AKADEMIA TARNOWSKA
Wydział Politechniczny

Kierunek: ***Informatyka***

2024/2025

Arkadiusz Kupiec

PRACA INŻYNIERSKA

Aplikacja do zamawiania posiłków

Promotor pracy:

mgr inż. Tomasz Gądek

Tarnów, *2025*

Spis treści

1. Wstęp

Przedmiotem niniejszej pracy jest aplikacja internetowa dostosowana do urządzeń mobilnych do zarządzania procesem zamawiania posiłków w pizzeri, która świadczy również szereg innych funkcjonalności z zakresu interakcji społecznościowych oraz personalizacji usług dzięki rejestracji konta.

1.1. Motywacja

Współczesny rynek gastronomiczny stoi przed wyzwaniami wynikającymi z konieczności usprawnienia procesów obsługi klientów oraz zarządzania zamówieniami. Dynamiczny rozwój technologii informatycznych umożliwia tworzenie rozwiązań, które nie tylko automatyzują codzienne operacje, ale także poprawiają jakość świadczonych usług. Niniejsza praca inżynierska skupia się na wykonaniu aplikacji dedykowanej do zarządzania zamówieniami w pizzerii, która integruje kluczowe funkcjonalności wspierające operacje biznesowe w tej branży. Projekt uwzględnia nowoczesne technologie oraz podejścia projektowe, aby zapewnić wysoką wydajność, elastyczność i możliwość dalszego rozwoju systemu.

1.2. Cel pracy

Celem pracy było opracowanie aplikacji, która w pełni wykorzystuje obecne możliwości technologiczne do zarządzania zamówieniami w placówce gastronomicznej i która spełnia oczekiwania dzisiejszych standardów. Celem było nie tylko sprostać potrzebom rynku, ale także stworzyć solidną podstawę do dalszego rozszerzania systemu o dodatkowe moduły, takie jak aplikacja personelu oraz panel administracyjny.

1.3. Zakres pracy

Zakres niniejszej pracy obejmuje projekt, implementację oraz testowanie aplikacji służącej do zarządzania procesem zamawiania posiłków w pizzerii. Kluczowe elementy realizacji obejmują:

- **Analizę wymagań funkcjonalnych i niefunkcjonalnych**: Identyfikacja kluczowych potrzeb użytkowników oraz opracowanie specyfikacji technicznej systemu.
- **Zaprojektowanie systemu:** Opracowanie diagramów określających pożądane działanie systemu, opracowanie przypadków użycia oraz zaprojektowanie interfejsów graficznych.

- **Wybór technologii**: Rozpoznanie wiodących technologii na rynku, analiza pod kątem danego przypadku oraz dobór technologii, który sprawdzi się najlepiej.
- **Implementację systemu**: Opracowanie aplikacji serwerowej, warstwy komunikacji z bazą danych oraz warstwy komunikacji z aplikacjami użytkowników. W następnej kolejności implementacja aplikacji mobilnej wraz z integracją całego systemu.
- **Testowanie aplikacji**: Dodanie kompleksowych testów jednostkowych i integracyjnych gwarantujących prawidłowe działanie systemu.
- **Opracowanie potencjalnych kierunków rozwoju**: Zdefiniowanie możliwości rozszerzenia systemu o nowe funkcjonalności, takie jak panel administracyjny, aplikacje dla personelu czy integrację z zewnętrznymi platformami gastronomicznymi.

2. Analiza biznesowa

Analiza biznesowa jest kluczowym etapem projektowania aplikacji, którego celem jest dokładne zrozumienie wymagań użytkowników oraz potrzeb biznesowych, jakie aplikacja ma zaspokajać. W przypadku aplikacji mobilnej do zamawiania jedzenia w pizzerii, analiza biznesowa pozwala zidentyfikować nie tylko funkcjonalności, jakie powinna oferować aplikacja, ale także określić, jak ma wyglądać interakcja użytkownika z systemem oraz jakie technologie będą najbardziej odpowiednie do wdrożenia projektu. W ramach tego rozdziału zostaną omówione takie zagadnienia jak wymagania funkcjonalne i niefunkcjonalne.

2.1. Wymagania funkcjonalne

2.1.1. Moduł logowania i rejestracji

- **Logowanie**: Użytkownik może się zalogować, podając adres e-mail i hasło.
- **Rejestracja**: Użytkownik może się zarejestrować, tworząc konto za pomocą adresu e-mail i hasła, oraz podając podstawowe dane kontaktowe (opcjonalnie).
- **Opcjonalne korzyści logowania**:
    - Autouzupełnianie formularza adresowego i danych kontaktowych.
    - Możliwość śledzenia statusu zamówienia.
    - Możliwość komentowania i recenzowania pozycji menu.
- **Reset hasła**: Użytkownik może zresetować swoje hasło przy pomocy wiadomości e-mail.

2.1.2. Moduł przeglądania menu

- **Wyświetlanie listy pozycji**: Użytkownik może przeglądać dostępne pozycje menu (pizze i inne potrawy) wraz z cenami i podstawowymi informacjami (bez konieczności logowania).
- **Wyświetlanie szczegółów pozycji**:
    - Składniki danej pizzy.
    - Opinie użytkowników (bez konieczności logowania).

- **Komentowanie i recenzowanie**: Zalogowani użytkownicy mogą dodawać własne opinie, aktualizować je oraz je usuwać.
- **Dodawanie do koszyka:** Użykownik może wybrać wariant danej pozycji np. rozmiar i dodać do koszyka oraz inkrementować/dekrementować ilość pozycji.

2.1.3. Moduł koszyka

- **Dodawanie do koszyka**: Użytkownik może dodawać pozycje do koszyka (z poziomu menu).
- **Edycja zawartości koszyka**: Możliwość zmiany ilości, usuwania pozycji z koszyka lub całkowitego wyczyszczenia koszyka.
- **Podsumowanie koszyka**: Wyświetlanie całkowitej kwoty zamówienia, wraz z opcjonalnymi kosztami dostawy.

2.1.4. Formularz zamówienia

- **Formularz adresowo-kontaktowy:** Pola na adres dostawy i dane kontaktowe (dane autouzupełniane po zalogowaniu).
- **Wybór metody dostawy:** Opcje dostawy, np. dostawa na adres lub odbiór osobisty.
- **Potwierdzenie zamówienia:** Użytkownik ma możliwość przeglądnięcia podsumowania zamówienia przed jego finalnym potwierdzeniem.

2.1.5. Moduł śledzenia zamówienia

- **Historia zamówień**: Zalogowani użytkownicy mają dostęp do listy swoich wcześniejszych zamówień.
- **Aktualne zamówienia**: Wyświetlanie statusu obecnych zamówień (np. „oczekujące”, „w przygotowaniu”, „w drodze”, „dostarczone”).

- **Anulowanie zamówienia**: Możliwość zgłoszenia prośby o anulowanie zamówienia, jeśli jest w stanie „oczekującym”.

2.1.6. Panel użytkownika

- **Zakładka profilowa**:
  - Edytowanie danych profilowych, w tym zdjęcie profilowe, adres, numer kontaktowy.
  - Usuwanie konta.
- **Zakładka recenzji**:
  - Dostęp do własnych recenzji.
  - Możliwość edycji lub usuwania recenzji.
- **Zakładka zamówień**:
  - Historia zamówień.
  - Status obecnych zamówień.
  - Możliwość anulowania zamówienia w stanie “pending”

2.2. Wymagania niefunkcjonalne

2.2.1. Wydajność

- **Czas uruchamiania aplikacji**: Aplikacja mobilna powinna się uruchamiać w czasie nie dłuższym niż 1,5 sekundy przy zimnym starcie, ponieważ badania dotyczące UX wskazują, że 53% użytkowników opuszcza aplikację lub stronę, jeśli ładowanie trwa dłużej niż 3 sekundy, co czyni 1,5 sekundy bezpieczną wartością docelową [1].
- **Responsywność interfejsu**: Wszystkie akcje użytkownika powinny być przetwarzane w czasie poniżej 200 ms, ponieważ latencje wyższe niż ta wartość mogą być interpretowane jako wolne działanie systemu, co zmniejsza zadowolenie użytkownika [1].
- **Optymalizacja ruchu sieciowego**: Aplikacja powinna minimalizować ilość przesyłanych danych, stosując optymalizację zapytań HTTP i mechanizmy

cache'owania, aby zmniejszyć obciążenie backendu oraz zużycie danych mobilnych.

2.2.2. Użyteczność

- **Dostosowanie do urządzeń mobilnych**: Interfejs aplikacji powinien być zaprojektowany w oparciu o zasady projektowania dla urządzeń mobilnych (m.in. wyraźne przyciski, ograniczona liczba kroków do wykonania akcji).
- **Łatwość obsługi**: Aplikacja powinna być zoptymalizowana do komfortowej obsługi w taki sposób by najważniejsze przyciski do interakcji użytkownika z aplikacją były w łatwo dostępnym miejscu.
- **Responsywność**: Aplikacja powinna płynnie działać na szerokiej gamie urządzeń z systemami Android i iOS, zachowując spójność interfejsu na różnych rozdzielczościach ekranów.

2.2.3. Bezpieczeństwo

- **Bezpieczne przechowywanie danych w aplikacji mobilnej**: Dane użytkownika (np. tokeny logowania) powinny być przechowywane lokalnie w bezpieczny sposób, np. Secure Storage na Androidzie i iOS.
- **Szyfrowanie komunikacji**: Cała komunikacja między aplikacją mobilną a backendem powinna być zabezpieczona za pomocą HTTPS oraz dodatkowego szyfrowania danych wrażliwych.
- **Autoryzacja i uwierzytelnianie**: Backend powinien obsługiwać autoryzację z wykorzystaniem JWT zapewniając bezpieczne zarządzanie sesją użytkownika [12].

2.2.4. Niezawodność

- **Odporność na brak połączenia**: Aplikacja mobilna powinna zapewniać stabilność działania w przypadku utraty połączenia internetowego. Dane powinny być zapisywane lokalnie i synchronizowane po odnowieniu.

2.2.5. Obsługa błędów

- **Przyjazne komunikaty błędów dla użytkowników**: Aplikacja mobilna powinna informować użytkownika o wystąpieniu błędów w przystępny sposób, sugerując potencjalne działania, np. „Sprawdź połączenie internetowe”.
- **Logowanie błędów na backendzie**: Backend powinien rejestrować błędy i niepowodzenia zapytań w centralnym systemie logowania, aby zespół mógł je analizować i poprawiać.
- **Zarządzanie błędami sieciowymi**: Aplikacja powinna obsługiwać błędy sieciowe (np. timeout) z wyraźnym komunikatem dla użytkownika oraz powinna umożliwiać ponowne próby wykonania danej operacji.

2.2.6. Rozszerzalność

- **Łatwość dodawania funkcji**: Architektura aplikacji mobilnej powinna umożliwiać łatwe dodawanie nowych funkcji (np. możliwość zamówienia dodatkowych składników) bez wpływu na podstawowe działanie aplikacji.
- **Modularność backendu**: Backend powinien być podzielony na moduły (np. moduł zarządzania zamówieniami, moduł użytkowników), aby umożliwić łatwe wprowadzanie zmian i modyfikacje w poszczególnych częściach systemu bez wpływu na całość.

2.2.7. Zgodność technologiczna

- **Kompatybilność z systemami Android i iOS**: Aplikacja powinna działać na popularnych wersjach systemów Android i iOS (np. Android 8.0+ oraz iOS 12+).
- **Zgodność z API backendu**: Aplikacja powinna być zgodna z protokołami i strukturą danych zwracanych przez aplikację serwerową (REST API lub GraphQL).
- **Zgodność z bibliotekami mobilnymi**: Aplikacja powinna wykorzystywać zalecane biblioteki do zarządzania siecią, np. natywny “fetch js”.

2.2.8. Optymalizacja zasobów

- **Optymalizacja zużycia baterii**: Aplikacja powinna minimalizować zużycie baterii, zwłaszcza przy dłuższym użytkowaniu, np. optymalizując odświeżanie statusów zamówienia w tle.
- **Minimalizacja użycia pamięci**: Aplikacja powinna być zoptymalizowana pod kątem wykorzystania pamięci urządzenia, szczególnie na starszych urządzeniach mobilnych, poprzez efektywne zarządzanie pamięcią cache oraz usuwanie niepotrzebnych zasobów.
- **Obsługa aktualizacji aplikacji i backendu**: Aplikacja mobilna powinna obsługiwać aktualizacje backendu bez konieczności każdorazowej aktualizacji aplikacji, zapewniając stabilność i spójność wersji.

3. Analiza technologiczna

W ramach analizy technologicznej określone zostaną kluczowe komponenty oraz technologie, które umożliwią efektywną realizację założeń projektu. Obejmuje to wybór środowisk programistycznych, języków, narzędzi do budowy interfejsu użytkownika oraz infrastruktury serwerowej, a także bazę danych do przechowywania danych.

