Kierunek: ***Informatyka***

Specjalność: *Informatyka Stosowana*

Specjalizacja: *Inżynieria oprogramowania*

2023/2024

Krzysztof Książek

PRACA INŻYNIERSKA

System zarządzania zasobami żywnościowymi oraz dietą

Promotor pracy:

mgr inż. Tomasz Gądek

Tarnów, 2024

Spis treści

1. Wstęp

W dzisiejszych czasach rosnącej świadomości ekologicznej marnowanie żywności nabiera kluczowego znaczenia dla społeczeństwa. Niepokojąca statystyka mówiąca o wyrzucaniu miliardów ton jedzenia rocznie wskazuje na pilną potrzebę wprowadzenia narzędzi, które pozwolą ludziom efektywniej zarządzać zasobami żywnościowymi, przynosząc zarówno korzyści ekonomiczne, jak i ekologiczne.

1.1 Motywacja

W dobie wszechobecnej dostępności różnorodnych produktów paradoksalnie wiele osób zmaga się z problemem nadmiernego gromadzenia i wyrzucania jedzenia. Brak planowania posiłków, które wiążą się z nieefektywnymi zakupami, prowadzi nie tylko do degradacji środowiska, ale także do niepotrzebnych wydatków.

Chociaż obecnie mamy do dyspozycji mnóstwo źródeł informacji dotyczących zdrowego żywienia, wydaje się, że brakuje nam praktycznych narzędzi do efektywnego zarządzania zakupami i posiłkami. Tymczasem marnowane jedzenie to nie tylko strata pieniędzy. To również ogromne marnotrawstwo zasobów naturalnych oraz przyczyna znaczącej emisji gazów cieplarnianych. Proces produkcyjny, transport, przechowywanie — każdy z tych etapów wymaga zasobów i energii, które są marnowane, gdy żywność ląduje na śmietniku.

Dodatkowo nierównowaga w dostępie do żywności na świecie powoduje, że podczas gdy jedni marnują, inni głodują. Konsekwencje ekologiczne, ekonomiczne i społeczne marnowania jedzenia są zatem złożone i powiązane ze sobą.

W tym kontekście narzędzia, które pomagają ludziom w efektywnym zarządzaniu ich zasobami żywnościowymi, stają się kluczowe. Nie tylko do optymalizacji wydatków, ale również do edukacji konsumentów w zakresie zrównoważonej konsumpcji, dbałości o środowisko i globalnej odpowiedzialności.

1.2 Cel pracy

Głównym celem niniejszej pracy dyplomowej było zaprojektowanie oraz implementacja aplikacji internetowej do zarządzania zasobami żywnościowymi. Aplikacja miała za zadanie zintegrować funkcje zarządzania zapasami, planowania diety, monitorowania dat ważności produktów oraz generowanie listy zakupów. Ponadto aplikacja promuje ideę zrównoważonej konsumpcji, wspierając użytkowników w świadomym i efektywnym zarządzaniu zakupami oraz zapasami żywności.

1.3 Zakres pracy

W projekcie inżynierskim został zaimplementowany system składający się z trzech głównych elementów:

- aplikacja serwerowa,
- aplikacja internetowa,
- baza danych.

2. Analiza biznesowa

W tym rozdziale zostaną przedstawione główne możliwości oraz założenia funkcjonalne, jak i niefunkcjonalne aplikacji do zarządzania zasobami żywnościowymi.

2.1 Wymagania funkcjonalne

Główną funkcją aplikacji jest monitorowanie dostępnych produktów oraz ich dat ważności. Ważne jest, aby umożliwić użytkownikom:

- Dodawanie produktów do systemu, wraz z informacją o dacie ważności, kategorii produktu oraz zarządzanie nimi.
- Wprowadzenie podstawowych danych dotyczących użytkownika np. wzrost, waga.
- Dobór indywidualnego planu żywieniowego według preferencji użytkownika.
- Podgląd przepisów kulinarnych.
- Generowanie lub tworzenie listy zakupów.

2.2 Wymagania niefunkcjonalne

2.2.1 Bezpieczeństwo

Głównym priorytetem było zabezpieczenie danych użytkownika, co obejmuje nie tylko jego dane osobowe, ale także historię zakupów i spożytych produktów. Kluczowe było wdrożenie zaawansowanego systemu uwierzytelnienia i autoryzacji. W celu zapewnienia dodatkowego poziomu bezpieczeństwa, aplikacja korzysta z systemu tokenów JWT (JSON Web Tokens). Ta metoda pozwala na weryfikację i przesyłanie informacji w sposób bezpieczny. Używając tej kombinacji tokenów, aplikacja może zapewnić bezpieczeństwo sesji użytkowników. Eliminuje to potrzebę ciągłego wprowadzania danych logowania, jednocześnie utrzymując wysoki poziom bezpieczeństwa.

2.2.2 Responsywność

Aplikacja powinna być w pełni zoptymalizowana, aby działać płynnie niezależnie od ilości danych, które są przechowywane. Ważne jest, aby interfejs był odpowiednio

zoptymalizowany, tak by możliwe było korzystanie z aplikacji zarówno na komputerach, jak i na urządzeniach mobilnych.

Rozwój technologii i zmieniające się nawyki użytkowników sprawiają, że dostosowanie serwisów i aplikacji do urządzeń przenośnych stała się nie tylko trendem, ale fundamentalnym wymogiem w dzisiejszym cyfrowym świecie. W roku 2022 ponad 90% użytkowników przeglądało sieć za pomocą urządzeń mobilnych, co wskazuje na istotną przewagę nad korzystaniem z komputerów stacjonarnych czy laptopów.

Przyjęcie strategii wstępnego projektowania dla urządzeń mobilnych uwzględnia potrzeby i oczekiwania użytkowników smartfonów już na wczesnym etapie. Ta strategia ma kluczowe znaczenie, zwłaszcza dla produktów i usług cyfrowych, które muszą zapewniać płynne doświadczenie na różnorodnych urządzeniach.

2.2.3 Intuicyjny interfejs użytkownika

Interfejs stanowi główne “okno” komunikacji pomiędzy aplikacją, a użytkownikiem. Jego przejrzystość, intuicyjność oraz łatwość obsługi mają kluczowe znaczenie dla sukcesu każdej platformy cyfrowej. Dobrze zaprojektowany interfejs zapewnia użytkownikowi płynne, przyjemne doświadczenie, co przekłada się na zwiększone zaangażowanie i częstotliwość korzystania z aplikacji.

Elementy nawigacyjne muszą być zaprojektowane w sposób spójny, umożliwiając intuicyjne przechodzenie między poszczególnymi sekcjami aplikacji. Ikony, przyciski i menu powinny być łatwo rozpoznawalne, a ich umiejscowienie powinno być logiczne i przemyślane, aby umożliwić użytkownikowi szybki dostęp do najbardziej pożądanych funkcji.

3. Opis wykorzystanych technologii

3.1 JavaScript

JavaScript to dynamiczny, słabo typowany język programowania, który pierwotnie został zaprojektowany do manipulowania treściami na stronach internetowych w czasie rzeczywistym. Język został zaprojektowany przez Brendana Eicha w 1995 roku oraz rozwinięty przez firmę Netscape. Początkowo nosił nazwę LiveScript, ale ze względu na popularność Javy przemianowano go na JavaScript. Ma ogromne zastosowanie, jednym z jego kluczowych rozszerzeń jest Node.js [3]. Wieloplatformowe środowisko uruchomieniowe, które pozwala na tworzenie aplikacji serwerowych w języku JavaScript.

Współczesny standard JavaScript, znany jako ECMAScript (lub ES), rozwija się nieustannie. Szczególnie przełomowy był ES6, wydany w 2015 roku, który wprowadził liczne nowe funkcje i składnię, odmieniając oblicze języka. Co ważne, od tego czasu ECMAScript regularnie otrzymuje aktualizacje, gwarantując, że JavaScript pozostaje na czele nowoczesnych technologii aplikacji internetowych.

Kluczowe cechy języka JavaScript:

- Asynchroniczność.
- Bazowanie na zdarzeniach (ang. *Event-driven).*
- Kompatybilność międzyplatformowa.

3.2 TypeScript

TypeScript to rozszerzenie języka JavaScript wdrożony przez firmę Microsoft w 2012 roku, którego głównym przeznaczeniem jest wprowadzenie statycznego typowania, co z kolei ma na celu zwiększenie efektywności i bezpieczeństwa podczas tworzenia aplikacji na dużą skalę.

TypeScript nie tylko wprowadza statyczne typowanie, ale także oferuje inne zaawansowane funkcje, takie jak interfejsy, typy wyliczeniowe (ang. *Enum)* czy dekoratory. Wszystkie te elementy mają na celu poprawę jakości kodu, uczynienie go bardziej czytelnym oraz łatwiejszym do lokalizowania błędów i utrzymania aplikacji.

Kluczowe cechy języka TypeScript:

- Statyczne typowanie.
- Kompatybilność z istniejącym kodem JavaScript.
- Zaawansowanie techniki w paradygmacie programowania obiektowego.

3.3 React.js

React jest biblioteką języka JavaScript zaprojektowaną z myślą o budowaniu zaawansowanych interfejsów użytkownika dla aplikacji internetowych oraz mobilnych [2]. Został opracowany i jest aktywnie rozwijany przez Facebooka, a jego premiera miała miejsce w 2013 roku. Jednym z głównych celów biblioteki React jest zapewnienie możliwości tworzenia szybkich i interaktywnych aplikacji w skomplikowanym środowisku internetowym.

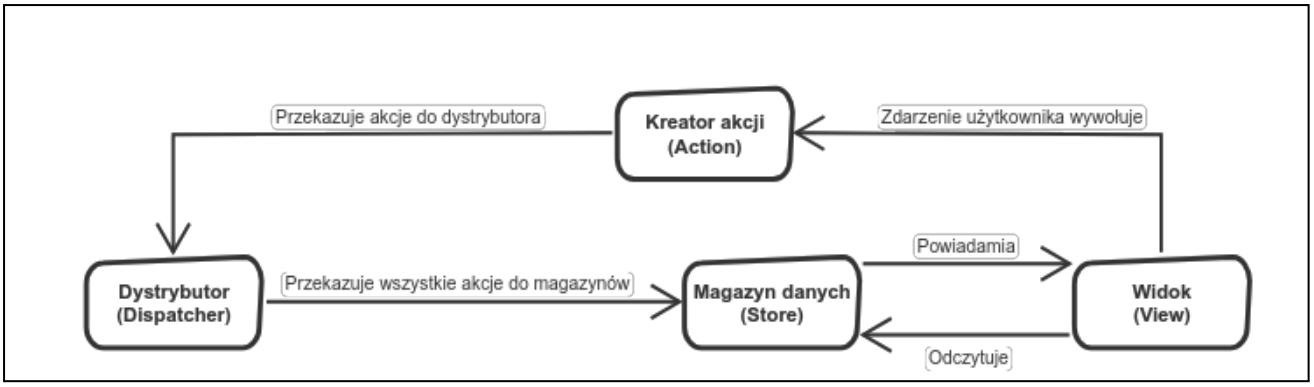
React wyróżnia się na tle innych bibliotek kilkoma kluczowymi cechami:

- Podział na komponenty.
- Wirtualny DOM.
- Jednokierunkowy przepływ danych.

Dzięki tym cechom React stał się jednym z najbardziej popularnych narzędzi do budowy interfejsów użytkownika. Jego zdolność do tworzenia aplikacji o wysokiej wydajności, które są jednocześnie elastyczne i łatwe w utrzymaniu, sprawiła, że został on przyjęty przez wiele dużych firm i społeczności programistów na całym świecie.

3.4 Redux

Redux jest to biblioteka służąca do zarządzania stanem między komponentami lub stanem całej aplikacji w środowisku JavaScript. Często wykorzystywany jest w połączeniu z biblioteką React.js. Została zaimplementowana na podstawie wzorca projektowego Flux, który składa się z czterech części opartych o jednokierunkowy przepływ danych (ang. *One-Way Data Flow*). [13]



Rysunek 3.1 Architektura Flux [opracowanie własne]

Opis architektury Flux:

- Widok – elementy budujące interfejs użytkownika, takie jak przyciski, pola tekstowe.
- Magazyn danych – obiekty, które przechowują stan aplikacji oraz reagują na konkretne akcje. Każdy magazyn jest zarejestrowany w dystrybutorze i otrzymuje powiadomienie o każdej akcji.
- Dystrybutor – jedyny punkt wejścia dla wszystkich akcji w aplikacji, gdy wykonywana jest akcja, która zostaje przekazana do magazynu.
- Akcje – obiekty zawierające informacje o tym, co się stało w aplikacji.

3.5 Node.js

Node.js to platforma umożliwiająca uruchamianie kodu JavaScript poza przeglądarką. Została opracowana w 2009 roku przez Ryana Dahla jako asynchroniczne środowisko uruchomieniowe JavaScript sterowane zdarzeniami. Node.js został zaprojektowany do tworzenia skalowalnych aplikacji sieciowych [3]. Podstawą jest silnik V8, który jest również używany w przeglądarce Chrome.

Asynchroniczna architektura tej platformy umożliwia obsługę wielu zapytań jednocześnie bez oczekiwania na zakończenie poprzednich operacji. Dzięki temu aplikacje bazujące na niej są szybkie i efektywne, co jest szczególnie ważne w środowiskach otrzymujących liczne żądania.

3.6 ExpressJS

ExpressJS to minimalistyczna i elastyczna platforma dla aplikacji internetowych oparta na Node.js, która dostarcza solidnego zestawu funkcji dla aplikacji serwerowych i mobilnych. Dzięki niezliczonym metodom użytkowym HTTP oraz oprogramowaniu pośredniczącemu, tworzenie wytrzymałego interfejsu API jest szybkie i łatwe [4].

Dzięki modularnej budowie ExpressJS pozwala na dodawanie dodatkowych funkcji poprzez oprogramowanie pośredniczące (ang. *middleware*). Użytkownicy mogą także korzystać z licznych wtyczek i bibliotek przeznaczonych specjalnie dla Express, które dostarczają dodatkowej funkcjonalności i ułatwiają tworzenie skomplikowanych aplikacji.

Elementy charakterystyczne ExpressJS:

- ExpressJS został zaprojektowany, aby ułatwić tworzenie interfejsów API i aplikacji internetowych.
- Umożliwia bardzo szybkie budowanie aplikacji bazujących na Node.js.
- Pozwala na obsługę różnych rodzajów żądań HTTP.

3.7 PostgreSQL

PostgreSQL to system zarządzania obiektowo-relacyjnymi bazami danych (ang. *Object-Relational Database Management System*) oparty na postgres opracowany na Wydziale Informatyki Uniwersytetu Kalifornijskiego w Berkeley [5]. PostgreSQL jest teraz dostępny jako wolne oprogramowanie (ang. *open-source*). Ten system obsługuje obszerną część standardu SQL oraz wprowadza wiele funkcji, które pozwalają na:

- Tworzenie złożonych zapytań SQL.
- Otwarta architektura umożliwia dostosowanie indywidualnych potrzeb.
- Wsparcie dla JSON, XML i innych nietypowych formatów danych.
- Silne wsparcie dla operacji równoczesnych dzięki wielowątkowości i optymalizacji zadań.

PostgreSQL jest nie tylko potężny, ale także elastyczny, co czyni go idealnym wyborem dla różnorodnych projektów od prostych aplikacji internetowych po zaawansowane systemy biznesowe.

3.8 TypeORM

TypeORM to biblioteka mapowania obiektowo-relacyjnego zaprojektowana do efektywnego i łatwego dostępu do baz danych w aplikacjach opartych na platformie Node.js. Wyróżniającą cechą TypeORM jest jej głęboka integracja z językiem TypeScript, chociaż równie dobrze współpracuje z klasycznym JavaScriptem [6].

Dzięki temu narzędziu, twórcy oprogramowania mogą definiować modele baz danych przy użyciu klas i dekoratorów, co sprawia, że struktura bazy danych jest czytelna, łatwa w utrzymaniu. TypeORM umożliwia pracę z wieloma systemami baz danych, w tym PostgreSQL, MySQL, SQLite i wieloma innymi.

Główne cechy TypeORM:

- Aktywna obsługa TypeScript i JavaScript.
- Deklaratywna definicja schematów baz danych za pomocą dekoratorów lub schematów JSON.
- Wsparcie dla większości powszechnie używanych systemów baz danych.
- Łatwa migracja danych oraz generowanie migracji na podstawie zmian w kodzie.

TypeORM sprawia, że prace z bazami danych stają się bardziej spójne i jednolite, niezależnie od użytego systemu bazodanowego, co czyni go idealnym wyborem dla projektów, które mogą potrzebować wsparcia dla różnych baz danych w trakcie ich życia.

3.9 Docker

Docker to otwarta platforma wspomagająca dostarczanie oraz uruchamianie aplikacji. Pozwala na izolację aplikacji od infrastruktury, co przekłada się na przyspieszenie procesu dostarczania oprogramowania.

