas kompilacji na podstawie adnotacji będących konfiguracją[11]. Całość kodu „boiler plate” z powodów wyżej wymienionych powyżej jest dla nas generowana podczas kompilacji***.***

1. JJWT

JWT czyli JSON Web Token jest popularnym mechanizmem używany do zabezpieczenia aplikacji serwerowych. Polega on na wygenerowaniu tokenu z danych użytkownika po wczesnej autentykacji. Generowanie tokeny jest oparte na szyfrowania danych o użytkowniku. Po wygenerowaniu tokenu jest on zwracany użytkownikowi. Jeśli użytkownik próbuję uzyskać dostęp do chronionego zasobu musi okazać prawidłowy token. Przykładowy token jest przedstawiony poniżej i składa się z następujących części :



*Rys 1. Struktura tokenu JWT*

Token jest podzielony na 3 elementy oddzielone kropkami są to szyfrowane obiekty typu JSON. Część A przechowuje informację o „alg” który został użyty do szyfrowania danych natomiast „typ” oznacza typ użytego tokenu w tym przypadku JWT[12]. Kolejną częścią jest B są to informacje przenoszone wewnątrz tokenu ustalane przez osobę projektującą system uwierzytelniania. Część C jest to zaszyfrowane hasło będące, ciągiem 256 bitów określony przez aplikację, który powinien być bezwzględnie utajniony w swojej czystej formie.

1. Swagger

Swagger jest to opisowy język interfejsów służący do portretowania RESTful API używając formatu JSON. Swagger jest używany w połączeniu z innymi otwartymi technologiami co czyni z niego jeden z najlepszych sposobów dokumentowania API dodatkowo ułatwiając jego rozwój i testowanie. Oczywiście wszystkie opisane funkcjonalności są realizowane w sposób maksymalnie automatyczny. Biblioteką służącą do generowania dokumentacji API w javie jest to biblioteka „springfox-boot-starter” która implementuje specyfikację OpenAPI z której korzysta Swagger.

1. Postman

Darmowe narzędzie postman służy do testowania API. Kluczową funkcjonalnością aplikacji jest wykonanie zapytań do serwera oraz prowadzenie ich historii. Wielkim atutem tego narzędzia jest także interfejs który pozwala na ustawienie w sposób graficzny parametry zapytania. Dodatkowo znajduję się też możliwość implementacji własnych testów w języku JavaScript. Narzędzie to ułatwia pracę programiście ale także osobie nie posiadającej dużej wiedzy technicznej z dziedziny protokołów sieciowych.

1. Intelij IDEA

Intelij IDEA czyli najpopularniejsze środowisko programistyczne dla języka Java od firmy JetBrains. IDE posiada szeroką gamę funkcjonalności przydatnych przy tworzeniu aplikacji internetowej. Wsparcie wielu języków uwalnia programistę od szukania dodatkowych rozwiązań w celu pracy nad interfejsem użytkownika. Możliwość połączenia z bazą danych jest przydatna ponieważ nie musimy używać dedykowanych narzędzi. Kolejną zaletą tego edytora tekstu jest ilość podpowiedzi które udziela użytkownikowi analizując napisany już kod. Intelij przyśpiesza także pracę programisty oferując skróty do poszczególnych funkcjonalności takich jak refaktoryzacja kodu, generowanie czy analiza kodu ale także poruszanie się po edytorze. Dzięki tak wysokiej popularności IDE jest zintegrowane z takimi systemami jak git, github oraz popularnymi bibliotekami.

1. Google Maps JavaScript API

Popularny serwis udostępniający mapę świata od firmy Google udostępnia także API oraz bibliotekę służącą do umieszczenia mapy na stronie internetowej oraz dodaniu do niej własnej funkcjonalności. Mapa którą umieszczamy na stronie jest responsywna a także konfigurowalna. Dzięki konfiguracji możemy określić jakie funkcjonalności mają być dostępne dla klienta końcowego. Językiem który wykorzystuje biblioteka jest JavaScript. Ów biblioteka jest klientem który łączy się asynchronicznie z Google Maps API[13] ładując kolejne elementy mapy oraz zamieszczone na niej informację.

1. Angular

Angular czyli otwarto źródłowy framework oparty na języku Type-Script opracowany przez firmę Google służący do pisania aplikacji internetowych typu SPA(Single Page Application). Umożliwia pracę ze stroną internetową dzieląc ją na poszczególne małe komponenty. Podstawowy komponent składa się na trzy pliki .ts implementujący logikę komponentu, .css agregujący style elementu oraz .html posiadający informacje o jego strukturze. Dzięki takiemu podziałowi jesteśmy wstanie utrzymać rozdzielenie języków pomiędzy plikami. Jest to bardzo dobra praktyka która jest często poruszana przez Roberta C. Martina. Kolejnym aspektem Angulara jest jego modularność. Modularność pozwala na podział elementów składowych aplikację w większe podgrupy dzięki czemu możemy poprawić wydajność aplikacji. Bardzo dużą zaletą Angulara jest silnik Dependency Injcection. Kolejną zletą Angulara którą ciężko znaleźć u konkurencyjnych bibliotek to spejcjalne zależności Angulara. Są to biblioteki przeznaczone tylko dla tego frameworku. Dzięki temu biblioteki te są znacznie lepiej przygotowane pod współprace z framworkiem. Obecnie Angular jest najbardziej złożoną biblioteką do tworzenia SPA.

1. Angular Material

Angular material jest to zależność frameworka Angular która udostępnia komponenty zgodne ze specyfikacją Material UI. Material UI jest to specyfikacja która zawiera opis docelowego wyglądu interfejsu użytkownika. Każdy z komponentów użytych przez bibliotekę jest konfigurowalny oraz można go dostosować do wyglądy i kolorystki aplikacji.

1. RXJS

RXJS jest to biblioteka JavaScript służąca do przetwarzania zdarzeń używając sekwencji obserwatorów. Dostarcza główną klasę Observable oraz szereg operatorów służących do przetwarzania tej klasy w sposób asynchroniczny. RXJS umożliwia programowanie reaktywne w którym każde nadchodzące zdarzenie jest przetwarzane przez ciąg operatorów. Jeżeli chcielibyśmy przetwarzać dane podane przez użytkownika piszącego na klawiaturze w podejściu imperatywnym musielibyśmy ustawić timer np. na 50ms i sprawdzać zmiany. Natomiast podejście reaktywne znacząco uproszcza sprawę ponieważ określamy jedynie ciąg zdarzeń które mają się wykonać po otrzymaniu danego typu zdarzenia. Głównymi cechami RXJS są :

* Obiekt Observable reprezentujący wywoływalną kolekcje przyszłych wartości lub zdarzeń
* Observer czyli ciąg zdarzeń mających się wykonać po wywołaniu obiektu Observable
* Subscirption który jest abstrakcją wywołania obiektu Observable

1. Sass

Sass jest to prekompilator języka CSS. Język CSS odpowiada za określenie styli strony internetowej czyli jej wyglądu. Jednakże język CSS jest bardzo ograniczony oraz powtarzalny. Przez tak duża powtarzalność kodu oraz brak mechanizmów na zmniejszenie jego objętości utrzymanie dużej bazy kodu staję się trudne. Rozwiązaniem tego problemu jest prekompilator CSS czyli Sass. Jest to język który jest kompilowany do języka CSS. Sass znacznie wzbogaca język CSS po przez takie narzędzia jak mixins, zmienne czy klasy. Dzięki tym funkcjonalnością możemy znacznie zmniejszyć powtarzalność oraz objętość kodu.

1. GIT/GITHUB

Git jest to system kontroli wersji który stał się standardem w dzisiejszych czasach. Został stworzony przez Linusa Torvaldsa w 2005 roku aby umożliwić pracę nad jądrem systemu operacyjnego Linux. Systemy kontroli wersji znacznie wpłynęły na rozwój programowania oraz umożliwiły współpracę programistów nawet nie mających ze sobą bezpośredniego kontaktu. Obecnie niemal każdy komercyjny projekt korzysta z systemu kontroli wersji. Natomiast Git stał się narzędziem obowiązkowym w warsztacie każdego programisty. Dzięki temu że Git jest najpopularniejszym systemem kontroli wersji możemy znaleźć wiele usług udostępniających repozytoria takich jak na przykład GitHub. Serwis ten umożliwia korzystanie ze zdalnego repozytorium git a także udostępnia szereg usprawnień.

1. HEROKU

Heroku czyli otwarta platforma umożliwiająca wdrożenie aplikacji serwerowej. Platforma ta jest oparta o konteneryzację umieszczającą aplikację w chmurze. Usługa pozwala na łatwe wdrożenie, skalowanie oraz rozwijanie aplikacji internetowych. Programista korzystający z tej jest zdjęty z obowiązku utrzymania infrastruktury, serwerów czy zużycia energii. Heroku oferuję także szereg darmowych usług takich jak np. baza danych. Dodatkowo jesteśmy w stanie wykupić dodatkowe usługi jednak te wiążą się z dodatkową opłatą. Opłaty te są wyliczane na podstawie wykorzystanych zasobów oraz liczby zapytań do serwera. Platforma jest także w pełni zintegrowana z systemem Git dzięki czemu przy każdej zmianie głównej gałęzi repozytorium nowa wersja aplikacji jest wdrażana.

# Wykorzystane praktyki

### Clean Code

Definicja słowa „CleanCode” została zdefiniowana przez Roberta C. Martina w książce o tym samym tytule. Programiści używali tego sformułowania jako pochwały dla kodu przejrzystego, łatwo modyfikowalnego oraz skalowalnego. Jednakże każdy programista miał swoją własną definicję tej frazy. Natomiast w większości przypadków definicje te posiadały cechy wspólne. Z punktu widzenia osoby która przejmuje się celem oraz przeznaczeniem programu może powiedzieć ona że liczy się prawidłowość działania programu a nie stan jego kodu ,ponieważ użytkownik końcowy nie widzi kodu źródłowego aplikacji. Jednak programy komputerowe nie są i nie mogą być produkowane jak produkty z linii produkcyjnej. Zwykle program wypuszczony na produkcję musi być utrzymywany. Dodatkowym problemem jest koszt wytworzenia oprogramowania. Łatwo sobie wyobrazić że łatwiej jest rozszerzyć istniejący system niż tworzyć nowy od początku. Jeżeli pracę nad kodem wydłużają się z powodu kodu złej jakości to zaczynamy o mówić problemie długu technicznego. Dług techniczny jest to sformułowanie dotyczące czasu pracy programisty który musi być poświęcony aby doprowadzić kod do porządku. Jednak największym problemem jest to że wysokość długu technicznego jest odwrotnie proporcjonalna do wydajności pracy programisty. Dzieję się tak ponieważ wprowadzenie zmiany do kodu złej jakości jest poprzedzone zrozumieniem zawiłego programu oraz prowadzi do powstania ogromnej ilości błędów. Podsumowując kod złej jakości jest błędogenny, potrafi doprowadzić programistów do bólu głowy oraz przedsiębiorstwo do bankructwa.

### Dobre praktyki

* DRY czyli „Don’t Repeat Yourself” oznacza że, nie wskazane jest posiadanie podobnych linii kodu lub całych bloków różniących się jedynie poszczególnymi wartościami.
* YAGNI czyli *„You aren’t gonna need it”* trzeba to rozumieć że kod który piszemy ma spełniać określone wymagania i nic ponad to. Jeżeli mamy do spełnienia funkcjonalność to implementujemy ją zachowując jakość kodu ale nie piszemy funkcji, metod bądź klas które nie są potrzebne teraz ale mogą się przydać.
* KISS czyli „*Keep It Simple Stupid*”. Oznacza to po prostu tyle że kod pisany przez nas ma być maksymalnie najprostszy. Architektura oraz złożoność kodu ma być adekwatna do rozwiązywanego problemu. Programista nie powinien szukać oraz implementować wyszukanych rozwiązań do stosunkowo małych problemów.
* TDA czyli *„Tell Don’t Ask”.* Zasada dotyczy obowiązków pomiędzy naszymi klasami. Obiekt z którego korzysta metoda powinien być pytano o wykonanie metody lub operację która powinna dziać się samoczynnie. Nie powinniśmy wykonywać operacji bazując na stanie obiektu z którego korzystamy. Spełnianie tej zasady pozwala na zmniejszenie zależności pomiędzy klasami.

