

Spis treści

Wykaz skrótów	3
1 WPROWADZENIE	4
1.1 Motywacja	4
1.1.1 Wykluczenie społeczne	4
1.1.2 Dynamiczny rozwój rynku aplikacji mobilnych	4
1.1.3 Brak gotowych rozwiązań	4
1.2 Cel pracy	4
1.2.1 Inspiracja	5
2 ANALIZA PROBLEMU	6
2.1 Trasowanie po sklepie	6
2.1.1 Podobieństwa i różnice z problemem komiwojażera (TSP)	6
2.1.2 Wnioski	6
2.2 Przetwarzanie języka naturalnego	7
2.2.1 Przewaga modeli językowych nad modelami generatywnymi	7
2.3 Obecne rozwiązania	8
2.3.1 Walmart App	8
2.3.2 Asystenci głosowi	9
2.3.3 Wnioski	9
3 PROPOZYCJA ROZWIĄZANIA PROBLEMU	10
4 OPIS ROZWIĄZANIA	11
4.1 Architektura systemu	11
4.2 Baza danych	12
4.2.1 Opis bazy danych	12
4.2.2 Szczegółowy opis tabel	12
4.3 Interfejs użytkownika	14
4.3.1 Opis dostępnych widoków	14
4.4 Serwer aplikacji	27
4.4.1 JavaScript	27
4.4.2 Node.js	27
4.4.3 Express.js	27
5 OPIS TECHNICZNY	28
5.1 Aplikacja wit.ai	28
5.1.1 Intencje i encje	28
5.1.2 Kreator	29
5.1.3 Testowanie i publikacja	30
5.1.4 Integracja z aplikacją mobilną	30
5.1.5 HTTP API	30
5.2 Serwer	31
5.2.1 Struktura serwera	31
5.2.2 CRUD	35
5.2.3 Autoryzacja	35
5.2.4 Baza danych	36

6 INSTRUKCJA UŻYTKOWANIA	39
7 WYNIKI TESTÓW	40
8 PODSUMOWANIE	41
8.1 Napotkane wyzwania	41
8.1.1 Obsługa nadajników BLE	41
8.1.2 Praca w zespole	41
8.2 Przyszły rozwój aplikacji	41
8.2.1 Panel Administratora	41
8.2.2 Rozwinięcie modelu językowego	42
8.2.3 Rozwój aplikacji mobilnej	42
8.3 Wnioski	42
8.3.1 Osiągnięte cele	42
8.3.2 Wartość praktyczna	43

Wykaz skrótów

AI Artificial Intelligence

NLP Natural Language Processing

API Application Programming Interface

NLI Natural Language Interfaces

BLE Bluetooth Low Energy

SQL Structured Query Language

ORDBMS Object-Relational Database Management System

Rozdział 1

WPROWADZENIE

Na przestrzeni ostatnich kilku dekad miał miejsce gwałtowny rozwój technologii. Przyczyniło się to do zwiększenia tempa życia każdego. Ludzie starają się optymalizować codzienne czynności, w celu odzyskania swojego czasu wolnego. W odpowiedzi na ten trend, powstaje wiele rozwiązań mających na celu usprawnienie życia codziennego ich użytkownika.

1.1 Motywacja

1.1.1 Wykluczenie społeczne

W obecnych czasach, internet jest dostępny w każdym miejscu na Ziemi. Przyczyniło się to do zwiększenia świadomości społecznej na temat inkluzywności. Produkty wypuszczane obecnie na rynek, starają się być dostępne dla każdego. Niesety to samo nie dotyczy rozwiązań i produktów dostępnych teraz na rynku. Jednym z sektorów, gdzie nie widać postępu w dostępności dla osób niepełnosprawnych jest sektor sprzedaży detalicznej. Osoby z wadami wzroku, słuchu lub ruchu nie mogą liczyć na wiele udogodnień w trakcie robienia zakupów. W związku z powyższym, główną motywacją stojącą za zrealizowaniem tego projektu jest chęć stworzenia aplikacji, która usprawni robienie zakupów osobom niepełnosprawnym. Jej założeniem jest ułatwienie robienia zakupów do tego stopnia, że osoba mająca problemy z poruszaniem się, widzeniem lub kontaktem z innymi ludźmi, mogłaby zrobić zakupy bez pomocy innej osoby. Znacznie wpłynełoby to nie tylko na komfort użytkownika, ale też odbiłoby się pozytywnie na wizerunku marki sklepu, który posiada taką aplikację.

1.1.2 Dynamiczny rozwój rynku aplikacji mobilnych

Smartfony (ang. *Smartphone*) są dziś w kieszeni każdego. W związku z tym, można zauważać dynamiczny rozwój rynku aplikacji mobilnych. Firmy i deweloperzy starają się odpowiedzieć na coraz bardziej wygórowane potrzeby konsumentów.

1.1.3 Brak gotowych rozwiązań

Dogłębna analiza rynku nie wykazała istnienia gotowych rozwiązań, które spełniałyby wszystkie wymagania postawione przed aplikacją. W związku z tym, postanowiono stworzyć własne rozwiązanie, które spełniałoby wszystkie wymagania. Więcej na temat analizy rynku można znaleźć w sekcji 2.3.

1.2 Cel pracy

Celem pracy jest wytworzenie kompletnej aplikacji mającej na celu ułatwienie robienia zakupów. Wymagania funkcjonalne świadczące o kompletności aplikacji to:

1. Interfejs służący do nawigacji po sklepie
2. Baza danych ze sklepami, wraz z ich lokalizacją i rozkładem produktów

3. Asystent AI pomagający w obsłudze aplikacji
4. Interfejs głosowy pozwalający na obsługę aplikacji przez osobę niedowidzącą
5. System zgrywania koszyka do kodu QR w celu szybszego zakończenia zakupów

Aplikacja spełniająca powyższe wymagania ma za zadanie nie tylko usprawnić robienie zakupów przeciętnemu użytkownikowi, ale przede wszystkim ułatwić tę czynność osobom niedowidzącym i seniorom. Następymi krokami, będą nawiązanie współpracy z klientem i komercjalizacja aplikacji. Projekt ma za zadanie rozszerzyć kompetencje autorów i pozwolić na napisanie aplikacji mającej realne szanse wejścia na rynek.

1.2.1 Inspiracja

Sklepy samoobsługowe

Na rynku istnieją już sklepy, które nie wymagają obsługi przez kasjera. Klient samodzielnie skanuje produkty i płaci za nie przy wyjściu. Przykładem takiego sklepu jest Amazon Go, lub Żabka Nano. Oba powyższe projekty działają na podobnej zasadzie. Przed wejściem do sklepu klient musi zeskanować kod QR ze swojej aplikacji mobilnej (z przypisaną kartą płatniczą), lub przyłożyć kartę do czytnika. Po zakończeniu autoryzacji drzwi się otwierają (Żabka Nano) lub otwiera się bramka (Amazon Go). W całym sklepie na suficie umieszczone są kamery. Ich zadaniem jest monitorowanie każdego z produktów. Kiedy klient zdejmuje produkt z półki, kamera rejestruje ten fakt. Kiedy klient odłoży produkt z powrotem na półkę, kamera rejestruje również to rejestruje. Po zakończeniu zakupów, klient wychodzi ze sklepu. System automatycznie oblicza wartość zakupów i ściąga odpowiednią kwotę z konta lub karty klienta.

Taki model sklepu zainspirował jeden z elementów aplikacji wytworzony w ramach tej pracy. Jest to system skanowania koszyka do kodu QR, który pozwala na szybsze zakończenie zakupów.

Rozdział 2

ANALIZA PROBLEMU

2.1 Trasowanie po sklepie

Sklep można przedstawić jako graf, w którym wierzchołki reprezentują sekcje sklepu, a te sąsiednie połączone są w grafie krawędziami. Sekcje to krótkie fragmenty alejki o długości 1–2 metrów, do których przypisane są konkretne produkty w bazie danych. Dzięki temu podejściu cały sklep można traktować jako graf, a to z kolei pozwala na zastosowanie dobrze znanych algorytmów do znajdowania ścieżek.

Przestępny graf sklepu ma prostą i powtarzaną strukturę: składa się z kilku długich, równoległych alejek, połączonych jedną poziomą alejką na dole i drugą u góry. Wagi krawędzi, są tak na prawdę odzwierciedlonymi wartościami odległości od obu środków sąsiednich sekcji. Są więc one często stałe, a wyjątki pojawiają się głównie na przejściach między alejkami i zakrętach.

Proces trasowania otrzymuje na wejściu listę wierzchołków zmapowaną z listy zakupów użytkownika. System nawigacji musi znaleźć najkrótszą drogę w grafie, zaczynając od aktualnej pozycji użytkownika (dostarczanej przez system pozycjonowania) i przechodząc przez wszystkie wierzchołki z listy.

2.1.1 Podobieństwa i różnice z problemem komiwojażera (TSP)

Na pierwszy rzut oka nasz mechanizm może kojarzyć się problem komiwojażera (*Traveling Salesman Problem, TSP*), w którym zadaniem jest odwiedzenie każdego punktu na grafie dokładnie raz i powrót do punktu startowego. Jak wskazano w pracy Gutin i Punnen (2002): „Problem komiwojażera jest jednym z najbardziej znanych problemów optymalizacji kombinatorycznej, a jego rozwiązanie wymaga innowacyjnych podejść ze względu na jego złożoność” [Gutin and Punnen(2002)]. Na szczęście, te drobne różnice sprawiają, że problem trasowania po sklepie jest znacznie prostszy:

- W trasowaniu po sklepie krawędzie i wierzchołki mogą być odwiedzane wielokrotnie, jeśli wymaga tego optymalna trasa.
- Nie ma konieczności powrotu do punktu początkowego, co znacząco upraszcza problem.
- Graf sklepu ma bardzo regularną strukturę i często stałe wagi krawędzi, co pozwala na duże optymalizacje.

2.1.2 Wnioski

Specyfika grafu sklepu oraz brak potrzeby dokładnego odwzorowania problemu TSP umożliwia wykorzystanie prostszych algorytmów, takich jak algorytm Dijkstry z powodzeniem. Dodatkowo przy dłuższych listach z produktami można zastosować heurystyki, takie jak algorytm 2-opt, które dodatkowo potrafi poprawić trasę. Jak podkreślił Croes (1958): „Algorytm 2-opt oferuje prostą, ale skuteczną metodę poprawy tras dla problemów trasowania w grafach, co czyni go odpowiednim w praktycznych zastosowaniach” [Croes(1958)].

2.2 Przetwarzanie języka naturalnego

„Przetwarzanie języka naturalnego (ang. *Natural Language Processing*) to dziedzina badań i zastosowań, która eksploruje, jak komputery mogą rozumieć i manipulować naturalnym językiem w formie tekstu lub mowy w celu wykonania użytecznych zadań”
[Chowdhary(2020)]

NLP znajduje zastosowanie w różnych obszarach, takich jak tłumaczenie maszynowe, przetwarzanie tekstu, streszczenie, interfejsy użytkownika, rozpoznawanie mowy i systemy ekspertowe. W szczególności w aplikacjach handlowych NLP może poprawić wyszukiwanie informacji i interakcje z użytkownikami.

Budowanie systemów NLP obejmuje analizę na kilku poziomach:

1. Foniczny i fonologiczny: wymowa i dźwięk.
2. Morfologiczny: analiza najmniejszych jednostek językowych.
3. Syntaktyczny: struktura zdań.
4. Semantyczny: znaczenie słów i zdań.
5. Dyskursywny i pragmatyczny: kontekst i wiedza zewnętrzna (Liddy, 1998; Feldman, 1999). [Chowdhary(2020)]

Natural Language Interfaces (NLI) umożliwiają użytkownikom zadawanie pytań w języku naturalnym, co może być szczególnie przydatne w aplikacjach zakupowych, np. „Gdzie znajdę makaron?” lub „Dodaj do koszyka mleko”. W przypadku aplikacji będącej tematem pracy, NLI zostało wykorzystane również do pomocy w obsłudze aplikacji.