To narzędzie umożliwia efektywne zarządzanie infrastrukturą, zachowując jednocześnie spójność i wydajność. Dostarcza również narzędzia do pakowania oraz uruchamiania aplikacji w kontenerach, co gwarantuję izolację i bezpieczeństwo. Kontenery zawierają wszelkie niezbędne elementy niezależnie od środowiska użytkownika. Posiada również możliwość zarządzania cyklem życia kontenerów, od początkowego rozwoju i testów po wdrażanie w środowisku produkcyjnym. [8]

3.10 JSON

JSON (JavaScript Object Notation) to format wymiany danych, który jest łatwy do odczytania i zapisania przez ludzi, a także prosty w analizie i generowaniu przez maszyny. Opiera się na podzbiorze standardu ECMA-262 3rd Edition języka programowania JavaScript. JSON jest formatem tekstowym, który jest całkowicie niezależny od języka programowania.

Format JSON opiera się na dwóch strukturach:

- Kolekcji par nazwa/wartość. W różnych językach jest to realizowane jako obiekt, rekord, struktura, słownik, tablica haszująca, lista kluczowana lub tablica asocjacyjna.
- Uporządkowanej liście wartości. W większości języków jest to realizowane jako tablica, wektor, lista lub sekwencja. [9]

Listing 3.1 Struktura obiektu JSON [opracowanie własne]

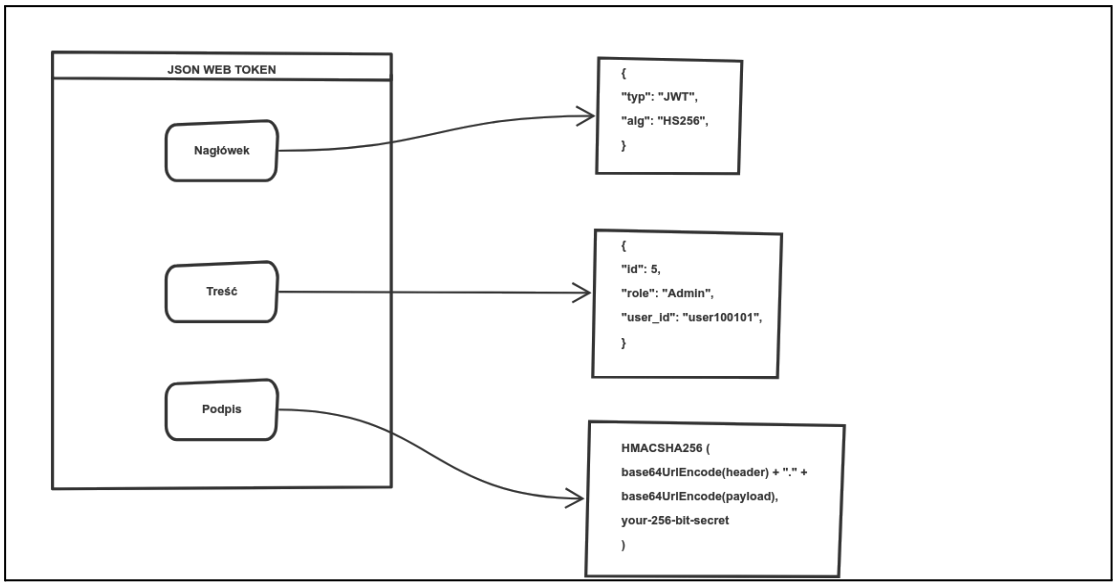
```
{
  "id": 5,
  "name": "Sałatka warzywna",
  "calories": 150,
  "protein": 7.5,
  "carbs": 20.3,
  "fat": 5.8,
  "isFresh": true,
  "ingredients": [
    {
      "id": 8,
      "productId": 3,
      "quantity": 2
    },
    {
      "id": 9,
      "productId": 6,
      "quantity": 1
    }
]
}
```

3.11 JWT

JSON Web Token (JWT) to otwarty standard (RFC 7519), który definiuje kompaktowy i samowystarczalny sposób bezpiecznego przesyłania informacji w formacie obiektu JSON między stronami. Te informacje można zweryfikować, ponieważ są one cyfrowo podpisane. JWT mogą być podpisywane przy użyciu klucza tajnego (z algorytmem HMAC) lub pary kluczy publiczny/prywatny przy użyciu RSA, lub ECDSA.

Struktura JSON Web Token:

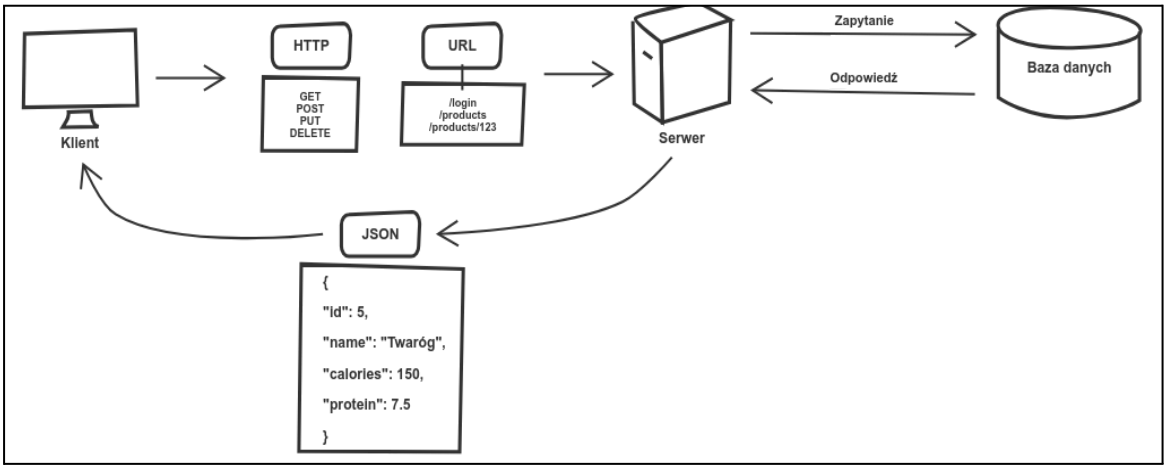
- Nagłówek (ang. *Header*): To pierwsza część JWT, zawiera informacje o typie tokenu (JWT) oraz używanym algorytmie podpisywania (na przykład HS256).
- Treść (ang. *Payload*): To druga część tokenu, która przechowuje deklaracje, czyli oświadczenia o podmiocie (zwykle użytkowniku) i dodatkowych danych. Deklaracje mogą być zarejestrowane, publiczne lub prywatne.
- Podpis (ang. *Signature*): To trzecia część tokenu, która jest wynikiem podpisania zakodowanego nagłówka, zakodowanej treści, tajnego klucza i algorytmu podpisywania. Służy do weryfikacji integralności tokenu oraz w przypadku tokenów podpisanych kluczem prywatnym, potwierdzenia tożsamości nadawcy JWT. [7]



Rysunek 3.2 Struktura tokenu JWT [opracowanie własne]

3.12 REST API

Representational State Transfer (REST) to zbiór ogólnych zasad projektowania interfejsów API (ang. *Application Programming Interface*) dla nowoczesnych usług sieciowych opartych o architekturę klient-serwer, w której następuje rozdział pomiędzy interfejsem użytkownika a przechowywaniem danych, które są gromadzone na jednym głównym serwerze, co poprawia bezpieczeństwo aplikacji. Wspomniana architektura ma na celu osiągnięcie wysokiej skalowalności aplikacji oraz łatwej przenośności pomiędzy platformami systemowymi lub sprzętowymi. [11]



Rysunek 3.3 Schemat architektury REST API [opracowanie własne]

3.13 HTTP

Hypertext Transfer Protocol (HTTP) to prosty protokół na podstawie modelu klient-serwer zazwyczaj przesyłany połączeniem warstwy aplikacji TCP/IP. Przedstawia on sposób komunikacji pomiędzy klientem, który wysyła żądanie HTTP do serwera, który przetwarza je i wysyła do klienta odpowiedź. [10]

Żądanie HTTP składa się z:

- Metody – określa rodzaj żądania, jakie klient przesyła na serwer.
- Nagłówków – posiadają one dodatkowe informacje o żądaniu klienta np. sposób kodowania obsługiwany przez klienta.
- Adresu URL – lokalizacja zasobu, do którego klient przesyła żądanie.

Metoda	**Przedmiot żądania**
GET	Pobiera zasoby
PUT	Aktualizuje zasoby na serwerze
POST	Dodaje zasoby do serwera
DELETE	Żąda usunięcia zasobu z serwera

Tabela 3.1 Metody żądań HTTP [opracowanie własne]

Odpowiedź HTTP posiada:

- Kod odpowiedzi – trzycyfrowy kod diagnostyczny informującym o pomyślnym lub nieprawidłowym przetworzeniu żądania.
- Nagłówki – posiadają dodatkowe informacje o odpowiedzi np. sposób kodowania zawartości.
- Treść – dane przesyłane w odpowiedzi serwera do klienta.

Grupa	**Znaczenie**	**Przykład**
1xx	Informacja o żądaniu	100 – serwer zaakceptował żądanie klienta
2xx	Poprawne przetworzenie żądania	200 – żądanie zostało poprawnie przetworzone
3xx	Przekierowanie	301 – żądany zasób znajduje się pod innym adresem
4xx	Błąd po stronie klienta	404 – nie znaleziono zasobu
5xx	Błąd po stronie serwera	500 – wewnętrzny błąd serwera

Tabela 3.2 Kody diagnostyczne serwera [opracowanie własne]

3.14 Swagger

Swagger to narzędzie przeznaczone dla interfejsów API, które umożliwia programistom w sposób jasny i czytelny opisanie, udokumentowanie i testowanie aplikacji. Jego głównym celem jest zrealizowanie przejrzystej dokumentacji API, co zdecydowanie ułatwia zrozumienie funkcji oraz wymaganych parametrów dla innych programistów. Pozwala również na testowanie API bez potrzeby posiadania lub korzystania z aplikacji klienckiej, co jest szczególnie ważne przed procesem integrującym aplikację serwerową wraz z aplikacją kliencką.

Dodatkowo Swagger oferuje mechanizmy definiowania walidacji danych wejściowych i wyjściowych dla poszczególnych punktów aplikacji. Dzięki takiemu rozwiązaniu można skutecznie zabezpieczyć aplikację przed potencjalnymi błędami związanymi z nieprawidłowym przesyłem danych

W ramach ekosystemu Swagger istnieje również Swagger UI, który w sposób graficzny przedstawia w pełni interaktywną dokumentację, co ułatwia zrozumienie struktury projektu oraz poszczególnych modułów aplikacji. [12]

Przykładowy kod w postaci komentarza znajduje się na listingu 3.2. Adnotacja **@swagger** definiujący dokumentację dla punktu końcowego (ang. *endpoint*) logowania użytkownika:

Listing 3.2 Struktura swagger docs [opracowanie własne]

```
/**
 * @swagger
 * /logowanie:
 *   post:
 *     summary: Logowania użytkownika
 *     tags: [Autoryzacja]
 *     responses:
 *       '200':
 *         description: Sukces logowania
 *       '400':
 *         description: Błędne zapytanie
 */
```

4. Architektura systemu

Aplikacja System zarządzania zasobami żywnościowymi oraz dietą został zaimplementowany za pomocą architektury klient–serwer, zgodnie z konwencją zasad REST API. Podstawowym nośnikiem informacji jest format JSON ze względu na swoją uniwersalność oraz fakt, że większość aplikacji została napisana za pomocą Języka JavaScript. System został podzielony na trzy niezależne platformy:

- Aplikacja serwerowa,
- Aplikacja internetowa.
- Baza danych.

Dzięki takiemu rozwiązaniu możliwy jest niezależna rozwój dwóch głównych komponentów systemu, elastyczna rozbudowa w trakcie jej funkcjonowania, jak również sprawny przesył danych w gotowej aplikacji.

4.1 Aplikacja serwerowa

Aplikacja serwerowa realizuje zadania przyjmowania oraz obsługi żądań HTTP wysyłanych przez aplikacje klienckie. Kluczowym zadaniem jest bezpieczeństwo danych zwłaszcza w kontekście uwierzytelniania. Wprowadza to uwierzytelnienie użytkownika, która ma na celu sprawdzenie, czy dany podmiot jest rzeczywiście tym, za którego się podaje oraz autoryzacja, która weryfikuje dostęp do zasobów zgodnie z posiadaną rolą. W przypadku tej aplikacji jest to szczególnie ważne, ponieważ wymaga to gromadzenia wielu informacji osobistych takich jak waga, wzrost, styl życia. Przechowuje również konfigurację serwera wraz z połączeniem do bazy danych i obsługę zapytań w kontekście CRUD (ang. Create Read Update Delete).

4.2 Aplikacja internetowa

Aplikacja internetowa służy do utworzenia indywidualnego konta użytkownika wraz z wprowadzeniem podstawowych danych. Przejrzysty interfejs aplikacji ma na celu płynne i intuicyjne poruszanie się w środowisku klienckim. Każdy zalogowany użytkownik będzie posiadał dostęp do własnej spiżarni produktów, w której w łatwy sposób będzie zarządzał swoimi zasobami. Dodatkowo będzie posiadał stały podgląd do kalkulatora BMI, który pozwoli na dobór odpowiedniego planu żywieniowego.

4.3 Diagram przypadków użycia

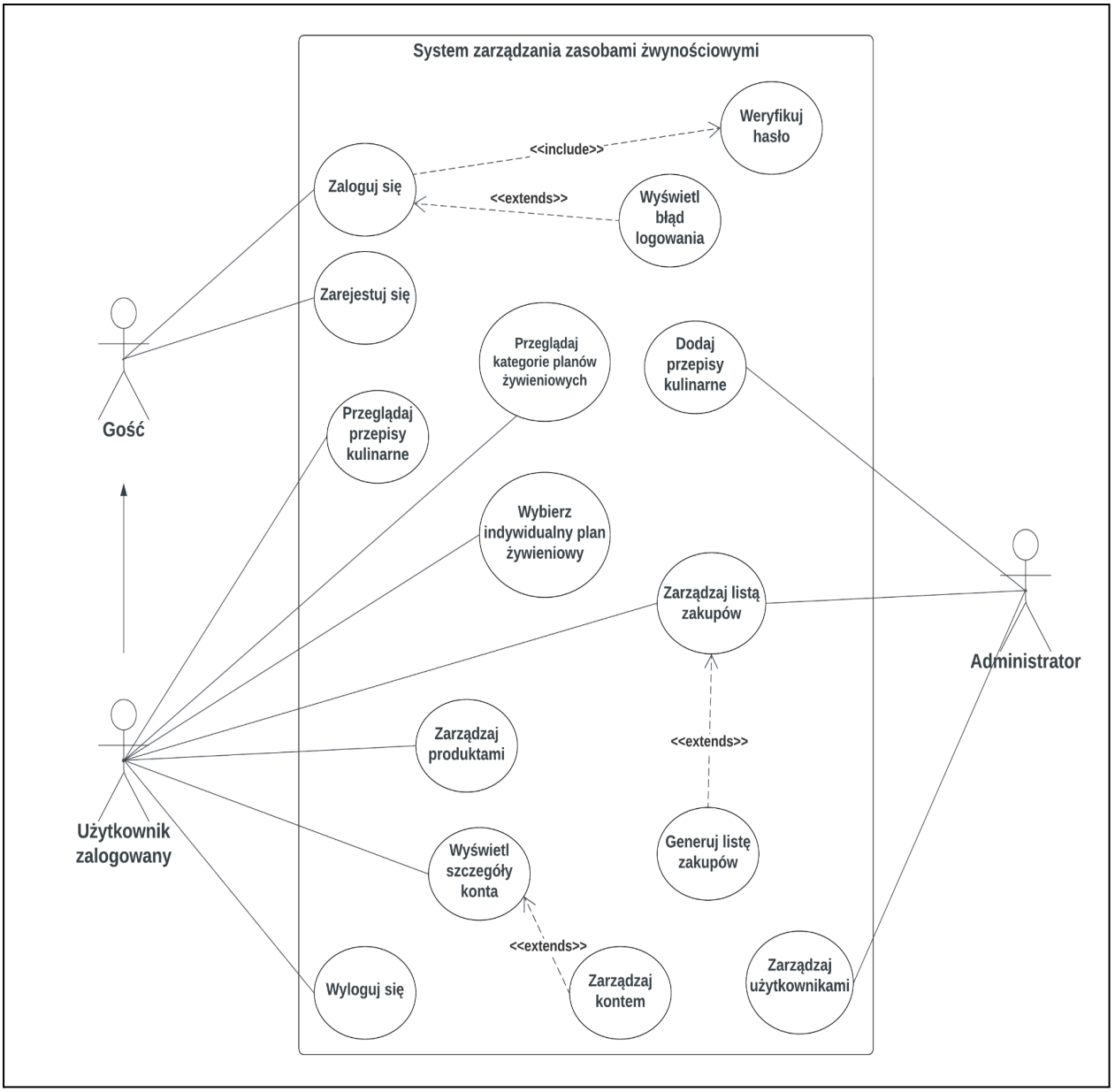
Diagram przypadków użycia (ang. *Use Case Diagram*) przedstawia interakcję poszczególnych użytkowników systemu zarządzania zasobami żywnościowymi oraz dietą. Opisuje różne przypadki, w których użytkownicy są podzieleni na role:

- Gość,
- Użytkownik zalogowany,
- Administrator.