### Wzorce projektowe

Wzorzec projektowy czyli uniwersalne rozwiązanie, sprawdzone w praktyce, rozwiązujące powtarzalny problem projektowy[14]. Stanowi sam przepis na rozwiązanie problemu a nie jego implementację. Dzięki wykorzystaniu wzorców projektowych możemy w sprawdzony sposób zadbać o jakość naszego kodu. Wzorce projektowe dotyczą programowania obiektowego i dzielimy je na 3 grupy:

* Wzorce kreacyjne wprowadzają szerokie mechanizmy tworzenia obiektów które znacząco zwiększają elastyczność i uniwersalność istniejącego kodu.
* Wzorce strukturalne tłumaczą jak zebrać większe grupy klas oraz obiektów w większe struktury zachowując elastyczność i efektywność ich użycia.
* Wzorce behawioralne koncentrują się wokół algorytmów oraz podziałem obowiązków pomiędzy obiektami.

Wzorce projektowe użyte w projekcie:

* Metoda fabrykująca – Wzorzec kreacyjny który dostarcza interfejs do tworzenia obiektów jako nadklasa jednak dający możliwość tworzenia obiektów podklas.
* Budowniczy - Umożliwia tworzenie obiektu określając jego wszystkie parametry bez potrzeby Implementacji konstruktora posiadającego wiele parametrów.
* Adapter – Potrafi zmusić do działania dwa pozornie niewspółgrające interfejsy.
* Komenda – Behawioralny wzorzec który zmienia zapytanie na komendę trzymaną przez pojedynczy obiekt. Dzięki temu możemy parametryzować metody po przez różne zapytania, opóźniać ich wykonanie lub kolejkować je.
* Obserwator – Wzorzec behawioralny który pozwala definiuje mechanizm subskrypcji oraz powiadomień. Kiedy obiekt obserwowany zmienia stan lub tworzy zdarzenie subskrybenci czyli obserwatorzy zostają o tym poinformowani.
* Strategia – Dzięki strategii możemy zdefiniować obiekt który używa określonej rodziny algorytmu po czym zaimplementować każdy z tych algorytmów jako osobną klasę.

### Paradygmaty programowania

Istnieje kilka paradygmatów programowania. Każdy z nich określa w jaki sposób programista patrzy na przepływ sterowania oraz wykonanie programu.

### Programowanie funkcyjne

Metodyka będąca odmianą programowania deklaratywnego. Programowanie funkcjonalne[15] traktuje funkcję jako podstawą jednostkę budującą program. Najważniejszym pojęciem istniejącym w programowaniu funkcyjnym jest czysta funkcja. Czysta funkcja z angielskiego „Pure Function” jest funkcją która dla podanych argumentów zwraca zawsze taki sam wynik bez efektów ubocznych. Efektem ubocznym może być zmiana wartości istniejącej poza funkcją a także użycie strumienia wyjściowego. Dążenie do wydzielania czystych funkcji prowadzi do zmniejszenia ilości błędów w kodzie a także ułatwia debugowanie oraz testowanie programu. Programowania funkcyjne posiada także szereg wzorców z czego najważniejszy są:

* Highr Order Funcion – funkcja przyjmująca wskaźnik na inną funkcję jako argument wejściowy. Dzięki temu wzorcowi możemy zwiększyć ilość ponownego użycia naszego kodu.
* Closures – funkcja w której zwracamy jako argument inną funkcję podają do niej wartość jako argument. Podana wartość jest zapieczętowana w zasięgu funkcji.
* Decorator – funkcja podobnie jak High Order Function przyjmuje wskaźnik na inna funkcje. Jednak jako argument wyjściowy zwraca nowo powstałą funkcję która dynamicznie poszerzania odpowiedzialność funkcji która została przekazana jako argument wejściowy.

### Programowanie procedularne

Podstawowa odmiana programowania imperatywnego. Opiera się na serii następujących po sobie inwokacji procedur. Jest podzielona na serię oddzielnych kroków. Procedura może być zarówno funkcją jak i metoda. Jednak może posiadać efekt uboczy oraz nie ma określonych zasad jeśli chodzi o argumenty wejściowe i wyjściowe.

### Programowanie Obiektowe

Paradygmat programowania obiektowego jest oparte na koncepcie obiektów. Obiekt w kontekście programowania jest to zbiór danych w przedstawionych w formie pól a kod w formie procedur zagnieżdżonych w obrębie wspomnianych pól. Wspomniane pola mogą być użyte w ramach procedury. Większość języków obiektowych korzysta z klas. Klasa jest to definicja typu obiektu. Natomiast obiekt jest instancją klasy. Podstawowymi pojęciami programowania zorientowanego obiektowo są:

* Abstrakcja – Obiekty w programie są pracownikami którzy mogą wykonywać prace przydzieloną ich pracę bez ujawniania szczegółów implementacyjnych.
* Hermetyzacja – Inaczej enkapsulacja polega na ukrywaniu implementacji obiektu przed resztą aplikacji. Klasa udostępnia swego rodzaju interfejs reszcie aplikacji.
* Polimorfizm – Do referencji obiektów mogą być przypisywane obiekty różnego typu. Wywołanie procedury dla poszczególnego obiektu wywoła procedurę adekwatną dla tego obiektu.
* Dziedziczenie – Umożliwia porządkowanie definicji obiektów oraz wydzielanie dla nich części wspólnych.
* Bezpośrednio z programowaniem obiektowym wiąże się zbiór praktyk określonych przez skrót SOLID:
* S – Single-responsibility principle – Każda klasa w aplikacji powinna posiadać tylko jedną odpowiedzialność.
* – Open-closed principle – Każda klasa powinna być otwarta na rozszerzenia ale zamknięta na zmiany. W każdym momencie rozwoju aplikacji powinien istnieć tylko jeden powód do zmiany istniejącej klasy. Rozszerzalność klasy powinna być realizowana przez dziedziczenie a nie zmianę już istniejącego kodu.
* L – Liskov substiton principle – Instancja klasy nadrzędnej powinna być wymieniana z instancją klasy podrzędnej bez potrzeby zmiany logiki programu.
* I – Interface segregation principle – Wiele interfejsów jest lepsze niż jeden ogólny.
* D – Dependency inversion principle – Interfejs powinien bazować na abstrakcji obiektów a nie na konkretnej implementacji.

### Łączenie paradygmatów programowania

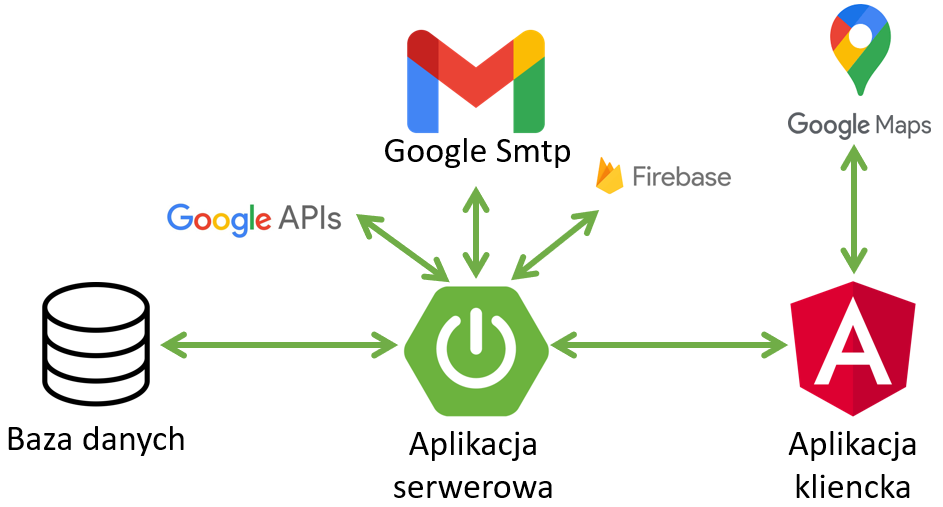
Języki programowania wspierają wiele paradygmatów programowania. Łączenie paradygmatów w obrębie jednej aplikacji jest nieuniknione. Ponieważ każdy paradygmat wspiera rozwiązanie określnej grupy zadań. Jednak niepoprawne łączenie paradygmatów programowania może prowadzić do powstania długu technicznego. Wymogiem jest wyraźne oddzielenie fragmentów kodu napisanego używając różnych paradygmatów.

# Architektura aplikacji

Aplikacja „PolishStation” składa się z trzech elementów. Kolejne elementy są to baza danych, aplikacja serwerowa oraz aplikacja kliencka. Aplikacja do poprawnego działania potrzebuje również dostępu do zewnętrznych usług takich jak Google Api oraz Firebase. Częścią aplikacji którą widzi użytkownik jest to interfejs zaimplementowany przy użyciu frameworka Angular. Jest to aplikacja SPA(Single Page Application) działająca w przeglądarce. Dodatkowo jest rozszerzona o PWA które ma na celu iluzje aplikacji natywnej. Dodatkowo PWA pozwala na wykorzystanie takich funkcjonalności jak PushNotifications. Aplikacja kliencka do poprawnego działania potrzebuje połączenia z Google Map Api w celu wyświetlenia mapy. Część interfejsu użytkownika korzysta bezpośrednio z części serwerowej. Główną częścią aplikacji jest część serwerowa zrealizowana przy użyciu frameworka Spring Boot. Dodatkowo jest także zintegrowana z zewnętrznymi api takimi jak Google API czy Firebase[16]. Zewnętrzne usługi umożliwiają zrealizowanie następujących funkcjonalności:

* zmiana adresu fizycznego na położenie geograficzne
* zmiana położenia geograficznego na adres fizyczny
* wysłanie wiadomości email
* wysyłanie notyfikacji
* wyszukiwanie stacji względem położenia geograficznego

Część serwerowa jest także odpowiedzialna za wszystkie operacje dziejące się w obrębie aplikacji. Ostatnim elementem jest baza danych z której bezpośrednio korzysta część serwerowa. W celu poprawnego działania aplikacji wymagane jest uruchomienia wszystkich części.



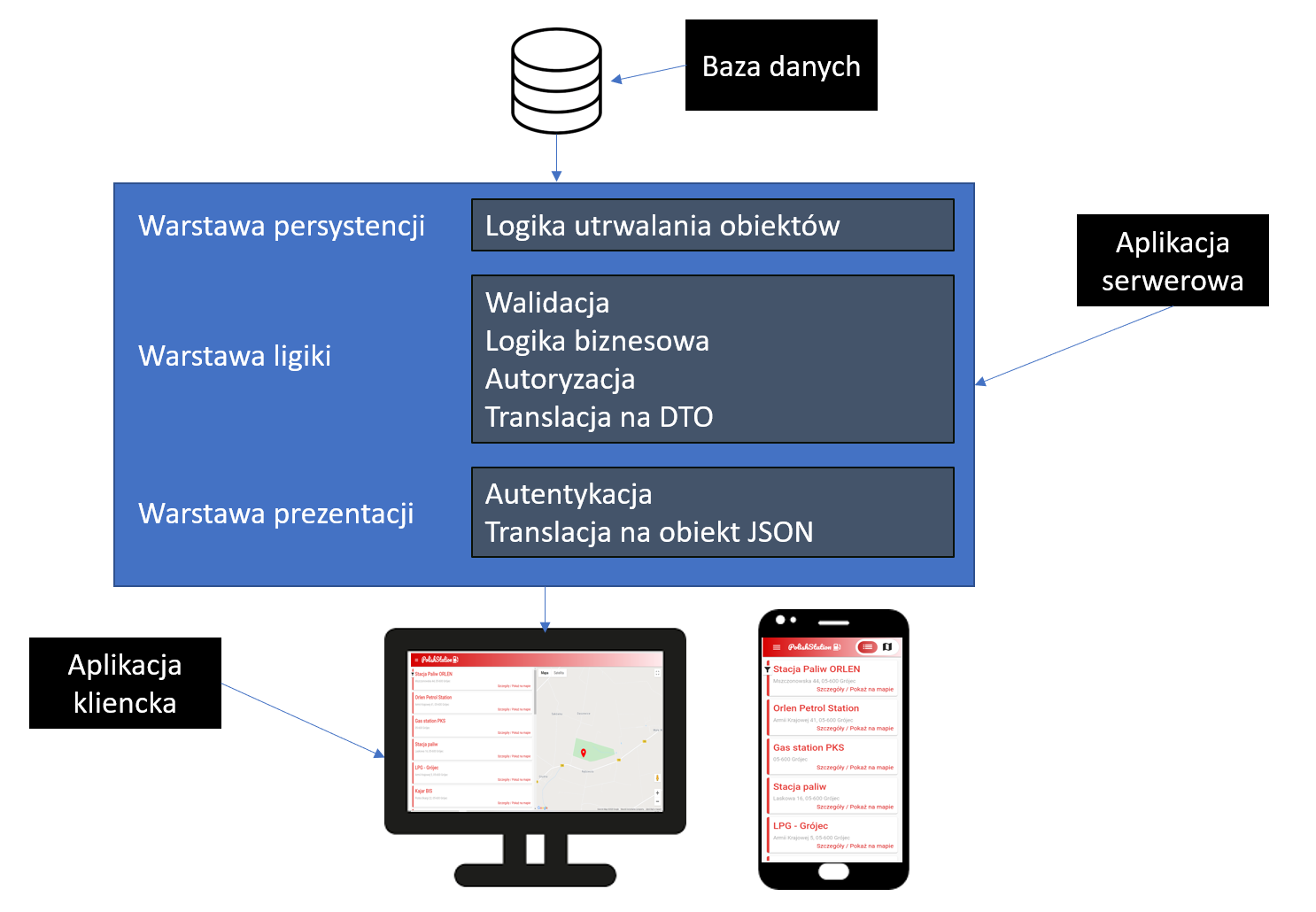
Rys.2 Uproszczony schemat architektury

#### Architektura monolityczna

Najbardziej skomplikowaną a także główną częścią aplikacji jest serwer. Aplikacja tego typu może być realizowana w oparciu o różne typy architektury. Najbardziej powszechnymi wzorcami architektonicznymi są kolejno:

* Mikroserwisy
* Architektura trójwarstwowa
* Architektura heksagonalna

Każda z wymienionych architektur ma swoje zalety. Dwie z nich czyli architektura heksagonalna oraz trójwarstwowa[17] to architektury monolityczne. Architektura monolityczna oznacza że, Wybór architektury musi być adekwatny do wymagań projektu ale także do metodologii pracy jak i systematyki pracy. Architektura mikroserwisów opiera się na podzieleniu odpowiedzialności systemu na mniejsze elementy. Każdy z mniejszych elementów jest oddzielną aplikacją która do poprawnego działania wymaga kontaktu z innymi elementami. Główną zaletą tego podejścia jest możliwość podziału dużego zespołu programistów na mniejsze zespoły które nie muszą mieć ze sobą bezpośredniego kontaktu. Architektura ta świetnie nadaje się do dużych komercyjnych projektów które są rozwijane latami. Jednak jest to bardzo złożone podejście i dla mniejszych aplikacji korzyści wynikające z jej implementacji są niewymierne w stosunku do stopnia skomplikowania. Kolejną jest architektura trójwarstwowa. Cechuje się prostotą oraz małym narzutem pracy potrzebnej do jej zaimplementowania. Dodatkowo jest to podejście domyślne w większości frameworków. Umożliwia także obsługę operacji dziejących się w aplikacji zgodnie z wzorce CRUD(Create, Read, Update, Delete). Wzorzec ten zakłada że, każdy obiekt istniejąc w aplikacji udostępnia cztery podstawowe typy operacji czyli kolejno tworzenie, edycja, odczyt i usunięcie. Wadą takiego podejścia jest niska skalowalność systemu opartego o taką architekturę. Ostatnią architekturą jest heksagonalna. Główną jej cechą jest oddzielenie logiki aplikacji od technologii. Dzięki czemu zmiana technologii a także koszt przepisania systemu jest maksymalnie obniżony. Stopień złożoności jest mniejszy niż w przypadku architektury mikroserwisowej przy zachowaniu wysokiej skalowalności. Cechą która może zniechęcić programistów jest narzut podejścia do obsługi zapytań w oparciu o wzorzec CQRS(Command and Query Responsibility Segregation). Wzorzec ten zakłada że, wszystkie zapytania dzieją ce się w aplikacji są podzielone na dwie grupy czyli querry i command. Querry odpowiada za operacje związane z odczytywaniem danych. Natomiast command odpowiada za operację modyfikujące dane. Wzorzec ten jest trudniejszy do zaimplementowania niż CRUD. Kolejnym problemem jest niepotrzebny narzut pracy związany z implementacja różnych operacji. W przypadku CRUD większość operacji jest obsługiwana przez jedną z czterech podstawowych operacji. CRUD jest dużo lepszym wyborem w przypadku małej ilości operacji dal każdego z elementów lub gdy ich złożoność jest stosunkowo niewielka. Podsumowując z racji że, aplikacja jest stworzona przez jednego programistę oraz ilość operacji dla poszczególnych obiektów jest nie duża najbardziej optymalnym wyborem jest architektura trójwarstwowa.



Rys.3 Uproszczony schemat architektury aplikacji serwerowej

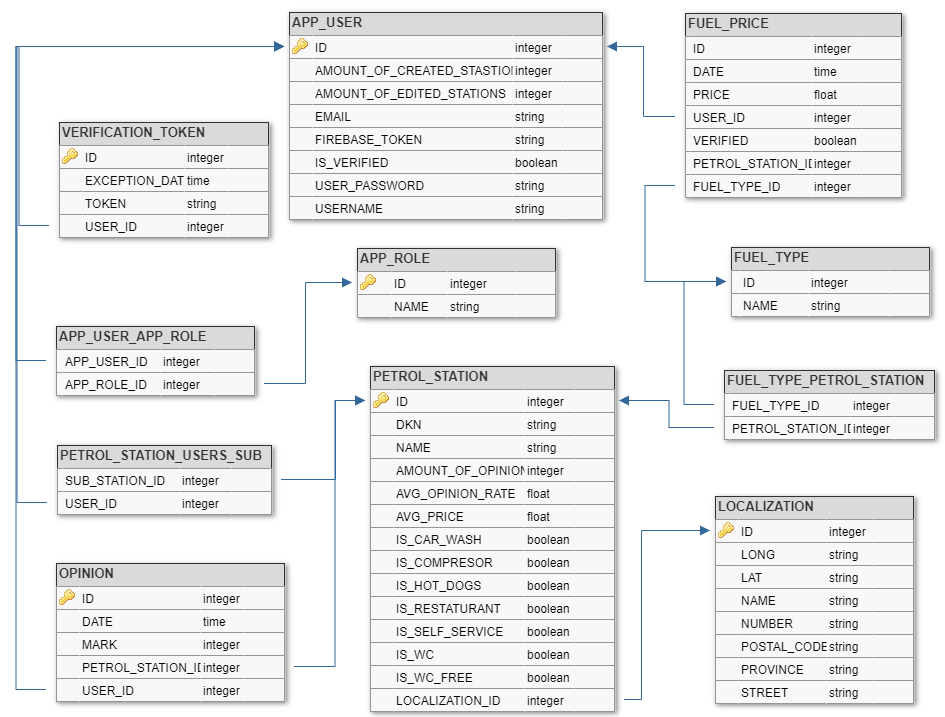
Aplikacja jest podzielona na trzy warstwy. Warstwę perzystencji, logiki i prezentacji. Warstwa presystencji jest odpowiedzialna za utrwalanie obiektów w bazie danych oraz za odczytywanie danych z tabeli. Dodatkowo znajdują się w niej także zaimplementowane metody służące do wyszukiwana rekordów oraz ich odwzorowania na obiekty. Następną warstwą jest warstwa logiki. Zawiera ona całość operacji które udostępnia aplikacja. Implementuje ona biznesową logikę aplikacji. Dodatkowo spełnia wymagania niefunkcjonalne takie jak walidacja poszczególnych obiektów pochodzących od użytkownika czy autoryzacja. Warstwa prezentacji obsługuję translację obiektów JSON na obiekty Java oraz na odwrót. Dodatkowo jest na nią nałożony mechanizm weryfikujący użytkownika. Jedną z głównych zasad tej architektury jest to że warstwa niższa nie korzysta z warstwy wyższej. Natomiast warstwa wyższa może korzystać tylko z warstwy o jeden poziom niższej. Warstwą najniższą jest warstwa persystancji która jako jedyna ma dostęp do bazy danych. Warstwa prezentacji czyli najwyższa korzysta z warstwy logiki oraz spełnia rolę fasady między użytkownikiem a resztą systemu. Obiekty które istnieją w obrębie aplikacji różnią się postacią pomiędzy warstwami. Warstwa presystancji oraz logiki ma dostęp do obiektów odzwierciedlających tabele z bazy danych. Natomiast warstwa prezentacji a tym samym aplikacja kliencka widzi obiekty DTO(Data transfer object). Rozdzielenie implementacji obiektów widzianych przez użytkownika a tych na których operuje aplikacja znacznie zwiększa skalowalność takiej aplikacji. Aplikacja serwerowa korzysta także z wzorca REST API. Który opisuje w jaki sposób mają być udostępnione zasoby. Specyfikacja REST określa następujące zasady tworzenia punktów końcowych aplikacji. Każdy adres udostępniający zasób aplikacji ma się zaczynać od nazwy udostępnionego zasobu. Czasowniki protokołu http odpowiadają kolejno różnym operacją dziejącym się na zasobie. W celu określenia pojedynczego zasobu podajemy identyfikator obiektu jako kolejny człon adresu. Jeżeli chcemy odwołać się do pola znajdującego się w obiekcie podajemy go za identyfikatorem. Przykładowo adres „GET /station/1 „ odpowiada za pobranie stacji o identyfikatorze 1. Natomiast „PATCH /user/2” spowoduje aktualizację użytkownika o identyfikatorze 2. Skorzystanie z wzorca REST pozawala na uniezależnienie aplikacji serwerowej od aplikacji klienckiej. Dodatkowo pozwala na ponowne użycie aplikacji serwerowej. Ponieważ rozszerzenie aplikacji o kolejny interfejs użytkownika w postaci natywnej aplikacji natywnej miałoby mniejszy narzut niż tworzenie całego systemu od nowa. Powodem mniejszego narzutu jest możliwość skorzystania z już istniejącego REST API które jest kompatybilne z urządzeniami obsługującymi protokół http. Dodatkowo z powodu korzystania kilku aplikacji z jednej aplikacji serwerowej nie ma potrzeby synchronizacji danych pomiędzy nimi. Kolejną zaletą użycia tego wzorca jest jego popularność. Praktycznie każdy większy system może być zintegrowany z naszą aplikacja co daje możliwość wykorzystania API przez inny podmiot.

#### Baza danych

Baza danych jest jednym z kluczowych elementów w systemie. Pozwala ona na zachowanie danych podczas pracy aplikacji jak i poza jej pracą. Może też udostępniać dane kilku instancją aplikacji. Baza danych użyta do projektu to PostgreSQL. Jest to baza relacyjna. Z czego wynika wymóg określenia schematu bazy danych. Schemat bazy danych odzwierciedla faktyczne obiekty istniejące w systemie za pomocą typów udostępnianych przez bazę poniżej znajduje się schemat bazy danych:

Podczas projektowania bazy danych ważną rzeczą jest normalizacja schematu[18]. Polega ona na dążeniu do spełnienia trzech postaci normalnych:

* Dane muszą być atomowe.
* Każda tabela ma pokazywać jedynie dane określonej klasy obiektów
* Kolumna nienależąca do klucza głównego nie zależy też od innej kolumny

Im więcej postaci zostanie spełnione tym bardziej rozszerzalna jest baza danych.