2.2.1 Przewaga modeli językowych nad modelami generatywnymi

Natural Language Processing (NLP), czyli przetwarzanie języka naturalnego, odgrywa kluczową rolę w budowie nowoczesnych aplikacji zorientowanych na interakcję z użytkownikiem. W porównaniu do bardziej ogólnych modeli generatywnych, takich jak ChatGPT, rozwiązania NLP oferują szereg istotnych przewag, które czynią je bardziej odpowiednimi do zastosowań w określonych domenach.

Specjalizacja w przetwarzaniu języka

NLP zostało zaprojektowane z myślą o analizie, interpretacji i przetwarzaniu języka naturalnego w sposób ukierunkowany. Pozwala to na efektywne rozpoznawanie intencji użytkownika, ekstrakcję kluczowych informacji z tekstu oraz mapowanie tych informacji na konkretne działania w aplikacji. Na przykład w aplikacjach zakupowych system NLP może łatwo rozpoznać zapytanie użytkownika takie jak „Dodaj mleko do listy zakupów” i przypisać je do odpowiedniej akcji. Modele generatywne, choć potężne, mogą być mniej precyzyjne w scenariuszach wymagających określonej interpretacji danych.

Wydajność i efektywność zasobów

Rozwiązania NLP takie jak Wit.ai, są znacznie bardziej zoptymalizowane pod względem zużycia zasobów obliczeniowych w porównaniu do dużych modeli generatywnych, takich jak ChatGPT. W przypadku aplikacji mobilnych lub działających na urządzeniach z ograniczonymi zasobami, NLP pozwala na:

- szybsze przetwarzanie zapytań użytkownika,
- mniejsze zapotrzebowanie na moc obliczeniową,
- lepszą integrację z lokalnym systemem operacyjnym.

Dzięki temu rozwiązania NLP są bardziej dostępne dla szerokiego grona aplikacji użytkowych, w tym takich, które muszą działać w czasie rzeczywistym.

Integracja z procesami biznesowymi

Jednym z kluczowych atutów NLP jest jego zdolność do ścisłej integracji z procesami biznesowymi. Technologie te umożliwiają definiowanie przepływów (flows), które w sposób deterministyczny realizują zadania związane z rozpoznaną intencją użytkownika. W praktyce oznacza to, że intencje wykryte przez NLP mogą być bezpośrednio mapowane na działania, takie jak:

- dodanie produktu do koszyka,
- generowanie przypomnień,
- integracja z innymi systemami.

W porównaniu, modele generatywne wymagają dodatkowego przetwarzania danych, aby zinterpretować i przekształcić wygenerowaną odpowiedź w działanie aplikacji.

Kontrola i przewidywalność odpowiedzi

Rozwiązań NLP charakteryzują się większą przewidywalnością działania. W systemach takich jak Wit.ai intencje i akcje są definiowane w sposób jawny, co pozwala na pełną kontrolę nad procesem odpowiedzi. Modele generatywne, takie jak ChatGPT, generują odpowiedzi na podstawie wzorców w danych, co sprawia, że ich działanie może być mniej przewidywalne. W aplikacjach użytkowych, gdzie kluczowe jest zaufanie użytkownika i spójność działania, deterministyczne podejście NLP jest bardziej pożądane.

2.3 Obecne rozwiązania

W ramach analizy biznesowej przebadano aplikacje posiadające następujące funkcje:

- Interfejs służący do nawigacji po sklepie
- Baza danych ze sklepami, wraz z ich lokalizacją i rozkładem produktów
- Asystent AI pomagający w obsłudze aplikacji
- Interfejs głosowy pozwalający na obsługę aplikacji przez osobę niedowidzącą
- System zgrywania koszyka do kodu QR w celu szybszego zakończenia zakupów

Analiza miała na celu rozpoznanie dostępnych rozwiązań na rynku, które mogłyby być inspiracją dla aplikacji będącej tematem pracy.

W tabeli 2.1 przedstawiono znalezione rozwiązania i ich pokrycie funkcjonalne.

Aplikacja	Nawigacja po sklepie	Baza danych sklepów	Asystent AI	Interfejs głosowy	Koszyk QR
Walmart App	✓	✓			
Target App	✓	✓			
Amazon Alexa			✓	✓	
Carrefour App					✓
IKEA Place	✓				
Instacart		✓			
Google Shopping			✓		
Shopper	✓	✓	✓	✓	✓

Tabela 2.1: Porównanie funkcjonalności aplikacji zakupowych

2.3.1 Walmart App

Aplikacja Walmart App jest najbliższym rozwiązaniem do aplikacji Shopper. Posiada ona interfejs służący do nawigacji po sklepie, bazę danych ze sklepami, wraz z ich lokalizacją i rozkładem produktów. Niestety, nie posiada ona asystenta AI, interfejsu głosowego ani systemu zgrywania koszyka do kodu QR.

2.3.2 Asystenci głosowi

Asystenci głosowi, takie jak Amazon Alexa czy Google Shopping, są rozwiązaniem dla osób, które chcą zrobić zakupy bez użycia rąk. Niestety, nie posiadają one interfejsu służącego do nawigacji po sklepie, bazy danych ze sklepami, wraz z ich lokalizacją i rozkładem produktów ani systemu zgrywania koszyka do kodu QR.

2.3.3 Wnioski

Jak widać, żadna z aplikacji nie spełnia wszystkich wymagań postawionych przed aplikacją. Ponadto, wszystkie aplikacje poza Instacart i Google Assistant, są rozwiązaniem dla jednego sklepu. Aplikacja Shopper ma na celu nie tylko implementację wszystkich wymagań w ramach jednego projektu, ale też stworzenie aplikacji, która będzie działać w każdym sklepie.

Rozdział 3

PROPOZYCJA ROZWIĄZANIA PROBLEMU

W niniejszym rozdziale przedstawiono propozycję rozwiązania problemu zidentyfikowanego w poprzednich częściach pracy, uwzględniając kluczowe wymagania i założenia projektowe. Proponowanym rozwiązaniem jest kompletna aplikacja Shopper ułatwiająca realizację zakupów w wcześniej przystosowanym środowisku sklepowym. W ramach przygotowania sklepu do współpracy z aplikacją należy wykonać następujące czynności:

- Zainstalować nadajniki BLE na terenie sklepu.
- Zimportować asortyment sklepu do bazy danych aplikacji
- Zmodyfikować oprogramowanie kas samoobsługowych w celu korzystania z szybkiego kasowania (ang. *express checkout*).

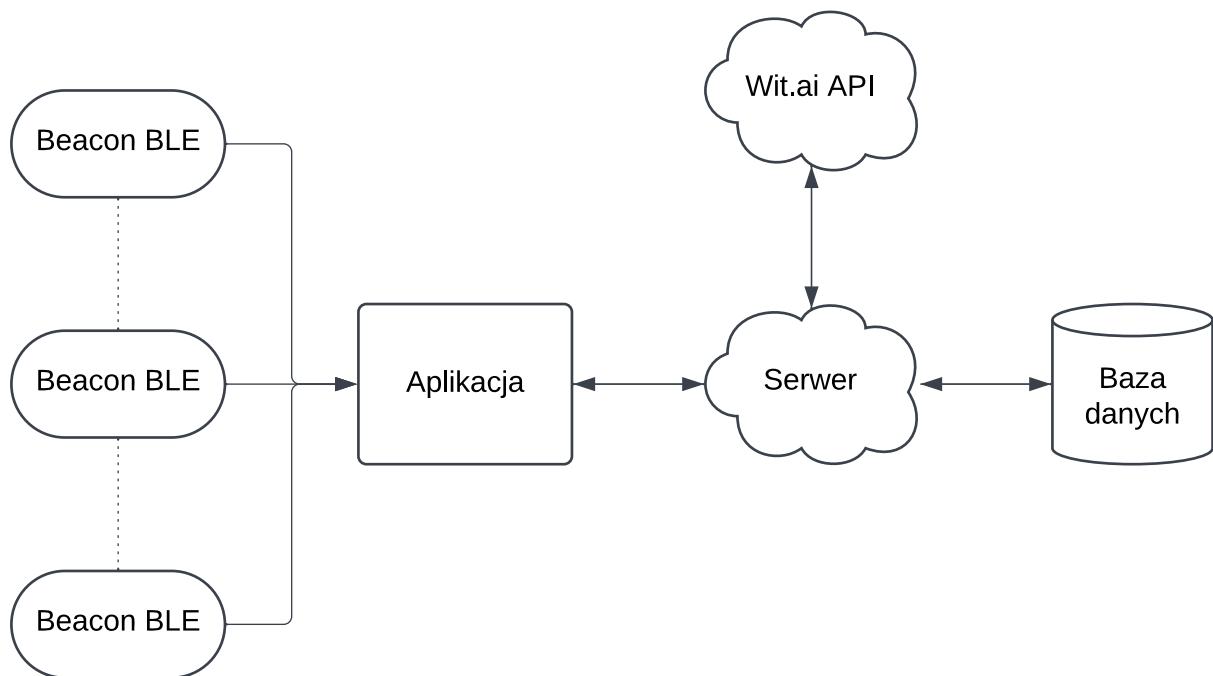
Po wykonaniu powyższych kroków i przetestowaniu aplikacji w nowym środowisku można rozpocząć testy publiczne.

Rozdział 4

OPIS ROZWIĄZANIA

4.1 Architektura systemu

System składa się z pięciu głównych komponentów: nadajników BLE (ang. *BLE beacon*), aplikacji mobilnej, systemu Wit.ai, serwera oraz bazy danych. Grafikę przedstawiającą architekturę systemu można zobaczyć na rysunku 4.1.



Rysunek 4.1: Architektura systemu.

Aplikacja mobilna jest odpowiedzialna za odbieranie oraz przetwarzanie sygnału z nadajników. Jej zadaniem jest również interakcja z użytkownikiem i wysyłanie zapytań do serwera. Serwer przetwarza żądania użytkownika, wysyła zapytania do API (ang. *Application Programming Interface*) serwisu Wit.ai, oraz komunikuje się z bazą danych. Baza danych przechowuje dane i modyfikuje lub udostępnia je na żądanie serwera. Komunikacja między aplikacją mobilną a serwerem odbywa się za pomocą protokołu HTTP. Serwer jest odpowiedzialny za przetwarzanie żądań użytkownika, a także za komunikację z bazą danych. Baza danych przechowuje dane o produktach, użytkownikach, koszykach, sklepach itp.

4.2 Baza danych

4.2.1 Opis bazy danych

Baza danych została zaimplementowana w PostgreSQL. Wybór tej bazy danych wynika z jej wszechstronności, wydajności oraz możliwości łatwego skalowania.