Gość (osoba korzystająca z aplikacji, która nie posiada obecnie konta lub nie jest zalogowana) jest to użytkownik o najniższym poziomie uprawnień w aplikacji. Posiada on dostęp do rejestracji konta, zalogowania się oraz przeglądania dostępnych przepisów kulinarnych.

Po udanym procesie logowania użytkownik otrzymuje dużo więcej możliwości w aplikacji. Poza przeglądaniem przepisów kulinarnych może wprowadzić podstawowe dane osobiste, na podstawie których będzie mógł wygenerować lub ustalić swój indywidualny plan żywieniowy zgodnie z preferencjami. Dodatkowo użytkownik może zarządzać własną spiżarnią, co doskonale współgra z funkcją generowania listy zakupów oraz monitorować termin ważności produktów.

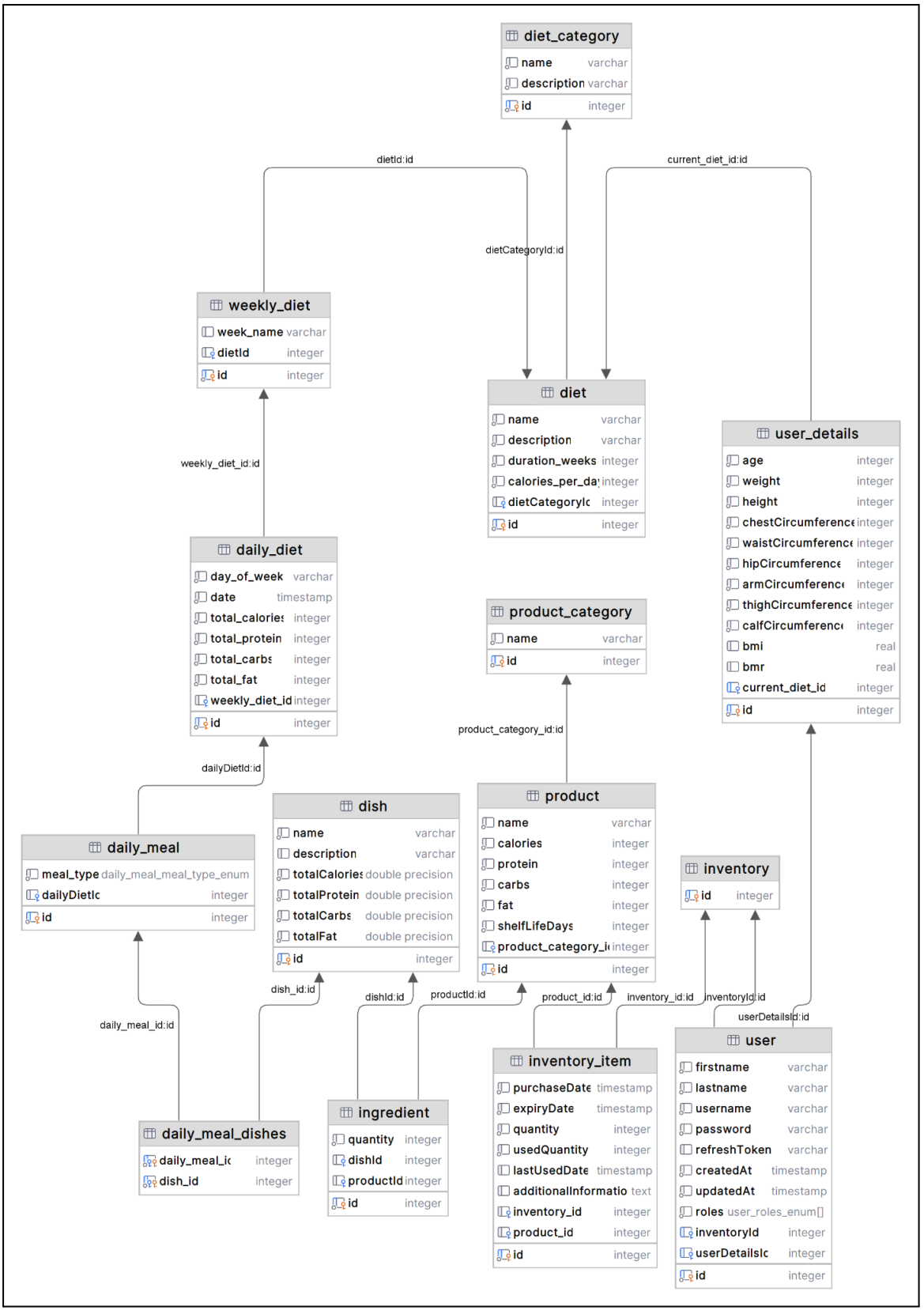
Najwyższy poziom uprawnień należy do Administratora, jest to osoba, która posiada dostęp do wszystkich funkcji użytkownika zalogowanego, jak również szczegółowe zarządzanie osobami posiadającymi konta (resetowanie haseł użytkowników, usuwanie użytkowników z systemu), dodawanie nowych przepisów kulinarnych oraz monitorowanie marnowanej żywności przez użytkowników.



Rysunek 4.1 Diagram przypadków użycia [opracowanie własne]

4.4 Diagram relacji encji

Diagram encji (ang. *Entity Relationship Diagram*) przedstawia strukturę danych w systemie zarządzania zasobami żywnościowymi oraz dietą. Opisuje powiązanie ze sobą różnych elementów systemu, obejmuje informacje o użytkownikach, rolach, danych osobistych, planach żywieniowych, produktach znajdujących się w spiżarni użytkownika oraz listy zakupów. Dzięki dokładnemu zaplanowaniu struktury bazy danych możliwe jest efektywne implementowanie założonych celów aplikacji, lepsze zrozumienie funkcji systemu, co przekłada się na sprawny proces wdrożenia aplikacji i minimalizację błędów.



Rysunek 4.2 Diagram relacji encji [opracowanie własne]

4.5 Bezpieczeństwo systemu

Przed przystąpieniem do realizacji projektu dokonano analizy bezpieczeństwa aplikacji, ze szczególnym uwzględnieniem bezpiecznego przechowywania danych poufnych użytkowników, takich jak hasła oraz informacji związanych z prywatnością. Obecny system przewiduje trzy role użytkownika: gość, użytkownik zalogowany, administrator. W kontekście zabezpieczenia aplikacji oraz autoryzacji zdecydowano się na zastosowanie standardu JWT (ang. JSON Web Token). Tokeny są interpretowane jako wirtualne klucze, które potwierdzają tożsamość użytkownika podczas komunikacji z serwerem. Użytkownik po udanym procesie logowania otrzymuje dwa żetony: żeton dostępu *(*ang. *access token*) oraz żeton odnowienia (ang. *refresh token*). Dla podniesienia poziomu bezpieczeństwa tokeny JWT posiadają określony czas ważności. Żeton dostępu jest krótkoterminowy, co oznacza, że jest skuteczny tylko przez określony czas (5 minut) natomiast, token odnowienia ma dłuższy okres ważności (1 dzień) i umożliwia on wydanie nowego tokena dostępu po jego wygaśnięciu.

Kolejnym istotnym aspektem aplikacji jest zastosowanie ról użytkowników oraz zabezpieczenia tras końcowych (ang. endpoint). Dzięki takiemu rozwiązaniu można rozwiązać potencjalne zagrożenia związane dostępem do funkcji aplikacji przez nieuprawnionych użytkowników. Role użytkowników pozwalają na klarowne zdefiniowanie poziomu dostępu.

Dodatkowo, dla zwiększenia bezpieczeństwa istotne jest, aby hasła użytkowników nie były przechowywane w sposób jawny. Dlatego przed zapisaniem hasła do bazy danych podlega ono procesowi szyfrowania. Ten proces obejmuje generowanie unikalnej soli (ang. salt), która jest następnie dodawana do hasła, co sprawia, że nawet dla tych samych haseł, generowane są unikalne zaszyfrowane dane. W rezultacie jeśli dane trafią w niepowołane ręce, bardzo trudno będzie odzyskać rzeczywiste hasła użytkowników.

5. Implementacja aplikacji serwerowej

5.1 Opis zadań realizowanych przez aplikację serwerową

Aplikacja serwerowa jest pośrednikiem pomiędzy interfejsem użytkownika a bazą danych, pełni kluczową rolę zapewniającą płynną komunikację oraz skuteczne zarządzanie danymi. Jej podstawowym zadaniem jest przyjmowanie, przetwarzanie żądań zewnętrznych, w tym logiki biznesowej, a także zagwarantowanie bezpieczeństwa zasobów systemu. Realizuje ona następujące zadania:

- Użytkownik niezalogowany (nieautoryzowany):
    - Rejestracja konta w systemie (proces zapisywania danych użytkownika takich jak login, hasło poddawany odpowiednim zabezpieczeniom oraz szyfrowaniu hasła),
    - Logowanie do systemu (proces uwierzytelnienia, sprawdzenie poprawności danych logowania),
    - Weryfikacja tokenów (proces weryfikacji poprawności tokenów JWT w celu potwierdzenia autentyczności użytkownika i dostępu do zasobów systemu)
- Użytkownik zalogowany (autoryzowany)
    - Wprowadzanie szczegółowych danych oraz zarządzanie nimi (możliwość wprowadzenia danych użytkownika, np. wzrost, waga. Te dane są wykorzystywane do obliczeń wskaźnika masy ciała BMI),
    - Dodawanie produktów do spiżarni (proces, w którym użytkownik ma możliwość wprowadzania informacji o produktach, które przechowuje w swojej spiżarni),
    - Wybieranie oraz dodawanie diet oraz ich kategorii (umożliwienie wybierania przez użytkownika diety na określony czas dostępnej w systemie, jak również generowanie planu żywieniowego zgodnie z indywidualnymi preferencjami),
    - Generowanie listy zakupów (możliwość generowania listy zakupów na podstawie posiadanych produktów w spiżarni i wybranego planu żywieniowego)

5.2 REST API

Aplikacja serwerowa przetwarza żądania aplikacji internetowej na podstawie architektury REST API (ang. *Representational State Transfer Application Programming Interface*). W ramach tej architektury kluczową rolę odgrywają tak zwane kontrolery, które pełnią funkcję punktów końcowych odpowiedzialnych za obsługę i udostępnianie określonych operacji w systemie. Dodatkowo architektura ta korzysta z różnym metod HTTP do komunikacji z kontrolerami:

- GET – używana do pobierania zasobów,
- PUT – służy do umieszczania danych lub aktualizacji danych,
- POST – umożliwia dodawania nowych danych,
- DELETE – wykorzystywana do usuwania zasobów.

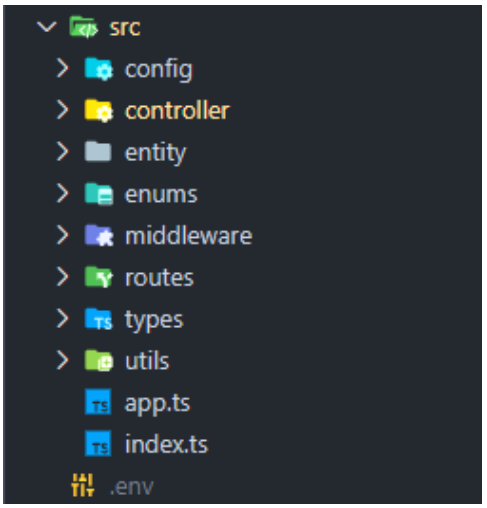
Przy implementacji aplikacji serwerowej zostało utworzonych 9 kontrolerów, których działanie zostało udokumentowane za pomocą narzędzia Swagger, dzięki czemu dokumentacja API stanowi szczegółowe informacje dotyczące dostępnych operacji w systemie, umożliwiając łatwe zrozumienie sposobu korzystania z interfejsu programistycznego.

5.3 Struktura folderów

W celu skutecznego zarządzania kodem oraz zwiększeniu czytelności projektu zaimplementowano i wdrożono starannie zaplanowaną strukturę folderów. Jej głównym celem jest umożliwienie łatwego i intuicyjnego przeglądania kodu źródłowego. Poniżej znajduje się szczegółowe przedstawienie poszczególnych elementów:

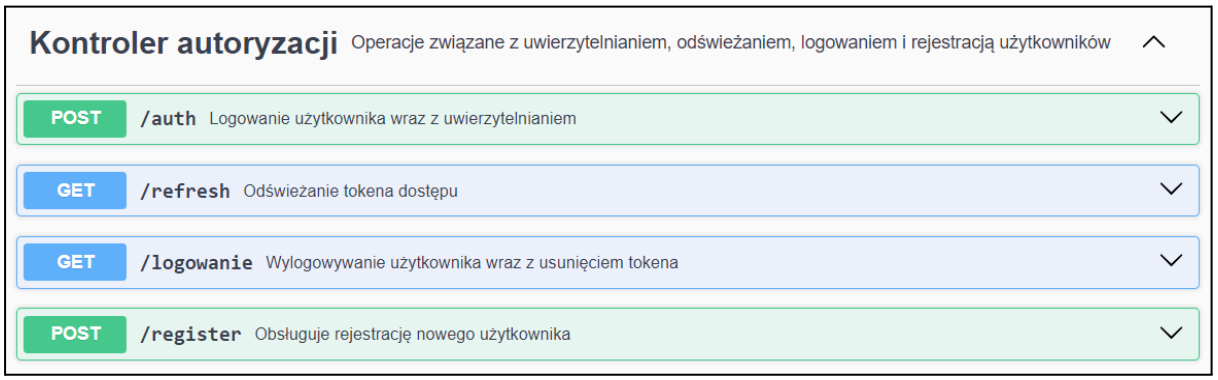
- config: W tym folderze umieszczone są pliki konfiguracyjne związane z połączeniem z bazą danych oraz konfiguracje dozwolonych źródeł komunikacji.
- controller: Sekcja ta obejmuje kontrolery aplikacji, które odpowiadają za obsługę żądań od klienta.
- entity: Zawiera definicje encji (modeli danych), które opisują strukturę kolumn i tabel przechowywanych w bazie danych.

- enums: Folder ten przechowuje typy wyliczeniowe, które pełnią funkcję jednoznacznego definiowania zestawów stałych, które są używane w różnych miejscach aplikacji.
- middleware: Miejsce dla funkcji pośredniczących, wykonujących operacje w trakcie przetwarzania żądań.
- routes: Zawiera zdefiniowane ścieżki (ang. *endpoints*), czyli punkty końcowe umożliwiające komunikację z aplikacją na podstawie metod HTTP.
- types: Zawiera zdefiniowane typy danych, co przyczynia się do lepszej czytelności i większą kontrolą nad pisanym kodem.
- utils: Narzędzia pomocnicze, które mogą być wykorzystywane w różnych częściach projektu.
- app.ts: Główny plik startowy serwera, w którym zdefiniowane są główne ustawienia serwera.
- index.ts: Plik startowy serwera, inicjujący serwer HTTP na odpowiednim porcie oraz realizujący połączenie z bazą danych.
- .env: Plik zawierający zmienne środowiskowe, takie jak klucze API, tajne tokeny.