Rys.4 Schemat bazy danych aplikacji

#### Aplikacja serwerowa

Aplikacja serwerowa jest to REST API. Aplikacja udostępnia zbiór punktów końcowych które odnoszą się do konkretnych operacji na zasobach. Punkt końcowy obsługuję protokół http a jego działanie jest uzależnione od parametrów zapytania określonego przez użytkownika. Wydzielony podzbiór punktów końcowych jest chroniony i aby się do niego odwołać należy okazać token autoryzacyjny. Odpowiedzi są w postaci obiektów JSON. Całość specyfikacji api która demonstruje dostępne punkty końcowe jest także udostępnianie przez aplikację w formie graficznej dzięki technologii Swagger. Niektóre z punktów końcowych odpowiadają jedynie za zmianę zasobu. Funkcjonalności realizowane przez aplikację serwerową:

#### Wyszukiwanie stacji

Wyszukiwanie stacji jest kluczową funkcjonalnością aplikacji za którą stoi szerszy proces. Punktem końcowym spełniającym te wymaganie jest

GET /petrolstation?lat=51.8655118&long=20.8666034&fuelType=95&avgOpinion=2.5&maxDistance=10&sort=price&facilities=isWC,isCompressor

Opis parametrów

|  |  |  |  |
| --- | --- | --- | --- |
| Nazwa | Typ | Czy wymagany? | Opis |
| lat | double | TAK | Szerokość geograficzna lokalizacji dla której wyszukiwane są stacje |
| long | double | TAK | Długość geograficzna lokalizacji dla której wyszukiwane są stacje |
| maxDistance | int | TAK | Promień odległości w kilometrach według którego są wyszukiwane stacje. Maksymalnie 50 kilometrów |
| fuelType | String | NIE | Nazwa typu paliwa który jest wyszukiwany. Jeśli nie podany akceptowane są wszystkie typy. |
| avgOpinion | double | NIE | Minimalna opinia od której akceptowane są stacje. |
| facilities | String[] | NIE | Ciąg nazw oznaczających udogodnienia. |
| sort | string | nie | Nazwa pola po którym sortowane są znalezione stacje |

Kiedy użytkownik odpytuje punkt końcowy o dane w tym momencie prócz wyszukiwania na w aplikacji dzieje się seria zdarzeń. Po odebraniu zapytani i wyłuskaniu jego parametrów wysyłane jest zapytanie do Google API w celu wyszukania stacji w okolicy. W celu wyszukania stacji wymagane jest podanie długości, szerokości geograficznej oraz odległości od wskazanego punktu. Google API narzuca maksymalną odległość wyszukiwania czyli 50km. Kiedy serwer Google odpowie miejsca dane o miejscach przez niego znalezionych są konwertowane do instancji klasy reprezentującej stacje. Wyszukiwanie stacji po przez zewnętrzną usługę Google jest bardzo ograniczone. Brakuje tam danych o typach paliw, cenie oraz dodatkowych udogodnienia. Jednak API to świetnie sprawdza się do zapełnienie bazy danych faktycznie istniejącymi punktami. Po pobraniu punktów z Google wyszukiwane są stacje z bazy danych aplikacji. Podczas tego wyszukiwania nie uwzględniamy jeszcze kryteriów użytkownika. Kiedy otrzymamy wyniki wyszukane względem odległości i położenia z bazy danych porównujemy je z wynikami z Google. Jeżeli w bazie danych brakowało jakiegoś rekordu to zostaje on dodany. W sytuacji kiedy dane dla pojedynczej stacji różnią to akceptowane są dane z Google. Dzięki takiemu rozwiązaniu możemy w sposób leniwy zapełniać bazę danych informacjami o nowych stacjach. Daje to kilka korzyści. Po pierwsze aplikacja już na starcie jest bogata w dane. Mimo pobranych już stacji nadal istnieje możliwość ich aktualizacji lub dodanie nowych. Jeśli aplikacja będzie używana intensywniej przez dany obszar to więcej danych dla tego obszaru będzie dostępne. Dodatkowo rozwiązanie to zmniejsza koszty. Usługi Google są płatne a cena zależy od ilości zapytań. Agregując dane z serwisu gogle w sposób leniwy nagłych i dużych kosztów. Zoptymalizowany została także start aplikacji ponieważ dokładanie danych o stacjach jest sprawniejsze niż odczytywanie ich przy starcie aplikacji. Kolejnym krokiem jest wyszukiwanie faktycznych wyników dla użytkownika. Kiedy dane są już wyszukane dodawane są metadane takie jak ostatnio zaktualizowane ceny.

#### Dodawanie ceny

Bardzo ważną funkcjonalnością z punktu widzenia użytkownika jest możliwość dodania nowej ceny paliwa. Jednak umożliwienie dodania ceny bez wstępnej walidacji może doprowadzić do wstawiania fałszywych danych przez złośliwych użytkowników. W celu uchronienia się przed tego typu akcjami zaimplementowana dodatkowa walidacja. Kiedy serwer dostaję zapytanie mające na celu aktualizację ceny na stacji ta przechodzi przez proces weryfikacji. Na początku wyszukujemy ostanie ceny dla danego typu paliwa i stacji. Następnie porównujemy kandydata na nową cenę z ostatnią. Jeżeli różnica ceny wynosi więcej niż maksymalny procent różnicy ustalony w konfiguracji (domyślnie 20%) to cena ta jest uważana za fałszywą i odrzucana. Jeżeli różnica w cenie jest mniejsza niż wartość ustalona to cena zostaje dodana jako niezweryfikowana. Cena która nie jest zweryfikowana to taka która spełnia większość kryteriów jednak jest podana tylko przez jednego użytkownika. Jeżeli inny użytkownik poda taką samą cenę to cena ta jest uznawana za zweryfikowaną i uznawana za nową cenę na stacji. Taki algorytm pozwala na uniknięcie wprowadzania fałszywych danych przez użytkowników.

#### Statystyki

API udostępnia również trzy punkty końcowe służące do pobierania statystyk. Pierwszy punkt końcowy udostępnia rankingi stacji. Rankingi są to listy 10 stacji które mają najlepszą średnią ocenę oraz listy stacji z najmniejszymi cenami danego typu paliwa.

Drugi punkt końcowy udostępnia statystyki wskazanej stacji paliw dla podanego przedziału czasu.

GET / /statistics/ petrol-station?dateFrom={dateFrom}&dateTo={dateTo}&petrolStationId={id}

Opis parametrów

|  |  |  |  |
| --- | --- | --- | --- |
| Nazwa | Typ | Czy wymagany? | Opis |
| dateFrom | DATE | TAK | Początkowa data przedziału |
| dateTo | DATE | TAK | Końcowa data przedziału |
| id | long | TAK | Identyfikator stacji paliw |

Dane zwrócone przez powyższy punkt końcowy są to wszystkie zweryfikowane ceny, pogrupowane zgodnie z typami paliw, podane w zadanym okresie czasu dla wskazanej stacji paliw.

Trzeci punkt końcowy udostępnia statystki pojedynczego użytkownika. Są to inforamcję takie jak ilość wystawionych ocen, ilość dodanych cen czy ilość zebranych punktów.

GET //app-user/{id}/stats

Opis parametrów

|  |  |  |  |
| --- | --- | --- | --- |
| Nazwa | Typ | Czy wymagany? | Opis |
| id | long | TAK | Identyfikator użytkownika |

#### Notyfikacje

Aplikacja serwerowa posiada także system notyfikacji. Notyfikacje są wysyłane do użytkowników kiedy stacja paliw obserwowana przez nich otrzymuje nową cenę. Dla każdego użytkownika po zalogowaniu do systemu dodawany jest token który jest używany przez Firebase do wysłania notyfikacji do określonego urządzenia. Każdy użytkownik posiada listę stacji które obserwuje. Jeżeli na stacji podana jest cena która zostaje uznana za zweryfikowaną to serwer wyszukuje użytkowników którzy obserwują stacje której cena została właśnie zaktualizowana. Kiedy lista użytkowników jest już znana to system odrzuca tych dla których nie został podany token. Następnie budowane jest żądanie w którym określone zostaje kontent wysyłanej notyfikacji. Kiedy żądanie jest już przygotowane to zostaje wysłane do Firebase. Firebase rozsyła notyfikacje do urządzeń wskazanych po przez listę tokenów. Odebranie notyfikacji uruchamia procedurę która jest określona przez aplikację kliencką. Dzięki użyciu Firebase możliwa jest sprawna implementacja tej funkcjonalności. Ponieważ API to automatyzuję autoryzację takich notyfikacji oraz udostępnia biblioteki które implementują nasłuchiwanie bądź wysyłanie żądania z notyfikacją.

#### Rejestracja

Na potrzeby aplikacji została też zaimplementowana dwuetapowa rejestracja. API wystawia punkt końcowy który obiera zapytanie mające na celu stworzenie konta użytkownika.

POST

/app-user

Jako ciało zapytania podajemy obiekt JSON według następującego schematu.

{

„username”: „$username”,

„email”: „$email”,

„password”: „$password”

}

Opis pól

|  |  |  |
| --- | --- | --- |
| Nazwa | Typ | Opis |
| username | string | Nazwa użytkownika |
| Email | string | Adres email użytkownika |
| password | string | Hasło do konta |

Jeżeli podane dane spełniają kryteria użytkownik z podanymi parametrami zostaje dodany do bazy danych jako niezweryfikowany. Następnie na podany email wysyłany jest email z linkiem weryfikacyjnym. Link weryfikacyjny tworzony jest według następującego schematu:

GET

{serverAdress}/verify/{token}

Opis pól

|  |  |  |
| --- | --- | --- |
| Nazwa | Typ | Opis |
| serverAdress | String | Docelowy adres serwera |
| token | String | Token weryfikacyjny |
| password | String | Hasło do konta |

Sam token weryfikacyjny tworzony jest podczas dodawania użytkownika do bazy danych. Token jest zapisywany do bazy danych w celu późniejszej weryfikacji oraz wykorzystany do budowy linku weryfikacyjnego. Token jest to JWT(JSON web token) jest tworzony z następujących danych email, hasło użytkownika, data utworzenia konta oraz sekret. Użycie klucza JWT gwarantuje że, sfałszowanie tokenu będzie trudne oraz umożliwia wstawienie daty utworzenia tokenu. Dzięki dacie utworzenia tokenu wiemy czy ten już wygasł. Użytkownik niezweryfikowany nie może zalogować się do systemu pomimo że w nim istnieje. Kiedy użytkownik klika w link aktywacyjny to zostaje wysłane żądanie do punktu końcowego. Z adresu URL żądania zostaje wyciągnięty token który jest walidowany oraz porównywany z tokenem z bazy danych. Jeżeli token jest prawidłowy oraz równy temu z bazy danych to weryfikacja użytkownika kończy się powodzeniem. Od tego momentu użytkownik może logować się do systemu. Dodatkowo są mu przydzielane wartości domyślne w systemie lojalnościowym. Po zarejestrowaniu istnieje możliwość zmiany hasła także oparta o wysyłanie linku weryfikacyjnego.

#### Autoryzacja

Większość danych udostępnianych przez aplikację może być publiczna. Są to dane takie jak ceny, statystyki czy położenie stacji. Jednak możliwość edycji danych oraz dane wrażliwe powinny być chronione. Aplikacja posiada system autoryzacji oparty o JWT(JSON web token) oraz filtry. W celu ułatwienia pracy wykorzystane zostały biblioteki Spring Security oraz JJWT. Tylko niektóre z punktów końcowych aplikacji są w pełni dostępne. Aby uzyskać dostęp do większości z nich wysyłane zapytanie musi pomyślnie przejść proces autoryzacji. W tym celu został użyty mechanizm filtrów. Przed przetworzeniem zapytania jest ono weryfikowane. Dla chronionych punktów końcowych z zapytania pobierany jest nagłówek z kluczem „Authorization” pod którym znajduje się token weryfikacyjny. Jeżeli zapytanie odwołuje się do chronionego zasobu i nie posiada tokena w nagłówku to jest ono odrzucone. Po wyłuskaniu tokenu jest on weryfikowany. Token jest to JWT a w jego parametrach znajdują się dane o emailu użytkownika oraz haśle, dacie logowania oraz sekrecie. Jeżeli sekret jest prawidłowy oraz hasło i email są poprawne a sam token nie wygasł to zostaje udzielony dostęp do zasobu. Aby uzyskać token autoryzujący należy przejść pomyślnie proces logowania. Punkt końcowy służący do logowania znajduje się poniżej.

POST

/auth

Jako ciało zapytania podajemy obiekt JSON według następującego schematu.

{

„email”: „$email”,

„password”: „$password”}

Opis pól:

|  |  |  |
| --- | --- | --- |
| Nazwa | Typ | Opis |
| email | String | Adres email urzytkownika |
| password | String | Hasło do konta |

Jeżeli podane dane są prawidłowe to w odpowiedzi zwracany jest obiekt JSON który pod kluczem „authToken” posiada token autoryzujący.