PostgreSQL to obiektowo-relacyjny system zarządzania bazami danych (ang. ORDBMS *Object-Relational Database Management System*), którego rozwój rozpoczął się już w 1977 roku. Jego korzenie sięgają projektu o nazwie Ingres, realizowanego na Uniwersytecie Kalifornijskim w Berkeley. Uznawany jest za jeden z najbardziej zaawansowanych systemów baz danych o otwartym kodzie źródłowym na świecie. Oferuje wiele funkcji, które do tej pory były kojarzone głównie z komercyjnymi rozwiązaniami klasy enterprise. [Worsley and Drake(2002)]

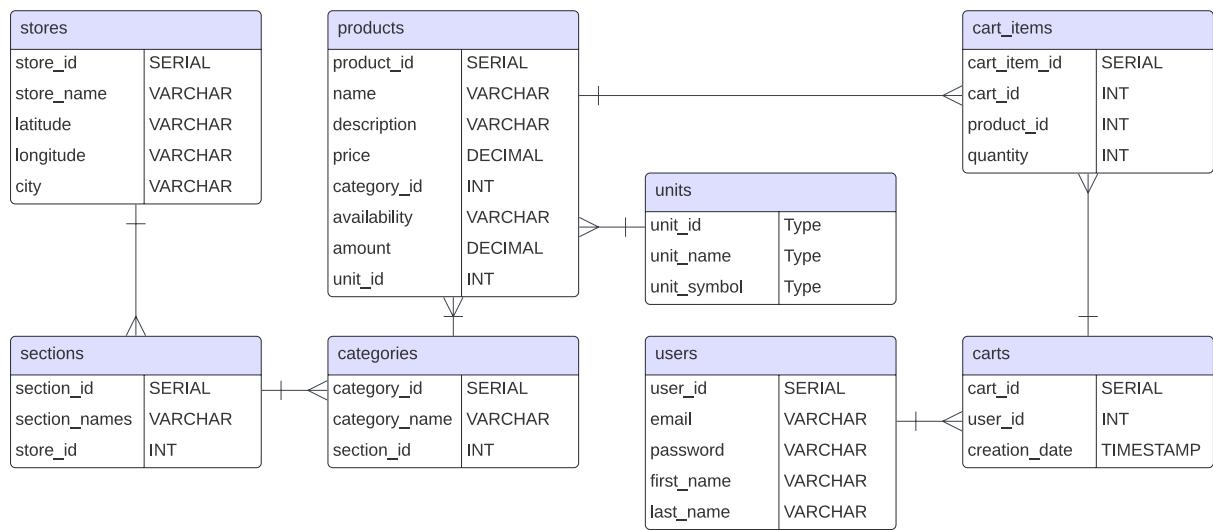
Baza danych przechowuje informacje o produktach, użytkownikach, koszykach oraz sklepach. Schemat bazy danych przedstawia rysunek 4.2.

Struktura bazy danych została zaprojektowana w sposób modularny, umożliwiając efektywne zarządzanie danymi dotyczącymi sklepów, użytkowników oraz produktów. Główną tabelą bazy danych jest tabela *stores*, która przechowuje informacje o sklepach, takie jak nazwa, współrzędne geograficzne oraz miasto. Związek tej tabeli z tabelą *sections* umożliwia podział sklepów na sekcje, które z kolei są przypisane do tabeli *categories*, zawierającej dane o kategoriach produktów.

Produkty są przechowywane w tabeli *products*, gdzie każdy rekord zawiera szczegóły takie jak nazwa, opis, cena, dostępność, ilość oraz jednostka miary, przechowywana w tabeli *units*. Relacje między tabelami *categories* i *products* pozwalają na przypisanie każdego produktu do konkretnej kategorii, co ułatwia organizację i wyszukiwanie danych.

Użytkownicy systemu są reprezentowani w tabeli *users*, gdzie zapisywane są ich dane personalne, takie jak imię, nazwisko, adres e-mail oraz zaszyfrowane hasło. Każdy użytkownik może posiadać wiele koszyków zakupowych, co jest odzwierciedlone w tabeli *carts*, przechowującej informacje o koszykach, takie jak data utworzenia i powiązanie z użytkownikiem. Szczegóły dotyczące zawartości koszyków są zapisane w tabeli *cart_items*, która łączy produkty z koszykami i zawiera informacje o liczbie sztuk danego produktu.

Relacje pomiędzy tabelami są realizowane za pomocą kluczy obcych, z zastosowaniem reguły ON DELETE CASCADE, co zapewnia integralność danych oraz automatyczne usuwanie powiązanych rekordów w przypadku usunięcia danych z tabel nadzorzących. Taka organizacja umożliwia łatwe skalowanie bazy danych oraz wspiera utrzymanie spójności danych w systemie.



Rysunek 4.2: Schemat bazy danych.

4.2.2 Szczegółowy opis tabel

Tabela stores

- *store_id* - SERIAL PRIMARY KEY: Unikalny identyfikator każdego sklepu.

- store_name - VARCHAR(255) NOT NULL: Nazwa sklepu.
- latitude - VARCHAR(255) NOT NULL: Szerokość geograficzna określająca położenie sklepu.
- longitude - VARCHAR(255) NOT NULL: Długość geograficzna określająca położenie sklepu.
- city - VARCHAR(255) NOT NULL: Miasto, w którym znajduje się sklep.

Tabela sections

- section_id - SERIAL PRIMARY KEY: Unikalny identyfikator sekcji sklepu.
- section_name - VARCHAR(255) NOT NULL: Nazwa sekcji w sklepie.
- store_id - INT REFERENCES stores(store_id) ON DELETE CASCADE: Klucz obcy wskazujący sklep, do którego należy sekcja.

Tabela categories

- category_id - SERIAL PRIMARY KEY: Unikalny identyfikator kategorii.
- category_name - VARCHAR(255) NOT NULL: Nazwa kategorii produktów.
- section_id - INT REFERENCES sections(section_id) ON DELETE CASCADE: Klucz obcy wskazujący sekcję, do której przypisana jest kategoria.

Tabela units

- unit_id - SERIAL PRIMARY KEY: Unikalny identyfikator jednostki miary.
- unit_name - VARCHAR(50) NOT NULL: Pełna nazwa jednostki miary (np. "kilogram").
- unit_symbol - VARCHAR(10) NOT NULL: Skrót jednostki miary (np. "kg").

Tabela products

- product_id - SERIAL PRIMARY KEY: Unikalny identyfikator produktu.
- name - VARCHAR(255) NOT NULL: Nazwa produktu.
- description - TEXT: Opis produktu.
- price - DECIMAL(10,2) NOT NULL: Cena produktu w formacie dziesiętnym (np. 123.45).
- category_id - INT REFERENCES categories(category_id) ON DELETE CASCADE: Klucz obcy wskazujący kategorię, do której należy produkt.
- availability - VARCHAR(50) NOT NULL: Status dostępności produktu (np. "w magazynie").
- amount - DECIMAL(10,2) NOT NULL: Ilość dostępna w magazynie.
- unit_id - INT REFERENCES units(unit_id) ON DELETE CASCADE: Klucz obcy wskazujący jednostkę miary produktu.

Tabela users

- user_id - SERIAL PRIMARY KEY: Unikalny identyfikator użytkownika.
- email - VARCHAR(255) UNIQUE NOT NULL: Adres e-mail użytkownika.
- password - VARCHAR(255) NOT NULL: Hasło użytkownika (w formie zaszyfrowanej).
- first_name - VARCHAR(50) NOT NULL: Imię użytkownika.
- last_name - VARCHAR(50) NOT NULL: Nazwisko użytkownika.

Tabela carts

- cart_id - SERIAL PRIMARY KEY: Unikalny identyfikator koszyka.
- user_id - INT REFERENCES users(user_id) ON DELETE CASCADE: Klucz obcy wskazujący użytkownika, do którego należy koszyk.
- creation_date - TIMESTAMP DEFAULT CURRENT_TIMESTAMP: Data i czas utworzenia koszyka.

Tabela cart_items

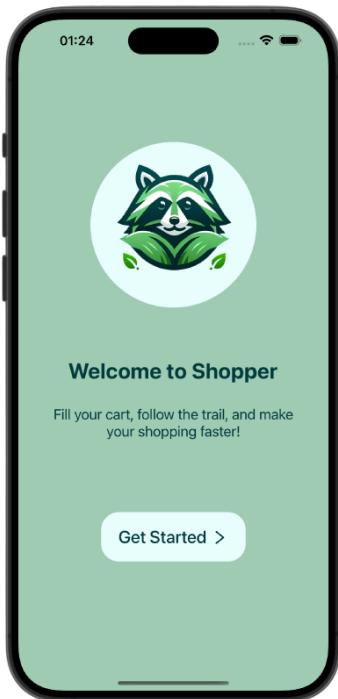
- cart_item_id - SERIAL PRIMARY KEY: Unikalny identyfikator pozycji w koszyku.
- cart_id - INT REFERENCES carts(cart_id) ON DELETE CASCADE: Klucz obcy wskazujący koszyk, do którego należy pozycja.
- product_id - INT REFERENCES products(product_id) ON DELETE CASCADE: Klucz obcy wskazujący produkt dodany do koszyka.
- quantity - INT NOT NULL: Liczba sztuk danego produktu w koszyku.

4.3 Interfejs użytkownika

4.3.1 Opis dostępnych widoków

Strona tytułowa

Strona tytułowa aplikacji pełni funkcję ekranu startowego, witając odbiorcę logiem oraz hasłem zachęcającym do korzystania z aplikacji. Widnieje na niej przycisk *Get Started*, który umożliwia przejście do kolejnych widoków. Jeśli eksploatautor był zalogowany w ciągu ostatnich 7 dni, aplikacja przekierowuje go do widoku logowania. W przeciwnym wypadku trafia on do ekranu profilu.

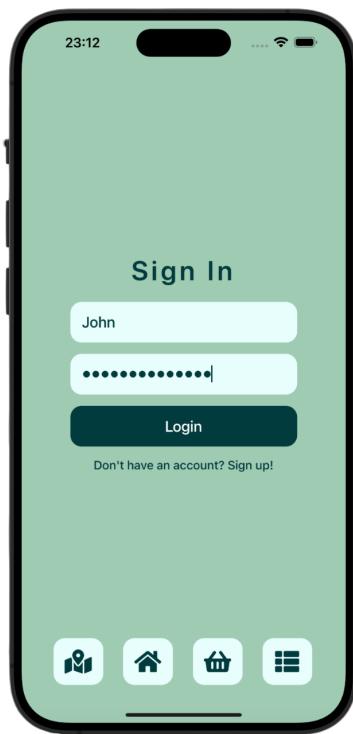


Całość utrzymana jest w przyjaznej stylistyce, z dominującym odcieniem zieleni oraz spójną paletą kolorów, wygenerowaną przez narzędzie DALL·E 3 od OpenAI.



Ekran logowania

Ekran logowania umożliwia odbiorcy uwierzytelnienie w aplikacji, poprzez wprowadzenie adresu e-mail oraz hasła. Jego głównym celem jest weryfikacja tożsamości eksplotatora oraz przekierowanie do dalszych widoków w zależności od wyniku logowania.



Górna część widoku stanowi nagłówek *Sign In*, który podkreśla cel ekranu. Tuż pod nim znajduje się formularz składający się z dwóch pól tekstowych: jednego przeznaczonego do wprowadzania adresu e-mail, a drugiego do hasła, z cechą ukrywania wprowadzanych znaków. Naciśnięcie przycisku *Login* uruchamia logikę, która weryfikuje tożsamość odbiorcy na podstawie danych wprowadzonych w polach tekstowych. W przypadku niepowodzenia gość otrzymuje odpowiedni komunikat, informujący o niepoprawnym e-mailu lub haśle. Dla nowych kupujących, widok zapewnia przejście do ekranu rejestracji, poprzez kliknięcie w link *Don't have an account? Sign up!*.

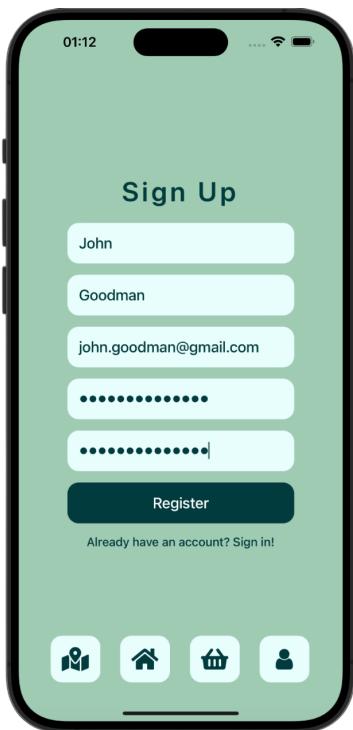
Dolną część ekranu zajmuje pasek nawigacyjny z ikonami, które przenoszą do kolejnych podstron: mapy sklepów, profilu odbiorcy, kategorii produktów wybranego sklepu oraz strony tytułowej. Tylko ta ostatnia jest dostępna dla niezalogowanych użytkowników.