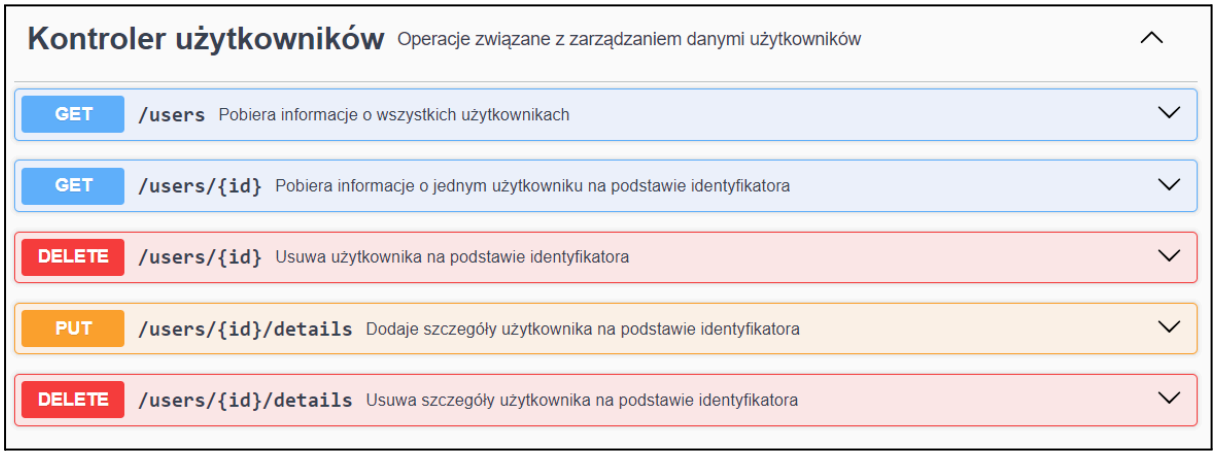


Rysunek 5.1 Struktura folderów aplikacji serwerowej [opracowanie własne]

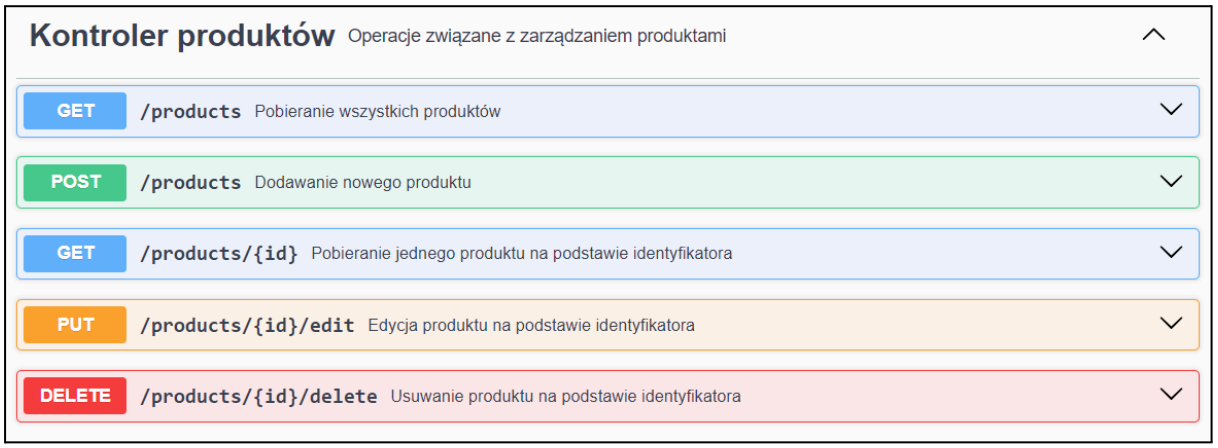
5.4 Kontrolery aplikacji serwerowej



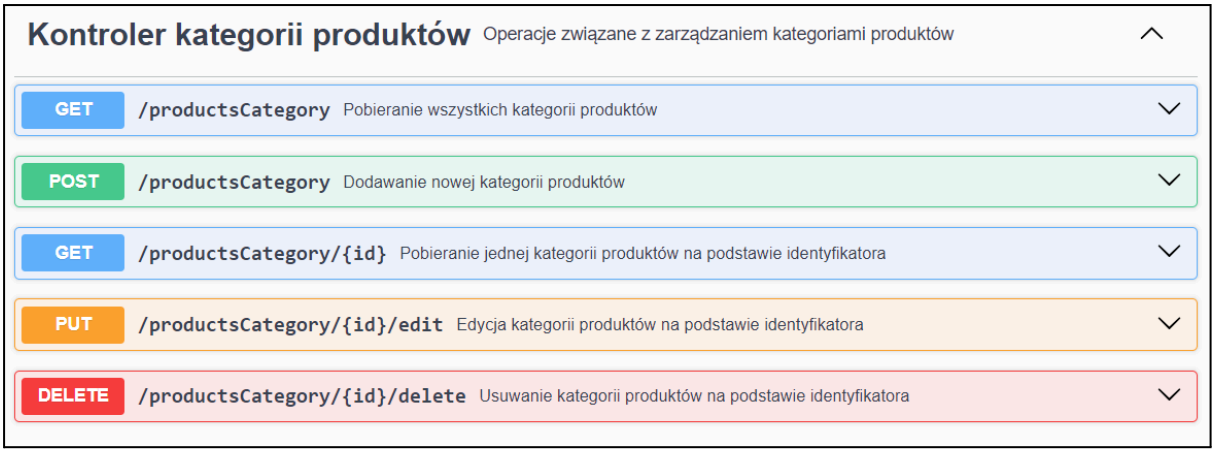
Rysunek 5.2 Kontroler autoryzacji [opracowanie własne]



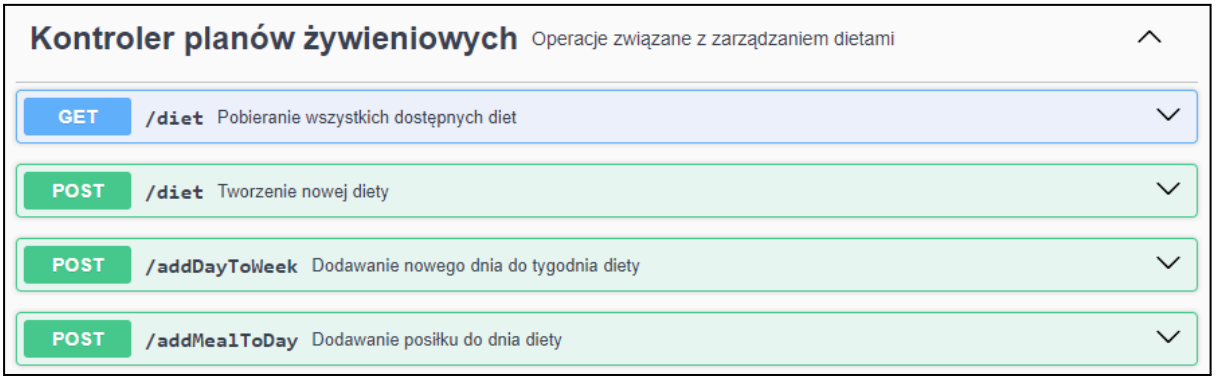
Rysunek 5.3 Kontroler użytkowników [opracowanie własne]



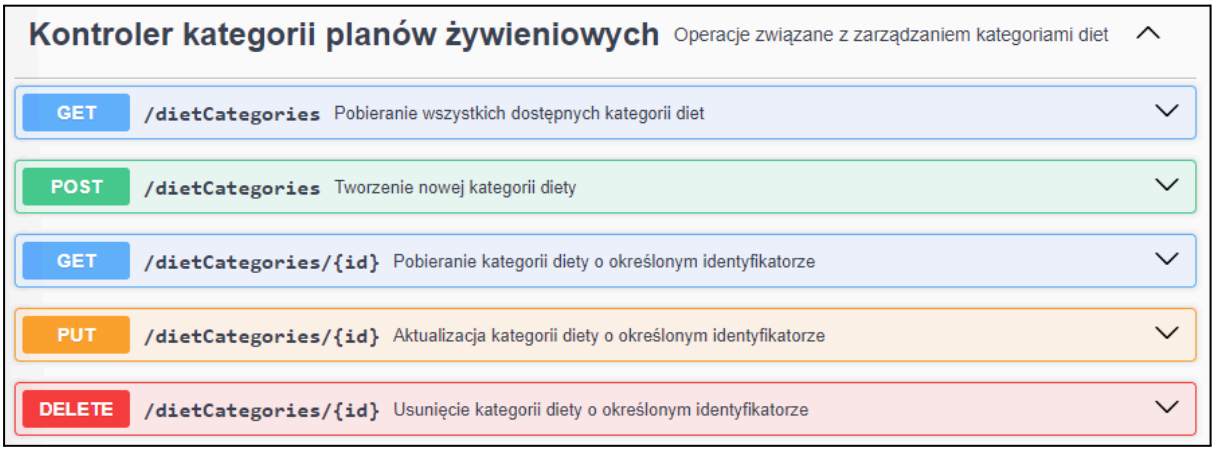
Rysunek 5.4 Kontroler produktów [opracowanie własne]



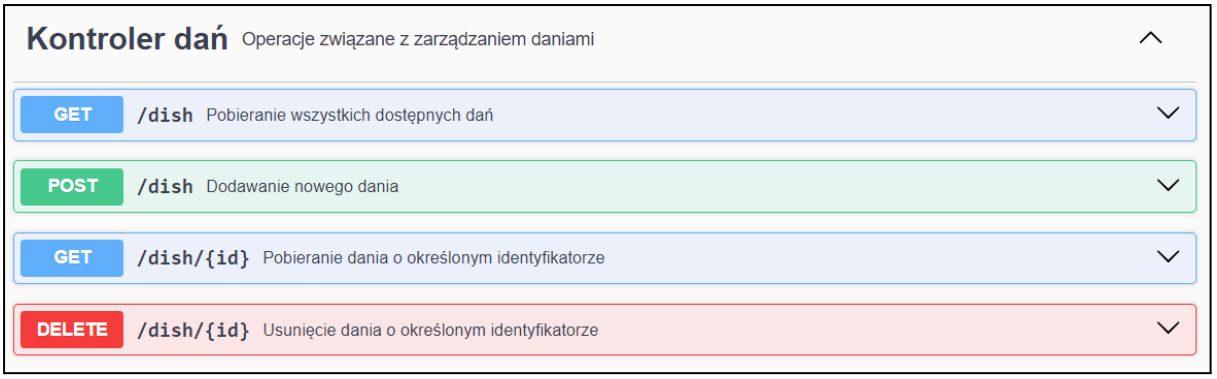
Rysunek 5.5 Kontroler kategorii produktów [opracowanie własne]



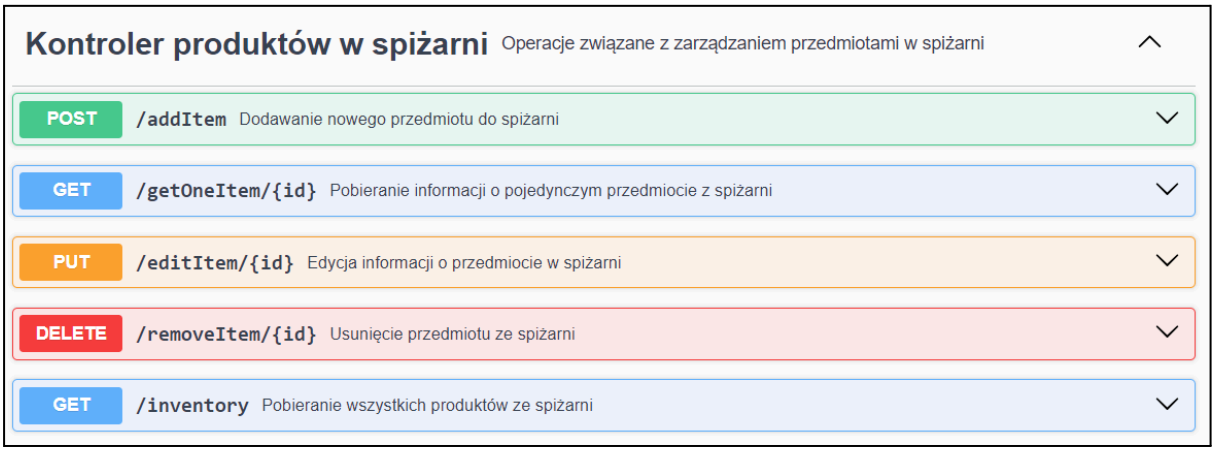
Rysunek 5.6 Kontroler planów żywieniowych [opracowanie własne]



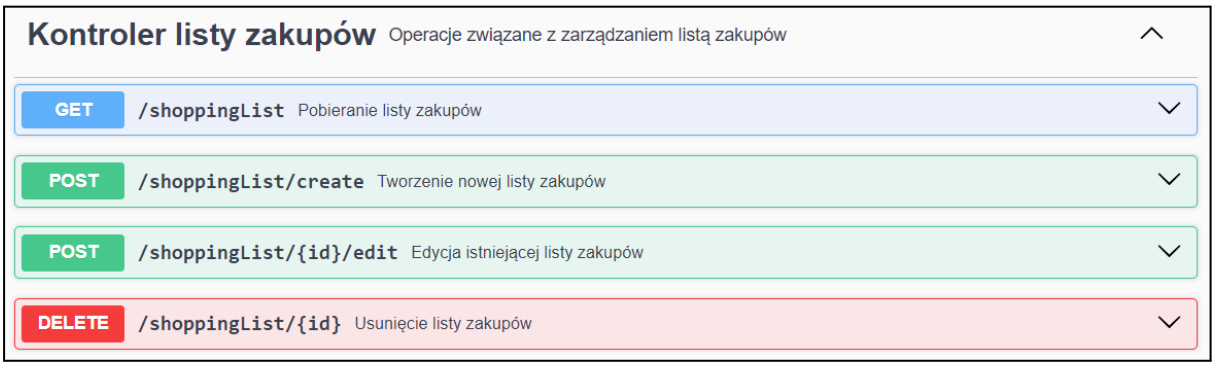
Rysunek 5.7 Kontroler kategorii planów żywieniowych [opracowanie własne]



Rysunek 5.8 Kontroler dań [opracowanie własne]



Rysunek 5.9 Kontroler produktów w spiżarni [opracowanie własne]



Rysunek 5.10 Kontroler listy zakupów [opracowanie własne]

5.3 Analiza wybranych elementów implementacyjnych

W tym rozdziale zostaną przedstawione kluczowe elementy implementacyjne. Przedstawione zostaną wybrane aspekty, które odgrywają kluczową rolę w funkcjonowaniu aplikacji.

5.3.1 Tworzenie nowego użytkownika

W celu umożliwienia użytkownikom pełnego wykorzystania potencjału aplikacji należy zarejestrować swoje konto w systemie. Aby proces rejestracji zakończył się pomyślnie, wymagane jest od użytkownika podanie podstawowych danych takich jak:

- Imię,
- Nazwisko,
- Nazwa użytkownika,
- Hasło.

Wprowadzenie tych danych polega na wypełnieniu formularza rejestracji, w którym podaje wymienione informacje, następnie aplikacja serwerowa odbiera żądanie od klienta i przystępuje do walidacji danych. Polega ona na sprawdzeniu, czy użytkownik wprowadził wymagane dane oraz weryfikuje, czy podany login użytkownika nie występuje już w bazie danych, jeśli istnieje, to zwraca komunikat o konieczności wybrania innej nazwy użytkownika.

Dla każdego użytkownika tworzony jest specjalny obszar w bazie danych nazywany spiżarnią, która służy do wirtualnego przechowywania produktów spożywczych. Nowo utworzona spiżarnia jest przypisana do użytkownika i zapisywana w bazie danych. W celu zabezpieczenia hasła użytkownika jest ono poddawane procesu szyfrowania za pomocą funkcji szyfrującej, która przekształca jawne hasło w niemożliwy do odczytania przez osoby postronne ciąg znaków.

Dodatkowo w celu zabezpieczenia procesu uwierzytelniania, stosowany jest mechanizm odświeżania tokena (refresh token). Jest on przypisywany przez serwer do konta użytkowników oraz jest przechowywany w sposób bezpieczny po stronie klienta, służy on do uzyskiwania nowego tokena dostępu, gdy ten pierwotny wygaśnie.

Listing 5.1 Rejestracja użytkownika [opracowanie własne]

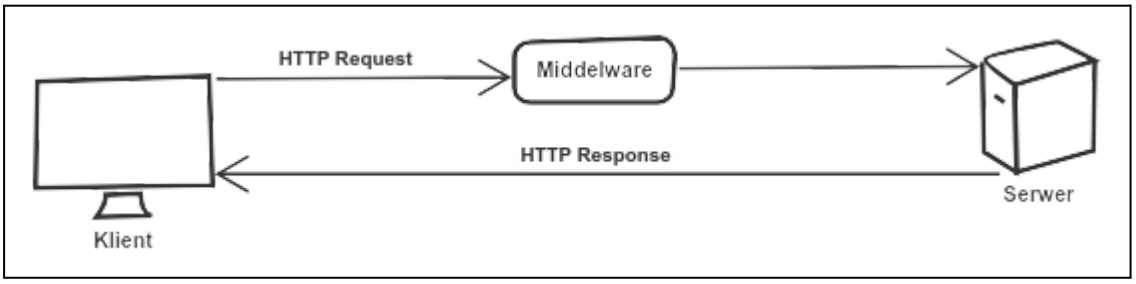
```
async handleNewUser(req: Request, res: Response) {
  const { firstname, lastname, username, password } = req.body;

  this.validUserData(res, firstname, lastname, username, password);
  this.checkDuplicateUser(res, username);
  const userInventory = Object.assign(new Inventory(), {
    items: []
  });

  const savedInventory = await this.inventoryRepository.save(userInventory);
  const hashedpwd = await bcrypt.hash(password, 10);
  const user = Object.assign(new User(), {
    firstname,
    lastname,
    username,
    password: hashedpwd,
    refreshToken: process.env.REFRESH_TOKEN_SECRET,
    inventory: savedInventory,
  });

  const newUser = this.userRepository.create(user);

  const savedUser = await this.userRepository.save(newUser);

  return res.status(201).json(savedUser);
}
```

Przedstawiony kod w listingu 5.1 implementuje obsługę żądania rejestracji nowego użytkownika w systemie. Proces ten rozpoczyna się od odebrania żądania w postaci podstawowych informacji od użytkownika. Po wstępnej walidacji danych oraz sprawdzeniu unikalności nazwy użytkownika tworzona jest instancja spiżarni w bazie danych, która w kolejnym etapie zostanie przypisana bezpośrednio do użytkownika. Realizowane jest również szyfrowanie hasła za pomocą funkcji **hash()** pochodzącej z biblioteki **bcrypt.** Ostatnim etapem jest interakcja z bazą danych i zapisanie użytkownika i jego spiżarni w postaci encji. Odpowiedź HTTP w przypadku pomyślnej rejestracji użytkownika zwraca kod statusu 201 (created).