3.1. Środowisko mobilne

Aplikacje mobilne zyskały ogromną popularność dzięki szerokiemu dostępowi do smartfonów i tabletów, które oferują wysoką wydajność, intuicyjne interfejsy oraz stały dostęp do Internetu. Rozwój technologii mobilnych umożliwia tworzenie zaawansowanych aplikacji o szerokiej funkcjonalności, od komunikatorów, poprzez aplikacje do bankowości, aż po gry i narzędzia pracy zdalnej.

3.1.1. IOS

iOS to mobilny system operacyjny stworzony przez firmę Apple, dedykowany urządzeniom takim jak iPhone, iPad oraz iPod Touch. iOS wyróżnia się wysoką jakością wykonania aplikacji oraz stabilnością, co jest efektem ścisłej kontroli nad ekosystemem i wymaganiami sprzętowymi. Apple dostarcza narzędzia, które ułatwiają tworzenie aplikacji zgodnych z wytycznymi estetycznymi i funkcjonalnymi systemu.

Cechy charakterystyczne iOS

- **Ekosystem zamknięty**: Apple kontroluje zarówno sprzęt, jak i oprogramowanie, co pozwala na osiąganie wysokiej optymalizacji aplikacji pod kątem wydajności i bezpieczeństwa.
- **Wysokie standardy jakości**: Aplikacje przechodzą rygorystyczny proces akceptacji, zanim zostaną opublikowane w App Store.
- **Jednolity interfejs**: Apple zaleca stosowanie jednolitych standardów i bibliotek, takich jak UIKit czy SwiftUI, aby aplikacje zachowywały spójny wygląd i sposób działania.

Technologie i języki programowania iOS:

- **Objective-C**: Pierwotny język programowania dla iOS, używany w starszych aplikacjach. Charakteryzuje się złożoną składnią i jest obecnie wypierany przez nowszy język – Swift.
- **Swift**: Nowoczesny język programowania, wprowadzony przez Apple w 2014 roku. Swift jest szybki, łatwiejszy do nauki oraz bezpieczniejszy w użyciu w porównaniu z Objective-C. Jest aktualnie standardem dla nowych aplikacji iOS.

3.1.2. Android

Android to mobilny system operacyjny oparty na jądrze Linux, rozwijany przez Google. Jest to system otwarty, co pozwala na szeroką personalizację oraz obsługę dużej liczby urządzeń produkowanych przez różnych producentów, takich jak Samsung, Xiaomi czy Huawei. Dzięki temu Android jest obecnie najpopularniejszym systemem mobilnym na świecie.

Cechy charakterystyczne Androida [7]:

- **Otwartość**: System jest otwarty na modyfikacje przez producentów sprzętu, co zwiększa jego elastyczność, ale może prowadzić do problemów z fragmentacją (wieloma różnymi wersjami systemu na rynku).
- **Duża liczba urządzeń**: Android działa na szerokiej gamie urządzeń, co stwarza wyzwania związane z optymalizacją aplikacji dla różnych rozdzielczości i specyfikacji sprzętowych.
- **Sklep Google Play**: Android zapewnia łatwy dostęp do dystrybucji aplikacji przez Google Play, chociaż wymogi akceptacji aplikacji są mniej rygorystyczne niż w App Store

Technologie i języki programowania dla Androida [7]:

- **Java**: Pierwotny język programowania dla Androida, który nadal jest szeroko stosowany. Java zapewnia wysoką stabilność i kompatybilność wsteczną.
- **Kotlin**: Oficjalnie wspierany przez Google od 2017 roku, Kotlin to nowoczesny język programowania, który poprawia czytelność kodu, ułatwia tworzenie bezpieczniejszych aplikacji i jest bardziej ekspresyjny niż Java.

3.2. JavaScript i ReactNative

Z powodu wyzwań związanych z tworzeniem osobnych aplikacji na iOS i Androida rozsądną opcją jest wykorzystanie frameworka “ReactNative”, który umożliwia stworzenie jednej aplikacji, która działa na różnych systemach operacyjnych z wykorzystaniem jednego kodu bazowego. Podejście to pozwala na znaczną redukcję kosztów i czasu pracy potrzebnych do stworzenia aplikacji

React Native to framework open-source, opracowany przez Facebook, pozwalający na tworzenie aplikacji mobilnych z użyciem języka JavaScript. React Native wykorzystuje komponenty natywne systemu, co pozwala na tworzenie aplikacji o wysokiej wydajności i responsywności, zbliżonych do aplikacji natywnych [1].

- **Zalety**: Szybki czas wdrażania, szeroka społeczność, możliwość wykorzystania natywnych elementów UI dla iOS i Androida, co poprawia wydajność.
- **Wady**: Potrzebna jest znajomość specyfiki natywnych systemów operacyjnych w przypadku bardziej złożonych aplikacji.

JavaScript, który jest wykorzystywanym językiem w tym frameworku jest jednym z najpopularniejszych języków programowania na świecie. Początkowo został zaprojektowany jako język skryptowy dla stron internetowych, aby zapewnić interaktywność w przeglądarkach internetowych. Obecnie jednak jest wykorzystywany zarówno w środowiskach frontendowych (biblioteka React, framework Angular), jak i serwerowych (np. Node.js). Jego uniwersalność pozwala na tworzenie aplikacji internetowych, mobilnych, a także aplikacji desktopowych i systemów rozproszonych.

JavaScript jest językiem [3]:

- **Wieloplatformowym**: Działa na praktycznie każdej platformie – od przeglądarek po serwery i urządzenia mobilne.
- **Dynamicznie typowanym**: Typy zmiennych są przypisywane dynamicznie podczas działania programu, co ułatwia pracę, ale wymaga ostrożności, aby unikać błędów związanych z typami danych.

- **Interpretowanym**: Kod JavaScript jest interpretowany przez przeglądarki lub silniki JavaScript, takie jak V8 (stosowany w Google Chrome i Node.js), co przyspiesza rozwój, ale wymaga optymalizacji wydajności.
- **Zorientowanym obiektowo i funkcyjnie**: JavaScript wspiera programowanie obiektowe (z prototypowym dziedziczeniem) oraz funkcyjne, co pozwala na różnorodne podejście do rozwiązywania problemów.

3.3. TypeScript

TypeScript to język programowania opracowany przez Microsoft, który jest nadzbiorem JavaScript i dodaje do niego statyczne typowanie. Umożliwia programistom korzystanie z typów takich jak liczby, stringi, obiekty oraz bardziej złożone struktury danych, co przyczynia się do większej czytelności i przewidywalności kodu. TypeScript kompiluje się do czystego JavaScriptu, dzięki czemu można go używać wszędzie tam, gdzie działa JavaScript, a szczególnie dobrze sprawdza się w projektach o większej skali i w rozbudowanych zespołach programistycznych.

Główne cechy TypeScript:

- **Statyczne typowanie**: TypeScript wymaga określenia typów zmiennych i funkcji, co umożliwia wykrywanie błędów na etapie kompilacji. To szczególnie istotne w większych projektach, gdzie statyczne typowanie zapobiega wielu błędom, które mogłyby się pojawić dopiero podczas działania programu.
- **Interfejsy i klasy**: TypeScript rozszerza możliwości obiektowości JavaScriptu przez wprowadzenie interfejsów i ulepszonych klas. Interfejsy pomagają w definiowaniu struktur danych i kontraktów dla obiektów, co sprzyja utrzymaniu porządku i zgodności w kodzie.
- **Obsługa zaawansowanych typów**: TypeScript wspiera typy złożone, takie jak typy unijne i typy generyczne, które ułatwiają tworzenie elastycznych i wielokrotnego użytku komponentów.

- **Lepsza integracja z narzędziami programistycznymi**: TypeScript poprawia możliwości narzędzi programistycznych (takich jak VS Code) przez automatyczne podpowiadanie kodu, sprawdzanie błędów i refaktoryzację. Dzięki zdefiniowanym typom, środowisko może wyświetlać bardziej trafne podpowiedzi oraz błyskawicznie wskazywać niezgodności.
- **Transpilacja do JavaScriptu**: TypeScript transpiluje się do czystego JavaScriptu, co oznacza, że można go używać na każdej platformie obsługującej JavaScript. Transpilacja pozwala również na zgodność z różnymi wersjami JavaScriptu (od ES5 do ESNext), dzięki czemu jest bardzo elastyczny.

3.4. Technologie serwerowe

3.4.1. Java

Java to jeden z najbardziej popularnych, obiektowo zorientowanych języków programowania, stworzony przez firmę Sun Microsystems (obecnie będącą częścią Oracle) w połowie lat 90. Celem twórców Javy było stworzenie języka, który byłby niezależny od platformy i umożliwiał tworzenie przenośnych, stabilnych i wydajnych aplikacji. Dzięki temu Java zyskała ogromną popularność zarówno w aplikacjach korporacyjnych, jak i w oprogramowaniu na urządzenia mobilne.

Kluczowe cechy Javy [2]:

1. **Obiektowość:** Java jest w pełni obiektowym językiem, co oznacza, że kod jest organizowany wokół klas i obiektów. Programowanie obiektowe ułatwia zarządzanie kodem oraz pozwala na lepsze modelowanie rzeczywistości poprzez mechanizmy, takie jak dziedziczenie, enkapsulacja i polimorfizm.
2. **Przenośność:** Aplikacje napisane w Javie są kompilowane do kodu bajtowego (bytecode), który może być uruchamiany na różnych platformach za pomocą Java Virtual Machine (JVM). Dzięki temu aplikacje Java są niezależne od systemu operacyjnego i sprzętu, co zapewnia ich uniwersalność.
3. **Bezpieczeństwo i zarządzanie pamięcią:** Java posiada mechanizmy zarządzania pamięcią, takie jak Garbage Collector (automatyczny system zwalniania nieużywanych zasobów), co minimalizuje ryzyko błędów związanych z ręcznym zarządzaniem

pamięcią, poprawia stabilność aplikacji oraz eliminuje potrzebę manualnego zarządzania obszarem pamięci.

4. **Wielowątkowość:** Java wspiera programowanie wielowątkowe, co pozwala na jednoczesne przetwarzanie wielu zadań na wielu rdzeniach procesora. Jest to szczególnie przydatne w aplikacjach serwerowych i systemach o wysokiej wydajności, gdzie ważna jest równoczesna obsługa wielu zapytań. Środowisko Java posiada wiele wbudowanych narzędzi do łatwego i uporządkowanego implementowania programów wielowątkowych.
5. **Rozbudowany ekosystem i wsparcie społeczności**: Java posiada rozbudowany ekosystem bibliotek, narzędzi i frameworków (np. Hibernate, Spring), co umożliwia szybki rozwój aplikacji, ich testowanie oraz optymalizację. Dodatkowo ogromna społeczność użytkowników oznacza wsparcie w postaci dokumentacji, forów i otwartego dostępu do rozwiązań problemów oraz mnóstwo bibliotek open-source.

Java jest stosowana w wielu dziedzinach, m.in.:

- **Aplikacje korporacyjne:** Dzięki stabilności i skalowalności, Java jest podstawą wielu systemów klasy Enterprise.
- **Aplikacje mobilne na Androida:** Java do 2019 roku była oficjalnym językiem używanym w tworzeniu aplikacji na Androida. Chociaż ustąpiła miejsca językowi Kotlin, to Kotlin wciąż kompiluje się do kodu bajtowego JVM.
- **Systemy rozproszone:** Java, wspierana przez platformę JVM, świetnie sprawdza się w systemach rozproszonych i przetwarzaniu dużych ilości danych.

3.4.2. Spring Boot

Spring Boot to framework oparty na Spring Framework, który jest jednym z najpopularniejszych frameworków Java, przeznaczonym głównie do budowy aplikacji webowych i serwisów backendowych. Spring Boot został stworzony, aby ułatwić proces tworzenia aplikacji – eliminuje potrzebę skomplikowanej konfiguracji oraz dostarcza gotowe narzędzia, które pozwalają szybko i efektywnie uruchomić aplikację.