Rejestracja użytkownika

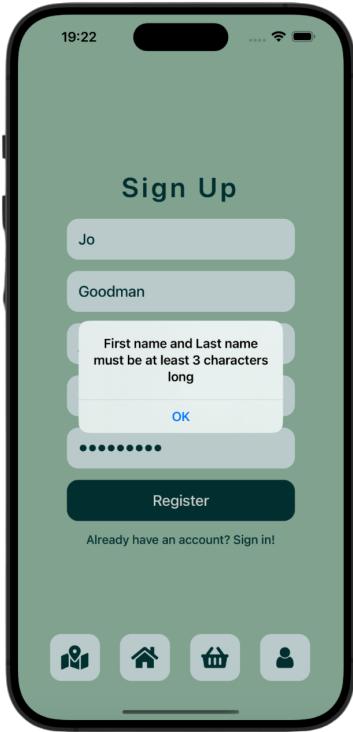
Ekran rejestracji umożliwia nowym odbiorcom założenie konta, co jest niezbędne do korzystania z funkcji wymagających uwierzytelnienia, takich jak dodawanie produktów do koszyka czy przeglądanie sklepów na mapie. Formularz rejestracyjny składa się z kilku pól:

- *Firstname* i *Lastname* – wymagane minimum trzech znaków, by dane były wystarczająco szczegółowe.
- *Email* – odbiorca podaje swój adres e-mail, który jest weryfikowany pod kątem poprawności formatu (obecność znaku @ oraz domeny).
- *Password* i *Repeat password* – hasło musi mieć co najmniej osiem znaków i być identyczne w obu polach. Dodatkowo, tekst wprowadzony w tych polach jest ukrywany, aby zapewnić prywatność.



Po wypełnieniu formularza, należy wcisnąć guzik *Register*, po którym odbywa się validacja danych. W przypadku spełnienia warunków wpisanych pól, nowy użytkownik, wraz z koszykiem jest rejestrowany w bazie danych. Po zakończeniu rejestracji następuje automatyczne zalogowanie, a imię oraz ID są zapisywane w lokalnej pamięci urządzenia, w bezpieczny sposób, aby dane nie trafiły w niepowołane ręce. Pozwala to na sprawne obsłużenie mechanizmu sesji oraz dostępu do koszyka jego właściciela. Następuje po tym przekierowanie do ekranu kategorii produktów wybranego sklepu. Jeśli dane są nieprawidłowe, wyświetlane są stosowne komunikaty, informujące o błędach, takich jak niewłaściwy format e-maila, zbyt krótkie hasło czy jego niezgodność. Osoby posiadające już konto mogą skorzystać z linku *Already have an account? Sign in!*, znajdującego się pod formularzem, aby przejść do ekranu logowania.

Dolną część ekranu zajmuje pasek nawigacyjny, który zawiera przyciski prowadzące do kolejnych sekcji aplikacji. Pierwszy z nich przekierowuje do widoku mapy sklepów. Kolejny przekierowuje na stronę główną – jest to jedyny dostępny dla niezalogowanych odbiorców. Trzeci zawiera ikonę koszyka, w którym użytkownicy mogą przeglądać swoje produkty, a czwarty prowadzi do profilu odbiorcy, gdzie można zarządzać danymi konta.



Kategorie produktów

Ekran kategorii stanowi pierwszy krok do przeglądania dostępnych produktów w wybranym sklepie. Centralnym elementem tego widoku jest siatka kategorii, umożliwiająca użytkownikowi intuicyjne poruszanie się po różnych grupach produktów.



W górnej części widoku znajduje się pasek wyszukiwania, składający się z pola tekstowego oraz ikony lupy, umożliwiający szybkie filtrowanie kategorii na podstawie wprowadzonego tekstu. Wprowadzenie frazy w pole wyszukiwania automatycznie ogranicza widoczne wyniki, wyświetlając jedynie pasujące grupy produktów. Każdy kafelek siatki zawiera nazwę kategorii, a jego kliknięcie przekierowuje

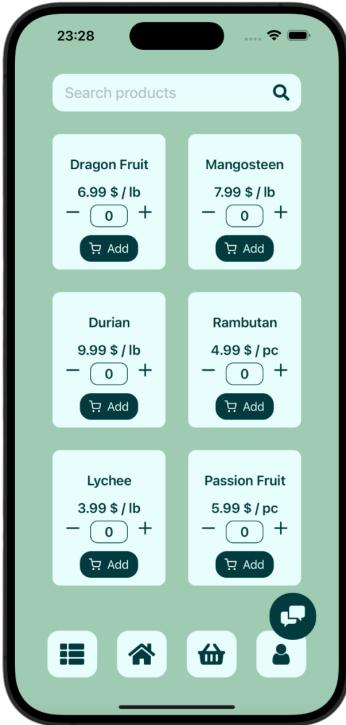
użytkownika do widoku produktów należących do wybranej kategorii. Siatkę można przewijać w pionie, aby odkrywać kolejne kafelki dostępne z listy.

Dolną część ekranu zajmuje pasek nawigacyjny, który umożliwia szybki dostęp do innych kluczowych widoków aplikacji, takich jak mapa sklepów, profil użytkownika, koszyk oraz strona tytułowa. Widok zawiera również przeciągany dymek czatu, który pozwala na szybki kontakt z obsługą klienta. Jest on towarzyszem większości ekranów aplikacji, przez co został mu poświęcony osobny artykuł w sekcji o numerze x.x.x.



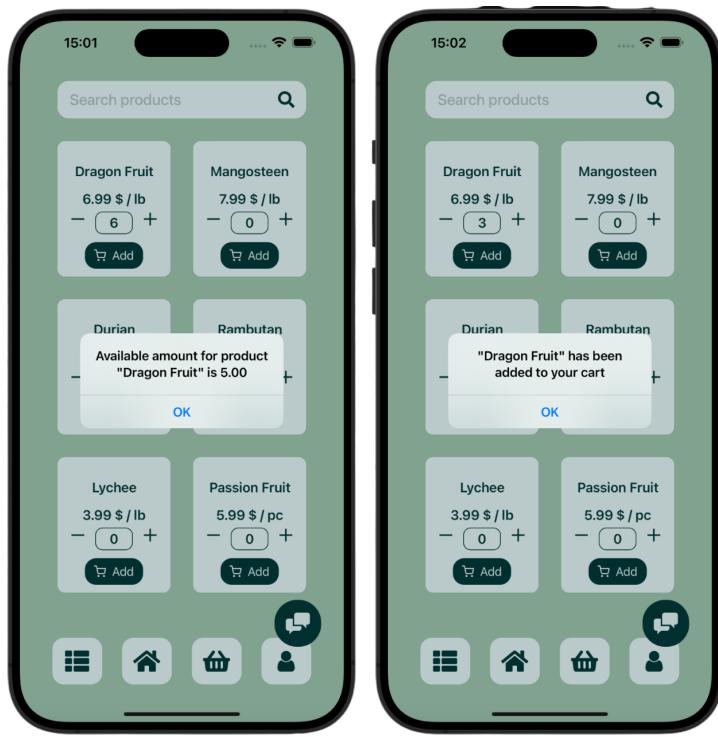
Ekrany produktów

Ekrany produktów pozwala użytkownikowi na przeglądanie i dodawanie do koszyka artykułów należących do wybranej kategorii.



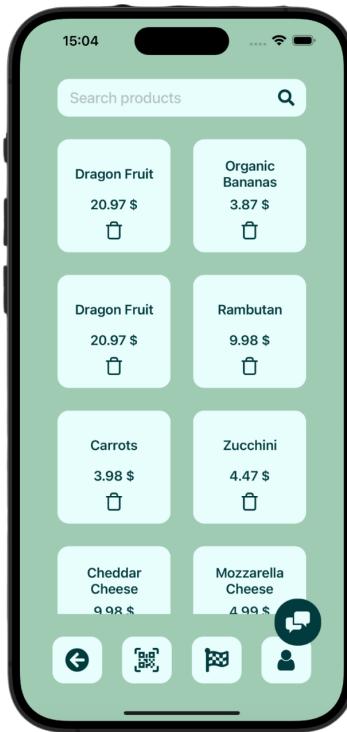
Na samej górze widoku znajduje się pasek wyszukiwania, umożliwiający szybkie filtrowanie produktów po ich nazwie. Pasek zawiera pole tekstowe oraz ikonę lupy. Wprowadzenie tekstu w tym polu automatycznie zawęża widok, prezentując jedynie pasujące wyniki. Poniżej znajduje się przewijalna siatka produktów, w której każdy kafelek zawiera nazwę produktu, cenę oraz symbol jednostki miary. Dodatkowo dla każdego produktu dostępne są przyciski + i -, umożliwiające zwiększenie lub zmniejszanie ilości produktu, jaką użytkownik chce dodać do koszyka. Wartość wprowadzana przez kupującego jest automatycznie walidowana. Walidacja polega na sprawdzeniu, czy wprowadzony symbol jest liczbą całkowitą. Wartości mniejsze od zera nie są akceptowane. Pod każdym kafelkiem produktu znajduje się przycisk *Add*, który dodaje wybraną ilość artykułu do koszyka gościa. Po naciśnięciu przycisku aplikacja odpowiednio weryfikuje możliwość dokonania zakupu poprzez sprawdzenie ilości produktu w magazynie. W przypadku błędnych danych, takich jak ilość większa niż dostępna, użytkownik otrzymuje odpowiedni komunikat w postaci alertu.

Dół ekranu zdobi pasek nawigacyjny, który umożliwia powrót do ekranu kategorii, sprawdzenie stanu koszyka, przejście do profilu użytkownika czy strony głównej.



Koszyk użytkownika

Ekran koszyka pozwala użytkownikowi zarządzać produktami dodanymi do koszyka oraz przygotować się do finalizacji zakupów.

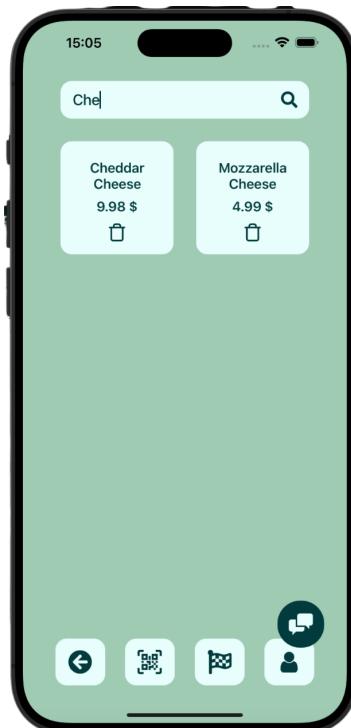


W górnej części ekranu znajduje się pasek wyszukiwania, umożliwiający filtrowanie produktów według nazwy. Użytkownik może wprowadzić frazę w polu tekstowym, aby zawęzić listę wyświetlanego produktów, co znacznie ułatwia nawigację w przypadku dużej liczby elementów. Listę produktów w koszyku prezentuje przewijalna siatka, mieszcząca się pod paskiem wyszukiwania. Każdy element listy produktów w koszyku zawiera następujące informacje:

- Nazwę produktu.
- Cenę jednostkową.
- Ilość produktu w koszyku, wyrażoną w odpowiedniej jednostce miary (np. sztuki, kilogramy).
- Łączną cenę dla danej pozycji, obliczoną jako iloczyn ceny jednostkowej i ilości.