5.3.2 Bezpieczeństwo zasobów

Nieodłącznym fundamentem każdej nowoczesnej aplikacji jest bezpieczeństwo. Obejmuje to kompleksowy zestaw praktyk mechanizmów, które skupiają się na ochronie danych użytkowników oraz integralności całej aplikacji.

W kontekście aplikacji serwerowej niezwykle użyteczne jest narzędzie middleware. Jest to funkcja, która posiada dostęp do obiektu żądania (ang. *request object*), obiektu odpowiedzi (*ang. response object*) oraz funkcji **next()**. Middleware jest wykonywany pomiędzy momentem otrzymania żądania a wysłaniem odpowiedzi, umożliwiając elastyczną manipulację danymi, przetwarzanie żądań oraz wdrażanie warstw zabezpieczeń.



Rysunek 5.11 Schemat middleware [opracowanie własne]

Obecna aplikacja implementuje użycie middleware **verifyRoles()**, sprawdza czy użytkownik wysyłający żądanie do serwera posiada dostęp do danego zasobu.

Listing 5.2 Middleware weryfikacja ról użytkownika [opracowanie własne]

```
import { Request, Response, NextFunction } from "express";

interface VerivyRoleRequest extends Request {
  roles?: string[];
}

export const verifyRoles = (...allowedRoles) => {
  return (req: VerivyRoleRequest, res: Response, next: NextFunction) => {
    if (!req?.roles) {
      return res.sendStatus(401);
    }
    const rolesArray = [...allowedRoles];

    const result = req.roles
      .map((role) => rolesArray.includes(role))
      .find((val) => val === true);
```

```
  if (!result) {
    return res.sendStatus(401);
  }
  next();
};};
```

Kolejnym elementem bezpieczeństwa jest zastosowanie middleware o nazwie **verifyJWT()**, które jest odpowiedzialne za proces uwierzytelniania z wykorzystaniem mechanizmu JWT (JSON Web Token). Funkcja przedstawiona na listingu 5.3 sprawdza, czy przekazany token JWT jest ważny. Jeżeli jest ważny, to funkcja dodaje informacje do obiektu żądania takie jak nazwa użytkownika oraz jego rola w systemie. Dzięki takiemu zastosowaniu każde kolejne zapytanie przechodzące przez ten middleware ma dostęp do tych informacji, co umożliwia kontrolę dostępu do zasobów na podstawie uwierzytelnionego użytkownika.

Listing 5.3 Middleware weryfikacja tokenów JWT [opracowanie własne]

```
import * as jwt from "jsonwebtoken";
import { Request, Response, NextFunction } from "express";

interface AuthRequest extends Request {
  username?: string;
  roles?: string[];
}

export const verifyJWT = (
  req: AuthRequest,
  res: Response,
  next: NextFunction
) => {
  const authHeader = Array.isArray(req.headers.authorization)
    ? req.headers.authorization[0]
    : req.headers.authorization;

  if (!authHeader?.startsWith("Bearer ")) return res.sendStatus(401);
  const token = authHeader.split(" ")[1];

  jwt.verify(token, process.env.ACCESS_TOKEN_SECRET, (err, decoded) => {
    if (err) {
      return res.sendStatus(403);
    }
    req.username = decoded.username;
    req.roles = decoded.UserInfo.roles;
```

```
    next();
  });
};
```

5.3.3 Dodawanie produktów do spiżarni

Jedną z podstawowych możliwości aplikacji jest śledzenie i monitorowanie dat przydatności produktów w spiżarni. Aby było to możliwe do zrealizowania, użytkownicy mogą dodawać produkty do swojej spiżarni. Dzięki tej funkcji system będzie posiadał odpowiednie dane do zarządzania inwentarzem spożywczym. Aby dodać produkt do spiżarni, należy skorzystać z dedykowanego punktu końcowego (ang. *endpoint*) w kontrolerze aplikacji znajdującego się na ścieżce "/addItem".

Listing 5.4 Ścieżka dodawania produktów [opracowanie własne]

```
{
  method: "post",
  route: "/addItem/:userId",
  controller: InventoryItemController,
  validation: [
    body("inventoryId").isInt(),
    body("productId").isInt(),
    body("purchaseDate").isNumeric(),
    body("expiryDate").isNumeric(),
    body("quantity").isInt(),
  ],
  action: "addItem",
  secure: true,
  roles: [UserRole.ADMIN, UserRole.MODERATOR, UserRole.USER],

}
```

Fragment kodu przedstawiony na listingu 5.4 definiuje ścieżkę API, która wykorzystuje metodę HTTP POST (przesłanie danych do serwera) na trasie "/addItem". Wskazuje kontroler oraz metodę odpowiedzialną za obsługę logiczną żądania. Określa wstępne reguły walidacji danych przesyłanych w ciele żądania (*ang. request body*). Informuje również o tym, czy dostęp do ścieżki wymaga uwierzytelniania. W tym przypadku do tej funkcji wymagane są role z uprawnieniami:

- Admin,
- Moderator,
- Użytkownik.

Listing 5.5 Kontroler dodawania produktów [opracowanie własne]

```
async addItem(req: Request, res: Response) {
  const { inventoryId, productId, purchaseDate, expiryDate, quantity } =
    req.body;
  const { userId } = parseInt(req.params.userId);
  const inventory = await this.inventoryRepository.findOne({
    where: {
      id: inventoryId,
      userId: userId
    },
    relations: {
      items: true,
    },
  });
  if (!inventory) {
    return res
      .status(404)
      .json({ message: `Inventory with _id: ${inventoryId} not found!` });
  }

  const product = await this.productRepository.findOne({
    where: { id: productId },
  });

  if (!product) {
    return res
      .status(404)
      .json({ message: `Product with _id: ${productId} not found!` });
  }

  const calculatePurchaseDate = purchaseDate ? purchaseDate : new Date();
  const calculateExpiryDate = expiryDate
    ? expiryDate
    : addDays(calculatePurchaseDate, product.shelfLifeDays);
  const inventoryItem = new InventoryItem();
  inventoryItem.product = product;
  inventoryItem.purchaseDate = calculatePurchaseDate;
  inventoryItem.expiryDate = calculateExpiryDate;
  inventoryItem.quantity = quantity;
  inventoryItem.usedQuantity = 0;

  inventory.items.push(inventoryItem);
```

```
  const savedInventoryItem = await this.inventoryItemRepository.save(
    inventoryItem
  );

  const saved = await this.inventoryRepository.save(inventory);

  return res.status(201).json({ saved, savedInventoryItem });
}
```

Kod kontrolera przedstawiony na listingu 5.5 obsługuje żądanie odpowiedzialne za dodanie produktów do spiżarni. Metoda **addItem()** przyjmuje dwa parametry obiekt żądania (ang. *request*) oraz odpowiedzi (ang. *response*). W ciele żądania (ang. *request body*) pobierane są informacje dotyczące numeru identyfikacyjnego spiżarni **inventoryId**, identyfikator produktu **productId**, którego chcemy dodać do naszej spiżarni oraz ilość danego produktu **quantity**. Również odbierane przez żądanie są data zakupu **purchaseDate** i data ważności **expiryDate**, które są przydatne w procesie monitorowania dat i świeżości produktów.

Kolejnym etapem jest weryfikacja, czy użytkownik próbuje skorzystać z właściwej spiżarni, aby uniknąć sytuacji, w której niepowołana osoba otrzyma dostęp do nie swoich zasobów. Sprawdzane jest istnienie spiżarni o podanym identyfikatorze i numerze identyfikacyjnym jej właściciela. Jeżeli spiżarnia nie istnieje lub niewłaściwy użytkownik próbuje dostać się do jej zasobów, zwracany jest komunikat o błędzie przerywający działanie metody **addItem()**. Następnym etapem jest obliczanie dat, w którym przewidziane zostały dwie możliwości:

- Jeśli nie podano daty zakupu, używana jest aktualna data.
- Jeśli nie podano daty ważności, jest ona obliczana na podstawie daty zakupu i okresu ważności produktu.

W kolejnym etapie tworzony jest obiekt **InventoryItem** reprezentujący nowy produkt dodawany do spiżarni, który jest dołączony do listy produktów w spiżarni, po czym następuje zapisanie produktu oraz aktualizacji spiżarni w bazie danych. Ostatnim etapem jest zwrócenie odpowiedzi z kodem 201 (Utworzono) wraz ze szczegółami zapisanych danych.

Proces ten umożliwia użytkownikom skuteczne zarządzanie zawartością swojej spiżarni oraz agregowania danych pomagających w monitorowaniu dat przydatności produktów, co pomaga w zminimalizowaniu marnowania żywności.

5.3.4 Ustalanie indywidualnego planu żywieniowego

Najważniejszą funkcją dostarczaną przez system jest możliwość ustalenia indywidualnej diety użytkownika, pozwala ona na szczegółowe zaplanowanie dziennego jadłospisu użytkownika zgodnie z jego preferencjami. Planowanie diety niesie za sobą wiele korzyści. Umożliwia dostosowanie posiłków do określonych potrzeb dziennego zapotrzebowania kalorycznego oraz makroskładników takich jak białko, węglowodany, tłuszcze. Dodatkowo proces planowania diety ułatwi organizację zakupów spożywczych, co będzie miało pozytywny wpływ na zmniejszenie ilości marnowanego jedzenia. To z kolei umożliwia użytkownikowi zredukować koszt zakupów spożywczych.

Na listingu 5.6 przedstawiona została asynchroniczna metoda **createDiet()**, która przyjmuje następujące dane z żądania:

- name – nazwa diety użytkownika,
- description – krótki opis diety,
- durationWeeks – długość trwania diety wyrażona w tygodniach,
- caloriesPerDay – dzienne zapotrzebowanie kaloryczne,
- dietCategoryId – numer identyfikacyjny kategorii diety, który użytkownik chce przyporządkować do swojego planu żywieniowego.

Na podstawie danych przekazanych w żądaniu tworzony jest obiekt reprezentujący dietę w bazie danych. Następnie w pętli, w sposób iteracyjny tworzone są obiekty reprezentujące poszczególne tygodnie na podstawie zmiennej **durationWeeks**. Utworzone tygodniowe diety są następnie zapisywane w bazie danych. Na koniec kontroler zwraca odpowiedź w formie obiektu JSON zawierającego informację o sukcesie operacji lub niepowodzeniu.

Listing 5.6 Kontroler tworzenia diety [opracowanie własne]

```
 async createDiet(req: Request, res: Response) {
  const { name, description, durationWeeks, caloriesPerDay, dietCategoryId } =
    req.body;
  const diet = this.dietRepository.create({
    name,
    description,
    durationWeeks,
    caloriesPerDay,
    dietCategory: { id: dietCategoryId },
  });
  const createdDiet = await this.dietRepository.save(diet);
  const weeklyDiets = [];
  for (let i = 1; i <= durationWeeks; i++) {
    const weekName = `Tydzień ${i}`;
    const weeklyDiet = this.weeklyDietRepository.create({
      weekName,
      diet: { id: createdDiet.id },
    });
    weeklyDiets.push(weeklyDiet);
  }
  const savedWeeklyDiet = await this.weeklyDietRepository.save(weeklyDiets);

  res.status(201).json({
    message: "Diet created successfully",
    createdDiet,
    savedWeeklyDiet,
  });
 }
```

Listing 5.7 przedstawia przykładowy obiekt żądania JSON dla punktu końcowego (ang. *endpoint*) "/diet".

Listing 5.7 Obiekt żądania dla procesu tworzenia diety [opracowanie własne]

```
{
  "name": "Noworoczna dieta",
  "description": "Dieta redukcyjna na nowy rok",
  "durationWeeks": 4,
  "caloriesPerDay": 1200,
  "dietCategoryId": 1
}
```

Natomiast listingu 5.8 przedstawia strukturę obiektu odpowiedzi JSON, który jest zwracany po udanym utworzeniu diety wraz z poszczególnymi tygodniami

Listing 5.8 Obiekt odpowiedzi dla procesu tworzenia diety [opracowanie własne]

```
{
    "message": "Diet created successfully",
    "createdDiet": {
        "name": "Noworoczna dieta",
        "description": "Dieta redukcyjna na nowy rok",
        "durationWeeks": 4,
        "caloriesPerDay": 1200,
        "dietCategory": {
            "id": 1 },
        "id": 1
    }, "savedWeeklyDiet": [
        {
            "weekName": "Tydzień 1",
            "diet": {
                "id": 1
            },
            "id": 1
        },
        {
            "weekName": "Tydzień 2",
            "diet": {
                "id": 1
            },
            "id": 2
        },
        {
            "weekName": "Tydzień 3",
            "diet": {
                "id": 1
            },
            "id": 3
        },
        {
            "weekName": "Tydzień 4",
            "diet": {
                "id": 1
            },
            "id": 4
     }
    ] }
```

Następnym krokiem w procesie tworzenia indywidualnego planu żywieniowego jest dodanie szczegółowych informacji do istniejącej już diety. Listing 5.9 prezentuje obiekt żądania JSON określający identyfikatory tygodni **weeklyDietsIds**, które chcemy zaplanować, datę rozpoczęcia diety **startDate**, dzienne zapotrzebowanie białka **totalProtein**, węglowodanów **totalCarbs**, oraz dzienne zapotrzebowanie tłuszczów **totalFat.**

Listing 5.9 Szczegółowe dane diety [opracowanie własne]

```
{
 "weeklyDietIds": [1],
 "startDate": "2024-01-01",
 "totalProtein": 160,
 "totalCarbs": 288,
 "totalFat": 55
}
```

Na listingu 5.10 przedstawiony został kod kontrolera odpowiedzialny za dodanie szczegółowych informacji dla każdego dnia. Dostęp do wspomnianej metody możliwy jest pod ścieżką "/addDayToWeek".

Listing 5.10 Kod kontrolera dodający szczegóły diety [opracowanie własne]

```
async addDayToWeek(req: Request, res: Response) {
  const {
    weeklyDietIds,
    startDate,
    totalCalories,
    totalProtein,
    totalCarbs,
    totalFat,
  } = req.body;

  const polishDaysOfWeek = [
    "Poniedziałek",
    "Wtorek",
    "Środa",
    "Czwartek",
    "Piątek",
    "Sobota",
    "Niedziela",
  ];

  const newDays = [];
  let currentDate = new Date(startDate);

  const startDayOfWeek = new Date(currentDate);
  startDayOfWeek.setDate(startDayOfWeek.getDate() - startDayOfWeek.getDay());

  for (let i = 0; i < weeklyDietIds.length; i++) {
    const weeklyDietId = weeklyDietIds[i];
    const weeklyDiet = await this.weeklyDietRepository.findOne({
      where: {
```

```
        id: weeklyDietId,
      },
    });

    if (!weeklyDiet) {
      return res.status(404).json({ error: "WeeklyDiet not found" });
    }

    currentDate = new Date(startDayOfWeek);


    for (let dayOfWeek = 1; dayOfWeek <= 7; dayOfWeek++) {
      const dayName = polishDaysOfWeek[dayOfWeek - 1];

      const newDay = this.dailyDietRepository.create({
        weeklyDiet,
        dayOfWeek: dayName,
        date: currentDate,
        totalCalories,
        totalProtein,
        totalCarbs,
        totalFat,
      });

      newDays.push(newDay);
      currentDate.setDate(currentDate.getDate() + 1);
    }

    startDayOfWeek.setDate(startDayOfWeek.getDate() + 7);
  }

  await this.dailyDietRepository.save(newDays);

  res.status(201).json({
    message: "Days added to the weeks successfully",
    newDays,
  });
}
```

Po pomyślnym wykonaniu żądania kodu kontrolera z listingu 5.10 serwer zwraca obiekt JSON, który zawiera potwierdzenie dodania szczegółowych informacji dla każdego dnia diety. Szczegółowa struktura obiektu JSON została przedstawiona na Listing 5.11.