Kluczowe cechy Spring Boot [11]:

1. **Minimalna konfiguracja dzięki podejściu „konwencja nad konfiguracją”**: Spring Boot jest skonfigurowany domyślnie, co oznacza, że wiele elementów działa automatycznie, a programista może skoncentrować się na logice biznesowej. Większość konfiguracji jest zautomatyzowana, co oszczędza czas i upraszcza projekt.
2. **Wbudowany serwer aplikacyjny:** Spring Boot oferuje wbudowany serwer, np. Tomcat lub Jetty, co umożliwia uruchamianie aplikacji bezpośrednio bez konieczności zewnętrznej konfiguracji serwera. Pozwala to na prostsze testowanie aplikacji oraz jej łatwe wdrażanie na serwery produkcyjne.
3. **Spring Boot Starters:** Spring Boot posiada tzw. Starter POMs, czyli zestawy predefiniowanych zależności i konfiguracji dla różnych funkcji (np. integracja z bazami danych, obsługa bezpieczeństwa), co pozwala szybko dodać odpowiednie funkcjonalności do projektu.
4. **Obsługa REST API:** Dzięki wbudowanej obsłudze REST, Spring Boot umożliwia szybkie tworzenie serwisów webowych i aplikacji o architekturze RESTful. Aplikacje takie są często stosowane jako backendy dla aplikacji mobilnych i webowych, a także mikrousługi.
5. **Integracja z bazami danych i ORM**: Spring Boot pozwala na bezpośrednią integrację z popularnymi bazami danych (np. MySQL, PostgreSQL), a także z frameworkami ORM, takimi jak Hibernate, co umożliwia mapowanie obiektów na tabele w bazie danych i automatyczne zarządzanie danymi.
6. **Obsługa mikrousług:** Dzięki dodatkowym narzędziom, takim jak Spring Cloud, Spring Boot jest szeroko stosowany do budowy mikrousług. Mikrousługi umożliwiają dzielenie aplikacji na mniejsze, niezależnie działające komponenty, które można łatwo skalować i wdrażać.

Spring Boot to wybór często stosowany w przypadku [9]:

- Aplikacji webowych i backendowych: Dzięki REST API i integracji z bazami danych, Spring Boot jest idealny do tworzenia złożonych backendów.
- Systemów korporacyjnych i bankowych: Ze względu na stabilność i wsparcie dla transakcji, Spring Boot jest popularny w sektorach, które wymagają wysokiej niezawodności.

- Mikrousług: Spring Boot, w połączeniu ze Spring Cloud, wspiera projektowanie mikrousług – architektury umożliwiającej budowę systemów rozproszonych.

3.5. Baza danych PostgreSQL

Bazy danych to struktury przechowujące zbiory danych, które mogą być uporządkowane, przeszukiwane i zarządzane w sposób efektywny. Są one kluczowym elementem współczesnych aplikacji, ponieważ umożliwiają organizowanie, przechowywanie oraz szybkie przetwarzanie dużych ilości danych. Dzięki bazom danych możliwe jest m.in. przechowywanie informacji o użytkownikach aplikacji, historii zakupów, dokumentach czy katalogach produktów.

Bazy danych wykorzystują **język SQL (Structured Query Language)** do zarządzania danymi i komunikacji z aplikacjami. Poprzez zapytania SQL programista może odczytywać, modyfikować oraz usuwać dane zgodnie z potrzebami aplikacji. Dzięki tym operacjom systemy bazodanowe są w stanie obsługiwać różne żądania aplikacji, od wyszukiwania rekordów po agregację danych na dużą skalę.

PostgreSQL to zaawansowany system zarządzania relacyjną bazą danych (RDBMS – Relational Database Management System), znany z solidności, skalowalności oraz wsparcia dla złożonych typów danych. Jest dostępny jako oprogramowanie open-source, co oznacza, że można z niego korzystać bez opłat licencyjnych, a także dostosowywać go do specyficznych potrzeb projektu.

System ten wyróżnia się dużą elastycznością oraz zgodnością ze standardami SQL, ale oferuje również rozszerzenia wykraczające poza relacyjny model danych, co sprawia, że jest stosowany w wielu różnych dziedzinach, od systemów korporacyjnych po zaawansowane aplikacje analityczne.

Kluczowe cechy PostgreSQL [5]:

1. **Zgodność ze standardem SQL:** PostgreSQL obsługuje standard SQL, dzięki czemu programiści mogą pisać zapytania i korzystać z wielu zaawansowanych operacji na danych, takich jak transakcje, widoki, indeksy oraz funkcje agregujące.

2. **Obsługa złożonych typów danych i rozszerzeń:** PostgreSQL oferuje wsparcie dla różnorodnych typów danych, w tym danych JSON, XML, a także struktur geograficznych (PostGIS), co czyni go doskonałym wyborem dla aplikacji pracujących z niestandardowymi danymi. Można go również rozszerzać za pomocą dodatkowych modułów i narzędzi, które zwiększają jego możliwości.
3. **Transakcyjność i spójność danych:** PostgreSQL zapewnia pełną obsługę transakcji (ACID compliance), co oznacza, że operacje na danych są atomowe, spójne, izolowane i trwałe. Dzięki temu jest stosowany w systemach, gdzie kluczowe jest zachowanie integralności danych, takich jak bankowość czy systemy e-commerce.
4. **Wydajność i skalowalność**: PostgreSQL jest zoptymalizowany do obsługi dużych ilości danych i może działać na wielu serwerach jednocześnie, co pozwala na zwiększenie przepustowości i wydajności przy zachowaniu spójności danych. Obsługuje partycjonowanie tabel i indeksowanie, co przyspiesza działanie aplikacji, nawet przy dużym obciążeniu.
5. **Społeczność i wsparcie:** PostgreSQL posiada silne wsparcie społeczności oraz bogatą dokumentację. Jako projekt open-source, jest nieustannie rozwijany przez programistów z całego świata, co sprawia, że jego funkcje są regularnie aktualizowane i dostosowywane do zmieniających się potrzeb rynku.

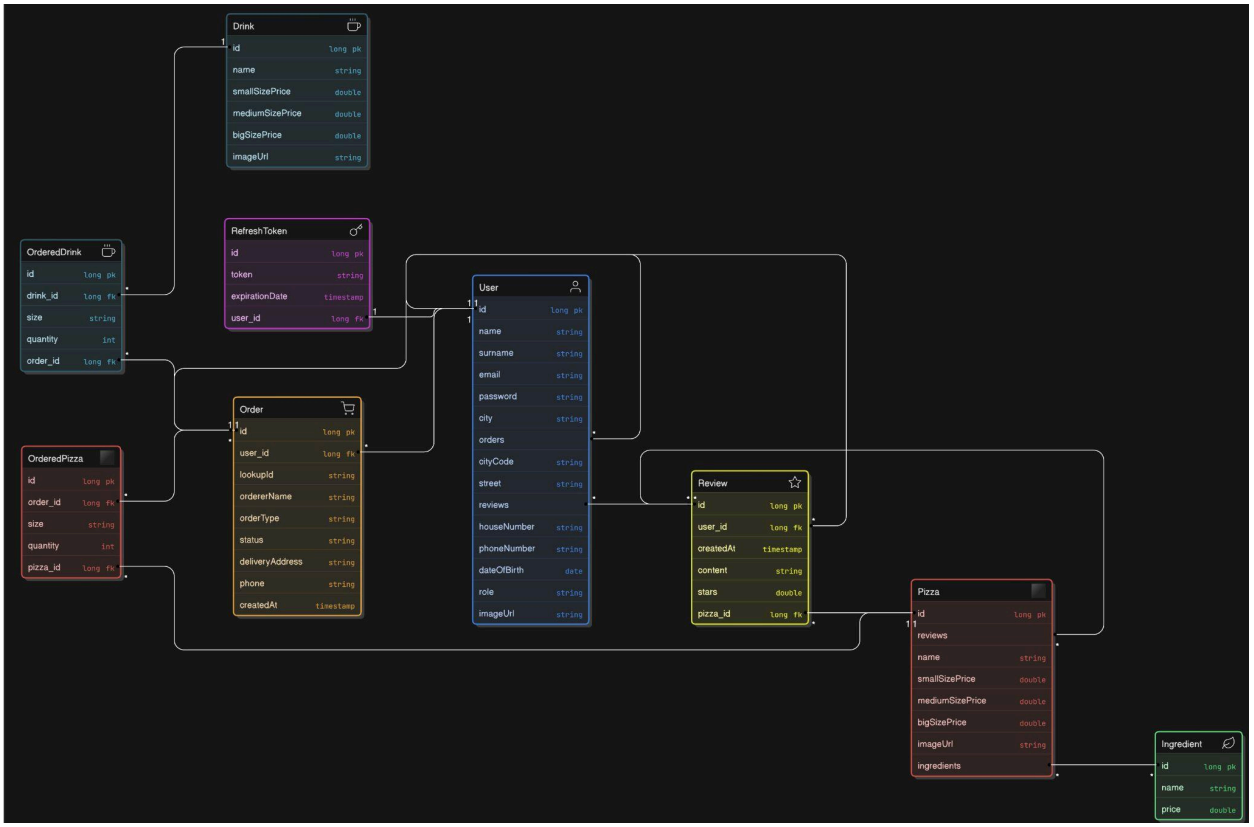
4. Implementacja systemu

4.1. Baza danych

Aby skutecznie zaimplementować system do zamawiania jedzenia, pierwszym krokiem jest zdecydowanie jakie dane są potrzebne do działania aplikacji przy istniejących założeniach, a następnie zaimplementowanie tego procesu. Mając na uwadzę docelową funkcjonalność systemu kluczowe będzie przechowywanie informacji o:

- **Użytkownikach** – w celu umożliwienia rejestracji, logowania i zarządzania kontem. Dane te obejmują m.in. podstawowe informacje personalne, dane kontaktowe oraz szczegóły dotyczące autoryzacji, jak np. role użytkowników.
- **Ofercie gastronomicznej** – np. szczegóły dotyczące pizzy i napojów dostępnych w menu. Informacje te obejmują nazwę, ceny, rozmiary oraz składniki każdego produktu, co umożliwia użytkownikowi na zapoznanie się z szczegółami danej pozycji
- **Recenzjach** – wystawianych przez użytkowników dla konkretnych pozycji z menu, co pozwala na zbieranie opinii, ich analizowanie oraz wyświetlanie innym użytkownikom.
- **Zamówieniach** – dane o zamówieniach pozwolą na ich śledzenie, przechowywanie historii, a także na zarządzanie statusem każdego zamówienia.
- **Innych ukrytych elementach systemu** – takich jak tokeny sesji do obsługi uwierzytelniania i bezpieczeństwa, które są niezbędne do zarządzania aktywnością użytkowników w systemie i zabezpiecza process logowania oraz dostępu do zasobów.

Do zaimplementowania bazy danych zostało użyte narzędzie **JPA (Java Persistence API)** dodane do zależności w Spring Boot, które umożliwia mapowanie obiektowo-relacyjne (ORM). Dzięki JPA możliwe jest automatyczne generowanie zapytań do bazy danych na podstawie deklaracji klas w języku Java co sprawia że cała logika jest tworzona w jednym miejscu oraz w jednym języku - Java [13]. JPA przekształca klasy z Javy na tabele dla określonej bazy danych. Podejście to umożliwia zarządzanie danymi w sposób oszczędzający czas i który pozwala zachować wysoką czytelność kodu. Schemat ogólny bazy danych w postaci diagramu ERD przedstawia rysunek 4.1.



Rys. 4.1. Diagram ERD bazy danych aplikacji

Tabela User: (rysunek 4.2) przedstawia kod JPA odzwierciedlający tabelę *User* z diagramu ERD (rysunek 4.1). Wewnątrz tabeli przechowywane są dane użytkowników aplikacji, takie jak imię, nazwisko, adres e-mail, hasło, miasto, kod pocztowy, adres, numer telefonu, data urodzenia, rola użytkownika oraz ścieżka do zdjęcia profilowego. Każdy użytkownik może mieć przypisaną rolę (np. użytkownik, administrator, pracownik), która określa poziom dostępu w aplikacji. Użytkownik jest powiązany z wieloma zamówieniami oraz recenzjami. Relacja „jeden do wielu" z tabelami *Order* i *Review* pozwala na przechowywanie historii zamówień oraz opinii wystawianych przez użytkownika

```java
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Table(name = "user")
public class User implements UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String surname;
    @Column(nullable = false, unique = true)
    private String email;
    private String password;
    private String city;
    private String cityCode;
    private String street;
    private String houseNumber;
    private String phoneNumber;
    private LocalDate dateOfBirth;
    private Role role;
    private String imageUrl;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
    private List<Order> orders;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
    private List<Review> reviews;

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of(new SimpleGrantedAuthority(role != null ? role.name() : null));
    }

    @Override
    public String getUsername() {
        return email;
    }

    public String toString() {
        return "%s %s - id:%s, email: %s".formatted(name, surname, id, email);
    }
}
```

Rys. 4.2. Tabela "*User*" bazy danych przedstawiona za pomocą JPA [opracowanie własne]

Tabela Pizza (Drink analogicznie): Tabela wygenerowana przez kod (Rysunek 4.3) reprezentuje oferowaną pizzę w menu. Dla każdej pozycji przechowywane są informacje, takie jak nazwa, ceny dla różnych rozmiarów oraz lista składników.