W przypadku chęci usunięcia produktu z koszyka, użytkownik posiada możliwość kliknięcia na ikonę kosza na śmieci danego produktu. Aplikacja automatycznie aktualizuje widok koszyka po każdej zmianie.

Dolną część ekranu zajmuje pasek nawigacyjny, który umożliwia szybkie przejście do kolejnych widoków aplikacji. Należą do nich: generowanie kodu qr, mapa nawigująca po sklepie, profil użytkownika oraz powrót do poprzedniego ekranu.



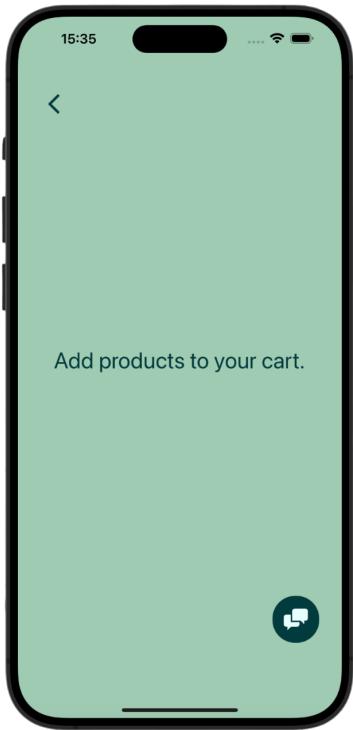
Generowanie kodu qr

Po przejściu na ten ekran, pobierana jest lista produktów z koszyka danego kupującego. Dane na temat produktów zostają zawarte w wygenerowanym kodzie, który wyświetla się na środku ekranu po załadowaniu listy. W przypadku, gdy koszyk jest pusty, wyświetlany jest komunikat zachęcający do jego uzupełnienia.



Użytkownik ma możliwość zeskanowania kodu qr za pomocą czytnika danego sklepu, co pozwala na szybkie zinterowanie danych z aplikacją kasą samoobsługowej i nabicia zakupów na paragon. Za integrację kodu ze swoją aplikacją odpowiedzialny jest sam sklep.

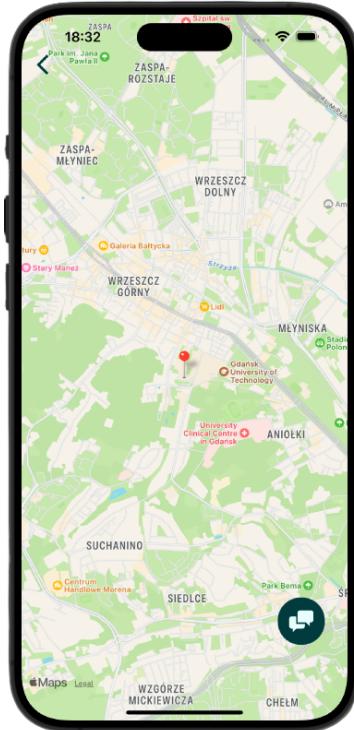
Aby wyjść z ekranu, należy nacisnąć symbol ptaszka, który przenosi użytkownika z powrotem do ekranu produktów.



Mapa sklepów

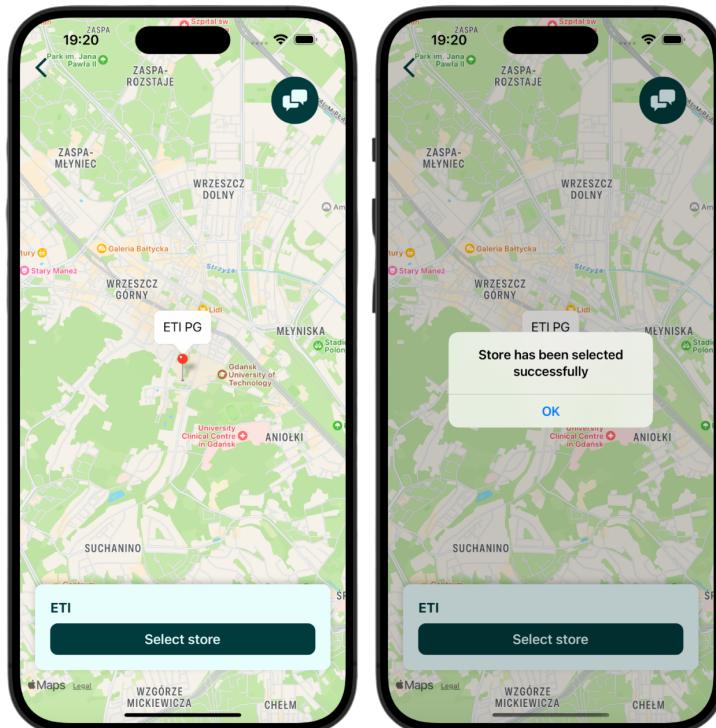
Ekran wyboru sklepu pozwala użytkownikowi w intuicyjny sposób wybrać lokalizację sklepu, w którym planuje zrobić zakupy. Głównym elementem, pokrywającym cały widok jest interaktywna mapa, na której

zaznaczone są wszystkie dostępne sklepy w formie markerów. Użytkownik może dowolnie przesuwać mapę oraz przybliżać i oddalać widok, aby lepiej zapoznać się z rozmieszczeniem sklepów.



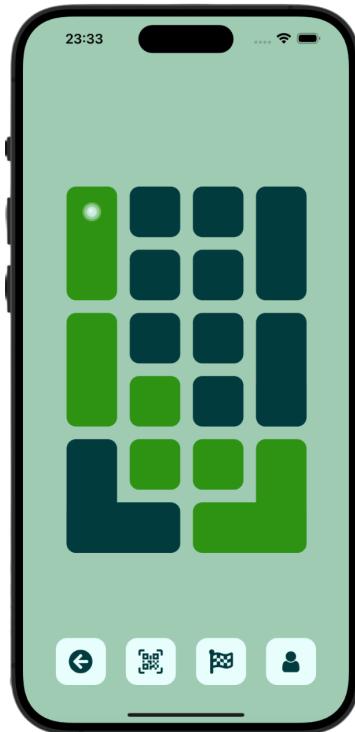
Punkt początkowy został ustawiony na sklep *ETI PG* na potrzebę prezentacji. Kliknięcie na marker sklepu wyświetla szczegóły wybranej lokalizacji w dolnym panelu ekranu. Panel ten prezentuje nazwę sklepu oraz przycisk *Select store*, który pozwala na potwierdzenie wyboru. Wybór sklepu jest sygnalizowany wyświetlaniem komunikatu informującego o poprawnym zapisaniu decyzji. Dzięki temu użytkownik ma pewność, że jego wybór został zarejestrowany i aplikacja poprawnie zapisała ID sklepu, z którego należy pobrać kategorie i produkty.

W górnym lewym rogu ekranu znajduje się przycisk powrotu, który umożliwia szybkie przejście do poprzedniego widoku.



Ekran nawigacji

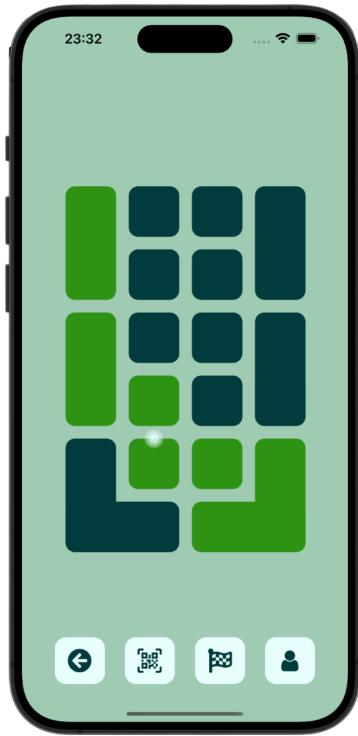
Ekran ten pozwala użytkownikowi na łatwe nawigowanie po sklepie, w celu zrealizowania zakupów w jak najkrótszym czasie. Jest to kluczowy widok, który odpowiada za podświetlenie klientowi trasy, jaką należy pokonać, aby uzyskać ten efekt.



Ekran w znacznej większości składa się mapy sklepu, wygenerowanej ręcznie za pomocą grafiki SVG. Grafika ta, jest tworzona po wcześniejszej konsultacji ze sklepem. Podczas wejścia na ekran, aplikacja pobiera listę produktów z koszyka i na jej podstawie, wyznacza najkrótszą trasę, w postaci sektorów, które należy przebyć. Kolor trasy zmienia się na zielony, podczas gdy reszta sekcji pozostaje morska. Następnie za pomocą rozmieszczonych nadajników BLE, zostaje określone położenie użytkownika, które jest aktualizowane w czasie rzeczywistym, w postaci pulsującej, białej kropki.

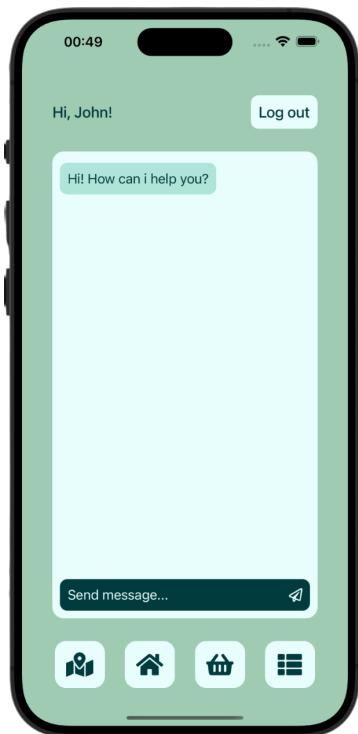
Nawigacja w każdej chwili może zostać przerwana, chociażby poprzez powrót do koszyka, aby dodać kolejne produkty. W takim przypadku, po powrocie na ekran nawigacji, kupujący widzi nowo wygenerowaną trasę. Sam algorytm wyznaczania najkrótszej trasy, obsługa nadajników i pozycjonowanie eksploataatora zostaną omówione w osobnym rozdziale.

Widok ten zawiera również pasek nawigacyjny, umożliwiający przejście do innych widoków, takich jak koszyk, profil kupującego czy generowanie kodu qr.



Profil użytkownika

Ekran użytkownika umożliwia interakcję z chatbotem oraz zarządzanie kontem użytkownika, w tym możliwość wylogowania się z aplikacji.



Na górze ekranu znajduje się powitanie użytkownika, wyświetlające jego imię, które jest pobierane z lokalnej pamięci aplikacji, aktualizowanej podczas poprawnego logowania. Obok powitania znajduje się przycisk *Log out*, umożliwiający wylogowanie się z aplikacji i przekierowanie użytkownika na stronę główną. Wraz z kliknięciem tego przycisku czyszczona jest pamięć podręczna aplikacji, co skutkuje przerwaniem sesji i koniecznością ponownego zalogowania.

Główna część ekranu zajmuje chatbot, który pozwala na interakcję z użytkownikiem. Działa on w identyczny sposób, co wspomniany we wcześniejszych sekcjach dymek chatu, stąd zostanie on omówiony w osobnym rozdziale.

Dolną część ekranu zajmuje pasek nawigacyjny, który umożliwia przejście do innych sekcji aplikacji. Należą do nich: mapa sklepów, kategorie produktów, koszyk oraz strona tytułowa.

4.4 Serwer aplikacji

Serwer jest odpowiedzialny za przetwarzanie żądań użytkownika, a także za komunikację z bazą danych. Został zaimplementowany w języku JavaScript przy użyciu platformy Node.js oraz struktury (ang. framework) Express.js. Node.js pozwala na uruchamianie JavaScript po stronie serwera, co umożliwia tworzenie wydajnych i skalowalnych aplikacji. Express.js, będący minimalistycznym frameworkiem działającym na Node.js, upraszcza proces budowy aplikacji internetowych. Serwer nasłuchiwa na zapytania HTTP, przetwarza je i zwraca odpowiedź, a do komunikacji z bazą danych PostgreSQL wykorzystuje odpowiednie moduły Node.js.