Listing 5.11 Obiekt odpowiedzi szczegółów diety [opracowanie własne]


```
{
  "message": "Days added to the weeks successfully",
  "newDays": [
    {
      "dayOfWeek": "Poniedziałek",
      "date": "2024-01-07T00:00:00.000Z",
      "totalCalories": 2300,
      "totalProtein": 160,
      "totalCarbs": 288,
      "totalFat": 55,
      "weeklyDiet": {
        "id": 1,
        "weekName": "Tydzień 1"
      },
      "id": 1071
    },
    {
      "dayOfWeek": "Wtorek",
      "date": "2024-01-07T00:00:00.000Z",
      "totalCalories": 2300,
      "totalProtein": 160,
      "totalCarbs": 288,
      "totalFat": 55,
      "weeklyDiet": {
        "id": 1,
        "weekName": "Tydzień 1"
      },
      "id": 1072
    },
    {
      "dayOfWeek": "Środa",
      "date": "2024-01-07T00:00:00.000Z",
      "totalCalories": 2300,
      "totalProtein": 160,
      "totalCarbs": 288,
      "totalFat": 55,
      "weeklyDiet": {
        "id": 1,
        "weekName": "Tydzień 1"
      },
      "id": 1073
    },
    {
      "dayOfWeek": "Czwartek",
      "date": "2024-01-07T00:00:00.000Z",
      "totalCalories": 2300,
      "totalProtein": 160,
```

```
      "totalCarbs": 288,
      "totalFat": 55,
      "weeklyDiet": {
        "id": 1,
        "weekName": "Tydzień 1"
      },
      "id": 1074
    },
    {
      "dayOfWeek": "Piątek",
      "date": "2024-01-07T00:00:00.000Z",
      "totalCalories": 2300,
      "totalProtein": 160,
      "totalCarbs": 288,
      "totalFat": 55,
      "weeklyDiet": {
        "id": 1,
        "weekName": "Tydzień 1"
      },
      "id": 1075
    },
    {
      "dayOfWeek": "Sobota",
      "date": "2024-01-07T00:00:00.000Z",
      "totalCalories": 2300,
      "totalProtein": 160,
      "totalCarbs": 288,
      "totalFat": 55,
      "weeklyDiet": {
        "id": 1,
        "weekName": "Tydzień 1"
      },
      "id": 1076
    },
    {
      "dayOfWeek": "Niedziela",
      "date": "2024-01-07T00:00:00.000Z",
      "totalCalories": 2300,
      "totalProtein": 160,
      "totalCarbs": 288,
      "totalFat": 55,
      "weeklyDiet": {
        "id": 1,
        "weekName": "Tydzień 1"
      },
      "id": 1077
    }
  ]
}
```

Ostatnim krokiem w procesie ustalania indywidualnego planu żywieniowego jest wybór i przyporządkowanie odpowiednich dań do poszczególnych posiłków oraz dni, z uwzględnieniem określonych wymagań żywieniowych i preferencji użytkownika. Na listingu 5.12 przedstawiony został kod kontrolera odpowiedzialny za przyporządkowanie dań do poszczególnych dni. Kontroler ten znajduje się na ścieżce "/addMealToDay" oraz otrzymuje trzy argumenty:

- dailyDietId – identyfikator danego dnia w diecie,
- mealType – rodzaj posiłku np. śniadanie, obiad, kolacja,
- dishIds – identyfikatory dań, które użytkownik zamierza przyporządkować do danego posiłku.

Listing 5.12 Kod kontrolera dodawania posiłku [opracowanie własne]

```
async addMealToDay(req: Request, res: Response) {
  const { dailyDietId, mealType, dishIds } = req.body;

  // Sprawdź, czy istnieje taki dzień diety
  const dailyDiet = await this.dailyDietRepository.findOne({
    where: {
      id: dailyDietId,
    },
    relations: {
      dailyMeals: true,
    },
  });
  if (!dailyDiet) {
    return res.status(404).json({ error: "DailyDiet not found" });
  }

  const existingMeal = dailyDiet.dailyMeals.find(
    (meal) => meal.mealType === mealType
  );
  if (existingMeal) {
    return res.status(400).json({
      error: `Meal of type ${mealType} already exists for this day`,
    });
  }

  // Pobierz dania na podstawie przesłanych identyfikatorów
  const dishes = await this.dishRepository.findBy({
    id: In([...dishIds]),
  });

  // Dodaj nowy posiłek do danego dnia
```

```
  const newMeal = this.dailyMealRepository.create({
    dishes,
    mealType,
    dailyDiet,
  });
  const savedMeal = await this.dailyMealRepository.save(newMeal);

  res
    .status(201)
    .json({ message: "Meal added to the day successfully", savedMeal });
}
```

Na listingu 5.13 przedstawiony został przykładowy obiekt żądania JSON, który jest wysyłany na serwer za pośrednictwem ścieżki "/addMealToDay"

Listing 5.13 Obiekt żądania dla procesu dodawania posiłku [opracowanie własne]

```
{
 "dailyDietId": 1,
 "mealType": "BREAKFAST",
 "dishIds": [1, 2]
}
```

Pomyślna odpowiedź serwera zawiera obiekt JSON potwierdzający dodanie posiłku do danego dnia, wraz ze szczegółami. Na listingu 5.14 znajduje się struktura tego obiektu.

Listing 5.14 Obiekt odpowiedzi dla procesu dodawania posiłku [opracowanie własne]

```
{
  "message": "Meal added to the day successfully",
  "savedMeal": {
    "mealType": "Śniadanie",
    "dailyDiet": {
      "id": 1,
      "dayOfWeek": "Poniedziałek",
      "date": "2023-12-31T23:00:00.000Z",
      "totalCalories": 1500,
      "totalProtein": 100,
      "totalCarbs": 200,
      "totalFat": 50,
      "dailyMeals": []
    },
    "dishes": [
```

```
        {
          "id": 1,
          "name": "Sałatka z kurczakiem",
          "description": "Pyszna sałatka z kawałkami kurczaka",
          "totalCalories": 0,
          "totalProtein": 0,
          "totalCarbs": 0,
          "totalFat": 0
        },
        {
          "id": 2,
          "name": "Sałatka z kurczakiem",
          "description": "Pyszna sałatka z kawałkami kurczaka",
          "totalCalories": 0,
          "totalProtein": 0,
          "totalCarbs": 0,
          "totalFat": 0
        }
      ],
      "id": 1
    }
}
```

6. Implementacja bazy danych

Podstawowym zadaniem bazy danych jest przechowywanie informacji związanych z aplikacją serwerową. Tworzona struktura obejmuje tabele, relacje, które składają się na logiczną organizację danych. W ramach tego rozdziału zostanie przedstawiony sposób implementacji tabel wraz z relacjami, wykorzystując bibliotekę TypeORM do mapowania obiektowo-relacyjnego. Dodatkowo w procesie deweloperskim wykorzystano narzędzie **Docker**, które ułatwia zarządzanie instancją bazy danych.

6.1 Konfiguracja bazy danych

Kod na listingu 6.2 przedstawia plik konfiguracyjny Docker Compose definiujący usługę kontenera PostgreSQL zapisany w formacie YAML (ang. Yet Another Markup Language). Uruchomienie kontenera odbywa się za pomocą komendy w terminalu Listing 6.1

Listing 6.1 Komenda uruchamiająca narzędzie docker [opracowanie własne]

```
docker-compose up - d
```

Listing 6.2 Plik konfiguracyjny docker-compose [opracowanie własne]

```
version: "3"
services:
  postgres:
    image: postgres
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
    ports:
      - "5432:5432"
```

Elementy pliku konfiguracyjnego:

- version – określa wersję składni pliku Docker Compose,
- services – definiuje listę usług, w tym przypadku jest to postgres,
- image – określa obraz kontenera, który został użyty w projekcie. W tym przypadku jest to PostgreSQL,

- environment – zdefiniowane zmienne środowiskowe dla kontenera PostgreSQL, nazwa użytkownika i hasło,
- ports – ustawia połączenie portów pomiędzy kontenerem a hostem, umożliwiając komunikację aplikacji z bazą danych na porcie 5432.

6.2 Tabele

Implementacja tabel w bazie danych odbywa się za pomocą biblioteki TypeORM, która umożliwia definiowanie modeli danych za pomocą języka TypeScript. Każda tabela w bazie danych reprezentowana jest jako klasa, nazywana również encją. Encja posiada pola, które są odpowiednikami kolumn w bazie danych. Wspomniana biblioteka wykorzystuje również dekoratory, które informują o szczegółach struktur bazy danych.

Na listingu 6.3 przedstawiona została encja User, która odzwierciedla strukturę danych użytkownika. Klasa reprezentująca tabelę użytkownika ozdobiona jest dekoratorami, które dostarczają informacje o przyporządkowaniu pól do kolumn w bazie danych oraz relacjach między nimi. Poniżej opis wykorzystanych dekoratorów w ramach aplikacji:

- @Entity(“user”) – określa, że klasa User jest encją, czyli reprezentacją tabeli w bazie danych.
- @PrimaryGeneratedColumn() – oznacza, że pole id (numer identyfikacyjny użytkownika) jest automatycznie generowanym identyfikatorem głównym.
- @Column() – dekorator dla pól, które reprezentują kolumny w bazie danych, może również przyjmować taką opcję jak “nullable”, która określa czy pole może przyjmować wartość null.
- @CreateDateColumn() – automatycznie ustawia wartość pola na datę utworzenia rekordu w bazie danych.
- @UpdateDateColumn – ustawia wartość pola na datę ostatniej aktualizacji rekordu.
- @JoinColumn() – wskazuje, że w tabeli bazy danych dla danej encji, będzie używana kolumna łącząca, przechowująca klucz obcy do powiązanej encji.
- @OneToOne(() => Inventory, { cascade: true, eager: true }) – określa relację jeden do jeden z encją Inventory.

- “Cascade” ustawione na wartość “true” oznacza to, że operacje wykonywane na obiekcie nadrzędnym (User), takie jak zapisywanie będą również dotyczyć obiektu zależnego (Inventory).
- “Eager” ustawione na wartość “true” oznacza, że związane dane (w tym przypadku Inventory) są ładowane automatycznie przy pobieraniu obiektu nadrzędnego (User).

Dekoratory dostarczają TypeORM informacji na temat struktury bazy danych i relacji pomiędzy encjami, co znacznie ułatwia definiowanie struktury bazy danych oraz poprawia czytelność dzięki zastosowaniu języka TypeScript.

Listing 6.3 Schemat encji użytkownika w TypeORM [opracowanie własne]

```
import { Entity,PrimaryGeneratedColumn, Column, CreateDateColumn,
UpdateDateColumn, OneToOne, JoinColumn, OneToMany,
} from "typeorm";
import { UserDetails } from "./UserDetails";
import { UserRole } from "../../enums/UserRole";
import { Inventory } from "../inventory/Inventory";

@Entity("user")
export class User {
 @PrimaryGeneratedColumn()
 id: number;

 @Column()
 firstname: string;

 @Column()
 lastname: string;

 @Column()
 username: string;

 @Column()
 password: string;

 @Column({ nullable: true })
 refreshToken: string;

 @CreateDateColumn()
 createdAt: Date;

 @UpdateDateColumn()
 updatedAt: Date;
```

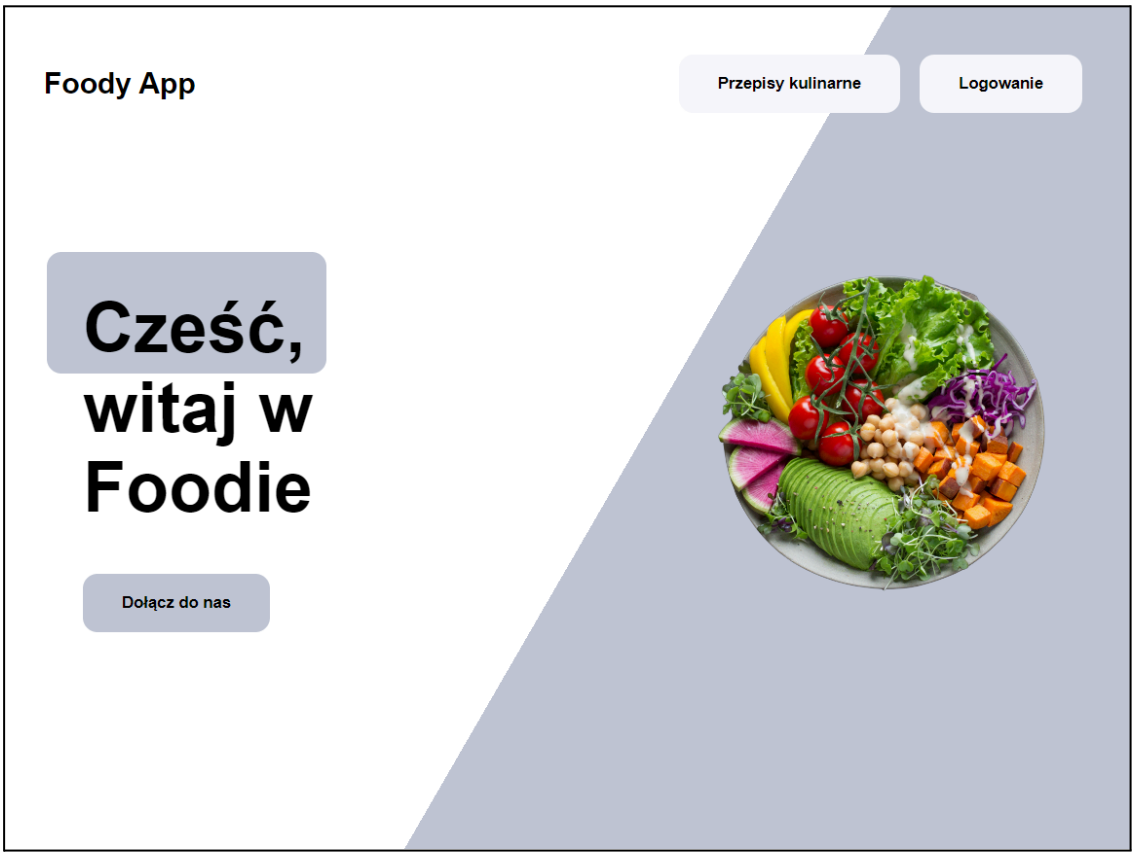
```
  @Column({
    type: "enum",
    enum: UserRole,
    array: true,
    default: [UserRole.USER],
  })
  roles: UserRole[];

  @OneToOne(() => Inventory, { cascade: true, eager: true })
  @JoinColumn()
  inventory: Inventory;

  @OneToOne(() => UserDetails, { cascade: true, eager: true })
  @JoinColumn()
  user_details: UserDetails;
}
```

TypeORM jest obszernym narzędziem, integrującym się z językiem TypeScript, oferuje deklaratywną składnię oraz przyjazne dla programisty dekoratory do definiowania struktury bazy danych.

7. Implementacja interfejsu użytkownika

W tym rozdziale omówiony zostanie interfejs użytkownika aplikacji. Przedstawiony zostanie wygląd aplikacji, opis poszczególnych widoków, możliwości spełniające wymagania funkcjonalne oraz oczekiwania użytkowników.

7.1 Ekran powitalny



Rysunek 7.1 Ekran powitalny [opracowanie własne]

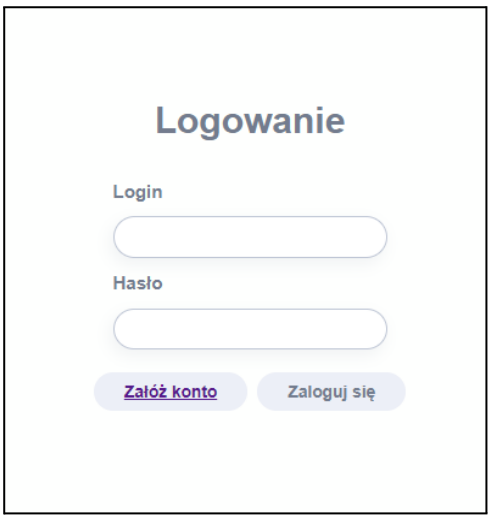
Na rysunku 7.1 przedstawiony został ekran powitalny (ang. *landing page*), którego celem jest przyciągnięcie uwagi użytkownika oraz zachęcenie do skorzystania z usług odwiedzanej aplikacji internetowej. Ekran ten charakteryzuje się minimalistycznym podejściem, który współgra z tematyką aplikacji. W centralnej części umieszczony został wyraźny tytuł wraz z przyciskiem przenoszącym użytkownika do panelu rejestracji w systemie oraz ilustracja talerza zawierającego pełnowartościowe produkty zachęcając do podejmowania zdrowych nawyków żywieniowych.

7.2 Codzienne propozycje kulinarne

Rysunek 7.2 Przepisy kulinarne [opracowanie własne]

Rysunek 7.2 przedstawia sekcję będącą kontynuacją ekranu powitalnego. Sekcja ta została zaprojektowana w celu dodatkowego zachęcenia użytkownika do skorzystania z usług aplikacji. W tym miejscu każdy niezalogowany użytkownik codziennie może znaleźć i wykorzystać kilka starannie przygotowanych przepisów, które nawiązują do hasła "Zrób coś z niczego". Przycisk **Podgląd** umożliwia pełny wgląd w wybrany przepis, który zawiera kompletne informacje potrzebnych do ugotowania smacznej potrawy.