Pizza jest połączona z wieloma składnikami (*Ingredient*) w relacji „wiele do wielu”, ponieważ każda pizza może mieć kilka składników, a jednocześnie ten sam składnik może być użyty w wielu pizzach. Dodatkowo, pizza ma relację „jeden do wielu” z tabelą *Review*, co umożliwia powiązanie wielu recenzji z konkretną pizzą.

```
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Pizza {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String name;
   private double smallSizePrice;
   private double mediumSizePrice;
   private double bigSizePrice;

   @ManyToMany
   private List<Ingredient> ingredients;

   @OneToMany(mappedBy = "pizza", cascade = CascadeType.ALL, orphanRemoval = true)
   private List<Review> reviews;
   private String imageUrl;

   public double getPrice(PizzaSizes size) {
       return switch (size) {
           case SMALL -> smallSizePrice;
           case MEDIUM -> mediumSizePrice;
           case BIG -> bigSizePrice;
           default -> 0.0;
       };
   }
}
```

Rys. 4.3. Tabela “*Pizza*” bazy danych przedstawiona za pomocą JPA [opracowanie własne]

Tabela Order (Rysunek 4.4): przechowuje dane dotyczące zamówień składanych przez użytkowników. Każde zamówienie zawiera informacje niezbędne do jego identyfikacji, realizacji oraz śledzenia statusu. Tabela ta pełni kluczową funkcję w systemie zamawiania, łącząc dane użytkowników z ich zamówieniami, a także przechowując szczegóły, takie jak adres dostawy i status zamówienia. Dzięki temu możliwe jest śledzenie, zarządzanie i aktualizowanie zamówień na różnych etapach realizacji.

```
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Order {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   private String lookupId;

   @ManyToOne(fetch = FetchType.LAZY)
   @JoinColumn(name = "user_id")
   private User user;

   private String ordererName;

   @Enumerated(EnumType.STRING)
   private OrderType orderType;

   @Enumerated(EnumType.STRING)
   private OrderStatus status;

   @OneToMany(mappedBy = "order", cascade = CascadeType.ALL)
   private List<OrderedPizza> orderedPizzas;

   @OneToMany(mappedBy = "order", cascade = CascadeType.ALL)
   private List<OrderedDrink> orderedDrinks;

   private String deliveryAddress;
   private String phone;
   @CreationTimestamp
   @Column(nullable = false, updatable = false)
   private LocalDateTime createdAt;
}
```

Rys. 4.4 - Tabela "*Order*" bazy danych przedstawiona za pomocą JPA [opracowanie własne]

4.2. Aplikacja serwerowa

4.2.1. Kluczowe funkcje

Serwer w systemie komputerowym to kluczowy komponent, który przetwarza zapytania od klientów (w tym przypadku aplikacji mobilnej) i odpowiada, dostarczając wymagane dane lub wykonując określone operacje. Serwer działa jako „centrum dowodzenia", integrując różne części systemu: odbiera żądania od użytkowników, komunikuje się z bazą danych, przetwarza logikę aplikacji i zwraca odpowiedzi w zrozumiałym dla klientów formacie.

Komunikacja między aplikacją a serwerem odbywa się za pomocą protokołu **HTTP (Hypertext Transfer Protocol)**. Protokół ten definiuje sposób przesyłania żądań (requests) i odpowiedzi (responses) między klientem a serwerem, co umożliwia płynne przesyłanie danych.

4.2.2. API i kontrolery

Serwer w aplikacji działa w oparciu o API (Application Programming Interface) – zestaw predefiniowanych punktów końcowych (ang. *endpoints*), za pomocą których aplikacja mobilna komunikuje się z serwerem. W Spring Boot punkty końcowe definiowane są w tzw. **kontrolerach**, które pełnią rolę bramy wejściowej do aplikacji. Każdy kontroler odpowiada za obsługę żądań związanych z określoną funkcjonalnością aplikacji.

W zaprojektowanej aplikacji kontrolery zostały podzielone tematycznie, co zwiększa modularność i czytelność kodu. Aplikacja zawiera osiem kontrolerów, z których każdy pełni określoną rolę:

- **DrinkController**
  Odpowiada za operacje związane z napojami w menu. Obsługuje zapytania dotyczące przeglądania listy napojów, pobierania szczegółowych informacji o konkretnym napoju oraz dodawania nowych napojów do oferty (dla administratorów lub właścicieli).
- **PizzaController**
  Zajmuje się operacjami z zakresu konkretnego elementu oferty gastronomicznej - pizzy. Pozwala na uzyskiwanie informacji na temat listy pizz, ich szczegółowych składników,

cen, a także dodawanie nowych pizz do bazy danych (dla administratorów lub właścicieli).

- **IngredientController**
  Odpowiada za zarządzanie składnikami dostępnymi w ofercie. Zapewnia funkcjonalności takie jak pobieranie listy składników, dodawanie nowych składników czy modyfikowanie istniejących.
- **MenuController**
  Gromadzi dane z innych kontrolerów, dostarczając pełne menu restauracji. To centralny punkt dla użytkowników, którzy chcą przejrzeć całą ofertę gastronomiczną.
- **OrderController**
  Obsługuje proces składania zamówień. Umożliwia użytkownikom składanie nowych zamówień, przeglądanie szczegółów ich aktualnych zamówień, anulowanie zamówienia (jeśli jest w statusie oczekującym) oraz przeglądanie historii zamówień. Zawiera również szereg innych funkcjonalności dla personelu co nie jest przedmiotem tej strony aplikacji a jedynie punktem wyjścia do rozszerzenia systemu.
- **UserController**
  Zarządza operacjami związanymi z użytkownikami, takimi jak rejestracja, logowanie, aktualizacja profilu, zarządzanie danymi kontaktowymi, a także dostęp do zamówień, recencji danego użytkownika. Zawiera również API dla administratora do zarządzania.
- **ImageController**
  Służy do zarządzania obrazami produktów i użytkowników. Obsługuje funkcje przesyłania zdjęć, pobierania ich na podstawie ścieżki oraz aktualizacji zdjęć w bazie danych.
- **ReviewController**
  Odpowiada za obsługę recenzji wystawianych przez użytkowników. Umożliwia dodawanie recenzji dla określonych pozycji w menu, modyfikację wcześniej dodanych opinii oraz ich usuwanie.

4.2.3. Serwisy i logika biznesowa

Kontrolery, choć kluczowe dla komunikacji z klientem, nie zawierają logiki biznesowej. Zamiast tego, kierują żądania do odpowiednich **serwisów** – klas, które implementują funkcjonalności aplikacji. Serwisy występujące w aplikacji to:

- **DrinkService**
  Zawiera logikę związaną z napojami. Odpowiada za pobieranie napojów z bazy danych, walidację wprowadzanych danych (np. ceny, nazwy), a także za dodawanie nowych napojów do menu.
- **PizzaService**
  Zarządza logiką dotyczącą pizz, w tym pobieraniem danych o pizzach, obliczaniem ceny na podstawie rozmiaru oraz tworzeniem nowych pizz.
- **IngredientService**
  Realizuje operacje związane ze składnikami. Sprawdza, czy składnik może być powiązany z pizzą, obsługuje aktualizację danych składników oraz ich walidację.
- **MenuService**
  Agreguje dane z serwisów DrinkService i PizzaService, aby utworzyć kompleksowe menu aplikacji. Zapewnia sortowanie, filtrowanie i grupowanie danych menu na żądanie klienta.
- **OrderService**
  Obsługuje proces zamawiania, w tym walidację zamówień, kalkulację cen, zmianę statusu zamówienia (np. z oczekującego na realizowane) oraz historię zamówień użytkownika.
- **UserService**
  Zajmuje się logiką dotyczącą użytkowników, taką jak rejestracja, szyfrowanie haseł, aktualizacja danych profilowych i weryfikacja autoryzacji.
- **ImageService**
  Obsługuje przesyłanie i pobieranie obrazów, zarządzając przechowywaniem plików oraz integracją z bazą danych w celu aktualizacji ścieżek do plików.
- **ReviewService**
  Implementuje funkcjonalności związane z recenzjami, takie jak walidacja ocen,

zarządzanie dodawaniem opinii oraz zapewnienie spójności danych między użytkownikami a produktami.

4.2.4. Repozytoria i integracja z bazą danych

Każdy serwis komunikuje się z odpowiednim repozytorium opartym na JPA (Java Persistence API), które umożliwia bezpośrednią interakcję z bazą danych. Repozytoria udostępniają zestaw metod do operacji CRUD (Create, Read, Update, Delete), takich jak:

- Znajdowanie rekordów na podstawie kluczy (np. findById).
- Zapisywanie i aktualizowanie danych (save).
- Usuwanie danych (delete).

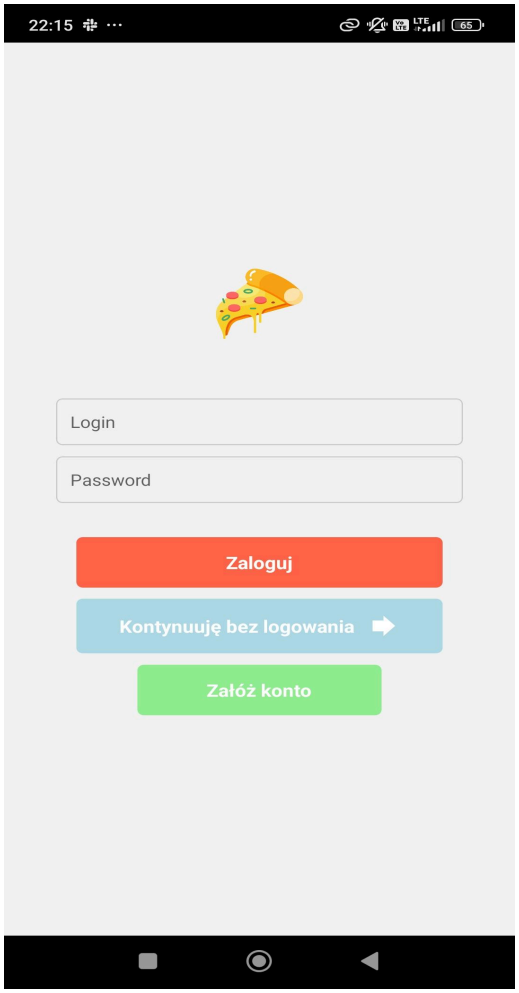
Dzięki automatycznemu mapowaniu klas encji na tabele w bazie danych, JPA upraszcza proces komunikacji z bazą danych, pozwalając skupić się na logice aplikacji zamiast na ręcznym pisaniu zapytań SQL. Wszystkie zapytania są generowane dynamicznie na podstawie struktury klas oraz deklaracji w repozytorium, co znacznie przyspiesza rozwój aplikacji i zmniejsza ryzyko błędów.

5. Interfejs użytkownika

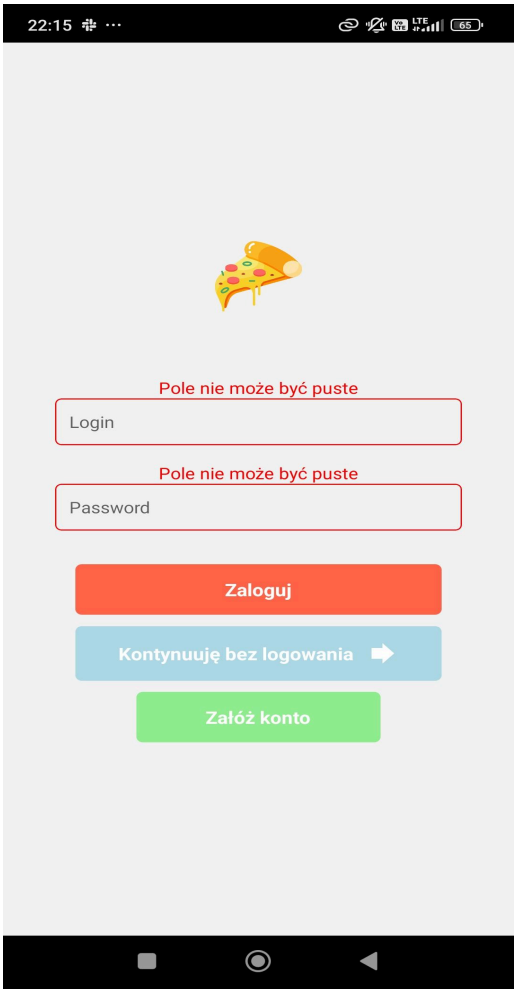
5.1. Logowanie

Ekran logowania (rys. 5.1) umożliwia użytkownikom autoryzację w systemie za pomocą adresu e-mail i hasła. Jest minimalistyczny, zawiera pola tekstowe do wprowadzenia danych oraz posiada walidację wprowadzanych danych. Zawiera przyciski:

- "Zaloguj", który rozpoczyna proces uwierzytelniania.
- “Załóż konto” , który przenosi do ekranu rejestracji.
- “Kontynuuj bez logowania”, który wprowadza użytkownika do aplikacji właściwej.



