Spis treści

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.

1.1.2 Dynamiczny rozwój rynku aplikacji mobilnych

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

1.1.3 Brak gotowych rozwiązań

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 użytkowniku, ale przede wszystkim ułatwić tę czynność osobom niedowidzącym i seniorom. Następnymi 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

Chęć pomocy

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 krwędzią. 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 pozawala na zastosowanie dobrze znanych algorytmów do znajdowania ścieżek.

Przciętny graf sklepu ma prostą i powtarzalną 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 poporawić 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, streszczanie, 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.

Rozdział 3

PROPOZYCJA ROZWIĄZANIA PROBLEMU

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

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

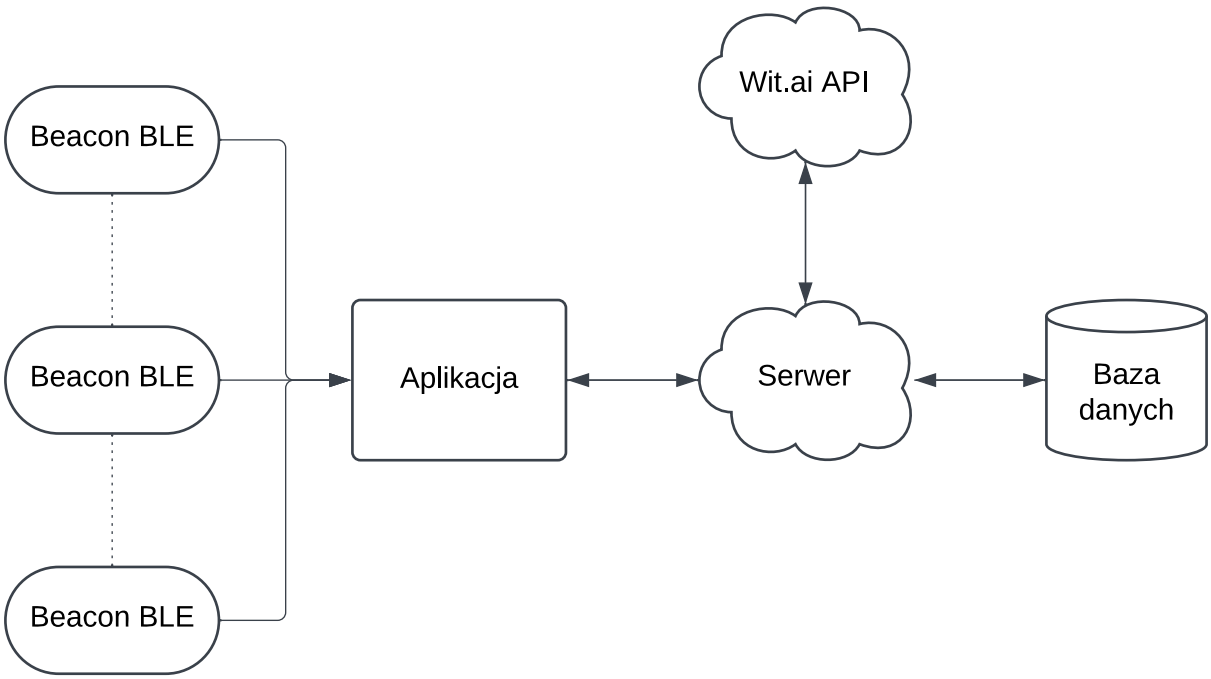
Po wykonaniu powyższych kroków i przetestowaniu aplikacji w nowym środowisku można rozpocząć testy publiczne.

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 żadanie 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)]

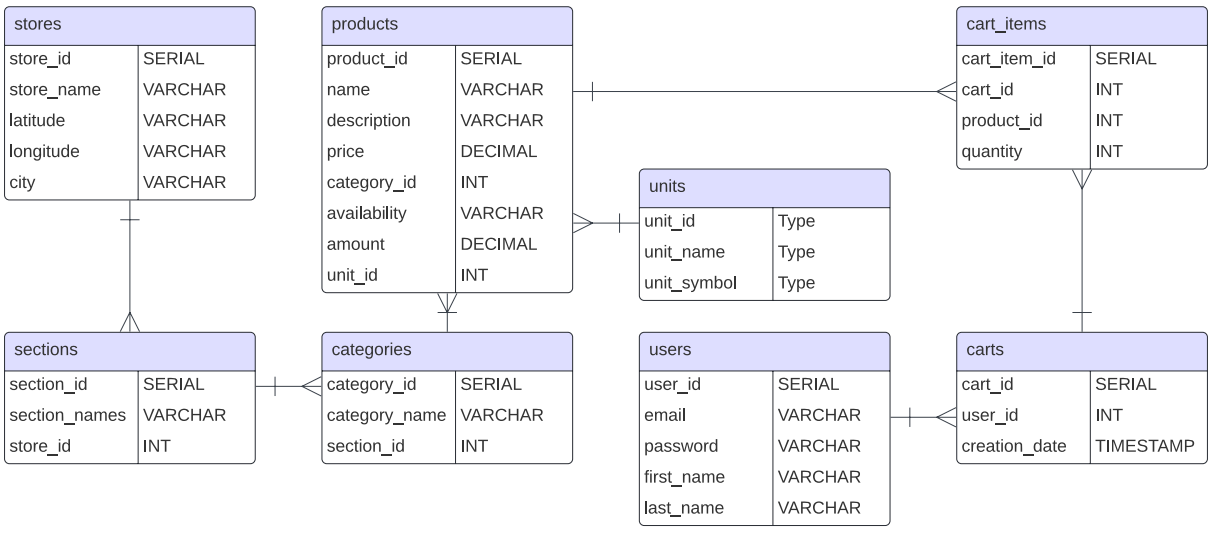
Baza danych przechowuje informacje o produktach, użytkownikach, koszykach oraz sklepach. Schemat bazy danych przedstawia rysunek 4.2.