#### System lojalnościowy

Każdy użytkownik posiada także liczbę punktów która oznacza role użytkownika w systemie. Role są hierarchiczne i każda. Im wyższa role tym użytkownik jest w stanie więcej edytować w systemie. Każda z ról odpowiada innemu przedziałowi punktów. Rola dziedziczy możliwości ról niższych. Nowy użytkownik posiada najniższą rolę. Przedziały te ustawiane są w konfiguracji. Poniżej zamieszczono tabele ról wraz z możliwościami które odblokowują.

|  |  |
| --- | --- |
| Nazwa roli | Możliwości edycji |
| Nowy | -Dodanie nowej ceny  -Dodanie opinii |
| Początkujący | -Edycja dostępności dodatkowych udogodnień |
| Zaufany | -Edycja danych o stacji takich jak np. adres |
| Stały | -Dodanie nowej stacji  -Dodanie nowej ceny nie wymaga jej weryfikacji |

Implementacja nadawania roli na podstawie punktów daje większą elastyczność. Umożliwia wprowadzenie większej ilości ról oraz zmianę kryteriów ich przydzielania. Dodatkowo punkty mogą być wykorzystane do implementacji nowych funkcjonalnością takich jak np. rabaty. Nadawanie punktów użytkownikowi oparte jest o podejmowane przez niego operację. Dla każdej operacji takiej jak np. dodanie nowej ceny przydzielane są mu dodatkowe punkty. Przydzielanie punktów było sprawą problematyczną z uwagi na ilość kodu który wymagałby edycji. Ponieważ każda operacja musi być określona pod względem dostępności dla danej roli a także ilości punktów które przydziela. Świetnym rozwiązanie tego problemu jest programowanie aspektowe które jest umożliwione przez bibliotekę Spring AOP. Dzięki temu możliwe było dodanie nowych procedur do istniejących już operacji bez ingerencji w ich kod. Każda operacja określona jest jako obiekt istniejący w kontenerze aplikacji. Dzięki temu że, aplikacja wykorzystuje bibliotekę spring obiekty wywoływane są po przez obiekty proxy. Obiekt proxy jest to wzorzec projektowy który definiuje że, operacja nie jest wykonywana przez obiekt bezpośrednio lecz przechodzi początkowo przez inny obiekt stanowiący swego rodzaju dodatkową warstwę. Właśnie do tej warstwy możemy dodać dodatkowe operację. Określając punkty przecięcia czyli metody o danej sygnaturze rejestrujemy operację które mają się wykonać przed lub po wykonaniu określonych metod. Dzięki takiemu podejściu jesteśmy w stanie określić ilość dodawanych punktów dla każdej operacji w systemie edytując tylko jeden plik. Warto wspomnieć że, nie należy nadużywać mechanizmu aspektów ponieważ duża ich ilość znacząco utrudnia pracę z kodem ponieważ procedury dziejące się w aspektach nie są wywoływane jawnie. Osoba nie znająca aspektów znajdujących się w systemie nie jest w stanie dowiedzieć się o nich czytając kod programu. Programowanie aspektowe powinno stosować się do specyficznych funkcjonalności lub bardzo technicznych fragmentów aplikacji.

#### Konfiguracja i profile

Aplikacja serwerowa posiada dużą gamę wartości stałych. Wartości te są czysto techniczne np. adres bazy danych czy konfiguracja połączenia z serwerem poczty. Oprócz technicznych stałych istnieją także wartość określające funkcjonalność np. tekst notyfikacji. Wartości te muszą być różne dla środowiska programisty i produkcyjnego. Ponieważ nie możliwe jest optymalne pracowanie z projektem skonfigurowanego tak aby działał na środowisku produkcyjnym. W przypadku opisywanej aplikacji jest to wręcz niemożliwe a także kosztowne ponieważ API łączy się z serwisami które każą sobie płacić za ich używanie. Można więc założyć że programista potrzebuje określonej konfiguracji serwera. Rozwiązaniem tego problemu jest skorzystanie z frameworka Spring który pozwala na wyniesienie stałych do osobnych plików .yaml. Pliki te zawierają zbiór wartości określających aspekty techniczne oraz funkcjonalności. Dodatkową rzeczą jest określenie tak zwanych profili aby nie było konieczne zmienianie wielu wartości przy każdym starcie aplikacji. Profil jest to zbiór wartości które nadpisują wartości domyślne. Wartości domyślne określone są w pliku application.yaml. Natomiast pliki z profilem określone są w plikach {profile}-applicaiton.yaml. Dzięki temu możliwe było wyciągnięcie dwóch profili. Profil produkcyjny określony w pliku prod-application.yaml posiada parametry które wyłączają zapełnianie bazy danych sztucznymi danymi oraz konfigurują połączenie z bazą Postgre na platformie Heroku. Profil deweloperski określony w pliku dev-application.yaml uruchamia zapełnianie bazy danych sztucznymi danymi oraz inicjalizuje lżejszą bazę danych H2. Takie podzielenie profili pozwala na szybszy start aplikacji, brak potrzeby dbania o zawartość produkcyjnej bazy danych oraz brak konieczności przejmowania się kosztami podczas testów. Wszystko to znacznie usprawnia pracę nad projektem oraz start aplikacji w środowisku produkcyjnym.

# Aplikacja kliencka

Aplikacja kliencka została wykonana przy użyciu frameworka Angular. Angular pozwala na pisanie aplikacji SPA(Single Page Application). Wybór typu aplikacji jako przeglądarkowej SPA był poparty kilkoma argumentami. Dzięki aplikacji przeglądarkowej możliwe było dotarcie do szerszej grupy odbiorców niż dzięki aplikacji natywnej. Aplikacja natywna wymaga zaprogramowania jej wersji na wszystkie systemy operacyjne. Rozpatrując że, nasza aplikacja ma służyć jako aplikacja mobilne konieczne staję się napisanie dwóch aplikacji dla IOS i Android. Dodatkowo zalety aplikacji natywnej są nieznaczne w przypadku rozpatrywanej aplikacji. Ponieważ zalety takie jak skorzystanie z systemu plików czy innych natywnych funkcjonalności nie znajdują zastosowania. Natomiast wszystkie dostępne natywne funkcjonalności takie jak pobranie lokalizacji urządzenia są dostępne z poziomu przeglądarki. Dodatkowo użycie SPA znacząca ulepsza doświadczenie użytkownika. Ponieważ aplikacja SPA nie przeładowuje całej strony przy przejściu do innego segmentu aplikacji. Poza użyciem SPA zostały zaimplementowane funkcjonalności PWA(Progressive Web Application). Dzięki PWA aplikacja daje złudzenie aplikacji natywnej mimo iż jest to nadal aplikacja natywna. PWA udostępnia takie funkcjonalności jak możliwość zainstalowania aplikacji tak by była widoczna na pulpicie urządzenia, notyfikacje, działanie w trybie offline czy szybsze ładowanie aplikacji. Istnieje także możliwość dodania aplikacji do sklepu Google Play czy App Store. Taki wybór typu aplikacji pozwala na zdobycie szerokiej gamy użytkowników.

##### Opis poszczególnych widoków aplikacji klienckiej

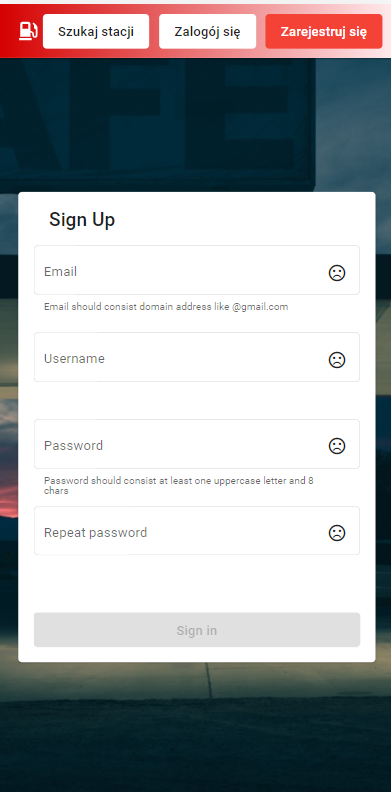
1. Ekran powitalny



*Rys.5 Wygląd ekranu powitalnego*

Nowy użytkownik po wejściu do aplikacji widzi ekran powitalny. Na ekranie powitalnym widnieje zachęta do wyszukania stacji oraz możliwość logowania i rejestracji.

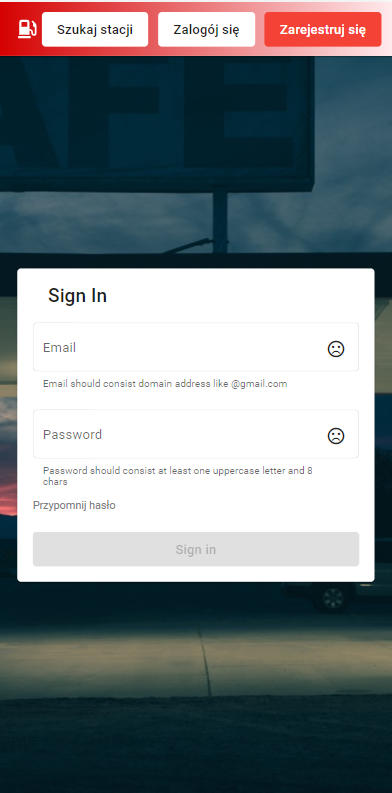
1. Ekran rejestracji



*Rys.6 Ekran rejestracji*

Ekran rejestracji posiada szereg pól wymagających uzupełnienia. Są to kolejno Email nazwa użytkownika, hasło oraz powtórzone hasło. Każde pole jest walidowane a niepoprawna wartość jest sygnalizowana wyłączonym przyciskiem rejestracji, czerwoną ramką pola które zawiera niepoprawną wartość oraz emotikonką. Jeżeli dane są poprawne po wciśnięciu przycisku „Zarejestruj się” zostanie wysłany mail z linkiem weryfikacyjnym oraz nastąpi przekierowanie do strony logowania.

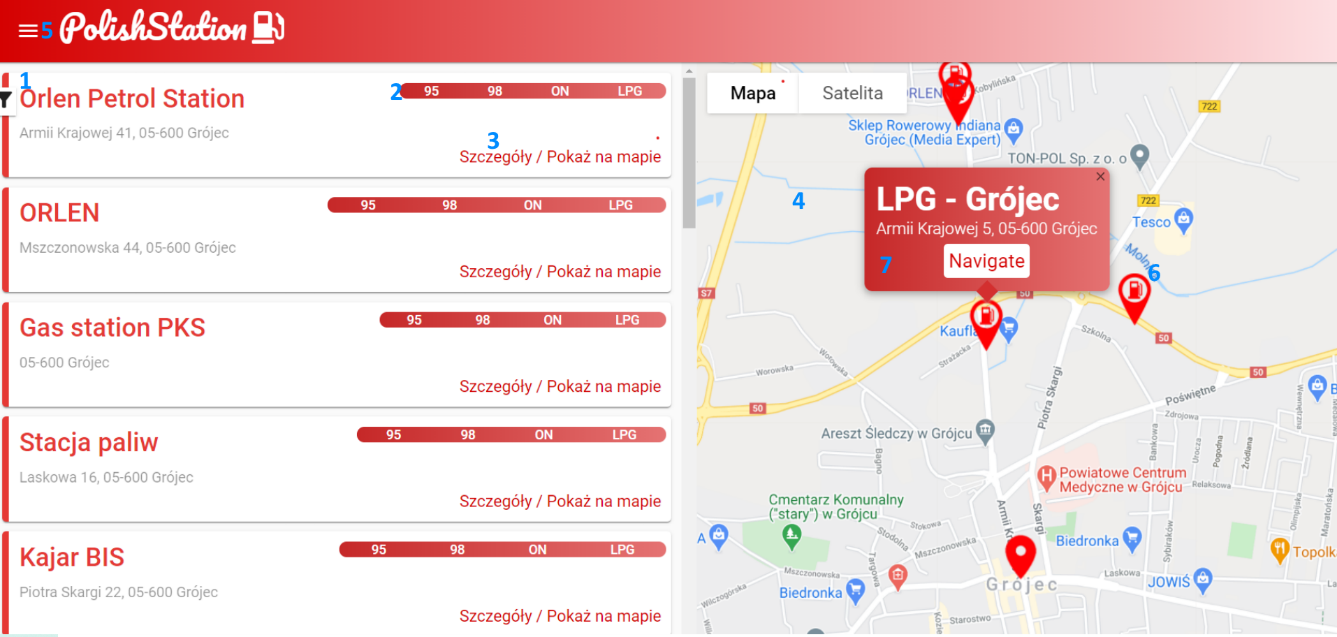
1. Ekran powitalny

******

*Rys.7 Wygląd ekranu logowania*

Ekran logowania jest bardzo podobny do ekranu rejestracji. Ponownie występują to walidowane pola czyli hasło i email użytkownika. Jeżeli wprowadzone dane są poprawne po wciśnięciu przycisku „Zaloguj się” nastąpi przekierowanie do strony wyszukiwania stacji.