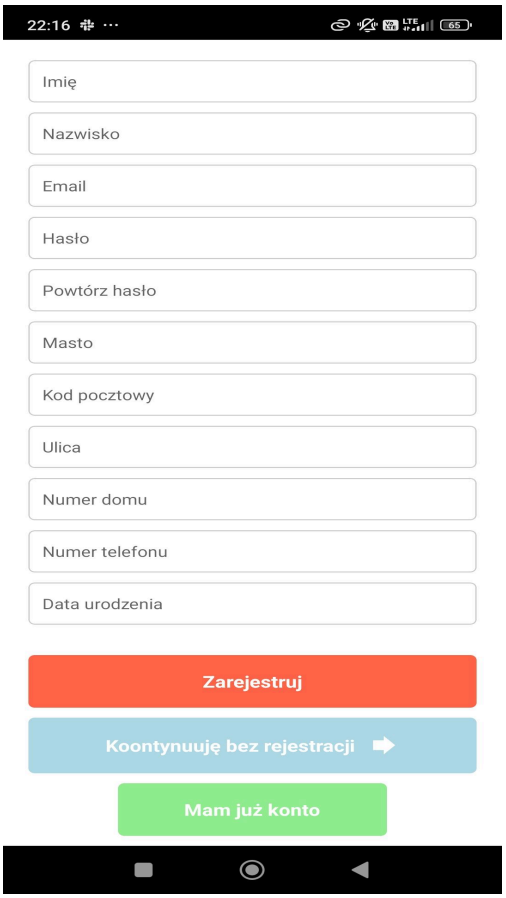
Rys. 5.1. Ekran logowania [opracowanie własne]



Rys. 5.2. Walidacja w ekranie logowania [opracowanie własne]

5.2. Rejestracja

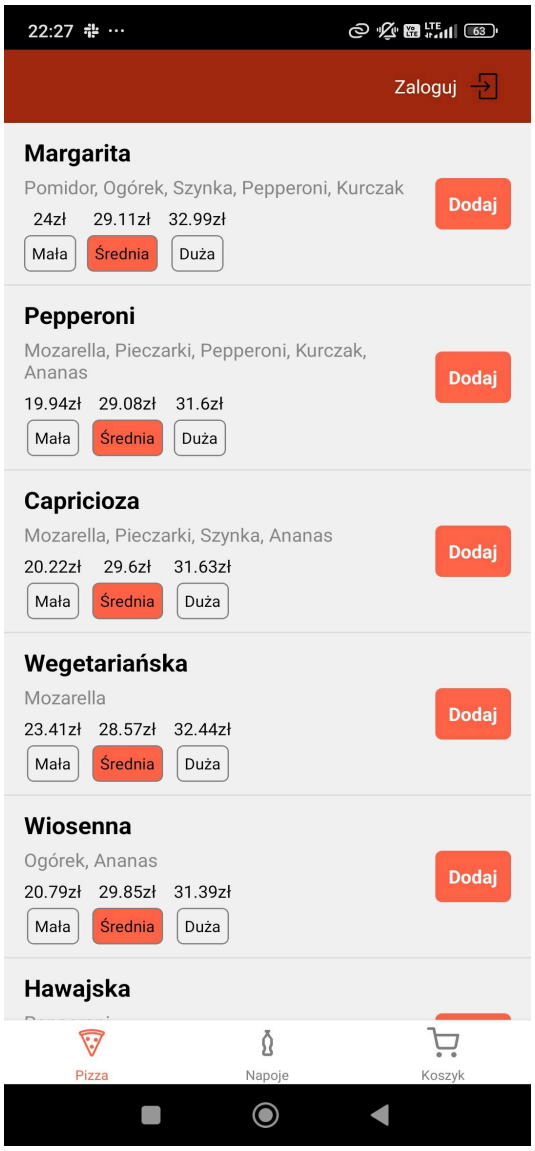
Ekran rejestracji (rys. 5.3) pozwala użytkownikom na utworzenie nowego konta. Formularz wymaga podania podstawowych danych takich jak imię, nazwisko, adres e-mail, hasło oraz opcjonalnie innych danych profilowych, które mogą być wykorzystane do zamawiania na "stały" adres takich jak pełny adres zamieszkania. Po udanej rejestracji użytkownik jest przekierowywany do ekranu logowania.



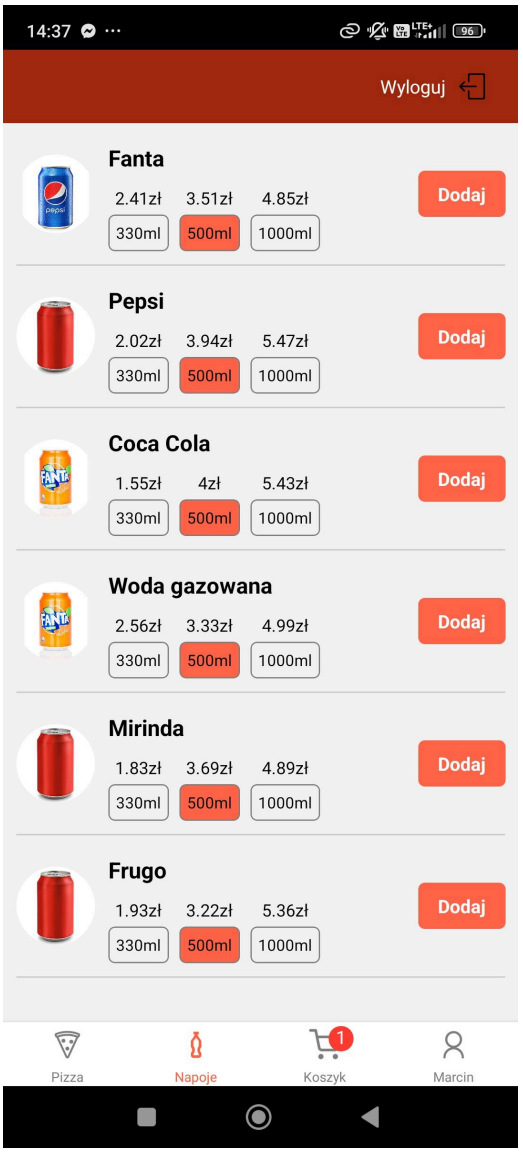
Rys. 5.3. Ekran rejestracji [opracowanie własne]

5.3. Menu

Główna część aplikacji - Menu (rys. 5.4 i 5.5), w której użytkownik może przeglądać dostępne produkty (np. pizze i napoje). Wyświetlana jest lista pozycji menu wraz z obrazkami, nazwą, ceną i możliwością rozwinięcia szczegółów, które zawierają pełne informacje o pozycji wraz z dodatkowymi opiniami innych użytkowników. Ekran umożliwia dodatkowo wybór ilości, rozmiaru z podanej oferty i dodanie go do koszyka, przyciskiem który znajduje się przy każdej opcji.



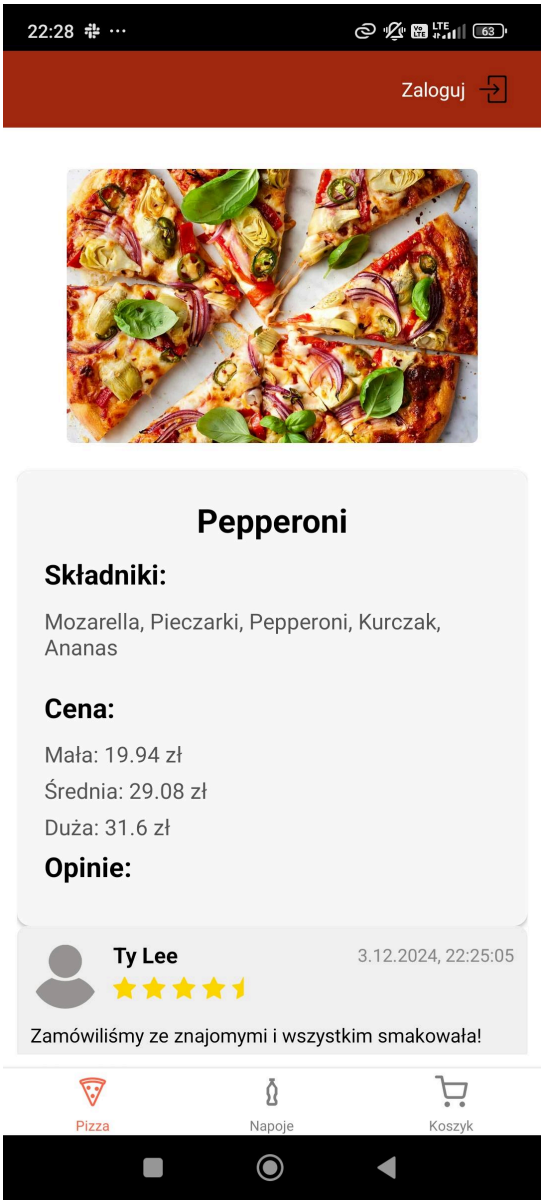
Rys. 5.4. Menu pizzy [opracowanie własne]



Rys. 5.5. Menu napojów [opracowanie własne]

5.4. Szczegóły pozycji

Ekran szczegółów (rys. 5.6) umożliwia użytkownikowi zapoznanie się z dodatkowymi informacjami na temat wybranej pozycji, np. składnikami pizzy, cenami za różne rozmiary czy opiniami innych użytkowników. Znajduje się tam także przycisk dodania produktu do koszyka oraz sekcja pozwalająca na dodanie recenzji (dla zalogowanych użytkowników).



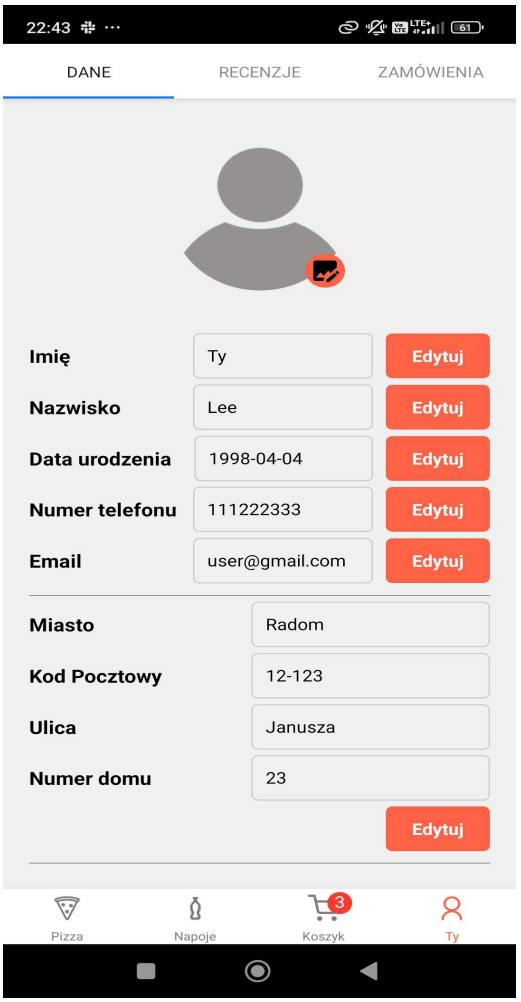
Rys. 5.6. Ekran szczegółów pozycji [opracowanie własne]

5.5. Panel Użytkownika

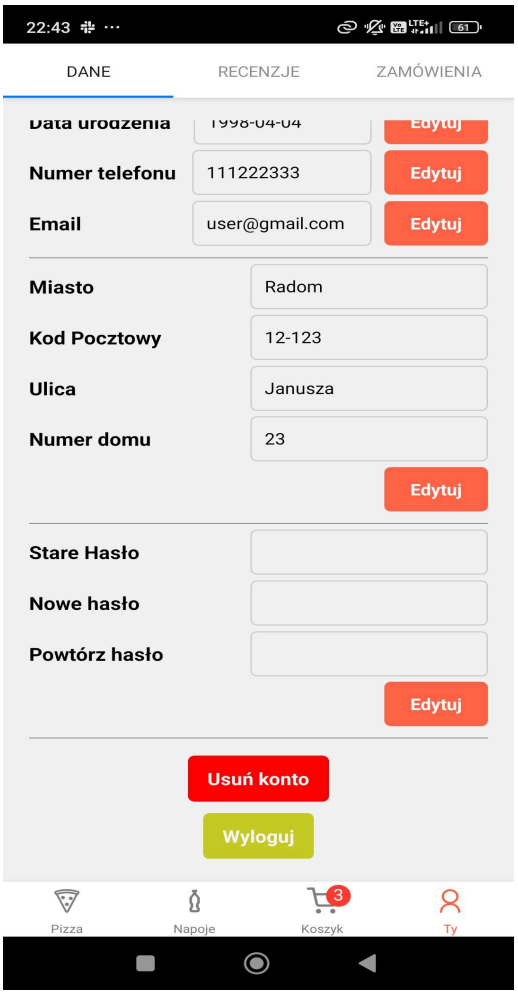
Panel użytkownika gromadzi funkcjonalności związane z zarządzaniem kontem oraz dostępem do personalnych informacji użytkownika. Składa się z trzech zakładek: **Ustawienia**, **Recenzje**, **Zamówienia**. Pierwszą domyślnie wybraną zakładką są *ustawienia*.