4.4.1 JavaScript

JavaScript jest podstawowym językiem programowania w sieci Web. Zdecydowana większość współczesnych stron internetowych wykorzystuje JavaScript, a wszystkie nowoczesne przeglądarki internetowe — zarówno na komputerach stacjonarnych, konsolach do gier, tabletach, jak i smartfonach — posiadają wbudowane interpreterы tego języka. Dzięki temu JavaScript stał się najbardziej wszechobecnym językiem programowania w historii. Wraz z HTML, odpowiadającym za treść stron, oraz CSS, definiującym ich wygląd, JavaScript stanowi podstawowy zestaw technologii, które każdy programista webowy musi opanować, aby określić zachowanie stron internetowych. [Flanagan(2011)]

4.4.2 Node.js

Node.js pozwala programistom wykorzystywanie JavaScript po stronie serwera, co pozwala na tworzenie aplikacji full-stack przy użyciu jednego języka. Dzięki architekturze wspierającej asynchroniczne i nieblokujące operacje I/O świetnie sprawdza się w obsłudze wielu jednoczesnych połączeń. [Peters(2017)]

Dzięki architekturze opartej na zdarzeniach oraz jednowątkowemu modelowi działania, Node.js idealnie nadaje się do tworzenia aplikacji czasu rzeczywistego, takich jak czaty, narzędzia do współpracy czy usługi streamingowe. [Peters(2017)]

4.4.3 Express.js

Express.js to minimalistyczny framework dla Node.js, który pozwala na szybkie tworzenie aplikacji internetowych. Dzięki swojej prostocie i elastyczności jest jednym z najpopularniejszych frameworków dla Node.js.

Framework Express.js pozwala dynamiczny routing, pozwalając programistom na definiowanie wzorców URL i przypisywanie ich do określonej logiki aplikacji. Taka elastyczność ułatwia zarządzanie złożonymi strukturami aplikacji poprzez wiązanie punktów końcowych z odpowiednimi kontrolerami. [Peters(2017)]

Rozdział 5

OPIS TECHNICZNY

5.1 Aplikacja wit.ai

W ramach projektu została stworzona aplikacja w systemie *wit.ai*. Wit.ai to platforma do tworzenia interfejsów interaktywnych, która pozwala na budowanie aplikacji, które rozumieją naturalny język. Wit.ai pozwala na tworzenie modeli językowych, które są w stanie rozpoznawać intencje użytkownika na podstawie zdefiniowanych przez programistę fraz. Aplikacja ta jest wykorzystywana w projekcie do rozpoznawania intencji użytkownika na podstawie zdefiniowanych przez programistę fraz.

5.1.1 Intencje i encje

W celu nauczenia modelu językowego aplikacji *wit.ai*, należy zdefiniować intencje (ang. *intents*), które mają być rozpoznawane przez aplikację. Intencje to frazy, które użytkownik może napisać, a które mają być zrozumiane przez aplikację. Każda intencja może zawierać wiele przykładów fraz, które są z nią związane. Przykładowe intencje, które zostały zdefiniowane w aplikacji *wit.ai* to:

- *add_product_to_cart* - intencja dodania produktu do koszyka,
- *remove_product_from_cart* - intencja usunięcia produktu z koszyka,
- *check_cart* - intencja sprawdzenia zawartości koszyka,
- *check_item_prices* - intencja sprawdzenia cen produktów,
- *check_item_price_in_store* - intencja sprawdzenia ceny produktu w sklepie

Poza intencjami, w aplikacji *wit.ai* definiuje się również encje (ang. *entities*). Encje to frazy, które mają być rozpoznawane przez aplikację jako konkretne wartości. Encje pomagają również w wykryciu intencji użytkownika. Przykładowe encje, które zostały zdefiniowane w aplikacji *wit.ai* to:

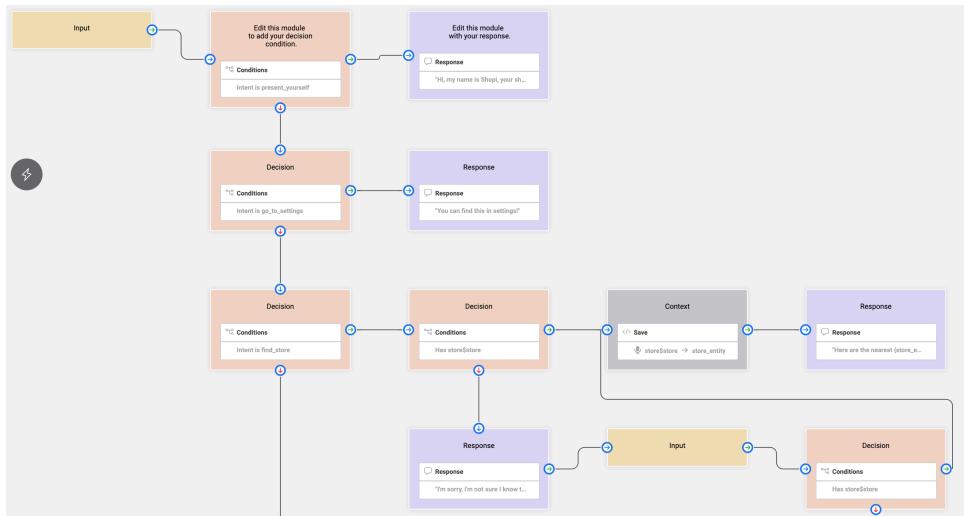
- *product* - encja reprezentująca nazwę produktu,
- *store* - encja reprezentująca nazwę sklepu,
- *view* - encja reprezentująca widok w aplikacji,
- *category* - encja reprezentująca nazwę kategorii produktów

Po zdefiniowaniu intencji i encji, aplikacja *wit.ai* pozwala na trenowanie modelu językowego. Trenowanie modelu polega na przesłaniu do aplikacji *wit.ai* przykładów fraz, które mają być rozpoznawane przez aplikację. Po przesłaniu przykładów, aplikacja *wit.ai* trenuje model językowy. Po wstępny treningu, aplikacja stara się sama sugerować intencje w procesie trenowania modelu. Pozwala to na sprawdzanie w czasie rzeczywistym, czy model językowy poprawnie rozpoznaje encje i intencje. Przykładowe frazy wykorzystane w procesie trenowania modelu to:

- *Where to buy apples*
- *Remove cheese from cart*,
- *My app is stuck on loading*,
- *Is there dairy in castorama?*

5.1.2 Kreator

Wytrenowany model należy zaprogramować. W tym celu wit.ai udostępnia kreator (ang. *composer*), który pozwala na zdefiniowanie akcji, które mają być wykonywane po rozpoznaniu intencji przez aplikację. Przykładowe akcje, które zostały zdefiniowane w aplikacji wit.ai przedstawiono na rysunku 5.1.



Rysunek 5.1: Kreator aplikacji wit.ai

Akcje

W ramach kreatora dostępne są 4 moduły blokowe definiujące akcje. Są to:

- *Decision* - moduł decydujący o dalszym przebiegu akcji,
- *Context* - moduł przechowujący kontekst akcji,
- *Input* - moduł pobierający dane wejściowe,
- *Response* - moduł generujący odpowiedź.

Decision

Moduł *Decision* pozwala na zdefiniowanie warunków, które muszą być spełnione, aby akcja mogła zostać wykonana. Dostępne są poniższe warunki:

- *Intent* - sprawdza, czy intencja użytkownika jest zgodna z zdefiniowaną intencją,
- *Entity* - sprawdza, czy encja użytkownika jest zgodna z zdefiniowaną encją,
- *Context* - sprawdza, czy kontekst akcji jest zgodny z zdefiniowanym kontekstem,
- *Trait* - sprawdza, czy cecha akcji jest zgodna z zdefiniowaną cechą.
- *Not/And/Or* - służy do łączenia warunków.

Context

Moduł *Context* pozwala na zdefiniowanie kontekstu akcji. Kontekst to zmienna, która przechowuje informacje o stanie akcji. Kontekst może być wykorzystywany w kolejnych akcjach. W ramach tego modułu można wykonać 4 akcje:

- *Set* - ustawia wartość kontekstu,
- *Save* - zapisuje rozpoznaną encję do kontekstu,
- *Copy* - kopiuje wskazaną wartość kontekstu,
- *Clear* - czyści kontekst.

Input

Moduł *Input* pozwala na pobranie danych wejściowych. Jest on wykorzystywany na początku kreatora, w celu przyjęcia wiadomości od użytkownika. Można go również użyć do uzyskania dodatkowych informacji od użytkownika.

Response

Moduł *Response* pozwala na zdefiniowanie odpowiedzi, która ma zostać zwrocona do użytkownika. W odpowiedzi można wykorzystać zmienne zdefiniowane w kontekście akcji. Można zwrócić tekst, obraz, dźwięk, link, czy dowolny inny format. Oprócz tego, można również zwrócić nazwę funkcji, która ma zostać wykonana po zakończeniu akcji.

5.1.3 Testowanie i publikacja

Po zdefiniowaniu akcji, aplikacja wit.ai pozwala na przetestowanie modelu językowego. W tym celu można wpisać dowolną frazę, a aplikacja zwróci intencję oraz encje, które rozpoznała. Po zakończeniu testowania, model językowy można opublikować. Po opublikowaniu modelu, aplikacja wit.ai generuje token, który pozwala na integrację modelu z dowolną aplikacją. Token ten jest wykorzystywany w aplikacji mobilnej, aby komunikować się z modelem językowym.

5.1.4 Integracja z aplikacją mobilną

Aplikacja wit.ai pozwala na integrację z dowolną aplikacją mobilną. W tym celu należy wygenerować token, który pozwala na komunikację z modelem językowym. Token ten jest wykorzystywany w aplikacji mobilnej, aby wysyłać zapytania do modelu językowego. Po wysłaniu zapytania, model językowy zwraca intencję oraz encje, które rozpoznał. Oprócz tego, w określonych przypadkach zwracan jest też nazwa funkcji wraz z argumentami, którą należy wywołać po otrzymaniu odpowiedzi. Aplikacja mobilna na podstawie tych danych wykonuje odpowiednie akcje i odpowiada użytkownikowi w czacie.

5.1.5 HTTP API

Wit.ai udostępnia wiele funkcji za pomocą HTTP API. API to pozwala na integrację modelu językowego z dowolną aplikacją. API pozwala na trenowanie, testowanie, publikowanie, oraz integrację modelu z aplikacją mobilną. Do obsługi czatu w aplikacji wykorzystano metodę POST /event. Metoda ta zwraca zaktualizowaną mapę kontekstu po wykonaniu wdrożonego przepływu Composer dla podanej mapy kontekstu oraz opcjonalnego zapytania użytkownika. Może również zwrócić treść odpowiedzi przeznaczoną dla użytkownika oraz wskazanie akcji do wykonania. [Wit.ai(2024)] Na listingu 5.2 przedstawiono przykładowe zapytanie HTTP API. Na listingu 5.1.5 przedstawiono przykładową odpowiedź HTTP API.

Listing 5.1: Przykładowe zapytanie HTTP API

```
1 curl -XPOST \
2 -H 'Authorization: Bearer $TOKEN' \
3 'https://api.wit.ai/event?v=20241211&session_id=prod5i&tag=1.0&context_map=%7B
4   %7D' \
-d '{"type": "message", "message": "where is the nearest lidl?"}'
```

Listing 5.2: Przykładowe zapytanie HTTP API

```
1 {
2     "context_map": {
3         "store_entity": [
4             {
5                 "body": "lidl",
6                 "confidence": 1,
7                 "end": 25,
8                 "entities": {},
9                 "id": "955144219872453",
10                "name": "store",
11                "role": "store",
12                "start": 21,
13                "type": "value",
14                "value": "Lidl"
15            }
16        ]
17    },
18    "is_final": true,
19    "response": {
20        "text": "Here\u202e\u202e\u202e the\u202e\u202e\u202e nearest\u202e\u202e\u202e Lidl\u202e\u202e\u202e stores"
21    },
22    "type": "FULL_COMPOSER"
23 }
```

W odpowiedzi można znaleźć następujące pola:

- *context_map* - mapa kontekstu,
- *is_final* - flaga określająca, czy odpowiedź jest ostateczna,
- *response* - odpowiedź dla użytkownika,
- *type* - typ odpowiedzi.