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 *p*roducts 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 nadrzędnych. 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

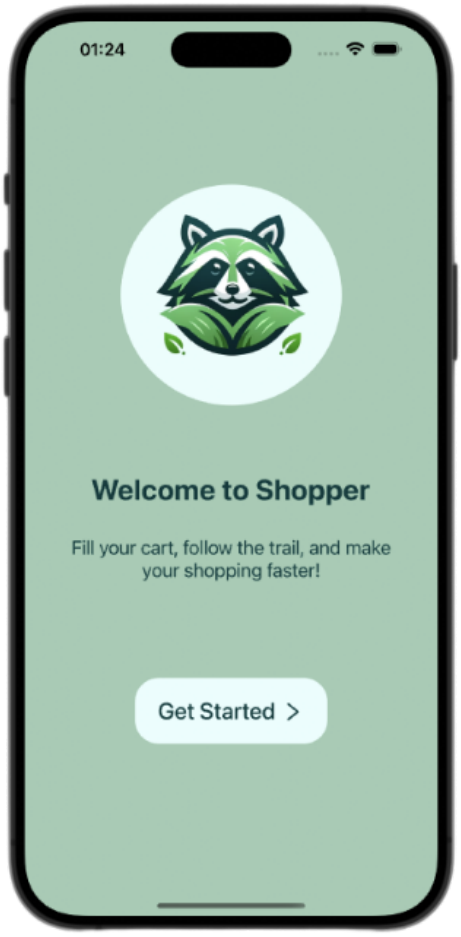
- cart_item_id - SERIAL PRIMARY KEY: Unikalny identyfikator pozycji w koszyku.
- cart_id - INT REFERENCES carts(cart_id) ON DELETE CASCADE: Klucz obcy wskazujący koszyk, do którego należy pozycja.
- product_id - INT REFERENCES products(product_id) ON DELETE CASCADE: Klucz obcy wskazujący produkt dodany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

tekst

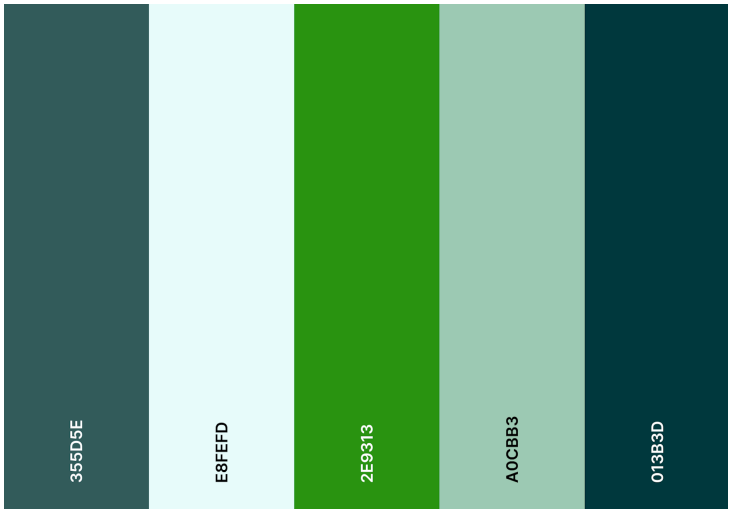
4.3 Interfejs użytkownika

4.3.1 Opis dostępnych widoków

Strona tytułowa



Strona tytułowa aplikacji pełni funkcję ekranu startowego, witając użytkownika logiem oraz hasłem zachęcającym do korzystania z aplikacji. Ponadto, widnieje na niej przycisk ”Get Started”, który umożliwia przejście do kolejnych widoków. Jeśli użytkownik 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

text

4.3.2 Szczegółowe omówienie implementacji

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 nasłuchuje 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 interpretery 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 określać zachowanie stron internetowych. [Flanagan(2011)]