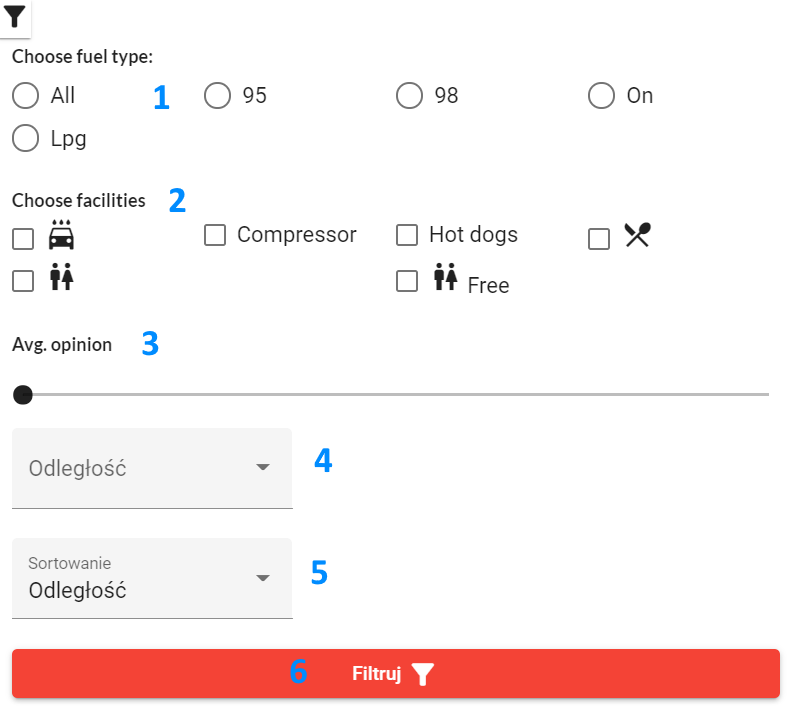
1. Ekran powitalny

******

*Rys.8 Widok wyszukiwania dla wersji desktopowej*

Ekran wyszukiwania posiada wiele funkcjonalności mających na celu ułatwienie znalezienia stacji. Dane prezentowane są w postaci listy lub punktów na mapie.

1. Przycisk wysuwający menu do ustalenia kryteriów wyszukiwania oraz sortowania
2. Lista dostępnych typów paliwa oraz ostanie ceny jeżeli te są znane
3. Przycisk szczegóły przekierowuje do strony prezentującej szczegółowe dane o stacji. Natomiast przycisk pokaż na mapie przewija mapę tak aby pokazywała zaznaczoną stację.
4. Mapa reprezentująca znalezione stacje
5. Hamburger buton rozwijający menu aplikacji
6. Ikona reprezentująca położenie znalezionej stacji
7. Skrót informacyjny pokazywany po najechaniu kursorem na ikone lub kliknięcie ikony. Przycisk „Nawiguj” przekierowuje do nawigacji która kieruje nas do tego punktu.

******

*Rys.9 Panel do wyszukiwania stacji*

1. Pole jednokrotnego wyboru służące do wyboru typu paliwa
2. Pole wielokrotnego wyboru służące do określenia wyszukiwanych udogodnień
3. Suwak do określenia wymaganej średniej oceny stacji
4. Lista do wybrania odległości od położenia użytkownika w której będą wyszukiwane stacje
5. Kryterium sortowania znalezionych stacji
6. Przycisk filtruj inicjalizuje wyszukiwanie

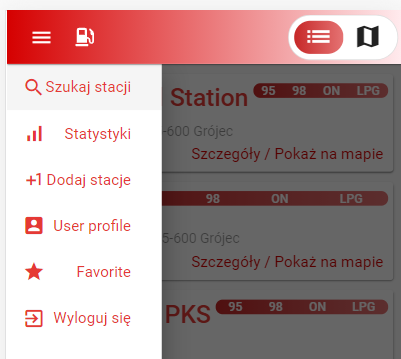
******

*Rys.10 Mobilna wersja ekranu wyszukiwania*

Ekran mobilny do wyszukiwania jest bardzo podobny do wersji desktopowej. Jednak ma jedna różnicę. Z powodu szerokości ekranu niemożliwe jest dodanie mapy. Dlatego w prawym górnym rogu został dodany przycisk umożliwiający zmianę reprezentacji znalezionych rekordów. Istnieje także wariant ekranu wyszukiwania stacji dla użytkownika niezalogowanego jednak ten różni jedynie brakiem przycisku „Szczegóły” w liście stacji.

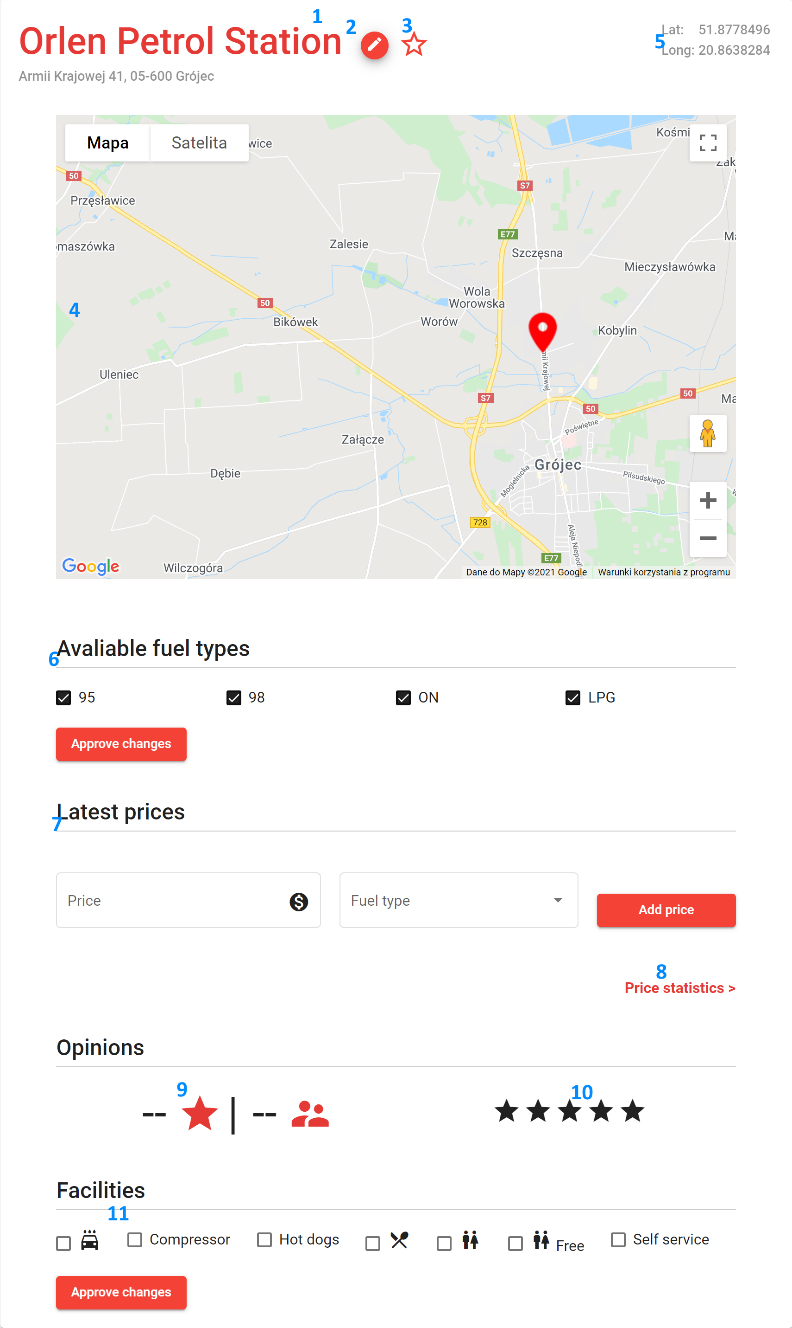
1. Menu

Menu służy do nawigowania po aplikacji. Kliknięcie przycisku z rozwijalnej listy przekierowuje do odpowiedniej podstrony. Wyjątkiem jest przycisk „Wyloguj się” który przekierowuje użytkownika do strony logowania.



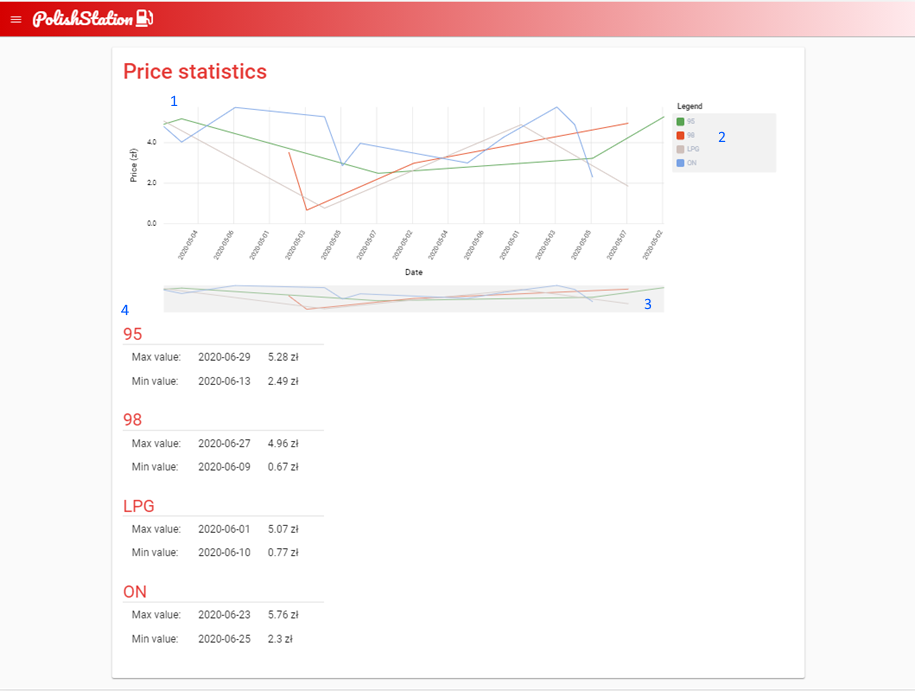
*Rys.11 Rozwijalne menu*

1. Ekran szczegółów stacji



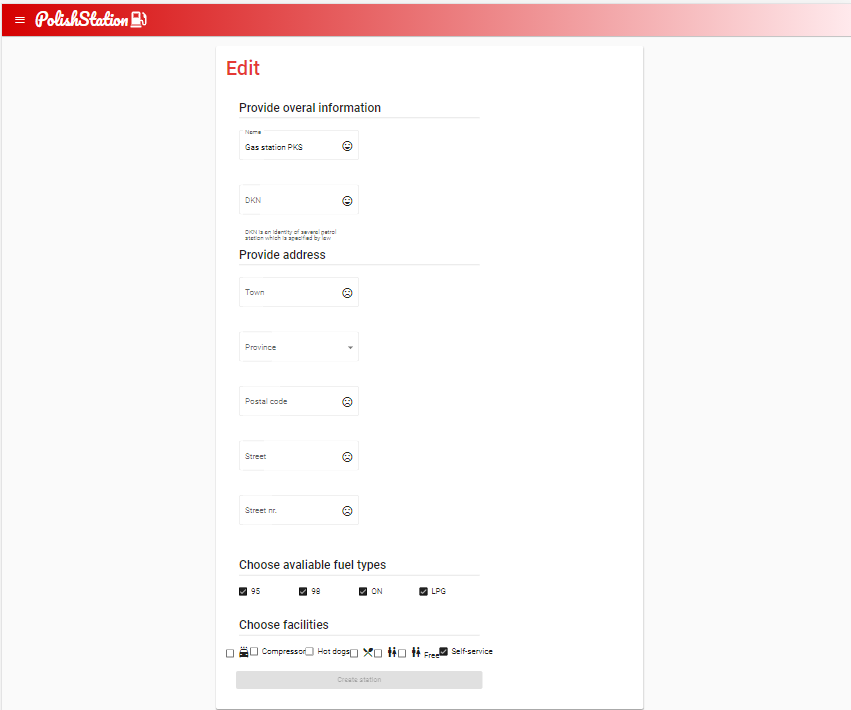
*Rys.12 Wygląd ekranu szczegółów*

1. Nazwa i adres stacji
2. Przycisk umożliwiający edycje danych takich jak nazwa czy adres. Dostępny tylko dla zaufanych użytkowników.
3. Przycisk dodania stacji do ulubionych
4. Położenie stacji na mapie
5. Długość i szerokość geograficzna punktu
6. Dostępne typy paliwa. Edycja możliwa tylko dla użytkowników posiadających odpowiednie role
7. Moduł umożliwiający dodanie nowej ceny
8. Przycisk przekierowuje do statystyk cen na danej stacji
9. Pola pokazują średnią ocen oraz liczbę użytkowników którzy wystawili ocenę
10. Pole służy do podania oceny. Ocena jest w skali od 1 do 5 z krokiem 1
11. Moduł służący do podania dostępnych udogodnień na stacji. Edycja możliwa tylko dal użytkowników o określonej roli.
12. Ekran statystyk cen na stacji