5.6. Ustawienia

Zakładka (rys. 5.7, 5.8) zawiera pola z danymi osobowymi oraz ustawieniami konta, które użytkownik może modyfikować. Są tutaj opcjonalne pola takie jak zdjęcie profilowe, które będzie wyświetlane przy recenzjach. W tym ekranie istnieje także opcja usunięcia konta i wszystkich powiązanych z nim danych.



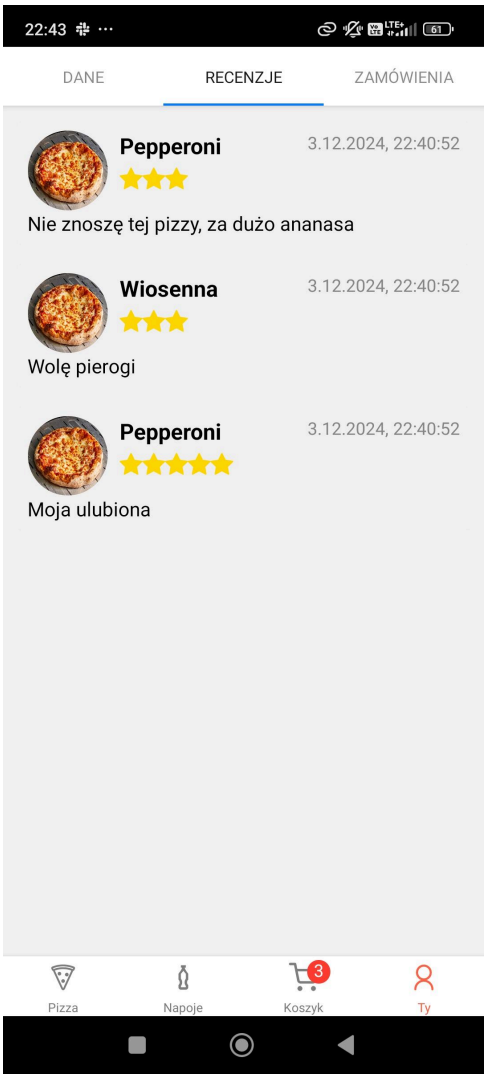
Rys. 5.7. Ekran ustawień cz. 1 [opracowanie własne]



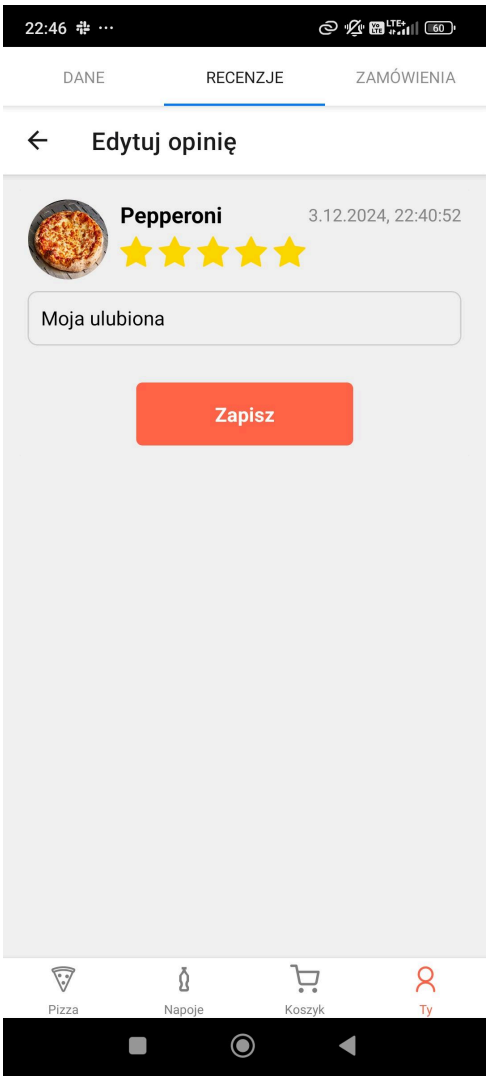
Rys. 5.8. Ekran ustawień cz. 2 [opracowanie własne]

5.7. Recenzje

Zakładka wyświetla listę wszystkich recenzji napisanych przez użytkownika. Istnieje możliwość edycji lub usunięcia recenzji. Interfejs jest prosty i intuicyjny, oferuje szybki podgląd treści opinii oraz oceny. Recenzja składa się z gwiazdek, które można zmienić za pomocą kliknięcia, a także pola tekstowego. Możliwe jest ustawianie połowicznych wartości gwiazdek takich jak “4,5” co daje większy wachlarz możliwości oceny, a także możliwość dodania jedynie jednej z tych opcji oceny, np. samego komentarza.



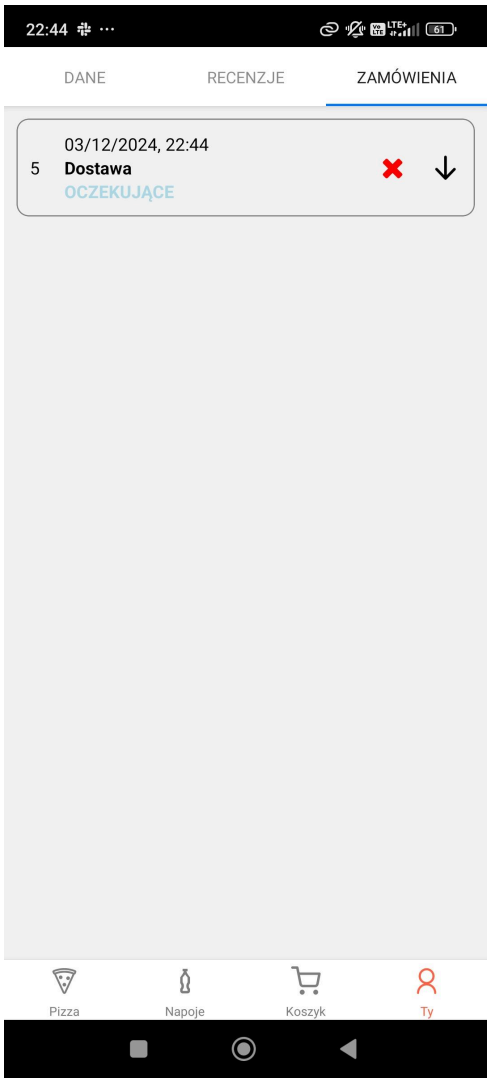
Rys. 5.9. Ekran recenzji [opracowanie własne]



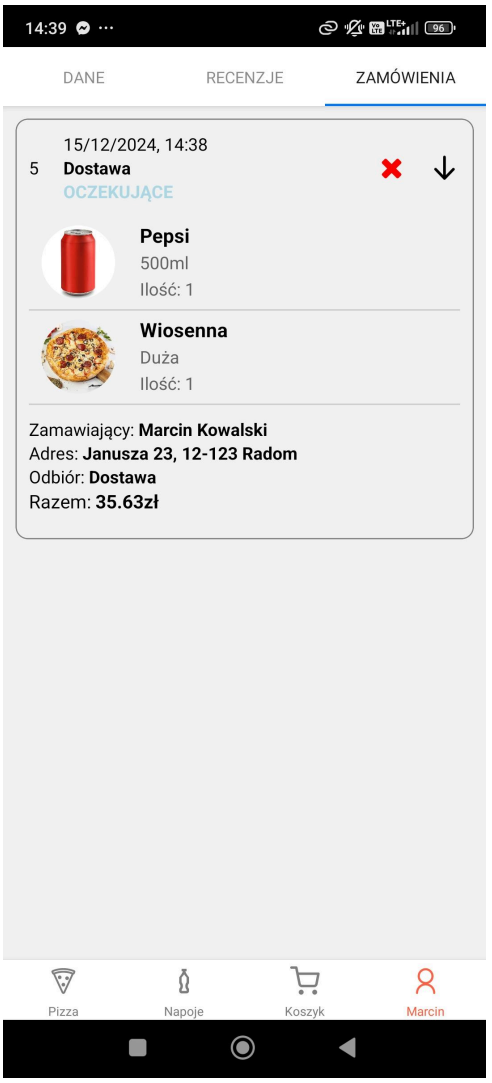
Rys. 5.10. Ekran edycji recenzji [opracowanie własne]

5.8. Zamówienia

Zakładka prezentuje historię wszystkich zamówień wraz z bieżącymi, które są w trakcie realizacji. Dla zamówień w stanie "Pending" istnieje możliwość anulowania ich. Każde zamówienie zawiera szczegóły takie jak unikalny identyfikator, lista pozycji, kwota końcowa oraz metoda dostawy wraz z opcjonalnymi informacjami takimi jak adres, jeżeli został podany. Dodatkowe informacje są wyświetlone poprzez menu rozwijane po interakcji użytkownika, po to by nie zabierać zbyt dużej przestrzeni, gdy historia zamówień jest duża.



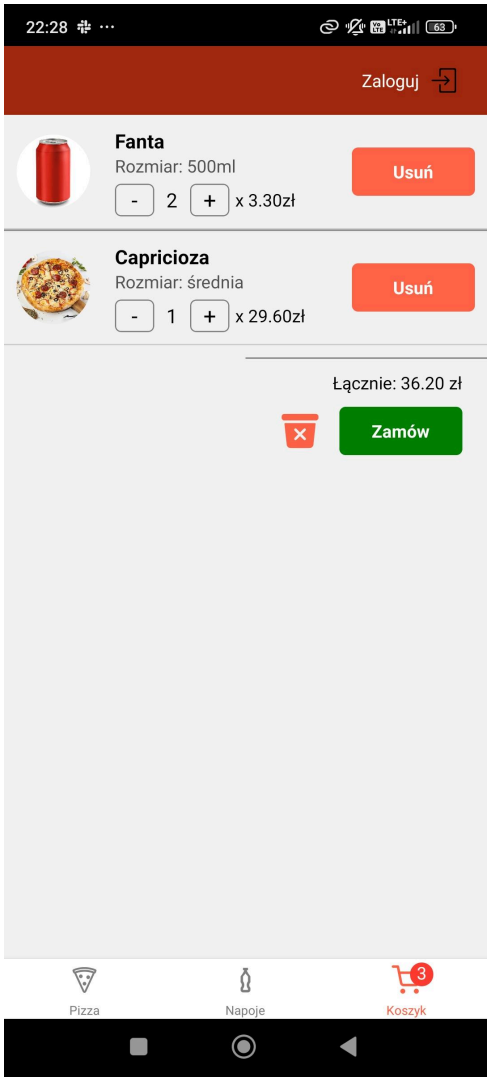
Rys. 5.11. Ekran zamówień [opracowanie własne]



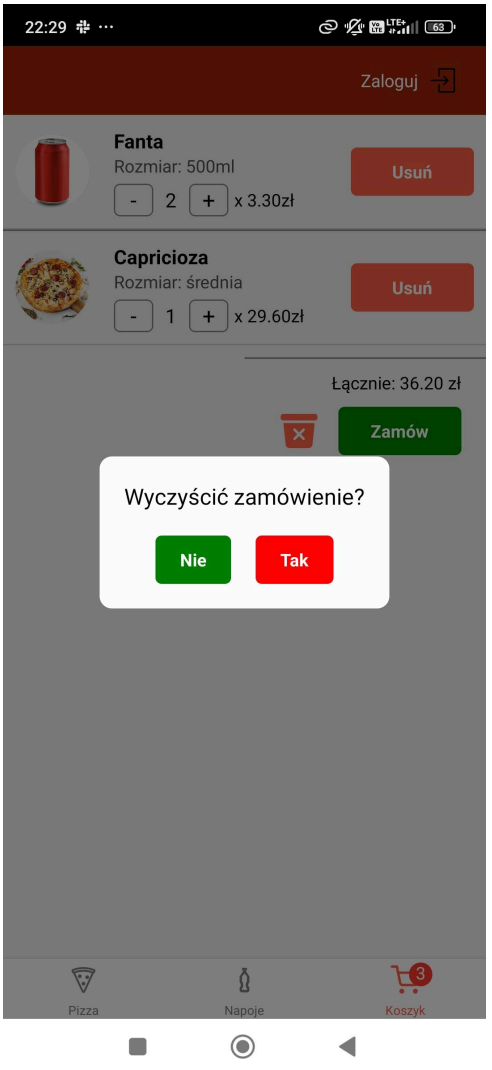
Rys. 5.12. Ekran z rozwiniętym zamówieniem [opracowanie własne]

5.9. Koszyk

Ekran koszyka (Rys. 5.13) pozwala użytkownikowi zarządzać wybranymi produktami przed finalizacją zamówienia. Wyświetlane są wszystkie dodane pozycje, ich liczba, rozmiar oraz suma końcowa. Użytkownik może usuwać produkty lub zmieniać ich liczbę za pomocą przycisków “+” lub “-”. Przy każdej pozycji istnieje także pole liczbowe dla konkretnej wartości liczbowej. Istnieje także możliwość wyczyszczenia całego koszyka (Rys. 5.14), wyświetla się wówczas okno dialogowe, którym trzeba potwierdzić operację. Przycisk “*Zamawiam*” przenosi użytkownika do okna z finalizacją zamówienia.