4.4.2 Node.js

Node.js umożliwia 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 umożliwia 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:

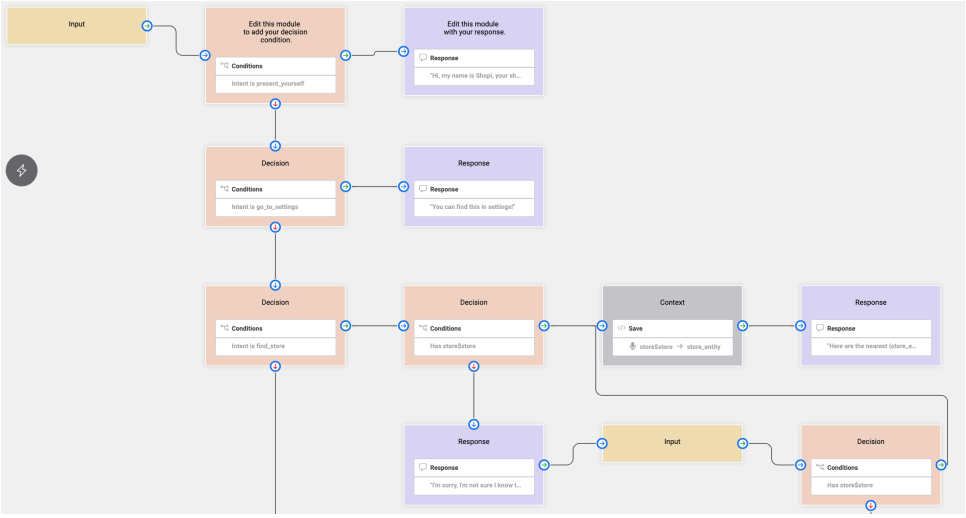
- *product* - encja reprezentująca nazwę produktu,
- *store* - encja reprezentująca nazwę sklepu,
- *view* - encja reprezentująca widok w aplikacji,
- *category* - encja reprezentująca nazwę kategorii produktów

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 wstępnym 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 możn a 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 zwrócona 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.

5.2 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.

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 5.1.

Listing 5.1: Przykładowa definicja ścieżki API

```
import { Router } from 'express';
import { userController } from './user.controller.js';
import { authMiddleware } from '../../middleware/auth.middleware.js';

export const userRouter = Router();

// Apply authMiddleware to all user routes
userRouter.use(authMiddleware);

// Protected user routes
userRouter.get('/', userController.getAll);
userRouter.get('/:id', userController.getById);
userRouter.post('/', userController.create);
userRouter.put('/:id', userController.update);
userRouter.delete('/:id', userController.delete);
```

Controller

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 5.2.

Listing 5.2: Przykładowa definicja kontrolera

```
import { userService } from './user.service.js';

export const userController = {
    getAll: async (req, res, next) => {
    try {
        const users = await userService.getAll();

        res.json(users);
    } catch (error) {
        next(error);
    }
    },
    getById: async (req, res, next) => {
    try {
        const id = req.params.id;

        const user = await userService.getById(id);

        res.json(user);
    } catch (error) {
        next(error);
    }
    },
    create: async (req, res, next) => {
    try {
        const message = await userService.create(req.body);

        res.json(message);
    } catch (error) {
        next(error);
    }
    },
    update: async (req, res, next) => {
    try {
        const id = req.params.id;

        const message = await userService.update(req.body, id);

        res.json(message);
    } catch (error) {
        next(error);
    }
    },
    delete: async (req, res, next) => {
    try {
        const id = req.params.id;

        const message = await userService.delete(id);

        res.json(message);
    } catch (error) {
        next(error);
    }
    },
};
```

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 5.3.

Listing 5.3: Przykładowa definicja serwisu

```
    create: async (newUser) => {
        const user = await client.query(
          "INSERT␣INTO␣users␣(email,␣password,␣first_name,␣last_name)␣VALUES␣("${
              newUser.email}",␣"${newUser.password}",␣"${newUser.name}",␣"${newUser.
              last_name}")␣returning␣*;"
        );

        if (!user.rows.length) {
          throw new ErrorWithStatus("Couldn't␣create␣new␣user␣with␣given␣data:\n${
              newUser}", 400);
        }

        return {
          message: 'User␣has␣been␣successfully␣created.',
        };
      },
```

5.3 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.

Rozdział 6

INSTRUKCJA UŻYTKOWANIA

tekst

Rozdział 7

WYNIKI TESTÓW

tekst

Rozdział 8

PODSUMOWANIE

8.1 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.

8.1.1 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 treściami 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 zgłośenia 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

Rozdział 9

LITERATURA

tekst

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.

[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.

[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.