W polu *response*, może znaleźć się jeszcze pole *action* które określa jaką akcję należy wykonać po otrzymaniu odpowiedzi.

5.2 Serwer

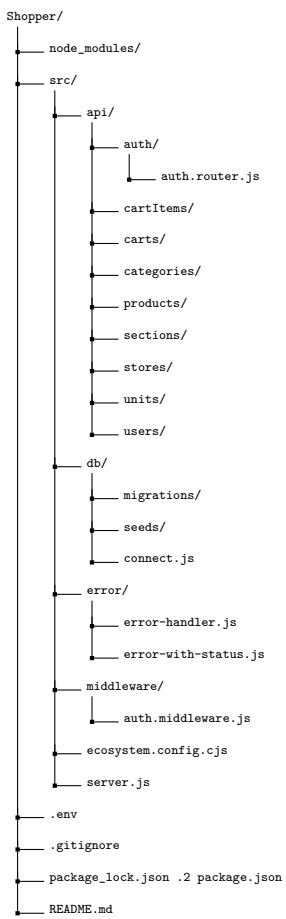
5.2.1 Struktura serwera

Na serwerze zaimplementowano kilka modułów, które są odpowiedzialne za przetwarzanie żądań użytkownika. Każdy moduł odpowiada za obsługę jednego zasobu, takiego jak użytkownik, produkt, czy koszyk. Każdy moduł składa się z trzech plików:

- *router* - plik zawierający definicję ścieżek API,
- *controller* - plik zawierający logikę przetwarzania żądań,
- *service* - plik zawierający logikę dostępu do bazy danych.

Oprócz modułów, na serwerze zaimplementowano również oprogramowanie pośredniczące (ang. *middleware*), które jest odpowiedzialne za przetwarzanie żądań przed przekazaniem ich do modułów. Middleware w tym przypadku jest wykorzystywane do autoryzacji użytkownika. Poza wyżej wymienionymi, serwer zawiera również kod potrzebny do migracji lub ponownego postawienia bazy danych.

Drzewo katalogów



W celu przejrzystego przedstawienia struktury serwera, powyższe drzewo katalogów nie przedstawia w plików każdego modułu. Aby dochować kompletości dokumentacji, należy uzupełnić, że nazewnictwo plików w każdym z modułów jest zgodne z poniższym wzorcem.

```
{nazwamodułu}/
├── {nazwamodułu}.router.js
├── {nazwamodułu}.controller.js
└── {nazwamodułu}.service.js
```

Router

Router jest odpowiedzialny za definiowanie ścieżek API. Każda ścieżka API odpowiada jednej akcji, która ma zostać wykonana. Router przekazuje żądanie do kontrolera, który jest odpowiedzialny za przetworzenie żądania. Przykładowa definicja ścieżki API w pliku router przedstawiona jest na listingu 5.3.

Listing 5.3: Przykładowa definicja ścieżki API

```
1 import { Router } from 'express';
2 import { userController } from './user.controller.js';
3 import { authMiddleware } from '../../../../../middleware/auth.middleware.js';
4
5 export const userRouter = Router();
6
7 // Apply authMiddleware to all user routes
8 userRouter.use(authMiddleware);
9
10 // Protected user routes
11 userRouter.get('/', userController.getAll);
12 userRouter.get('/:id', userController.getById);
13 userRouter.post('/', userController.create);
14 userRouter.put('/:id', userController.update);
15 userRouter.delete('/:id', userController.delete);
```

Powyższy przykład przedstawia definicję ścieżki API, która jest odpowiedzialna za obsługę żądań związanych z użytkownikami. Każda ścieżka API odpowiada jednej akcji, która ma zostać wykonana. W przypadku powodzenia, router przekazuje żądanie do kontrolera. W przypadku niepowodzenia, router przekazuje błąd do middleware, który jest odpowiedzialny za obsługę błędów.

Controller

Kontroler (ang. *Controller*) jest odpowiedzialny za przetwarzanie żądań. Każda metoda kontrolera odpowiada jednej akcji, która ma zostać wykonana. Kontroler przekazuje żądanie do serwisu, który jest odpowiedzialny za dostęp do bazy danych. Przykładowa definicja kontrolera przedstawiona jest na listingu 5.4.

Listing 5.4: Przykładowa definicja kontrolera

```
1 import { userService } from './user.service.js';
2
3 export const userController = {
4     getAll: async (req, res, next) => {
5         try {
6             const users = await userService.getAll();
7
7             res.json(users);
8         } catch (error) {
9             next(error);
10        }
11    },
12    getById: async (req, res, next) => {
13        try {
14            const id = req.params.id;
15
16            const user = await userService.getById(id);
17
18            res.json(user);
19        } catch (error) {
20            next(error);
21        }
22    },
23    create: async (req, res, next) => {
24        try {
25            const message = await userService.create(req.body);
26
27            res.json(message);
28        } catch (error) {
29            next(error);
30        }
31    },
32    update: async (req, res, next) => {
33        try {
34            const id = req.params.id;
35
36            const message = await userService.update(req.body, id);
37
38            res.json(message);
39        } catch (error) {
40            next(error);
41        }
42    },
43    delete: async (req, res, next) => {
44        try {
45            const id = req.params.id;
46
47            const message = await userService.delete(id);
48
49            res.json(message);
50        } catch (error) {
51            next(error);
52        }
53    },
54};
```

Powyższy przykład przedstawia definicję kontrolera, który jest odpowiedzialny za obsługę żądań związanych z użytkownikami. Każda metoda kontrolera odpowiada jednej akcji, która ma zostać wykonana. W przypadku powodzenia, kontroler zwraca odpowiedź. W przypadku niepowodzenia, kontroler przekazuje błąd do middleware, który jest odpowiedzialny za obsługę błędów.

Service

Service jest odpowiedzialny za dostęp do bazy danych. Każda metoda serwisu odpowiada jednej akcji, która ma zostać wykonana. Serwis przekazuje żądanie do bazy danych, a następnie zwraca wynik do kontrolera. Przykładowa definicja funkcji serwisu przedstawiona jest na listingu 5.5.

Listing 5.5: Przykładowa definicja serwisu

```
1  create: async (newUser) => {
2      const user = await client.query(
3          "INSERT INTO users (email, password, first_name, last_name) VALUES (${
4              newUser.email}", "${newUser.password}", "${newUser.name}", "${newUser.
5                  last_name}") RETURNING *";
6  );
7
8  if (!user.rows.length) {
9      throw new ErrorWithStatus("Couldn't create new user with given data:\n${
10          newUser}", 400);
11 }
12
13 return {
14     message: 'User has been successfully created.',
15 };
},
```

Powyższy przykład przedstawia funkcję serwisu, która dodaje nowego użytkownika do bazy danych. Funkcja ta przyjmuje obiekt `newUser`, który zawiera dane nowego użytkownika. Następnie funkcja ta wykonuje zapytanie do bazy danych, które dodaje nowego użytkownika. W przypadku niepowodzenia, funkcja zwraca błąd 400. W przypadku powodzenia, funkcja zwraca wiadomość o sukcesie.

5.2.2 CRUD

CRUD to skrót od angielskich słów Create, Read, Update, Delete. Jest to zestaw podstawowych operacji, które można wykonać na bazie danych. Operacje te mają na celu zapewnienie spójności danych w przestrzeni bazy danych i serwera. W ramach projektu zaimplementowano operacje CRUD dla wszystkich encji występujących w systemie. Każda encja posiada swoje metody CRUD, które są odpowiedzialne za dodawanie, odczytywanie, aktualizowanie i usuwanie danych z bazy danych.

5.2.3 Autoryzacja

Autoryzacja to proces weryfikacji tożsamości użytkownika, który w pierwszej kolejności musi zarejestrować się w systemie, aby uzyskać dostęp do aplikacji. W ramach projektu zaimplementowano autoryzację opartą na tokenach JWT. Po zalogowaniu użytkownika generowany jest token JWT, który jest zapisywany w pamięci podręcznej przeglądarki. Token ten jest przesyłany w nagłówku HTTP przy każdej interakcji z serwerem. Serwer weryfikuje poprawność tokena i na tej podstawie zwraca odpowiedź. W przypadku nieprawidłowego tokena żądanie jest odrzucane, a serwer zwraca odpowiedni błąd.

Na serwerze, plikiem odpowiedzialnym za autoryzację jest `auth.middleware.js`. W pliku tym zdefiniowane są funkcje, które są odpowiedzialne za weryfikację tokena JWT. W przypadku nieprawidłowego tokena, serwer zwraca odpowiedni błąd. Na listingu 5.6 przedstawiono zawartość pliku `auth.middleware.js`.

Listing 5.6: Przykładowa definicja autoryzacji

```
1 import jwt from 'jsonwebtoken';
2 import { ErrorWithStatus } from '../error/error-with-status.js';
3
4 export const authMiddleware = (req, res, next) => {
5   const token = req.cookies.token;
6
7   if (!token) {
8     throw new ErrorWithStatus('Authentication required', 401);
9   }
10
11  try {
12    const decoded = jwt.verify(token, process.env.JWT_SECRET);
13    req.user = decoded;
14    next();
15  } catch (error) {
16    throw new ErrorWithStatus('Invalid token', 401);
17  }
18};
```

Po otrzymaniu żądania, serwer weryfikuje token JWT używając funkcji `jwt.verify()`. W przypadku po-prawnego tokena, serwer zwraca odpowiedź. Jeśli w żadaniu nie ma tokena, serwer zwraca błąd 401. W przypadku nieprawidłowego tokena, serwer również zwraca błąd 401.

5.2.4 Baza danych

Inicjalizacja bazy danych

W przypadku tworzenia nowego środowiska konieczna jest inicjalizacja nowej bazy danych. W tym celu powstał plik `init_tables.sql`, który zawiera skrypt tworzący nową bazę danych. Skrypt ten tworzy nową bazę danych oraz tabele. Na listingu 5.7 przedstawiono zawartość pliku `init_tables.sql`.