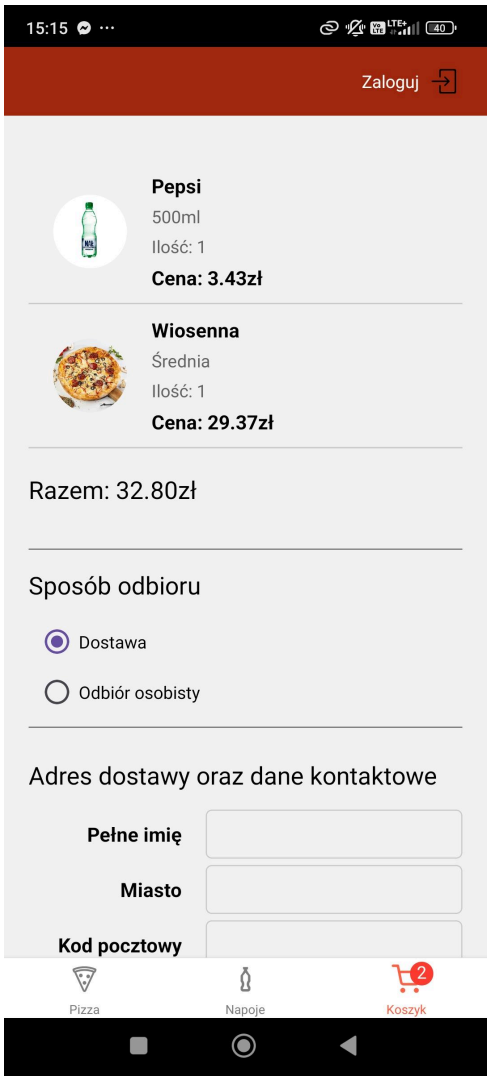
Rys. 5.13. Koszyk [opracowanie własne]



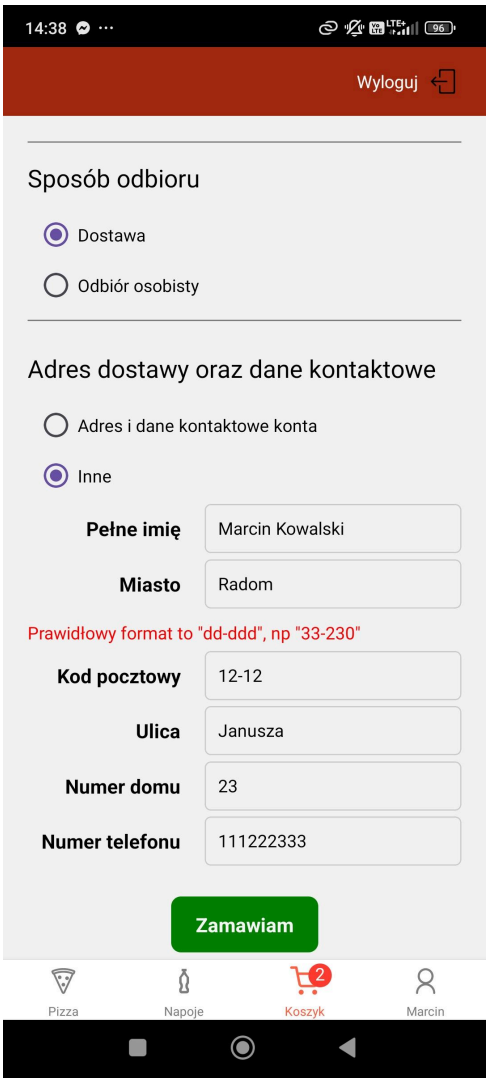
Rys. 5.14. Dialog opróżnienia koszyka [opracowanie własne]

5.10. Formularz zamówienia

Po zatwierdzeniu koszyka wyświetla się formularz (rys. 5.15, 5.16), który umożliwia wprowadzenie danych wymaganych do realizacji zamówienia, takich jak sposób dostawy, adres i numer telefonu. Zalogowani użytkownicy mają te dane automatycznie wypełnione na podstawie swojego profilu, co przyspiesza proces, chyba że zdecydują się wprowadzić inne dane, wówczas istnieje taka możliwość. Niezalogowani użytkownicy muszą wprowadzać te dane każdorazowo. Na końcu znajduje się przycisk finalizujący zamówienie.



Rys. 5.15. Ekran finalizacji zamówienia cz.1 [opracowanie własne]



Rys. 5.16. Ekran finalizacji zamówienia cz.2 [opracowanie własne]

6. Testy

W ramach procesu tworzenia aplikacji kluczowym etapem jest testowanie, które pozwala na wykrywanie błędów, weryfikację poprawności działania oraz zapewnienie wysokiej jakości oprogramowania. Testowanie aplikacji obejmuje zarówno funkcjonalność poszczególnych modułów, jak i ich współdziałanie. W tym rozdziale przedstawione zostaną metody i narzędzia używane do testowania aplikacji oraz zostaną omówione poszczególne przypadki.

6.1. Rodzaje testów w aplikacji serwerowej

W ramach testowania aplikacji serwerowej można wyróżnić:

- **Testy jednostkowe** - Testy jednostkowe to testy, które koncentrują się na sprawdzeniu pojedynczej jednostki kodu w izolacji. Jednostką może być np. pojedyncza funkcja, metoda czy klasa. Testy te powinny być szybkie, ponieważ ich celem jest zapewnienie, że poszczególne fragmenty kodu działają poprawnie. Testy jednostkowe nie powinny zależeć od innych jednostek systemu, takich jak bazy danych, serwisy zewnętrzne czy system plików. Cechy testów jednostkowych [6]:
    - **Izolacja**: Testy jednostkowe sprawdzają tylko jedną jednostkę kodu, niezależnie od innych części systemu. W tym celu często wykorzystuje się techniki takie jak “mockowanie” (tworzenie sztucznych obiektów, które imitują zachowanie prawdziwych obiektów).
    - **Niezależność**: Testy te powinny być niezależne od siebie, co oznacza, że wynik jednego testu nie powinien mieć wpływu na wynik kolejnego. Każdy test powinien być samowystarczalny.
    - **Szybkość**: Testy jednostkowe są zazwyczaj bardzo szybkie, ponieważ testują małe fragmenty kodu, a ich wykonanie nie wymaga integracji z zewnętrznymi systemami.
    - **Automatyzacja**: Testy jednostkowe są idealne do automatycznego uruchamiania, dzięki czemu mogą być używane w procesie ciągłej integracji (CI) i ciągłego dostarczania (CD).
- **Testy integracyjne** - to rodzaj testów, które mają na celu sprawdzenie współdziałania różnych modułów lub komponentów aplikacji, aby upewnić się, że funkcjonują one

prawidłowo jako całość. W odróżnieniu od testów jednostkowych, które koncentrują się na testowaniu pojedynczych funkcji w izolacji, testy integracyjne oceniają komunikację między różnymi częściami systemu, takimi jak bazy danych, API czy usługi zewnętrzne. Testy integracyjne skupiają się na punktach styku między modułami, np. komunikacji między warstwą aplikacyjną a bazą danych, czy wymianą danych za pomocą API. Często obejmują różne warstwy systemu, takie jak backend, frontend i zewnętrzne usługi (np. systemy płatności). Celem testów integracyjnych jest wykrycie problemów wynikających z błędnej integracji lub niezgodności między modułami oraz upewnienie się, że system jako całość spełnia wymagania funkcjonalne. Testy integracyjne często używają rzeczywistych baz danych lub komponentów symulujących środowisko produkcyjne. Wykorzystywane są frameworki takie jak Spring Test, Mockito, czy narzędzia do testowania API, np. Postman [8].

- **Testy funkcjonalne** są jednym z kluczowych rodzajów testów wykorzystywanych w procesie zapewnienia jakości oprogramowania. Ich celem jest sprawdzenie, czy system lub aplikacja działają zgodnie z wymaganiami funkcjonalnymi, tzn. czy implementują wszystkie funkcje określone przez specyfikację systemu. Testy funkcjonalne są wykonywane na wyższym poziomie abstrakcji niż testy jednostkowe, ponieważ koncentrują się na sprawdzeniu zachowania systemu jako całości. Cechy testów funkcjonalnych [8]:
    - **Skupienie na funkcjonalności systemu**: Testy te koncentrują się na testowaniu funkcji, które system ma wykonywać, a nie na tym, jak dokładnie te funkcje są zaimplementowane w kodzie. Oznacza to, że testerzy nie muszą znać szczegółów implementacyjnych, a jedynie oczekiwanego zachowania systemu.
    - **Testowanie wymagań**: Testy funkcjonalne sprawdzają, czy system spełnia określone wymagania użytkowników i interesariuszy. Obejmuje to funkcjonalności, takie jak logowanie, rejestracja, przetwarzanie płatności, generowanie raportów, itd.
    - **Brak skupienia na wewnętrznej implementacji**: Testy te są często wykonywane w taki sposób, że testerzy nie znają wewnętrznej struktury systemu ani kodu. Skupiają się na tym, jakie dane wprowadzają, a jakie wyniki oczekują.

- **Symulacja rzeczywistych scenariuszy użytkowania**: Testy funkcjonalne symulują scenariusze, w których użytkownicy będą korzystać z systemu, aby upewnić się, że wszystkie wymagane funkcjonalności są dostępne i działają poprawnie.

6.2. Narzędzia używane do testowania

W projekcie zostały wykorzystane narzędzia i biblioteki to testowania:

- **JUnit 5**
  JUnit to najpopularniejsza biblioteka do pisania testów jednostkowych w środowisku Java. Pozwala na definiowanie, organizowanie oraz uruchamianie testów w prosty i czytelny sposób. Przykłady w kodzie wykorzystują JUnit do testowania metod logiki biznesowej [4].
- **Mockito**
  Mockito umożliwia tworzenie atrap (mocków), które symulują zachowanie zależności w testach jednostkowych. Dzięki temu można izolować testowany komponent od zewnętrznych zależności, np. baz danych czy zewnętrznych usług [11].
- **Spring Test**
  Framework Spring Test wspiera testowanie aplikacji opartych na Springu. Umożliwia konfigurację kontekstów aplikacji oraz testowanie komponentów zintegrowanych z frameworkiem [11].

6.3. Przypadki testowe

W tym rozdziale przedstawione zostaną testy aplikacji serwerowej, które sprawdzają czy poszczególne części aplikacji zwracają oczekiwany rezultat w kontekście logiki biznesowej. Testy te wykonują się każdorazowo przed uruchomieniem aplikacji zarówno w środowisku produkcyjnym jak i deweloperskim, co dodaje warstwę bezpieczeństwa, która sprawdza czy wprowadzane zmiany w aplikacji są spójne ze stanem pożądanym.