******

*Rys.13 Ekran statystyk*

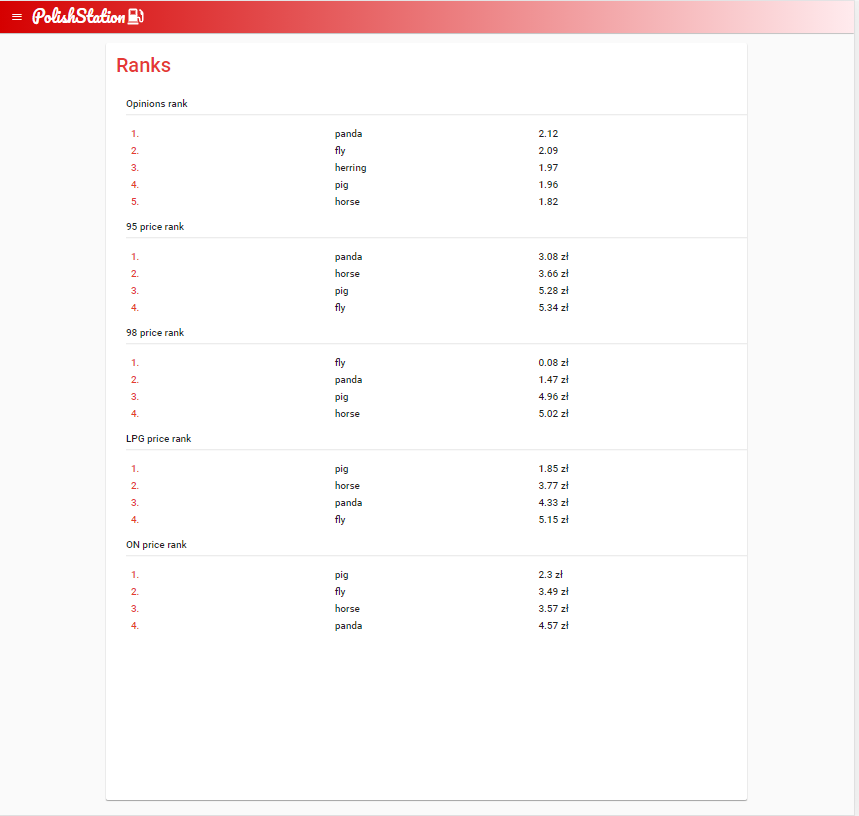
1. Wykres cen w zależności od czasu
2. Legenda przypisująca typ paliwa do konkretnego koloru
3. Miniatura wykresu z której możemy wybrać przedział czasu. Zakres przedziału to maksymalnie rok.
4. Minimalne i maksymalne ceny dla każdego typu z ostatniego roku.
5. Ekran edycji szczegółów stacji

******

*Rys.14 Ekran edycji danych o stacji*

Ekran jest dostępny jedynie dla ograniczonej grupy użytkowników. Ekran pozwala na edycje takich danych jak adres, nazwa, numer DKN, dostępne typy paliwa i udogodnienia. Wszystkie pola są walidowane a możliwość ich edycji jest dostępne tylko jeśli wszystkie wprowadzone dane są poprawnie. Położenie geograficzne jest edytowane automatycznie przez serwer. Jeśli dane o adresie zmienią się wtedy aplikacja serwerowa korzysta z Google API w celu określenia położenia geograficznego po przez adres a następnie zapisuje zmiany w bazie danych.

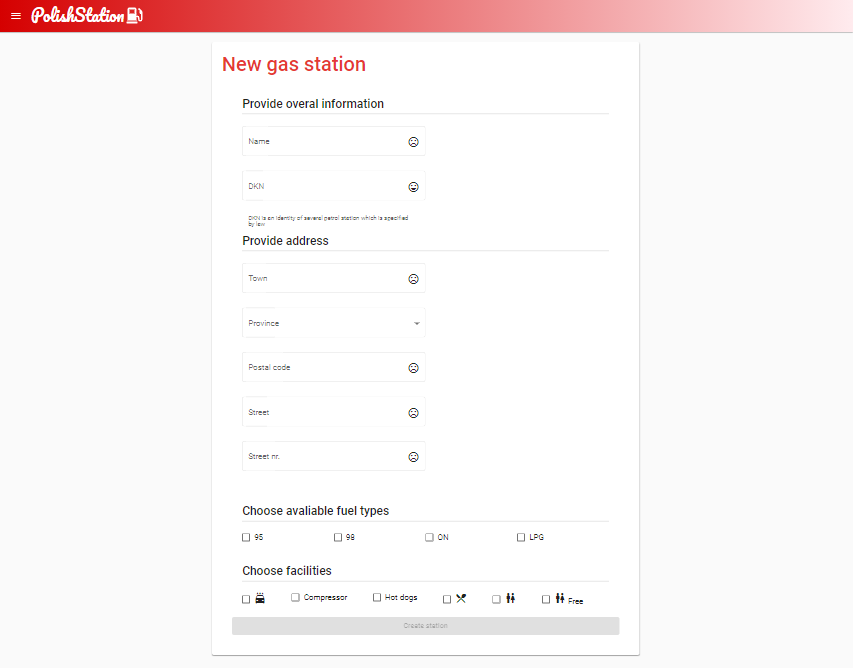
1. Ekran rankingów



*Rys.15 Ekran rankingów*

Ekran prezentuje 5(lub mniej jeżeli liczba danych nie pozwala na określenie dłuższej listy) najlepszych stacji według kryteriów. Kryteria to średnia ocena stacji oraz najniższa cena danego typu paliwa.

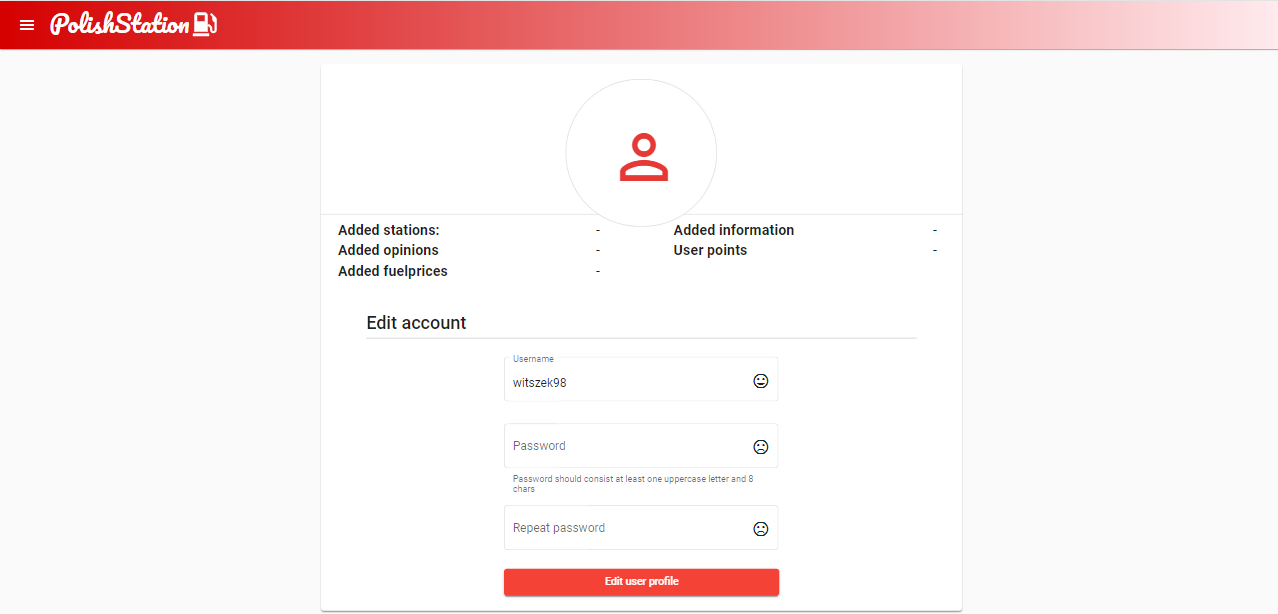
1. Ekran dodania nowej stacji



*Rys.16 Ekran dodawania nowej stacji*

Ekran bardzo podobny do ekranu edycji danych o stacji. Różnicą między tymi ekranami jest operacja aplikacji serwerowej do której się odwołują.

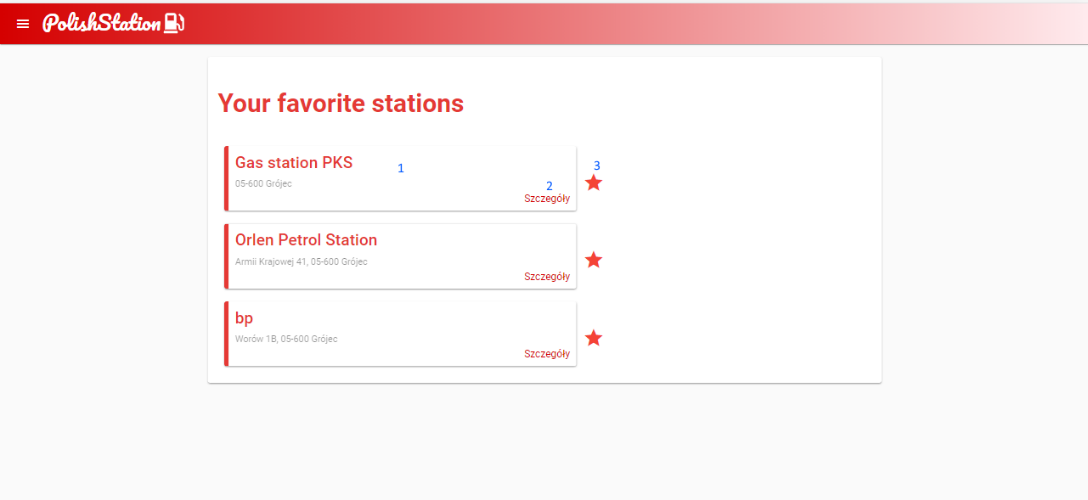
1. Profil użytkownika



*Rys.17 Profil użytkownika*

Profil użytkownika prezentuje statystyki akcji przez niego podjętych. Akcje są to kolejno liczba opinii, dodanych cen, informacji oraz ilość uzyskanych przez niego punktów. Ekran pozwala także na zmianę hasła i nazwy użytkownika.