Listing 5.7: Przykładowa definicja inicjalizacji bazy danych

```
1  DROP TABLE IF EXISTS stores CASCADE;
2  DROP TABLE IF EXISTS cart_items CASCADE;
3  DROP TABLE IF EXISTS carts CASCADE;
4  DROP TABLE IF EXISTS products CASCADE;
5  DROP TABLE IF EXISTS users CASCADE;
6  DROP TABLE IF EXISTS units CASCADE;
7  DROP TABLE IF EXISTS categories CASCADE;
8  DROP TABLE IF EXISTS sections CASCADE;
9
10 CREATE TABLE stores (
11     store_id SERIAL PRIMARY KEY,
12     store_name VARCHAR(255) NOT NULL,
13     latitude VARCHAR(255) NOT NULL,
14     longitude VARCHAR(255) NOT NULL,
15     city VARCHAR(255) NOT NULL
16 );
17
18 CREATE TABLE sections (
19     section_id SERIAL PRIMARY KEY,
20     section_name VARCHAR(255) NOT NULL,
21     store_id INT REFERENCES stores(store_id) ON DELETE CASCADE
22 );
23
24 CREATE TABLE categories (
25     category_id SERIAL PRIMARY KEY,
26     category_name VARCHAR(255) NOT NULL,
27     section_id INT REFERENCES sections(section_id) ON DELETE CASCADE
28 );
29
30 CREATE TABLE units (
31     unit_id SERIAL PRIMARY KEY,
32     unit_name VARCHAR(50) NOT NULL,
33     unit_symbol VARCHAR(10) NOT NULL
34 );
35
36 CREATE TABLE products (
37     product_id SERIAL PRIMARY KEY,
38     name VARCHAR(255) NOT NULL,
39     description TEXT,
40     price DECIMAL(10,2) NOT NULL,
41     category_id INT REFERENCES categories(category_id) ON DELETE CASCADE,
42     availability VARCHAR(50) NOT NULL,
43     amount DECIMAL(10,2) NOT NULL,
44     unit_id INT REFERENCES units(unit_id) ON DELETE CASCADE
45 );
46
47 CREATE TABLE users (
48     user_id SERIAL PRIMARY KEY,
49     email VARCHAR(255) UNIQUE NOT NULL,
50     password VARCHAR(255) NOT NULL,
51     first_name VARCHAR(50) NOT NULL,
52     last_name VARCHAR(50) NOT NULL
53 );
54
55 CREATE TABLE carts (
56     cart_id SERIAL PRIMARY KEY,
57     user_id INT REFERENCES users(user_id) ON DELETE CASCADE,
58     creation_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
59 );
60
61 CREATE TABLE cart_items (
62     cart_item_id SERIAL PRIMARY KEY,
63     cart_id INT REFERENCES carts(cart_id) ON DELETE CASCADE,
64     product_id INT REFERENCES products(product_id) ON DELETE CASCADE,
65     quantity INT NOT NULL
66 );
```

Po utworzeniu bazy danych, konieczne jest dodanie danych do tabel. W tym celu powstał plik *init_data.sql*, który zawiera skrypt dodający dane do tabel. Na listingu 5.8 przedstawiono część zawartości pliku *init_data.sql*.

Listing 5.8: Przykładowa definicja inicjalizacji danych

```
1 DELETE FROM cart_items;
2 DELETE FROM carts;
3 DELETE FROM users;
4 DELETE FROM products;
5 DELETE FROM units;
6 DELETE FROM categories;
7 DELETE FROM sections;
8 DELETE FROM stores;
9
10 INSERT INTO stores (store_name, latitude, longitude, city) VALUES
11 ('Downtown_Supermarket', '40.7128', '-74.0060', 'Gdansk'),
12 ('Uptown_Grocery', '40.7831', '73.9712', 'Gdansk'),
13 ('Midtown_Market', '40.7549', '73.9840', 'Gdansk'),
14 ('East_Side_Shop', '40.7614', '73.9776', 'Gdansk'),
15 ('West_End_Store', '40.7736', '73.9566', 'Gdansk');
```

Po wywołaniu obu skryptów baza danych jest gotowa do użycia.

Rozdział 6

INSTRUKCJA UŻYTKOWANIA

tekst

Rozdział 7

WYNIKI TESTÓW

tekst

Rozdział 8

PODSUMOWANIE

8.1 Napotkane wyzwania

Podczas pracy nad projektem napotkano kilka wyzwań, które utrudniły proces implementacji. Poniżej przedstawiono najważniejsze z nich.

8.1.1 Obsługa nadajników BLE

Dla każdego członka projektu, technologia Bluetooth Low Energy była nowością. W związku z tym, konieczne było zapoznanie się z dokumentacją techniczną oraz zasadami działania tej technologii. W trakcie implementacji, napotkano kilka problemów związanych z obsługą nadajników BLE, które wymagały głębszej analizy i zrozumienia zasad działania tej technologii. Pierwszą przeszkodą było zasilenie nadajnika. Jako, że urządzenie ma jedynie piny VCC, GND, TX i RX, konieczne było zasilanie nadajnika z zewnętrznego źródła. W tym celu wypożyczono sprzęt w postaci płyt stykowych, przewodów połączeniowych oraz urządzenia Raspberry Pi. Następnie podłączono nadajnik do Raspberry Pi. Zasilone urządzenie należało następnie wyszukać wśród dostępnych urządzeń bluetooth. Do zidentyfikowania sygnału i obsługi urządzenia, użyto dokumentu technicznego opisującego interfejs nadajnika. [DSD TECH(n.d.)]

8.1.2 Praca w zespole

Kolejnym z wyzwań, które napotkano już na początku projektu była organizacja pracy zespołowej. Każdy członek zespołu miał swoje zadanie do wykonania, lecz wytwarzane komponenty były zależne od kodu innych członków zespołu. W związku z tym, konieczne było zapewnienie ciągłej komunikacji w zespole. W tym celu ustanowiono cotygodniowe spotkanie mające na celu omówienie postępów w pracy, a także rozwiązywanie napotkanych problemów. W trakcie spotkań omawiano również plany na kolejny tydzień, a także ustalano priorytety zadań. Takie podejście pozwoliło na efektywną pracę zespołową i na osiągnięcie celu projektu.

8.2 Przyszły rozwój aplikacji

Po zakończeniu pracy nad projektem w ramach pracy inżynierskiej, aplikacja jest gotowa do wdrożenia. W ramach przyszłego rozwoju aplikacji można by było zaimplementować kilka dodatkowych funkcjonalności, które rozszerzyłyby możliwości aplikacji. Poniżej przedstawiono kilka z nich.

8.2.1 Panel Administratora

W ramach przyszłego rozwoju aplikacji, warto byłoby dodać panel administratora, który pozwoliłby na zarządzanie użytkownikami, ich uprawnieniami, a także na zarządzanie treścią aplikacji. W panelu administratora można by było dodawać nowe kategorie, podkategorie, a także zarządzać częściami w ramach tych kategorii. Należy rozróżnić panel administratora dostępny dla managera sklepu, od panelu

dostępnego dla administratora systemu. W ramach panelu managera mogłyby zostać zaimplementowane następujące funkcje:

- Zarządzanie asortymentem sklepu
- Zarządzanie użytkownikami w zakresie sklepu
- Możliwość zgłoszenia problemu z aplikacją

Panel dostępny dla administratora systemu rozszerzałby powyższy panel o dodatkowe funkcje, takie jak:

- Zarządzanie użytkownikami w zakresie całego systemu
- Zarządzanie kategoriami i podkategoriami
- Zarządzanie treściami w ramach kategorii
- Zarządzanie zgłoszeniami problemów z aplikacją
- Zarządzanie panelami dostępowymi

8.2.2 Rozwinięcie modelu językowego

W ramach przyszłego rozwoju aplikacji, warto byłoby rozwinać model językowy, który pozwoliłby na bardziej zaawansowane rozpoznawanie intencji użytkownika. W ramach rozwoju modelu językowego można by było zaimplementować dodatkowe intencje, które pozwoliłyby na bardziej zaawansowane zapytania użytkownika. Przykładowe intencje, które można by było zaimplementować to:

- Zapytanie o skład produktu
- Zapytanie o listę składników do przygotowania dania
- Zapytanie o promocje w sklepie
- Zapytanie o godziny otwarcia sklepu

8.2.3 Rozwój aplikacji mobilnej

Inwentaż

Kolejnym planowanym rozszerzeniem aplikacji mobilnej jest funkcjonalność inwentarza. Miałoby to na celu ułatwienie układania listy zakupów. Użytkownik dodawałby do inwentarza produkty, które posiada już w domu. Następnie ustaliłby ilość każdego z produktów, jaką chciałby mieć na codzień w domu. Na tej podstawie, aplikacja generowałaby listę produktów, których brakuje w domu.

Planer zakupów i posiłków

Następnym możliwym rozszerzeniem jest planer zakupów. W ramach planera, użytkownik mógłby ustalić posiłki, które chciałby zjeść w tym tygodniu. W tym celu aplikacja musiała być rozszerzona o funkcję sprawdzania składników poszczególnych posiłków. Na tej podstawie, aplikacja generowałaby listę zakupów, która zawierałaby produkty, które są potrzebne do przygotowania posiłków na cały tydzień.

8.3 Wnioski

8.3.1 Osiągnięte cele

W ramach pracy inżynierskiej udało się zrealizować wszystkie cele postawione przed projektem. Aplikacja spełnia wszystkie wymagania funkcjonalne, które zostały zdefiniowane na początku projektu. W związku z tym, można uznać, że projekt zakończył się sukcesem.

8.3.2 Wartość praktyczna

Aplikacja ma duże znaczenie praktyczne zarówno dla użytkowników indywidualnych, jak i dla właścicieli sklepów. Umożliwiając łatwiejsze i bardziej efektywne zakupy, aplikacja przyczynia się do zwiększenia komfortu klientów, szczególnie tych, którzy borykają się z ograniczeniami ruchowymi, wzrokowymi lub innymi problemami zdrowotnymi. Funkcja nawigacji po sklepie z wykorzystaniem algorytmów optymalizacyjnych pozwala zaoszczędzić czas i zmniejszyć stres związany z wyszukiwaniem produktów w dużych placówkach. Dodatkowo, dzięki funkcji generowania kodów QR, proces finalizacji zakupów został znacznie uproszczony, co jest korzystne zarówno dla klientów, jak i dla sklepów, które mogą ograniczyć kolejki i usprawnić działanie kas samoobsługowych.

Dla branży handlowej aplikacja stanowi innowacyjne rozwiązanie, które może przyciągnąć nowych klientów, poprawić wizerunek firmy i zwiększyć jej konkurencyjność. Rozwiązanie to jest szczególnie cenne w kontekście współczesnych trendów, takich jak personalizacja usług czy dążenie do inkluzywności. Możliwość integracji aplikacji z istniejącymi systemami sklepowymi, jak również jej potencjał do działania w wielu sklepach, a nie tylko w jednej sieci, zwiększa elastyczność wdrożenia.

Oprócz zastosowań komercyjnych, aplikacja może znaleźć zastosowanie w programach społecznych wspierających osoby z niepełnosprawnościami, zwiększając ich samodzielność i komfort życia. Dzięki temu projekt wpisuje się w ideę społecznej odpowiedzialności biznesu (CSR) i może stanowić wzór dla innych rozwiązań technologicznych dążących do integracji i wsparcia osób wykluczonych.

Bibliografia

- [Chowdhary(2020)] K. R. Chowdhary. *Natural Language Processing*, pages 603–649. Springer India, New Delhi, 2020. ISBN 978-81-322-3972-7. doi:10.1007/978-81-322-3972-7_19. URL https://doi.org/10.1007/978-81-322-3972-7_19.
- [Croes(1958)] G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812, 1958. URL <https://www.jstor.org/stable/167074>.
- [DSD TECH(n.d.)] DSD TECH. DSD TECH HM-10 Datasheet. <https://people.ece.cornell.edu/land/courses/ece4760/PIC32/uart/HM10/DSD%20TECH%20HM-10%20datasheet.pdf>, n.d. [Accessed: 2024-12-10].
- [Flanagan(2011)] David Flanagan. *JavaScript: The definitive guide: Activate your web pages.* ” O'Reilly Media, Inc.”, 2011.
- [Gutin and Punnen(2002)] Gregory Gutin and Abraham P. Punnen. *The Traveling Salesman Problem and Its Variations*. Springer, Berlin, 2002. doi:10.1007/b101971. URL <https://link.springer.com/book/10.1007/b101971>.
- [Peters(2017)] Christian Peters. Building rich internet applications with node.js and express.js. *Rich Internet Applications w/HTML and Javascript*, page 15, 2017.
- [Wit.ai(2024)] Wit.ai. *Wit.ai HTTP API Documentation*, 2024. URL <https://wit.ai/docs/http/20240304/>. Accessed: 2024-12-11.
- [Worsley and Drake(2002)] J. Worsley and J.D. Drake. *Practical PostgreSQL*. Practical Series. O'Reilly Media, 2002. ISBN 9781565928466. URL <https://books.google.pl/books?id=G8dh95j5NgcC>.