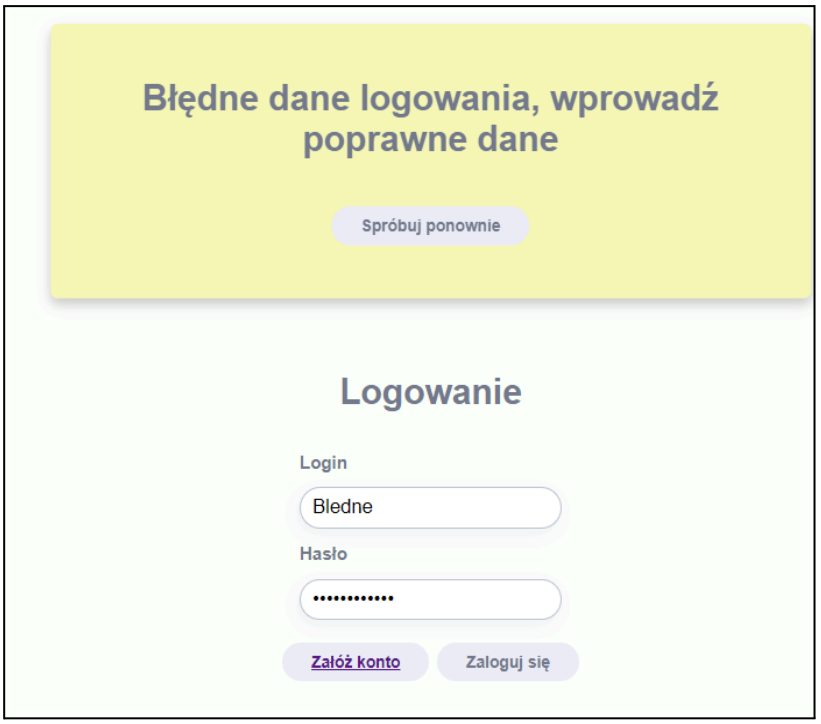
7.3 Ekran logowania



Rysunek 7.3 Ekran logowania [opracowanie własne]

Na rysunku 7.3 znajduje się ekran logowania. Stanowi on kluczowy element interakcji użytkownika z aplikacją. Zadaniem użytkownika jest wprowadzenie poprawnych danych logowania. Po wprowadzeniu danych inicjowany jest proces uwierzytelniania. Na tym etapie aplikacja sprawdza zgodność wprowadzonych danych z danymi przechowywanymi w bazie danych. W celu zabezpieczenia transmisji danych wykorzystywany jest standard JWT (JSON Web Token), co gwarantuje bezpieczne zarządzanie sesjami logowania. Po pomyślnym uwierzytelnianiu użytkownik zostaje automatycznie przekierowany do spersonalizowanego konta.

7.4 Ekran nieudanego logowania



Rysunek 7.4 Ekran nieudanego logowania [opracowanie własne]

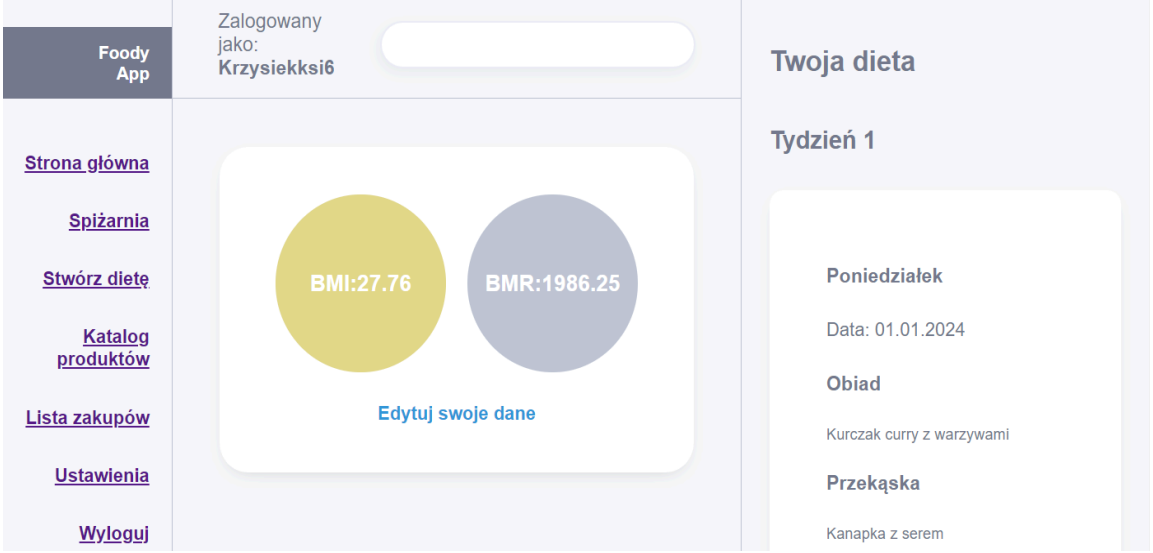
W przypadku nieudanego logowania użytkownikowi ukazuje się okno informujące o błędzie (rysunek 7.4) zachęcające do ponownej próby logowania.

7.5 Ekran rejestracji użytkownika

Rysunek 7.5 przedstawia ekran rejestracji użytkownika w systemie. Na tym ekranie nowi użytkownicy mają możliwość wprowadzenia niezbędnych informacji takich jak imię, nazwisko, unikalny login i hasło. Proces rejestracji jest intuicyjny oraz bezpieczny dla użytkownika. Istotnym elementem jest zastosowanie szyfrowania hasła po stronie serwera za pomocą algorytmu bcrypt. Po pomyślnym procesie rejestracji użytkownik zostaje przekierowany do ekranu logowania.

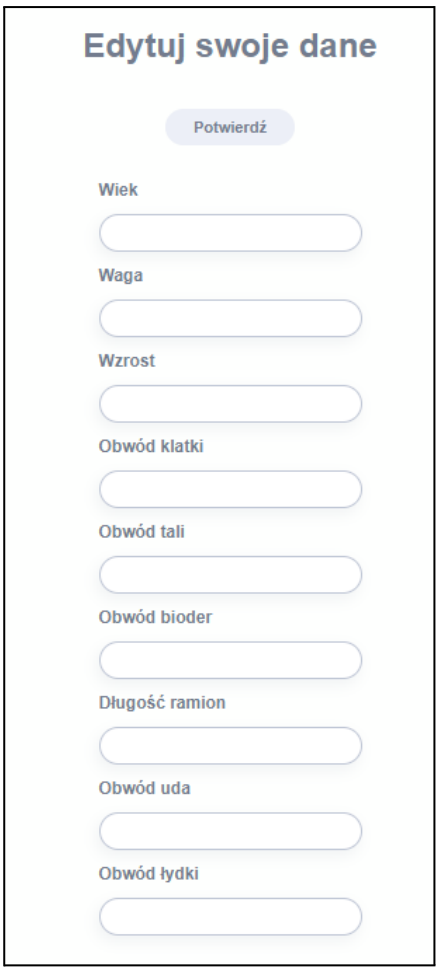
Rysunek 7.5 Ekran rejestracji użytkownika [opracowanie własne]

7.6 Ekran główny aplikacji



Rysunek 7.6 Ekran główny aplikacji [opracowanie własne]

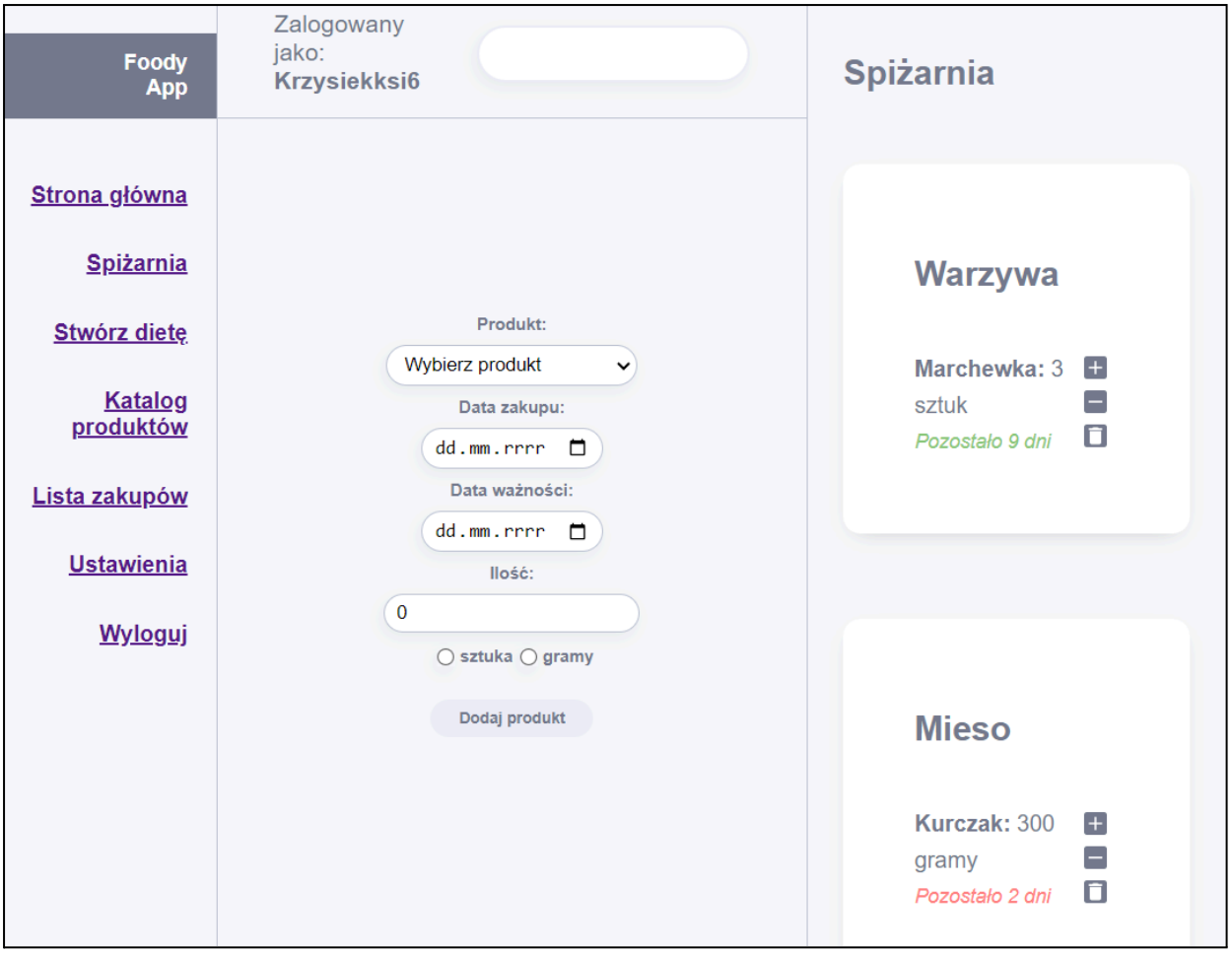
Po poprawnym procesie logowania użytkownik uzyskuje dostęp do głównego panelu aplikacji przedstawionego na rysunku 7.6. Po lewej stronie znajduje się nawigacja zapewniająca szybki dostęp do poszczególnych modułów aplikacji. W środkowej części panelu głównego prezentowane są kluczowe dane użytkownika, wskaźnik BMI oraz BMR. Warto dodać, że w każdym momencie użytkownik może skorzystać z panelu edycji znajdującego się na rysunku 7.7. Te informacje stanowią podstawę dla spersonalizowanej analizy stanu zdrowia. Po prawej stronie panelu użytkownik ma dostęp do spersonalizowanego planu diety na poszczególne dni wraz z możliwością podglądu szczegółów danego dnia.



Rysunek 7.7 Ekran dodawania szczegółowych użytkownika [opracowanie własne]

7.8 Spiżarnia

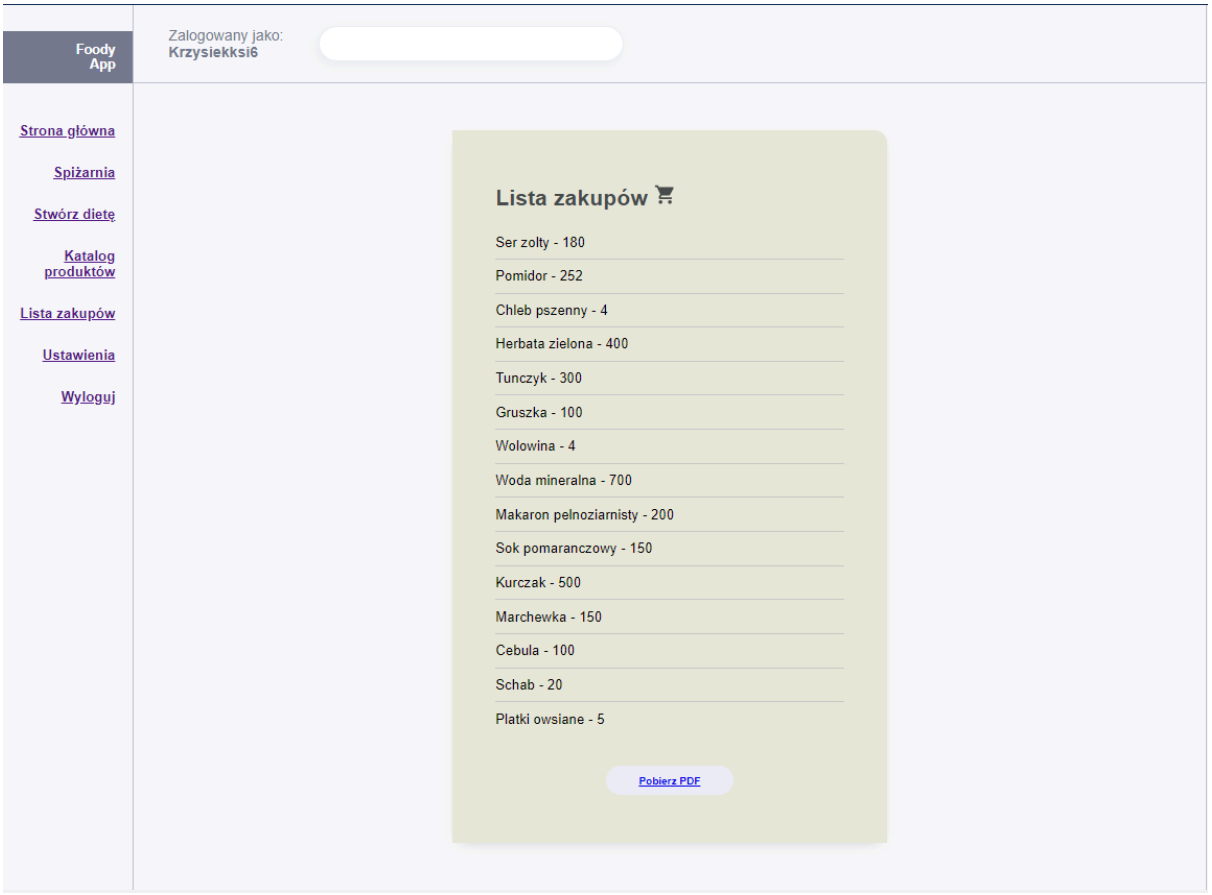
Jedną z podstawowych możliwości aplikacji jest zarządzanie domową spiżarnią. Rysunek 7.8 prezentuje widok panelu do zarządzania żywnością oraz przedstawia jej zawartość. Intuicyjny interfejs spiżarni pozwala szybko zorientować się w dostępnych zapasach żywności oraz dodawać produkty do spiżarni.



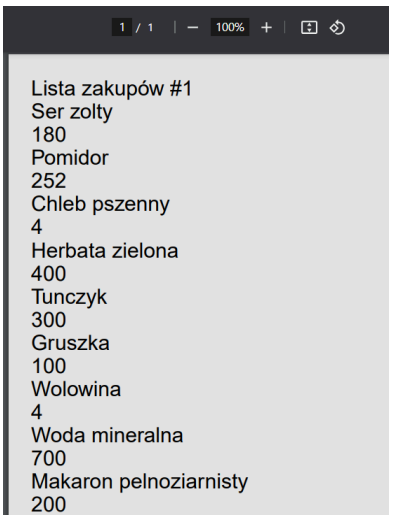
Rysunek 7.8 Ekran spiżarni [opracowanie własne]

7.9 Generowanie listy zakupów

Dodatkową funkcją dostępną w systemie jest możliwość generowania listy zakupów na podstawie ustalonej diety użytkownika. Dzięki temu narzędziu użytkownicy będą w stanie efektywnie planować posiłki, ograniczając jednocześnie wydatki na zakupy spożywcze. Rysunek 7.9 przedstawia ekran listy zakupów, dostępna jest również funkcja pobierania listy w formacie PDF (rysunek 7.10).



Rysunek 7.9 Ekran generowania listy zakupów [opracowanie własne]



Rysunek 7.10 Listy zakupów PDF [opracowanie własne]

8. Testy

W tym rozdziale przedstawiony zostanie proces testowania poszczególnych fragmentów aplikacji.

8.1 Testy jednostkowe

Testy jednostkowe służą do sprawdzenia pojedynczych fragmentów kodu, np. metod lub obiektów. Celem testów jest zweryfikowanie, czy zachowuje się on zgodnie z oczekiwaniami, pozwala również wykryć potencjalne błędy na etapie wytwarzania oprogramowania. Szczególne pomocne jest stosowanie techniki Test Driven Development (TDD), w której to testy są tworzone jeszcze przed implementacją kodu.