1. Ekran ulubionych stacji

******

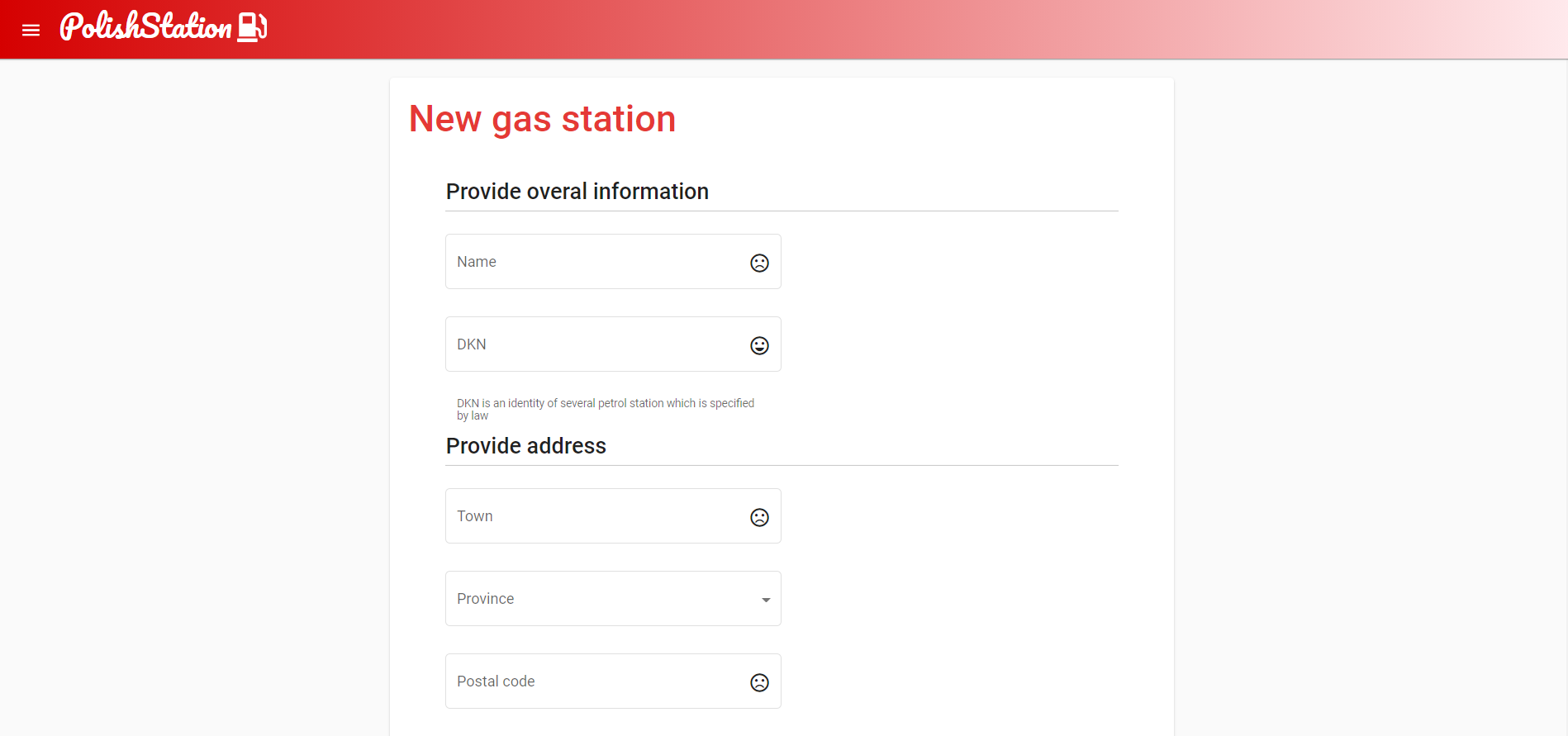
*Rys.18 Lista polubionych stadcji*

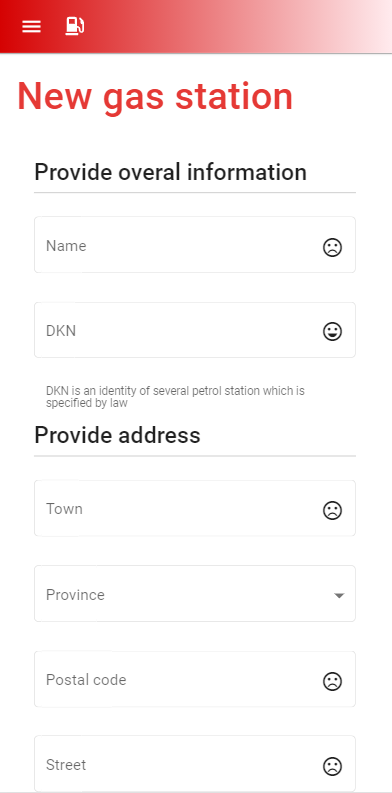
Ekran prezentuje polubione przez użytkownika stacji. Notyfikacje przychodzące na urządzenie użytkownika dotyczą stacji zaprezentowanych na tym widoku.

1. Polubiona stacja paliw
2. Przycisk przenoszący do szczegółów stacji
3. Przycisk pozwalający na anulowanie polubienia. Anulowanie polubienia zatrzyma wysyłanie notyfikacji dotyczących tej stacji.

##### RWD

Skrót RWD oznacza Responsive Web Design. Podejście do tworzenia aplikacji przeglądarkowych w taki sposób aby samoczynnie dostosowały swój wygląd do różnych rozmiarów urządzeń. Samo skalowanie ma na celu poprawienie doświadczenia użytkownika ponieważ układ strony który wygląda estetycznie na komputerze jest nieużyteczny w wersji mobilnej. Skalowanie strony oparte jest na funkcjonalności CSS który pozwala na zmianę wartości np. szerokości w zależności od szerokości ekranu. Dostępne są także jednostki określające rozmiar który jest zależny od szerokości ekranu. Dodatkowo rozmiar piksela w języku CSS jest zależny od współczynnika który jest wyliczany dzieląc ilość pikseli na cal ekranu. Dzięki takiemu traktowaniu rozmiaru piksela możliwe jest określenie rozmiaru urządzenia. Dla przykładu dzisiejsze urządzenia mobilne posiadają ekrany o rozdzielczościach dorównujących ekranom monitorów. Jednak w rzeczywistości wyświetlając element w tych samych rozmiarach laptopie i urządzeniu mobilnym okaże się że element jest nieczytelny na telefonie. Mówiąc łopatologicznie jeden piksel w żaden sposób nie odpowiada fizycznemu pikselowi znajdującego się na w matrycy. Niekiedy jeden piksel na urządzeniu mobilnym odpowiada kilkudziesięciu pikselom fizycznym. Standardem stał się podział ekranu na 12 kolumn dla urządzenia desktopowego. Natomiast urządzenia o ekranie mniejszym posiadają ilość kolumn rzędu 3 lub 6. Takie podejście pozwala na zmianę pozycji elementu na stronie w zależności od szerokości urządzenia stosując jedynie kilka progów szerokości urządzenia. Zastosowanie takiego podejścia w opisywanej aplikacji pozwala na korzystanie z niej w znacznie większej ilości urządzeń.

******

******

*Rys.19 Przykład wykorzystania RWD w aplikacji*

##### PWA

PWA czyli progressive web application jest to aplikacja webowa która korzysta z natywnych funkcjonalności urządzenia dając w rezultacie doświadczenie podobne do korzystania z aplikacji natywnej. PWA opiera się na wykorzystaniu funkcjonalności przeglądarki czyli Service Worker. Service Worker jest to specjalny wątek przeglądarki któremu przekazane jest określone zadanie. Jest to bardzo przydatne z punktu widzenia efektywnego działania aplikacji ponieważ w rzeczywistości język Java Script który obsługuje przeglądarka jest jednowątkowy. Natomiast Service Worker posiada faktycznie osobny wątek. Dzięki zastosowaniu PWA w aplikacji znacznie zoptymalizowano czas ładowania aplikacji ponieważ ta nie musi zaciągać wszystkich plików przy każdym jej starcie. Wszystkie pliki potrzebne do działania są już zainstalowane na urządzeniu użytkownika. Jednak aby możliwe było aktualizowanie aplikacji ta musi posiadać aktywny service worker który będzie odpowiedzialny za sprawdzanie czy aktualizacje są dostępne. Kolejnym atutem była możliwość wykorzystania powiadomień. Natywne notyfikacje mogą być wykorzystane jedynie stosując PWA.

# Podsumowanie

Opracowana aplikacja do udostępniania danych o cenach paliw spełnia postawione wymagania oraz oczekiwania. Najważniejszymi wymaganiami było zaimplementowanie części serwerowej aplikacji korzystając z architektury REST oraz opracowanie aplikacji klienckiej korzystającej z PWA. Kolejnym narzuconym założeniem było skorzystanie z natywnych funkcjonalności urządzeń takich jak np. moduł GPS. Kolejne wymagania stawiane przed autorem pochodziły z bardzo szerokiego zakresu tematów takich jak bezpieczeństwo, architektura systemu, działanie przeglądarki etc. Szczególną uwagę trzeba zwrócić na stosowanie praktyk w trakcie pisania kodu dzięki czemu zaimplementowania sprzyja skalowalności systemu oraz jego rozwojowi. Wspomniany rozwój jest możliwy na bardzo wielu płaszczyznach począwszy od bardzo detalicznych takich jak wprowadzenie nowego typu użytkownika kończąc na możliwości wykorzystania aplikacji serwerowej w innym systemie czy programie.

Powstały system powstał z myślą o rozszerzeniu funkcjonalności istniejących już systemów takich jak np. Google Maps które choć pokazują stacje w okolicy nie dają możliwości udostępnienia ceny czy innych aspektów. Aplikacja jest zintegrowana z wieloma zewnętrznymi serwisami takimi jak Firebase czy Google Map. Wykorzystanie zewnętrznych serwisów stawiało dodatkowe problemy takie jak synchronizacja danych czy inne podejście do uwierzytelniania. Bez wykorzystania zewnętrznych serwisów stworzenie opisywanej aplikacji nie byłoby możliwe lub zmniejszyło by to jej użyteczność.

Choć temat pracy został zrealizowany autor uważa że system można rozszerzyć o kolejne funkcjonalności:

Wprowadzenie kolejnych powiadomień związanych chociażby ze zweryfikowaniem ceny

* Dodanie modułu udostępniających rabaty oraz reklamy co mogłoby zachęcić twórców innych systemów do zintegrowania się z opisywaną aplikacją
* Stworzenie natywnych aplikacji klienckich na systemy Android czy IOS
* Pokrycie kodu testami w większym stopniu. Z naciskiem na testy integracyjne
* Umożliwienie zarządzania danymi o posiadanej stacji jej właścicielom. Wymaga to obmyślenia skutecznej logiki pozwalającej na zweryfikowanie czy podmiot jest właścicielem stacji.

Podsumowując, warty zauważenia jest ilość rozwiązanych problemów na wielu płaszczyznach. Największym problemem było posiadanie lub zdobyci wiedzy z wielu różnych dziedzin informatyki. Opracowanie tej aplikacji wymagało od autora znajomości zagadnień związanych z bazą danych, aplikacjami serwerowymi, aplikacjami klienckimi, architekturą oprogramowania, paradygmatami programowania i wielu innych. Auto opracowując opisywaną aplikację zdobył wiedzę i umiejętności z bardzo szerokiego zakresu tematów co znacząco sprzyja dalszej karierze zajmującej się rozwojem oprogramowania.

# Bibliografia

[ 1 ] „Data Never Sleeps 8.0” <https://www.domo.com/learn/data-never-sleeps-8>

[ Data uzyskania dostępu 15.02.2021 ].

[ 2 ] „How much is your data worth?” <https://www.forbes.com/sites/stephanzoder/2019/08/06/how-much-is-your-data-worth/?sh=2c96c00470fc>

[ Data uzyskania dostępu 15.02.2021 ].

[ 3 ] Rober C. Martin „Clean Code”

[ 4 ] Cay S. Horstmann „Java. Podstawy. Wydanie X”

[ 5 ] Cay S. Horstmann „Java. Techniki zaawansowane. Wydanie X”

[ 6 ] „The State of Developer Ecosystem 2020” <https://www.jetbrains.com/lp/devecosystem-2020/>

[ Data uzyskania dostępu 15.02.2021]

[ 7 ] Pivotal Software, Incorporate „Spring Data”

<https://spring.io/projects/spring-data>

[ Data uzyskania dostępu 15.02.2021]

[ 8 ] „Google Geocoding API Documentation” <https://developers.google.com/maps/documentation/geocoding/overview>

[ Data uzyskania dostępu 15.02.2021]

[ 9 ] „Infrastruktura paliw ciekłych” <https://dane.gov.pl/pl/dataset/1091,infrastruktura-paliw-ciekych-stacje-paliw>

[ Data uzyskania dostępu 15.02.2021 ]

[ 10 ] „Mapstruct documentation:”

<https://mapstruct.org/documentation/installation/>

[ Data uzyskania dostępu 15.02.2021 ]

[ 11 ] „Lombok documentation”

<https://projectlombok.org/features/all>

[ Data uzyskania dostępu 17.02.2021]

[ 12 ] „JWT Documentation „

<https://jwt.io/introduction>

[ Data uzyskania dostępu 15.02.2021 ]

[ 13 ] „Google Maps Platform, Documenation of Javascript API” <https://developers.google.com/maps/documentation/javascript/overview>

[ Data uzyskania dostępu 15.02.2021 ]

[ 14 ] „Design patterns”

<https://refactoring.guru/design-patterns/what-is-pattern>

[ Data uzyskania dostępu 15.02.2021 ]

[ 15 ] „Functional Programming in Java”

<https://www.baeldung.com/java-functional-programming>

[ Data uzyskania dostępu 16.02.2021 ]

[ 16 ] „Firebase Documentation”

<https://firebase.google.com/docs>

[ Data uzyskania dostępu 17.02.2021 ]

[ 17 ] „Layered Architecture”

<https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html>

[ Data uzyskania dostępu 17.02.2021 ]

[ 18 ] „Projektowanie i normalizacja bazy danych”

<https://www.sqlpedia.pl/projektowanie-i-normalizacja-bazy-danych/>

[ Data uzyskania dostępu 17.02.2021 ]