6.3.1. Testy jednostkowe

Test (Rys. 6.1) weryfikuje poprawność działania metody kontrolera *ImageController*, odpowiedzialnej za obsługę pobrania obrazu przechowywanego w systemie. Test sprawdza, czy

kontroler prawidłowo zwraca obraz wraz z odpowiednim nagłówkiem informującym o typie zawartości. Test:

- Oczekuje, że odpowiedź ma status *HTTP 200 OK*.
- Weryfikuje, że nagłówek *Content-Type* w odpowiedzi zawiera *image/png*.
- Sprawdza, czy zawartość odpowiedzi to dokładnie dane obrazu (bajty).

```java
package com.example.Pizzeriabackend.controller;

import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@WebMvcTest(controllers = ImageController.class)
class ImageControllerTest {

   @MockBean
   private ImageService imageService;

   @Autowired
   private MockMvc mockMvc;

   @Test
   @WithMockUser(username = "username", roles = "USER")
   void should_ReturnImageWithStatus200AndCorrectContentType_WhenGettingImage() throws Exception {
       String folderName = "user";
       String imageName = "user_default.png";

       byte[] imageBytes = new byte[]{1, 2, 3};
       ByteArrayResource imageResource = new ByteArrayResource(imageBytes);

       when(imageService.getImage(folderName, imageName)).thenReturn(imageResource);
       when(imageService.resolveMediaType(imageName)).thenReturn(MediaType.IMAGE_PNG);

       mockMvc.perform(MockMvcRequestBuilders.get("/images/{folderName}/{imageName}", folderName, imageName))
               .andExpect(status().isOk())
               .andExpect(MockMvcResultMatchers.header().string(HttpHeaders.CONTENT_TYPE, MediaType.IMAGE_PNG_VALUE))
               .andExpect(MockMvcResultMatchers.content().bytes(imageBytes));
   }
}
```

Rys. 6.1. Test metody kontrolera odpowiedzialnej za zwracanie plików zdjęciowych [opracowanie własne]

Test metody *userService.loginUser* w serwisie użytkownika, który sprawdza czy w przypadku przekazania prawidłowych danych logowania zostaje zwrócona prawidłowa odpowiedź z danymi użytkownika wraz z tokenem upoważniającym do wykonywania zapytań oraz żądań

związanych z przypisanym do tokenu użytkownikiem. Test jest częścią klasy testowej *UserServiceTest* i łączy w sobie kilka warstw aplikacji w kolejności, w której są wywoływane:

- repozytorium użytkownika
- warstwę security (*AuthenticationManager*)
- serwis tokenów
- serwis użytkownika

```java
@Test
void should_ReturnAuthenticationResponse_WhenLoginSuccessful() {
    String userEmail = "test@example.com";
    String userPassword = "password123";
    String encodedPassword = "encodedPassword";
    String generatedAccessToken = "token";
    String generatedRefreshToken = "mockRefreshToken";
    Long userId = 1L;

    User mockUser = User.builder()
            .id(userId)
            .email(userEmail)
            .password(encodedPassword)
            .role(Role.USER)
            .build();

    AuthenticationRequest authenticationRequest = new AuthenticationRequest(userEmail, userPassword);

    when(userRepository.findByEmail(userEmail)).thenReturn(mockUser);

    RefreshToken mockRefreshToken = RefreshToken.builder()
            .id(1L)
            .token(generatedRefreshToken)
            .user(mockUser)
            .build();

    when(authenticationManager.authenticate(any(UsernamePasswordAuthenticationToken.class)))
            .thenReturn(new UsernamePasswordAuthenticationToken(mockUser, null, null));

    when(jwtService.generateRefreshToken(mockUser)).thenReturn(mockRefreshToken);
    when(jwtService.generateToken(mockUser)).thenReturn(generatedAccessToken);

    AuthenticationResponse authenticationResponse = userService.loginUser(authenticationRequest);

    assertNotNull(authenticationResponse);
    assertEquals(userEmail, authenticationResponse.getUserDetails().getEmail());
    assertNotNull(authenticationResponse.getToken());
    assertNotNull(authenticationResponse.getRefreshToken());
    assertEquals(mockRefreshToken.getToken(), authenticationResponse.getRefreshToken());
    assertEquals(generatedAccessToken, authenticationResponse.getToken());

    verify(authenticationManager, times(1)).authenticate(any(UsernamePasswordAuthenticationToken.class));
}
```

Rys. 6.2. Test jednostkowy sprawdzający proces uwierzytelniania [opracowanie własne]

Test (Rys. 6.3) weryfikujący, czy metoda *saveUserImage* w usłudze *UserService* prawidłowo zapisuje obraz użytkownika, gdy zostanie podany poprawny plik. Kluczowym aspektem jest potwierdzenie, że użytkownik oraz związane z nim dane są poprawnie aktualizowane i zapisane w bazie danych. Głównymi zadaniami testu są:

- Upewnienie się, że metoda *serviceUtils.getLoggedUser()* jest wywoływana w celu uzyskania informacji o aktualnym użytkowniku.
- Weryfikacja, czy metoda *imageService.saveImage()* jest wywoływana z odpowiednimi argumentami (plik obrazu, ścieżka zapisu, nazwa pliku).
- Sprawdzenie, czy obiekt użytkownika został zaktualizowany w repozytorium poprzez wywołanie metody *userRepository.save()*.

```
@Test
void should_SaveUserImage_WhenValidImageProvided() {
    // given
    String userEmail = "test@example.com";
    Long userId = 1L;
    User mockUser = User.builder()
            .id(userId)
            .email(userEmail)
            .build();

    MockMultipartFile mockMultipartFile = new MockMultipartFile(
            "image.png",
            "test.png",
            "image/png",
            "Spring Framework".getBytes()
    );

    when(serviceUtils.getLoggedUser()).thenReturn(mockUser);
    when(imageService.saveImage(any(MultipartFile.class), any(StaticAppInfo.IMAGE_FOLDER.class), any(String.class)))
            .thenReturn("http://localhost:8080/images/user/user_1.png");

    // when
    userService.saveUserImage(mockMultipartFile);

    // then
    verify(userRepository).save(mockUser);
}
```

Rys. 6.3. Test jednostkowy metody zapisującej obraz [opracowanie własne]

6.3.2. Testy integracyjne

Test (Rys. 6.5) sprawdza wszystkie warstwy aplikacji w procesie przetwarzania żądania o listę wszystkich dostępnych pizz. Istotną częścią jest konfiguracja (Rys. 6.4) środowiska testowego tak, by cały kontekst aplikacji był załadowany przed wykonaniem testu, dzieje się to za pomocą adnotacji *SpringBootTest*.

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class PizzaControllerIntegrationTest {

    @LocalServerPort
    private int port;

    @Autowired
    private PizzaRepository pizzaRepository;

    @Autowired
    private StaticAppInfo staticAppInfo;

    private final RestTemplate restTemplate = new RestTemplate();

    @AfterEach
    void cleanUp() {
        pizzaRepository.deleteAll();
    }
```

Rys. 6.4. Konfiguracja klasy testowej *PizzaControllerIntegrationTest* [opracowanie własne]

```java
@Test
void shouldReturnAllPizzas() {
    // Arrange
    Pizza pizza1 = new Pizza(null, "Margherita", 10.0, 12.0, 15.0, Collections.emptyList(), Collections.emptyList(), null);
    Pizza pizza2 = new Pizza(null, "Pepperoni", 11.0, 13.0, 16.0, Collections.emptyList(), Collections.emptyList(), null);
    pizzaRepository.save(pizza1);
    pizzaRepository.save(pizza2);

    String url = "http://" + this.staticAppInfo.getApplicationHost() + "/api/pizza";

    // Act
    var response = restTemplate.getForEntity(url, Map.class);

    // Assert
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
    List<Map<String, Object>> result = (List<Map<String, Object>>) response.getBody().get("result");
    assertThat(result).hasSize(2);
    assertThat(result.get(0).get("name")).isEqualTo("Margherita");
    assertThat(result.get(1).get("name")).isEqualTo("Pepperoni");
}
```

Rys. 6.5. Test integracyjny procesu zwracania listy wszystkich pizz [opracowanie własne]

Test (Rys. 6.6) sprawdzający poprawność procesu tworzenia nowej pizzy przez admina czy właściciela. Test weryfikuje:

- czy odpowiedź serwera posiada kod *201 (CREATED)*.
- czy odpowiedź nie jest pusta oraz czy pole *name* jest takie samo jak podczas tworzenia.
- czy baza danych posiada nowy dany rekord.

```java
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class PizzaControllerIntegrationTest {

    @LocalServerPort
    private int port;

    @Autowired
    private PizzaRepository pizzaRepository;

    @Autowired
    private StaticAppInfo staticAppInfo;

    private final RestTemplate restTemplate = new RestTemplate();

    @AfterEach
    void cleanUp() {
        pizzaRepository.deleteAll();
    }

    @Test
    void shouldCreatePizza() {
        // Arrange
        CreatePizzaRequest request = new CreatePizzaRequest();
        request.setName("Hawaiian");
        request.setSmallSizePrice(10.0);
        request.setMediumSizePrice(12.0);
        request.setBigSizePrice(14.0);

        String url = this.staticAppinfo.getApplicationHost() + "/api/pizza";

        // Act
        var response = restTemplate.postForEntity(url, request, Pizza.class);

        // Assert
        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.CREATED);
        assertThat(response.getBody()).isNotNull();
        assertThat(response.getBody().getName()).isEqualTo("Hawaiian");

        var pizzaFromDb = pizzaRepository.findAll();
        assertThat(pizzaFromDb).hasSize(1);
        assertThat(pizzaFromDb.get(0).getName()).isEqualTo("Hawaiian");
    }
  // other tests...
}
```

Rys. 6.6. Test integracyjny tworzenia nowego rekordu *Pizza* [opracowanie własne]

7. Podsumowanie i wnioski

Aplikacja opisywana w tej pracy jest odpowiedzią na komercyjne potrzeby współczesnego rynku, która dostarcza elastyczne i nowoczesne rozwiązania dla zarządzania zamówieniami w branży gastronomicznej. System został zaprojektowany w sposób prosty, przejrzysty, ale jednocześnie wystarczająco wszechstronny, aby obsłużyć kluczowe procesy operacyjne. Wykorzystane technologie, takie jak Spring Boot, PostgreSQL, oraz podejście "RESTful", gwarantują stabilność, bezpieczeństwo oraz wydajność systemu.

Implementacja uwzględnia modularną budowę, co umożliwia łatwe wprowadzanie nowych funkcjonalności oraz utrzymanie wysokiej jakości kodu. Zostały również spełnione najważniejsze wymagania związane z użytecznością i skalowalnością, co czyni aplikację gotową do wykorzystania w rzeczywistych warunkach biznesowych.

W trakcie tworzenia systemu, pojawiły się wnioski:

1. **Technologie i ich zalety**: Wybrane technologie są stabilne, dobrze udokumentowane i szeroko stosowane w branży, co pozwala na szybki rozwój kolejnych funkcjonalności i długoterminowe utrzymanie systemu. Spring Boot umożliwia szybkie prototypowanie, a PostgreSQL zapewnia niezawodną obsługę danych.
2. **Możliwości rozszerzenia**: Obecna wersja aplikacji pozostawia otwarte drzwi do rozwoju:
    - **Aplikacja dla personelu**: Możliwość stworzenia dedykowanego modułu lub aplikacji mobilnej umożliwiającej personelowi zarządzanie zamówieniami w czasie rzeczywistym.
    - **Panel administracyjny**: Rozszerzenie systemu o funkcje zarządzania użytkownikami, produktami i raportami dla administratorów.
    - **Integracja z innymi technologiami**: Nowe moduły mogą być implementowane w różnych technologiach, np. aplikacja przeglądarkowa w Angularze lub frameworkach bazujących na bibliotece React.
3. **Perspektywy wdrożenia**: Aplikacja może być wdrożona zarówno w małych, jak i średnich firmach, dostosowując się do ich specyficznych potrzeb operacyjnych. Elastyczna architektura pozwala na personalizację i dostosowanie systemu.

4. **Potencjał dalszego rozwoju**: Dzięki dobrze zaprojektowanej strukturze kodu oraz szerokiemu zakresowi testów, system może być rozbudowany o nowe funkcjonalności, takie jak integracja z systemami płatności online, śledzenie zamówień w czasie rzeczywistym czy generowanie zaawansowanych raportów i analiz bazujących na gromadzonych danych.

Bibliografia

[1] Akshat, P. React Native for Mobile Development: Harness the Power of React Native to Create Stunning iOS and Android Applications. 2019.

[2] Bloch, J. Effective Java. Addison-Wesley, 2018.

[3] Haverbeke, M. Eloquent JavaScript: Nowoczesny język programowania JavaScript. No Starch Press, 2018.

[4] Kaczanowski, T. Practical Unit Testing with JUnit and Mockito. 2019.

[5] Kuzmichev, S.; Grippa, V. M. Jak zaprojektować i wdrożyć wydajną bazę danych. Wydanie II. Helion, 2022.

[6] Langr, J. Pragmatic Unit Testing in Java 8 with JUnit. 2015.

[7] Meike, G. Blake; Schiefer, L. Inside the Android OS: Building, Customizing, Managing and Operating Android System Services. Addison-Wesley Professional, 2021.

[8] Mohan, G. Full Stack Testing. O'Reilly Media, 2022.

[9] Walls, C. Spring w akcji. Wydanie V. Helion, 2019.

[10] React.js – dokumentacja techniczna. [Online] https://pl.reactjs.org/ [dostęp: grudzień 2024].

[11] Spring – dokumentacja techniczna. [Online] https://docs.spring.io/ [dostęp: grudzień 2024].

[12] JSON Web Token. [Online] https://jwt.io/introduction [dostęp: grudzień 2024].

[13] Spring Data JPA. [Online] https://spring.io/projects/spring-data-jpa [dostęp: grudzień 2024].

Spis rysunków