8.1.2 Test poprawności obliczeń wskaźnika Body Mass Index (BMI)

Użytkownik po pomyślnym zalogowaniu ma możliwość wprowadzenia szczegółowych danych o sobie, takie jak wzrost, wiek, waga. Są one wykorzystywane między innymi do monitorowania postępów w zakresie zdrowia kondycji fizycznej. Dodatkowo służą do obliczania wskaźnika masy ciała Body Mass Index (BMI). Wskaźnik ten pozwala ocenić proporcję między masą ciała a wzrostem użytkownika. Kluczowym elementem jest dokładne przetestowanie metody obliczającej współczynnik BMI. Ewentualne błędy wynikające z niepoprawnych obliczeń kalkulatora mogą przyczynić się do dezorientacji użytkowników, co mogłoby prowadzić do rezygnacji korzystania z platformy. Na listingu 8.1 , 8.2 i 8.3 przedstawiony został kod źródłowy testów kalkulatora BMI. W przypadku listingu 8.1 przetestowany został przypadek dla prawidłowej wagi. Listing 8.2 przedstawia kod dla przypadku użytkownika z niedowagą, natomiast listing 8.3 prezentuje kod dla przypadku użytkownika z nadwagą.

Listing 8.1 Test BMI dla wagi prawidłowej [opracowanie własne]

```
it("should calculate BMI correctly for normal weight", () => {
      // Given
      const weight = 70;
      const height = 175;
      // When
      const bmi = userDetailsController.calculateBMI(weight, height);
      // Then
      expect(bmi).toBeGreaterThanOrEqual(18.5);
      expect(bmi).toBeLessThan(24.9);});
```

Listing 8.2 Test BMI dla niedowagi [opracowanie własne]

```
it("TC_BMI_004: should calculate BMI correctly for underweight", () => {
      // Given
      const weight = 50;
      const height = 175;
      // When
      const bmi = userDetailsController.calculateBMI(weight, height);
      // Then
      expect(bmi).toBeLessThan(18.5);
    });
```

Listing 8.3 Test BMI dla nadwagi [opracowanie własne]

```
it("TC_BMI_005: should calculate BMI correctly for overweight", () => {
      // Given
      const weight = 90;
      const height = 175;
      // When
      const bmi = userDetailsController.calculateBMI(weight, height);
      // Then
      expect(bmi).toBeGreaterThan(24.9);
    });
```

Przedstawione testy zaprezentowane na listingach 8.1, 8.2, 8.3 uwzględniają konwencję pisania testów Given-When-Then:

- Given – określenie warunków początkowych,
- When – wywołanie funkcji obliczającej BMI,
- Then – sprawdzenie, czy obliczone BMI spełnia przyjęte kryteria.

```
$ npx jest bmi.test.ts
 PASS  __tests__/unit/bmi.test.ts (6.231 s)
  UsedDetailsController BMI Calculator
    Calculate BMI
      √ TC_BMI_003: Should calculate BMI correctly for normal weight (7 ms)
      √ TC_BMI_004: Should calculate BMI correctly for underweight (1 ms)
      √ TC_BMI_005: Should calculate BMI correctly for overweight

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        6.596 s, estimated 29 s
Ran all test suites matching /bmi.test.ts/i.
```

Rysunek 8.1 Rezultat testów kalkulatora BMI [opracowanie własne]

8.2 Testy integracyjne

Testy integracyjne służą do oceny współpracy i interakcji pomiędzy różnymi komponentami lub modułami aplikacji. Ich celem jest sprawdzenie, czy poszczególne elementy komunikują się w prawidłowy i oczekiwany sposób.

8.2.1 Test integracji aplikacji serwerowej z bazą danych

Jednym z kluczowych elementów funkcjonowania aplikacji jest prawidłowa komunikacja aplikacji serwerowej oraz bazy danych. W ramach testu integracyjnego przeprowadzone zostaną następujące testy:

- test połączenia aplikacji serwerowej z bazą danych,
- test zapisu trzech użytkowników do bazy danych,
- test pobrania informacji o wszystkich zarejestrowanych użytkownikach w systemie.

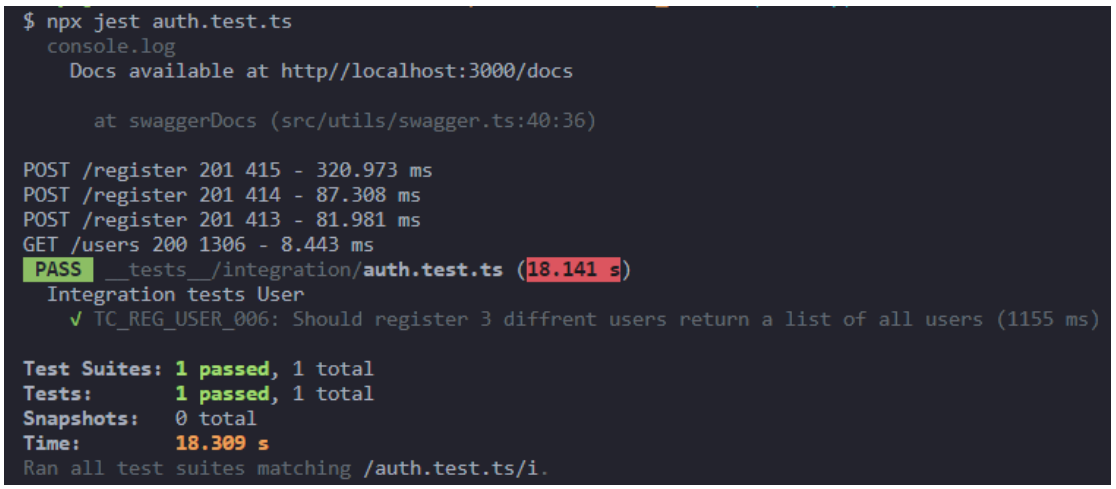
Listing 8.4 Test integracyjny [opracowanie własne]

```
let server: Server;
let myDataSource = connectDatabase;
describe("Integration tests User", () => {
  beforeEach(async () => {
    await myDataSource.connect();
    await myDataSource.synchronize(true);
    server = app.listen(config.port);
  });
  afterEach(async () => {
    server.close();
    await connectDatabase.getRepository(User).clear();
    myDataSource.close();
  });

  it("TC_REG_USER_006: Should register 3 diffrent users return a list of all users", async () => {
    // Given
    const mockUser1 = createMockUser();
    const mockUser2 = createMockUser();
    const mockUser3 = createMockUser();
    mockUser1.id = 1;
    mockUser1.username = "Moster";
    mockUser2.id = 2;
    mockUser2.username = "Albon";
    mockUser3.id = 3;
    mockUser3.username = "Thom";
```

```
    // When
    const registerResponse1 = await request(app).post("/register").send({
      firstname: mockUser1.firstname,
      lastname: mockUser1.lastname,
      username: mockUser1.username,
      password: mockUser1.password,
    });

    const registerResponse2 = await request(app).post("/register").send({
      firstname: mockUser2.firstname,
      lastname: mockUser2.lastname,
      username: mockUser2.username,
      password: mockUser2.password,
    });

    const registerResponse3 = await request(app).post("/register").send({
      firstname: mockUser3.firstname,
      lastname: mockUser3.lastname,
      username: mockUser3.username,
      password: mockUser3.password,
    });

    // Then
    expect(registerResponse1.status).toBe(201);
    expect(registerResponse2.status).toBe(201);
    expect(registerResponse3.status).toBe(201);

    // When
    const getUsersResponse = await request(app).get("/users");

    // Then
    expect(getUsersResponse.status).toBe(200);
    expect(getUsersResponse.body.length).toBe(3);
  });
});
```

Listing 8.4 przedstawia kod źródłowy testu integracyjnego. Pierwszym krokiem jest inicjalizacja obiektu serwera **server** oraz instancji bazy danych przechowywanej w obiekcie **myDataSource**. Wprowadzona została również konfiguracja **beforeEach(async () => {...}**, która odpowiedzialna jest za nawiązanie połączenia serwera z bazą danych przed rozpoczęciem każdego testu oraz konfiguracja **afterEach(async () => {...}**, która czyści zawartość tabel w bazie danych oraz zamyka połączenie aplikacji serwerowej z bazą danych. Kolejnym etapem jest utworzenie trzech różnych użytkowników (**mockUser1**, **mockUser2**,

mockUser3) oraz wysłanie trzech żądań HTTP typu POST na ścieżkę "/register" w celu zarejestrowania kont w systemie. Dla każdego z trzech żądań spodziewany kod odpowiedzi 201 (Utworzono). Na tym etapie w bazie danych zarejestrowane zostają trzy różne konta użytkowników z unikalnymi danymi. W celu potwierdzenia, czy utworzone konta użytkowników zostały poprawnie zapisane w bazie danych serwer wysyła żądanie HTTP typu GET na ścieżkę "/users" w celu pobrania listy użytkowników. Następnie sprawdzane jest, czy operacja pobierania użytkowników zakończyła się sukcesem oraz, czy pobrana lista użytkowników zawiera oczekiwane trzy elementy. Na rysunku 8.2 przedstawiony został rezultat omawianego testu.



Rysunek 8.2 Rezultat testów integracji serwera z bazą danych [opracowanie własne]

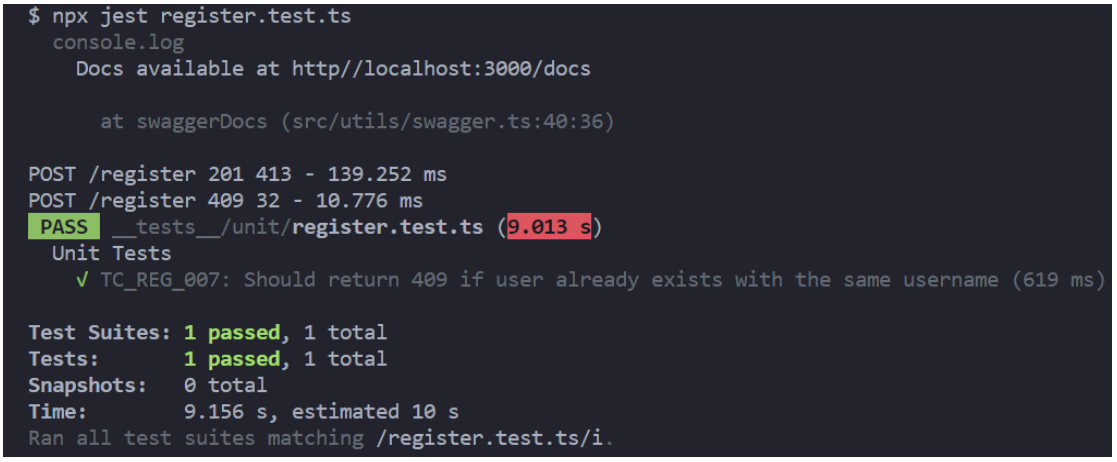
8.2.2 Test rejestracji użytkownika w aplikacji

Podczas procesu rejestracji konta w systemie użytkownik zobowiązany jest do podania podstawowych danych, między innymi imię, nazwisko, unikalną nazwę użytkownika i hasło. W celu zabezpieczenia przed sytuacją, w której dwóch użytkowników próbuje założyć konto z tą samą nazwą użytkownika, został opracowany i zaimplementowany test jednostkowy sprawdzający poprawność obsługi tego typu wyjątku. Na listingu 8.5 został przedstawiony kod sprawdzający, jak kontroler reaguje na próbę rejestracji użytkownika z istniejącym już loginem w bazie danych

Listing 8.5 Test rejestracji dwóch użytkowników o tym samym loginie [opracowanie własne]

```
it('TC_REG_007: Should return 409 if user already exists with the same username', async () => {
  // Rejestrowanie pierwszego użytkownika
  const firstUser = {
    firstname: 'John',
    lastname: 'Doe',
    username: 'john.doe',
    password: 'password123',
  };
  await request(app)
    .post('/register')
    .send(firstUser)
    .expect(201);

  // Rejestrowanie drugiego użytkownika z tym samym loginem (username)
  const duplicateUser = {
    firstname: 'Jane',
    lastname: 'Doe',
    username: 'john.doe', // To samo username co wcześniej
    password: 'anotherpassword',
  };
  await request(app)
    .post('/register')
    .send(duplicateUser)
    .expect(409);

});
```

W pierwszej fazie testu został utworzony obiekt **firstUser**, który posiada podstawowe dane do rejestracji, takie jak **firstname (**imię), **lastname** (nazwisko), **username (**unikalna nazwa użytkownika) i **password** (hasło). Następnym krokiem było przeprowadzenie rejestracji pierwszego użytkownika wysyłając odpowiednie zapytanie do ścieżki "/register" z danymi obiektu **firstUser**. Oczekiwana odpowiedź pierwszej rejestracji to kod HTTP 201 (Utworzono), co świadczy o pomyślnym utworzeniu pierwszego konta użytkownika w systemie. Drugi etap polegał na utworzeniu obiektu **duplicateUser**, który miał podobne dane do poprzedniego. Następnie ponownie zostało wysłane zapytanie do ścieżki "/register" z danymi **duplicateUser**. Oczekiwaną odpowiedzią jest kod stanu 409 (Konflikt), co sygnalizuje że istnieje konflikt związany z unikalnością nazwy użytkownika. Na rysunku 8.3 został przedstawiony wynik omawianego testu.



Rysunek 8.3 Rezultat testu rejestracji użytkownika [opracowanie własne]

9. Podsumowanie i wnioski

Głównym założeniem niniejszej pracy dyplomowej było zaimplementowanie aplikacji internetowej służącej do zarządzania zasobami żywnościowymi domowej spiżarni oraz dietą. Podczas realizacji tego projektu skoncentrowano się na zaprojektowaniu i zaimplementowaniu kompleksowego systemu aplikacji internetowej, aplikacji serwerowej oraz bazy danych.

9.1 Osiągnięte założenia

Projekt spełnił wszystkie z wymienionych wymagań funkcjonalnych oraz niefunkcjonalnych:

- Zarządzanie spiżarnią – zaimplementowano funkcję dodawania produktów wraz z podziałem na kategorie oraz możliwością wprowadzania daty zakupu w celu monitorowania świeżości produktu.
- Planowanie diety – wprowadzone zostało narzędzie do personalizowania planu żywieniowego. Funkcja przyczynia się do racjonalnego gospodarowania zasobami żywnościowymi.
- Generowanie listy zakupów – na podstawie ustalonego planu żywieniowego oraz aktualnego stanu spiżarni system pozwala na automatyczne generowanie listy zakupów, dzięki takiemu podejściu użytkownik nie tylko oszczędza czas na ręcznym tworzeniu listy zakupów, ale także minimalizuje ryzyko zakupu niepotrzebnych artykułów spożywczych.
- Wprowadzanie informacji o użytkowniku – kluczowy etap pozwalający na monitorowanie parametrów użytkownika takich jak podstawowa przemiana materii oraz wskaźnik masy ciała.
- Podgląd przepisów kulinarnych – istotna funkcja systemu umożliwiająca użytkownikom zapoznanie się z różnorodnymi przepisami kulinarnymi. Prezentowane przepisy charakteryzują się niskim kosztem składników oraz szybkością wykonania.

Osiągnięte założenia projektu stanowią solidny fundament dla efektywnego zarządzania zasobami żywnościowymi. Projekt spełnił wszystkie założone wymagania funkcjonalne i niefunkcjonalne, ale także dostarczył praktyczne narzędzia, które ułatwiają codzienne podejmowanie świadomych decyzji żywieniowych

9.2 Propozycje dalszych prac

W kontekście dalszego rozwoju projektu oraz poszerzenia zakresu dostępnych funkcji jest realizacja zintegrowanej z obecnym systemem aplikacji mobilnej. Nowsza wersja aplikacji mogłaby zawierać dodatkowo możliwość śledzenia aktywności fizycznej użytkownika, jak również posiadać dostęp do listy zakupów znajdującej się w internetowej wersji aplikacji. Dzięki tej funkcji użytkownik zawsze będzie miał dostęp do kluczowych informacji, co w szczególności ułatwi mu zakupy spożywcze.

10. Bibliografia

[1] Anthony Accomazzo, Nate Murray, Ari Lerner, Fullstack React, Fullstack.io, 2017

[2] React.js [Online] *https://react.dev/*

[3] Node.js [Online] *https://nodejs.org/en*

[4] Express [Online] *https://expressjs.com/en/api.html*

[5] PostgreSQL [Online] *https://www.postgresql.org/docs/*

[6] TypeORM [Online] *https://typeorm.io/*

[7] JWT [Online] *https://jwt.io/*

[8] Docker [Online] *https://docs.docker.com/*

[9] JSON [Online] *https://www.json.org/json-en.html*

[10] Andrew S. Tanenbaum, David J. Wetherall, Sieci komputerowe. Wydanie V 2012

[11] Mark Massé, REST API Design Rulebook 2012.

[12] Swagger [Online] *https://swagger.io/*

[13] Redux [Online] *https://redux.js.org/*

11. Spis tabel

12. Spis rysunków

13. Spis listingów